Limitations of Partial Compaction: Towards Practical Bounds

NACHSHON COHEN and EREZ PETRANK, Technion Institute of Technology, Israel

Compaction of a managed heap is a costly operation to be avoided as much as possible in commercial runtimes. Instead, partial compaction is often used to defragment parts of the heap and avoid space blowup. 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

Additional Key Words and Phrases: Memory management, compaction, fragmentation, theory, lower bounds

ACM Reference Format:
Nachshon Cohen and Erez Petrank. 2017. Limitations of partial compaction: Towards practical bounds. ACM Trans. Program. Lang. Syst. 39, 1, Article 2 (March 2017), 44 pages.
DOI: http://dx.doi.org/10.1145/2994597

1. INTRODUCTION

The theoretical foundations of memory management have not been comprehensively studied. Little is known about the limitations of various memory management functionalities; information about space consumption of various memory management methods in particular is lacking. Previous work consists of Robson's classic results on fragmentation when no compaction is employed [Robson 1974, 1971], a result on the hardness of achieving cache consciousness [Petrank and Rawitz 2002] and work on the effectiveness of conservative garbage collection and lazy reference counting [Boehm 2002, 2004]. In a recent new work, Bendersky and Petrank [2011] attempted to bound the overhead mitigation that can be achieved by partial compaction.

Memory managers typically suffer from fragmentation. Allocation and deallocation of objects in the heap create "holes" between objects that may be too small for future allocation, causing available heap space to be wasted. Compaction can eliminate this problem, but compaction algorithms are notoriously costly [Jones et al. 2011; Abuaiadh et al. 2004; Kermany and Petrank 2006]. Instead, memory managers today either seldom apply full compaction or employ partial compaction, where only a (small) fraction of the heap objects are compacted to make space for further allocation [Ben-Yitzhak et al. 2002; Bacon et al. 2003; Click et al. 2005; Detlefs et al. 2004; Pizlo et al. 2008].

This work was supported by the Israeli Science Foundation Grant No. 274/14 and by the Nechemya LevZion VATAT scholarship.
Authors' addresses: N. Cohen, School of Computer and Communication Sciences, EPFL, Route Cantonale, 1015 Lausanne, Switzerland; email: nachshonc@gmail.com; E. Petrank, Computer Science Department, Technion, Haifa, Israel; email: erez@cs.technion.ac.il.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2017 ACM 0164-0925/2017/03-ART2 $15.00
DOI: http://dx.doi.org/10.1145/2994597

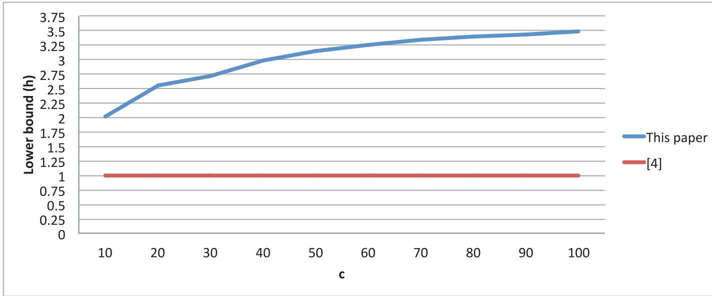

Fig. 1. Lower bound on the waste factor h for realistic parameters ($M = 256\text{MB}$ and $n = 1\text{MB}$) as a function of c.

Bendersky and Petrank [2011] study the limitations of partial compaction. But despite their work being novel and opening a new direction for bounding the effectiveness of partial compaction, their results are only significant for huge heaps and objects. When used in a realistic setting of system parameters today, their lower bounds become meaningless. Suppose, for example, that a program uses a live heap space of 256MB and allocates objects of size at most 1MB. For such a program, even if the memory manager is limited and can only compact 1% of the allocated objects, the results in Bendersky and Petrank [2011] would only imply that the heap must be at least 256MB, which is obvious and not very useful.

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

In general, the more objects we let the memory manager move, the lower the space overhead that it may suffer. To put a bound on the amount of compaction work undertaken by the memory manager, we use the model set in Bendersky and Petrank [2011] and bound the compaction by a constant fraction $1/c$ of the total allocated space. This means that at any point in the execution, if a space of s words has been allocated so far, the total compaction allowed up to this point in the execution is s/c words, where c is the compaction bound.

The results we obtain involve some non-trivial mathematical arguments, and the obtained lower bound is presented in a rather complex formula. The general lower bound is stated in Theorem 2.1. But to make clear the improvement over the previously known results, we have depicted in Figure 1 the space overhead factor for the parameters mentioned above (which we consider realistic). Namely, for a program that uses live space of $M = 256\text{MB}$ and whose largest allocated object is at most $n = 1\text{MB}$, we drew for different values of c the required heap size as a factor of the 256MB live space. Note that if we were willing to execute full compaction after each deallocation, then the overhead factor would have been 1. We could have used a heap size of 256MB and serve all allocation and deallocation requests. But with limited (partial) compaction, our results show that this is not possible. In Figure 1 we drew our lower bound as well as the lower bound obtained from Bendersky and Petrank [2011] for these parameters. In fact, throughout the range of $c = 10, \ldots, 100$, the lower bound from Bendersky and Petrank [2011] gives nothing but the trivial lower bound overhead factor of 1, meaning that 256MB are required to serve the program. In contrast, our new techniques show that the space overhead factor must be at least 2, that is, 512MB when 10% of the

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

In general, the result described in this work is theoretical and does not provide a new system or algorithm. Instead, it describes a limitation that memory managers can never overcome. As such, it does serve a practical need, by letting practitioners know what they cannot aspire to and should not expend efforts in trying to achieve. The lower bounds we provide are for a worst-case scenario and do not rule out better behavior on a suite of benchmarks. But providing a better guaranteed bound on fragmentation (as required for critical systems such as real-time systems) is not possible. Note that this bound holds for manual memory managers as well as automatic ones, even when applying sophisticated methods such as copying collection, mark-compact, Lang-Dupont, or MC^2 [Jones et al. 2011].

Finally, we slightly improve the related, state-of-the-art upper bound. The upper bound is shown by presenting a memory manager that keeps fragmentation low against all possible programs. The new upper bound slightly improves the result of Bendersky and Petrank [2011] by providing a better memory manager and a better analysis for its worst space overhead. This slight improvement is depicted in Figure 3, where the new upper bound is compared to the previous upper bound of Bendersky and Petrank [2011]. The upper bound theorem is stated rigorously as Theorem 2.2. 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 article, we stand on the shoulders of prior work, and in particular our techniques build on and extend the techniques proposed in Robson [1971] and Bendersky and Petrank [2011].

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. In Section 5, we provide the proof of the upper bound, and we conclude in Section 7.

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 deallocation requests as a series of sub-interactions of the following form:

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

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

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

A second important parameter of the execution of a program (from a memory management point of view) is the size variation of the objects it allocates. If all objects are of fixed size, say, 1, then a heap space of M is always sufficient. Although holes can

be created by deallocating objects, these holes can always be filled by newly allocated objects. If we denote the least size of an object by 1, then 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 $\mathcal{P}(M, n)$ 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 $\mathcal{P}_2(M, n)$ the set of programs whose allocated objects sizes are always a power of 2.

As explained in the Introduction, if the heap is compacted after every deallocation, then 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 article, we adopt the definition of Bendersky and Petrank [2011] and consider memory managers that limit their compaction efforts by a predetermined fraction of the allocated space. For a constant $c > 1$, a memory manager is a c-partial memory manager if it compacts at most $\frac{1}{c}$ of the total space allocated by the program. We denote the set of c-partial memory managers by $\mathcal{A}(c)$.

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

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

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

2.2. Previous work

For programs that allocate only objects whose size is a power of 2, the size of largest object n divides M, and for all memory managers that do not use compaction, Robson [1974, 1971] proved lower and upper bounds that match. For his lower bound, he presented a "bad" program $P_o \in \mathcal{P}_2(M, n)$ that forces any memory manager (that does not use compaction) to have a large heap. Specifically,

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

Note that here, and throughout the article, $\lg \triangleq \log_2$. For an upper bound, Robson presented an allocator A_o that satisfies the allocation requests of any program in

$\mathcal{P}_2(M, n)$ using a heap size of

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

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

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

The techniques in the proof of Robson [1971] actually yield a slightly stronger result for (realistic) cases in which $M >> n$:

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

In the rest of this article, we think of that result as being Robson's best upper bound and show how we improve over that.

Subsequent to the above work, Robson [1974] went on and improved the *asymptotic* heap size upper bound to be

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

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

When some compaction is allowed (but not an unlimited compaction effort), much less is known. For the upper bound, Bendersky and Petrank [2011] have shown a simple compacting collector $A_c \in \mathcal{A}(c)$ that uses a heap space of at most

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

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

They have also shown a "bad" program P_W that forces all memory managers with a c-partial compaction bound to use a large heap:

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

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 Bendersky and Petrank [2011] is important for modeling the problem, providing some tools for solving it, and although the authors provide some tools for solving it, this bound is meaningful only for huge objects and heaps. In particular, it is higher than the obvious M only for $M > n \geq 16TB$. In this work we extend the theory enough to obtain meaningful results for practical values of M and n.

THEOREM 2.1. *For any c-partial memory manager A and for any $M > n > 1$, there exists a program $P_F \in \mathcal{P}_2(M, n)$ such that for any integral $\gamma \leq \lg(\frac{3}{4}c) : \gamma \in \mathbb{N}$,*

$$\min_{A \in \mathcal{A}(c)} HS(A, P_F) \geq M \cdot h, \tag{2}$$

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

where h is set to:

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

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

Since h is given in a complicated formula, the implications of $HS(A, P_F) \ge M \cdot h$ are not very intuitive. Therefore, we chose some realistic parameters to check how this bound behaves in practice. We chose M, the size of the allocated live space to be 256MB, and n, the size of the largest allocatable object, to be 1 MB. With these parameters fixed and with the parameter γ set to the value that maximizes the bound, we drew a graph of h as a function of the compaction quota bound c. This graph appears in Figure 1. The x-axis has c varying between 10 and 100. Setting $c = 10$ means that we have enough budget to move 10% of the allocated space, whereas setting $c = 100$ means that we have enough budget to move 1% of the allocated space. For these c's, the y-axis represents the obtained lower bound as a multiplier of M. For example, when compaction of 2% of all allocated space is allowed ($c = 50$), any memory manager will need to use a heap size of at least $3.15 \cdot M$. Even with 10% of the allocated space being compacted, a heap size of $2 \cdot M = 512$MB is unavoidable. For these practical parameters, previous results in Robson [1974] and in Bendersky and Petrank [2011] do not provide any bound, except for the obvious one that the heap must be at least of size M.

