Space Overhead Bounds for Dynamic Memory Management with Partial Compaction

Anna Bendersky
Computer Science Department
Technion
Haifa 32000
Israel
annaben@cs.technion.ac.il

Erez Petrank
Computer Science Department
Technion
Haifa 32000
Israel
erez@cs.technion.ac.il

Abstract

Dynamic memory allocation is ubiquitous in today's runtime environments. Allocation and de-allocation of objects during program execution may cause fragmentation and foil the program's ability to allocate objects. Robson has shown that a worst case scenario can create a space overhead within a factor of $\log n$ of the space that is actually required by the program, where n is the size of the largest possible object. Compaction can eliminate fragmentation, but is too costly to be run frequently. Many runtime systems employ partial compaction, in which only a small fraction of the allocated objects are moved. Partial compaction reduces some of the existing fragmentation at an acceptable cost. In this paper we study the effectiveness of partial compaction and provide the first rigorous lower and upper bounds on its effectiveness in reducing fragmentation at a low cost.

Categories and Subject Descriptors D.1.5 [*Object-oriented Programming*]: Memory Management; D.3.3 [*Language Constructs and Features*]: Dynamic storage management; D.3.4 [*Processors*]: Memory management (garbage collection); D.4.2 [*Storage Management*]: Garbage Collection

General Terms Languages, Performance, Algorithms, Theory.

Keywords Runtime systems, Memory management, Storage allocation, Dynamic storage allocation, Compaction, Partial compaction.

1. Introduction

The study of the theoretical foundations of memory management has not been very extensive. In particular, not much is known about the theoretical potential and limitations of memory management functionalities. Previous work that we are aware of includes a study of fragmentation [16], a study of cache-conscious memory placement [11], and a study of the space limitations of conservative garbage collection and lazy reference counting [4, 5]. In this work we attempt to extend these foundations and study the potential and limitations of partial compaction.

Modern software employs dynamic memory allocation to support its memory needs. Allocation and de-allocation of memory create fragmentation: holes between the allocated objects in the memory may be too small to further satisfy future allocation. Fragmentation creates a space overhead, since the memory consumed may become larger than the memory required to satisfy the allocation requests when no fragmentation exists.

Robson [15, 16] studied the amount of space overhead that may be caused by fragmentation when no objects are moved. He showed that in the worst case, fragmentation causes quite a large space overhead. In particular, he presented a program (or an allocation and de-allocation sequence) that never keeps more than M words allocated simultaneously, but any allocator that attempts to satisfy this sequence would require a space of (almost) $\frac{1}{2}M \log n$ words. The parameter n stands for the largest possible allocated size in the system. Robson also provided a simple allocation strategy that can handle any allocation sequence in (approximately) $M \log n$ words.

Frequent compaction eliminates fragmentation completely. If one compacts the heap after each de-allocation, then no fragmentation appears at all and M words of space always suffice. However, full compaction is costly since a substantial fraction of the objects may be moved and all references need to be updated [1, 9, 10]. Therefore, modern systems tend to either use compaction infrequently, or employ partial compaction, moving some objects in an attempt to reduce fragmentation and keep its cost acceptable [2, 3, 7, 12, 13]. Of course, the larger the space that is being compacted, the less fragmented the heap becomes; on the other hand, the overhead that is posed on the executing program increases. The question that arises is what is the trade-off between the amount of space the can be moved and the space overhead that may occur in the heap? In this work we provide the first lower and upper bounds for such a scenario.

Since the amount of moved space makes a difference, we need to bound the amount of relocated space. We choose to study a scenario in which a predetermined percentage of all allocated space can be moved. One could generalize the question to allow a quota of $B(S)$ movement for an arbitrary function B of the allocated space S. Other budgeting decisions are also possible; for example, one could limit the relocation according to the amount of live space, or the deleted space. All these make sense as well, but we felt that budgeting by a fraction of the allocated space is interesting, as the amount of allocation typically represents allocation "time" in the memory management literature, most notably for generational garbage collection.

In this work we present general results that bound the heap size required for allocation in the presence of partial compaction. Let the compaction budget at any point in the execution be $1/c$ of the space allocated so far by the program. To provide an upper bound, we present an allocation and compaction strategy. We show that for programs that never keep more than M words allocated simultaneously, it suffices to use a heap whose size is the minimum between $M \cdot (c+1)$ and $M \cdot \log n$, where n is the largest possible object that can be allocated. When the compaction budget is high, i.e., c is small, then the heap size can be significantly smaller than the heap size obtained when no compaction is used. The above is formally asserted in Theorem 1 in Section 3.

To show a lower bound on the space overhead, we present a program that incurs a large space overhead for any allocator (whose compaction budget is limited by $\frac{1}{c}$ of the allocated objects). This program never keeps more than M words allocated simultaneously, and it makes any allocator use a heap whose size is at least the minimum of $\frac{1}{10}M \cdot c$ and $\frac{1}{10}M \cdot \frac{\log n}{\log c+1}$ words. When the compaction budget is large (c is small), the minimum is obtained with $\frac{1}{10}M \cdot c$. When the compaction budget is tighter, the heap size is at least $\frac{1}{10}M \cdot \frac{\log n}{\log c+1}$. Thus, these asymptotic bounds show that partial compaction can reduce the heap size, but only to a limited extent, which depends on the compaction quota. This result is stated as Theorem 2 in Section 4.

The above results hold for any possible allocator. We continued this investigation with a study of a specific widely used memory allocator: the segregated free list allocator (used for example in [2, 6, 8]). Fixing a specific allocator allows a more accurate analysis and yields better bounds. We first examine this allocator when no compaction is allowed and improve Robson's bounds. It turns out that the bounds depend on the number of free-lists used by the allocation scheme. One extreme case is that the allocator maintains a free-list for any possible object size between 1 and n. In this case the required space becomes (almost) $\frac{4}{5}M\sqrt{n}$ words. This is much higher than the $\frac{1}{2}M \log n$ words presented in Robson's papers. The other extreme case is that a small number of free lists is allowed, specifically, one for each power of 2. In this case, we show a lower bound of $M \log n$ words, which is two times stronger than the lower bound for the general allocator. In practice, the number of free lists kept is somewhere in between, and our analysis can be applied to any specific choice of sizes to yield the corresponding lower bound. These results are formalized in Theorems 3 and 4 in Section 5.2.

Finally, we examine the effects of adding partial compaction to the segregated free list allocator. We show that a heap size of $M \cdot c + k \cdot n$ suffices in this case for any program, where k is the number of different free lists. So when the compaction budget is large, i.e., c is small, partial compaction helps reducing fragmentation. See Theorem 5 in Section 5.3. For the lower bound, the number of free lists employed is important. We first show that when using free lists for any object size, the space required is at least the minimum between $\frac{1}{6}M\sqrt{n}$ and $\frac{1}{12} \cdot M \cdot c$. So the true overhead is large if we don't allow a substantial compaction budget. If we only keep free lists for objects sizes that are powers of 2, then the heap size required is at least the minimum between $\frac{1}{4} \cdot M \cdot \log n$ and $\frac{1}{8} \cdot M \cdot c$. See Theorems 6 and 7 in Section 5.3.

This work initiates a study of the cost and effectiveness of partial compaction. New techniques are developed and novel bounds are shown. The results are asymptotic. They hold for any possible object size limit n, any possible live space bound M, and any possible compaction budget ratio c. However, although they represent the tightest known bounds, their applicability for specific realistic parameters of modern systems is limited. We hope that these techniques can be extended in future work to provide tighter bounds and help us better understand the behavior of memory managers in practice.

Organization. In Section 2 we survey previous results on fragmentation when no compaction is allowed, and formally define the compaction budget model used in this paper. In Section 3 we present and prove a simple upper bound on the effectiveness of partial compaction. In Section 4 we state and prove the lower bound on the effectiveness of partial compaction. We divide the proof into two parts. First, we prove the lower bound for the restricted case of aligned objects. This proof contains most of the ideas and is easier to follow. The full proof of the lower bound (with all the hairy details) follows. In Section 5 we study the specific segregated free list allocator and prove all the results stated above. We conclude in Section 6.

2. Problem Description

2.1 Framework and previous work

Dynamic storage allocation that does not move objects can suffer from fragmentation. J. M. Robson [15, 16] provided lower and upper bounds on the space overhead for a memory management system that allocates and deallocates, but cannot move objects. The lower bound demonstrates the existence of a "bad" program that makes *any* allocator use a lot of space overhead. The upper bound provides an allocator, whose space overhead is limited, for *any* program (including the bad program that demonstrates the lower bound). Clearly, if the program allocates a lot and does not de-allocate space, then a lot of space is required to satisfy the request. It is thus interesting to ask how much space is required to satisfy the allocation requests of a program that never keeps more than M words alive simultaneously. If compaction is frequently used, a space of M words suffices. The memory manager can simply compact the heap after each de-allocation. However, when no compaction is allowed, some allocation sequences demand a large space overhead in order to satisfy the allocation requests.

An allocator should be able to handle any allocation (and de-allocation) sequence, and for each allocator, we can ask what is the smallest heap that can handle all allocation sequences that never allow more than M words of memory simultaneously allocated. The bounds that are going to be derived depend on the size of the largest possible object in the system. We denote by n the size of such an object and assume that no allocation in the sequence requests more than n words for the allocated object.

Some allocators can move allocated objects in the memory. This is not always the case, as moving objects requires updating all pointers referencing them, and some runtime systems cannot always distinguish pointers from integers that hold the same values. Such systems do not move objects whose references cannot be identified. Other runtime systems avoid compaction because of its costs.

The interaction between the program and the memory allocator is divided into stages. Each stage consists of three parts:

1. *Deletion:* The program removes objects from the heap
2. *Compaction:* The memory allocator moves objects in the heap.
3. *Allocation:* The program makes allocation requests and the memory allocator returns the corresponding object addresses in the heap.

For the Upper and Lower bounds the program and the memory allocator can be viewed as opponents. The program objective is to make the memory allocator use as much space as possible. The memory allocator's objective is to allocate space in a way that will require as little space as possible. The upper and lower bounds for the case when is no compaction occurs were studied by Robson [16]. He showed a tight connection between the upper and lower bounds. We define $\mathcal{P}(M, n)$ as the set of all programs that never keep more than M words of space allocated simultaneously, and never allocate an object larger than n words. The function $HS(A, P)$ is defined as the heap size necessary for allocator A to answer the allocation requests of a program P, so that $P \in \mathcal{P}(M, n)$. The upper and lower bounds Robson showed demand an overhead of $\times \frac{1}{2} \log n$ of the actual allocated space M, i.e.:

- *Upper bound:* For all $M, n > 0$ such that $n|M$, there exists an allocator A_o, such that for all programs $P \in \mathcal{P}(M, n)$, it holds that:
$$HS(A_o, P) \leq M \cdot \log n + M.$$

- *Lower bound:* For all $M, n > 0$ such that $n|M$, there exists an program $P_o \in \mathcal{P}(M, n)$, such that for all allocators A, it holds that:

$$HS(A, P_o) \geq M \cdot \frac{1}{2} \log n + M - n + 1.$$

In this work we investigate the effect of moving objects during the run. We allow the allocator to move a fraction of the objects, i.e. to perform partial compaction. We explore how partial compaction affects these bounds.

2.2 The compaction budget

Our objective in this work is to explore upper and lower bounds on the space required by allocators that apply partial compaction. We now define partial compaction. If there were no limit on how much space can be moved, the total heap size required for the allocation would be M, the largest amount of live space throughout the application run. One simple method of compaction that uses only M words could be: allocate sequentially using a bump pointer and compact all of the memory once the M boundary is reached. The problem with this approach is that compacting the memory incurs a high computation overhead and is unacceptable in real systems. Practical systems consider trade-offs between the amount of compaction executed and the space overhead. Therefore, we introduce a bound on the amount of space that can be moved. This space is denoted by $B(S)$, which is the partial compaction function. Specifically, after the program allocates a space S, the compactor may move $B(S) = \frac{1}{c} \cdot S$ space. c is a constant larger than 1. The $B(S)$ function is calculated incrementally: the quota is added to on every allocation, and reduced with every compaction. For example: $B(S) = \frac{1}{10}S$ means that after every allocation of S, the compactor can move $\frac{1}{10}$ of the space allocated. In this example, if there were allocations of 200 words (and maybe some deletions) without any compactions, space of size of $\frac{1}{10} \cdot 200 = 20$ could be moved. If only 8 words are moved - a quota of 12 words remains. We define an algorithm that works within the $B(S)$ limit as the $B(S)$-*bounded partial compaction*. In this document we always use the compaction threshold function $B(S) = \frac{1}{c} \cdot S$, for some constant $c > 1$.

2.3 The setting

To present our results we first specify what a program is, what an allocator is, in what way they are limited, and how we measure their interaction. We consider a program P that executes a sequence of allocations and de-allocations. The program's execution is adaptive in the sense that it may choose its upcoming allocation according to the locations in which the allocator chose to place its previous allocation. However, to save further quantification over the inputs, we assume that P already has some fixed embedded input, so that the allocation sequence depends only on P and the allocation decisions of the allocator (but not on any additional inputs of P). We also consider an allocator A that receives the sequence of allocations and de-allocations one by one (sequentially, or on-line) from the program and must satisfy each allocation request as it arrives.

Given a specific pair of a program P and an allocator A, their joint execution is well defined and we measure the size of the heap that the allocator A uses to satisfy the requests of P. We denote this heap size by $HS(A, P)$. Of course, for the same program P, there could be a non-space-efficient allocator A_1 that requires a large heap size $HS(A_1, P)$ and there could be a better allocator A_2 for which $HS(A_2, P)$ is much smaller. For a lower bound, we look for a program P for which $HS(A, P)$ is high for all allocators. For an upper bound, we look for an allocator A that can serve all programs with a low $HS(A, P)$.

Finally, we define $\mathcal{P}(M, n)$ as the set of all programs that never keep more than M words of space allocated simultaneously, and never allocate an object larger than n words.

3. The Upper Bound

We start with the upper bound, which is simpler to show. Formally, we state and prove the following theorem.

Theorem 1. (Upper bound.) *For any real number $c \geq 1$, there exists a memory manager A_c that satisfies the compaction bound $\frac{1}{c}S$, and for all $M, n > 0$, such that $n|M$, and all programs $P \in \mathcal{P}(M, n)$:*

$$HS(A_c, P) \leq \min(M(c+1), M \cdot \log n + M).$$

Proof. For any real number $c \geq 1$, we present a specific memory manager A_c, that never exceeds the compaction quota $\frac{1}{c} \cdot S$ and that uses a heap size of at most $M \cdot (c+1)$ words, i.e., $HS(A_c, P) \leq M \cdot (c+1)$ for any program $P \in \mathcal{P}(M, n)$. The memory manager A_c allocates objects sequentially by bumping a pointer. When the bump pointer reaches the $M(c+1)$ boundary, the live space is fully compacted to the beginning of the heap. A pseudo-code for the memory allocator A_c is presented in Algorithm 1.

