Limitations of Partial Compaction: Towards Practical Bounds [*]

Nachshon Cohen
Technion
nachshonc@gmail.com

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

Abstract

Compaction of a managed heap is considered a costly operation, and is avoided as much as possible in commercial runtimes. Instead, partial compaction is often used to defragment parts of the heap and avoid space blow up. Previous study of compaction limitation provided some initial asymptotic bounds but no implications for practical systems. In this work, we extend the theory to obtain better bounds and make them strong enough to become meaningful for modern systems.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Dynamic Storage Management; D.3.4 [*Processors*]: Memory management (garbage collection); D.4.2 [*Storage Management*]: Allocation/deallocation strategies; D.1.5 [*Programming Technique*]: Object Oriented Programming

General Terms Algorithms, Theory, Languages.

Keywords Memory management, compaction, fragmentation, theory, lower bounds.

1. Introduction

The study of the theoretical foundations for memory management is mostly lacking. Little is known about the limitations of various memory management functionalities, and in particular on the space consumption of various memory management methods. Previous work consists of Robson's classical results on fragmentation when no compaction is employed [14, 15], a result on the hardness of achieving cache consciousness [11], and work on the effectiveness of conservative garbage collection and lazy reference counting [5, 6]. A recent new work by Bendersky et al. [4] attempted to bound the overhead mitigation that can be achieved by partial compaction.

Memory managers typically suffer from fragmentation. Allocation and de-allocation of objects in the heap create "holes" between objects that may be too small for future allocation and thus create a waste of available heap space. Compaction can eliminate this problem, but compaction algorithms are notoriously costly and are thus not frequently used [1, 9, 10]. Instead, memory managers today either use compaction seldom, or employ partial compaction, where only a (small) fraction of the heap objects are compacted to make space for further allocation [2, 3, 7, 8, 12].

The work in [4] studies limitations of partial compaction. But in spite of [4] being novel and opening a new direction for bounding the effectiveness of partial compaction, their results are only significant for huge heaps and objects. When used with realistic setting of system parameters today, their lower bounds become meaningless. For example, suppose a program uses a live heap space of 256MB and allocates objects of size at most 1MB. For such a program, even if the memory manager is limited and can only compact 1% of the allocated objects, the results in [4] would only imply that the heap must be of size at least $256MB$, which is obvious and not very useful.

In this work we extend the lower bounds on partial compaction to make them meaningful for practical systems. To this end, we propose a new "bad" program, that makes memory managers fail in preserving low space overheads. We then improve the mathematical analysis of the interaction between the bad program and the memory manager to obtain much better bounds. For example, using the parameters mentioned in the previous paragraph, our lower bound implies that a heap of size $896MB$ must be used, i.e., a space overhead of $3.5x$.

In general, the more objects we let the memory manager move, the lower the space overhead that it may suffer. To put a bound on the amount of compaction work that is undertaken by the memory manager, we use the model set in [4] and bound the fraction of allocated space that may be compacted by a constant fraction $1/c$. This means that at any point in the execution, if a space of s words has been allocated so far, then the amount of total compaction that is allowed up to this point in the execution is s/c words, where c is the compaction bound.

The results we obtain involve some non-trivial mathematical arguments and the obtained lower bound is presented in a complex formula that is not easy to digest. The general lower bound is stated in Theorem 1 (on Page 3). But in order to grasp the improvement over the previously known results, we have depicted in Figure 1 (on Page 3) the space overhead factor for the parameters mentioned above (which we consider realistic). Namely, for a program that uses live space of $M = 256MB$ and whose largest allocated object is of size at most $n = 1MB$, we drew for different values of c the required heap size as a factor of the $256MB$ live space. Note that if we were willing to execute a full compaction after each de-allocation, then the overhead factor would have been 1. We could have used a heap size of $256MB$ and serve all allocation and de-allocation requests. But with limited (partial) compaction, our results show that this is not possible. In Figure 1 we drew our lower bound as well as the lower bound obtained from [4] for these parameters. In fact, throughout the range of $c = 10, \ldots, 100$, the lower bound from [4] gives nothing but the trivial lower bound overhead factor of 1, meaning that $256MB$ are required to serve the program. In contrast, our new techniques show that the space overhead must be at least $2x$, i.e., $512MB$ when 10% of the allocated space can be

[*] This work was supported by the Israeli Science Foundation grant No. 283/10

compacted. And when the compaction is limited to 1% of the allocated space, then an overhead of $3.5x$ is required for guaranteeing memory management services for all programs.

In general, the result described in this work is theoretical, and does not provide a new system or algorithm. Instead, it describes a limitation that memory managers can never achieve. As such, it does serve a practical need, by letting practitioners know what they cannot aspire to, and should not spend efforts in trying to achieve. The lower bounds we provide are for a worst-case scenario and they do not rule out achieving a better behavior on a suite of benchmarks. But providing a better guaranteed bound on fragmentation (as required for critical systems such as real-time systems) is not possible. Note that this bound holds for manual memory managers as well as automatic one, even when applying sophisticated methods like copying collection, mark-compact, Lang-Dupont, MC^2, etc. [9].

Finally, we also make a slight improvement over the state-of-the-art related upper bound. The upper bound is shown by presenting a memory manager that keeps fragmentation low against all possible programs. The new upper bound slightly improves over the result of [4], and it does so by providing a better memory manager and a better analysis for its worst space overhead. This slight improvement is depicted in Figure 3 (on Page 4), where we depict the new upper bound and compare it to the previous upper bound of [4]. The upper bound theorem is stated rigorously as Theorem 2 (on Page 3). Note that the proposed memory manager is not meant to be a practical efficient memory manager that can be used in real systems, but it demonstrates the ability to deal with worst-case fragmentation.

To achieve the bounds presented in this paper, we stand on the shoulders of prior work, and in particular our techniques build on and extend the techniques proposed in [15] and [4].

Organization. In Section 2 we provide some preliminaries, explain the execution of a memory manager, and state our results. In Section 3 we provide an overview over the lower bound and its proof. In Section 4 we provide the actual proof of the the lower bound, and we conclude in Section 5. A full version of this work is available [13], and contains the complete proof of the lower bound and the upper bound.

2. Problem Description and Statement of Results

2.1 Framework

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

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

In this work, we question the ability of a memory manager to handle programs within a given heap size. But if a program allocates continuously and never de-allocates any memory, then the heap size required is trivially unbounded. So to let the question make sense, we assume a bound on the space that the program may use simultaneously. This bound is denoted by M.

