

Upper Bound for Defragmenting Buddy Heaps

Delvin C. Defoe Sharath R. Cholleti Ron K. Cytron

Department of Computer Science and Engineering
Washington University
Saint Louis, MO 63130

{dcd2, sharath, cytron}@cse.wustl.edu

Abstract

Knuth's buddy system is an attractive algorithm for managing storage allocation, and it can be made to operate in real-time. At some point, storage-management systems must either over-provide storage or else confront the issue of defragmentation. Because storage conservation is important to embedded systems, we investigate the issue of defragmentation for heaps that are managed by the buddy system. In this paper, we present tight bounds for the amount of storage necessary to avoid defragmentation. These bounds turn out to be too high for embedded systems, so defragmentation becomes necessary.

We then present an algorithm for defragmenting buddy heaps and present experiments from applying that algorithm to real and synthetic benchmarks. Our algorithm relocates less than twice the space relocated by an optimal algorithm to defragment the heap so as to respond to a single allocation request. Our experiments show our algorithm to be much more efficient than extant defragmentation algorithms.

Categories and Subject Descriptors D.3.4 [Processors]: Memory management; D.4.2 [Storage Management]: Allocation/deallocation strategies

General Terms Algorithms, Management, Languages, Performance

Keywords Memory management, Defragmentation, Compaction, Buddy System, Storage Allocation

1. Introduction

When an application begins executing, the underlying storage allocator usually obtains a large block of storage, called the *heap*, from the operating system, and uses it to satisfy the application's allocation requests. In real-time or embedded systems the heap size is usually fixed *a priori*, as the application's needs are known. Storage allocators are characterized by how they allocate and keep track of free storage blocks. There are various types of allocators including unstructured lists, segregated lists, and buddy systems [9]. In this paper we study defragmentation of the buddy allocator, whose

worst case allocation time is more tightly bounded than extant allocators [5] and thus suitable for real-time applications.

Over time, the heap becomes *fragmented* so that the allocator might fail to satisfy an allocation request for lack of sufficient, contiguous storage. As a remedy, one can either start with a sufficiently large heap so as to avoid fragmentation problems, or devise an algorithm that can rearrange storage in bounded time to satisfy the allocation request. We consider both of those options in this paper, presenting the first tight bounds on the necessary storage and the first algorithm that defragments a buddy heap in tight worst case bounded time.

Our paper is organized as follows. Section 1.1 explains the buddy allocator and Section 1.2 discusses defragmentation; Section 2 shows how much storage is necessary for an application-agnostic, defragmentation-free buddy allocator; Section 3 gives the worst-case storage relocation necessary for defragmentation; and Section 4 presents an algorithm that performs within twice the cost of an optimal defragmentation algorithm. Section 5 presents various experimental results from our defragmentation algorithm and compares its efficiency with extant defragmentation approaches. We offer conclusions and discuss future work in Section 6.

1.1 Buddy Allocator

In the binary buddy system [7], separate lists are maintained for available blocks of size 2^k bytes, $0 \leq k \leq m$, where 2^m bytes is the heap size. Initially the entire block of 2^m bytes is available. When an application requests a block of 2^k bytes, if no block of that size is available, then a larger block is repeatedly split equally into two parts until a block of 2^k bytes becomes available.

When a larger block is split, the resulting two smaller blocks are called *buddies* (of each other). If these buddies become free at a later time, then they can be *coalesced* into a larger block. The most useful feature of this method is that given the address and size of a block, the address of its buddy can be computed easily, with just a bit flip. For example, the buddy of the block of size 16 beginning in binary location $xx \dots x10000$ is $xx \dots x00000$ (where the x 's represent either 0 or 1).

Address-Ordered Buddy Allocator There is a variety of policies that govern the behavior of the binary buddy allocator. One such policy is the *address-ordered* policy. This policy selects a block of requested or greater size with the lowest address. If there are no free blocks of the requested size then the allocator searches for a larger free block, starting from the lowest address, and continuing until it finds a sufficiently large block to divide to get a required-size block. Figure 1 illustrates how this policy works. The leaves of each tree represent the smallest blocks that can be allocated. Neighboring blocks are buddies of each other and can be coalesced to form larger blocks. The larger blocks are parents to their containing smaller block pairs. Every buddy pair has a single parent. Neighboring

Permission to make digital or hard copies of all or part 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 bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCES'05, June 15–17, 2005, Chicago, Illinois, USA.
Copyright © 2005 ACM 1-59593-018-3/05/0006...\$5.00.