Algorithm 1 Memory Manager A_c

Initially: The heap is empty, $free = 0$;
While (TRUE)
1: **Receive an allocation request for an object of size ℓ;**
2: **if** $free + \ell \geq M(c+1)$ **then**
3: Compact all allocated objects to the beginning;
4: $free \leftarrow$ first word after the allocated space;
5: **end if**
6: Allocate the object at address $free$;
7: $free \leftarrow free + \ell$;

Clearly, Algorithm 1 never uses more than $M \cdot (c+1)$ words. However, we need to show that there is enough compaction budget to execute Step 3, which compacts all live objects to the beginning of the heap. The required compaction quota for this step is at most $M - \ell$. Consider the time interval between any two compactions, denoted C_1 and C_2. Since $P \in \mathcal{P}(M, n)$, there are at most M allocated words at any point in time, and therefore, after the compaction C_1 there are at most M words allocated in the beginning of the heap, and $free$ is smaller than M. After executing C_1 and before starting to execute C_2, the allocator receives requests for allocations and deletions, until it fails to allocate ℓ words – when the $free$ pointer arrives at a point beyond $M(c+1) - \ell$ in the heap. The total size of the allocations executed between C_1 and C_2 must be at least $M(c+1) - M - \ell$, since this is (at least) the number of words that the free pointer has advanced between the two compactions. Therefore, the compaction quota at Step 3 is at least $\frac{1}{c} \cdot (M(c+1) - M - \ell) = M - \frac{\ell}{c}$. According to our assumptions, $c \geq 1$ (and $\ell \geq 0$), and therefore, $M - \frac{\ell}{c} \geq M - \ell$. Thus, enough budget is available in the quota so that all of the $M - \ell$ allocated words can be moved to the beginning of the heap.

This memory manager is effective when $c + 1 \leq \log n$. Otherwise, c is large, i.e. the compaction budget is small, and then it is possible to obtain a good heap size without moving objects at all. We can use the allocator presented in Robson's paper, not use compaction at all, and consume at most $M \log n + M$ words for any possible program $P \in \mathcal{P}(M, n)$. This concludes the proof of Theorem 1. □

4. The Lower Bound

In this section we provide a specific program that can force any memory allocator to incur a space overhead. To simplify the pre-

sentation, we start with a program that assumes that the allocators respect some alignment restriction. We later extend this proof to eliminate the alignment restriction. Using a similar program, we provide a proof for Theorem 2.

4.1 Bounding the space consumption with aligned objects

We start by defining what aligned object allocation means. Aligned object allocation places a limitation on the memory allocator, which is not allowed to allocate objects in an arbitrary location, but only in an aligned manner. This constraint simplifies the proofs, and has less corner cases.

For the lower bound we only consider objects whose size is a power of two (because the program we build only allocates such sizes). Thus, it is fine to define alignment only with respect to objects of size 2^i for some integer i. Without loss of generality, we assume that the smallest object is of size 1. Otherwise, one could think of the object sizes as multiples of the smallest object size.

Definition 4.1. *We say that an object of size 2^i is* aligned *if it is located at address $k \cdot 2^i$ for some integer k.*

Definition 4.2. *We say that a memory allocator uses* aligned allocation *if all the objects it allocates are aligned.*

We present a specific program, called *Aligned-Waster* and denoted P_{AW}, and prove that any memory allocator that satisfies the compaction bound and uses aligned allocation, must incur a space overhead when supporting the allocation requests of P_{AW}. The program P_{AW} receives the compaction bound c, the largest object size n and the bound on the live space M as its input, and for any c, M, n it holds that $P_{AW}(c, M, n) \in \mathcal{P}(M, n)$. We will show that the program P_{AW} satisfies the following lemma.

Lemma 4.3. *For all $c \geq 1$, and all $M \geq n > 4$, there exists a program $P_{AW} \in \mathcal{P}(M, n)$ such that for all allocators A that use aligned allocation, and satisfy the compaction bound $\frac{1}{c}(S)$, the following holds:*

$$HS(A, P_{AW}) \geq \begin{cases} \frac{1}{6}M \cdot min\left(c, \frac{\log n}{\log c} - \frac{6n}{M}\right) & \text{if } c \leq 4\log n \\ \frac{1}{3}M \cdot \frac{\log n}{\log\log n} - n & \text{if } c > 4\log n \end{cases}$$

Note that this lemma is very much like Theorem 2, except that it is stated for aligned allocation and (therefore) the leading constants are better: a $1/6$ instead of a $1/10$.

We start with some intuition for constructing P_{AW} and then provide its algorithm. P_{AW} works in phases. In each phase, it generates a series of allocation requests and lets the allocator allocate the requests. Once it sees the locations of the allocated objects, it then decides which objects to delete.

The *Aligned-Waster* algorithm follows. The idea is that at each phase i, for $i = 0, 1, 2, \ldots, \log n$ *Aligned-Waster* requests allocation of objects of size 2^i. It allocates as many of them as possible while keeping the allocated space below M. It then examines where the allocator places the objects and decides on object deletions. The deletion is intended to prevent space reuse in the next phase, Phase $i+1$. In the aligned setting, an allocated object of size 2^{i+1} must be placed on a full aligned interval of size 2^{i+1}. *Aligned-Waster* examines each such interval, and if it is not empty, *Aligned-Waster* makes sure that some object is kept there to avoid re-allocation of a new object on this interval in the next phase.

However, not leaving an aligned interval of length 2^{i+1} empty is not enough to prevent its reuse in the presence of partial compaction. The allocator may choose to move the objects on such an interval and clear it for re-allocation. Therefore, *Aligned-Waster* applies a stricter policy when deciding which objects to delete. In particular, it attempts to keep enough allocated space in each interval so that it will not be worthwhile for the allocator to waste its compaction budget and move all these objects out of the interval. Given the compaction bound ratio $\frac{1}{c}$, the algorithm P_{AW} sets a density $\frac{1}{d} > \frac{1}{c}$, and when deleting objects, P_{AW} attempts to keep a density of $\frac{1}{d}$ in each of the aligned 2^{i+1}-sized interval. The algorithm follows.

Algorithm 2 Aligned-Waster Program

Input: M, n, and c.
Compute d as a function of c (to be determined).
for $i = 0$ to $\log n$ **do**
  { Deletion step: }
    Divide the memory into consecutive areas of size 2^i
    Remove as many objects as possible from each area,
      subject to leaving at least $2^i/d$ space occupied.
  { Allocation step: }
    Request allocation of as many objects as possible of
      size 2^i (not exceeding the overall allocated size M).
end for

By definition $P_{AW} \in \mathcal{P}(M, n)$ since it never allocates an object larger than n and its overall allocation space never exceeds M words.

We note that (intuitively) it does not make sense to set $d \geq c$. If the allocated space that remains on an interval is of size $2^i/c$ (or less), then it makes sense for the allocator to move these allocated objects away and make space for reuse. This costs the compaction budget $2^i/c$, but this budget is completely refilled when the 2^i-sized object is later allocated on the cleared interval. So if we want to make the allocator pay for a reuse, we must set $d < c$ or in other words, let P_{AW}'s deletion step leave more allocated space on each interval. We leave the setting of the density $1/d$ as a parameter, but one possible good setting sets d to be $c/2$. In this case, the deletion step of P_{AW} leaves enough allocated space on each interval so that not much reuse is possible within the given allocation budget. If reuse cannot be applied, then the allocator has to get more fresh space and increase the heap in order to satisfy the allocation requests of P_{AW}.

The rest of this section provides a rigorous analysis of P_{AW} showing that it outputs allocation and de-allocation requests that make any allocation strategy incur a space overhead as asserted in Lemma 4.3. Section 4.2 extends this algorithm and analysis for the non-aligned case, extending the analysis to obtain the proof of Theorem 2.

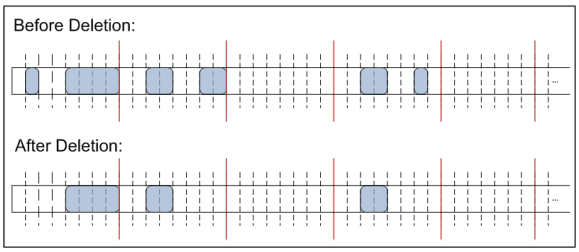

Figure 1. An example of deleting objects in *Aligned-Waster*, with parameter $1/d = 1/4$, and phase $i = 3$.

To bound the space required to satisfy the allocation requests, we bound, on one hand, the amount of space allocated by P_{AW} in the execution, and on the other hand, the amount of space that the allocator manages to reuse during the execution. Let $S(A, P_{AW})$ denote the total space allocated during an execution of the program P_{AW} with an allocator A. Also, let $R(A, P_{AW})$ denote the total reused space. A space is considered reused if some object is allocated on it, the object is then deleted or relocated by the compaction, and later the same space is used for allocation of another

object. The same space can be counted as reused more than once if more than two objects are placed on it during the execution. Clearly, during the execution of P_{AW} and A, the size of the heap required to allocate all objects is at least the total size of the allocated objects, minus the size of reuse that the allocator manages to make. Formally, we can state this claim as follows.

Claim 4.4. *For any program P and any allocator A, the space $HS(A,P)$ required by A to satisfy the allocation requests of P during their joint execution satisfies:*

$$HS(A,P) \geq S(A,P) - R(A,P).$$

We stress that Claim 4.4 holds not only for the specific program P_{AW}, or in the aligned setting. It holds for any program and any allocator.

To prove Lemma 4.3 we use Claim 4.4 and show that a lot of space is allocated and not much space is reused. In other words, we bound $S(A,P_{AW})$ from below, and $R(A,P_{AW})$ from above, for any possible allocator A.

We start with an upper bound on the space reuse $R(A,P_{AW})$. The intuition is that reuse can typically be done only after relocating the objects out of this area. For *Aligned-Waster*, there are not too many objects that can be moved out, as discussed earlier. Nonetheless, the actual proof is more complicated than this simple intuition, because as phases advance, different areas converge into a single one. An area of Phase i consists of eight areas from Phase $i-3$. Some of these eight areas may be empty, while others may be dense, still creating a sparse area of size 2^i.

Let $Q(A,P_{AW})$ denote the total size of compacted space (i.e., relocated objects) throughout the run of the program P_{AW} against a memory manager A that allocates in an aligned manner and satisfies the compaction threshold $\frac{1}{c}S$. Let d be the deletion threshold of *Aligned-Waster*. We show that the reuse $R(A,P_{AW})$ is bounded according to the following claim.

Claim 4.5. *For any allocator A that allocates objects in an aligned manner and satisfies the compaction bound $\frac{1}{c}S$, and a deletion factor $d \geq 1$ employed by P_{AW}, the following bound holds.*

$$R(A,P_{AW}) \leq Q(A,P_{AW}) \cdot d.$$

In the proof of this claim we use several definitions of areas, reuses and compaction quota that hold in the run of Program *Aligned-Waster*:

Definition 4.6. *If o is an object that is placed on the heap during the run of Program* Aligned-Waster*, then $|o|$ is it's size. r_o is the space size reused by the allocation of the object o. r_o is defined as the total size of objects parts that existed in the space that is later occupied by object o. These objects were removed or compacted, but there was no object placed on top of them between their removal/compaction and until the allocation of Object o. Namely, the relocation was used for the reuse of o, and not for any previous reuse. q_o is the total size of objects that were located in the space occupied later by object o. These objects were compacted, but there was no object placed on top of them until the allocation of o.*

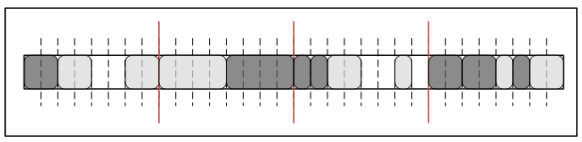

Figure 2. An example the definitions o, r_o and q_o in the execution of *Aligned-Waster*, with parameter $1/d = 1/4$, and phase $i = 5$.

An example of these definitions is in Figure 2. In this figure, a fraction of the heap of size 32 words is shown. This fraction of the heap is an empty aligned area, upon which an object o is placed in Phase 5 in the run of program *Aligned-Waster*. The dark gray squares represent words in the heap, that the last operation on this word was a compaction of an object. The light gray squares represent words in the heap, that the last operation on this word was a deletion of an object. The white squares are words on the heap that were empty since the beginning of the run of the program. In this example, q_o equals to the total size of the objects that were last compacted - i.e. to the total size of the dark words, 13 words. r_o equals to the total size of the deleted + compacted objects in this area, i.e. 27 words.

Proof of Claim 4.5. Fix an allocator A and consider a specific execution of P_{AW} with A. We break the amount of compaction on $Q(A,P_{AW})$ and the amount of reuse $R(A,P_{AW})$ into small increments. In particular, consider the space reuse in the execution. A reuse occurs when a new object is placed upon an empty aligned area that previously contained allocated objects. Consider each such allocation of an object o_j on an aligned area a of size 2^i. Just before the placement of o_j, we check the size of deletions and compactions that occurred in order to make space for the allocation. We show in Claim 4.7 below that $r_{o_j} \leq q_{o_j} \cdot d$ for any allocated object o_j. Now, summing over all allocations during in the execution of *Aligned-Waster*, we get:

$$R(A,P_{AW}) = \sum_j r_{o_j} \leq \sum_j q_{o_j} \cdot d \leq Q(A,P_{AW}) \cdot d \quad (4.1)$$

The second inequality follows from Claim 4.7. The last inequality follows since any relocation that is later used for space reuse is part of the entire set of relocations executed by A during the execution. It remains to show that Claim 4.7 holds. □

Claim 4.7. *Consider any phase i during the execution of* Aligned-Waster*. Observe any object o just before it is being allocated in the heap during phase i. It holds:*

$$r_o \leq q_o \cdot d$$

Proof. We look at the area a where the object o is placed, just before the allocation of o in phase i. The space of a is empty at that time. This space can be sub-divided into aligned sub-areas of size 2^k $s.t.$ $0 < k < i$, using the Algorithm 3.

Algorithm 3 Division of an area a into sub-areas

Input: An area a of size 2^i.
List $L \leftarrow$ a new empty list
Stack $S \leftarrow$ a new empty stack
$S.Push(a,i)$
while S is not empty **do**
  $(x,k) \leftarrow S.Pop()$
  if $k = 1$ **then**
    $L.Push(x,k)$
  else
    Look at the removal step in phase k, during the run of *Aligned-Waster*. Check if there were any removals in area x during that step.
    if there was at least one removal **then**
      $L.Push(x,k)$
    else
      Divide x into its two equal sub areas x_1, x_2.
      $S.Push(x_1, k-1)$
      $S.Push(x_2, k-1)$
    end if
  end if
end while
return L.

