View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Washington University St. Louis: Open Scholarship

Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCSE-2003-2

2003-01-31

Storage Allocation in Bounded Time

Sharath Reddy Cholleti

The correctness of a real-time system is very much dependent on the time at which a specific task is completed. Hence, satisfying a storage allocation request within bounded time is important. Fragmentation of the heap after repeated allocations and deallocations is a major issue for real-time systems, as most allocators depend on garbage collection for defragmentation of the heap, which might not finish in time to honor deadlines. We present the storage requirement for a defragmentation-free binary-buddy allocator. We also study a localized defragmentation algorithm to satisfy a single allocation request, within bounded time, instead of requiring defragmentation of... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Cholleti, Sharath Reddy, "Storage Allocation in Bounded Time" Report Number: WUCSE-2003-2 (2003). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/1067

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/1067

Storage Allocation in Bounded Time

Sharath Reddy Cholleti

Complete Abstract:

The correctness of a real-time system is very much dependent on the time at which a specific task is completed. Hence, satisfying a storage allocation request within bounded time is important. Fragmentation of the heap after repeated allocations and deallocations is a major issue for real-time systems, as most allocators depend on garbage collection for defragmentation of the heap, which might not finish in time to honor deadlines. We present the storage requirement for a defragmentation-free binary-buddy allocator. We also study a localized defragmentation algorithm to satisfy a single allocation request, within bounded time, instead of requiring defragmentation of the entire heap. We prove that the cost of the algorithm is within twice the optimal cost. Results are presented from applying the defragmentation algorithm, with different heap sizes, on various programs. The amount of storage relocated with our defragmentation algorithm is compared with other compaction algorithms. Also, the amount of storage relocated by selecting a minimally occupied block is compared with the policy of selecting a block randomly.

Washington University in St. Louis

School of Engineering & Applied Science

Short Title: Storage Allocation Cholleti, M.Sc. 2002

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

STORAGE ALLOCATION IN BOUNDED TIME

by

Sharath Reddy Cholleti, B.Tech.

Prepared under the direction of Prof. Ron K. Cytron

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

December, 2002

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

STORAGE ALLOCATION IN BOUNDED TIME

by Sharath Reddy Cholleti

ADVISOR: Prof. Ron K. Cytron

December, 2002
Saint Louis, Missouri

The correctness of a real-time system is very much dependent on the time at which a specific task is completed. Hence, satisfying a storage allocation request within bounded time is important. Fragmentation of the heap after repeated allocations and deallocations is a major issue for real-time systems, as most allocators depend on garbage collection for defragmentation of the heap, which might not finish in time to honor deadlines. We present the storage requirement for a defragmentation-free binary-buddy allocator. We also study a localized defragmentation algorithm to satisfy a single allocation request, within bounded time, instead of requiring defragmentation of the entire heap. We prove that the cost of the algorithm is within twice the optimal cost.

Results are presented from applying the defragmentation algorithm, with different heap sizes, on various programs. The amount of storage relocated with our defragmentation algorithm is compared with other compaction algorithms. Also, the

amount of storage relocated by selecting a minimally occupied block is compared with the policy of selecting a block randomly.

to my mom, dad and sis

Contents

List of Figures . . . . **vi**

Acknowledgments . . . . **vii**

1 Introduction **1**

1.1 Storage Management 1

1.2 Real-Time Systems 1

1.3 Embedded Systems 2

1.4 The Problems 2

1.5 Road Map 3

2 Background **4**

2.1 Unstructured Lists 4

2.2 Segregated Free Lists 6

2.3 Binary Buddy Systems 6

2.4 Buddy Allocator Policy 7

2.4.1 Address-Ordered Buddy Allocator 7

2.4.2 Address-Ordered Best-Fit Buddy Allocator 8

2.5 Defragmentation 8

2.6 Notation 9

3 Binary Buddy Heaps without Defragmentation **11**

3.1 Trivial Upper Bound 11

3.2 Failures $2M$, $3M$, . . . and Automatic generation of Counter Example . 12

3.3 Tight Bound 14

4 Binary Buddy Heaps with Defragmentation **18**

4.1 Big to Small Allocation Never Locks 19

4.2 Worst case relocation with heap of M bytes 20
4.3 Optimal Sequence of Relocation can be Sorted from Small to Big . . 21
4.4 Greedy Heuristic with 2M Heap 22
4.4.1 Heap Manager Algorithm 26
4.4.2 The Algorithm – Pseudo Code 28
4.4.3 Time and Storage Bounds 30

5 Experiments **34**
5.1 The Test Programs 34
5.1.1 Java Test Programs 35
5.1.2 C Test Programs 36
5.2 Simulator 36
5.2.1 Allocation 37
5.2.2 Defragmentation 37
5.2.3 Left-First Compaction 37
5.2.4 Right-First Compaction 38
5.2.5 Compaction Without Using Buddy Properties 39
5.3 Results 39
5.3.1 Defragmentation with $2M$-byte Heap 40
5.3.2 Defragmentation with M-byte Heap 43
5.3.3 Minimally Occupied vs Random Block Selection 43

6 Conclusion and Future Work **47**

References **49**

Vita **51**

List of Figures

2.1 Unstructured List Example. 5
2.2 Buddy Example: Heap of 16 bytes 8

3.1 Counter example with $2M$ storage. 13

4.1 Data Structure 27

5.1 Java Test Program Characteristics. 36
5.2 C Test Program Characteristics. 36
5.3 Left-First Compaction 38
5.4 Right-First Compaction 39
5.5 Non-Buddy Compaction 40
5.6 Java Test Programs – Amount of Relocation 41
5.7 C Test Programs – Amount of Relocation 42
5.8 Minimally Occupied vs Random – Amount of Relocation 45
5.9 Minimally Occupied vs Random – Number of Relocations 46

Acknowledgments

I thank my advisor, Ron K. Cytron, for all his help from introducing the fragmentation issues in buddy allocators to his insights to the approach we take to tackle the problem.

Also, I would like to thank all my friends for numerous discussions on my thesis and supporting me all the time directly or indirectly to enjoy life in general. In particular, Morgan Deters, Michael Plezbert, Ramaprabhu Janakiraman, Krishnakumar Balasubramanian and Anand Krishnan for their suggestions, discussing the proofs and proof reading; Dante Cannarozzi, Steve Donahue, Matthew Hampton and Delvin Defoe for their experimental data I used in my thesis; Martin Linenweber, Irfan Pyrali and Nanbor Wang for their support in talking about the thesis and participating in various sports that kept me active; My college friends K.V.M Naidu and Sanjeev Dwivedi for their support. In case I missed anyone, I thank all the people, things and places — known/unknown, near/far, seen/unseen — who/which were part of my life during the last two years in a positive way.

Sharath Reddy Cholleti

Washington University in Saint Louis
December 2002

Chapter 1

Introduction

1.1 Storage Management

In any program of significant size, storage management is a necessity. When a program starts, the storage allocator has a large block of storage, called the *heap*, that it can use for allocations. A program can request a block of storage from the allocator, for example in C++ by using the operator `new`, and can deallocate a block by using the operator `delete`. After repeated allocations and deallocations, holes can start to appear in the heap. This is called *fragmentation* of the heap. Sometimes the allocator cannot satisfy a request because of fragmentation of the heap into many, small free chunks. The allocator could request more storage from the operating system, but every system has a limited amount of storage. If the operating system cannot provide more then the allocator cannot satisfy a request from a program, even if it has enough free storage scattered among the small chunks.

An alternative is to combine some of the small, free chunks to obtain a larger, free chunk. This process, called *defragmentation*, requires rearranging live storage so that the storage holes become adjacent and can coalesce.

1.2 Real-Time Systems

A real-time system is one in which correctness of the system depends not only on the logical results, but also on the time at which the results are provided [10]. Real-time systems exist in various areas such as avionics, air-traffic control, telecommunications,

multimedia, virtual reality, medical applications and defense applications. Almost all safety-critical systems are real-time systems.

In a time-critical situation, storage-management functions must be reasonably bounded with regard to their execution time. When a program requests a block of storage, the allocator has to satisfy the request in bounded time. The bound is necessary because task-execution times are typically submitted to a scheduler to determine if all of the tasks' deadlines can be met [7]. Thus, if a storage-management function exceeds its time limit, there could be catastrophic results in the overall system.

1.3 Embedded Systems

Embedded systems are characterized by a set of predefined functions to be performed on the resources available, where storage, power and computational functionality are limited. Typically embedded systems are components of a larger system. Embedded systems exist in systems like microwave ovens, watches, ATMs, avionics, defense equipment. Many embedded systems are part of a larger system with real-time requirements.