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

As explained in the introduction, if the heap is compacted after every de-allocation, fragmentation never occurs. However, frequent compaction is costly and so memory managers either perform a full compaction infrequently, or just move a small fraction of the objects occasionally. In this paper we adopt the definition of [4], and consider memory managers that limit their compaction efforts with a predetermined fraction of the allocated space. For a constant $c > 1$, a memory manager is c-partial memory manager if it compacts at most $\frac{1}{c}$ of the total space allocated by the program. We denote the set of c-partial memory manager by $\mathscr{A}(c)$.

Given a program P and a memory manager A, the execution of P where A serve as its memory manager is well defined. The total heap size that A uses in this case is denoted by $HS(A,P)$

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

Our model is phrased above as one that lets the program know the address of each allocated object. This knowledge helps the program create the fragmented memory. We remark that it is enough to let the program know the allocator's algorithm and when GC is invoked (namely, when de-allocation actually happens) in order to obtain this information and create the large fragmentation.

2.2 Previous work

For programs that allocate only objects with size that is a power of 2, and $M|n$, and for all memory managers that do not use compaction, Robson [14, 15] proved lower and upper bounds that match. For his lower bound, he presented a "bad" program $P_o \in \mathscr{P}_2(M,n)$ that makes any memory manager (that does not use compaction) need a large heap. Specifically, [1]

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

For an upper bound, Robson presented an allocator A_o that satisfies the allocation requests of any program in $\mathscr{P}_2(M,n)$ using a heap size of

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

For programs that may allocate objects of arbitrary size (and not only powers of 2), one may round each allocation to the closest higher power of two. This rounding may (at the most) double the size of each object, which means that if the program is allowed to allocate $2M$ words simultaneously, then we obtain a doubled upper bound of $2(M(1/2\cdot\log(n)+1)-n+1)$.

When some compaction is allowed (but not an unlimited compaction effort), much less is known. For the upper bound, Bendersky and Petrank [4] have shown a simple compacting collector $A_c \in \mathscr{A}(c)$, that uses a heap space of at most

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

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

[1] Here, and throughout the paper, all logarithms are of base 2.

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

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

2.3 This work

Our main contribution is a new lower bound on the ability of a memory manager to keep the heap de-fragmented. While the lower bound of [4] is important for modeling the problem, providing some tools for solving it, and an asymptotical lower bound, their bound is meaningful only for huge objects and heaps. In particular, it provides a lower bound that is higher than the obvious M only for $M > n \geq 16TB$. In this work we extend the theory enough to obtain meaningful results for practical values of M and n.

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

$$\min_{A\in\mathscr{A}(c)} HS(A, P_F) \geq M \cdot h \qquad (1)$$

where h is set to:

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

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

Since h is given in a complicated formula, the implications of $HS(A, P_F) \geq M \cdot h$ are not very intuitive. Therefore, we chose some realistic parameters to check how this bound behaves in practice. We chose M, the size of the allocated live space to be $256MB$, and n, the size of the largest allocatable object to be $1MB$. With these parameters fixed and with the parameter γ set to the value that maximizes the bound, we drew a graph of h as a function of the compaction quota bound c. This graph appears in Figure 1. The x-axis has c varying between 10 to 100. Setting $c = 10$ means that we have enough budget to move 10% of the allocated space, whereas setting $c = 100$ means that we have enough budget to move 1% of the allocated space. For these c's, the y-axis represents the obtained lower bound as a multiplier of M. For example, when compaction of 2% of all allocated space is allowed ($c = 50$), any memory manager will need to use a heap size of at least $3.15 \cdot M$. Even with 10% of the allocated space being compacted, a heap size of $2 \cdot M = 512MB$ is unavoidable. For these practical parameters, previous results in [4, 14] do not provide any bound, except for the obvious one, that the heap must be at least of size M.

We also depicted the lower bound as a function of a varying maximum object size n. We fixed the compaction budget to $c = 100$, and the total size of live objects to $M = 256n$. The rational for the last parameter setting is that it is uncommon for a single object to create a significant part of the heap (larger than half a percent). Setting M to a larger value does not change the bound. We let the size of the largest object n vary between $1KB$ and $1GB$, and for these n values, the y-axis represents the obtained lower bound as a multiplier of M. The graph is depicted in Figure 2.

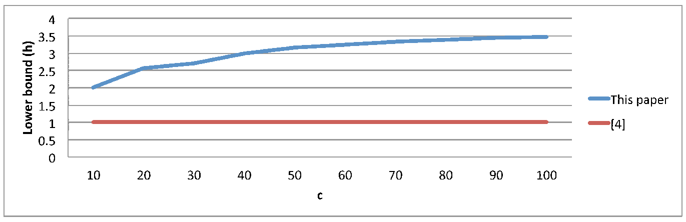

Figure 1. Lower bound on the waste factor h for realistic parameters ($M = 256MB$ and $n = 1MB$) as a function of c

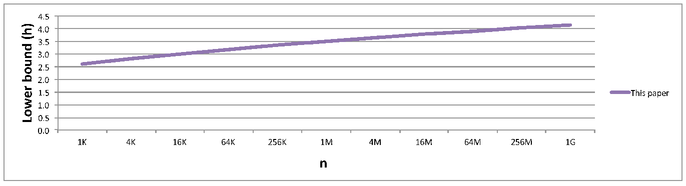

Figure 2. Lower bound on the waste factor h as a function of n (c=100, M=256n)

We could also depict the lower bound as a function of M, where n and c remain fixed. However, in a practical setting, the size of the largest object is much smaller than the total live space (i.e., n/M is small). Hence the lower bound as a function of M is very close to a constant function and it does not provide an additional interesting information.

We also consider the upper bound on the size of heap required. In [4] an upper bound of the form $(c+1)M$ was presented. However, this upper bound may become non-interesting when Robson's upper bound is stronger, meaning that the same heap size may be obtained without moving objects at all. This happens when $c > \log n + 1$. As partial compactors often use a large c to limit the fraction $1/c$ of moved objects, such a scenario seems plausible. We provide some improvement to Robson's algorithm when little compaction is allowed and obtain better upper bound as follows.