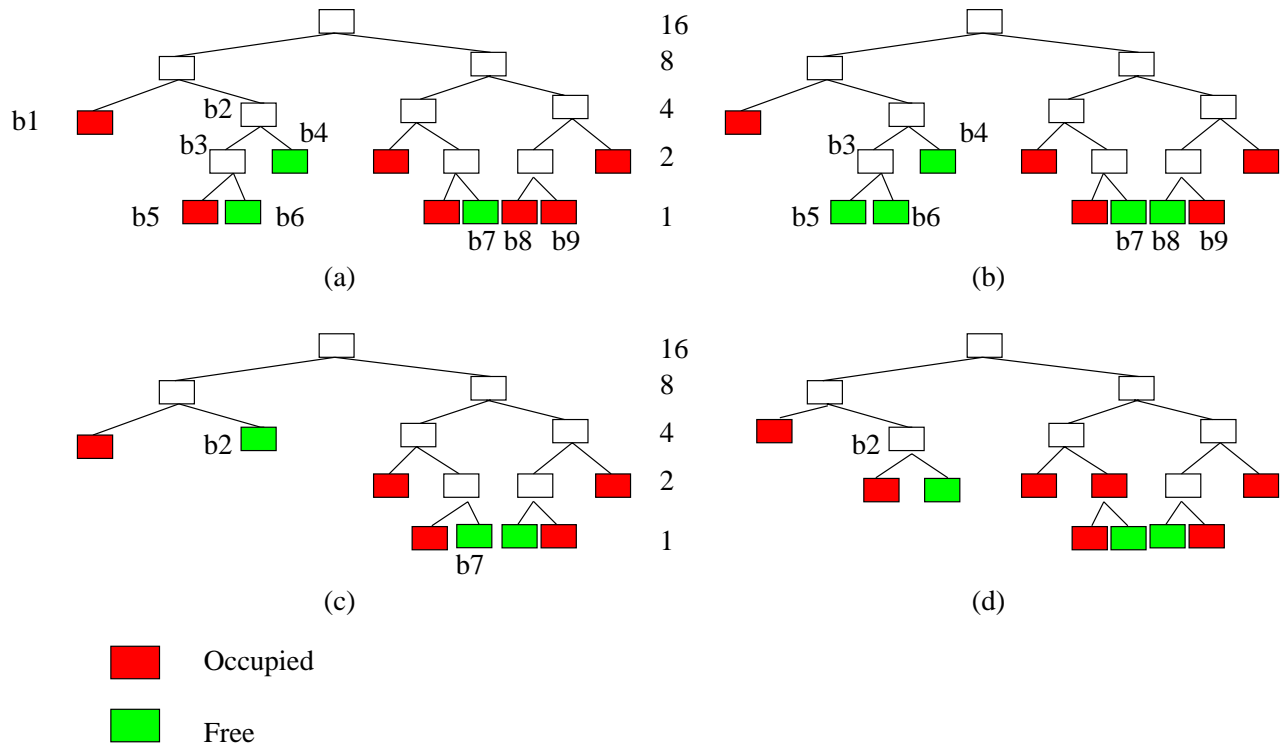


Figure 1. Buddy Example

parents are also buddies of each other and can be coalesced to form yet larger blocks. The root of each tree represents the size of the heap. All sizes are powers of two. Now, consider a request for a 1-byte block in Figure 1(c). Even though there is a 1-byte block available in the heap, block b_2 of size 4 bytes is split to obtain a 1-byte block. This policy gives preference to the lower addresses of the heap leaving the other end unused unless the heap gets full or heavily fragmented. Our analysis of the binary-buddy heap requirement is based on an allocator that uses this policy, which we call an Address-Ordered Binary Buddy Allocator.

Address-Ordered Best-Fit Buddy Allocator In this policy a block of smallest possible size equal to or greater than the required size is selected with preference to the lowest address block. If a block of required size is not available then a larger block is selected and split repeatedly until a block of required size is obtained. For example, if a 1-byte block is requested in Figure 1(c), block b_7 is chosen. If a 2-byte block is requested then block b_2 is chosen. Our implementation of the binary-buddy allocator is based on this policy, which we call an Address-Ordered Best-Fit Buddy Allocator¹.

1.2 Defragmentation

Defragmentation can be defined as moving already allocated storage blocks to other addresses so as to create contiguous free blocks that can be coalesced to form larger free blocks. Generally, defragmentation is performed when the allocator cannot satisfy an allocation request for a block. This can be performed by garbage collection [9]. A garbage collector tries to detect and reclaim dead objects. By coalescing blocks reclaimed from dead objects and unused blocks, larger blocks are formed, which the allocator can po-

tentially use to satisfy future allocation requests. Some programming languages like C and C++ need the program to specify when to deallocate and reclaim an object, whereas for programming languages like Java, the same is done by using some garbage collection technique. Generally, garbage collection is done by either identifying all the live objects and assuming all the unreachable objects to be dead as in mark and sweep collection [9] or by tracking the objects when they die as in reference counting [9] and contaminated garbage collection techniques [1].

In our study, we assume we are given both the allocation and deallocation requests. For this study it does not matter how the deallocated blocks are found – whether it is by explicit deallocation request using `free()` for C programs or by using some garbage collection technique for Java programs. For real-time purposes we get the traces for Java programs using a contaminated garbage collector [1], which keeps track of objects when they die, instead of using a mark and sweep collector which might take an unreasonably long time.

Our work makes use of the following definitions:

1. *Maxlive*, M , is defined as the maximum number of bytes alive at any instant during the program's execution.
2. *Max-blocksize*, n , is the size of the largest block the program can allocate.
3. $\log n$ as used in this article means $\log_2 n$.
4. $\theta \mapsto \theta'$ means θ maps into θ'

2. Defragmentation-Free Buddy Allocator

We examine the storage requirements for a binary-buddy allocator so that heap defragmentation is *never* necessary to satisfy an allocation request. If the amount of storage a program can use at any instant is known, assuming worst-case fragmentation, then a system with that much storage suffices.

¹Note the difference between this policy and the policy we used for our analysis - the address-ordered policy. We chose this policy for our implementation because it is easier to implement and it approximates the address-ordered policy. See Section 5 for our results.