As mentioned earlier, embedded systems have limited storage. The smaller the amount of storage needed to run a system, the more it cuts the down cost of the system as well as its power consumption. Since embedded systems have very specific functions to do, the maximum storage required can usually be calculated *a priori*. But dynamic storage allocation causes fragmentation of the heap, making the calculation difficult. Some systems use static allocation of objects, avoiding dynamic allocation. But this leads to wasted space as the objects are live all the time [11]. We study how to tackle the fragmentation problem when using dynamic allocation, while at the same time keeping in mind the limitations of embedded systems.

1.4 The Problems

To explain the problems we study in this thesis, first we need to define *defragmentation* and *maxlive*. *Defragmentation* can be defined as moving the allocated storage blocks to some other address so as to create contiguous free blocks that can be coalesced to form a larger free block. *Maxlive* is the maximum amount of storage the program requires at any instant during its execution.

Motivated by the requirements of real-time systems and embedded systems as discussed in sections 1.2 and 1.3, we study two related problems:

1. Given the maxlive of a program, what is the maximum amount of storage required by a buddy allocator (described in Section 2.3) without any defragmentation, so that it is always able to satisfy an allocation request in spite of the fragmentation?

2. Given the maxlive of a program and twice-maxlive heap, can the heap be defragmented to satisfy a single request and what is the efficiency of that defragmentation?

1.5 Road Map

In Chapter 2, we provide background on the problem we study and on our approach. In Chapter 3, we examine the issue of how much storage is required so that all requests for storage can be satisfied without ever having to defragment the heap. In Chapter 4, we present a new heap-defragmentation algorithm; experiments using that algorithm are presented in Chapter 5. In Chapter 6, we present conclusions and future work concerning this research.

Chapter 2

Background

When a program starts, the storage allocator obtains a large block of storage, called the *heap*, which it uses to satisfy the program's allocation requests. Depending on the system, the storage allocators can generally increase or decrease the size of the heap by sending an appropriate request to the operating system [2]. Storage allocators are characterized by how they allocate and keep track of free blocks. This chapter describes the most generally used storage allocation mechanisms [13], including the buddy allocator [6] upon which our research is based.

In Section 2.1 and Section 2.2, allocators based on unstructured lists and segregated free lists, respectively, are described. In Section 2.3, the binary-buddy system is explained. In Section 2.4 the buddy allocator policies used in this thesis are defined. In Section 2.5, defragmentation is described. In Section 2.6, our notation for allocation and deallocation is given.

2.1 Unstructured Lists

These allocators have a universal list of all free blocks of storage. Generally the list is doubly-linked lists or circular-linked to ease deletion from the list. These allocators use various policies:

Best Fit A best fit allocator searches the free list to find the smallest free block large enough to satisfy the request. If an exact match is found the search stops otherwise the whole list is searched exhaustively. For example, if a 100-byte block is requested in Figure 2.1, the whole list is searched and the 104-byte block is chosen for the allocation; the remaining 4-byte block is put on the list.

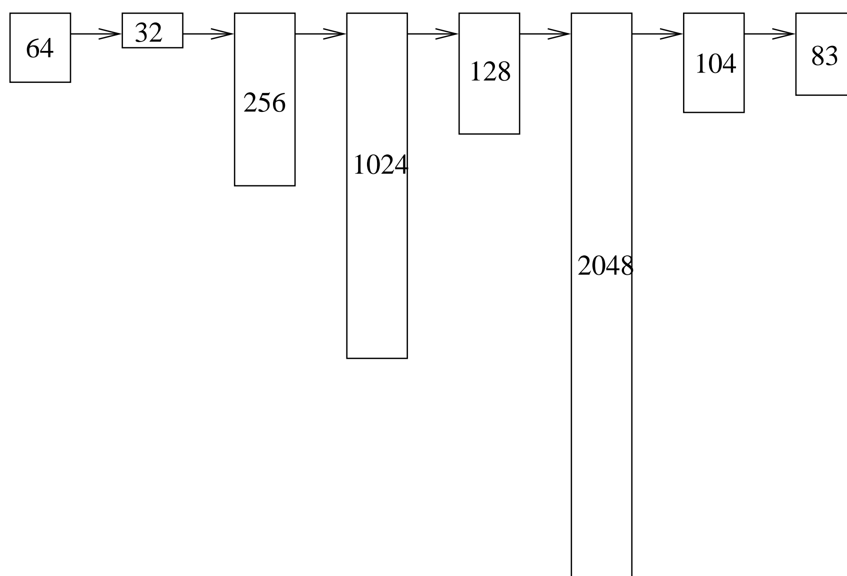


Figure 2.1: Unstructured List Example.

Though the basic motivation for best fit is to find an exact match, it can be a bad policy if the best fits are approximate, with the resulting fragments too small to be useful.

Worst Fit A worst fit allocator searches the free list to find the largest free block that is larger than the requested size. For example, if a 100-byte block is requested in Figure 2.1, the whole list is searched and the 2048-byte block is chosen for the allocation. The remaining 1948-byte block is added to the list. This avoids the problem of having too many tiny fragments as in best fit allocation.

First Fit A first fit sequential fit allocator searches the free list from the beginning to the end until it finds a free block large enough to satisfy the allocation request. If the block is larger than required, then it is split and the remainder is added to the free list. For example, if a 100-byte block is requested in Figure 2.1, then the 256-byte block is chosen for the allocation and the remaining 156-byte block is put on the list. A problem with first fit allocation is that the blocks at the beginning of the list are split and many small blocks accumulate at the beginning (only if the policy is to put the remaining block on the list at the same place, which is generally the case). This leads to increased search time for larger blocks.

Next Fit This is an optimization over first fit. Instead of starting the search at the beginning of the list, it is started from the position where the last request was satisfied. A pointer is maintained to remember the last position. For example, after requesting a 100-byte block as in Figure 2.1, a subsequent request for 1200-byte block starts from 156-byte block and the 2048-byte block is split. The remaining 848-byte is put on the free list. The main motivation is to reduce the search time.

2.2 Segregated Free Lists

This policy uses a set of free lists, where each free list holds different-sized objects. To satisfy an allocation request, a free list of the appropriate size is chosen. The size is rounded to next larger size if there is no perfect match. When an object is freed it is added to the free list of that object's size.

Simple Segregated Storage Here, larger free blocks are not split to satisfy a request for a smaller size and smaller free blocks are not coalesced to satisfy a request for a larger size. If the appropriate list for satisfying a request is empty then more storage is requested from the operating system, split into same-sized blocks, and then put on the free list.

Segregated Fits This variation relaxes the constraint that all objects in a size class be exactly the same size. This variant uses a set of free lists, with each list holding free blocks of any size between this size to the next larger size class. To satisfy a request the appropriate size class is searched for a large enough block to hold it.

2.3 Binary Buddy Systems

In binary buddy system [6], 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 a block of 2^k bytes is requested, and if no blocks of that size are available, then a larger block is split into two equal parts repeatedly until a block of 2^k bytes is obtained.

When a block is split into two equal sized blocks, these blocks are called *buddies*. If these buddies become free at a later time, then they can be *coalesced* into one 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 very easily, with just a bit flip. For example, the buddy of the block of size 16 beginning in binary location $xx \ldots x10000$ is $xx \ldots x00000$ (where the x's represent either 0 or 1). For more details see [6] page 442.

Tree Notation We present a tree notation for the buddy system, as in Figure 2.2, to represent the state of a buddy heap. We use this tree notation to visualize the worst case for defragmentation-free buddy allocation as well as other problems we tackled. The root of the tree is the whole heap and the next level denotes two halves of the heap, each of which is in turn split into two and so on. The allocation and deallocation (freeing) operations on the buddy system can visualized from the Figure 2.2 as follows: Figure 2.2(a) shows buddy system with the heap size 16 bytes. Blocks b_1 and b_2 are buddies. Similarly, b_3 & b_4; b_5 & b_6 and b_8 & b_9 are buddies. After block b_5 and b_8 are freed Figure 2.2(b) is obtained. Since buddies b_5 and b_6 are free they coalesce to form a free larger block b_3 which in turn coalesces with b_4 to form free block b_2, as in Figure 2.2(c).

When block b_8 is freed, even though there is a free block b_7 adjacent to it, in Figure 2.2(b), they cannot coalesce because they are not buddies. At that instant, as in Figure 2.2(c), if there is a request for allocation of a 2-byte block, block b_2 has to be split to satisfy the request. The resultant state of the buddy system can be seen in Figure 2.2(d).

There are other variants of buddy systems, such as Fibonacci Buddy, Weighted Buddy and Double Buddy. They use block sizes which may not be powers of two. The series of sizes differ for all these buddy systems. These variants are discussed in [13].