We also depicted the lower bound as a function of a varying maximum object size n. We fixed the compaction budget to $c = 50$ and the total size of live objects to $M = 256n$. The rationale for the last parameter setting is that, typically, a single object does not make up a significant part of the heap (larger than half a percentage). Setting M to a larger value does not change the bound. We argue below that fixing $c = 50$ (i.e., compacting 2% of the allocated objects) represents a realistic setting. Under stable conditions, where the space occupied by live objects is stable and the heap is partially compacted whenever it is full, one can translate the ratio c of compacted objects over allocated objects into a bound on compaction as a percentage of the live objects space M. As an example, suppose the live space M forms a third of the entire heap space HS and suppose we compact when the heap is exhausted. In that case, $2M$ words are allocated before compaction happens and then compacting 2% of the allocated space $2M$ means compacting 4% of the live space M. Such partial compaction reflects realistic settings. For example, Pizlo et al. [2008] executes partial compaction every five garbage collection cycles. In each partial compaction cycle, at most 10% of the pages that are less than 50% occupied are compacted. In the setting above, this means compacting 1% of the $3M$ heap size once $2M$ memory is allocated or $c = 66$. In another system, Bacon et al. [2003] employs partial compaction and measures the amount of compaction compared to the live size (during a garbage collection cycle). The amount of compaction falls in the range of 0.6%–4.11%. Translating into c values, this correlates to the range of 48–300.

We let the size of the largest object n vary between 1KB and 1GB, and for these n values, the y-axis represents the obtained lower bound as a multiplier of M. The graph is depicted in Figure 2. We could also depict the lower bound as a function of M, where n and c remain fixed. However, in a practical setting, the size of the largest object is much smaller than the total live space (i.e., n/M is small). Hence, the lower bound as

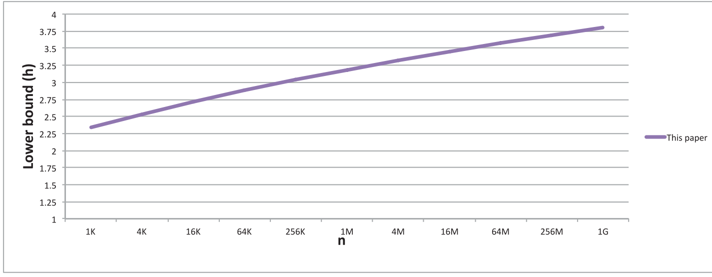

Fig. 2. Lower bound on the waste factor h as a function of n ($c = 50$, $M = 256n$).

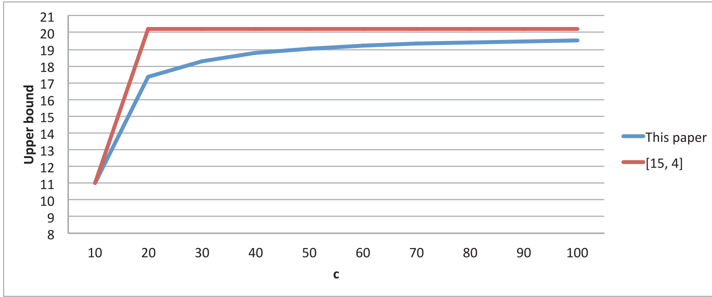

Fig. 3. Upper bound on the waste factor for realistic parameters ($M = 256$MB and $n = 1$MB) as a function of c.

a function of M is very close to a constant function and does not provide additional interesting information.

We also consider the upper bound on the size of heap required. In Bendersky and Petrank [2011], an upper bound of the form $(c + 1)M$ was presented. However, this upper bound may become non-interesting when Robson's upper bound is stronger, meaning that the same heap size may be obtained without moving objects at all. This happens when $c > \lg n + 1$. As partial compactors often use a large c to limit the fraction $1/c$ of moved objects, such a scenario seems plausible. We provide some improvement to Robson's algorithm when little compaction is allowed and obtains better upper bound as follows.

THEOREM 2.2. *There exists a c-partial memory manager $A \in \mathcal{A}(c)$ that satisfies allocation requests of any program $P \in \mathcal{P}(M, n)$ with heap size at most*

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

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

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

As the formulas in this theorem are also not easy to grasp, we drew a graph comparing previously known bounds with the new result. It can be seen in Figure 3 that for c's between 20 and 100, Theorem 2.2 improves the upper bound, with the largest

improvement being 15% at $c = 20$. We consider this result minor and the lower bound the major result in this article.

2.4. Alternative Budgeting Models

Following previous work [Bendersky and Petrank 2011], we budget compaction as a fraction of the total allocation space. The main reason for that is that this model is clean and it is possible to prove strong rigorous theoretical results on it. Practitioners that build garbage collected systems may think of this budgeting as unnatural because, in practice, compaction is run with garbage collection, and the volume of compaction depends on the amount of live objects (because that determines the triggering of a collection) and on the shape of the heap (e.g., freeing memory pages that are sparsely occupied). See, for example, Bacon et al. [2003], Detlefs et al. [2004], and Pizlo et al. [2008]. Let us say a few words about the alternatives.

Perhaps the most natural alternative is to budget allocation as a fraction of the amount of live space. But this alternative is not as clean, it is harder to work with, and it poses some problems. First, to adopt this budgeting model, one must also specify how often a fraction of the live space can be moved. Triggering of garbage collection may vary in practice because the size of the heap is usually not fixed. But suppose we fix the size of the heap and trigger garbage collection when the heap is exhausted. When do we decide that the heap is full?

In practice, that would depend on fragmentation. But suppose that we also ignore fragmentation to make the budgeting cleaner. We claim that in this simplified case and under the additional assumption that the amount of live space is stable, the resulting budget is correlated to the one used in this article (i.e., a percentage of the allocated objects). To see this, think of the amount of live space as stable and denote this size by M. The space allocated between two garbage collection cycles becomes fixed in this case: HeapSize (HS) minus M. So after allocating $HS - M$ words, we allow moving a fraction of M. Since we have assumed that HS and M are fixed, then we can write HS as a multiple of M. For example, $HS = 3M$ is common in practice. In this setting, a fraction of M is also a fraction of $HS - M$.

For example, Theorem 2.1 implies that $HS = 3 \cdot M$ is impossible to serve with $c = 50$. Alternatively, this may be phrased as an impossibility to serve a live fraction that is bigger than a third of the heap. In this case, one can think of the memory manager as being restricted from compacting more than $\frac{1}{25} \cdot M$ words after allocating $2 \cdot M$ words. In general, we stick to the budgeting model of Bendersky and Petrank [2011] because it is clean and requires no further assumptions and we can produce meaningful results with it.

It is worthwhile to notice that under the assumptions discussed above, namely that the amount of live space is stable and the heap size is fixed, the ratio between the heap size and M is the live ratio. Thus no allocator can keep the live ratio bigger than $\frac{1}{h} = M/HS$ in all cases. Recall that h is the lower bound presented in Theorem 2.1 whose value is set in Equation (3).

3. OVERVIEW AND INTUITIONS

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

The bad program will work in steps, allocating, in each step, only objects of size 2^i, for some i that it will determine. Consider such a step and consider a memory region that

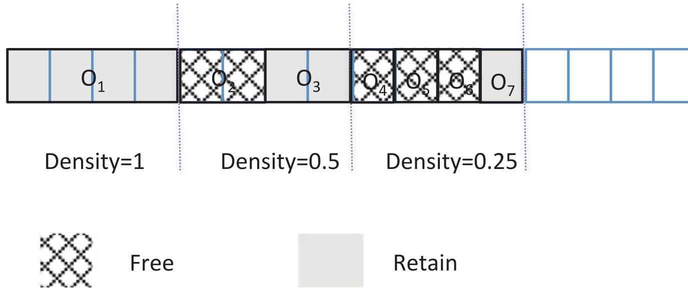

Fig. 4. Example of different densities.

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

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

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

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

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

Finally, let us discuss how we deal with objects that are not aligned and thus reside on the border of two chunks. If objects are aligned, then an object is allocated exactly on one full single chunk. In order to allocate an object, the chunk it is put on must be entirely free of objects. When an object's allocation is not aligned, it may reside on two chunks. We start by looking at smaller chunks, whose size is one-quarter of the allocated object. This means that a non-aligned allocated object must entirely fill three chunks (and partially sit on two more chunks). These three chunks must be completely

free of allocated objects before the allocation can take place. If one of these three chunks is about to be reused, then the memory manager compacts away every object that resides (even partially) on the reused chunk and loses some compaction budget. Note that we have to make sure that if two adjacent chunks are reused, we do not double count budget loss due to the moving of a non-aligned object that resides on both.

3.1. Improvements Over Prior Work [Bendersky and Petrank 2011]

In this subsection, we give an overview of the three main improvements of this work over Bendersky and Petrank [2011]. These improvements allowed us to achieve a better lower bound, meaningful with realistic parameter settings.

The first improvement follows from noting that, in the first steps, the allocated objects are large while chunk sizes are small. Compaction is of no benefit to the memory manager in this case. Therefore, in these first steps, we run a program that is very similar to Robson's [1974] bad program but adapted to this case. We develop a reduction theorem to show that this program creates fragmentation even when compaction is allowed. Furthermore, these first steps nicely integrate with the algorithm that runs in the second stage during the rest of the steps. This improvement of using Robson's simpler algorithm for the first stage cannot be extended to the second stage. When moving into later steps where the chunk sizes get large, it becomes possible for the memory manager to gain substantially from moving objects. Moving a relatively small object and gaining a free chunk for further allocations is highly beneficial for the memory manager in order to be able to work with a smaller heap. Robson's bad program is not appropriate to be used in such a setting, because it assumes no compaction happens. Therefore, we go back to using the ideas from Bendersky and Petrank [2011] and use a program similar to theirs in the second stage. This program attempts to keep the density of objects high in each chunk. When the density of allocated objects is high enough in a chunk, too much space needs to be moved from a chunk in order to allow allocating on it again.

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

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

4. LOWER BOUND: CREATING FRAGMENTATION

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

Let us start by explaining the ideas behind the construction of P_F. The program P_F works in two stages. The second stage is probably the major contributor to the fragmentation, but the first stage is also necessary to obtain our results. The first stage is an adaptation of Robson's malicious program [Robson 1974], which attempts to fragment the memory as much as possible, when working against a memory manager that cannot move objects. We will discuss in Section 4.2 how this algorithm behaves against a memory manager that can move objects and show that it buys some fragmentation in this case as well. After running for 2γ steps, a second stage starts, which behaves differently. The second stage works in steps $i = 2\gamma, \ldots, \lg(n)-2$, and in each step it only requests allocations of objects of size 2^{i+2} words. At each such step of the execution, we consider a partition of the heap space into aligned chunks of size 2^i words. This means, for example, that each allocated object either consumes four full consecutive chunks if its allocation is aligned, or it consumes at least three full consecutive chunks.

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

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

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

We denote by $O_D(t)$ the set of objects that P_F associates with chunk D at time t, and we sometimes omit the t, when its value is clear from the context. When an object lies on the border of two chunks, we sometimes choose to associate it with both. In this case, we associate exactly half of it with each of the chunks (ignoring the actual way the object is split between them). This even split in association is used to preserve the property that object sizes (and association sizes) are a power of 2. 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

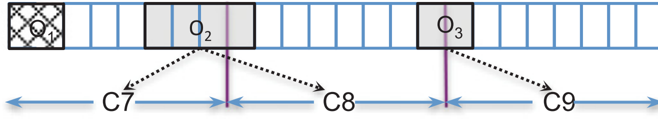

Fig. 5. Association of objects and half objects with chunks. O_3 is associated with Chunk C_9 (only), so its density is 0.25, even though only one of the eight words of O_3 is covered. In addition, half of O_2 is associated with C_8 even though only a quarter actually intersects C_8. The other half is associated with C_7. This makes the densities of C_7, C_8, and C_9 0.25.