Theorem 2. *For any $c > \frac{1}{2}\log n$, there exists a c-partial memory manager $A \in \mathscr{A}(c)$, which satisfies allocation requests of any program $P \in \mathscr{P}$ with heap size at most*

$$\max_{P\in\mathscr{P}(M,n)} HS(A_C, P) \leq 2M \cdot \sum_{i=0}^{\log n} \max\left(a_i, \frac{1}{4-2/c}\right) + 2n\log n$$

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

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

As the formulas in this theorem are also not easy to grasp, we also drew a graph comparing previously known bounds with the new result. It can be seen in Figure 3 that for c's between 20 and 100 we get improvement with the largest improvement being 15% at $c = 20$. We consider this result minor and the lower bound the major result in this paper. The proof of the upper bound appears in the full paper [13].

3. Overview and Intuitions

In this section we review the proof of the lower bound. The main tool in this proof is the presentation of a "bad" program that cause large fragmentation. Our bad program will always allocate objects

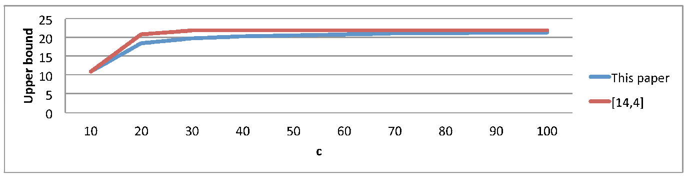

Figure 3. Upper bound on the waste factor for realistic parameters ($M = 256MB$ and $n = 1MB$) as a function of c

of size which is an exponent of 2, and to simplify the discussion (in this overview), we assume that the memory manager is restricted to using only aligned allocation. This means that an object of size 2^i is placed in an address that is dividable by 2^i.

The bad program will work in steps, where in each step of the execution, the bad program will allocate only objects of size 2^i, for some i that it will determine. Consider such a step and consider a memory region that starts at an address dividable by 2^i and spans 2^i words. Denote such a memory region a chunk. If a chunk is fully populated by objects, there is no fragmentation in this chunk. On the other extreme, if a chunk is empty, a new object of size 2^i can be placed in this "hole", creating a fully populated chunk. Fragmentation actually occurs (w.r.t. a chunk) when a chunk is sparsely populated. In this case, the utilization of this chunk is low, yet it can not be used for placing a new object. In the case that no compaction is allowed, a "bad" program should attempt to leave a small object in each such chunk, by de-allocating as many objects as possible, leaving one object in each chunk.

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

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

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

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

3.1 Improvements over prior work [4]

In this subsection we review the main improvements of this work over [4]. These improvements enabled the achievement of a better lower bound, which is meaningful with realistic parameter settings. We mention the three major improvements in this overview.

The first improvement follows from noting that in the first steps, the size of allocated objects are large compared to the chunk sizes (which are small in the first steps). Therefor, in these first steps, it is useful to run a program that is very similar to Robson's bad program [14]. Robson's program is designed for memory managers that do not compact objects, but when objects are large compared to the chunks, it is not beneficial to the memory manager to do any compaction, and a reduction theorem is developed to show that this program (or actually a similar program) creates fragmentation even where compaction is allowed. Furthermore, these first steps nicely integrate with the general algorithm that runs in the rest of the steps.

The second improvement consists of a small twist in the algorithm that creates a more regimented behavior during the execution, which then allows analyzing the execution and obtaining the improved bound. Recall that the bad program attempts to de-allocate as much space as possible in each step so that it can allocate as many objects as possible in the next step. It turns out that this behavior can create scenarios that are hard to analyze. This happens when we allocate a lot of objects in one step, and then very few in the following steps. The second improvement that helps us obtain higher fragmentation is in bounding the amount of memory that is allocated per step. This, perhaps, does not use all the space that can be used in one step, but it guarantees a sufficient amount of allocation in all steps. Allocating a fixed amount of memory at each step allows a stronger analysis of the program behavior and results in a better bound. Indeed the stronger analysis of the more regimented execution is another improvement of this new work.

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

4. Lower bound: creating fragmentation

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

Let us start by explaining the ideas behind the construction of P_F. The program P_F works in two stages. The second stage is probably the major contributor to the fragmentation achieved, but the first stage is also necessary to obtain our results. The first stage is an adaptation of Robson's malicious program [14] that attempts to fragment the memory as much as possible, when working against

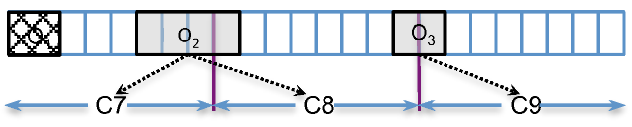

Figure 4. association of objects and half objects with chunks

a memory manager that cannot move objects. We will discuss in Section 4.2 how this algorithm behaves against a memory manager that can move objects and show that it buys some fragmentation in this case as well. After running for 2γ steps, a second stage starts, which behaves differently. The second stage works in steps $i = 2\gamma, \ldots, \log(n) - 2$ and in each step it only requests allocations of objects of size 2^{i+2} words. At each such step of the execution, we consider a partition of the heap space into aligned chunks of size 2^i words. This means, for example, that each allocated object either consumes four full consecutive chunks if its allocation is aligned, or it consumes at least three full consecutive chunks.

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

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

The association of objects with chunks is chosen carefully to establish the bound. Note that chunk sizes dynamically change as we move from step to step. On a step change, each pair of adjacent chunks become a single joint chunk. So, association of chunks with objects changes between steps. The association is also updated during the execution, as objects get allocated and de-allocated. The program actively maintains the set of objects associated with each chunk, and also uses this set in the second stage to determine which objects to de-allocate. Association of an object with a chunk is only removed when P_F de-allocates the object. It is not removed when an object is compacted. The fact that we do not "move" association during compaction and attempt to claim that an additional chunk must be used for the move follows from a analysis strategy: when an object is moved, usually it is possible to put it in used chunks, that are not fully occupied. Since the analysis can gain nothing from checking the object's new location, the bad program will just de-allocate any object that's being moved immediately, again, not attempting to consider the location to which it was moved into, but instead, de-allocating it and using its space for future allocation. Note that the chunk that it did occupy will remain part of the heap forever, so associating it with the old chunk makes sense.