The result of Algorithm 3 is a list L that contains sub-areas of the area a. The sub-areas are distinct, their size is 2^k for $k \in \mathbb{N}, 0 \leq k \leq i$. The union of the sub-areas is exactly the area a. For every sub area that is the output of algorithm 3, one of the following holds:

1. The sub was either was always empty, or was occupied by an object that was compacted away. The sub-area size is 1.
2. The sub area size is 2^k $s.t.$ $k \geq 1$, and last time objects were deleted from this sub-area happened during the deletion step of phase k in Algorithm *Aligned-Waster*.

The statement above follows directly from the behavior of Algorithm 3. An area of size 2^k is sub-divided only if it had no deletions in phase k. Therefore, when an area reaches size 1, it never had any deletions, and statement (1.) is true. Since Algorithm 3 divides each aligned area of size 2^k to its two equal halves, the resulting sub-areas are the exact same areas that algorithm *Aligned-Waster* considered in phase $k-1$. The algorithm actually goes back from phase i back to phase 1 looking for the last deletion for each sub-area. For all sub-areas $sa(k)$ resulting from algorithm 3, we examine cases (1.) and (2.).

In case (1.), it holds that there were no removals at all from the sub-area $sa(k)$. The reuse, if larger than 0, occurs since an object was allocated and later compacted (possibly more than once). In this case, the reuse in phase i of the sub-area $sa(k)$ equals to the size of the object that was last compacted away from this sub-area: $r_{sa(k)} = q_{sa(k)}$. Since $d \geq 1$, it holds that:

$$r_{sa(k)} \leq q_{sa(k)} \cdot d.$$

In case (2.), for each sub-area $sa(k)$, phase k captures the time of the last deletion from the sub area $sa(k)$. If the sub-area $sa(k)$ was not reused between phases k and i (the first allocation on sub-area $sa(k)$ after phase k, occurred in phase i), then we compute the reuse based on the following. According to the definition of the adversarial program *Aligned-Waster* behavior, for each such area, after each removal, at least $1/d$ of the area remains occupied, we denote these objects by q. These objects must be later compacted away from this sub-area, so that the space could be reused. Therefore, if we multiply the size of this compaction by d, we get the size of the sub-area $sa(k)$. The reuse size in this sub-area is smaller or equal to the sub-area size, therefore, in this case

$$r_{sa(k)} \leq q_{sa(k)} \cdot d.$$

Another option, is that the area was already reused (fully or partially) - an object was already placed on the sub-area $sa(k)$, and was later compacted away. In this case, it holds that the size that was compacted but never reused is at least $1/d$ of the sub-area $sa(k)$ size. This statement is true since after the last deletion, at least $1/d$ of the area remained occupied. If we notice only these locations in the heap, we can see that objects were compacted, and possibly partially reused by other objects between phases k and i, but these "reusing" objects were later compacted, as the sub-area is eventually empty. Therefore, in this case also there is always $1/d$ of the area that was compacted away, and not yet reused. So we get that:

$$r_{sa(k)} \leq q_{sa(k)} \cdot d.$$

This statement is true for all sub-areas. The sum of $r_{sa(k)}$, where k denotes the sub-area k of a, holds $r_a = \Sigma_k r_{sa(k)}$, and the sum of $q_{sa(k)}$, where k denotes the sub-area k of a, holds $q_a = \Sigma_k q_{sa(k)}$. Since the area a is exactly the location of the newly placed object o, it holds that

$$r_o \leq q_o \cdot d.$$

This concludes the proof of Claim 4.7 □

Claim 4.8. *Let A be any allocator that satisfies the compaction bound $\frac{1}{c}S$, such that $c \geq 1$. Let the deletion threshold of P_{AW} be $d \geq 1$ and let the total space allocated during the execution of P_{AW} with A be $S(A, P_{AW})$. It holds that:*

$$HS(A, P_{AW}) \geq S(A, P_{AW}) \left(1 - \frac{d}{c}\right).$$

Proof. By Claim 4.4, it holds that

$$HS(A, P_{AW}) \geq S(A, P_{AW}) - R(A, P_{AW}). \tag{4.2}$$

By Claim 4.5, it holds that

$$R(A, P_{AW}) \leq Q(A, P_{AW}) \cdot d, \tag{4.3}$$

where Q is the total compacted space. Furthermore, by the bound on the partial compaction we know that only $1/c$ of the allocated space can be compacted, i.e.,

$$Q(A, P_{AW}) \leq \frac{1}{c} \cdot S(A, P_{AW}). \tag{4.4}$$

Using Equations 4.2, 4.3, and 4.4, we get:

$$HS(A, P_{AW}) \geq S(A, P_{AW}) - S(A, P_{AW}) \cdot \frac{d}{c}$$

as required, and we are done with the proof of Claim 4.8. □

We now return to Lemma 4.3, which asserts the lower bound (for the aligned case). We break the argument into its two different cases, according to the compaction bound range. We first state the two key claims, then derive the lower bound from them as a conclusion, and finally, provide the proofs of these two claims. The first claim considers the case of a small compaction allowance. In particular, the compaction ratio c and the density parameter d are larger than the logarithm of the largest object size n.

Claim 4.9. *For all allocators A such that A satisfies the compaction bound $\frac{1}{c}S$, s.t. $c > d \geq \frac{1}{2}\log n$, and $M \geq n > 4$, it holds that:*

$$HS(A, P_{AW}) \geq \frac{1}{3} M \frac{\log n}{\log \log n} - 2n \left(1 - \frac{d}{c}\right).$$

The second key claim considers the general case in which more compaction can be executed and only requires that $c > d > 1$.

Claim 4.10. *For all allocators A such that A satisfies the compaction bound $\frac{1}{c}S$, for all $M \geq n > 2$, and for all $c > d > 1$, it holds that:*

$$HS(A, P_{AW}) \geq min\left(\frac{1}{3}Md, \left(\frac{1}{3}M\frac{\log n}{\log d} - 2n\right) \cdot \left(1 - \frac{d}{c}\right)\right).$$

Having stated these two key claims, we now show that they imply Lemma 4.3, which is the main focus of this section. To this end we choose the density parameter d for P_{AW} to be $d = \frac{c}{2}$. Claim 4.9 holds for $d \geq \frac{1}{2}\log n$, and therefore, for $c \geq 4 \log n$. In this case we get

$$HS(A, P_{AW}) \geq \frac{1}{3} \cdot M \cdot \frac{\log n}{\log \log n} - n.$$

For the case that $c \leq 4 \log n$ we can use Claim 4.10 and deduce that

$$HS(A, P_{AW}) \geq \frac{1}{6} \cdot M \cdot min\left(c, \frac{\log n}{\log c} - \frac{6n}{M}\right).$$

These two equations yield the lower bound of Lemma 4.3 exactly, as required.

To finish the proof of the lower bound, we need to prove Claims 4.9 and 4.10. We start with some properties of areas as induced by the activity of the program P_{AW}.

Claim 4.11. *Consider any execution of P_{AW} with any allocator A, and any Phase i, $0 \le i \le \log n$. Let d be the deletion threshold. Just after the deletion step in Phase i, any non-empty aligned area of size 2^i has one of the following two possible configurations:*

1. *The allocated space in the area is smaller than $\frac{2^i}{d} \cdot 2$ (and each object size is smaller than $\frac{2^i}{d}$).*
2. *The area contains exactly one object that is larger or equal to $\frac{2^i}{d}$.*

Proof. Recall that the deletion in Phase i attempts to delete as much space as possible, while still leaving a space of at least $\frac{2^i}{d}$ words allocated. Suppose that after the deletion we have an object whose size is at least $\frac{2^i}{d}$. Then before the deletion this object existed in the area and if the deletion left this object allocated, then it must delete all other objects in the area and this object must be the only object alone in this area, satisfying the second case.

Otherwise, after the deletion all objects are of size smaller than $\frac{2^i}{d}$. Let the smallest of the remaining objects be of size 2^j for some $j \le i$. Removing this object was not possible in the deletion step, therefore, the occupied space in this area can be at most $\frac{2^i}{d} + 2^j - 1$, which is smaller than $\frac{2^i}{d} \cdot 2$. □

The above claim shows that small objects must be sparsely allocated after a deletion step. In particular, objects smaller than $\frac{2^i}{d}$ must be allocated on an area with density smaller than $\frac{2}{d}$. We generalize this sparseness of small objects in the next claim. Let i be the phase number, and let $k \le d$ be a size threshold, i.e., we consider objects of size $\frac{2^i}{k}$ as small objects. Denote by $x_i(k)$ the total space consumed by small objects after Phase i. The following claim asserts that the heap size must be large enough to accommodate these small objects.

Claim 4.12. *Consider an execution of P_{AW} against any allocator A. Let $k \le d$ be a size threshold, and $x_i(k)$ be the total space consumed by objects that are smaller than $2^i/k$ after the deletion step in Phase i. After the deletion step of any Phase i in the execution, it holds that:*

$$HS(A, P_{AW}) \ge \begin{cases} x_i(k) \cdot k & \text{if } k \le \frac{1}{2}d \\ x_i(k) \cdot \frac{1}{2} \cdot k & \text{if } \frac{1}{2}d \le k \le d \end{cases}$$

Proof. Consider the state of the heap after the deletion step in any Phase i. In this phase the areas considered for allocation and deletion are of size 2^i. All objects that are smaller than $\frac{2^i}{k}$ belong to $x_i(k)$. Consider any area that holds an object in $x_i(k)$. According to Claim 4.11, either there is only one object in the area and its size lies between $\frac{2^i}{d}$ and $\frac{2^i}{k}$, or the size of all remaining objects in the area is at most $\frac{2^i}{d} \cdot \frac{1}{2}$. If $k \le \frac{1}{2}d$, then the size of the area is larger by a factor of at least k than the occupied space on this area with objects from $x_i(k)$. If $\frac{1}{2}d \le k \le d$, then the size of the area is larger by a factor of at least $\frac{1}{2}k$ than the occupied space on this area with objects from $x_i(k)$. The size of the heap must be at least the sum of all areas containing objects in $x_i(k)$. Therefore, we get $HS(A, P_{AW}) \ge x_i(k) \cdot k$ or $HS(A, P_{AW}) \ge x_i(k) \cdot \frac{1}{2}k$ as required. Note that the heap size is monotone, because the heap size required for Phases $1, \ldots, i, i+1$ is at least the space required for Phases $1, \ldots, i$. Therefore, the obtained bound holds for the heap size of the entire execution. □

We now prove the key claims. Intuitively, it can be argued that either there is a point in the run of the program with many small objects that are so sparsely allocated that the lower bound follows easily, or there is no such point. In the latter case, there must be many memory allocations during the execution of P_{AW} with A, i.e., $S(A, P_{AW})$ is large, and therefore, Claim 4.8 implies the correctness of the lower bound.

Proof of Claim 4.9. We need show that if $\frac{1}{2}d > \log n$, then

$$HS(A, P_{AW}) \ge \frac{1}{3} \cdot M \cdot \frac{\log n}{\log \log n} - 2n\left(1 - \frac{d}{c}\right).$$

We use Claim 4.12, and set the size bound k to be $\log n$, which is smaller than $\frac{1}{2}d$. This means that in Phase i we consider an object to be small if it is smaller than $\frac{2^i}{\log n}$. We divide the analysis into two cases. In one case there is a phase in which a lot of small objects are allocated, and in the other case all phases do not have a large number of small objects allocated. So let α be some fraction, $0 < \alpha < 1$ (to be set later) and consider the case in which the execution has a Phase i for which small objects (smaller than $\frac{2^i}{\log n}$) occupy more than $(1-\alpha)M$ words. Then by Claim 4.12, the heap size at this point (and on) is larger than $(1-\alpha)M \cdot \log n$. Otherwise, in each of the Phases $i = 0, \ldots, \log n$, after every allocation phase i, the total space occupied in the heap is at least $M - 2^i$. It holds that more than $\alpha \cdot M - 2^i$ of the heap consists of objects that are larger than $\frac{2^i}{\log n}$. The size of the object tells us when it was allocated. Objects smaller than $\frac{2^i}{\log n}$ words were allocated in phases that are at least $\log \log n$ earlier than the current phase, whereas large objects were created in the last $\log \log n$ phases. This means that at least $\alpha \cdot M - 2^i$ of the live space must consist of objects that were created in the last $\log \log n$ phases. An execution consists of $\log n$ phases. We divide the execution into discrete sections of $\log \log n$ phases. In the last phase of each such section, at least $\alpha \cdot M - 2^i$ of the allocated space must have been allocated during this section. This is true for all sections. Therefore, if we compute the amount of space allocated in all $\frac{\log n}{\log \log n}$ sections, we get that that the amount of space allocated in the entire execution, $S(A, P_{AW})$, satisfies

$$S(A, P_{AW}) \ge \alpha M \frac{\log n}{\log \log n} - \sum_{i=\log \log n}^{\log n} 2^i \ge \alpha M \frac{\log n}{\log \log n} - 2n.$$

According to the relation between the amount of allocation and the heap size, shown in Claim 4.8, we get that

$$\begin{aligned} HS(A, P_{AW}) &\ge S(A, P_{AW}) \cdot \left(1 - \tfrac{d}{c}\right) \\ &\ge \left(\alpha M \cdot \tfrac{\log n}{\log \log n} - 2n\right) \cdot \left(1 - \tfrac{d}{c}\right). \end{aligned}$$

Setting $\alpha = \frac{1}{3\left(1 - \frac{d}{c}\right)}$, and using the fact that $d \le \frac{c}{2}$, we get that

$$HS(A, P_{AW}) \ge \frac{1}{3} M \cdot \log n$$

in the first case and that

$$HS(A, P_{AW}) \ge \frac{1}{3} M \cdot \frac{\log n}{\log \log n} - 2n\left(1 - \frac{d}{c}\right)$$

in the second case and we are done with the proof of Claim 4.9. □

Proof of Claim 4.10. We need to show that

$$HS(A, P_{AW}) \ge min\left(\frac{1}{3}Md, \left(\frac{1}{3}M\frac{\log n}{\log d} - 2n\right) \cdot \left(1 - \frac{d}{c}\right)\right).$$

Similarly to the proof of Claim 4.9, we use Claim 4.12. This time we set $k = d$, implying that in Phase i objects are considered small if they are smaller than $\frac{2^i}{d}$. If there is a Phase i where more than $\frac{2}{3}M$ of the live space consists of objects smaller than $\frac{2^i}{d}$, then the

total heap size is larger than $\frac{2}{3}M \cdot \frac{1}{2}d = \frac{1}{3}Md$. Otherwise, in every phase i in the execution, it holds that more than $\frac{1}{3}M - 2^i$ of the live space consists of objects that are larger than $\frac{2^i}{d}$. Partitioning the execution into disjoint consecutive sections of $\log d$ phases, we get that in each such section, at least $\frac{1}{3}M - 2^i$ space was allocated. Therefore,

$$S(A, P_{AW}) \geq \frac{1}{3}M \cdot \frac{\log n}{\log d} - \sum_{i=\log\log d}^{\log d} 2^i \geq \frac{1}{3}M \cdot \frac{\log n}{\log d} - 2n.$$

And again, according to Claim 4.8, we get that

$$HS(A, P_{AW}) \geq \left(\frac{1}{3}M \cdot \frac{\log n}{\log d} - 2n\right) \cdot \left(1 - \frac{d}{c}\right).$$

One of these cases must hold, and therefore, the heap size satisfies:

$$HS(A, P_{AW}) \geq min\left(\frac{1}{3}Md, \left(\frac{1}{3}M\frac{\log n}{\log d} - 2n\right) \cdot \left(1 - \frac{d}{c}\right)\right),$$

as required and we are done with the proof of Claim 4.10. □

4.2 Bounding the space consumption when the objects are not necessarily aligned

In the general form of the allocation problem, there are no alignment constraints on the memory allocator, and the objects can be allocated anywhere. In this section, we modify the *Aligned-Waster* program to a different, *Waster* program denoted P_W, and extend the proofs presented in the previous section to fit any allocator that works with the program *Waster*. Specifically, our aim in this section is to prove Theorem 2.