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

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

An example of density threshold and association set is depicted in Figure 5. Let the density threshold $2^{-\gamma}$ be 1/4, which consists of 2 words per chunk of size 8. Half of O_2 is associated with Chunk $C7$, the other half is associated with Chunk $C8$, and the object O_3 is associated with Chunk $C9$ only. These objects suffice to make the density of each chunk at least 1/4. The program can free the object O_1 since a density of 1/4 is preserved even without it. In this example, we actually count the associated space of an object rather than the actual space the object occupies on a chunk. When clearing a chunk, an entire object must be moved, even if only one word of it resides on the chunk. For the sake of the analysis, we count half of this moving cost for each of the object's chunks. If we move the object, then we pay for the compaction of its entire size, but both chunks gain space for allocation.

For the first steps in the computation, maintaining a high-enough density is irrelevant: Chunks are small enough so when even one word is allocated on the chunk, the density is $2^{-\gamma}$. Therefore, it is never useful for the memory manager to compact, and we can simply adopt Robson's "bad" program [Robson 1974] with a technical variation that enables it to deal with compaction. While compaction is not beneficial to the memory manager, it may still occur and our bad program must deal with it, even though Robson's original program does not. We build a program P_F that will be "similar" to Robson's program in the sense that it will keep a similar heap shape, it will make very similar decisions on which objects to deallocate, and it will allocate the same amount of space in each step. We will then present a reduction showing that if there exists a memory manager M that can maintain low fragmentation while serving P_F with bounded compaction, it is possible to create another memory manager M' that does not move objects and maintains low fragmentation against Robson's program. Since no memory manager can keep low fragmentation against Robson's program, we get the lower bound we need.

When objects are moved by the memory manager, the simulation of the original interaction of Robson's program with its memory manager cannot simply proceed, because Robson's program does not handle movements of objects. Several new factors

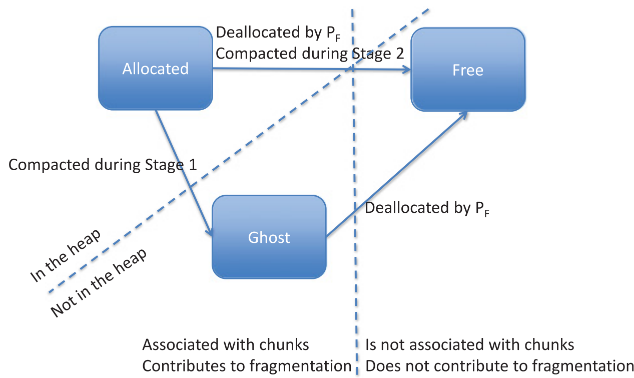

Fig. 6. Ghost objects. When objects are moved by the memory manager at Stage I, they become ghosts until they are de-allocated by Stage I of P_F that simulates the behavior of Robson's algorithm. In Stage 2, moved objects are deallocated immediately by P_F, and so ghost objects are not required.

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

The program P_F maintains a list of ghost objects along with their original locations. These are objects that have been relocated by the memory manager. For all of its deallocation considerations, P_F makes the same object deallocation decisions as Robson's original program, except that it treats ghost objects as if they still reside at their original location. In fact, each ghost object continues to exist until the original deallocation procedure of Robson's program determines that it should be deallocated. Of course, these objects were deallocated in the execution of P_F when they became ghosts, so no actual deallocation is required in the current execution. Therefore, when Robson's program deallocates them, these objects are simply removed from the list of ghost objects and are not considered further by the deallocation procedure. A diagram presenting object states is provided in Figure 6.

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

Definition 4.1 [A ghost object]. We call an object that was compacted by the memory manager during the execution and immediately deallocated by the program P_F a *ghost* object. In the first stage of the algorithm, such objects are considered by P_F as still

residing as ghosts in the original location where they were allocated. They do not affect the behavior of the memory manager, which can allocate objects to a space occupied by ghosts. When ghost objects are deallocated by the program, they disappear and are no longer considered by P_F in subsequent steps.

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

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

Definition 4.2 [an f-occupying object with respect to step i]. For each step i during the execution of P_F and an integer f satisfying $0 \leq f < 2^i$, an object is f-occupying with respect to step i if it occupies a word at address $k \cdot 2^i + f$ for some $k \in \mathbb{N}$.

As a new step kicks in, the chunk size is doubled from 2^i to 2^{i+1}, where each chunk contains two adjacent chunks of the previous step i. P_F needs to pick a new offset f_{i+1} for the larger chunks of size 2^{i+1}, that is, $0 \leq f_{i+1} \leq 2^{i+1}$. The new offset will be either the old offset on the left 2^i-sized sub-chunk, that is, $f_{i+1} = f_i$ or the old offset on the right 2^i-sized sub-chunk, that is, $f_{i+1} = f_i + 2^i$. Robson chooses the new offset in a way that wastes the most space. In a way, Robson attempts to keep the smallest objects that will still occupy words at the f_{i+1} offset. So if one of these two offsets allows capturing more space with smaller objects, this becomes the new offset f_{i+1}. To formalize this, Robson chooses f_{i+1} to be either f_i or $f_i + 2^i$, according to which one maximizes

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

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

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

The bad program P_F is presented in Algorithm 1 (on page 14). Recall that we denote the density that the program attempts to maintain in each chunk by $2^{-\gamma}$. Other inputs to P_F include M, n, which is the size of the largest allocatable object, and c, the compaction budget factor. A complete list of parameters appears in Table I.

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

Table I. Table of Notations

$HS(A, P)$	$\triangleq$	the heap size used by allocator A when servicing program P.
M	$\triangleq$	the maximum size of live objects
n	$\triangleq$	the maximum size of an object
c	$\triangleq$	the compaction bound
$\lvert o\rvert$	$\triangleq$	the size of an object o
P_F	$\triangleq$	the bad program; Algorithm 1.
A chunk at Step i	$\triangleq$	the set of memory words $\left[k \cdot 2^i, (k+1) \cdot 2^i\right)$ for some integral k
$2^{-\gamma}$	$\triangleq$	The desired density by P_F
h	$\triangleq$	the memory overhead factor. Its value is defined in Theorem 2.1
s_i	$\triangleq$	the total size of objects allocated at Stage i $(i = 1, 2)$
q_i	$\triangleq$	the total size of objects compacted at Stage i $(i = 1, 2)$
$\mathcal{D}(i)$	$\triangleq$	the set of all 2^i-aligned chunks of size 2^i
$u_D(t)$	$\triangleq$	the potential of chunk D at time t. See Definition 4.3
$u(t)$	$\triangleq$	the potential function at time t. A lower bound on the heap size. See Definition 4.4
		Terms used in Stage 1
P_R	$\triangleq$	Robson's bad program; Algorithm 2.
Ghost objects	$\triangleq$	objects compacted during the first stage of P_F and treated specially by P_F. See Definition 4.1
a f_i-occupying object	$\triangleq$	an object intersecting with word $k \cdot 2^i + f_i$ for some integral k. Used by Robson's bad program. See Definition 4.2
		Terms used in Stage 2
O_D	$\triangleq$	a set of objects associated with a chunk D. O_D contains a subset of the objects that intersect D. A chunk cannot be reused for allocation unless the objects in O_D are compacted. See Definition 4.12
$\mathcal{E}$	$\triangleq$	when three chunks are covered by 2 half objects, the middle chunk is in $\mathcal{E}$. This middle chunk is covered, even though no specific object is associated with it. See Definition 4.13
$q(o)$	$\triangleq$	objects compacted to location o. See Definition 4.15.
$x \cdot M$	$\triangleq$	the total size of memory allocated at each step of P_F's second stage. The value of x is computed at the beginning of P_F's execution.

4.1. Analysis of Program P_F

We first claim that P_F always terminates. The only loop that is not strictly bounded is the while loop at Line 17. To see that this loop terminates, we need to see that the set *ChunksToHandle* eventually becomes empty. The size of this (finite) set decreases in each iteration by one, except when Line 26 is executed, in which case the set size is incremented by one. However, the number of times Line 26 is executed is bounded. Line 26 reduces the number of half-objects associated with chunks by 2 (because it combines two half objects into a single full object). Since the number of half objects is finite, the number of times Line 26 can be executed is bounded and therefore so is the while loop.

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

The execution of P_F proceeds in steps $i = 0, 1, \ldots, 2\gamma - 1, 2\gamma, \ldots, \lg(n) - 2$. The steps $0, 1, \ldots, 2\gamma - 1$ define the first stage (however, nothing is done in steps $\gamma + 1, \ldots, 2\gamma - 1$).

The rest of the steps are executed in the second stage. At each first stage step, we use a partition of the heap into chunks of size 2^i, in an aligned manner, that is, each chunk starts at an address that is divisible by 2^i. We denote by $\mathcal{D}(i)$ the set of all aligned chunks of size 2^i.

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

During the analysis of the second stage, we will need to give special treatment to some of the chunks. The set of special chunks will be denoted by $\mathcal{E}$ and defined later in Definition 4.13. For the analysis of the first stage, one can simply think of $\mathcal{E}$ 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 in step i. Let D be a chunk of size 2^i. The function $u_D(t)$ is defined as follows:

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

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

Definition 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:

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

Let us explain the intuition behind the potential function as a sum of the $u_D(t)$ items. The malicious program P_F attempts to keep the density of each chunk to at least $2^{-\gamma}$. If it succeeded for a chunk D at time t, then we consider D to be under the control of P_F; thus, D's size (2^i) is added to the potential function. If, however, the density of the chunk is smaller than $2^{-\gamma}$, that is, the space consumed is $\mu \cdot 2^i < 2^{-\gamma} \cdot 2^i$, then we add $2^i \cdot \mu/2^{-\gamma}$ to the potential function. This estimates the success of P_F in controlling this chunk as a fraction $\mu/2^{-\gamma}$ of its length. Intuitively, P_F's target is to increase the amount of memory under its control, which reflects increasing the potential function.