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

From the memory manager point of view, a chunk that contains a small number of allocated words is a good candidate for compaction and reuse. Compaction allows reuse of a chunk's space for more allocations. As the program P_F controls which objects get de-allocated, P_F will attempt to make sure that each chunk has enough associated objects to make it non-beneficial for the memory manager to clear a chunk. The *density* of a chunk is the total size of objects that are associated with it, divided by the chunk size. We define a *density parameter* $2^{-\gamma}$ so that P_F attempts to keep the density of a chunk at least $2^{-\gamma}$. This means that the program P_F will never choose to de-allocate an object if the de-allocation makes a chunk too sparse, in the sense that its density goes below $2^{-\gamma}$. This density will be chosen to be larger than $1/c$ to make the compaction of objects from such a chunk not beneficial.

Loosely speaking, according to the compaction budget rules, the memory manager gains an extra compaction budget of $\frac{1}{c}|o|$ when it allocates o. However, if it needs to move $2^{-\gamma} \cdot |o|$ words to make space for this allocation, then its overall compaction budget decreases. (Recall that $2^{-\gamma} > \frac{1}{c}$). An example of density threshold and association set is depicted in Figure 4. Let the density threshold $2^{-\gamma}$ be $1/4$, which consists of 2 words per chunk of size 8. Half of O_2 is associated with Chunk $C7$, the other half is associated with Chunk $C8$, and the object O_3 is associated with Chunk $C9$ only. These objects suffice to make the density of each chunk at least $1/4$. The program can free the object O_1 since a density of $1/4$ is preserved even without it.

The whole issue of maintaining a high enough density is not relevant for the first steps in the computation. In these steps chunks are small enough so that when even one word is allocated on the chunk, a density of $2^{-\gamma}$ is achieved. Therefore, we can simply adopt Robson's "bad" program [14] with a technical variation so that it can deal with the memory manager's compaction activity. Robson's program works well to blow the heap up for memory managers that do not compact the heap. In our scenario, where the chunks are small and so the density is always high, compaction is not very useful, and so Robson's original program can work for us too. However, while compaction is not beneficial to the memory manager, compaction may still occur and our bad program must deal with it. We build a program P_F that will be "similar" to Robson's program in the sense that it will keep a similar heap shape, it will make very similar decisions on which objects to de-allocate and it will allocate the same amount of space in each step. We will then show a reduction saying that if there exists a memory manager M that can maintain low fragmentation while serving P_F, then it is possible to create another memory manager M', that does not move objects, and maintains low fragmentation against Robson's program. Since no memory manager can keep low fragmentation against Robson's program, we get the lower bound we need.

In order to make P_F work similarly to Robson's program, we need to handle compaction. When objects are moved, several differences are created that might influence the execution. First, space gets occupied where an object is moved, so new objects can no longer be allocated there. Second, vacancy is created in the old space from which an object was moved, so objects might be allocated there. And finally, the different shape of allocated objects may change the de-allocation decisions of the bad program. To handle the first difference, P_F simply de-allocates each moved object immediately after it gets moved. This makes sure that new objects can be allocated as before. To handle the other two problems we introduce *ghost objects*, which are not really in the heap, but are used by P_F to remember where objects existed so that Robson's program behavior can still be imitated.

The program P_F maintains a list of ghost objects. These are objects that have been relocated by the memory manager along with their original location. For all of its de-allocation considerations, P_F treats ghost objects as if they still reside at their original location. In fact, each ghost object continue to exist until the de-allocation procedure (of Robson's) determines that it should be de-allocated. Of-course, these objects have been de-allocated when they became ghosts, so no actual de-allocation is required by the memory manager, but at that point, they are removed from the list of ghost objects and are not considered further by the de-allocation procedure.

Note that memory space on which ghost objects reside may receive allocations of new objects by the memory manager, who is not aware of the ghost objects. This is fine. The de-allocation procedure can view both objects as residing on the same location while making its decisions. This seeming collision is later resolved in the reduction theorem, by noting a property of the de-allocation procedure. The de-allocation procedure only cares about location of objects moduli 2^i. Therefore, one can think of the ghost objects as existing in a separate (far away) space at the same address moduli 2^i. This additional space will not be counted as part of the heap size, but it will allow the program P_F and a real memory manager to work consistently together. Details follow.

Definition 4.1. [A ghost object] *We call an object that was compacted by the memory manager during the execution and immediately de-allocated by the program P_F a* ghost *object. In the first stage of the algorithm, such objects are considered by P_F as still residing as ghosts in the original location where they were allocated. They do not impact the behavior of the memory manager, which can allocate objects on a space consumed by ghosts. When ghost objects are de-allocated by the program, they disappear and are no longer considered by P_F in subsequent steps.*

At each step i of the first stage, $i = 0, 1, \dots, \gamma$, we start by considering a partition $\mathscr{D}(i)$ of the heap into all aligned chunks of size 2^i. (Aligned here means that they start on an address that is divisible by 2^i.) The main decision that is taken at each step is which objects should be de-allocated (by the malicious program). To this end, Robson picks an offset f_i and examines the word at offset f_i (from the beginning of the chunk) for all chunks. De-allocation then is executed for all objects that do not intersect the f_i word of a chunk. Note that all objects that will be allocated thereafter are all of size at least 2^i, and therefore, two adjacent chunks that have their f_i offset word occupied, will never be able to hold a new object between them.

Definition 4.2. [an f-occupying object with respect to step i] *An object is f-occupying with respect to step i if it occupies a word at address $k \cdot 2^i + f$ for some $k \in \mathbb{N}$.*

As a new step kicks in, the chunks sizes get doubled to 2^{i+1}, where each chunk contains two adjacent chunks of the previous step i. We would like to pick a new offset f_{i+1} for the larger chunks of size 2^{i+1}. The new offset will be either the old offset on the left 2^i-sized sub-chunk or the old offset on the right 2^i-sized sub-chunk. Robson chooses the new offset to be the one that maximizes the wasted space. In a way, Robson attempts to keep the smallest objects that will still occupy words at the f_{i+1} offset. So if one of these two offsets allows capturing more space with smaller objects, this becomes the new offset f_{i+1}. To formalize this, Robson chooses f_{i+1} to be either f_i or $f_i + 2^i$, according to which maximizes

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

It is not necessary to understand the details of Robson's analysis, as we adopt it without repeating it for the first stage of our program. However, to be able to link Robson's program to P_F's first stage, we also work with ghost objects. In the summation above, Robson naturally considers all objects in the heap that have been allocated but not de-allocated yet, i.e., the set of live objects. In our modified algorithm, we also consider ghost objects. Namely, we sum over all live objects and also over all objects that were compacted from their original location and thereafter de-allocated by the program. The ghost objects are considered to reside at the location they were allocated (and are not considered at the location to which they were compacted into, as they were already deleted from the heap.)