Theorem 2. (Lower bound.) *For all $c \geq 1$, and all $M \geq n > 4$, there exists a program $P_W \in \mathcal{P}(M, n)$ such that for all allocators A that satisfy the compaction bound $\frac{1}{c}(S)$ the following holds:*

$$HS(A, P_W) \geq \begin{cases} \frac{1}{10}M \cdot min\left(c, \frac{\log n}{\log c+1} - \frac{5n}{M}\right) & \text{if } c \leq 4\log n \\ \frac{1}{6}M \cdot \frac{\log n}{\log\log n+2} - \frac{n}{2} & \text{if } c > 4\log n \end{cases}$$

The program *Waster*, or P_W is specified in Algorithm 4. This program is almost identical to the algorithm of P_{AW} (*Aligned-Waster*), except for the deletion step. In Phase i, P_W allocates objects of size 2^i exactly like P_{AW}, but unlike P_{AW}, the program P_W considers areas of size 2^{i-2} in the deletion step. P_W removes objects on these smaller areas if enough space is left allocated on these areas. A new phenomenon that occurs in the execution of P_W is the fact that objects can spread across area boundaries. Such an object can be removed only if both areas on which it is placed remain dense enough, i.e., have enough allocated space on them, after the removal.

Algorithm 4 Waster Program

Input: M, n, and c.
1: Compute d as a function of c (to be determined).
2: **for** $i = 0$ to $\log n$ **do**
3: { Deletion step: }
4: Partition the memory into aligned areas of size 2^{i-2}
5: Remove as many objects as possible from each area,
6: subject to leaving at least $2^{i-2}/d$ space occupied.
7: { Allocation step: }
8: Request allocation of as many objects as possible of
9: size 2^i (not exceeding the overall allocated size M).
10: **end for**

Notice that since the deletion considers areas whose size is $1/4$ of the next allocation size, then newly allocated objects will always cover at least 3 such areas fully. The remaining quarter of the object may partially cover the other two neighboring areas.

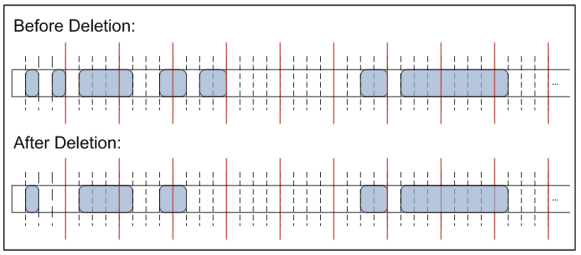

Figure 3. An example of deleting objects in *Waster*, with parameter $1/d = 1/4$, and Phase $i = 4$. The areas size is $2^{4-2} = 4$, and the next allocation size is 2^4.

Similarly to the aligned version of the proofs, we bound the total space used as the total allocated space S in the execution minus the amount of reuse R. Note that Equation 4.4 stating that

$$HS(A, P) \geq S(A, P) - R(A, P)$$

holds in the general case as well. We now proceed to bounding S and R with no alignment restrictions and given the modified program P_W.

Claim 4.13. *Consider the execution of P_W with any memory manager A that satisfies the compaction threshold $\frac{1}{c}S$. Let Q be the total amount of compaction during the execution, let R be the reuse, and S the total allocated space, and d be the deletion threshold set by* Waster. *Then following holds:*

$$R(A, P_W) \leq \frac{1}{4}S(A, P_W) + Q(A, P_W) \cdot d.$$

We prove Claim 4.13 by summing on the reuse of all of the allocations during the run of P_W. In order to do that we define the following: (same as the definitions in Section 4.1)

Definition 4.14. *If o is an object that is placed on the heap during the run of Program* Waster, *then $|o|$ is it's size. r_o is the space size reused by the allocation of the object o. r_o is defined as the total size of objects parts that existed in the space that is later occupied by object o. These objects were removed or compacted, but there was no object placed on top of them between their removal/compaction and until the allocation of Object o. Namely, the relocation was used for the reuse of o, and not for any previous reuse. q_o is the total size of objects that were located in the space occupied later by object o. These objects were compacted, but there was no object placed on top of them until the allocation of o.*

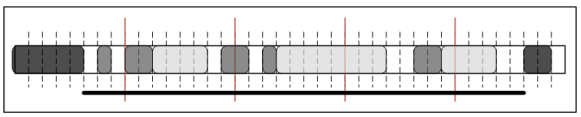

Figure 4. An example the definitions o, r_o and q_o in the execution of *Waster*, with parameter $1/d = 1/4$, and phase $i = 5$.

An example of these definitions is in Figure 4. In this figure, a fraction of the heap of size 36 words is shown. The dark line beneath the squares denotes the location where object o is placed in Phase 5 during the execution of *Waster*. The horizontal lines denote the separations between the areas as defined in Phase 5. $|o| = 32$, and the areas size is 8. The fraction of the heap where o is placed is empty at the moment of placement. The black squares denote objects that are present in the heap at the time of the allocation. The dark gray squares represent words in the heap, that the last operation on this word was a compaction of an object. The light gray squares represent words in the heap, that the last operation

on this word was a deletion of an object. The white squares are words on the heap that were empty since the beginning of the run of the program. In this example, q_o equals to the total size of the objects that were last compacted - i.e. to the total size of the dark gray words, 8 words. r_o equals to the total size of the deleted + compacted objects in where o is placed, i.e. 24 words.

Proof of Claim 4.13. According to definition 4.14, the total reuse R can be calculated as the sum of reuse of all allocated objects. $R = \sum_j r_{o_j}$, and the total compaction Q is larger or equal to the size of compacted reused objects $Q \geq \sum_j q_{o_j}$, where the index j traverses over all of the objects that were allocated during the Program *Waster* run. According to Claim 4.15, for each allocated object o_j it holds that $r_{o_j} \leq \frac{1}{4}|o_j| + q_{o_j} \cdot d$, therefore:

$$R = \sum_j r_{o_j} \leq \sum_j \frac{1}{4}|o_j| + \sum_j q_{o_j} \cdot d \leq \frac{1}{4}S + Q \cdot d.$$

In order to finish this proof it remains to present and prove Claim 4.15.

□

Claim 4.15. *Consider any phase i during the execution of* Waster. *Observe any object o just before it is being allocated in the heap during phase i. It holds:*

$$r_o \leq \frac{1}{4}|o| + q_o \cdot d.$$

Proof. According to Algorithm 4, in Phase i the size of allocated objects is 2^i, therefore $|o| = 2^i$. Furthermore, in the deletion step of Phase i, the memory is divided into areas of size 2^{i-2}. Therefore, the object o covers at least three such areas fully, and at most $\frac{1}{4}|o|$ covers two areas partially. In the partially covered areas, the reuse is at most the size of the covering part: $r_{o(partial)} \leq \frac{1}{4}|o|$. We now apply Algorithm 5 to each of the fully covered areas a_1, a_2, a_3. The result of Algorithm 5 is a sub division of an input area into distinct aligned sub-areas of size 2^k s.t. $0 \leq k \leq 2^{i-2}$. We then bound the reuse in each sub-area. Summing over the reuse in the sub-areas equals to the total reuse in the full area - which is exactly what we are trying to bound.

Algorithm 5 Division of an area a into sub-areas

Input: An area a of size 2^{i-2}.
1: List $L \leftarrow$ a new empty list
2: Stack $S \leftarrow$ a new empty stack
3: $S.Push(a, i-2)$
4: **while** S is not empty **do**
5: $(x, k) \leftarrow S.Pop()$
6: **if** $k = 1$ **then**
7: $L.Push(x, k)$
8: **else**
9: Look at the removal step in phase $k+2$, during the run of *Waster*. Check if there were any removals in area x during that step.
10: **if** there was at least one removal **then**
11: $L.Push(x, k)$
12: **else**
13: Divide x into its two equal sub areas x_1, x_2.
14: $S.Push(x_1, k-1)$
15: $S.Push(x_2, k-1)$
16: **end if**
17: **end if**
18: **end while**
19: **return** L.

The result of Algorithm 5 is a list L that contains sub-areas of the area a. The sub-areas are distinct, their size is 2^k for $k \in \mathbb{N}, 0 \leq k \leq i-2$. The union of the sub-areas is exactly the area a. For every sub area that is the output of algorithm 3, one of the following holds:

1. The sub was either was always empty, or was occupied by an object that was compacted away. The sub-area size is 1.
2. The sub area size is 2^k $s.t.$ $k \geq 1$, and last time objects were deleted from this sub-area happened during the deletion step of phase $k+2$ in Algorithm *Waster*.

The statement above follows directly from the behavior of Algorithm 5. An area of size 2^k is sub-divided only if it had no deletions in phase $k+2$. Therefore, when an area reaches size 1, it never had any deletions, and statement (1.) is true. Since Algorithm 5 divides each aligned area of size 2^k to its two equal halves, the resulting sub-areas are the exact same areas that algorithm *Waster* considered in phase $k+1$. The algorithm actually goes back from phase i (looking at areas of size $i-2$) back to phase 3 (looking at areas of size 1) looking for the last deletion for each sub-area. For all sub-areas $sa(k)$ resulting from algorithm 5, we examine cases (1.) and (2.).

In case (1.), it holds that there were no removals at all from the sub-area $sa(k)$. The reuse, if larger than 0, occurs since an object was allocated and later compacted (possibly more than once). In this case, the reuse in phase i of the sub-area $sa(k)$ equals to the size of the object that was last compacted away from this sub-area: $r_{sa(k)} = q_{sa(k)}$. Since the deletion threshold holds $d \geq 1$, then:

$$r_{sa(k)} \leq q_{sa(k)} \cdot d.$$

In case (2.), for each sub-area $sa(k)$, phase $k+2$ captures the time of the last deletion from the sub area $sa(k)$. If there was no reuse in sub-area $sa(k)$ between phases $k+2$ and i (the first allocation on sub-area $sa(k)$ after phase $k+2$, occurred in phase i), then we compute the reuse based on the following. According to the definition of the adversarial program *Waster* behavior, for each such area, after each removal, at least $1/d$ of the area remains occupied, we denote these objects by q. These objects must be later compacted away from this sub-area, so that the space could be reused (as in phase i, this sub-area is empty). Therefore, if we multiply the size of this compaction by d, we get the size of the sub-area $sa(k)$. The reuse size in this sub-area is smaller or equal to the sub-area size, therefore, in this case:

$$r_{sa(k)} \leq q_{sa(k)} \cdot d.$$

Another option, is that the area was already reused (fully or partially) - an object was already placed on the sub-area $sa(k)$, and was later compacted away. In this case, it holds that the size that was compacted but never reused is at least $1/d$ of the sub-area $sa(k)$ size. This statement is true since after the last deletion, at least $1/d$ of the area remained occupied. If we notice only these locations in the heap, we can see that objects were compacted, and possibly partially reused by other objects between phases $k+2$ and i, but these "reusing" objects were later compacted, as the sub-area is eventually empty. Therefore, in this case also there is always $1/d$ of the area that was compacted away, and not yet reused. So we get that:

$$r_{sa(k)} \leq q_{sa(k)} \cdot d.$$

This statement is true for all sub-areas. The sum of $r_{sa(k)}$, where k denotes the sub-area k of a, holds $r_a = \Sigma_k r_{sa(k)}$, and the sum of $q_{sa(k)}$, where k denotes the sub-area k of a, holds $q_a = \Sigma_i q_{sa(k)}$. Therefore, it holds that

$$r_a \leq q_a \cdot d$$

for each one of the full areas a_1, a_2, a_3. When we sum the total reuse r_o, we get:

$$r_o \leq \frac{1}{4}|o| + r_{a_1} + r_{a_2} + r_{a_3} \leq \frac{1}{4}|o| + (q_{a_1} + q_{a_2} + q_{a_3}) \cdot d \leq \frac{1}{4}|o| + q_o \cdot d.$$

This concludes the proof of Claim 4.15 □

Next we assert the analogue of Claim 4.8 for the non-aligned case.

Claim 4.16. *For all allocators A such that A satisfies the compaction bound $\frac{1}{c}S$, s.t. $c \geq 1$, $M, n > 0$. The total space allocated during the execution of P_W with A is S, and the deletion threshold is $d \geq 1$, it holds that:*

$$HS(A, P_W) \geq S(A, P_W) \left(\frac{3}{4} - \frac{d}{c} \right)$$

Proof. The heap size is at least the total allocated space minus the reuse, i.e.,

$$HS(A, P_W) \geq S(A, P_W) - R(A, P_W). \tag{4.5}$$

According to Claim 4.13, it holds that

$$R(A, P_W) \leq \frac{1}{4} S(A, P_W) + Q(A, P_W) \cdot d. \tag{4.6}$$

The total amount of compaction is bounded by:

$$Q(A, P_W) \leq \frac{1}{c} \cdot S(A, P_W). \tag{4.7}$$

By Equations 4.5, 4.6, and 4.7, we get

$$HS(A, P_W) \geq S(A, P_W) \left(\frac{3}{4} - \frac{d}{c} \right)$$

and we are done. □

Proceeding with the analogue proof for the non-aligned case, we now state the key claims, which assert the lower bound for large c's and for general c's. Note that the constants are less tight in this case.