2.4 Buddy Allocator Policy

2.4.1 Address-Ordered Buddy Allocator

The address-ordered 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. For example, if a 1-byte

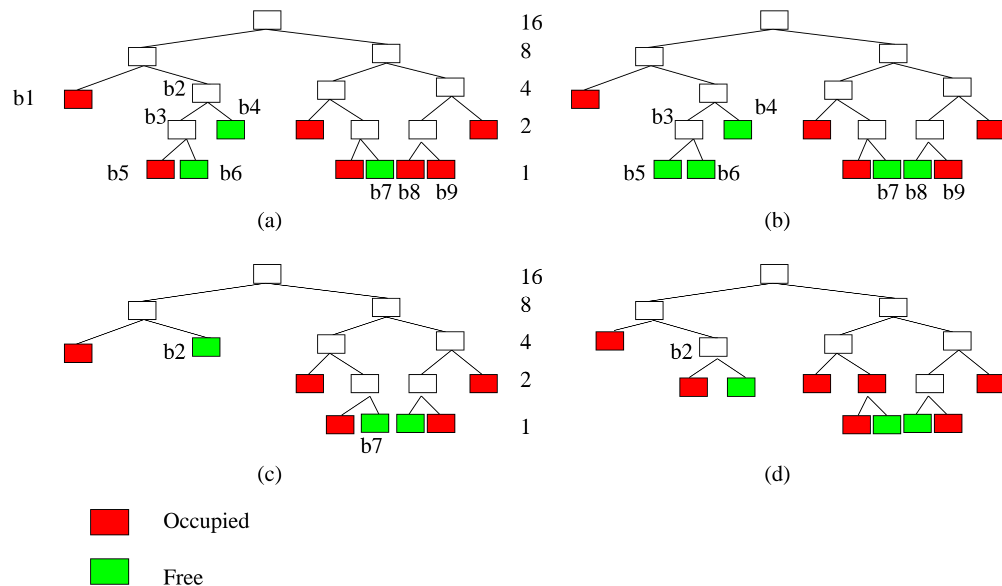


Figure 2.2: Buddy Example: Heap of 16 bytes

block is requested in Figure 2.2(c), block b_2 (of 4 bytes) is split to obtain a 1-byte block even though there is a 1-byte block available in the heap. Using this policy the lower addresses of the heap tend to get preference leaving the other end unused unless the heap gets full or fragmented a lot. Our analysis of the binary-buddy heap requirement is based on this policy.

2.4.2 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. When a block of required size is not available then a block of higher size is selected and split repeatedly until a block of required size is obtained. For example, if a 1-byte block is requested in Figure 2.2(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 is based on this policy.

2.5 Defragmentation

Defragmentation can be defined as moving already allocated storage blocks to some other address so as to create contiguous free blocks that can be coalesced to form a

larger free block. Generally, defragmentation is performed when the allocator cannot satisfy a request for a block. This can be performed by garbage collection [12]. A garbage collector tries to separate the live and deallocated or unused blocks, and by combining the unused blocks, if possible, usually a larger block is formed, which the allocator can use to satisfy the allocation requests. Some programming languages like C and C++ need the program to specify when to deallocate an object, whereas the programming languages, like Java, find which objects are live and which are not 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 collectors [12] or by tracking the objects when they die as in reference counting [12] 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 contaminated garbage collection [1], which keeps track objects when they die, instead of mark and sweep collectors which might take unreasonably long time.

2.6 Notation

A program issues a number of allocation and deallocation requests to the allocator throughout its running time, continuously changing the amount of storage used. We use the following notation given below throughout the thesis to study the program's storage usage. All storage sizes are in bytes.

1. ϕ is a sequence of allocation and deallocation requests of the program. ϕ_i, $i = 1, 2, \ldots, T$ denotes the ith request, which is either:

   - $Alloc(n)$, an allocation request for a block of size n, or
   - $Dealloc(k)$, a deallocation request for the block allocated in the kth allocation request.

   Also, the kth allocation request is denoted as $Alloc_k(n)$, where n is the size of the request.

2. $\phi \circ \phi_t$ denotes a sequence of requests ϕ followed by an allocation or a deallocation request ϕ_t.

3. log denotes $\log_2$

4. $|Alloc(n)|$ denotes the size of the block actually allocated. For binary-buddy $|Alloc(n)| = 2^{\lceil \log n \rceil}$.

5. $|Dealloc(k)|$ denotes the negative value of the size of the block deallocated, which is also the size of the kth allocated block.
$|Dealloc(k)| = -|Alloc_k(n)| = -2^{\lceil \log n \rceil}$.

6. $$|\phi_i| = \begin{cases} 2^{\lceil \log n \rceil} & \text{if } \phi_i = Alloc(n) \\ -2^{\lceil \log n \rceil} & \text{if } \phi_i = Dealloc(k) \end{cases}$$

7. $Curlive$ is the amount of storage in use at that particular instant. $Curlive_t$ denotes the $Curlive$ after $\phi_1, \phi_2, \ldots, \phi_t$.

$$Curlive_i = \begin{cases} 0 & \text{if } i = 0 \\ Curlive_{i-1} + |\phi_i| & \forall i > 0 \end{cases}$$

8. M denotes the *maxlive* — the maximum number bytes alive at any instant during the program's execution. If ϕ_T is the last request,

$$M = \max_{0 \leq i \leq T} (Curlive_i)$$

9. n is the *max-blocksize* — size of the largest block (in bytes) the program can allocate.

10. H is the heap size.

11. $B(H, \phi)$ is a predicate which is true if the address-ordered buddy allocator, starting with a heap of size H bytes, satisfies the request sequence ϕ. We also use the terminology, $B(H, \phi)$ *is satisfied* if it is true and $B(H, \phi)$ *failed or locked* if it is false.

Chapter 3

Binary Buddy Heaps without Defragmentation

In this chapter, 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. This could be useful in embedded systems, for that matter any systems, which are supposed to work without breaking down. For real-time systems, this is even more useful because the program cannot wait for an allocator to defragment the heap to satisfy a request. Also, some languages like C and C++ do not allow relocating allocated blocks because of the uncertainity of what is a "pointer".

In Section 3.1, a trivial upper bound for a defragmentation-free heap size is discussed. In Section 3.2, examples are given to show that $O(M)$ storage, where M is maxlive, is not enough for a defragmentation-free heap. In Section 3.3, a tight bound for defragmentation-free heap is proved.

3.1 Trivial Upper Bound

The below theorem gives a trivial upper bound for the buddy allocator without defragmentation.

Theorem 3.1.1 *Let* $k = |\{n | \exists i \; n = |\phi_i|, n > 0\}|$. kM *bytes storage is sufficient to satisfy any sequence of allocation and deallocation requests of the program without any defragmentation.*

Proof: Assume kM bytes storage is not sufficient. That is, to satisfy a particular allocation request kM bytes is not enough. But by associating each of the k sizes with a pool of M bytes and allocating blocks of a particular size from its own pool of M bytes, the allocator does not need any extra storage to satisfy a request for any of the k sizes as maxlive is only M. Hence a contradiction. ■

Corollary 3.1.2 *Assuming M is a power of 2, $M \log M$ bytes storage is sufficient with buddy system allocator.*

Proof: As M is maxlive, there are at most $\log M$ different sizes $(1, 2, 4, \ldots, M/2)$, the program can request, excluding M-byte block. So $M \log M$ is sufficient. When M-byte block is requested $curlive = 0$ ■

3.2 Failures $2M$, $3M$, ... and Automatic generation of Counter Example

As the trivial upper-bound seemed an overkill, we next find the exact bound on the amount storage required for the defragmentation free buddy allocator. First we experimented with constant multiple of maxlive storage. We generate counter examples of allocation and deallocation patterns that force the buddy allocator to run out of the given storage because of fragmentation. As mentioned in 2.4.1, we deal with address-ordered buddy allocator.

Consider an example where M is 16 bytes and buddy allocator has $2M$ storage (32 bytes), to satisfy any request. The counter example shown in Figure 3.1, fragments the storage so that the allocator fails to satisfy a request even though the live storage has not exceeded 16 bytes as follows:

1. Allocate sixteen 1-byte blocks, i.e., $\phi_i = Alloc_i(1)$ for $i = 1, \ldots, 16$. This fills up first 16 bytes.

2. Deallocate alternate 1-byte blocks, i.e., $\phi_i = Dealloc(2 * (i - 16))$ for $i = 17, \ldots, 24$. This creates fragmentation such that no 2-byte can be allocated in first 16 bytes (Figure 3.1(a)).

3. Allocate four 2-byte blocks, i.e., $\phi_i = Alloc_j(2)$ for $i = 25, \ldots, 28$ and $j = 17, \ldots, 20$. These occupy the third quarter of the 32 bytes.