Previous work on bounding storage requirement for allocators is found in [8]. Even though the bound given in that paper is for a system in which blocks allocated are always a power of 2, the allocator assumed is not a buddy allocator. We present a tight upper bound for the amount of storage that is necessary and sufficient to avoid defragmentation when using a buddy allocator. That bound is given in the theorems below and is shown to hold for any binary buddy allocator, address-ordered or not.

LEMMA 2.1. $M(\log n + 2)/2$ bytes of storage are sufficient for a defragmentation-free address-ordered binary buddy allocator, where M is the maxlive and $n < M$ is the max-blocksize.

PROOF. Let $I(n) = M(\log n + 2)/2$. We prove that $I(n)$ bytes are sufficient by showing that the buddy allocator cannot allocate an n -byte block beyond $I(n)$ bytes in the heap. The proof is by induction. Let $P(k)$ be the proposition that the first $I(2^k) = M(k+2)/2$ bytes of the heap is the limit beyond which an address-ordered binary buddy allocator cannot allocate a 2^k -byte block, where $2^k \leq n < M$.

Base case: $P(0)$ is true, as for a 1-byte block to be allocated beyond M bytes the first M bytes should be full (because of the address-ordered assumption) in which case maxlive is reached.

Induction assumption: Assume $P(k)$ is true $\forall k, 0 \leq k < \log n$. That is, $I(2^k)$ bytes is the limit beyond which a 2^k -byte block cannot be allocated, where $2^k < n < M$.

Now, for $k + 1$ we need to prove that $I(2^{k+1})$ is the limit.

According to the induction hypothesis, only blocks greater than 2^k bytes can be allocated beyond the first $I(2^k)$ bytes of the heap. Thus, in the worst case the first $I(2^k)$ bytes of the heap are fragmented using the minimum amount of storage required, such that a free block of size 2^{k+1} bytes cannot be found. This forces a buddy allocator to allocate 2^{k+1} -byte blocks beyond the $I(2^k)$ limit.

Let S be the *minimum* total storage currently allocated to avoid leaving a free block of size 2^{k+1} bytes in the first $I(2^k)$ bytes of the heap. The equation below follows from the induction assumption - $I(2^i)$, $0 \leq i \leq k$, is the limit. For example, a 1-byte block cannot be allocated beyond $I(1) = M$ bytes and a 2-byte block cannot be allocated beyond $I(2) = 3M/2 = M + M/2$ bytes. For the total amount of storage currently allocated to avoid leaving a free block of 2^{k+1} bytes to be the *minimum*, the first M bytes of the heap has to contain only 1-byte blocks and the next $M/2$ bytes has to contain only 2-byte blocks. Similarly, the next $M/2$ bytes has to contain only 4-byte blocks as $I(4) = 2M = M + M/2 + M/2$ bytes, and so on.

$$\begin{aligned} S &= \frac{M}{2^{k+1}} + \frac{M/2}{2^{k+1}} \cdot 2 + \frac{M/2}{2^{k+1}} \cdot 4 + \dots + \frac{M/2}{2^{k+1}} \cdot 2^k \\ &= \frac{M}{2^{k+1}} + \frac{M/2}{2^{k+1}} \cdot (2^{k+1} - 2) \\ &= \frac{M}{2^{k+1}} + \frac{M}{2^{k+1}} \cdot (2^k - 1) \\ &= \frac{M}{2} \end{aligned}$$

So at least $M/2$ bytes should be in use to force the allocator to allocate beyond $I(2^k)$ bytes. Since blocks less than 2^{k+1} bytes cannot be allocated beyond the $I(2^k)$ -byte limit, the other $M/2$ bytes of maxlive is allocated only as blocks of size greater or equal to 2^{k+1} bytes beyond $I(2^k)$ bytes, to stretch the limit. A 2^{k+1} -byte block is the smallest block allocatable beyond $I(2^k)$, so it cannot be allocated beyond $I(2^k) + M/2$ bytes without exceeding the maxlive. Let L be the limit of the heap with blocks of size 2^{k+1} .

$$\begin{aligned} L &= I(2^k) + \frac{M}{2} \\ &= \frac{M(k+2)}{2} + \frac{M}{2} \\ &= \frac{M[(k+1)+2]}{2} \\ &= I(2^{k+1}) \end{aligned}$$

Therefore $P(k + 1)$ is true. If the max-blocksize is n bytes then an n -byte block cannot be allocated beyond $I(n)$ bytes of the heap. That is, an address-ordered binary buddy allocator cannot use beyond $I(n)$ bytes of storage when the max-blocksize is n bytes and the maxlive is M bytes. So $I(n)$ bytes are sufficient for a defragmentation-free address-ordered binary buddy allocator. \square

LEMMA 2.2. $M(\log n + 2)/2$ bytes of storage are necessary for a defragmentation-free address-ordered binary buddy allocator, where M is the maxlive and $n < M$ is the max-blocksize.

PROOF. Let $I(n) = M(\log n + 2)/2$. The intuition for this proof is that a program requiring $I(n)$ bytes of storage to avoid defragmentation can be generated. We show the necessity of $I(n)$ bytes of storage by generating an allocation sequence which uses $I(n)$ bytes. This is an inductive proof.

Let $P(k)$ be the proposition that for a defragmentation-free address-ordered binary buddy allocator there exists an allocation sequence which uses $I(2^k)$ bytes, where 2^k bytes is the max-blocksize.