Claim 4.17. *For any $c > d > \log n$, for all allocators A that satisfy the compaction threshold $\frac{1}{c}S$, and for all $M \geq n > 4$, it holds that:*

$$HS(A, P_W) \geq \frac{1}{6} \cdot M \cdot \frac{\log n}{\log \log n + 2} - 2n \left(\frac{3}{4} - \frac{d}{c} \right)$$

Claim 4.18. *For all allocators A that satisfy the compaction threshold $\frac{1}{c}S$, s.t. $c, d \geq 1$, $M \geq n > 1$, it holds:*

$$HS(A, P_W) \geq min \left(\frac{1}{5} Md, \left(\frac{2}{5} M \frac{\log n}{\log d + 2} - 2n \right) \cdot \left(\frac{3}{4} - \frac{d}{c} \right) \right)$$

Proceeding in the same avenue used in Section 4.1 to prove Lemma 4.3, we now show that these two key claims imply Theorem 2. To this end we choose the density parameter d for P_W to be equal to $\frac{c}{2}$. Claim 4.17 holds for $\frac{1}{2}d > \log n$, and therefore for $c > 4 \log n$ and in this case we get

$$HS(A, P_W) \geq \frac{1}{6} \cdot M \cdot \frac{\log n}{\log \log n + 2} - \frac{n}{2}.$$

For the case that $c \leq 4 \log n$ we can use Claim 4.18 and deduce that

$$HS(A, P_W) \geq \frac{1}{10} \cdot M \cdot min \left(c, \frac{\log n}{\log c + 1} - \frac{5n}{M} \right).$$

These two equations yield exactly the lower bound of Theorem 2 as required.

To finish the proof of the lower bound, we need to prove Claims 4.17 and 4.18. Similarly to what we did in Section 4.1, we start with some properties of areas as induced by the activity of the program P_W.

During the execution of P_W with any allocator, after the deletion step in every phase i, the heap is partitioned into areas of size 2^{i-2}. The objects located in the heap could be objects of two types: (1.) objects that are located fully inside an area, and do not cross area boundaries or (2.) objects that are located on top of area boundaries. We first address the minimal heap size required by objects of type (1.), and later discuss the minimal heap size required by objects of type (2.). The maximal of the two is the total heap size required in the execution of phase i.

Claim 4.19 (The analogue of Claim 4.11 in Section 4.1)**.** *Consider any execution of P_W with any allocator A, and any Phase i, $2 \leq i \leq \log n$. Let d be the deletion threshold, so that $d \geq 1$. Consider the space consumed by objects that are fully contained inside any non empty aligned area of size 2^{i-2}, just after the deletion step in Phase i. There are two possible configurations for this space:*

1. *The space consumed by these objects is smaller than $\frac{2^{i-2}}{d} \cdot 2$ (and each object size is smaller than $\frac{2^{i-2}}{d}$).*
2. *The area contains fully exactly one object that is larger or equal to $\frac{2^{i-2}}{d}$.*

Note that each area might also contain objects that are not fully contained inside it. Namely, objects that cross area boundaries. We ignore these objects in this claim. Of-course, the proof holds in the presence of such objects. Let us now prove Claim 4.19

Proof. Recall that the deletion in Phase i attempts to delete as much space as possible from every area of size 2^{i-2}, while still leaving a space of size at least $\frac{2^{i-2}}{d}$ allocated. Suppose that after the deletion we have an object whose size is at least $\frac{2^{i-2}}{d}$, and it is fully allocated inside an area. Then before the deletion this object existed in the area and if the deletion left this object allocated, then it must delete all other objects that are fully allocated in the area, and it must be the only object that is fully allocated inside this area (there might only be objects that are located on top of this area boundaries, in addition to itself), satisfying the second case.

Otherwise, after the deletion all objects that are contained fully inside the area are smaller than $\frac{2^{i-2}}{d}$. Let the smallest of the remaining fully contained objects be of size 2^j for some $j \leq i-2$. Removing this object was not possible in the deletion step, therefore, the occupied space in this area can be at most $\frac{2^{i-2}}{d} + 2^j - 1$, which is smaller than $\frac{2^{i-2}}{d} \cdot 2$. □

The above claim shows that small objects that are fully contained inside areas must be sparsely allocated after a deletion step. In particular, the space consumed by objects that are smaller than $2^{i-2}/d$, and are fully allocated in an area, is smaller than $2/d$ of the area size. We generalize this sparseness of small objects in the next claim.

Claim 4.20. *Consider an execution of P_W against any allocator A. Let $k \leq d$ be a size threshold, and $x_i^1(k)$ be the total space consumed by objects that are smaller than $\frac{2^{i-2}}{k}$, and are located fully inside areas of size 2^{i-2} after the deletion step in Phase i. Then after the deletion step in any Phase i in the execution it holds that*

$$HS(A, P_W) \geq \begin{cases} x_i^1(k) \cdot k & \text{if } k \leq \frac{1}{2}d \\ x_i^1(k) \cdot \frac{1}{2} \cdot k & \text{if } \frac{1}{2}d \leq k \leq d \end{cases}$$

Proof. Consider the state of the heap after the deletion step in any Phase i. In this phase the areas considered for deletion are of size 2^{i-2}. All objects that are smaller than $2^{i-2}/k$, and reside fully inside an area belong to $x_i^1(k)$. Consider any area that contains an object in $x_i^1(k)$. According to Claim 4.19 either there is only

one object in the area and its size lies between $2^{i-2}/d$ and $2^{i-2}/k$, or the size of all remaining objects fully contained inside the area is at most $\frac{2^{i-2}}{d} \cdot \frac{1}{2}$. If $k \leq \frac{1}{2}d$, then the size of the area is larger by a factor of at least k than the occupied space on this area with objects from $x_i^1(k)$. If $\frac{1}{2}d \leq k \leq d$, then the size of the area is larger by a factor of at least $\frac{1}{2}k$ than the occupied space on this area with objects from $x_i^1(k)$. The size of the heap must be at least the sum of all areas containing objects in $x_i^1(k)$. Therefore, we get $HS(A, P_W) \geq x_i^1(k) \cdot k$ or $HS(A, P_W) \geq x_i^1(k) \cdot \frac{1}{2} \cdot k$ as required. Note that the heap size is monotone, and therefore the obtained bound holds for the heap size of the entire execution. □

Claim 4.21. *Consider an execution of P_W against any allocator A. Let $k \leq d$ be a size threshold, and $x_i^2(k)$ be the total space consumed by objects that are smaller than $2^{i-2}/k$, and are located on top of area boundaries, after the deletion step in Phase i. Then after the deletion of any Phase i in the execution it holds that*

$$HS(A, P_W) \geq x_i^2(k) \cdot k$$

Proof. We compute a lower bound on the total space used, by bounding the number of areas used after the deletion step of Phase i. The total number of existing areas in the heap is at least the number of area boundaries. The number of area boundaries must be at least the number of objects smaller than 2^{i-2}, that are located on top of them. The number of objects whose space is accounted for in $x_i^2(k)$ is at least the accumulated size $x_i^2(k)$ divided by the largest possible object size, i.e., $\frac{x_i^2(k)}{\frac{2^{i-2}}{k}} = \frac{x_i^2(k) \cdot k}{2^{i-2}}$. Each area size is 2^{i-2}, and therefore, the total space consumed is at least $x_i^2(k) \cdot k$ and we are done. □

Claim 4.22. *Consider an execution of P_W against any allocator A. Let $k \leq d$ be a size threshold, and $x_i(k)$ be the total space consumed by objects that are smaller than $2^{i-2}/k$ after the deletion step in Phase i. Then after the deletion step of any Phase i in the execution, it holds that*

$$HS(A, P_W) \geq \begin{cases} x_i(k) \cdot \frac{1}{2} \cdot k & \text{if } k \leq \frac{1}{2}d \\ x_i(k) \cdot \frac{1}{3} \cdot k & \text{if } \frac{1}{2}d \leq k \leq d \end{cases}$$

Proof. According to the definitions, $x_i(k) = x_i^1(k) + x_i^2(k)$. For $k \leq \frac{1}{2}d$, according to Claims 4.20 and 4.21, it holds that $HS(A, P_W) \geq x_i^1(k) \cdot k$ and also $HS(A, P_W) \geq x_i^2(k) \cdot k$. Therefore, the minimal value of the heap is resulted when $x_i^1(k) = x_i^2(k) = \frac{1}{2}x_i(k)$, which leads to the conclusion that

$$HS(A, P_W) \geq \frac{1}{2}x_i(k) \cdot k$$

. For $\frac{1}{2}d \leq k \leq d$, according to Claims 4.20 and 4.21, it holds that $HS(A, P_W) \geq x_i^1(k) \cdot \frac{1}{2} \cdot k$ and also $HS(A, P_W) \geq x_i^2(k) \cdot k$. To get the minimal heap value, both of these bounds should give the same value, i.e.:

$$x_i^1(k) \cdot \frac{1}{2} \cdot k = x_i^2(k) \cdot k.$$

Therefore, $x_i^2(k) = \frac{1}{3}x_i(k)$, and

$$HS(A, P_W) \geq \frac{1}{3}x_i(k) \cdot k,$$

as required. □

We now prove the key claims. Intuitively, the proof will argue that either there is a point in the run of the program with many small objects that are so sparsely allocated that the lower bound follows easily, or there is no such point, but then there must be many memory allocations during the execution of P_W with A, i.e. S is large, and therefore Claim 4.16 implies the correctness of the lower bound.

Proof of Claim 4.17. We show that for all allocators A that satisfy the compaction bound $\frac{1}{c}S$, s.t. $c \geq 1$ and $M \geq n > 4$, and given that the density threshold d set by P_W satisfies $\frac{1}{2}d > \log n$, the following holds:

$$HS(A, P_W) \geq \frac{1}{6} \cdot M \cdot \frac{\log n}{\log \log n + 2} - 2n\left(\frac{3}{4} - \frac{d}{c}\right).$$

We use Claim 4.22, and set the size bound $k = \log n$. We can use Claim 4.22 since $k = \log n < \frac{1}{2}d$. We also use a fraction parameter α, satisfying $0 < \alpha < 1$, to be determined later. We partition the analysis into two cases. First, if there is a Phase i where more than $(1-\alpha)M$ of the live space consists of objects smaller than $\frac{2^{i-2}}{\log n}$, then by Claim 4.22, the total heap size is larger than $\frac{1}{2}(1-\alpha)M \cdot \log n$. Otherwise, in all phases it holds that more than $\alpha \cdot M - 2^i$ of the live space consists of objects that are larger than $\frac{2^{i-2}}{\log n}$. This means that in every phase i, at least $\alpha \cdot M - 2^i$ of the live space must consist of objects that were created in the last $\log \log n + 2$ phases. The execution consists of $\log n$ phases. We partition the program execution into consecutive sections of $\log \log n + 2$ phases each. In each such section, at least $\alpha \cdot M - 2^i$ space was allocated. Therefore, the total allocation in the execution, S, satisfies

$$\begin{aligned} S(A, P_W) &\geq \alpha \cdot M \cdot \frac{\log n}{\log \log n + 2} - \sum_{i=\log \log n}^{\log n} 2^i \\ &\geq \alpha \cdot M \cdot \frac{\log n}{\log \log n + 2} - 2n. \end{aligned}$$

According to the relation between total allocation and heap size asserted in Claim 4.16, we get that

$$HS(A, P_W) \geq \left(\alpha \cdot M \cdot \frac{\log n}{\log \log n + 2} - 2n\right) \cdot \left(\frac{3}{4} - \frac{d}{c}\right).$$

Setting $\alpha = \frac{1}{6\left(\frac{3}{4} - \frac{d}{c}\right)}$ and using the fact that $d \leq \frac{c}{2}$, we get that the bound in the first case is

$$HS(A, P_W) \geq \frac{1}{6}M \cdot \log n.$$

With this α, the bound in the second case becomes

$$HS(A, P_W) \geq \frac{1}{6} \cdot M \cdot \frac{\log n}{\log \log n + 2} - 2n\left(\frac{3}{4} - \frac{d}{c}\right).$$

In both cases Claim 4.17 holds and we are done. □

Proof of Claim 4.18. We need to show that for all allocators A that satisfy the compaction bound $\frac{1}{c}S$, with $c \geq 1$, $M \geq n > 2$, it holds that:

$$HS(A, P_W) \geq \frac{1}{5} \cdot M \cdot min\left(d, \frac{2\log n}{\log d + 2} \cdot \left(\frac{3}{4} - \frac{d}{c}\right)\right).$$

Similarly to the proof of Claim 4.17, we use Claim 4.22 and look at two possible cases. This time we set $k = d$. If there is a Phase i where more than $\frac{3}{5}M$ of the live space consists of objects smaller than $\frac{2^{i-2}}{d}$, then the total heap size is larger than $\frac{1}{5}M \cdot d$.

Otherwise, in every phase i during the run of the program, it holds that more than $\frac{2}{5}M - 2^i$ of the live space consists of objects that are larger than $\frac{2^{i-2}}{d}$. Partitioning the program run into

consecutive sections of $\log d + 2$ phases each, we get that in each such section, at least $\frac{2}{5}M - 2^i$ words was allocated. Therefore,

$$S(A, P_W) \geq \frac{2}{5}M \cdot \frac{\log n}{\log d + 2} - \sum_{i=\log\log n}^{\log n} 2^i \geq \frac{2}{5}M \cdot \frac{\log n}{\log d + 2} - 2n.$$

By Claim 4.16, we get that

$$HS(A, P_W) \geq \left(\frac{2}{5}M\frac{\log n}{\log d + 2} - 2n\right) \cdot \left(\frac{3}{4} - \frac{d}{c}\right)$$

Combining the two cases we get that

$$HS(A, P_W) \geq min\left(\frac{1}{5}Md, \left(\frac{2}{5}M\frac{\log n}{\log d + 2} - 2n\right) \cdot \left(\frac{3}{4} - \frac{d}{c}\right)\right)$$

□

5. Segregated Free List Allocator

In the previous section we presented a specific program that manages to create a large space overhead for any possible memory allocation method. In practice, systems implement specific allocators that can be specifically studied. It is possible that a program can cause more overhead to a given specific allocator. One very common memory allocator is the *segregated free list* allocator, and in particular, the one that is block-oriented (e.g., [2, 6, 8]). In this section we study limits of this specific allocator and prove tighter lower bounds based on its specific behavior. We start by defining this segregated free list allocation method. We then show bounds on the space requirement when no compaction is allowed, and finally, in Section 5.3, we show bounds on the space requirements when partial compaction is used to aid in reducing fragmentation for a segregated free list allocation.

5.1 Definition of a Segregated Free List Allocator

A segregated free list allocator divides the free list into several subsets, according to the size of the free chunks. Each subset forms a list of free chunks of the same size (or a small range of sizes) and an array of pointers is used to index the various free lists. A freed object is placed on the appropriate list according to its size. An allocation request is serviced from the appropriate list.

Segregated free lists are typically implemented in a block oriented manner [2, 6, 8]. The heap is partitioned into blocks (typically, of a page size, i.e., 4KB) and each block may only contain objects of one size. Whenever a full block is freed, it is returned to the pool of free blocks. Whenever an allocation is requested for an object whose free list is empty, a free block is pulled from the block pool, it is partitioned into as many free chunks as possible in an aligned manner, and then the new chunks are added to the appropriate free list to allow allocation.