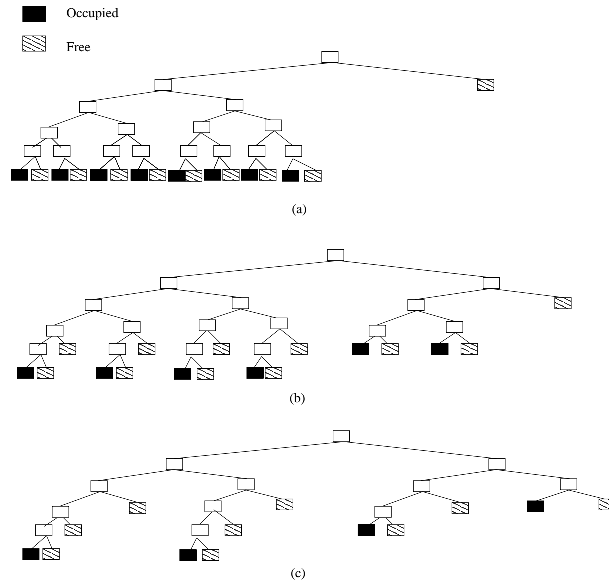


Figure 3.1: Counter example with $2M$ storage.

4. Deallocate alternate 1-byte blocks and 2-byte blocks, i.e. $\phi_i = Dealloc(k)$ for $k = 3, 7, 11, 15, 18, 20$ and $i = 29, \ldots, 34$. This creates fragmentation so that no 4-byte block can be allocated in first three quarters of 32 bytes (Figure 3.1(b)).

5. Allocate two 4-byte blocks, i.e. $\phi_i = Alloc_j(4)$ for $i = 35, 36$ and $j = 21, 22$. These occupy the fourth quarter.

6. Deallocate alternate 1-byte, 2-byte and 4-byte blocks, i.e., $\phi_i = Dealloc(k)$ for $i = 37, \ldots, 40$ and $k = 5, 13, 19, 22$. This reduces the current live storage to 8 bytes and creates fragmentation so that no 8-byte block can be allocated (Figure 3.1(c)).

By using the similar process of allocation followed by deallocation of every other block, counter examples can be generated for a heap of size $2.5M$ bytes (for maxlive 32 bytes, fails to allocate 16-byte block), heap of size $3M$ bytes (for maxlive 64 bytes, fails to allocate 32-byte block) and so on.

3.3 Tight Bound

As seen from Section 3.2, a constant multiple of M is insufficient to avoid the fragmentation problem in worst case. So how much storage is needed? We give the details of the maximum storage requirements in this section.

Below we give the minimum amount of storage required for defragmentation-free heap in the worst case. We assume the buddy allocator uses address-ordered policy (Section 2.4.1). The address-ordered policy gives preference for the lower-address free blocks, and hence one end of the heap stays very active and other end of the heap extends only if the currently active portion of the heap is full or fragmented.

Some previous work about the bound on storage requirement 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.

Lemma 3.3.1 $I(n) = M(\log n + 2)/2$ *bytes storage is* sufficient *for a defragmentation-free buddy allocator, where M is the maxlive and $n < M$ is the max-blocksize.*

Proof: We prove that $I(n)$ bytes is sufficient by showing that buddy allocator need not allocate a 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 a 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 in which case the maxlive is reached.

Induction assumption: Assume $P(k)$ is true $\forall k, 0 \leq k < \log n$. That is, $I(2^k) = M(k+2)/2$ bytes is the limit beyond which a 2^k-byte block need not 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. So in the worst case the first $I(2^k)$ bytes of the heap is fragmented using the minimum amount of storage required, such that it does not have a free block of size 2^{k+1} bytes. This forces a buddy allocator to allocate 2^{k+1}-byte blocks beyond the $I(2^k)$ limit.

Let S be the *minimum* total amount of storage concurrently allocated to avoid leaving a free block of size 2^{k+1} bytes in the first $I(2^k)$ bytes of the heap. The below equation follows from the induction assumption $I(2^i)$, $0 \leq i \leq k$, is the limit. For

example, since a 1-byte block cannot 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 concurrently allocated to avoid leaving a free block of 2^{k+1} bytes to be *minimum*, the first M bytes of the heap would involve only 1-byte blocks and next $M/2$ bytes would involve only 2-byte blocks. Similarly next $M/2$ bytes would involve only 4-byte blocks as $I(4) = 2M = M + M/2 + M/2$ bytes, and so on.

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

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 need not be allocated beyond $I(2^k)$-byte limit, other $M/2$ bytes of the maxlive can be allocated only as blocks of size greater or equal to 2^{k+1} bytes beyond $I(2^k)$ bytes to stretch the limit. Since 2^{k+1}-byte block is the smallest beyond $I(2^k)$, it need not 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. So if the max-blocksize is n bytes then a n-byte block cannot be allocated beyond $I(n)$ bytes of the heap. That is, a buddy allocator cannot use beyond $I(n)$ bytes of storage when the max-blocksize is n bytes and maxlive M bytes. So $I(n) = M(\log n + 2)/2$ is sufficient for a defragmentation-free buddy allocator. ■

Lemma 3.3.2 *$I(m) = M(\log n + 2)/2$ bytes storage is* necessary *for a defragmentation-free buddy allocator, where M is the maxlive and $n < M$ is the max-blocksize.*

Proof: We show that $I(n)$ bytes is necessary by showing the existence of an allocation sequence which uses $I(n)$ storage.

Again the proof is by induction. Let $P(k)$ be the proposition that for a defragmentation-free 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: Consider an example, for case $k = 1$, allocate M blocks of size 1. Now deallocate every other block, hence creating gaps of size 1 byte. Now the storage that is used up is $M/2$ bytes. The remaining $M/2$ bytes are allocated as blocks of size 2 bytes. Since a block of size 2 bytes cannot fit in any gaps in the first M bytes of storage, they occupy the next $M/2$ storage. So the total storage used is $3M/2 = I(2)$ bytes. Similarly for $k = 2$, deallocate every other block of size 1-byte and size 2-byte, hence freeing $M/2$ bytes storage which can be used to allocate blocks of size 4 bytes in $M/2$ bytes beyond $I(2)$ bytes of storage. For the *inductive step*, assume $P(k)$ is true for $1, 2, \ldots, k < \log n$. i.e. an allocation sequence is possible which uses $I(2^k) = M(k+2)/2$ bytes, where $M/2$ exists as blocks of size 2^k in the last $M/2$ bytes of active storage and the other $M/2$ exists 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 a 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 recovered $M/4$ 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. Since we freed the alternate blocks which totals $M/2$ bytes we recover another $M/4$ 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 used.

$$L = I(2^k) + \frac{M}{2}$$

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

Therefore, there is an allocation sequence which uses $I(2^{k+1})$. i.e. $P(k+1)$ is true. Hence, $I(n) = M(\log n + 2)/2$ bytes storage is *necessary* for a defragmentation-free buddy allocator. ■

Theorem 3.3.3 $I(n) = M(\log n + 2)/2$ *bytes storage is* necessary and sufficient *for a defragmentation-free buddy allocator, where M is the maxlive and $n < M$ is the max-blocksize.*

Proof: Follows from Lemma 3.3.1 and Lemma 3.3.2. ■

Corollary 3.3.4 $M(\log M + 1)/2$ *bytes storage is necessary and sufficient for a defragmentation-free buddy allocator, where M is the maxlive and the max-blocksize.*

Proof: This is the case with $n = M/2$ (since allocating a M-byte block means all the storage is free, so the allocator just needs M bytes of storage). ■

So this bound of $M(\log M + 1)/2$ bytes is an improvement of *almost half* over the trivial case of $M \log M$. Since this is a tight bound, it cannot be improved further and thus we should have $M(\log M + 1)/2$ bytes of storage in the system if we do not want the allocator to ever lock due to fragmentation. In practice, the programs might never need this much storage because of their *good* allocation policies, but for a real-time system which does not use defragmentation there is no other option but to have $M(\log M + 1)/2$ bytes to guarantee its performance.

Chapter 4

Binary Buddy Heaps with Defragmentation

As we have seen in Chapter 3, defragmentation-free address-ordered buddy allocator needs $\Theta(M \log M)$ heap. The $\log M$ factor makes the system more expensive as the M increases. $\Theta(M)$ heap would be much better, if the heap can be defragmented to satisfy the allocation requests within bounded time. This chapter explores buddy allocation with $2M$ heap and defragmentation in bounded time.

Defragmentation requires rearranging live storage to make the storage holes adjacent to each other, so that they can coalesce to form a larger free chunk. We call this rearranging of live storage block as *relocation*, which involves moving a live storage block from one address to another where there is enough free space. Relocation is measured in number of bytes or blocks relocated. Also, *relocation sequence* is a sequence of relocations taking place one after another to satisfy a single allocation request.