Base case: $P(0)$ is true as $I(2^0) = M(0 + 2)/2 = M$ is used by allocating M 1-byte blocks.

Induction Assumption: Assume $P(k)$ is true $\forall k, 0 \leq k < \log n$. That is, an allocation sequence is possible which uses $I(2^k) = M(k + 2)/2$ bytes, where $M/2$ bytes exist as blocks of size 2^k in the last $M/2$ bytes of active storage and the other $M/2$ bytes exist as blocks of smaller sizes with gaps less than or equal to 2^{k-1} bytes.

To prove $P(k + 1)$ is true, we have to show that an allocation sequence is possible, such that an address-ordered binary buddy allocator uses $I(2^{k+1}) = M[(k+1)+2]/2$ bytes of storage, when the max-blocksize is 2^{k+1} bytes. From the inductive assumption the last $M/2$ bytes of $I(2^k)$ bytes is filled up with blocks of size 2^k bytes. We can free alternate blocks among them and hence create gaps of 2^k bytes. From this step we recover $M/4$ bytes of storage. Similarly, we can free the alternate blocks of smaller sizes to increase the maximum gap size from 2^{k-1} to 2^k bytes. This recovers another $M/4$ bytes of storage, bringing the total to $M/2$ bytes. So now only $M/2$ bytes storage is used and the other $M/2$ bytes quota of maxlive can be allocated as blocks of size 2^{k+1} bytes, which have to go into storage beyond $I(2^k)$ bytes as there are no gaps of 2^{k+1} bytes. Let L be the total storage in use.

$$\begin{aligned} L &= I(2^k) + \frac{M}{2} \\ &= \frac{M(k+2)}{2} + \frac{M}{2} \\ &= \frac{M[(k+1)+2]}{2} \\ &= I(2^{k+1}) \end{aligned}$$

Therefore, there is an allocation sequence which uses $I(2^{k+1})$. That is, $P(k + 1)$ is true. Hence, $M(\log n + 2)/2$ bytes of storage are *necessary* for a defragmentation-free address-ordered binary buddy allocator. \square

1. Input:

θ is an allocation and deallocation sequence
 $s = |\theta|$
 $\beta(\cdot)$ is a function that takes $\theta_i \in \theta, 0 \leq i \leq s$,
as input and returns its
address under allocation policy β .
 $\psi(\cdot)$ is a function that takes θ'_j as input and
returns its address under
allocation policy ψ .
 $f(n)$ is the storage needed for β to satisfy θ .

2. Output:

θ' is an allocation and deallocation sequence
 $t = |\theta'|$

Figure 2. Translation Algorithm – F

THEOREM 2.3. $M(\log n + 2)/2$ bytes of storage are necessary and sufficient for a defragmentation-free address-ordered binary buddy allocator, where M is the maxlive and $n < M$ is the max-blocksize.

PROOF. The proof follows from Lemma 2.1 and Lemma 2.2. \square

Theorem 2.3 holds for an address-ordered binary buddy allocator, *AOBA*; however, we show that this result can be generalized to hold for the class of binary buddy allocators. There is not sufficient room to thoroughly present that proof; however, we present a short version of the proof after giving some intuition. The fullness of the proof can be found in [4]. The idea behind the proof is that regardless of how a buddy allocator picks the next block to allocate, there is an algorithm we call F that can force the allocator to allocate that block exactly as an *AOBA* would allocate it. We give the input and output of F as Figure 2.

The entire algorithm, F , is presented in [4]. We now present the generalization of Theorem 2.3 for the class of binary buddy allocators.

LEMMA 2.4. $M(\log n + 2)/2$ bytes of storage are sufficient for any defragmentation-free binary buddy allocator, where M is the maxlive and $n < M$ is the max-blocksize.

PROOF. Let P be a program that yields an allocation and deallocation sequence $\theta = \{\theta_1, \theta_2, \dots, \theta_s\}$. Suppose P is run on allocator α with allocation policy ψ . Knowing ψ , let F be a translation algorithm, Figure 2, that takes an *AOBA* as β and translates θ to θ' such that $\theta' = \{\theta'_1, \theta'_2, \dots, \theta'_t\}, s \leq t$, and $\theta \hookrightarrow \theta'$. F preserves the behavior of P since F is concerned only with allocation and deallocation requests and F does not alter source code. Since each θ_i , and each θ'_j represents an allocation or deallocation request and $\theta \hookrightarrow \theta'$, every $\theta_i \in \theta, 1 \leq i \leq s$ maps to some $\theta'_j \in \theta', 1 \leq j \leq t$. Thus, F possesses the following features.

- Each request contains an object ID; F uses that ID and allocation and/or deallocation address to map θ_i to θ'_j .
- In program P , if the allocation of object with ID_k depends on the address of object with $ID_{k-r}, 0 < r < k, k \leq ID_{lastObject}$, or on some other address A , F captures that address dependency in a data structure, for example a heap data structure of size $H \leq M$ bytes. We attribute H to overhead since every allocator uses some storage overhead.
- F potentially allocates and reclaims “pseudo-objects” but their storage does not count toward maxlive since P is ignorant of those potential requests.