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

ALGORITHM 1: Program P_F

1 Input: M,n,c,γ
2 **Initially:** Compute $x = \frac{1-2^{-\gamma}\cdot h}{\gamma+1}$
3 **During the execution:** If the memory manager compacts an object, then ask the memory manager to deallocate this object immediately (before any other action is taken) but add this object to the set of ghost objects with the same address it held when it was allocated.
4 //**Stage I:**
5 $f_0 := 0$
6 Allocate as many objects of size 1 as is possible (i.e., M such objects.)
7 **for** $i = 1$ *to* γ **do**
8   Pick f_i to be either f_{i-1} or $f_{i-1} + 2^{i-1}$, according to which of the two maximizes

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

9   Free every live or ghost object that is non f_i-occupying
10   Allocate $\left\lfloor \left(M - \sum_{o \text{ is live or ghost}} |o|\right)/2^i \right\rfloor$ objects of size 2^i
11 **end**
12 Associate objects with chunks: consider the chunk partition $\mathcal{D}(2\gamma-1)$ to chunks of size $2^{2\gamma-1}$. Each f_γ-occupying object is associated with the chunk that contains its f_γ-occupying word.
13 // **Stage II:**
14 **for** $i = 2\gamma$ *to* $\lg(n) - 2$ **do**
15   Consider the chunk partition $\mathcal{D}(i)$ 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: $O_D = O_{D_1} \cup O_{D_2}$
16   ChunksToHandle := $\mathcal{D}(i)$.
17   **while** ChunksToHandle $\neq \emptyset$ **do**
18   Pick $D \in$ ChunksToHandle
19   ChunksToHandle := ChunksToHandle $\setminus D$
20   Pick a maximal set $X \subset O_D$ such that $\sum_{o \in O_D \setminus X} |o| \geq 2^{i-\gamma}$.
21         $O_D := O_D \setminus X$
22   **foreach** $o \in X$ **do**
23   **if** *o is half an object* **then**
24   Let o' be the other half of o, let o_{full} be the entire object, and let D' be the chunk where o' resides.
25   Set $O_{D'} := O_{D'} \setminus \{o'\} \cup \{o_{full}\}$
26   ChunksToHandle := ChunksToHandle $\cup D'$.
27   **else**
28   Free o
29   **end**
30   **end**
31   **end**
32   Allocate $\lfloor x \cdot M \cdot 2^{-i-2} \rfloor$ objects of size 2^{i+2} but not exceeding the bound M on the total size of allocated memory.
33   Each allocated object o fully covers 3 chunks D_1, D_2, D_3, if it covers four, pick the first three.;
34   Let o' and o'' be the first and second halves of o, and set $O_{D_1} := \{o'\}$, $O_{D_2} := \emptyset$, $O_{D_3} := \{o''\}$.
35 **end**

It is for this reason that $u(t)$ is defined as the sum of all $u_D(t)$ minus a single $n/4$; recall that $n/4$ is the largest 2^i possible. With this additional term, $u(t)$ is guaranteed to be a lower bound on the size of the heap used. Next, we analyze the increase of $u(t)$ during the execution.

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

The guaranteed growth of $u(t)$ and the implied lower bound are shown in two lemmas, 4.5 and 4.6. Lemma 4.5 asserts the increase of $u(t)$ during the first stage and bounds from above the amount of space allocated during this stage. This bound will be used to prove the theorem. Recall that q_i (defined in the beginning of Section 4.1) represents the amount of memory compacted at stage i, and s_i represents the amount of memory allocated at stage i.

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*

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

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

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

The analysis of the second stage is summarized in Lemma 4.6. This lemma again asserts that $u(t)$ increases. However, the increase depends on the total space allocated in the second stage and also on the compaction budget in the second stage, which depends on the space allocated in both stages. Note that the allocator may have compaction budget left from Stage 1, so the amount of compaction at Stage 2 cannot be bounded as a fraction of allocation at Stage 2 only. To show that the increase in the potential function $u(t)$ is high, this lemma also bounds from below the amount of allocated space s_2 in the second stage. This second bound uses an additional parameter h, which depends on γ, c, n, and M and is set to the following complicated expression in order to achieve the strongest possible bound:

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

The parameter h is the wasted space factor; recall that M is a bound on the live space allocated simultaneously by P_F, and $h \cdot M$ is the lower bound we show on the size of the heap that the memory manager must use to satisfy the allocation requests of P_F. The analysis will show that either the memory manager uses more than $M \cdot h$ space, in which case we are done with the proof of Theorem 2.1, 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 2.1.

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,*

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

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

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

We now show how to obtain the lower bound stated in Theorem 2.1 using Lemma 4.5 and 4.6. Because the memory manager compacts at most $\frac{1}{c}$ of the total allocation, we know that $(q_1 + q_2) \leq \frac{1}{c}(s_1 + s_2)$. Thus,

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

Now, if $HS(A, P_F) \geq M \cdot h$, then we are done. Otherwise, Lemma 4.6 gives us a lower bound on s_2. The lower bound can be used here since the assumptions of Theorem 2.1 have $\gamma \leq \lg(\frac{3}{4}c)$ and so s_2 appears in a positive term in the inequality for $HS(A, P_F)$. Therefore, we use the inequality

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

In addition, Lemma 4.5 implies

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

In order to make the formula short and accessible, let us denote $\tau = \frac{\lg(n)-2\gamma-1}{\gamma+1}$ (which is one of the terms in Lemma 4.6) and $\ell = \gamma + 1 - \frac{1}{2}\sum_{i=1}^{\gamma}\frac{i}{2^i-1}$ (the term in Lemma 4.5). Now, substituting τ, ℓ in the above expressions, we may write:

$$s_2 \geq M \cdot \tau\left(1 - 2^{-\gamma} \cdot h\right) - 2n \qquad \text{(using Lemma 4.6)}$$

$$s_1 \leq M \cdot \ell \qquad \text{(using Lemma 4.5)}$$

and

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

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

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

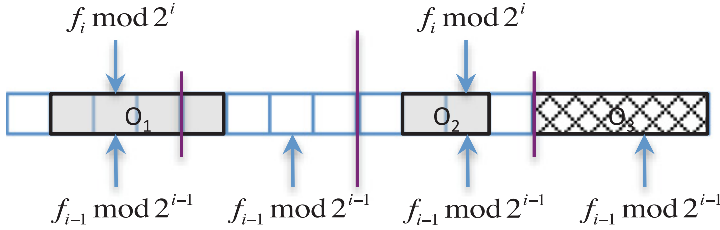

Fig. 7. Illustrating f_i-occupying objects. O_3 is f_{i-1}-occupying but not f_i-occupying. Therefore, the object O_3 will be freed in Line 4 of Algorithm 2.

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

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

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

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

a contradiction. Thus, the assumption that $HS(A, P_F) < M \cdot h$ cannot hold, and we are done with the proof of Theorem 2.1. □

4.2. Analysis of the First Stage

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

For completeness, let us recall Robson's program in Algorithm 2. In the algorithm, we use the term *f-occupying objects*, which was defined in Definition 4.2. Also, an object is *live* if it is in the heap, that is, has not been deallocated. A simple example of the behavior of Algorithm 2 is depicted in Figure 7. In this example, the object O_3 will be freed in Line 4, since it is not f_i-occupying.

When compaction occurs, P_F immediately deallocates the moved objects. But we would still like to adopt the original analysis of Robson without reanalyzing for the slightly modified version that was used in the first stage of P_F. To this end, we use a thought experiment in which we let Robson's original malicious program P_R run against an imaginary memory manager A' that does not move objects. Clearly, Robson's analysis holds for the execution (P_R, A'), as it holds for all memory managers that do not move objects. From this analysis, we will also be able to deduce a lower bound on the heap size that A uses while satisfying P_F's allocation and deallocation sequence. The only difference between the first stage of P_F and P_R is that P_F must deal with compacted objects. (Such objects are deallocated by P_F but still count for the decisions on future deallocation of all objects.) Otherwise, it behaves exactly like the original P_R.

ALGORITHM 2: Robson's "Bad" Program P_R

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

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

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

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

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

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

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

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

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

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

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

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

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

To show the first property, we need to show that an object is deallocated at Line 9 of program P_F if and only if the mapped object is deallocated at Line 4 of program P_R. Note that objects deallocated at Line 9 of P_F are permanently deleted, regardless of whether they were ghosts. An object that was not deallocated in one of these lines is either live or ghost (if it was compacted). We call objects that are not ghosts and not live *free* objects. We need to show that a non-free object is mapped to a non-free object. We combine this proof with showing that in each step the same number of objects are allocated. Since all objects allocated in Step i are of size 2^i, this provides the second property as well, that is, that the size of objects that map to each other is equal.

We now prove that a non-free object is mapped to non-free object, and the total number of objects allocated during Step i is equal in execution of (P_F, A) and (P_R, A'). The proof is by induction on the number of steps. Before Step 1, both programs allocate M objects, and there are no free objects (since P_F does not reach Line 28 and P_R does not reach Line 4). So all objects are live and of the same size, which provides a base for the induction. Next, suppose that before starting Step i, $i = 1, 2, \ldots,$ both programs allocated the same number of objects, and a non-free object is mapped to a non-free object. We show that the same holds for Step i.

In the beginning of the step execution, a new offset f_i is computed. To determine f_i, P_F considers the set of live objects, plus the set of compacted objects (in their original location) that have become ghosts but have not yet been deleted (as ghosts). By the induction hypothesis, these objects are mapped to objects that are live in the execution of (P_R, A'), and thus these are exactly the objects P_R considers when computing its new offset f_i. Moreover, by Property (3), an object is f_i-occupying if and only if its mapped object is also f_i-occupying, since by the induction hypothesis the object sizes are equal, and we also know that their addresses modulo 2^i are equal. Thus, both programs must compute the same new offset f_i. After setting the same f_i, we get that an object is non f_i-occupying if and only if the mapped object is also non f_i-occupying. Therefore, an object is freed at Line 9 of P_F if and only if the mapped object is freed at Line 4 of P_R.

Robson's program uses all extra space it has for allocation. Thus it allocates $\lfloor (M - \sum_{o \text{ is live}} |o|)/2^i \rfloor$ objects of size 2^i. Meanwhile, P_F allocates

$\lfloor (M - \sum_{o \text{ is live or ghost}} |o|)/2^i \rfloor$ such objects. But every live object in A' is mapped to a live or ghost object in A, so they allocate exactly the same number of objects, and we are done. □

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

CLAIM 4.9 (ROBSON [1974], INEQUALITY 1). *After the execution of Step i of P_R, there are at least $M\frac{i+1+1}{2^{i+1}} = M\frac{i+2}{2 \cdot 2^i}$ objects that are f_i-occupying.*

We proceed with proving the first part of Lemma 4.5 by using Claim 4.9 and the mapping of Claim 4.8. Let us first recall the definition of the potential function. By Definition 4.3:

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

Note that we ignore the members of $\mathcal{E}$ in Definition 4.3. This is because $\mathcal{E}$ is the empty set at the end of the first stage. By Definition 4.4 the potential function is

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

Recall also how f_γ-occupying objects are determined (Definition 4.2). For an integer f_γ satisfying $0 \leq f_\gamma < 2^i$ (computed by P_F), an object is f_γ-occupying if it occupies a word at address $k \cdot 2^\gamma + f_\gamma$ for some $k \in \mathbb{N}$. Finally, recall that t_{first} is the time immediately after executing Step 12. For a chunk D, $O_D(t)$ is the set of object that P_F associates with the chunk. In the first stage, association is determined by the program P_F in Step 12, where P_F associates an f_γ-occupying object with the chunk that contains its f_γ-occupying word. Thus, the number of f_γ-occupying objects is related to $O_D(t)$, which is related to the value of the potential function at time t_{first}. This relation is next used to transfer Robson's guarantee for many f_γ-occupying objects into a lower bound on the potential function.