The free lists are known as buckets. Each bucket is characterized by the chunk size (or range of chunk sizes) that can be allocated in it. The allocation within each bucket is executed in a first-fit manner, in the first free chunk in the bucket's list that can satisfy the allocation. The simplicity of this method comes with a cost. This method is susceptible to high external fragmentation, as shown below.

In the following sections we simplify the discussion by assuming that the maximal object size, n, equals the block size. This is close to what happens in practice, and small adjustments, e.g., if the block size is $2n$, have small impact on the results. As before, we denote by M the total space that can be allocated simultaneously.

5.2 Memory usage without compaction

Let us start with the case that no compaction is used. The general lower bound of Robson [15, 16] holds in this case, implying that an overhead factor of at least $\times\frac{1}{2}\log n$ can occur in practice for the worst fragmenting program. We extend this bound for the specific segregated free list allocator. Denote by $s_1, \ldots, s_k$ the different maximal object sizes inside each bucket, and assume for simplicity that the program only allocates objects whose sizes are one of $s_1, \ldots, s_k$. (When proving a lower bound, it is enough to show that there exists one bad program.) A block contains objects of the same size, but this can be any size as long as $s_k \leq n$. As stated earlier, for simplicity we also assume that $n = s_k$.

Different segregated free list allocators have different segregation policies, which are determined by the size vector $s_1, \ldots, s_k$. We study two rather extreme cases by looking at an allocator that keeps a bucket for each possible object size (i.e., a lot of buckets) and an allocator that has buckets for exponential object sizes, i.e., sizes that increase by a factor of two. Typical implementations use something in between these extreme cases and the tools developed here can be used to explore each specific set of sizes.

We first look at the allocator SFL_{all} that implements the segregated free list method, with a full assortment of bucket sizes: $s_1 = 1, s_2 = 2, s_3 = 3, s_4 = 4, ..., s_n = n$. Let the class $\mathcal{P}(M, [s_1, \ldots, s_n])$ be the class of all programs that always keep the allocated space smaller or equal to M, and allocates objects only of the sizes $s_1, ..., s_n$. The theorems below show that with no compaction, the space overhead is quite large. It is of the order of the square root of n. This is much higher than the logarithmic factor that was shown for the general allocator.

Theorem 3. *There exists a program $P \in \mathcal{P}(M, [s_1, \ldots, s_n])$, and the heap size required for allocator SFL_{all} to execute the allocation requests of P satisfies:*

$$HS(SFL_{all}, P) \geq \frac{4}{5}M\sqrt{n} - \frac{1}{2}n - \frac{1}{2}\sqrt{n}.$$

The result for the smaller set of buckets is very different. Particularly, let SFL_{log} be an allocator that implements the segregated free list method, with a logarithmic assortment of bucket sizes: $s_1 = 1, s_2 = 2, s_3 = 4, s_4 = 8, ..., s_{\log n+1} = n$. Let $\mathcal{P}(M, [s_1, \ldots, s_{\log n+1}])$ be a set of programs that always keep the live space smaller or equal to M, and allocates objects only of the sizes $s_1, ..., s_{\log n+1}$.

Theorem 4. *There exists a program $P \in \mathcal{P}(M, [s_1, \ldots, s_{\log n+1}])$, and the heap size required for allocator SFL_{log} to execute the allocation requests of P satisfies:*

$$HS(SFL_{log}, P) \geq M \log n - \frac{4}{3}M - n.$$

Note that Theorems 3 and 4 improve on the general results obtained by Robson [15, 16]. Robson could only show an overhead of $\frac{1}{2}\log n$ since he needed to work with a general allocator, unlike the segregated free list allocator that we assume in this section.

To prove these theorems, we present the *Bucket-Waster* program, denoted P_{BW}, which causes heavy space overhead for an allocator that uses the segregated free list allocation method. We then analyze its behavior and finally prove Theorems 3, and 4.

Algorithm 6 Program P_{BW} with bucket sizes $s_1, \ldots, s_k$.

1: **for** $i = 1$ to k do **do**
2: Allocate as many object as possible of size s_i
3: Deallocate as many objects as possible subject to leaving exactly one object in each allocated block.
4: **end for**

Below we investigate the execution of P_{BW} with a segregated free list allocator. This run is structured, and satisfies the following property. After each deallocation step, there will be a single object in each allocated block. Moreover, this single object will not be

deleted until the end of the execution. Therefore, the objects that are deallocated in Phase i, are only objects allocated in (the first step of) Phase i and they are all of size s_i. Denote by HS_k the size of the heap used by the *SFL* allocator to satisfy the allocation requests of P_{BW} up to the point at the end of Phase k. For now, the *SFL* allocator uses a general form of the bucket sizes: $s_1, \ldots, s_k$. The concrete choice of bucket sizes will be determined later. Denote by M_i the space that the allocator can allocate at the beginning of phase i. M_i depends on the total live space to allocate, M, and the space that is already in the heap in the form of objects that were placed on the heap, and were not removed by the beginning of phase i. We first present a lower bound on M_i, then continue to prove a lower bound on HS_k, and finally prove the theorems presented in the beginning of this section.

Claim 5.1. *In the execution of P_{BW} against the SFL allocator, let M_i be the total space available for allocation in Phase i. s_i the objects size in Phase i, and n the block size. In the first phase it holds that $M_1 = M$. In subsequent phases it holds that:*

$$M_{i+1} \geq M_i - \left\lceil \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rceil \cdot s_i.$$

Proof. In the first phase the space to allocate equals the maximal live space size: $M_1 = M$. In subsequent phases, M_i depends on how much live space is already allocated in the form of smaller objects. The number of objects allocated in Phase i is $\left\lfloor \frac{M_i}{s_i} \right\rfloor$. Any block can contain $\left\lfloor \frac{n}{s_i} \right\rfloor$ objects of size s_i. Note that the allocator must get new blocks for the allocation of Phase i since a block can only hold allocations of a particular size. Therefore, the number of blocks used in the allocation of Phase i is at least $\left\lceil \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rceil$. According to the deletion strategy in Algorithm 6, after each deletion step, a single object remains in every block. Therefore, the size of objects of size s_i remaining in the heap after the deletion step in Phase i is $\left\lceil \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rceil \cdot s_i$. The space remaining for allocation in Phase $i+1$ is the live space we had in Phase i minus what remains live in the heap after the removal step in Phase i:

$$M_{i+1} = M_i - \left\lceil \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rceil \cdot s_i.$$

□

First, we remove the integral operations by introducing an error factor.

Definition 5.2 (Definition of the error factor ϵ_i)**.** *Denote by ϵ_i, the difference between the number of actual blocks in Phase i, and the non-integral value:*

$$\epsilon_i = \left\lceil \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rceil - \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor}.$$

Using ϵ_i, we can simplify the value of M_i in the next claims.

Claim 5.3. *In the execution of P_{BW} against the SFL allocator, let M_i be the total space available for allocation in Phase i. s_i the objects size in Phase i, and n the block size. In the first phase it holds that $M_1 = M$. In subsequent phases it holds that:*

$$M_{i+1} \geq M - M \cdot \sum_{j=1}^{i} \frac{s_j}{n - s_j} - \sum_{j=1}^{i} \epsilon_j \cdot s_j.$$

Proof. First, we use the definition of ϵ_i within Claim 5.1, simplifying the value of M_i:

$$M_{i+1} = M_i - \left(\frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} + \epsilon_i \right) \cdot s_i.$$

Next, we bound the non integral parts in the above equation:

$$M_{i+1} \geq M_i - \left(\frac{\frac{M_i}{s_i}}{\frac{n}{s_i} - 1} + \epsilon_i \right) \cdot s_i \geq M_i \cdot \left(1 - \frac{s_i}{n - s_i} \right) - \epsilon_i \cdot s_i.$$

After repeatedly substituting the values of M_i in this equation, tracing back until $M_1 = M$, we get that:

$$M_{i+1} \geq M \cdot \prod_{j=1}^{i} \left(1 - \frac{s_j}{n - s_j} \right) - \sum_{j=1}^{i} \epsilon_j \cdot s_j \cdot \prod_{t=j+1}^{i} \left(1 - \frac{s_t}{n - s_t} \right).$$

We use the fact that for any positive $\alpha_1, \ldots, \alpha_i$ it holds that

$$\prod_{j=1}^{i} (1 - \alpha_j) \geq 1 - \sum_{j=1}^{i} \alpha_j. \tag{5.1}$$

to simplify the equation above. The result is:

$$\begin{aligned} M_{i+1} &\geq M \left(1 - \sum_{j=1}^{i} \frac{s_j}{n - s_j} \right) - \sum_{j=1}^{i} \epsilon_j \cdot s_j \left(1 - \sum_{t=j+1}^{i} \frac{s_t}{n - s_t} \right) \\ &\geq M \left(1 - \sum_{j=1}^{i} \frac{s_j}{n - s_j} \right) - \sum_{j=1}^{i} \epsilon_j \cdot s_j \end{aligned}$$

which ends the proof. □

This bound on M_i will be used in the next claims as we continue to prove a lower bound on HS_k.

Claim 5.4. *In the execution of P_{BW} against the SFL allocator, let HS_k be the heap size after phase k, M_i be the total space size available for allocation in Phase i. s_i the objects size in Phase i, and n the block size. The heap size after Phase k, HS_k, holds the following:*

$$HS_k = \sum_{i=1}^{k} \left\lceil \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rceil \cdot n.$$

Proof. The number of blocks used in the allocation of Phase i is at least $\left\lceil \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rceil$. Since each block has size n, this means that the additional heap size that is required for the new allocations of Phase i, is $\left\lceil \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rceil \cdot n$. According to the deletion strategy in Algorithm 6, after each deletion step, a single object remains in every block. Therefore, no blocks are ever removed, and the total heap size after Phase k, HS_k, equals the size of blocks occupied after this phase:

$$HS_k = \sum_{i=1}^{k} \left\lceil \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rceil \cdot n.$$

□

We simplify the bound above using the values of ϵ_i, and M_i achieved in previous claims.

Claim 5.5. *In the execution of P_{BW} against the SFL allocator, for any phase k, the heap size after phase k, HS_k satisfies:*

$$HS_k \geq M \cdot k - M \cdot \sum_{i=1}^{k-1} \frac{s_i}{n - s_i}(k-i) + \sum_{i=1}^{k} \epsilon_i \left(n - s_i \cdot (k-i)\right) - \sum_{i=1}^{k} s_i.$$

Proof. First, we use the error factor ϵ_i to simplify the value of HS_k as appears in Claim 5.4

$$\begin{aligned} HS_k &= \sum_{i=1}^{k} \left\lceil \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rceil \cdot n = \sum_{i=1}^{k} \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \cdot n + \epsilon_i \cdot n \\ &\geq \sum_{i=1}^{k} \frac{\frac{M_i}{s_i} - 1}{\frac{n}{s_i}} \cdot n + \epsilon_i \cdot n \geq \sum_{i=1}^{k} (M_i - s_i + \epsilon_i \cdot n). \end{aligned}$$

Next, we replace the value of M_i with the result we got in Claim 5.3:

$$\begin{aligned} HS_k &\geq \sum_{i=1}^{k} (M - M \cdot \sum_{j=1}^{i-1} \frac{s_j}{n - s_j} - \sum_{j=1}^{i-1} \epsilon_j \cdot s_j - s_i + \epsilon_i \cdot n) \\ &\geq M \cdot k - M \cdot \sum_{i=1}^{k-1} \frac{s_i}{n - s_i}(k-i) \\ &\quad + \sum_{i=1}^{k} \epsilon_i \left(n - s_i \cdot (k-i)\right) - \sum_{i=1}^{k} s_i \end{aligned}$$

which is what we wanted to prove in this claim. □

We now turn to proving the main theorems.

Proof of Theorem 3. The number of phases that P_{BW} executes depends on the phase in which it runs out of space to allocate. Using the notation above, this happens when $M_i < s_i$, and it is not possible to allocate an object in Phase i. However, Claim 5.5 holds for any number of phases in which the bucket size is smaller or equal to n, even if there were no actual memory allocated during that phase. The only downside of using advanced phases with Claim 5.5, is that the bound becomes less accurate after the last phase when objects were allocated. In order to prove Theorem 3 we use the bound on the heap size, received in Claim 5.5, for phase $\sqrt{n}$. This phase is close to the final phase for the setting of an arithmetic collection of buckets.

By setting the variables in Claim 5.5, we get:

$$\begin{aligned} HS_{\sqrt{n}}(\mathit{SFL}_{all}, P_{BW}) &\geq M \cdot \sqrt{n} - M \cdot \sum_{i=1}^{\sqrt{n}-1} \frac{i}{n-i}(\sqrt{n} - i) \\ &\quad + \sum_{i=1}^{\sqrt{n}} \epsilon_i \left(n - i \cdot (\sqrt{n} - i)\right) - \sum_{i=1}^{\sqrt{n}} i. \end{aligned}$$

We can further bound the equation from above by replacing all of the denominators with $n - \sqrt{n} + 1$. Additionally, notice that $(n - j \cdot (\sqrt{n} - j))$ has a positive value for all $1 \leq j \leq \sqrt{n}$. Therefore, we can bound this equation from above by setting $\epsilon_j = 0$ (Remember that for all values of j, $0 \leq \epsilon_j \leq 1$).

$$\begin{aligned} HS_{\sqrt{n}}(\mathit{SFL}_{all}, P_{BW}) &\geq M\sqrt{n} \\ &\quad - M\frac{1}{n - \sqrt{n} + 1} \sum_{i=1}^{\sqrt{n}-1} i(\sqrt{n} - i) - \sum_{i=1}^{\sqrt{n}} i. \end{aligned}$$

Using the closed sum for squares:

$$\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6} \tag{5.2}$$

and some algebra we get:

$$\begin{aligned} HS_{\sqrt{n}}(\mathit{SFL}_{all}, P_{BW}) &\geq M\sqrt{n} - \tfrac{1}{6}M\sqrt{n}\left(\frac{n-1}{n - \sqrt{n} + 1}\right) \\ &\quad - \frac{\sqrt{n}(\sqrt{n}+1)}{2} \\ &\geq \tfrac{4}{5}M\sqrt{n} - \tfrac{1}{2}n - \tfrac{1}{2}\sqrt{n}. \end{aligned}$$

□

Proof of Theorem 4. The number of phases that P_{BW} runs depends on the phase in which it runs out of available space to allocate. Using the notation above, this happens when $M_i < s_i$, and it is not possible to allocate an object in Phase i. However, Claim 5.5 holds for any number of phases in which the bucket size is smaller or equal to n, even if there were no actual space allocated on that phase. The only downside of using advanced phases is that the results of Claim 5.5, is that the bound becomes less accurate after the last phase when objects were allocated. In order to prove Theorem 3 we use the bound on the heap size, received in Claim 5.5, for Phase $\log n$. This phase is close to the final phase for the setting of a logarithmic collection of buckets.