After running Robson's program, P_F runs extra $\gamma - 1$ null steps in which it does not allocate anything. This is done just for making all objects in the heap be of size at most $2^{-\gamma}$ of a chunk size. As will be shown in the analysis, this helps ensuring that a lot of space can be de-allocated by P_F even while maintaining a density of $2^{-\gamma}$. With much space de-allocated, P_F gains ammunition for allocations in the steps of the second stage. Recall that P_F is limited and cannot allocate more than M words simultaneously, so it must de-allocate enough space before it can allocate again.

The bad program P_F is presented in Algorithm 1 (on Page 7). It starts by running some steps that are similar to Robson's algorithm and proceeds with newly designed algorithm to deal with compaction, and maintain some density in each chunk. When the memory manager moves objects using its compaction quota, the program will not try to take advantage of the moved objects in their new location. There are not enough of those to justify the trouble. Instead, it will simply immediately delete these objects, and use the reclaimed space for future allocation.

Recall that we denote the density that the program attempts to maintain in each chunk by $2^{-\gamma}$. Other inputs to P_F include M, n, which is the size of the largest allocatable object; and c, the compaction budget factor.

4.1 Analysis of Program P_F

Let us now analyze the behavior of the program P_F when executing against a c-partial memory manager A. We distinguish the behavior of the program in the two stages. Denote the set of objects that P_F allocates during the first stage by S_1 and during the second stage by S_2. Also, denote the total size of the objects in S_1 by s_1 and the total size of the objects in S_2 by s_2. Finally, denote the set of objects that the memory manager chooses to compact during the first stage by Q_1 and their total size by q_1. Similarly, the corresponding set of compacted objects in the second stage is Q_2 whose accumulated size is q_2.

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

Our analysis is simplified by using a potential function $u(t)$, which we define next. It will turn out that this function provides a lower bound on the heap usage during the execution, and our goal will be to show that it becomes large by the end of the execution. The function $u(t)$ will be written as a sum of chunk functions $u_D(t)$, one for each chunk in $\mathscr{D}(i)$. The function $u_D(t)$ will be zero for all chunks that are not used to allocate objects. On the other hand, $u_D(t)$ will always be at most 2^i (i.e., the size of the chunk D) for chunks that have been used until time t.

During the analysis of the second stage, we will need to give special treatment to some of the chunks. The set of special chunks

ALGORITHM 1: Program P_F

```
Input: M,n,c,$\gamma$
**Initially:** Compute $x = \frac{1-2^{\gamma}\cdot h}{\gamma+1}$
**During the execution:** If the memory manager compacts an object,
ask the memory manager to de-allocate this object immediately
(before any other action is taken), but add this object to the set of
ghost objects, with the same address it held when it was allocated.
 1: // Stage I:
 2: $f_0 := 0$
 3: Allocate as many objects of size 1 as is possible (i.e., $M$ such objects.)
 4: for $i = 1$ to $\gamma$ do
 5:         Pick $f_i$ to be either $f_{i-1}$ or $f_{i-1} + 2^{i-1}$, according to which of the
                two maximizes
                $\sum_{o \text{ is live or ghost and } o \text{ is } f_i\text{-occupying}} 2^i - |o|$
 6:         Free every live or ghost object that is non $f_i$-occupying
 7:         Allocate $\left\lfloor \left(M - \sum_{o \text{ is live or ghost}} |o|\right) / 2^i \right\rfloor$ objects of size $2^i$
 8: end for
 9: Associate objects with chunks: consider the chunk partition $\mathscr{D}(2\gamma-1)$
    to chunks of size $2^{2\gamma-1}$. Each $f_\gamma$-occupying object is associated with
    the chunk that contains its $f_\gamma$-occupying word.
10: // Stage II:
11: for $i = 2\gamma$ to $\log(n) - 2$ do
12:     Consider the chunk partition $\mathscr{D}(i)$ of chunks of size $2^i$. Each chunk
        $D$ is composed of chunks $D_1, D_2$ of the previous step, we set the
        association: $O_D = O_{D_1} \cup O_{D_2}$
13:     For each $2^i$ chunk, Free as many objects from $O_D$ as possible such
        that $\sum_{o \in O_D} |o| \geq 2^{i-\gamma}$.
            When a half object is freed, associate it with the chunk that
            contains the other half, and re-evaluate that chunk.
14:     Allocate $\lfloor x \cdot M \cdot 2^{-i-2} \rfloor$ objects of size $2^{i+2}$ if the total size of
        allocated memory will not exceed $M$.
            Each allocated object $o$ fully cover 3 chunks $D_1, D_2, D_3$,
                                    if it cover four, pick the first three.
            Set $O_{D_1} := \{o'\}, O_{D_2} := \emptyset, O_{D_3} := \{o''\}$.
15: end for
```

will be denoted by $\mathscr{E}$ and defined later in Definition 4.12. For the analysis of the first stage, one can simply think of $\mathscr{E}$ as the empty set. Let us now set the terminology and then define the potential function.

Definition 4.3. [The chunk function $u_D(t)$] *Let A be a c-partial memory manager, and let t be any time during the execution of P_F against A which happen at step i. Let D be a chunk of size 2^i. The function $u_D(t)$ is defined as follows.*

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

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

Definition 4.4. [The potential function $u(t)$] *Let A be a c-partial memory manager, and let t be any time during the execution of P_F against A. Let i be the step in which t occurs. The function $u(t)$ is defined as follows.*

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

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

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

The guaranteed growth of $u(t)$ and the implied lower bound are shown in two lemmas 4.5 and 4.6. The first lemma, Lemma 4.5, asserts the increase of $u(t)$ during the first stage. It also bounds from above the amount of space allocated during the first stage. This bound will be used to analyze the first stage.

Lemma 4.5. *Let A be a c-partial memory manager, and let t_{first} be the time that P_F finishes the execution of its first stage when executing with A as its memory manager. Then*

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

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

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

The analysis of the second stage is summarized in Lemma 4.6. This lemma again asserts that $u(t)$ increases. However, the increase depends on the total space allocated in the second stage and also on the compaction budget in the second stage, which depends on the space allocated (in both stages). To show that the increase in the potential function $u(t)$ is high, this lemma also bounds from below the amount of allocated space s_2 in the second stage. This second bound uses an additional parameter h, which depends on γ, c, n, and M and is set to the following complicated expression in order to achieve the strongest possible bound.