Suppose an *AOBA* can satisfy the sequence θ without the need for defragmentation. Then, according to Lemma 2.1, at most $M(\log n + 2)/2$ bytes of storage is needed since $M(\log n + 2)/2$ bytes of storage is sufficient for an *AOBA* to satisfy θ without the need for defragmentation. But allocator α with allocation policy ψ is also able to satisfy θ' without the need for defragmentation since $\psi(\cdot)$ is constrained by $f(n)$ and β is an *AOBA* policy. Thus, Lemma 2.4 holds. \square

LEMMA 2.5. $M(\log n + 2)/2$ bytes of storage are necessary for any defragmentation-free binary buddy allocator, where M is the maxlive and $n < M$ is the max-blocksize.

PROOF. Let P be a program with allocation and deallocation sequence θ . Suppose θ is the result of the following steps.

1. Allocate blocks of size $2^i, i = 0$, such that $\sum 2^i \leq M$
2. Deallocate every other block
3. Repeat steps 1 and 2 $\forall i, 0 < i \leq \log n$.

Lemma 2.2 shows that an *AOBA* requires exactly $M(\log n + 2)/2$ bytes of storage to satisfy θ . If translation algorithm F is used to translate θ to some θ' with β being an *AOBA* policy, ψ uses at least $M(\log n + 2)/2$ bytes of storage to satisfy θ' . Thus, there exists a sequence that requires at least $M(\log n + 2)/2$ bytes of storage. \square

THEOREM 2.6. The tight bound of $M(\log n + 2)/2$ bytes holds for any binary buddy-style storage manager (i.e., not just those that are address-ordered).

PROOF. The proof follows from Lemma 2.4 and Lemma 2.5. \square

Discussion Consider the case where $n = M/2$; then the tight bounds presented in Theorem 2.3 and Theorem 2.6 can be rewritten as $M(\log M + 1)/2$. While this bound of $O(M \log M)$ is usually quite satisfactory for most problems, consider its ramifications for embedded, real-time systems. For a system that has at most M bytes of live storage at any time, the bound implies that a factor of $\log M$ extra storage is required to avoid defragmentation. Even if M is only 1KB, $\log M$ is a factor of 10, meaning that the system needs 10x as much storage as it really uses to avoid defragmentation. Inflating the RAM or other storage requirements of an embedded system by that amount could make the resulting product noncompetitive in terms of cost.

3. Worst case relocation with heap of M bytes

Given that it is unlikely that sufficient storage would be deployed to avoid defragmentation, we next attack this problem from a different perspective. We find the amount of storage that has to be relocated in the worst case, if the allocator has M bytes and the maxlive of the program is also exactly M bytes. By definition, a program can run in its maxlive storage; however, having only that much storage places significant pressure on a defragmentor. The work that must be done is stated in the theorems below, with the proofs in [2].

THEOREM 3.1. With a heap of M bytes and maxlive M bytes, to allocate an s -byte block, where $s < M, \frac{s}{2} \log s$ bytes must be relocated, in the worst case.

THEOREM 3.2. With a heap of M bytes and maxlive M bytes, to allocate an s -byte block, $s - 1$ blocks must be relocated, in the worst case.

Discussion With the smallest possible heap, defragmentation may have to move $O(s \log s)$ bytes by relocating $O(s)$ blocks to satisfy an allocation request of size s . If an application tightly

bounds its maximum storage request, then the minimum work required to satisfy an allocation request is also bounded in s bytes. Unfortunately, finding the “right” blocks to relocate in this manner is non-trivial. Thus, a heap of just M bytes is not suitable for real-time, embedded applications, and a heap of $M \log M$ bytes, while avoiding defragmentation, is too costly for embedded applications.

4. Greedy Heuristic with 2M Heap

In this section we consider a practical solution to the above problems. We consider a heap slightly bigger than optimal—twice maxlive —and consider a heuristic for defragmentation that is linear in the amount of work required to relocate storage.

Here, we use the terms *chunk* and *block* (of storage). A 2^k -byte *chunk* is a contiguous region in the heap, which consists of either free or occupied *blocks* (objects). Our heuristic defragmentation algorithm is based on the following lemma:

LEMMA 4.1. *With a $2M$ -byte heap, when allocation for a block of 2^k bytes is requested, there is a 2^k -byte chunk with less than 2^{k-1} bytes live storage.*

PROOF. *Case 1:* If there is a free block of size greater than or equal to 2^k bytes, the lemma is proved.

Case 2: There is no free block of size greater than or equal to 2^k bytes.

Size of the block needed = 2^k bytes

Maximum possible storage currently live = $M - 2^k$

Total number of 2^k -byte chunks = $2M/2^k$

Average number of bytes live in 2^k -byte chunk

$$\begin{aligned} &\leq \frac{M - 2^k}{2M/2^k} \\ &= \frac{M - 2^k}{M} \cdot 2^{k-1} \\ &< 2^{k-1} \\ &= \text{Half the required size} \end{aligned}$$

Since the average number of bytes live in a 2^k -byte chunk is less than 2^{k-1} bytes, there must at least one 2^k -byte chunk with less than 2^{k-1} bytes live storage. \square

If there is an allocation request for a 2^k -byte block and there is no free block of 2^k bytes then, according to Lemma 4.1, less than 2^{k-1} bytes have to be relocated to create a free 2^k -byte block to satisfy the request. But to relocate these blocks we might have to empty some other blocks by repeated relocations if there are no appropriate free blocks.