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

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

PROOF. Consider the time when Robson's program P_R finished executing the first γ step. By Claim 4.9, there are at least $M\frac{\gamma+2}{2 \cdot 2^\gamma}$ live objects that are f_γ-occupying. Let t_{first} be the time immediately after P_F finished executing its first γ steps. By Claim 4.8, there exists a mapping between objects in the execution of P_R and P_F. Moreover, an object is f_γ-occupying in the execution of P_R if and only if the mapped object is f_γ-occupying in the execution of P_F. Thus, there are at least $M\frac{\gamma+2}{2 \cdot 2^\gamma}$ live or ghost objects that are f_γ-occupying at time t_{first} as well. Recall that q_1 is the total size of objects compacted during P_F's first stage. Therefore, the number of objects compacted cannot exceed q_1, and therefore the number of ghost objects is at most q_1. Thus, at the end of step γ, the number of *live* (non-ghost) f_γ-occupying objects is at least $M\frac{\gamma+2}{2 \cdot 2^\gamma} - q_1$.

Next, consider the partition of the heap into chunks of size $2^{2\gamma-1}$, and let D be a chunk in the partition. According to Step 12 of Algorithm P_F, the set O_D of the objects associated with chunk D is the set of live objects whose f_γ-occupying word falls inside

D; there are at most $2^{\gamma-1}$ such objects since there are at most $2^{\gamma-1}$ words in D whose address equals f_γ moduli 2^γ. According to Definition 4.3, the value of $u_D(t_{first})$ is the cumulative size of objects in O_D multiplied by 2^γ; this value is capped by $2^{2\gamma-1}$ in Step $2\gamma - 1$. Since the size of each associated object is at least 1, we have for each chunk D

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

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

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

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

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

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

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

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

PROOF. Recall that by Claim 4.8, the number of objects allocated by P_F (when working with the memory manager A) at each step equals the number of objects allocated by P_R at that step, when working with the modified memory manager A'. Thus, it is enough to bound the total size of objects allocated during the execution of P_R against $A'(A, P)$. In Step 0, P_R allocates M words. In Step i, P_R allocates $\lfloor (M - \sum_{o \text{ is live}} |o|)/2^i \rfloor$ objects, and the total allocated space is at most

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

Let x_i be the number of objects that P_R allocated in Step i. The size of each object is 2^i so the total space allocated is $2^i \cdot x_i$. By Claim 4.9, at least $M\frac{i+2}{2 \cdot 2^i}$ objects will be f_i-occupying live objects after the allocation of Step i. Exactly x_i of these are newly allocated objects (objects allocated during Step i) and the rest are old live objects. Therefore, before P_R starts allocating objects in Step i, the total number of live objects is at least $M \cdot \frac{i+2}{2 \cdot 2^i} - x_i$, and clearly the total space they span is no smaller than that. If old objects take at least $M \cdot \frac{i+2}{2 \cdot 2^i} - x_i$ words, then the space available for allocation in Step i is at most:

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

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

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

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

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

as required. □

The proof of Lemma 4.5 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} the time the second stage (and the entire algorithm) completes.

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

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

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

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

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

Recall that the potential function is defined as (Definition 4.4)

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

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

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

Definition 4.12. Let A be a c-partial memory manager, let t be a time during the execution of P_F second stage, and let i be the step in which t occurs. Let D be a 2^i chunk, and let O_D be the set of objects and half objects that P_F associates with the chunk D.

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

—P_F Line 12 (before the second stage started)
—P_F Line 34 (a new object is allocated).

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

—P_F Line 21 (P_F frees an object).
—P_F Line 34. This case occurs when an object o' is associated with a chunk D at time $t-1$. If at time t a new object o is allocated and placed on the chunk D, then the association set of D is changed to contain only o, and o' is deleted.

Compaction of an object *does not* de-associate it. Even thought the object is not live (because of P_F Line 3), it is still associated with the original chunk until P_F removes the association at Line 34.

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

—P_F Line 34. A new object is allocated.
—P_F Line 21. P_F frees an object.
—P_F Line 14. In this case a new step, i, kicks in, and two adjacent 2^{i-1} chunks are merge into one 2^i chunk. Objects never "disappear" during the merge; rather, the two association sets of the 2^{i-1} chunks are merged into a single association set.

We start by looking at allocations and show that whenever P_F allocates an object, either the potential function gets larger or some compaction occurs (and the potential function does not decrease). (We will later use the fact that compaction is limited to get the potential function growth we need.) Suppose an object o is allocated during Step i of the execution of P_F with A. By the definition of P_F, the size of o is $4 \cdot 2^i$. Thus, when the memory manager places o in the heap, it consumes at least three full consecutive chunks (of size 2^i) and then some additional space from the chunks neighboring these. Denote by D_1, D_2, and D_3 the three chunks that are fully covered by o and are selected in Step 33 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 $\mathcal{E}$ (which we define in the next paragraph) only changes for D_1, D_2, and D_3.

When an object o is allocated by P_F in Line 34, P_F associates D_1 with the first half of o and D_3 with the second half of o. But since an object is associated with at most two chunks (each half can be associated with a chunk), it follows that D_2 is left with no associated object, despite it being completely covered by the allocated object o. The goal of the set $\mathcal{E}$ 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 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 $\mathcal{E}$.

Definition 4.13 [The set $\mathcal{E}(t)$]. 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* $\mathcal{E}(t) \subset \mathcal{D}(i)$ is the set of chunks whose left adjacent chunk and right adjacent chunk were fully covered by an object o

allocated in Step i (thus, the chunk itself is also covered by o), but half of o was not associated with it. A chunk remains in $\mathcal{E}(t)$ until either a new step kicks in or an object is associated with this chunk. The latter may occur if o was compacted during step i, and another object is allocated there.

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

Claim 4.14. *Let t be a time during the execution of P_F against a memory manager A, and let i be the step in which t occurs. Let D_1 and D_2 be two consecutive chunks such that $D_1 \in \mathcal{E}(t)$. 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 $D_2 \in \mathcal{E}(t)$.*

Proof. Let t' be the time when D_1 joined the set $\mathcal{E}(t)$, and let o' be the object whose allocation caused D_1 to enter $\mathcal{E}$. By definition of P_F in step 32, half of o' was associated with D_2 at time t'. If this is still the case at time t, then we are done. Otherwise, let o'' be the last object that changed the association of D_2 before time t (which occurs in Step i). If half of o'' was associated with D_2, then we are done. Otherwise, o'' was associated with both chunks that neighbored D_2, which means that half of o'' was associated with D_1, contradicting the assumption that $D_1 \in \mathcal{E}$. It is easy to see that a similar argument holds for D_1 in the case where $D_2 \in \mathcal{E}(t)$. □

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

When the second case occurs, that is, the memory manager compacts away objects from a chunk before placing o, we are not able to show that the value of the potential function grows. However, we get that some compaction occurred, and we use that to bound the number of such events. To this end, we associate some compaction value with the newly allocated object. Note that the objects that were compacted away from a chunk and then deallocated immediately by P_F are still considered associated with the chunk until a new object is placed on it. Recall (Definition 4.12) that associations of objects with chunks are managed explicitly by P_F, and association changes only at Line 14 (phase change), Line 21 (P_F frees objects), and Line 34 (a new object). An association terminates only when P_F frees the object in Line 14 or a new object is placed on top of its former location (and a new association kicks in) at Line 34. 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 4.15 [Compaction space associated with an object]. Let o be an object allocated during the execution of P_F's second stage against a memory manager A. Let $t+1$ be the allocation time (and t be the time just before the allocation), and let D_1, D_2, and D_3 be the chunks picked by P_F in Step 33 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 4.16. *Consider any point of time in step i of the execution of P_F's second stage against a memory manager A. Then the following three properties hold:*

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

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

As for the second property, we show that if an object is not associated with any chunk, then the object is not live. First, note that when an object is allocated, it is associated (via its two halves) with two chunks. An object is de-associated only if either P_F frees this object (in steps 21 and 28) or another object is allocated and is associated with the same chunk with which the previous object was associated (in step 32). In the latter case, the object cannot reside on the same chunk where the new object resides. Thus, it was compacted by the memory manager and deallocated by P_F immediately. In both cases, P_F frees the object, so it is not live.

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

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

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

Definition 4.3.

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

Definition 4.4. $u(t) = (\sum_{D \in \mathcal{D}(i)} u_D(t)) - \frac{n}{4}$.

CLAIM 4.17. *Let A be a memory manager, let $u(t)$ be the potential function as defined in Definition 4.4, and let $q(o)$ be the compaction associated with an object as defined in Definition 4.15. Then, during the execution of P_F against A, the following properties of $u(t)$ hold:*

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

PROOF. The potential function only changes when the set of chunks changes during step transition or when the set of associated objects of a chunk changes. By Definition 4.12, there are three types of events that may cause such a change. Note that compaction of an object by A is not an event that influences the potential function, since the object association remains unchanged.

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

If either $D_1, D_2 \in \mathcal{E}(t-1)$, then by Claim 4.14 there exists an object o that was allocated in Step $i-1$, and half of o is associated with either D_1 or D_2. Objects allocated in Step $i-1$ are of size 2^{i+1}. Since half of o is associated with D, and the size of half of o is 2^i, it holds that $\sum_{o' \in O_D} |o'| \geq 2^i$. Also note that, by Definition 4.13, $D \notin \mathcal{E}$ since chunks are inserted to $\mathcal{E}$ only after the allocation step. Thus, by Definition 4.3 of $u_D(t)$,

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

The last inequality follows since, by Definition 4.3, $u_{D'}(t) \leq 2^{i-1}$ for every chunk D' in step $i-1$. Otherwise, both chunks are not in $\mathcal{E}$, and we know that the objects associated with them are disjoint. Therefore, according to Step 14 in P_F,

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

In this case, the statement follows since

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

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

P_F deallocates an object. By definition of P_F Step 32, P_F does not free an object from a chunk D if the deletion decreases $\sum_{o \in O_D} |o|$ below $2^{i-\gamma}$. Let t be a time (in step i) when an object was deleted from a chunk D. Thus, $\sum_{o \in O_D} |o| \geq 2^{i-\gamma}$, which implies that $u_D(t) \geq 2^i$. Since 2^i is the maximal value of $u_D(t)$ in step i, it was not higher before the deletion, and the claim holds in this case.

P_F allocates an object o. Let t be the time P_F allocates an object o, and let D_1, D_2, D_3 be the three chunks that P_F picked. By definition of P_F Step 34, only the sets $O_{D_1}, O_{D_2}, O_{D_3}$ are changed during allocation. To show that the value of $u(t)$ increases by $\frac{3}{4}|o| - 2^\gamma q(o)$ after the allocation of o, it is sufficient to show that $u_{D_1}(t) + u_{D_2}(t) + u_{D_3}(t) - (u_{D_1}(t-1) + u_{D_2}(t-1) + u_{D_3}(t-1)) \geq \frac{3}{4}|o| - 2^\gamma q(o)$.