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

Intuitively, h is the wasted space factor. Namely, If M is a bound on the live space allocated simultaneously by P_F, then $h \cdot M$ is the lower bound we show on the size of the heap that the memory manager must use to satisfy the allocation requests of P_F. The analysis will show that either the memory manager uses more than $M \cdot h$ space, and we are done with the proof of Theorem 1, or the program allocates a lot of space, as in the second part of the lemma, which will then be used to show that the heap space used in both stages is larger than $M \cdot h$, satisfying the assertion of Theorem 1.

Lemma 4.6. *Let A be a c-partial memory manager, and let t_{finish} be the time that P_F finishes its execution with A as its memory manager. Then,*

$$u(t_{finish}) - u(t_{first}) \geq \frac{3}{4}s_2 - 2^{\gamma} \cdot q2$$

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

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

We now show how to obtain the lower bound stated in Theorem 1 using Lemma 4.5 and 4.6. Using the fact that the memory manager compacts at most $\frac{1}{c}$ of the total allocation, we know that $(q_1 + q_2) \leq \frac{1}{c}(s_1 + s_2)$. Thus,

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

Now, if $HS(A, P_F) \geq M \cdot h$, then we are done. Otherwise, Lemma 4.6 gives us the lower bound on s_2:

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

In addition, Lemma 4.5 implies

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

and using simple algebra we get that

$$HS(A, P_F) \geq M \cdot h$$

which completes the proof of Theorem 1. ☐

4.2 Analysis of the first stage

We now focus our attention on the first stage and prove Lemma 4.5. We consider an execution of the malicious program P_F with any memory manager A and look at the first γ steps. In Step i, the size of the chunks is 2^i words, and therefore the size of any chunk throughout the first stage is not larger than 2^{γ} words. If any object is associated with a chunk, and since any object is of size at least one word, then the fraction (or density) of live space associated with that chunk must be at least $2^{-\gamma}$. When the density is guaranteed to be that high, and compaction is limited by the $\frac{1}{c} < 2^{-\gamma}$ fraction, compaction is very limited and not very beneficial to the memory manager. Therefore, for these initial steps (of the first stage) we chose to use a program that is very similar to a program presented by Robson. Robson used his program for the case where no compaction is allowed. We will analyze the slightly modified program to show that it is still useful when limited compaction is used by the memory manager.

For completeness, let us recall Robson's program in Algorithm 2. In the algorithm we use the term *f-occupying objects* that was defined in Definition 4.2. Also, an object is *live* if it is in the heap, i.e., has not been de-allocated. A simple example to the behavior of Algorithm 2 is depicted in Figure 5. In this example, the object O_3 will be freed in Line 5, since it is not f_i occupying.

ALGORITHM 2: Robson's "bad" program P_R

```
1: Initially: $f_0 := 0$
2: Allocate $M$ objects of size 1.
3: for $i = 1$ to $\gamma$ do
4:         Pick $f_i \in \{f_{i-1}, f_{i-1} + 2^{i-1}\}$ that maximizes
                 $\sum_{o \text{ is live and } f_i\text{-occupying}} 2^i - |o|$.
5:         Free all non-$f_i$-occupying objects
6:         Allocate as many objects of size $2^i$ as possible (within the $M$
               live space bound.)
7: end for
```

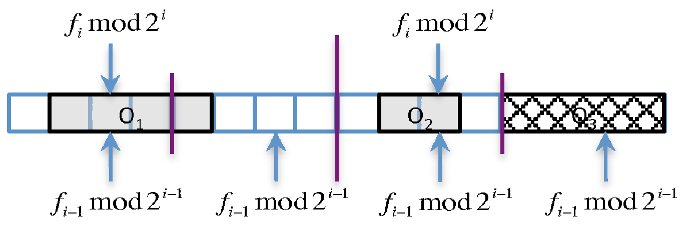

Figure 5. Lower bound on the waste factor h for realistic parameters ($M = 256MB$ and $n = 1MB$) as a function of c

The original program was designed to maximize fragmentation when no compaction is allowed. In our adaptation, appearing as the first stage of Program P_F, we also handle the case that the memory manager employs compaction. When a compaction occurs, P_F immediately de-allocates the moved objects. But we would still like to adopt the original analysis of Robson without redoing the entire analysis for the slightly modified version that was used in the first stage of P_F. To this end, we use a mind experiment in which we let Robson's malicious program P_R run against an imaginary memory manager A' that does not move objects. Clearly, Robson's analysis holds for the execution (P_R, A'), as it holds for all memory managers that do not move objects. From this analysis we will be able to also deduce a lower bound on the heap size that A uses while satisfying P_F's allocation and de-allocation sequence. To make a connection between the first stage of P_F and P_R we note that their only difference is that P_F must deal with compacted objects. (Such objects are de-allocated by P_F, but still count for the decisions on future de-allocation of all objects.) Otherwise, it behaves exactly like the original P_R.

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

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

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

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

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

Definition 4.7. [Memory manager $A'(A,P_F)$] *The memory manager $A'(A,P_F)$ works as follows. The k-th object that P' allocates is placed in a location in the memory whose address is equal modulo 2^γ to where A placed the k-th object that P_F allocated. There are infinitely many such locations and A' chooses one of them that does not contain any other object previously allocated in an arbitrary manner.*

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

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

Claim 4.8. *Consider the execution of P_F against a memory manager A, and the execution of P_R against $A'(A,P_F)$, and suppose that both finished their Step i. There is one to one mapping between objects in A and objects in A' with the following property.*

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

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

Proof: The one to one mapping we chose maps the k-th object that P_F allocated to the k-th object that P_R allocated. By the definition of A' their address is equal modulo 2^γ and we are done with the third property. The other two properties are more involved. The full proof appears in [13].

We now quote an implicit lemma from Robson's analysis, that will help us in bounding the value of the potential function at the end of the first stage of P_F.

Claim 4.9 ([14], inequality 1)**.** *After the execution of step i, there are at least $M\frac{i+1+1}{2^{i+1}} = M\frac{i+2}{2\cdot 2^i}$ objects that are f_i-occupying.*

By the end of the first stage execution, in P_F Line 9, an object o is associated with the chunk that contains its f_γ-occupying word. We now want to use Robson's guarantee for many f_γ-occupying objects in order to say that there are many associated objects, and prove the first part of Lemma 4.5.