By setting the variables in Claim 5.5, we get:

$$\begin{aligned} HS_{\log n}(\mathit{SFL}_{log}, P_{BW}) &\geq M \log n \\ &\quad - M \sum_{i=1}^{\log n - 1} \frac{2^{i-1}}{n - 2^{i-1}}(\log n - i) \\ &\quad + \sum_{i=1}^{\log n} \epsilon_i \left(n - 2^{i-1}(\log n - i)\right) - \sum_{i=1}^{\log n} 2^{i-1}. \end{aligned}$$

Similarly to what we did in the proof of Theorem 3, we can replace all the denominators with $3n/4$, and notice that $(n - 2^{i-1} \cdot (\log n - i))$ has a positive value for all $1 \leq j \leq \log n$. Therefore, we can bound this equation from above by setting $\epsilon_j = 0$.

$$\begin{aligned} HS_{\log n}(\mathit{SFL}_{log}, P_{BW}) &\geq M \log n \\ &\quad - M \tfrac{1}{3n/4} \sum_{i=1}^{\log n - 1} 2^{i-1}(\log n - i) - \sum_{i=1}^{\log n} 2^{i-1}. \end{aligned}$$

Using the following equation:

$$\sum_{i=1}^{k-1} i2^i = 2^k(k-2) + 2 \tag{5.3}$$

and some algebra, we get what we wanted to prove.

$$\begin{aligned} HS_{\log n}(\mathit{SFL}_{log}, P_{BW}) &\geq M \log n - M \tfrac{4}{3n}(n - \log n - 1) \\ &\quad -n + 1 \\ &\geq M \cdot \log n - \tfrac{4}{3}M - n. \end{aligned}$$

□

5.3 Space overheads with compaction

In Section 5.2, we built P_{BW} that wasted many blocks by leaving a single object in each. With no compaction that program created a lot of fragmentation. Modern collectors employ partial compaction especially for such scenarios, and they clear sparse blocks by moving their elements to a different block. The evacuated blocks can then be used for further allocations. In this section we first discuss a simple upper bound on the heap usage by looking at a simple compaction strategy (which is similar to what is used in actual systems in practice). We then provide a program that makes the allocator

waste blocks by leaving enough allocated space on them so that the compaction budget will not allow effective defragmentation.

5.3.1 The Upper Bound

Our memory manager allocates in a (block-oriented) Segregated free list manner as before, but it also has a compacting strategy. We call this memory manager a *Compacting Segregated Free List* allocator and denote it by *CSFL*. We specify the compaction strategy, and later show that using this compaction strategy the following upper bound on the heap size holds.

Theorem 5. *Let c be the compaction threshold, k be the number of buckets, and n be the block size. The maximal heap size required for CSFL to execute any program P that never uses more than M words of allocated space simultaneously and never allocates an object larger than n satisfies:*

$$HS(CSFL, P) \leq M \cdot c + k \cdot n.$$

In order to prove this upper bound, we first present the compaction strategy of *CSFL*, show that it is consistent (always has enough budget for compaction), and finally prove the bound in Theorem 5.

The definition below assumes a *last block* in any bucket. Between all blocks that are allocated for the bucket, the last block is the one that was most recently allocated for this bucket. By the behavior of the segregated free list allocator, all objects on all other blocks were exhausted before the last one was allocated.

Definition 5.6 (*CSFL* Compaction Strategy)**.** *We denote by b, a block that contains objects of size i. CSFL will compact all objects within Block b to other block(s), if Block b holds the following constraints:*

1. *Block b is not the last block for objects of size i,*
2. *At most $1/c$ of the space in Block b is allocated,*
3. *There is enough free space in blocks containing objects of size i to contain all of the objects that are currently placed in Block b.*

Claim 5.7. *CSFL with the compaction strategy of Definition 5.6 has enough compaction budget to execute all required compaction.*

Proof. We show that enough space was allocated on this block alone to provide the budget for compacting it at the appropriate time in the execution. By the allocation strategy of the segregated free list, any block that is not the last block, was filled with objects at some point in the execution. Otherwise, the next block would not have been allocated. There were no compactions made on this block since it was taken from the pool, according to the algorithm (by which every compaction frees an entire block and returns it to the blocks pool). Since this block was taken from the blocks pool, the accumulated compaction quota due to allocations on this block is at least $1/c$ of the block size. (It can be larger if space on this block was reused.) At the execution point in which compaction is triggered on this block, at most $1/c$ of the block is allocated, therefore all this space can be moved using only budget in the quota that originated from allocations on this block. After all of the objects in this block are compacted away, the block is returned to the block pool and can be reused in the future for other bucket sizes. □

Proof of Theorem 5. According to Claim 5.7, all blocks (except maybe the last block) are at least $1/c$ full (otherwise, the block would have been cleared using compaction). Therefore, the total heap size required for these blocks is at most: $c \cdot M_1$, where M_1 is the total space in all blocks except the last ones in each bucket size. Therefore, the total heap size required is at most $c \cdot M + k \cdot n$, where k is the number of different possible sizes, and n is the block size. □

5.3.2 The Lower Bound

We now construct a program that creates a lot of fragmentation for the compacting segregated free list allocator. As before, we look at a *CSFL* that uses a large number of buckets and at a *CSFL* that uses a small number of buckets. We provide lower bounds for these two extreme cases, and the same techniques can be used to work with any specific bucket sizes employed in any practical system.

Denote by $\mathcal{P}(M, [s_1, \ldots, s_k])$ the set of programs that never allocate more than M words simultaneously and allocate only objects whose size is in the set $\{s_1, ..., s_n\}$. The following theorems assert the two lower bounds.

Theorem 6. *Let $CSFL_{all}$ be an allocator that implements the segregated free list method and keeps the compaction bound of $\frac{1}{c} \cdot S$ for $c \geq 4$. Let it's bucket object sizes contain be the complete assortment of: $s_1 = 1, s_2 = 2, s_3 = 3, s_4 = 4, ..., s_n = n$ for $n \geq 2$. Then there exists a program $P \in \mathcal{P}(M, [s_1, \ldots, s_n])$, such that the heap size required for allocator $CSFL_{all}$ to execute the allocations and deallocations of P, $HS(CSFL_{all}, P)$, satisfy:*

$$HS(CSFL_{all}, P) \geq \begin{cases} \frac{1}{6} \cdot M \cdot \sqrt{n} + \frac{1}{16} M & \text{if } c \geq 2\sqrt{n} \\ \frac{1}{12} \cdot M \cdot c + \frac{1}{16} M & \text{if } c \leq 2\sqrt{n} \end{cases}.$$

Theorem 7. *Let $CSFL_{log}$ be an allocator that implements the segregated free list method and keeps the compaction bound of $\frac{1}{c} \cdot S$. Let it's bucket object sizes contain a logarithmic assortment of bucket sizes: $s_1 = 1, s_2 = 2, s_3 = 4, s_4 = 8, ..., s_{\log n+1} = n$. Then there exists a program $P \in \mathcal{P}(M, [s_1, \ldots, s_{\log n+1}])$, such that the heap size required for the allocator $CSFL_{log}$ to execute all allocations and deallocations of P, $HS(CSFL_{log}, P)$, satisfy:*

$$HS(CSFL_{log}, P) \geq \begin{cases} \frac{1}{4} \cdot M \cdot \log n - 2M & \text{if } c \geq 2 \log n \\ \frac{1}{8} \cdot M \cdot c - 2M & \text{if } c \leq 2 \log n \end{cases}$$

We now present the program *Segregated-Waster*, denoted P_{SW}, that creates a large fragmentation for the compacting segregated free list allocator.

Algorithm 7 Program *Segregated-Waster* for the compacting segregated free list allocator

1: Compute d as a function of the input compaction threshold c.
2: **for** $i = 1$ to k do **do**
3: Allocate as many objects as possible of size s_i
4: In each block, deallocate as many objects as possible subject to leaving at least $1/d$ of the space allocated in this block.
5: **end for**

Algorithm 7 is similar to Algorithm 6 presented in Section 5.2 in its allocation steps, but differs in its deallocation steps. The difference is that Algorithm 7 keeps the allocated space remaining in each block larger than $1/d$ of the space allocated within this block. Leaving more allocated space in a block makes it difficult to compact away all of the objects in it. A block can be used for allocating objects of a different size only when all of the objects in it are moved away.

Below we investigate the joint run of *CSFL* and P_{SW} and provide claims that lead to the proof of the Theorems 6 and 7. Note the difference between the execution of *CSFL* against a program P (with compaction enabled), and the execution of *SFL* against it (with compaction disabled). The difference is the fact that in the compacting scenario space can be reused. After objects on a block are compacted away, this same block can be reused for objects of larger sizes. Therefore, it will be necessary to calculate the heap size depending on compaction as well as allocation. The

Compaction is the reason why in Algorithm 7, deletions leave $1/d$ of the space in a block, where d is chosen according to the value of c. This is done in order to make the compaction more costly, but it has another effect: less deletion leaves less space for future allocations. Below, we present some claims that lead to the proof of Theorems 6 and 7 presented above.

Claim 5.8. *Consider the execution of a CSFL allocator with the program P_{SW}. For any phase i in the execution, let M_i denote the space available for allocation in Phase i, and let s_i denote the object size allocated in Phase i. Let k denote the number of phases in the execution. The heap size required for the execution of P_{SW} with CSFL satisfies:*

$$HS_k(\mathit{CSFL}, P_{SW}) \geq \frac{1}{2}\sum_{i=1}^{k} M_i - \frac{1}{2}\sum_{i=1}^{k} s_i.$$

Proof. According to the definition of P_{SW}, the program run consists of phases. In each phase the space available for allocation is M_i. The program requests allocations of as many objects as it can of size s_i. The number of allocation requests is, therefore, $\left\lfloor \frac{M_i}{s_i} \right\rfloor$, and the total space allocated in Phase i is $\left\lfloor \frac{M_i}{s_i} \right\rfloor \cdot s_i$. The total space allocated from the beginning of the run until (including) Phase k is

$$S_k = \sum_{i=1}^{k} \left\lfloor \frac{M_i}{s_i} \right\rfloor \cdot s_i \geq \sum_{i=1}^{k} M_i - \sum_{i=1}^{k} s_i. \tag{5.4}$$

The total quota for compaction is $S_k \cdot \frac{1}{c}$.

In each phase, the program removes as many objects as it can, subject to leaving at least $1/d$ of the space already allocated in each block. In order to free a bucket for reuse, all the objects in this block must be moved to another block. Therefore, the total space reuse during the execution is at most d times the space compacted, which is at most $d \cdot S_k \cdot \frac{1}{c}$. Setting $d = \frac{c}{2}$ in P_{SW}, we get that the space reuse is at most $\frac{1}{2}S_k$. The total heap size necessary is at least the space allocated, S_k, minus the space reused, which is at most $\frac{1}{2}S_k$. Now using the Inequality 5.4 for S_k gets the desired bound. □

Claim 5.9. *Consider the execution of a CSFL allocator with the Program P_{SW}. Let k be the number of phases in the execution, let M_i be the total space size available for allocation in Phase i, let s_i be the size of objects allocated in Phase i, let n be the block size, and let d be the density threshold. Then in all phases $1 \leq i < k$ it holds that:*

$$M_{i+1} \geq M_i \left(1 - \frac{1}{d} - \frac{s_i}{n - s_i}\right).$$

Proof. The allocator allocates according to the segregated free list method. This means that objects of different sizes are allocated in different blocks, and are allocated in these blocks sequentially. Therefore, if the space that could be allocated at the beginning of Phase i is M_i, and the size of objects to allocate in Phase i is s_i, then the number of objects actually allocated in Phase i is $\left\lfloor \frac{M_i}{s_i} \right\rfloor$. The number of objects in each full block is $\left\lfloor \frac{n}{s_i} \right\rfloor$. The number of fully allocated blocks is, therefore, $\left\lfloor \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rfloor$. Additionally, there might be a last block that was only partially allocated. The allocated space inside each full block is $\left\lfloor \frac{n}{s_i} \right\rfloor \cdot s_i$.

The program P_{SW} removes as many objects as possible from each block, subject to leaving at least $1/d$ of the objects that were allocated in each block. This means that the amount of space that remains allocated and cannot be freed for use in future allocations from the full blocks is at most the number of full blocks used in this phase times the allocated space remaining in each block:

$$\left\lfloor \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rfloor \cdot \left(\left\lfloor \frac{n}{s_i} \right\rfloor \cdot s_i \cdot \frac{1}{d} + s_i \right).$$

The additional s_i in the above equation appears because $1/d$ of the block size might not contain a integral number of objects of size s_i. Therefore an extra object might be needed in the block.

Now it remains to investigate the space remaining in the last block used in Phase i. This last block might have not been filled completely as there were not enough allocatable words to fill it. The number of objects allocated in the last block is the total number of objects allocated, minus what was allocated in the other (full) blocks:

$$\left\lfloor \frac{M_i}{s_i} \right\rfloor - \left\lfloor \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rfloor \cdot \left\lfloor \frac{n}{s_i} \right\rfloor.$$

The space remaining in the last block is the number of objects allocated in that block, multiplied by the object size, and by $1/d$, plus one additional object (for the case that the numbers do not divide well). If there is indeed a last block that contains less objects than all other blocks, then the live space remaining in it is at most:

$$\left(\left\lfloor \frac{M_i}{s_i} \right\rfloor - \left\lfloor \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rfloor \cdot \left\lfloor \frac{n}{s_i} \right\rfloor \right) \cdot s_i \cdot \frac{1}{d} + s_i.$$

This remaining space in the last block can be bounded from above by:

$$\left(\left\lfloor \frac{M_i}{s_i} \right\rfloor - \left\lfloor \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rfloor \cdot \left\lfloor \frac{n}{s_i} \right\rfloor \right) \cdot \left(s_i \cdot \frac{1}{d} + s_i \right),$$

which we will use in what follows.