Section 4.1 shows if the sizes of allocation requests are in non-increasing order then the allocator is always able to satisfy the request without blocking. Section 4.2 gives the worst case relocation. Section 4.3 shows that any optimal relocation sequence can be sorted in increasing order of sizes. Section 4.4 gives the algorithm for at most twice optimal relocation sequence.

4.1 Big to Small Allocation Never Locks

In this section we study the effect of allocation of blocks in non-increasing order of sizes. We show that the allocator does not *lock*, that is, it is able to satisfy the request because a block of required or bigger size is available.

Notation:

- $NFB_i(\phi)$ denotes the number of free blocks of size 2^i bytes after sequence of allocations and dellocations ϕ.

Lemma 4.1.1 $(\forall \phi)$ $(\forall i\ 0 \leq i < \log M)$ $[(s > 2^i) \rightarrow NFB_i(\phi) = NFB_i(\phi \circ Alloc(s))]$.

Proof: By cases:

1. *Allocation fails:* Then clearly $NFB_i(\phi) = NFB_i(\phi \circ Alloc(s))$.

2. *Allocation succeeds:*

   (a) If a block of size $2^{\lceil \log_2 s \rceil}$ bytes is available on the free list, that is allocated. So $NFB_i(\phi) = NFB_i(\phi \circ Alloc(s))$.

   (b) If a free block of $2^{\lceil \log_2 s \rceil}$ bytes is not available, a block of size 2^k bytes, where $k > \lceil \log_2 s \rceil$, is recursively split in to two equal blocks until a block of $2^{\lceil \log_2 s \rceil}$ bytes is available. So $NFB_i(\phi) = NFB_i(\phi \circ Alloc(s))$.

■

Corollary 4.1.2 $(\forall \phi)$ $(\forall i\ 0 \leq i < \log M)$ $[(\forall\ j \geq 1)(s_j > 2^i) \rightarrow NFB_i(\phi) = NFB_i(\phi \circ Alloc(s_1) \circ Alloc(s_2) \circ Alloc(s_3) \ldots)]$.

Theorem 4.1.3 $[(\phi = \phi_1, \phi_2, \ldots, \phi_r \wedge \phi_i = Alloc(s_i)),\ (\forall i, j\ 1 \leq i < j \leq r \rightarrow s_i \geq s_j),\ \sum_{i=1}^{r} |Alloc(s_i)| \leq M,\ H \geq M] \rightarrow B(H, \phi)$.

Proof: The theorem states that given an empty heap and if the allocations are in non-decreasing order of size of the blocks then the allocator never locks.

Let $P(k)$ be the proposition that the theorem is true for $\phi = \phi_1, \ldots, \phi_k$.
Basis step: $P(1)$ is clearly true.
Inductive step: Assume that $P(k)$, where $k < r$, is true.

Since initially the heap was empty, from the Theorem 4.1.2, after the allocations $Alloc(s_1), Alloc(s_2), \ldots, Alloc(s_k)$, all the free storage is in free blocks of size $|Alloc(s_j)|$, where $s_1 \geq s_j \geq s_k$. As $s_j \geq s_{k+1}$, $Alloc(s_{k+1})$ does not lock. Therefore, $P(k+1)$ is true. ■

Theorem 4.1.4 $[(\forall \phi), (\forall i\ 1 \leq i \leq r), (NFB_{s_i}(\phi) \geq 1), (\phi' = \phi \circ \phi_1 \circ \phi_2 \circ \ldots \circ \phi_r \wedge \phi_i = Alloc(2^{s_i})), (\forall i, j\ 1 \leq i < j \leq r \rightarrow s_i \geq s_j)] \rightarrow B(H, \phi')$.

Proof: Similar to the above proof. Since $(\forall \phi)$ $(\forall i, j\ 1 \leq i < j \leq r)$ $NFB_{s_j}(\phi) = NFB_{s_j}(\phi \circ Alloc(2^{s_k}))$, there is at least one free block of the required size and hence no allocation locks. ■

Corollary 4.1.5 *Pulling out all the blocks involved in relocation and allocating them starting with biggest to smallest, in sequence, does not lock.*

4.2 Worst case relocation with heap of M bytes

In this section we find the amount of storage that has to be relocated in worst case, if the allocator has M bytes and the maxlive of the program is also M bytes.

Theorem 4.2.1 *With heap of M bytes and maxlive M bytes, to allocate a s-byte block, where $s < M$, $\frac{s}{2} \log s$ bytes must be relocated, in worst case.*

Proof: In worst case, every block of s bytes used in relocation has only 1 byte free (using a fully occupied s-byte block is unnecessarily expensive). To free a block of s bytes, $s - 1$ bytes have to be relocated. These blocks might cause more relocation when they are relocated. To cause maximum (worst case) relocation these $s - 1$ bytes are made of $s/2, s/4, \ldots, 1$ byte blocks. This is because, instead of a $s/2$-byte block if there are smaller blocks they do not need as much relocation to move to other location.

Let $R(s)$ be the number of bytes to be relocated to *allocate* a s-byte block and $T(s)$ be the number of bytes to be relocated to *relocate* an s-byte block. Also, $T(1) = 1$.

$$
\begin{aligned}
R(s) &= T(s/2) + T(s/4) + T(s/8) + \ldots + T(1) \\
&= [s/2 + T(s/4) + T(s/8) + \ldots + T(1)] + T(s/4) + T(s/8) + \ldots + T(1) \\
&= s/2 + 2 * [T(s/4) + T(s/8) + \ldots + T(1)] \\
&= s/2 + 2 * s/4 + 4 * [T(s/8) + T(s/16) + \ldots + T(1)] \\
&= \frac{s}{2} \log s
\end{aligned}
$$

■

Theorem 4.2.2 *With heap of M bytes and maxlive M bytes, to allocate a s-byte block, $s-1$ blocks must be relocated, in worst case.*

Proof: Similar to the above proof, here also $s-1$ bytes have to be relocated.

Let $R(s)$ be the number of blocks to be relocated to *allocate* a s-byte block and $T(s)$ be the number of blocks to be relocated to *relocate* an s-byte block. $T(1) = 1$.

$$\begin{aligned} R(s) &= T(s/2) + T(s/4) + \ldots + T(1) \\ &= 1 + 2 * [T(s/4) + T(s/8) + \ldots + T(1)] \\ &= 1 + 2 + 4 + \ldots + s/2 \\ &= s - 1 \end{aligned}$$

∎

4.3 Optimal Sequence of Relocation can be Sorted from Small to Big

To allocate a block some times the heap has to be defragmented by relocating some blocks. There can be many solutions, relocation sequences, such that the changed heap has a free block of required size. Optimal relocation sequence is one which relocates minimum amount of storage to get a free block of required size. In this section we show that any optimal relocation sequence can be sorted such that the smaller blocks are relocated before the bigger ones to get another optimal relocation sequence.

Notation:

- b_i denotes a block of size 2^i bytes.
- $\gamma(b_i, l_1, l_2)$ is defined as relocation of block b_i from location l_1 to l_2.

Lemma 4.3.1 *To allocate a block b_i, no block b_j, where $i \leq j$, is relocated, for the relocation sequence to be optimal.*

Theorem 4.3.2 *No block b_i can be moved to location vacated by block b_j, where $i \leq j$, for the relocation sequence to be optimal.*

Proof: Assume otherwise. That is, assume $\gamma(b_j, j_1, j_2)$ and $\gamma(b_i, i_1, j_1)$ are part of the *optimal* relocation sequence, in that order.

Let b_a be the block to be allocated because of which the relocation is taking place. From the Lemma 4.3.1, $j < a$.

Due to the movement of block b_i to location j_1, which was vacated by b_j, block b_a cannot be allocated in the space containing location j_1. It has to be allocated some where else after some other sequence of relocations.

But without moving b_j from location j_1 and $\gamma(b_i, i_1, j_2)$ gives a configuration obtained with a lesser amount of relocation and *equivalent* to the first one as far as allocation of b_a is concerned. This is a contradiction. ■

Theorem 4.3.3 *In any optimal relocation sequence smaller block can be relocated before a bigger block.*

Proof: Assume otherwise. That is, an optimal sequence needs to relocate b_j prior to b_i, where $i < j$ (i.e. b_i cannot relocate before b_j does). The only thing which stops b_i from relocating prior to b_j is if b_i has to move in to the space b_j vacates. But this is impossible by the above theorem. Therefore, the optimal relocation sequence can be changed so that b_i is relocated before b_j. ■

Corollary 4.3.4 *Any optimal relocation sequence can be sorted so that smaller blocks are relocated first.*

Proof: By above theorem, a smaller block can be relocated before a bigger block. Hence, given an optimal relocation sequence, it can be sorted in ascending order of block sizes. ■

4.4 Greedy Heuristic with 2M Heap

In this section, we use $2M$-byte heap to see how the allocation requests are satisfied with this extra storage and how much storage we have to relocate if the need arises. Remember that $2M$-byte is not sufficient to satisfy all allocation requests as shown in Section 3.2. We present a theorem (Theorem 4.4.3), which directly forms a basis for a greedy algorithm [3] that relocates less than twice the storage an optimal algorithm would relocate.