Claim 4.10. *Let A be a memory manager, and let t_{first} be the time P_F finished the execution of its first stage against A. Then*

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

Proof sketch: By Claim 4.9 there are lots of f_i-occupying objects and by Claim 4.8 this also holds for the execution of P_F. The proof follows from these two and the definition of $u_D(t)$. □

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

Claim 4.11. *Let A be a memory manager, and consider the execution of P_F's first stage against A. The total size of memory that P_F allocated is at most*

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

Proof: Omitted for lack of space. □

The proof of Lemma 4.5 directly follows from Claim 4.10 and claim 4.11.

4.3 Analysis of the second stage

In this section we prove Lemma 4.6, which asserts a substantial growth in the potential function during the second stage. We let t_{first} represent the time where the first stage completes, and t_{finish} represent the time the second stage (and the entire algorithm) completes. Let us recall the statement of the lemma.

Lemma 4.6. *Let A be a c-partial memory manager, and let t_{finish} be the time that P_F finishes its execution with A as its memory manager. Then,*

$$u(t_{finish}) - u(t_{first}) \geq \frac{3}{4}s_2 - 2^\gamma \cdot q2.$$

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

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

To show that this lemma holds, we look at changes that might occur during the execution. This includes allocation of new objects (initiated by P_F), compaction of objects (by the memory manager), de-allocation of objects (by P_F), and a change of steps that changes the chunk sizes, and therefore the summation over the chunks and each chunks $u_D(t)$ function. We will show that we get substantial growth during allocations, and also that all other events do not decrease the potential function.

Recall that the potential function is defined as
$u(t) = \sum_{D\in\mathscr{D}(i)} u_D(t) - \frac{n}{4}$ and we need to show that the value of the potential function grows by at least $\frac{3}{4}s_2 - 2^\gamma \cdot q_2$ during the second stage of the execution.

We start by looking at allocations and show that whenever P_F allocates an object, either the potential function gets larger, or some compaction occur (and the potential function does not decrease). We then use the fact that compaction is limited to get the potential function growth we need. Suppose an object o is allocated during Step i of the execution of P_F with A. The size of o is $4 \cdot 2^i$ (by the definition of P_F) and so when the memory manager places o in the heap, it consumes at least three full consecutive chunks (of size 2^i)

and maybe some additional space from the chunk preceding and the chunk that comes after these three consecutive chunks. Denote by D_1, D_2 and D_3 the three chunks that are fully covered by o and are selected at Step 14 of Algorithm 1. All three must be empty when o is placed. We claim that this transition from empty chunks to full chunks makes the potential function grow. We will show that the value of $u_{D_1}(t)+u_{D_2}(t)+u_{D_3}(t)$ grows, which implies that the value of $u(t)$ grows. Note that $u_D(t)$ for any other chunk D (other than D_1, D_2 or D_3) is not affected by this allocation, because the set of associated objects as well as membership in $\mathscr{E}$ (which we define in the next paragraph) only changes for D_1, D_2, and D_3.

During allocation step of P_F (Line 14), when an object o is allocated, P_F associates D_1 with the first half of object o and D_3 with the second half of Object o. But since an object is associated with at most two chunks (each half can be associated with a chunk), it follows that D_2 is left with no object associated with it, in spite of it being completely covered by the allocated object o. The goal of the set $\mathscr{E}$ is to deal with these middle chunks. This set will contain all such middle chunks and make $u_{D_2}(t)$ of Definition 4.3 be set to 2^i. We note that this "anomaly", of a chunk being covered by an object but with no associated object, is temporary and it disappears at the next step change since the middle chunk is joined with either its left or right chunk, and they become a single chunk with which o (or o's half) can be associated. Let us now define $\mathscr{E}$.

Definition 4.12. [The set $\mathscr{E}(t)$] *Let A be a memory manager, and consider any time t during the execution of P_F's second stage with A as its memory manager. Let i be the step where t happens. The* set of middle chunks $\mathscr{E}(t) \subset \mathscr{D}(i)$ *is the set of chunks that both their left adjacent chunk and their right adjacent chunk were fully covered by an object o allocated at step i (thus, the chunk itself is also covered by o), but half of o was not associated with it. A chunk remains in $\mathscr{E}(t)$ until either an new step kicks in, or an object is associated with this chunk. The later may happen if o was compacted during step i, and another object is allocated there.*

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

Claim 4.13. *Let t be a time during the execution of P_F against a memory manager A, and let i be the step in which t occurs. Let D_1 and D_2 be two consecutive chunks such that $D_1 \in \mathscr{E}(t)$. Then there exists an object o that is allocated during Step i, such that at time t half of o is associated with D_2. The same holds for D_1 when $D_2 \in \mathscr{E}(t)$.*

Proof sketch: When a chunk joins $\mathscr{E}$ both chunks to its left and right are associated with half objects allocated in Step i. ☐

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

When the second case occurs, i.e., that the memory manager compacts away objects from a chunk before placing o, we are not able to show that the value of the potential function grows. However, we get that some compaction occurred, and we use that to bound the number of such events. To this end, we associate some compaction value with the newly allocated object. Recall that the objects that were compacted away from a chunk and then de-allocated immediately by P_F are still considered associated with the chunk until a new object is placed on the chunk. Therefore, to determine how much compaction occurred to free space for the allocation, we can just check the objects that were associated with these chunks right before the allocation. The formal definition follows.

Definition 4.14. [Compaction space associated with an object] *Let o be an object allocated during the execution of P_F's second stage against a memory manager A. Let $t+1$ be the allocation time (and t be the time just before the allocation), and let D_1, D_2, and D_3 be the chunks picked by P_F at Step 14 of P_F, after allocating o. Then we define* the compacted space associated with the object *o to be*

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

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

Claim 4.15. *Consider a time t during the execution of P_F's second stage against a memory manager A, and let i be the step where t happens. Then the following three properties hold:*

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

Proof sketch: The proof follows from the definition of the program P_F and its way of associating objects with chunks. The full proof is omitted for lack of space. ☐

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

Claim 4.16. *Let A be a memory manager, let $u(t)$ be the potential function as defined in Definition 4.4, and $q(o)$ be the compaction associated with an object as defined in Definition 4.14. Then during execution of P_F against A, the following properties of $u(t)$ holds*

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

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

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

If either $D_1, D_2 \in \mathscr{E}(t-1)$, then by Claim 4.13 there exists an object o that was allocated in Step $i-1$, and half of o is associated with either D_1 or D_2. Since half of o is associate with D, and the size of half of o is 2^i, it holds that $\sum_{o' \in O_D} |o'| = 2^i$. Thus (recall definition 4.3 of $u_D(t)$)

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

Otherwise, both chunks are not in $\mathscr{E}$ and we know that the objects that are associated with them are disjoint. Therefore, according to Step 12 in P_F,

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

and the statement follows by definition of $u_D(t)$.