How much storage must these recursive relocations move? For example, consider an allocation request for a 256-byte block but there is no such free block. According to Lemma 4.1, there is a 256-byte chunk in which less than 128 bytes are live. Assume there is a 64-byte live block that has to be moved out of the 256-byte chunk. But suppose there is no free block of 64 bytes. Then, a 64-byte chunk has to be emptied. Let that contain a block of 16 bytes. This 16-byte block cannot be moved in to either the 256 or the 64-byte chunk, as it is being relocated to empty those chunks in the first place. The result of accounting for the above work is captured by the following theorem and corollary. Their proofs can be found in [2].

THEOREM 4.2. *With a $2M$ -byte heap, the greedy approach of selecting a chunk with minimum amount of live storage for relocation, relocates less than twice the amount of storage relocated by an optimal strategy.*

COROLLARY 4.3. *With a $2M$ -byte heap, the amount of storage relocated to allocate an s -byte block is less than s bytes.*

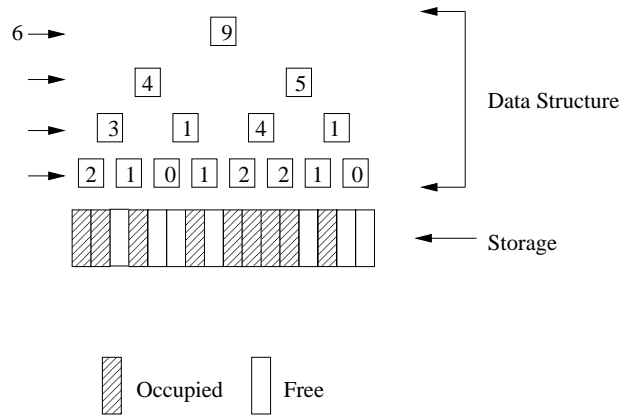


Figure 3. Data Structure

Heap Manager Algorithm A naive method for finding the minimally occupied chunk involves processing the entire storage pool, but the complexity of such a search is unacceptable at $O(M)$. We therefore use the data structure shown in Figure 3 to speed our search. Each level of the tree has a node associated with a chunk of storage that is allocatable at that level. In each node, an integer shows the number of live bytes available in the subtree rooted at that node. In Figure 3, the heap is 16 bytes. The level labeled with “2 \rightarrow ” keeps track of the number of live bytes in each 2-byte chunk. Above that, each node at the 4 \rightarrow level keeps the sum of the number of live bytes in its children, showing how many bytes are in use at the 4-byte level, and so on.

When a 2-byte block is allocated, the appropriate node at the 2 \rightarrow level is updated, but that information must propagate up the tree using parent-pointers, taking $O(\log M)$ time. To find a minimally occupied block of the required size, only that particular level is searched in the data structure, decreasing the time complexity of the search to $O(M/s)$, where s is the requested block size. We have used hardware to reduce this kind of search to near-constant time [5] and we expect similar results to be obtained here.

The algorithm’s details are straightforward and appear in [2], wherein proofs can be found for the following theorems:

THEOREM 4.4. *With a $2M$ -byte heap, defragmentation according to the Heap Manager Algorithm (Section 4) takes $O(Ms^{0.695})$ time to satisfy a single allocation request for an s -byte block.*

THEOREM 4.5. *With an M -byte heap, defragmentation according to the Heap Manager Algorithm (Section 4) takes $O(Ms)$ time to satisfy a single allocation request for an s -byte block.*

Discussion A constant increase in heap size (from M to $2M$) allows the heuristic to operate polynomially, but its complexity may be unsuitable for real-time systems. The work that must be done is bounded, in terms of the allocation-request size s , but more research is needed to find heuristics that operate in similar or better time.

5. Experimental Results

Storage requirements as well as allocation and deallocation patterns vary from one program to another. Hence, storage fragmentation and the need for defragmentation vary as well. To facilitate experimentation, we implemented a simulator that takes the allocation and deallocation information from a program trace and simulates the effects of allocation, deallocation, and defragmentation