Note the way we use the words *chunk* and *block* (of storage). A 2^k-byte *chunk* is a contiguous storage in the heap which consists of either free or occupied *blocks* (objects).

Lemma 4.4.1 *With $2M$-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 $= 2M/2^k$

$$\begin{aligned}\text{The average number of bytes live in a } 2^k\text{-byte chunk} &\leq \frac{M-2^k}{2M/2^k} \\ &= \frac{M-2^k}{M}\cdot 2^{k-1} \\ &< 2^{k-1} \\ &= \text{Half the required size}\end{aligned}$$

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. ∎

If there is an allocation request for a 2^k-byte block and there is no free block of 2^k bytes then, from the Lemma 4.4.1, less than 2^{k-1} bytes have to be relocated to create a free 2^k-byte block. But to relocate these blocks we might have to empty some other blocks by repeated relocations if there are no appropriate free blocks. So how much storage these recursive relocations move? Unlike in the above lemma, where the entire $2M$ storage space can be used to select a 2^k-byte block, in this case the 2^k-byte chunk in which the blocks to be relocated are present cannot be used because this 2^k-byte chunk is being emptied to satisfy an allocation request for a 2^k-byte block. Similarly, the chunks which are used for repeated relocation cannot be used. For example, there is an allocation request for 256-byte block but there is no such a free block. To satisfy the allocation some blocks have to be moved to some other locations to get a free block of 256 bytes. According Lemma 4.4.1, there is a 256-byte chunk in which less than 128 bytes are live. Assume there is a 64-byte block in that live storage, which has to be moved out of 256-byte chunk. Assume there is no free block of 64 bytes. Now 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 in the first place to empty those chunks.

Lemma 4.4.2 *With $2M$-byte heap, if there is an allocation request for a 2^k- byte block when there is no free block of size of greater or equal to 2^k bytes, for any 2^q-byte block to be* relocated[1], *where $q < k$, there is a 2^q-byte chunk with less than 2^{q-1} live storage .*

Proof: *Case 1:* 2^q-byte block is part of the 2^k-byte chunk to be emptied.

Maximum possible storage (in bytes) currently live except in the 2^k-byte chunk $= M - 2^k - 2^q$

Total amount of storage (in bytes) not counting the 2^k-byte chunk $= 2M - 2^k$

Total number of 2^q-byte chunks which can possibly be used for relocation $= (2M - 2^k)/2^q$

Let the average number of bytes live in a 2^q-byte block that can be used for relocation be A.

$$\begin{aligned} \text{A} &\leq \frac{M - 2^k - 2^q}{(2M - 2^k)/2^q} \\ &= \frac{M - 2^k - 2^q}{(M - 2^{k-1})/2^{q-1}} \\ &< 2^{q-1} \end{aligned}$$

Case 2: 2^q-byte block is part of a chunk that has to be emptied to allow repeated relocation. Assume, to empty a 2^k-byte chunk a 2^{q_1}-byte block is to be moved out and that there is no free block of 2^{q_1} bytes. So a 2^{q_1}-byte chunk has to be emptied — assume it has a 2^{q_2}-byte block, which has to be moved out, but there is no free block of 2^{q_2} bytes ($\forall i, j \; i < j \rightarrow q_i > q_j$). So a 2^{q_2}-byte chunk has to be emptied and a similar problem as above exists and goes on until a 2^{q_r}-byte chunk has to be emptied, which has a 2^q-byte block to be moved out. Now we have to prove that there exists a 2^q-byte chunk which has less than 2^{q-1} bytes of live storage.

Maximum possible storage (in bytes) currently live not counting the chunk 2^k and the blocks $2^q{}_1, 2^q_2, \ldots, 2^q_r$ and 2^q

$$= M - 2^k - 2^{q_1} - 2^{q_2} - \ldots - 2^{q_r} - 2^q$$

[1] *The relocation need not be the first one to empty 2^k-byte chunk. It could be any of the relocations that are needed during the repeated (recursive) relocations*

Total amount of storage (in bytes) that can possibly be used for relocation

$$= 2M - 2^k - 2^{q_1} - 2^{q_2} - \ldots - 2^{q_r}$$

Total number of 2^q-byte chunks that can be used for relocation

$$= (2M - 2^k - 2^{q_1} - 2^{q_2} - \ldots - 2^{q_r})/2^q$$

(Since 2^q-byte block cannot be moved to either the 2^k or 2^{q_i}-byte chunk, $1 \leq i \leq r$.)

Let the average number of bytes live in a 2^q-byte block that can be used for relocation be A.

$$\begin{aligned} \text{A} &\leq \frac{M - 2^k - 2^{q_1} - 2^{q_2} - \ldots - 2^{q_n} - 2^q}{(2M - 2^k - 2^{q_1} - 2^{q_2} - \ldots - 2^{q_n})/2^q} \\ &= \frac{M - 2^k - 2^{q_1} - 2^{q_2} - \ldots - 2^{q_n} - 2^q}{(M - 2^{k-1} - 2^{q_1-1} - 2^{q_2-1} - \ldots - 2^{q_n-1})/2^{q-1}} \\ &< 2^{q-1} \end{aligned}$$

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

■

Theorem 4.4.3 *With $2M$-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.*

Proof: To get a free 2^k-byte block, an optimal algorithm has to move at least the blocks present in a minimally occupied 2^k byte chunk. Let this minimally occupied chunk has m bytes of live storage.

The greedy algorithm also selects a 2^k byte chunk with live storage of m bytes. Let

$$m = 2^{i_1} + 2^{i_2} + \ldots + 2^{i_r}$$

Now the greedy algorithm has to find space for these blocks to relocate. From the Lemma 4.4.2, to relocate any 2^{i_j}-byte block we can find 2^{i_j}-byte chunk which has less than 2^{i_j-1} bytes of live storage.

Let the amount of live storage that needs to be relocated to help move out m bytes be m'.

$$m' < 2^{i_1-1} + 2^{i_2-1} + \ldots + 2^{i_r-1}$$

$$= (2^{i_1} + 2^{i_2} + \ldots + 2^{i_r})/2$$
$$= m/2$$

But m' bytes has to be relocated to some where else recursively, till there is no need of any relocation. Taking the same approach as above, m' bytes relocate less than $m'/2$ bytes, which in turn relocate less than $m'/4$ bytes and so on.

$$\text{The total amount of storage that is to be relocated} < m + m/2 + m/4 + \ldots + 1$$
$$= 2m - 1$$

Therefore, the greedy approach relocates less than twice the amount of storage that an optimal algorithm relocates. ■

Corollary 4.4.4 *With a $2M$-byte heap, the amount of storage relocated to allocate a s-byte block is less than s bytes.*

Proof: Follows from the Lemma 4.4.1 and the Theorem 4.4.3. ■

4.4.1 Heap Manager Algorithm

The main idea in Theorem 4.4.3 is to select a minimally occupied chunk of the required size and relocate all the sub blocks in that chunk to some other locations. The cost of such a relocation is proved to be less than twice the optimal cost when the heap size is $2M$ bytes. In this section, we describe a data structure and algorithms to find the minimally occupied block and relocate the sub blocks. Here we assume the assume the heap size $2M$ bytes.