P_F de-allocates an object. By definition, P_F does not free an object if this decrease $\sum_{o \in O_D} |o|$ below $2^{i-\gamma}$, so $u_D(t)$ does not decrease.

***P_F allocates an object** o.* Let t be the time P_F allocates an object o, and let D_1, D_2, D_3 be the three chunks that P_F picked. Only the sets $O_{D_1}, O_{D_2}, O_{D_3}$ are changed during allocation. After the allocation, size of each half an object is $2 \cdot 2^i$, so $u_{D_1}(t) = u_{D_3}(t) = \min(2^\gamma \cdot 2 \cdot 2^i, 2^i) = 2^i$, and $D_2 \in \mathscr{E}(t)$. Thus

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

Before the allocation, if either chunks was contained in $\mathscr{E}$, then by Claim 4.13 there exists an object o allocated at step i, and half of o is associated with D_1, D_2, or D_3. In this case $2^\gamma \cdot q(o) \geq 2^\gamma \cdot 2^{i+1} \geq 3 \cdot 2^i$. The last inequality follows since $\gamma \geq 1$. Otherwise, $u_{D_i}(t-1) \leq 2^\gamma \cdot \sum_{o' \in O_{D_i}(t-1)} |o'|$ for $i = 1,2,3$. In both case we have $2^\gamma \cdot q(o) \geq u_{D_1}(t-1) + u_{D_2}(t-1) + u_{D_3}(t-1)$. Subtracting the values before and after the allocation of o provides the bound in this case and we are done □

We now turn to bound the amount of memory that P_F allocates during the execution of its second stage. By its definition, In Step 14, P_F attempts to allocate a $\lfloor x \cdot M \cdot 2^{-i-2} \rfloor$ objects of size 2^{i+2} if the total size of allocated memory does not exceed M. In order to show that it allocates a lot, we need to show that the bound of not exceeding M simultaneously allocated words does not limit this allocation too much. To this end, we need to bound the space occupied by live objects in the heap from above, which implies a lower bound on the amount of memory available for allocation.

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

Proposition 4.17. *Consider the execution of P_F's second stage against a memory manager A. Let t be a time when P_F allocates an object, and let i be the step in which t happens. Then for every chunk $D \in \mathscr{D}(i)$ either $|O_D(t)| \leq 1$, or $\sum_{o \in O_D(t)} |o| \leq 2^{i-\gamma}$.*

Proof sketch: The proof of this proposition follows from the behavior of P_F as defined in Line 13. □

Next, we bound the space P_F allocates during its second stage.

Claim 4.18. *Consider the execution of P_F against a memory manager A. Then either A uses more than $M \cdot h$ heap space, or the number of words that that P_F allocates during its second stage satisfies*

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

Proof sketch: Recall that x is set at the beginning of Algorithm 1 to: $\frac{1-h \cdot 2^{-\gamma}}{\gamma+1}$. We show that in each step P_F's allocation of $M \cdot x$ words does not exceed the M bound, and then show that the accumulated size of allocation is large enough. □

Recall that in Claim 4.16 we bounded the increase in $u(t)$ when an object o is allocated by $\frac{3}{4}|o| - 2^\gamma \cdot q(o)$. Summing $\frac{3}{4}|o|$ over all objects allocated during stage two of P_F gives $\frac{3}{4}s_2$, which we also computed. We now bound the total sum of $q(o)$ over all objects allocated during stage two of P_F. The precise value of $q(o)$ was given in Definition 4.14.

Proposition 4.19. *Consider the execution of P_F against a memory manager A. S_2 is the set of objects allocated at P_F second stage, and q_2 is the total size of objects that were compacted during the second stage. Then $q_2 \geq \sum_{o \in S_2} q(o)$.*

Proof. Omitted for lack of space. □

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

Claim 4.20. *Consider the execution of P_F against a memory manager A, and let t_{finish} be the time when P_F finished it execution, and t_{first} be the time when P_F finished execution of first stage. Then*

$$u(t_{finish}) \geq u(t_{first}) + \frac{3}{4}s_2 - 2^\gamma \cdot q2$$

Proof: Omitted for lack of space. □

The proof of Lemma 4.6 directly follows from Claim 4.18 and Claim 4.20.

5. Conclusion

This work contributes to building a solid theoretical foundation for memory management. In particular, it extends previous work by providing new lower and upper bounds on the effectiveness of partial compaction. Our lower bound is the first bound in the literature that carries implications for practical systems existing today, by showing that some desirable (realistic) compaction goals cannot be achieved.

References

[1] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein. An efficient parallel heap compaction algorithm. In *OOPSLA 2004.*

[2] D. F. Bacon, P. Cheng, and V. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL 2003.*

[3] O. Ben-Yitzhak, I. Goft, E. Kolodner, K. Kuiper, and V. Leikehman. An algorithm for parallel incremental compaction. In ISMM 2002.

[4] A. Bendersky and E. Petrank. Space overhead bounds for dynamic memory management with partial compaction. *POPL 2011.*

[5] H.-J. Boehm. Bounding space usage of conservative garbage collectors. In *POPL 2002.*

[6] H.-J. Boehm. The space cost of lazy reference counting. *POPL 2004.*

[7] C. Click, G. Tene, and M. Wolf. The Pauseless GC algorithm. VEE 2005.

[8] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *ISMM 2004.*

[9] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall, Aug. 2011.

[10] H. Kermany and E. Petrank. The Compressor: Concurrent, incremental and parallel compaction. In *PLDI 2006.*

[11] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *POPL 2002.*

[12] F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time garbage collectors. In *PLDI 2008.*

[13] N. Cohen and E. Petrank. Limitations of Partial Compaction: Towards Practical Bounds. http://www.cs.technion.ac.il/%7eerez/%50apers/compaction-full.pdf.

[14] J. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM*, 21(3):491–499, 1974.

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