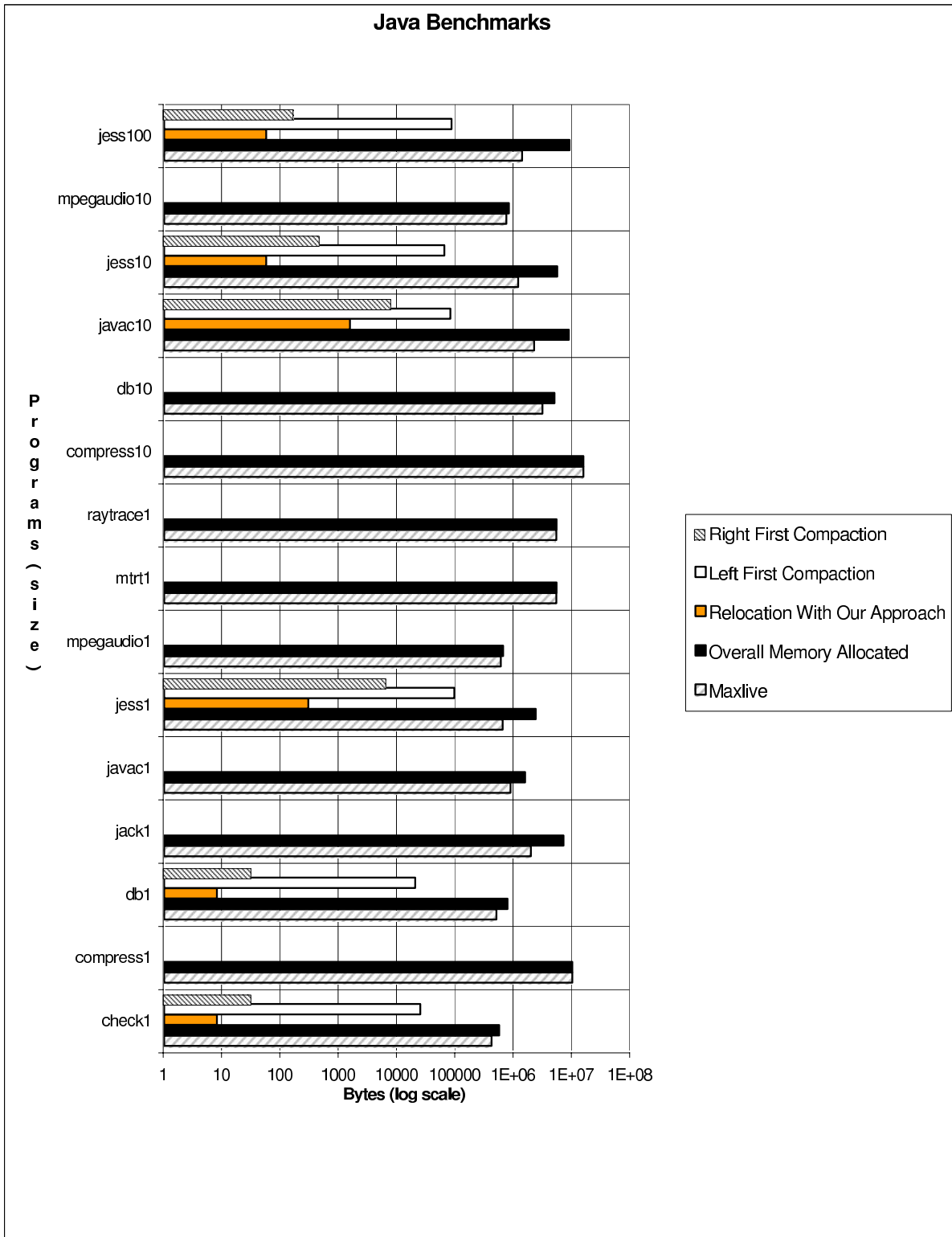


Figure 4. Java Test Programs – Amount of Relocation

Benchmark	check1	db1	jess1	javac10	jess10	jess100
Maxlive	411552	500832	638328	2204600	1177768	1377448
Overall Memory Allocated	559112	774696	2355552	8743504	5510056	8974528
Relocation: Minimally Occupied	8	8	296	1536	56	56
Relocation: Random Partially Occupied	8	8	968	1024	120	56

Table 1. Our relocation strategy - Minimally Occupied Selection, is compared with Random Block Selection. For each benchmark, the amount of memory relocated is shown when a particular strategy is used. The overall memory allocated and the maxlive for each benchmark are also shown.

Benchmark	check1	db1	jess1	javac10	jess10	jess100
Number of Relocations: Minimally Occupied	1	1	8	2	3	3
Number of Relocations: Random Partially Occupied	1	1	27	30	7	4

Table 2. Our relocation strategy - Minimally Occupied Selection, is compared with Random Block Selection. For each benchmark, the number of blocks relocated is shown when a particular strategy is used.

using the Address-Ordered Best-Fit Buddy policy described in Section 1.1 and our greedy heuristic described in Section 4.

For benchmarks, we used the Java SPEC benchmarks [3]. To illustrate the efficiency of our defragmentation algorithm, we compare our results with the following approaches currently in common practice.

Left-First Compaction This approach compacts all live storage to the *lower-addressed* free blocks of the heap, by filling them with the closest live blocks to their right that they can accommodate.

Right-First Compaction This approach is similar to left-first compaction in that it moves all live storage to the lower addresses of the heap; however, it picks the live blocks by scanning from the *higher addresses* of the heap, where we expect storage to be less congested due to our address-ordered allocation policy.

Compaction Without Buddy Properties We implemented this approach so we can compare our defragmentation algorithm to a naive compaction algorithm that simply slides live blocks to one end of the heap. This approach is among the general class of non-buddy allocators.

5.1 Defragmentation with $2M$ -byte Heap

We explored defragmentation in various benchmarks with a $2M$ -byte heap using the algorithm described in Section 4, which is based on Theorem 4.2. We were pleased that the Address-Ordered Best-Fit Buddy Allocator along with our greedy heuristic was able to satisfy the allocation and deallocation requests of every benchmark, without defragmentation, when a heap of size twice maxlive was used. So having twice the maxlive storage, the Address-Ordered Best-Fit Buddy Allocator was able to avoid relocation of live storage for those programs. Even some randomly generated program traces did not need any relocation with a heap of size $2M$ bytes. These results harmonize with what is known in practice that for the “average program” most allocation algorithms do not need defragmentation [6].

However, our theoretical results show that there are some programs for which defragmentation can be problematic. Real-time and embedded systems must be concerned with accommodating worst-case behaviors.