A simple way of finding the minimally occupied chunk is to go through the entire storage and seeing which block of required size has less live storage. But the time complexity of such a search is $O(M)$ (note that it is of the order of the heap size which is $2M$ byte). To decrease the search time we use a data structure, similar to the heap data structure [3] but without the sorting, to record the amount of live storage in chunks of size $2, 4, \ldots, M$ bytes. This data structure is similar to the heap data structure for locating the parent or child node using the node's address. For example, in Figure 4.1, the whole storage is 16 bytes. First level in the data structure keeps track of how full the 2-byte chunks of the storage are. It stores

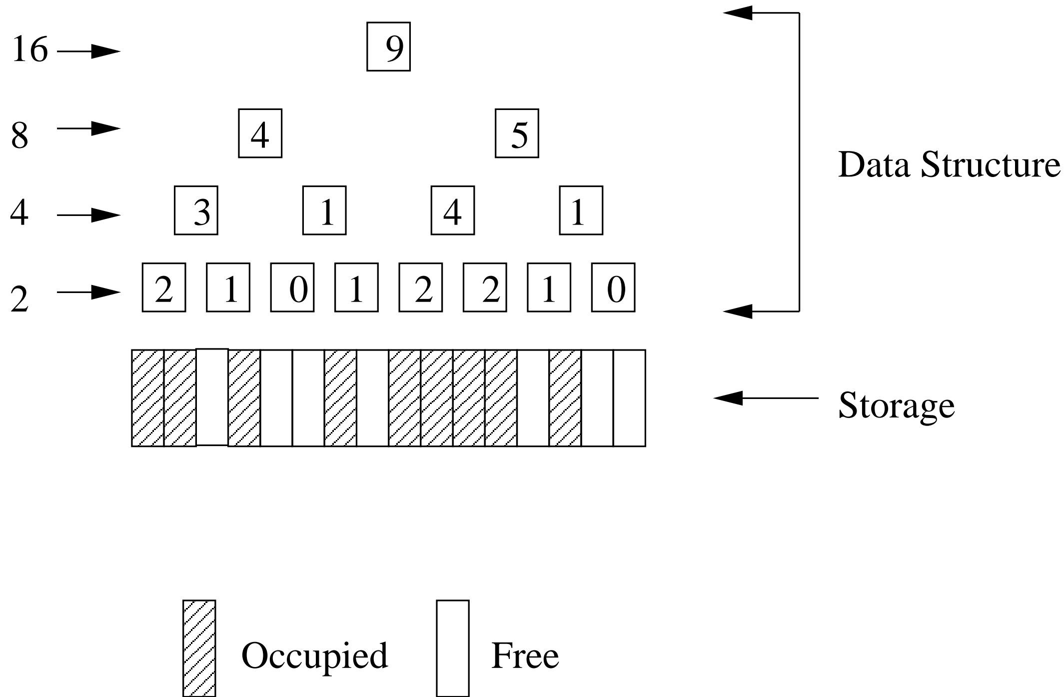


Figure 4.1: Data Structure

the number of live bytes each 2-byte chunk has. That 2-byte chunk can be two 1-byte blocks or a 2-byte block. The second level stores fullness of 4-byte chunks by taking the numbers from the first level which keeps track of 2-byte chunks. This way of recording the fullness goes for on $8, \ldots, M$-byte chunks. For example, when a 2-byte block is allocated appropriate node in the first level is updated. But that information has to propagated to second level and so on. That is where having a heap-like data structure is helpful in finding the appropriate parent node. To find a minimally occupied block of the required size, only that particular level is searched in the data structure. This decreases the time complexity of the search to $O(M/s)$, where s is the required block size (it has to go through $2M/s$ numbers to find a minimum). This is done by FindMinimallyOccupiedBlock() method in the pseudo code given below (pseudo code for all the methods mentioned in this section is given below in Section 4.4.2). After finding a minimally occupied block, all the sub blocks can be relocated to some other locations guaranteeing the less than twice the optimal relocation cost, by the Theorem 4.4.3. These relocations cost more search time. The overall time cost is given below in Section 4.4.3. But having this data structure adds

$2M - 1$ bytes additional storage requirement. In fact, the data structure does not need to keep track of 2-byte chunks as the need to find a minimally occupied block of 2 bytes never arises. That is because when there is a need to find a 2-byte block, less than M bytes storage is live out of $2M$ bytes, and therefore the average number of bytes live in a 2-byte chunk is less than 1 indicating there is 2-byte block free. So the data structure can start tracking at 4-byte chunk level. This reduces the extra storage requirement by half to $M - 1$ bytes (with a $2M$-byte heap).

In Initialization() method all the values in the data structure are initialized to 0. Allocate() method tries to allocate a block of requested size according to the address-ordered best-fit buddy allocation algorithm (Section 2.4.2). If there are no blocks of requested size or bigger, the normal buddy allocator [6] cannot satisfy the request and gives an error. But this Allocate() relocates some blocks to create a big enough chunk to satisfy the request. First it calls FindMinimallyOccupiedBlock() method to find a minimally occupied block and then calls Relocate() to relocate the sub blocks. Relocate() deallocates a sub block by calling deallocate() and then allocates that sub block else where, except in the block which is being freed. It is applied to all the sub blocks one after another.

The important method which is called after every allocation or deallocation is UpdateHeapManager(), which updates the data structure according to the changes in each sub tree (for example Figure 4.1). When an allocation (deallocation) occurs at a particular level, values in all the above levels are increased (decreased) by the size of the allocated (deallocated) block (costs $O(\log M)$ time units) and all the sub levels are *supposed* to be changed to full (empty). But noticing that sub levels are either full or empty, and that a particular level is not searched when there is a free block (normal buddy allocation finds a block and the relocation algorithm is not called) we need not update the sub levels and just assume *empty* indicates the chunk being full because of some block of bigger size using it.

4.4.2 The Algorithm – Pseudo Code

1. *Initialization()*

   for $i = 0$ to $H - 1$

   $heapManager[i] = 0$; /* empty heap */

2. *Allocate (S)*

   if there is a free block of size S

allocate the block of size S with the lowest address, A

UpdateHeapManager(S, A, "allocation")

else search for a free block of size bigger than S in increasing order of size

if found, select the block with the lowest address

split the block recursively until there is block of size S

select the block of size S with the lowest address, A

UpdateHeapManager(S, A, "allocation")

else

A = FindMinimallyOccupiedBlock(S) /* finds block to relocate */

Relocate(S, A) /* relocates the sub blocks from block A*/

allocate the block with address A

UpdateHeapManager(S, A, "allocation")

3. *FindMinimallyOccupiedBlock(S)*

   find i such that $heapManager[i]$ is minimum for $i = 2H/S - 1$ to H/S

   return address $A = i << \log_2 S$

4. *Relocate(S, A)*

   $subBlocks$ = FindSubBlocks(S, A);

   for each $SB \in subBlocks$

   Deallocate(SB), $\forall SB \in subBlocks$

   /* allocates the subblocks in a location other than A */

   Allocate(SB, S, A), $\forall SB \in subBlocks$

5. *Deallocate(extId)*

   find address A of block $extId$ and size S;

   free the block;

   UpdateHeapManager(S, A, "deallocation");

6. *UpdateHeapManager(S, A, type)*

   int $maxLevel = \log_2 H$;

   int $level = \log_2 S$;

   if type= "allocation"

   int $addr = A >> level$;

   if $S > MinBlockSize$

$heapManager[addr] = S$ /* block is fully occupied */
/* blocks above the allocation level */
$addr = A >> level;$
for $(i = level + 1; i <= maxLevel; i++)$
$addr = addr >> 1;$
$heapManager[addr] = heapManager[addr] + S;$
if type= "deallocation"
int $addr = A >> level;$
/* current block */
if $S > MinBlockSize$
$heapManager[addr] = 0;$
/* blocks above the deallocation level */
$addr = A >> level;$
for $(i = level + 1; i <= maxLevel; i++)$
$addr = addr >> 1;$ //continuing from above addr
$heapManager[addr] = heapManager[addr] - S;$

4.4.3 Time and Storage Bounds

Theorem 4.4.5 *If H is the heap size, Heap Manager Algorithm needs $O(H)$ storage.*

Proof: Our heap-like data structure needs $H - 1$ bytes. ■

Theorem 4.4.6 *With $2M$-byte heap, defragmentation according to the Heap Manager Algorithm(Section 4.4.1) to satisfy a single allocation request of a s-byte block takes $O(Ms^{0.695})$ time.*

Proof: Let the allocation request be for a s-byte block, where $s < M$. If there is a free block of s bytes then that is allocated and the allocation is satisfied. If there is no free block of s bytes then the allocator first defragments a s-byte chunk of storage. According to the Lemma 4.4.1, there is a s-byte chunk with less than $s/2$ bytes live. As mentioned in Section 4.4.1, search for a minimally occupied s-byte block has to go through $2M/s$ numbers (in short we call the time taken is $2M/s$ units). After finding a minimally occupied chunk all the blocks in that chunk have to be relocated to other locations to free up the s-byte chunk. Each block to be relocated needs searching for a minimally occupied block (only in case there is no free block of that size, which we assume for the purpose of worst case analysis). Since a s-byte chunk is less than half

full the largest live block in that chunk is of $s/4$ bytes, and in worst case blocks of sizes $s/4, s/8, \ldots, 2, 1$ bytes can exist in that chunk.