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

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

Next we show that $2^\gamma \cdot q(o) \geq u_{D_1}(t-1) + u_{D_2}(t-1) + u_{D_3}(t-1)$, which finishes the proof for the allocation case. Recall that, by Definition 4.15, we have

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

Before the allocation, if either chunk was contained in $\mathcal{E}(t-1)$, then by Claim 4.14 there exists an object o'' allocated in step i, and half of o'' is associated with D_1, D_2, or D_3. Since o'' was allocated at Step i, the size of half of o'' is 2^{i+1}. In this case, $q(o) \geq 2^{i+1}$, and $2^\gamma \cdot q(o) \geq 2^\gamma \cdot 2^{i+1} \geq 3 \cdot 2^i \geq u_{D_1}(t-1) + u_{D_2}(t-1) + u_{D_3}(t-1)$. The second inequality follows since $\gamma \geq 1$, and the last inequality follows since, by Definition 4.3, $u_D(t) \leq 2^i$ for every chunk D.

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

We now turn to bound the memory that P_F allocates during the execution of its second stage. By its definition, in Step 32, P_F attempts to allocate $\lfloor x \cdot M \cdot 2^{-i-2} \rfloor$ objects of size 2^{i+2} without exceeding the M bound for the total size of allocated memory. In Lemma 4.6 the growth in potential function depends positively on the memory allocated. Thus, we would like to show that P_F allocates a lot of memory. To this end, we need to maximize x but also show that there is enough allocation budget so the bound of M simultaneously live words does not hinder allocation. Namely, we need to show that there is enough allocation "budget" or that the space occupied by live objects in the heap is bounded from above.

The next proposition pertains to the shape of chunks. It asserts that any chunk D in the heap is either empty, it contains a single (large) object, or the total space of its associated objects O_D is bounded exactly by $2^i/2^\gamma$, which is the density that P_F attempts to preserve in each chunk. This proposition will later be used to bound the total size of live objects in the heap. Before stating the proposition we recall Lines 20 and 21 of the program P_F. These lines stand at the heart of the proposition.

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

PROPOSITION 4.18. *Consider the execution of P_F's second stage against a memory manager A. Let t be a time in step i when P_F allocates an object. Then for every chunk $D \in \mathcal{D}(i)$, either $|O_D(t)| \leq 1$ or $\sum_{o \in O_D(t)} |o| \leq 2^{i-\gamma}$.*

PROOF. Objects are allocated only in Lines 32–34 of P_F, so it is sufficient to prove the proposition during the execution of this line. Fix the time t' after P_F finished the execution of Line 31 in Step i. We show that the Proposition holds at time t', and it does not change during Lines 32–34 of P_F.

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

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

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

Finally, we show that the execution of Lines 32–34 of P_F does not break the guarantee of the proposition. If the set $O_D(t)$ changed during the execution of Line 34, then at most a single half object is associated with the chunk D. Therefore, the Proposition holds at any time t during execution of Line 34, as required. □

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

CLAIM 4.19. *Consider the execution of P_F against a memory manager A. Then either A uses more than $M \cdot h$ heap space or the number of words that P_F allocates during its second stage satisfies*

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

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

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

words during the execution of the step, where x is the algorithm constant that is determined at the beginning of Algorithm 1 (P_F) to be

$$x = \frac{1 - h \cdot 2^{-\gamma}}{\gamma + 1}. \tag{4}$$

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

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

Assume that A uses less than $M \cdot h$ heap space, and let us show that allocation of $\lfloor M \cdot x \cdot 2^{-i-2} \rfloor$ chunks of size 2^{i+2} is possible, in the sense that it does not exceed the total of M live words. In fact, we will show that it is possible to allocate $M \cdot x$ words, which is enough. Recall that Claim 4.16 states that every live object is associated with a chunk. Thus, it is sufficient to show that, after allocation of $M \cdot x$ words, the total size of objects associated with some chunk is bounded by M.

We consider chunks according to the space consumed by their associated objects. First, consider chunks whose associated objects (and half-objects) are small, that is, at most $2^{i-\gamma}$ words. By Proposition 4.18, the total space of objects associated with such a chunk is at most $2^{i-\gamma}$.

Next consider chunks with large associated objects of size at least $2^{i-\gamma+1}$ and at most 2^{i+2}. By Proposition 4.18, if a non-small object is associated with a chunk, then this object is the only object associated with this chunk. So to bound the space such objects take, we simply bound the maximum possible space allocated by all associated large objects. Since each stage has its object size, then large objects are objects that have been allocated recently. More specifically, an object's size is at least $2^{i-\gamma+1}$ if it was allocated in Step $i - \gamma - 1$ or thereafter. We will have the maximum number of such associated objects if all possible objects are allocated in each of the steps $j = i - \gamma - 1, \ldots, i$, and all of them are still associated with chunks at the time we consider the chunks. In each of these steps, at most $\lfloor M \cdot x \cdot 2^{-j-2} \rfloor$ objects of size 2^{j+2} are allocated, and at this point

(in Step i), each of them can be associated with a single chunk, or two half-objects can be associated with two chunks.

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

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

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

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

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

Finally, setting the value of x as in Equation (4), we get M, as required. □

Recall that for the proof of Lemma 4.6, we want to show that $u(t)$ increases substantially. Claim 4.17 asserts an increase of $u(t)$ by at least $\frac{3}{4}|o| - 2^{\gamma} \cdot q(o)$ for each allocation of an object o. Summing $\frac{3}{4}|o|$ over all objects allocated during stage two of P_F gives $\frac{3}{4}s_2$, which we also computed. To get a lower bound on how much $u(t)$ increases, we need to lower bound the sum of $q(o)$ over all objects allocated during stage two of P_F. (For the definition of $q(o)$, see Definition 4.15.)

PROPOSITION 4.20. *Consider the execution of P_F against a memory manager A. Let S_2 be the set of objects allocated at P_F's second stage, and let q_2 be the total size of objects compacted during the second stage. Then $\sum_{o \in S_2} q(o) \leq q_2$.*

PROOF. Consider an object $o \in S_2$ allocated at time $t + 1$, and let D_1, D_2, D_3 be the chunks that P_F picked at Line 33 after allocating o. Let o' be an object associated with one of the chunks D_1, D_2, D_3 at time t (just before o was allocated). By Claim 4.16, o' intersects one of these chunks D_1, D_2, D_3. Since o covers all these chunks, o' intersects o, and o' cannot remain in its original location at time $t + 1$. Also, o' was not deallocated at Line 28, since otherwise it would no longer be associated with any chunk. Thus, o' was compacted by A before o was placed and so o' is counted with $q(o)$. Moreover, at time $t + 1$, the object o' is not associated with any chunk (which follows by definition of P_F's Line 34). Therefore, o' is not considered in the summation $\sum_{o \in S_2} q(o)$ ever again (under $q(o'')$ of any other object o''). Thus, the total size of compaction A performed during second stage is at least $\sum_{o \in S_2} q(o)$. □

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

CLAIM 4.21. *Consider the execution of P_F against a memory manager A. Let t_{finish} be the time when P_F finished its execution, and let t_{first} be the time when P_F finished*

execution of first stage. Then

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

PROOF. By Claim 4.17, $u(t)$ does not decrease during the execution of P_F's second stage. Furthermore, during allocation of an object $o \in S_2$, $u(t)$ increases by at least $\frac{3}{4}|o| - 2^{\gamma} \cdot q(o)$. Thus,

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

The last equality follows from the definition of s_2 (total memory allocated at stage 2) and Proposition 4.20. □

The proof of Lemma 4.6 follows from Claim 4.19 and Claim 4.21.

5. UPPER BOUND: PROOF OF THEOREM 2.2

In this section, we provide an upper bound on fragmentation that improves over the best known upper bound for a memory manager that is c-partial. The upper bound is stated as Theorem 2.2. To prove Theorem 2.2, we present a c-partial memory manager A_C that keeps the heap size small.

THEOREM 2.2. *There exists a c-partial memory manager $A \in \mathcal{A}(c)$ that satisfies allocation requests of any program $P \in \mathcal{P}(M, n)$ with heap size at most*

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

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

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

Recall that M denotes the bound on the space that the program may use simultaneously, n is the maximum object size, and c is the compaction bound. Since the expression in Theorem 2.2 is not easy to digest, we presented a graph of the upper bound as a function of c in Figure 3. For example, if $n = 1$MB, $M = 256n$, and $c = 100$, the upper bound on the heap size is $19.53M$. Namely, our allocator could serve the requests of any appropriate program with a heap of size $19.53M$. If we change c to be 50, then the upper bound becomes 19.00, and for $c = 20$, the upper bound is $17.32M$[2].

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

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

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

all allocated objects are the same size. However, this yields a memory size bound of $2M(\lg(n) + 1)$, while our goal is to use partial compaction and be able to work with a smaller heap.

The allocator we present will allocate an object o of size $2^{i-1} < |o| \leq 2^i$ in any area j satisfying $j \leq i$. Note that we do not use padding in the actual algorithm. When an allocation request is received for an object o of size $2^{i-1} < |o| \leq 2^i$, the proposed memory manager will simply search for a free aligned location of size 2^i in all areas $j = 0, 1, \ldots, i$. If no 2^i-aligned space is free, then the heap must be sparsely populated. The analysis will show that in this case there must be a 2^i-aligned chunk in which only $1/(2c + 2)$ of the words are populated. In this case, the memory manager will compact the content of this chunk, recursively using the allocator on the compacted objects to find vacant places for them. The evacuated chunk is then used for satisfying the requested allocation. In essence, this is the algorithm we use. The choice of sizes for the area used to hold each object size, and the analysis showing that with such sizes everything fits well, are crucial.

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

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

$$a_i = \frac{2^{i+1}}{2^i + 1} \cdot \left(1 - \sum_{j=0}^{i-1} \max\left(\frac{1}{2c+2}, \frac{\lfloor 2^{j-1} + 1 \rfloor}{2^i}\right) \cdot a_j\right). \tag{5}$$

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

Definition 5.1 (*A 2^i-aligned Chunk*). Let S be a memory area. A 2^i-aligned chunk is a chunk of memory of size 2^i starting at an address divisible by 2^i.

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

Definition 5.2 (*Uncrossed Chunk*). Let S be a memory space, and let D be a 2^i-aligned chunk fully contained in S. Let D_{prev} and D_{next} be the two neighboring 2^i-aligned chunks of D. We call D is *uncrossed* if no object intersects with both D and one of the neighbors of D.

The intuition described above is formalized in Algorithm 3. This algorithm receives an allocation request (a size) from the program and it determines an area for allocation of this object.

In what follows, we prove the correctness of this memory manager, but let us first analyze the heap size it requires. Assuming a reasonable implementation in which

ALGORITHM 3: Memory Manager A_C

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

the areas are placed one after the other consecutively, the heap size required by this algorithm is simply the sum of the areas it uses. This is stated in the next claim.

CLAIM 5.3. *For any program* $P \in \mathcal{P}(M, n)$*, the size of memory* A_C *uses when serving* P *is at most*

$$HS(A_C, P) \le M \cdot \left(\sum_{i=0}^{\lg n} a_i \right) + n(\lg(n) + 1). \quad (6)$$

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

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

The claim follows. □

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

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

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

CLAIM 5.4. *For any program* $P \in \mathcal{P}(M, n)$ *and any allocation request of size* $|o|$*, the memory manager* A_C *will relocate objects of accumulating size that does not exceed* $|o|/c$.