5.2 Defragmentation with M -byte Heap

We noticed that using our algorithm with a $2M$ -byte heap induced no defragmentation for the benchmarks, so we experimented with

an exact-sized heap of M bytes. From Figure 4 we see that six of the fifteen (40%) programs required defragmentation. Among the programs that required defragmentation, javac(10) and jess(1) required the most. When we compared the performance of our defragmentation algorithm with the compaction methods, our algorithm outperformed them all. Our algorithm relocates only the amount of live storage necessary to satisfy a particular allocation request but the compaction methods defragment the entire heap each time they cannot satisfy an allocation request.

Among the compaction methods, only right-first compaction performed reasonably well. The other methods, namely left-first compaction and naive compaction, relocated significantly more storage, sometimes close to M bytes.

The above results show the weakness of the compaction methods when compared to our defragmentation algorithm. Our algorithm uses localized defragmentation to satisfy a particular allocation request but the compaction methods defragment the entire heap. Figure 4 and Table 2 show that there is no point in doing extra work to compact the entire heap, either in anticipation of satisfying future allocation requests or for some other reason.

5.3 Minimally Occupied vs Random Block Selection

Our heuristic selects the minimally occupied block for relocation. In this section we compare that strategy against what happens when the relocated block is chosen at random and the heap is sufficiently small so as to cause defragmentation (M bytes). We report results only for those benchmarks that needed defragmentation under those conditions.

From Table 1 we see that out of the 6 Java SPEC benchmarks that required defragmentation, for 3 of those programs - check(1), db(1), and jess(100), selecting the minimally occupied block for defragmentation did not reduce the amount of relocation required. For 2 programs, namely jess(1), and jess(10), selecting the minimally occupied block relocated less storage and for 1 program, javac(10), selecting a random block relocated less storage.

From Table 2, 2 Java SPEC benchmarks - check(1) and db(1) needed the same number of relocations with either selection heuristic while the 4 other benchmarks - jess(1), javac(10), jess(10), and jess(100) needed fewer relocations with selecting the minimally occupied block. Note that even though random selection needed more relocations for javac(10), the amount of storage relocated is less.

The above results indicate that using random block selection might be a good alternative, and it avoids searching for the minimally occupied block. We have not yet determined the theoretical time bound for random block selection. That is a matter for future work.

6. Conclusions and Future Work

We presented a tight bound for the storage required for a defragmentation-free buddy allocator. While the bound could be appropriate for desktop computers, for embedded systems there is an inflationary factor of $\log M$ for the storage required to avoid defragmentation. While a factor of 2 might be tolerable, a factor of $\log M$ is most likely prohibitive for embedded applications.

We presented a greedy algorithm that allocates an s -byte block using less than twice the optimal relocation on a $2M$ -byte (twice maxlive) heap. The Address-Ordered Best-Fit Buddy Allocator we implemented was very effective in satisfying allocation requests for the Java SPEC benchmarks when a heap of $2M$ -bytes was used. However, 40% of those benchmarks needed some relocation when the heap was M -bytes large. The results show better performance for a localized defragmentation algorithm, which defragments just a small portion of the heap to satisfy a single allocation request, over the conventional compaction algorithms.

We compared the minimally occupied selection heuristic of our greedy algorithm with random selection heuristic to find that our heuristic performed better, as expected. However, since the overall fragmentation is not overwhelming, the random heuristic takes less time and might work well in practice; a theoretical bound on its time complexity is needed.

As the effectiveness of the localized defragmentation, by relocation, is established in this paper, it is a good idea to concentrate on studying such algorithms instead of defragmenting the entire heap. We proposed only one algorithm based on a heap-like data structure, but further study could involve designing and implementing better data structures and algorithms to improve on the current ones.

Acknowledgments

This work is sponsored by DARPA under contract F33615-00-C-1697 and by the Chancellor's Graduate Fellowship Program at Washington University. We would like to thank both organizations for their support. The authors may be contacted at cytron@cs.wustl.edu and dcd2@cs.wustl.edu. We would also like to thank Morgan Deters of Washington University for his feedback and assistance in editing this article.

References

- [1] Dante J. Cannarozzi, Michael P. Plezbert and Ron K. Cytron. Contaminated garbage collection. *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 264–273, 2000.
- [2] Sharath Reddy Cholleti. Storage Allocation in Bounded Time. Master's thesis, Washington University, 2002.
- [3] SPEC Corporation. Java SPEC Benchmarks. Technical report, SPEC, 1999. Available by purchase from SPEC.
- [4] Delvin C. Defoe. Effects of Coalescing on the Performance of Segregated Size Storage Allocators. Master's thesis, Washington University in St. Louis, 2003. Available as Washington University Technical Report WUCSE-2003-69.
- [5] Steven M. Donahue, Matthew P. Hampton, Morgan Deters, Jonathan M. Nye, Ron K. Cytron, and Krishna M. Kavi. Storage Allocation for Real-Time, Embedded Systems. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software: Proceedings of the First International Workshop*, pages 131–147. Springer Verlag, 2001.
- [6] Mark S. Johnstone and Paul R. Wilson. The Memory Fragmentation Problem: Solved? In *International Symposium on Memory Management*, October 1998.
- [7] Donald E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts, 1973.
- [8] J. M. Robson. Bounds for Some Functions Concerning Dynamic Storage Allocation. *Journal of ACM*, 21(3):491-499, July 1974.
- [9] Paul R. Wilson. Uniprocessor Garbage Collection Techniques (Long Version). Submitted to ACM Computing Surveys, 1994.