The total space that becomes available for allocation in the next phase is the amount of space that gets deleted in the blocks in which allocation occurred in this phase. This can be computed as the space that was available for allocation in this phase minus the space that was allocated but not deleted in this phase. Note that this computation provides a lower bound on the amount of space available for allocation in the next phase as compaction that the program executes may make the memory manager delete more objects by moving objects from sparse blocks to dense ones and triggering deletions of objects from previous phases. The space available for allocation in the next Phase $i+1$ is at least:

$$\begin{aligned} M_{i+1} &\geq M_i - \left\lfloor \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rfloor \cdot \left(\left\lfloor \frac{n}{s_i} \right\rfloor \cdot s_i \cdot \tfrac{1}{d} + s_i \right) \\ &\quad - \left(\left\lfloor \frac{M_i}{s_i} \right\rfloor - \left\lfloor \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rfloor \cdot \left\lfloor \frac{n}{s_i} \right\rfloor \right) \cdot \left(s_i \cdot \tfrac{1}{d} + s_i \right) \\ &\geq M_i - \left\lfloor \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rfloor \cdot \left(s_i - \left\lfloor \frac{n}{s_i} \right\rfloor \cdot s_i \right) - \left\lfloor \frac{M_i}{s_i} \right\rfloor \left(\tfrac{s_i}{d} + s_i \right). \end{aligned}$$

Bounding $\left\lfloor \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \right\rfloor$ from above by $\frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor}$, we get:

$$\begin{aligned} M_{i+1} &\geq M_i - \frac{\left\lfloor \frac{M_i}{s_i} \right\rfloor}{\left\lfloor \frac{n}{s_i} \right\rfloor} \cdot \left(s_i - \left\lfloor \frac{n}{s_i} \right\rfloor \cdot s_i \right) - \left\lfloor \frac{M_i}{s_i} \right\rfloor \left(\frac{s_i}{d} + s_i \right) \\ &\geq M_i - \left\lfloor \frac{M_i}{s_i} \right\rfloor \cdot \left(\frac{s_i}{\left\lfloor \frac{n}{s_i} \right\rfloor} + \frac{s_i}{d} \right). \end{aligned}$$

Bounding $\left\lfloor \frac{M_i}{s_i} \right\rfloor$ from above by $\frac{M_i}{s_i}$, and bounding $\left\lfloor \frac{n}{s_i} \right\rfloor$ from below by $\frac{n}{s_i} - 1$, we get:

$$M_{i+1} \geq M_i - \frac{M_i}{s_i} \cdot \left(\frac{s_i}{\frac{n}{s_i} - 1} + \frac{s_i}{d} \right) \geq M_i \left(1 - \frac{1}{d} - \frac{s_i}{n - s_i} \right).$$

□

Claim 5.10. *Consider the execution of a CSFL allocator with the program P_{SW}. Let M_i be the size of the space available for allocation in Phase i, let s_i be the objects size in Phase i, let n be the block size, and let d be the density threshold. In all phases $2 \leq i \leq k$ it holds that:*

$$M_i \geq M - M \cdot \left(\frac{i-1}{d} + \sum_{j=1}^{i-1} \frac{s_j}{n - s_j} \right).$$

Proof. By removing the recursion from the result of Claim 5.9, we get:

$$M_i \geq M \prod_{j=1}^{i-1} \left(1 - \frac{1}{d} - \frac{s_i}{n - s_i} \right).$$

By using Equation 5.1 we substitute the product operation with the sum operation:

$$M_i \geq M \left(1 - \sum_{j=1}^{i-1} \left(\frac{1}{d} + \frac{s_i}{n - s_i} \right) \right).$$

Which is what we wanted to prove. □

Claim 5.11. *Consider the execution of a CSFL allocator with the program P_{SW}. Let HS_k be the total heap size required for the allocations of CSFL until the end of phase k. The heap size can be bounded by:*

$$\begin{aligned} HS_k(\textit{CSFL}, P_{SW}) \quad &\geq \frac{1}{2}Mk - M \cdot \frac{k(k-1)}{4d} \\ &\quad - \frac{1}{2}M \sum_{i=1}^{k-1} \frac{(k-i-1)s_i}{n - s_{k-1}} - \frac{1}{2}\sum_{i=1}^{k} s_i. \end{aligned}$$

Proof. According to Claim 5.8, the total heap size until Phase k is:

$$HS_k(\textit{CSFL}, P_{SW}) \geq \frac{1}{2}\sum_{i=1}^{k} M_i - \frac{1}{2}\sum_{i=1}^{k} s_i.$$

Substituting the value of M_i with the result of Claim 5.10, we get:

$$\begin{aligned} &HS_k(\textit{CSFL}, P_{SW}) \geq \\ &\geq \frac{1}{2}\sum_{i=1}^{k} \left(M - M \left(\frac{i-1}{d} + \sum_{j=1}^{i-1} \frac{s_j}{n - s_j} \right) \right) - \frac{1}{2}\sum_{i=1}^{k} s_i \\ &\geq \frac{1}{2}Mk - M \cdot \frac{k(k-1)}{4d} - \frac{1}{2}M \sum_{i=1}^{k-1} \frac{(k-i-1)s_i}{n - s_i} - \frac{1}{2}\sum_{i=1}^{k} s_i. \end{aligned}$$

By substituting $-s_i$ in the denominator, with the maximal value of s_i, which is s_{k-1}, we get the claim. □

We now use the above claims to prove Theorems 6 and 7.

Proof of Theorem 6. The object sizes in this case are: $s_1 = 1, s_2 = 2, s_3 = 3, s_4 = 4, ..., s_n = n$. We set these sizes in the expression of Claim 5.11:

$$\begin{aligned} HS_k(\textit{CSFL}_{all}, P_{SW}) \geq \quad &\frac{1}{2}Mk - M \cdot \frac{k(k-1)}{4d} \\ &- \frac{1}{2}M \sum_{i=1}^{k-1} \frac{(k-i-1)i}{n-k+1} - \frac{1}{2}\sum_{i=1}^{k} i. \end{aligned}$$

Using Equation 5.2 and some algebra, we get:

$$\begin{aligned} HS_k(\textit{CSFL}_{all}, P_{SW}) \geq \tfrac{1}{2}Mk - M \cdot \frac{k(k-1)}{4d} \\ - M \cdot \frac{(k-1)(k-2)(k-3)}{6(n-k+1)} - \frac{k(k+1)}{4}. \end{aligned}$$

The above bound is correct for any phase k, $1 \leq k \leq \sqrt{n}$, but the quality of the bound is best at where the computation actually stops. We divide into two possible cases. The first case is when $d \geq \sqrt{n}$. In this case, we choose $k = \frac{\sqrt{n}}{2}$ and get:

$$\begin{aligned} HS_{\frac{\sqrt{n}}{2}}(\textit{CSFL}_{all}, P_{SW}) \geq \tfrac{1}{4}M\sqrt{n} - M \cdot \frac{\sqrt{n}(\sqrt{n}-2)}{16d} \\ - M \frac{(\sqrt{n}-2)(\sqrt{n}-4)(\sqrt{n}-6)}{48(n - \frac{\sqrt{n}}{2} + 1)} - \frac{\sqrt{n}(\sqrt{n}+2)}{16}. \end{aligned}$$

Since we assume that $d \geq \sqrt{n}$, we can bound $\frac{\sqrt{n}}{d}$ from above by 1. Note also that for $n \geq 2$

$$\frac{(\sqrt{n}-2)(\sqrt{n}-4)(\sqrt{n}-6)}{(n - \frac{\sqrt{n}}{2} + 1)} \leq \sqrt{n} - 6.$$

Therefore,

$$\begin{aligned} HS_{\frac{\sqrt{n}}{2}}(\textit{CSFL}_{all}, P_{SW}) \geq \quad &\frac{1}{4}M\sqrt{n} - M\frac{\sqrt{n}-2}{16} \\ &- \frac{1}{48}M(\sqrt{n}-6) - \frac{n+2\sqrt{n}}{16}. \end{aligned}$$

According to our definition $M \geq n$. Therefore, we can replace the last $n + 2\sqrt{n}$ with $3M$, and get that:

$$HS_{\frac{\sqrt{n}}{2}}(\textit{CSFL}_{all}, P_{SW}) \geq \frac{1}{6}M\sqrt{n} + \frac{1}{16}M.$$

The other case is when $d \leq \sqrt{n}$. In this case, we choose $k = \frac{d}{2}$. Therefore:

$$\begin{aligned} &HS_{\frac{d}{2}}(\textit{CSFL}_{all}, P_{SW}) \geq \\ &\tfrac{1}{4}Md - M \cdot \frac{(d-2)}{16} - M\frac{(d-2)(d-4)(d-6)}{48(n - \frac{d}{2} + 1)} - \frac{d(d+2)}{16}. \end{aligned}$$

Since $\sqrt{n} \geq d$, we can replace n with d^2. We also use the fact that for $d \geq 2$,

$$\frac{(d-2)(d-4)(d-6)}{(d^2 - \frac{d}{2} + 1)} \leq d - 6.$$

Therefore,

$$HS_{\frac{d}{2}}(\textit{CSFL}_{all}, P_{SW}) \geq \frac{1}{4}Md - M \cdot \frac{d-2}{16} - M\frac{d-6}{48} - \frac{d^2+2d}{16}.$$

Since $M \geq n \geq d^2$, we can replace the last $d^2 + 2d$ with $3M$, and get that:

$$HS_{\frac{d}{2}}(\textit{CSFL}_{all}, P_{SW}) \geq \frac{1}{6}Md + \frac{1}{16}M$$

setting $d = c/2$ the claim holds. □

Proof of Theorem 7. The object sizes in this case are: $s_1 = 1, s_2 = 2, s_3 = 4, s_4 = 8, ..., s_n = 2^n$. Setting the object sizes in the expression of Claim 5.11 we get:

$$\begin{aligned} HS_k(\mathit{CSFL}_{log}, P_{SW}) &\geq \tfrac{1}{2}Mk - M \cdot \frac{k(k-1)}{4d} \\ &\quad -\tfrac{1}{2}M \sum_{i=1}^{k-1} \frac{(k-i-1)2^i}{n-2^{k-1}} - \frac{1}{2}\sum_{i=1}^{k} 2^i. \end{aligned}$$

Using equation 5.3 and some algebra, we get:

$$\begin{aligned} HS_k(\mathit{CSFL}_{log}, P_{SW}) &\geq \tfrac{1}{2}Mk - M \cdot \frac{k(k-1)}{4d} \\ &\quad -M \cdot \frac{2^{k-1}-k}{n-2^{k-1}} - \tfrac{1}{2}(2^{k+1}-1). \end{aligned}$$

We split the analysis into two possible cases. First, assume that $d \leq \log(n)$. In this case we look at the phase $k = \log n$. In this case, we get:

$$\begin{aligned} HS_{\log n}(\mathit{CSFL}_{log}, P_{SW}) &\geq \tfrac{1}{2}M\log n - M \cdot \frac{\log n(\log n - 1)}{4d} \\ &\quad -M \cdot \frac{\frac{n}{2} - \log n}{n - \frac{n}{2}} - \tfrac{1}{2}(2n-1) \\ &\geq \tfrac{1}{2}M\log n - M \cdot \frac{\log n(\log n - 1)}{4d} \\ &\quad -M + M \cdot \frac{2\log n}{n} - n + \tfrac{1}{2}. \end{aligned}$$

Since $d \geq \log n$, we can bound $\frac{\log n}{d}$ from above by 1. Also, we use the fact that $M \geq n \geq 1$ to get:

$$\begin{aligned} HS_{\log n}(\mathit{CSFL}_{log}, P_{SW}) &\geq \tfrac{1}{2}M\log n - M \cdot \frac{(\log n - 1)}{4} \\ &\quad -M + M \cdot \frac{2\log n}{n} - n + \tfrac{1}{2} \\ &\geq \tfrac{1}{4}M\log n - 2M. \end{aligned}$$

The other case is that $d \leq \log n$. In this case, we choose $k = d$ and obtain:

$$\begin{aligned} HS_d(\mathit{CSFL}_{log}, P_{SW}) &\geq \tfrac{1}{2}Md - M \cdot \frac{d(d-1)}{4d} \\ &\quad -M \cdot \frac{2^{d-1}-d}{n-2^{d-1}} - \tfrac{1}{2}(2^{d+1}-1). \end{aligned}$$

We replace n with 2^d, and use the fact that $M \geq n \geq 2^d$ to get:

$$\begin{aligned} HS_d(\mathit{CSFL}_{log}, P_{SW}) &\geq \tfrac{1}{2}Md - M\frac{(d-1)}{4} \\ &\quad -M \cdot \frac{2^{d-1}-d}{2^d - 2^{d-1}} - 2^d + \tfrac{1}{2} \\ &\geq \tfrac{1}{4}Md - 2M \end{aligned}$$

and again, setting $d = c/2$, the claim holds. □

6. Conclusion

In this work we studied the effectiveness of partial compaction for reducing the space overhead of dynamic memory allocation. We developed techniques for showing lower bounds on how much space must be used when the amount of compaction is limited by a given budget. It was shown that partial compaction can reduce fragmentation, but up to a limit, determined by the compaction budget. We also studied the effectiveness of partial compaction for a specific common allocator: the segregated free list allocator. Tighter bounds have been shown based on the specific behavior of this allocator.

This work extends our understanding of the theoretical foundation of memory management, specifically for compaction. We hope future work can build on our techniques and provide even tighter bounds to further improve our understanding of the effectiveness of partial compaction for modern systems.

References

[1] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. An efficient parallel heap compaction algorithm. In *Proceedings of the Nineteenth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 39(10), pages 224–236, Vancouver, Canada, October 2004.

[2] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices 38(1), pages 285–298, New Orleans, LA, USA, January 2003.

[3] Ori Ben-Yitzhak, Irit Goft, Elliot Kolodner, Kean Kuiper, and Victor Leikehman. An algorithm for parallel incremental compaction. In Hans-J. Boehm and David Detlefs, editors, *Proceedings of the Third International Symposium on Memory Management (June, 2002)*, ACM SIGPLAN Notices 38(2 supplement), pages 100–105, Berlin, Germany, February 2003.

[4] Hans-Juergen Boehm. Bounding space usage of conservative garbage collectors. In POPL 2002 [14].

[5] Hans-Juergen Boehm. The space cost of lazy reference counting. In *Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices 39(1), pages 210–219, Venice, Italy, January 2004.

[6] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[7] Cliff Click, Gil Tene, and Michael Wolf. The Pauseless GC algorithm. In Michael Hind and Jan Vitek, editors, *Proceedings of the First ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 46–56, Chicago, IL, USA, June 2005.

[8] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Erez Petrank, Igor Yanover, and Yossi Levanoni. Implementing an on-the-fly garbage collector for Java. In Craig Chambers and Antony L. Hosking, editors, *Proceedings of the Second International Symposium on Memory Management*, ACM SIGPLAN Notices 36(1), pages 155–166, Minneapolis, MN, October 2000.

[9] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[10] Haim Kermany and Erez Petrank. The Compressor: Concurrent, incremental and parallel compaction. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 41(6), pages 354–363, Ottawa, Canada, June 2006.

[11] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In POPL 2002 [14], pages 101–112.

[12] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgard. STOPLESS: A real-time garbage collector for multiprocessors. In Greg Morrisett and Mooly Sagiv, editors, *Proceedings of the Sixth International Symposium on Memory Management*, pages 159–172, Montréal, Canada, October 2007. ACM Press.

[13] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 43(6), pages 33–44, Tucson, AZ, USA, June 2008.

[14] *Conference Record of the Twenty-ninth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices 37(1), Portland, OR, USA, January 2002.

[15] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM*, 18(3):416–423, July 1971.

[16] J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM*, 21(3):419–499, July 1974.