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

Since $a_0 = 1$, then the size of the first area is (at least) M. Since the total size of live memory in this area (which exclude the newly allocated object o) is at most $M-1$, there must be at least one empty chunk of size 1. Therefore, no compaction is needed and we are done with the base of the induction.

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

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

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

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

CLAIM 5.5. *For each pair of integers i, j such that $0 \leq j \leq i$, let Area S be a memory region of size divisible by 2^i, with allocated objects on it that satisfy the following conditions:*

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

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

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

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

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

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

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

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

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

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

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

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

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

as required. □

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

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

CLAIM 5.6. *Upon an allocation request of size $2^{i-1} < |o| \leq 2^i$ by a program $P \in \mathcal{P}(M, n)$ of the memory manager A_C, there exists a chunk of size 2^i that is either vacant or uncrossed and whose density is less than $\frac{1}{2c+2}$. The claim holds for an allocation request made by a program $P \in \mathcal{P}(M, n)$ or recursive calls made by A_C while serving such a program P.*

PROOF. Assume by way of contradiction that there exists neither empty nor uncrossed chunks with density $< \frac{1}{2c+2}$. We will show that under this assumption, the total size of live memory, including the newly allocated object o, exceeds M.

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

Next, let $i \geq 1$ and consider a direct call to allocation (as oppose to recursive calls made by the allocator itself). To show a contradiction, we use Claim 5.5 to compute the accumulating sum of live memory that must be located in Area j for every $j \leq i$. We then sum up over the areas to show that the total size of live memory exceeds M.

Consider an area j for $j \leq i$. By definition of A_C, the size of each area a_i is divisible by n, and Properties (1) and (2) of Claim 5.5 hold. By our assumption, Property 3 also holds. Thus, the amount of live space at area $j < i$ is at least $\max(\frac{1}{2c+2}, \frac{\lfloor 2^{j-1}+1 \rfloor}{2^i})a_j$, and the amount of live space at area i is at least $\frac{2^i+1}{2^{i+1}} \cdot a_i$.

By definition of a_i, the following holds (even with equality):

$$\frac{2^i+1}{2^{i+1}} \cdot a_i + \sum_{j=0}^{i-1} \max\left(\frac{1}{2c+2}, \frac{\lfloor 2^{j-1}+1 \rfloor}{2^i}\right) \cdot a_j \geq 1. \tag{7}$$

The above arguments, in addition to multiplying the last equation by M, prove that the total size of live memory in areas $0, 1, \ldots, i$ is at least M. Contradiction follows since the newly allocated object o is also alive, and thus the total size of live objects exceeds M.

Finally, consider an execution of A_C that recursively calls A_C at Line 7. We denote the callee execution of A_C by E_{callee} and the caller execution of A_C by E_{caller}. Let o be the object that E_{caller} attempts to allocate and let D be the chunk that E_{caller} picked at Line 5, which is compacted to make room for o. Finally, let o' be the object that E_{callee} attempts to allocate in order to empty D. We want to show that E_{callee}, the recursive execution of A_C, successfully places o'. If $|o'| = 1$, then the same argument as for the non-recursive case shows that A_C will successfully place o' (there is always room for an object of size 1).

Next, consider $|o'| > 1$. We start with some intuition. We would like to apply Claim 5.5, to show that E_{callee} succeed in a similar manner to the non-recursive case. However, we cannot apply Claim 5.5 directly since we must reserve D for o, and so we cannot consider any empty location as a candidate for o'. To solve this issue (and be able to apply Claim 5.5), we consider an imaginary heap in which o is already placed in chunk D, and there is a new allocation request to place o'. Consider a snapshot of the real heap just before o is allocated and consider the following changes to the snapshot heap: removing all objects that reside on D and place o there. These modifications of the heap create an imaginary heap (that never existed in the execution). But these changes do not break conditions (1)–(3) of Claim 5.5.

Now apply Claim 5.5 on this imaginary heap and attempt to allocate o' in it. As in the non-recursive argument, we get that the total live space is at least M. Since we applied Claim 5.5 on the imaginary heap, this counts o but does not count o' (and all other objects that resided on D). Since both o and o' are alive after the allocation, the total live memory must be at least $M + |o'|$, which yields a contradiction to the assumption that there is no empty nor uncrossed chunks with density $< \frac{1}{2c+2}$, and Claim 5.6 follows. □

Claim 5.6 completes the correctness proof of the memory manager. Together with Lemma 5.3, this concludes the proof of Theorem 2.2.

6. MEASUREMENTS, DISCUSSION, AND FUTURE WORK

While this work advances our knowledge about partial compaction significantly, it still leaves a substantial gap between the obtained upper and lower bounds. As an interesting anecdotal test, we ran the bad program P_F from Section 4 against the good allocator A_C from Section 5. While such an experiment provides no rigorous information, it may shed some light on the direction in which this gap can be narrowed. We ran both programs with the following parameters: $M = 256$MB, $n = 1$MB, and $c = 50$. The result of this execution yielded a memory usage of $4.85M$. For this case, our proof shows an upper bound of $19M$ and a lower bound of $3.15M$. In what follows, we discuss these results.

The first thing to note is that in the experiment A_C performed no compaction. The reason is that A_C actually compacts objects only if the density of a chunk drops below $\frac{1}{2c+2}$. But P_F never lets this happen. It never frees objects if the result is a chunk

with density smaller than $2^{-\gamma} \geq \frac{4}{3c} > \frac{1}{2c+2}$. This drawback of A_C may indicate it can be improved by developing an algorithm that is able to compact memory even if a chunk density is higher than $\frac{1}{c}$. Such an allocator would be more complex than our A_C because the allocator cannot consistently compact such locations. It does not have enough budget for that.

The lack of compaction makes the first stage of P_F more effective than the second stage. Recall that the first stage uses Robson's program that ignores compaction while the second stage tries to handle compaction while still creating fragmentation. Indeed, the experiment shows that the first phase alone made the allocator use a space of $3.5M$ fragmentation, and the second phase only increased it by $1.35M$.

Another inaccuracy that builds the gap between the upper and lower bound is aligned allocation. The allocator A_C places objects in an aligned manner; that is, it puts an object of size 2^i in an address $k \cdot 2^i$ for some $k \in \mathbb{N}$. However, the bad program P_F does not use this "weakness" of the allocator. P_F attempts to handle the general case in which allocation is not aligned, thus allocating objects that are 4 times larger than a chunk and considering only three-quarters of the object that spans a full chunk. Closing the gap would also require building an allocator that places objects in non-aligned location to neutralize the bad behavior of a program and provide a better upper bound. Alternatively, maybe one can show that running a bad program against a non-aligned allocator is equivalent (up to a small factor) to running the program against an aligned allocator. We do not know how to show that.

Finally, we noticed that when we increased the value of x in P_F (Line 2), we got better results (more fragmentation). In this case, the heap usage was increased to $5.56M$. However, we do not know how to analyze an execution with such an x. Some of the iterations created a lot of fragmentation while other created almost zero fragmentation. We do not know how an increase of x will behave when P_F is run against a better (or optimal) allocator.

7. CONCLUSION

This work contributes to building a solid theoretical foundation for memory management by extending previous work to providing new lower and upper bounds on the effectiveness of partial compaction. Our lower bound is the first bound in the literature with implications for existing systems because it shows that some desirable (realistic) compaction goals cannot be achieved.

APPENDIX

A. A SIMPLER RELAXED UPPER BOUND EXPRESSION

The results of Claim 5.3 provide an upper bound on heap size that A_C uses. The upper bound is presented as a sum over a recursive series a_i. In this section, we show how to approximate this upper bound by an explicit expression (with no recursion).

We change the area sizes that A_C uses to a new series a'_i, which is simpler to analyze. We first show that the correctness proof of A_C still holds. Next, we show that the new series a'_i can be bounded by a geometric series, and so can be stated explicitly (and not only in a recursive form).

Define a'_i to be the recursive series where $a'_0 = 1$ and for $i \geq 1$,

$$a'_i = 2 - \sum_{j=1}^{i-1} \max\left(\frac{1}{c+1}, 2^{j-i}\right) \cdot a'_j - \max\left(\frac{1}{c+1}, 2^{-i+1}\right) \cdot a'_o. \tag{8}$$

The explicit area sizes a_i appear in the correctness proof of A_C only in Equation (7), in the proof of Claim 5.6. Thus, it is sufficient to show that our new series satisfies

$$\frac{2^i+1}{2^{i+1}} \cdot a'_i + \sum_{j=0}^{i-1} \max\left(\frac{1}{2c+2}, \frac{\lfloor 2^{j-1}+1 \rfloor}{2^i}\right) \cdot a'_j \geq 1.$$

If we substitute the new area sizes into Claim 5.6, then the correctness proof of A_C that uses area sizes a'_i follows.

Rearranging the terms in Equation (8) by moving all terms (except the 2) to the left side and dividing by 2, we get

$$\frac{1}{2}a'_i + \sum_{j=1}^{i-1} \frac{1}{2} \max\left(\frac{1}{c+1}, 2^{j-i}\right) \cdot a'_j + \frac{1}{2} \max\left(\frac{1}{c+1}, 2^{-i+1}\right) \cdot a'_o = 1.$$

Also, by Proposition A.2 below, we prove that a'_i can be rewrite as a positive sum of $a'_j : j < i$. Thus, a_i is always positive as it is a sum of positive terms. Now, Inequality (7) for the series a' follows:

$$\begin{aligned}
&\frac{2^i+1}{2^{i+1}} \cdot a'_i + \sum_{j=0}^{i-1} \max\left(\frac{1}{2c+2}, \frac{\lfloor 2^{j-1}+1 \rfloor}{2^i}\right) \cdot a'_j \\
&= \frac{2^i+1}{2^{i+1}} \cdot a'_i + \sum_{j=1}^{i-1} \max\left(\frac{1}{2c+2}, \frac{2^{j-1}+1}{2^i}\right) \cdot a'_j + \max\left(\frac{1}{2c+2}, \frac{1}{2^i}\right) \cdot a'_0 \\
&\geq \frac{1}{2} \cdot a'_i + \sum_{j=1}^{i-1} \max\left(\frac{1}{2c+2}, \frac{2^{j-1}}{2^i}\right) \cdot a'_j + \max\left(\frac{1}{2c+2}, \frac{1}{2^i}\right) \cdot a'_0 \\
&= \frac{1}{2}a'_i + \sum_{j=1}^{i-1} \frac{1}{2} \max\left(\frac{1}{c+1}, 2^{j-i}\right) \cdot a'_j + \frac{1}{2} \max\left(\frac{1}{c+1}, 2^{-i+1}\right) \cdot a'_o \\
&= 1.
\end{aligned}$$

The inequality at the third line follows by the fact that a'_i are positive. The equality at the fifth line follows by Equation (8) rearranged.

To keep the mathematical analysis clear, we assume that $c+1 = 2^k$ for some integral k. If c does not satisfy this assumption, then rounding it up only decrease the compaction budget available to the memory manager.

CLAIM A.1. *Let A_C be the memory manager from Algorithm 3 that uses area sizes a'_i. Then for any program $P \in \mathcal{P}(M, n)$, the heap size used by A_C is bounded by*

$$\begin{aligned}
HS(A_C, P) \;&\leq\; k + 1 + \frac{1 - \left(1 - 2^{-k-1}\right)^{\lg n - k}}{1 - \left(1 - 2^{-k-1}\right)} M \\
&\leq\; \left(\lg n + 1 - \frac{\lg^2(n/(c+1))}{4c+4} + \frac{\lg^3(n/(c+1))}{6(2c+2)^2}\right) M.
\end{aligned}$$

We start by bounding area sizes. We first consider area sizes for $i \leq k$ and show that $a'_i = 1 : i \leq k$. For $i = 0$, the equality $a'_0 = 1$ follows by the series a' definition. For $0 < i \leq k$, we show $a'_i = 1$ by induction. Let the induction basis be $i = 0$, and for the

induction step suppose that $a'_j = 1$ for all $j < i$. Thus

$$a'_i = 2 - \left(\sum_{j=1}^{i-1} 2^{j-i} \cdot 1 \right) - 2^{-(i-1)} = 2 - \left(\sum_{j=1}^{i-1} 2^{-j} \right) - 2^{-(i-1)} = 1.$$

The first equality follows by area a' definition and the induction assumption. The second equality arranges the summation terms.

Bounding area sizes for $i > k$ is harder. We start by expressing the series a'_i as a simpler recursive series that contains only additive terms. Recursive series that use only additive terms are easier to reason about since they depend positively on previous terms.

Proposition A.2. *Let a'_i be the series from Equation (8), and let $f(a'_{i-1}, a'_{i-2}, \ldots, a'_{i-k}) = \sum_{j=i-k}^{i-1} 2^{j-i} \cdot a'_j$. Then for $i > k$, the following holds:*

$$a'_i = f(a'_{i-1}, a'_{i-2}, \ldots, a'_{i-k}) = \sum_{j=i-k}^{i-1} 2^{j-i} \cdot a'_j.$$

Proof. We start with the following auxiliary equation:

$$\begin{aligned}
&a'_i - a'_{i-1} \\
&= \left(2 - \sum_{j=0}^{i-1} \max(2^{-k}, 2^{j-i}) \cdot a'_j - 2^{-k} \cdot a'_0 \right) - \left(2 - \sum_{j=0}^{i-2} \max(2^{-k}, 2^{j-i+1}) \cdot a'_j - 2^{-k} \cdot a'_0 \right) \\
&= - \sum_{j=i-k}^{i-1} 2^{j-i} \cdot a'_j - \sum_{j=0}^{i-k-1} 2^{-k} \cdot a'_j + \sum_{j=i-k-1}^{i-2} 2^{j-i+1} \cdot a'_j + \sum_{j=0}^{i-k-2} 2^{-k} \cdot a'_j \\
&= -\frac{1}{2} a'_{i-1} + \sum_{j=2}^{k} (2^{-j+1} - 2^{-j}) a'_{i-j} \\
&= -\frac{1}{2} a'_{i-1} + \sum_{j=2}^{k} 2^{-j} a'_{i-j}.
\end{aligned}$$

The proof follows from the auxiliary equation by

$$a'_i = a'_{i-1} + (a'_i - a'_{i-1}) = a'_{i-1} - \frac{1}{2} a'_{i-1} + \sum_{j=2}^{k} 2^{-j} a'_{i-j} = \sum_{j=1}^{k} 2^{-j} a'_{i-j},$$

as required. □

We now show that the series a'_i is bounded from above by a geometric series. Thus, the sum of the a'_i series can be bounded by the sum of the geometric series, which is analytically computable.

Proposition A.3. *Let a'_i be the series defined in Equation (8). Furthermore, let $q = 1 - 2^{-k-1} = 1 - \frac{1}{2c+2}$, and let q_i be the geometric series defined by $q_i = q^{i-k}$. Then*

$$\forall i \geq 0 : a'_i \leq q_i = q^{i-k}.$$

Proof. Let f be the function defined in Proposition A.2. By Proposition A.2, $a'_i = f(a'_j : j < i)$, and f contains only additive terms. To upper bound a'_i, we use the

monotonicity of f, meaning that the larger $a'_j : j < i$ are, the larger a'_i will be. It is sufficient to show that $q_i \geq f(q_{i-1}, q_{i-2}, \ldots, q_{i-k})$. The statement then follows by induction, since

Induction base: $q_i \geq a'_i : i \leq k$ follows, since $q^{i-k} \geq q^0 = 1 = a'_i$.

Induction step: Assume that $\forall j < i : q_j \geq a'_j$. Then $q_i \geq f(q_{i-1}, q_{i-2}, \ldots, q_{i-k}) \geq f(a'_{i-1}, a'_{i-2}, \ldots, a'_{i-k}) = a'_i$. The first inequality follows by our assumption. The second inequality follows by the monotonicity of f and the induction assumption.

$$
\begin{aligned}
f(q_{i-1}, q_{i-2}, \ldots, q_{i-k}) &= \sum_{j=1}^{k} 2^{-j} q_{i-j} \\
&\leq \sum_{j=1}^{k} 2^{-j} q^{i-j-k} \\
&= q^{i-k} \sum_{j=1}^{k} (2q)^{-j} \\
&= q^{i-k} \cdot \frac{1}{2q} \cdot \frac{1 - \frac{1}{(2q)^k}}{1 - \frac{1}{2q}} \\
&= q^{i-k} \cdot \frac{1 - 2^{-k} q^{-k}}{2q - 1} \\
&\leq q^{i-k} \cdot \frac{1 - 2^{-k}}{2q - 1} \\
&= q^{i-k} \cdot \frac{1 - 2^{-k}}{2(1 - 2^{-k-1}) - 1} \\
&= q^{i-k} \cdot \frac{1 - 2^{-k}}{1 - 2^{-k}} \\
&= q_i.
\end{aligned}
$$

This concludes the proof of Proposition A.3. □

We now finish with the proof of Claim A.1.

PROOF. The size of area $a'_0, \ldots, a'_k$ is $k + 1$. We bound the total size of areas $a'_{k+1}, \ldots, a'_{\lg n}$ by summing the geometric series q_i.

$$
\begin{aligned}
\sum_{i=k+1}^{\lg n} a'_i \cdot M &\leq \sum_{i=k+1}^{\lg n} q_i \cdot M \\
&\leq \frac{1 - \left(1 - 2^{-k-1}\right)^{\lg(n)-k}}{1 - \left(1 - 2^{-k-1}\right)} M \\
&= 2^{k+1} \cdot (1 - (1 - 2^{-k-1})^{\lg(n)-k}) M \\
&= (2c + 2) \cdot \left(1 - \left(1 - \frac{1}{2c + 2}\right)^{\lg(n)-k}\right) M
\end{aligned}
$$

$$\leq (2c+2)\left(1 - e^{-\frac{\lg(n)-k}{2c+2}}\right)M$$
$$\leq \left(\lg(n) - k - \frac{\left(\lg(n)-k\right)^2}{2(2c+2)} + \frac{\left(\lg(n)-k\right)^3}{6(2c+2)^2}\right)M$$
$$\leq \left(\lg(n) - k - \frac{\lg^2(n/(c+1))}{2(2c+2)} + \frac{\lg^3(n/(c+1))}{6(2c+2)^2}\right)M.$$

The fourth inequality follows by the Taylor expansion to e^{-x}. Explicitly, $1 - e^{-\frac{\lg(n)-k}{2c+2}} \leq \frac{\lg(n)-k}{2c+2} - \frac{(\lg(n)-k)^2}{2\cdot(2c+2)^2} + \frac{(\lg(n)-k)^3}{6\cdot(2c+2)^3}$. The last inequality follows by setting $k = \lg(c+1)$ and the mathematical equation $\lg(a) - \lg(b) = \lg(a/b)$.

The proof of the claim follows by summing the total size of areas $0, \ldots, k, k+1, \ldots, \lg n$. $\square$

REFERENCES

Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. 2004. An efficient parallel heap compaction algorithm. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (ACM SIGPLAN Notices 39(10))*. ACM Press, New York, NY, 224–236. DOI:http://dx.doi.org/10.1145/1028976.1028995

David F. Bacon, Perry Cheng, and V. T. Rajan. 2003. A real-time garbage collector with low overhead and consistent utilization. In *30th Annual ACM Symposium on Principles of Programming Languages (ACM SIGPLAN Notices 38(1))*. ACM Press, New York, NY, 285–298. DOI:http://dx.doi.org/10.1145/604131.604155

Ori Ben-Yitzhak, Irit Goft, Elliot Kolodner, Kean Kuiper, and Victor Leikehman. 2002. An algorithm for parallel incremental compaction. In *3rd International Symposium on Memory Management (ACM SIGPLAN Notices 38(2 supplement))*, Hans-J. Boehm and David Detlefs (Eds.). ACM Press, New York, NY, 100–105. DOI:http://dx.doi.org/10.1145/512429.512442

A. Bendersky and E. Petrank. 2011. Space overhead bounds for dynamic memory management with partial compaction. *Princip. Programm. Lang.* 34, 3 (2011), 491–499.

Hans-Juergen Boehm. 2002. Bounding space usage of conservative garbage collectors, See POPL 2002 [2002], 93–100. DOI:http://dx.doi.org/10.1145/503272

Hans-Juergen Boehm. 2004. The space cost of lazy reference counting. In *31st Annual ACM Symposium on Principles of Programming Languages (ACM SIGPLAN Notices 39(1))*. ACM Press, New York, NY, 210–219. DOI:http://dx.doi.org/10.1145/604131.604153

Cliff Click, Gil Tene, and Michael Wolf. 2005. The Pauseless GC algorithm. In *1st ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Michael Hind and Jan Vitek (Eds.). ACM Press, New York, NY, 46–56. DOI:http://dx.doi.org/10.1145/1064979.1064988

David Detlefs, Christine Flood, Steven Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *4th International Symposium on Memory Management*, David F. Bacon and Amer Diwan (Eds.). ACM Press, New York, NY, 37–48. DOI:http://dx.doi.org/10.1145/1029873.1029879

Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall.

Haim Kermany and Erez Petrank. 2006. The compressor: Concurrent, incremental and parallel compaction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (ACM SIGPLAN Notices 41(6))*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM Press, New York, NY, 354–363. DOI:http://dx.doi.org/10.1145/1133981.1134023

Erez Petrank and Dror Rawitz. 2002. The hardness of cache conscious data placement. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, NY, 101–112. DOI:http://dx.doi.org/10.1145/503272.503283

Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. 2008. A study of concurrent real-time garbage collectors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (ACM SIGPLAN Notices 43(6))*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM Press, New York, NY, 33–44. DOI:http://dx.doi.org/10.1145/1379022.1375587

POPL. 2002. *Twenty-Ninth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, New York, NY. DOI:http://dx.doi.org/10.1145/503272

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

J.M. Robson. 1974. Bounds for some functions concerning dynamic storage allocation. *J. ACM* 21, 3 (1974), 491–499.

Received August 2014; revised February 2016; accepted September 2016