Let $T(s)$ be the total number of time units taken for emptying the s-byte chunk including all the time taken for the relocations required. Searching for a minimally occupied block is not required for a 1-byte or a 2-byte block, so $T(1) = T(2) = 0$, $T(4) = 2M/4 = M/2$. Below, $F(i)$ denotes the ith Fibonacci number. Fibonacci series is given by the recurrence relation $F(s) = F(s-2) + F(s-1), F(0) = 0, F(1) = 1$. Also, $F(s) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^s - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^s$. Please see [9] for a proof. Note that log denotes $\log_2$.

$$
\begin{aligned}
T(s) &= \frac{2M}{s} + T(s/4) + T(s/8) + \ldots + T(4) \\
&= \frac{2M}{s} + \left\{\frac{2M}{s/4} + T(s/16) + T(s/32) + \ldots + T(4)\right\} + T(s/8) + \ldots + T(4) \\
&= \frac{2M}{s} + \frac{2M}{s/4} + T(s/8) + 2 * [T(s/16) + T(s/32) + \ldots + T(4)] \\
&= \frac{2M}{s} + \frac{2M}{s/4} + \frac{2M}{s/8} + 2 * T(s/16) + 3 * [T(s/32) + T(s/64) + \ldots + T(4)] \\
&= \frac{2M}{s} + \frac{2M}{s/4} + \frac{2M}{s/8} + 2 \cdot \frac{2M}{s/16} + 3 * T(s/32) + 5 * [T(s/64) + \ldots + T(4)] \\
&= \frac{2M}{s} \cdot \left[1 + 1 \cdot 4 + 1 \cdot 8 + 2 \cdot 16 + 3 \cdot 32 + \ldots + F\left(\log(s/4) - 1\right) \cdot 2^{\log(s/4)}\right] \\
&= \frac{2M}{s} \cdot \left[1 + 4 + 8 + 2 \cdot 16 + 3 \cdot 32 + \ldots + F\left(\log(s/8)\right) \cdot 2^{log(s/4)}\right] \\
&= \frac{2M}{s} \cdot \left[1 + 4\left[1 \cdot 1 + 1 \cdot 2 + 2 \cdot 4 + 3 \cdot 8 + \ldots + F\left(\log(s/8)\right) \cdot 2^{log(s/4)-2}\right]\right] \\
&= \frac{2M}{s} \cdot \left[1 + 4\left[F(1) \cdot 2^0 + F(2) \cdot 2^1 + \ldots + F(\log(s/8)) \cdot 2^{\log(s/16)}\right]\right] \\
&= \frac{2M}{s} \cdot \left[1 + 4\left[\sum_{i=1}^{\log(s/8)} F(i) \cdot 2^{i-1}\right]\right] \\
&= \frac{2M}{s} \cdot \left[1 + 4\left[\sum_{i=1}^{\log(s/8)} \left\{\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^i - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^i\right\} \cdot 2^{i-1}\right]\right] \\
&= \frac{2M}{s} \cdot \left[1 + 4\left[\frac{1}{10} \cdot \left\{(1+\sqrt{5})^{\log(s/8)+1} + (1-\sqrt{5})^{\log(s/8)+1} - 2\right\}\right]\right] \\
&= \frac{2M}{s} \cdot \left[1 + 4\left[\frac{1}{10} \cdot \left\{(1+\sqrt{5})^{\log(s/4)} + (1-\sqrt{5})^{\log(s/4)} - 2\right\}\right]\right] \\
&< \frac{2M}{s} \cdot \left[1 + \frac{2}{5}\left\{(1+\sqrt{5})^{\log(s/4)} + \frac{s}{4} - 2\right\}\right]
\end{aligned}
$$

$$
\begin{aligned}
&< \frac{2M}{s} \cdot \left[1 + \frac{2}{5}\left\{\left(\frac{s}{4}\right)^{1.695} + \frac{s}{4} - 2\right\}\right] \\
&= O(Ms^{0.695})
\end{aligned}
$$

■

Theorem 4.4.7 *With M-byte heap, defragmentation according to the Heap Manager Algorithm(Section 4.4.1) to satisfy an allocation request for a s-byte block takes $O(Ms)$ time.*

Proof: In worst case, every chunk of s bytes can occupied with $s-1$ bytes made up of $s/2, s/4, \ldots, 2, 1$-byte blocks. Searching for a minimally occupied s-byte chunk takes M/s units time. Please refer to the above proof for more explanation about relocation.

Let $T(s)$ be the total number of time units taken for emptying the s-byte chunk including all the time taken for the relocations required. Searching for a minimally occupied block is not required for a 1-byte so $T(1) = 0$, and $T(2) = M/2$.

$$
\begin{aligned}
T(s) &= \frac{M}{s} + T(s/2) + T(s/4) + \ldots + T(2) \\
&= \frac{M}{s} + \frac{M}{s/2} + 2\left[T(s/4) + T(s/8) + \ldots + T(2)\right] \\
&= \frac{M}{s} + \frac{M}{s/2} + 2 \cdot \frac{M}{s/4} + 4\left[T(s/8) + T(s/16) + \ldots + T(\frac{s}{s/2})\right] \\
&= \frac{M}{s}\left[1 + 1 \cdot 2 + 2 \cdot 4 + 4 \cdot 8 + \frac{s}{4} \cdot \frac{s}{2}\right] \\
&= \frac{M}{s}\left[1 + \frac{1}{3}\left(\frac{s^2}{2} - 2\right)\right] \\
&= \frac{M}{s} \cdot \frac{s^2 + 2}{6} \\
&= \frac{M}{6}\left(s + \frac{2}{s}\right) \\
&= O(Ms)
\end{aligned}
$$

■

The above two theorems give the worst case time bounds for the defragmentation to satisfy a single request. With $2M$ bytes heap defragmentation takes $O(M)$ time and with M bytes heap it takes $O(Ms)$ time to allocate a s-byte block. So

the system designer has to go for a trade-off between the amount of storage and the defragmentation time. If the time is a more important constraint then having $2M$ bytes storage is better than having only M bytes, and if the system can go easy on time then it is better to go with M-byte heap to keep the cost low.

Chapter 5 shows experimental results with heap of $2M$ and M bytes.

Chapter 5

Experiments

Storage requirements as well allocation and deallocation patterns vary from one program to another. Hence, storage fragmentation and the need for defragmentation varies as well. This chapter describes the implementation of the defragmentation algorithm presented in Section 4.4.1 and its application on some real benchmarks. To facilitate experimentation, we implemented a simulator that takes the allocation and deallocation information from a program trace[1] and simulates the effects of allocation and deallocation. The simulator uses address-ordered best-fit binary buddy algorithm to satisfy the program's allocation and deallocation requests. If it cannot satisfy an allocation request from the heap due to fragmentation (and not due to exceeding maxlive), then it carries out the defragmentation algorithm in Section 4.4.1 to create a sufficiently large free block to satisfy the request.

All the results mentioned in this chapter are based on Release 1.1 of the simulator.

In Section 5.1 information about the test programs is given. In Section 5.2 implementation of the simulator is explained and in Section 5.3 the results are given.

5.1 The Test Programs

Various programs with different allocation patterns were tested by the defragmentation algorithm. By instrumenting the allocator to print out the allocation and deallocation requests, we logged all the requests of the program which includes size

[1] This portion was implemented by Dante Cannarozzi, Steve Donahue and Matthew Hampton

of each request. We rounded the size of a requested block to its next power of 2, as our simulation uses buddy allocation algorithm.

As mentioned in Section 4.4.1 all the allocated blocks' information is maintained in separate storage of size maxlive. This is done to make the searching for the appropriate block faster if the need for defragmentation arises. To create space for block of size n our algorithm relocates minimally occupied chunk of size n. If we go through the whole heap it takes $O(M)$ time to find a minimally occupied of chunk. But if we have a heap-like data structure as in Section 4.4.1 to maintain the amount of live storage for each size the search time takes decreases to $O(M/n)$. Also, the overall defragmentation takes $O(Mn^{0.695})$ time with $2M$ bytes heap. For more details see Section 4.4.1 and the example in Figure 4.1. Note that the time taken is only for searching appropriate blocks to relocate without actually moving the storage, unlike the compaction algorithms which move all the storage to one end of the heap.

5.1.1 Java Test Programs

The allocation and deallocation information for the benchmarks was obtained by instrumenting the garbage collector in a **Java Virtual Machine** (JVM). Instead of the regular garbage collector, we used the Contaminated Garbage Collector [1]. Unlike mark and sweep collectors which mark live objects and assume all the unreachable objects to be dead, contaminated garbage collector associates each object with a stack frame such that it is collectable when the frame pops.

Some of the properties — notably maxlive and maximum number of objects live at any point — are specific to the Contaminated Garbage Collector. If the allocation and deallocation information were obtained using a different garbage collection technique, then these statistics would be different.

Figure 5.1 shows the properties of the SPEC benchmarks [4] that we used for the simulations: program size, maxlive, max objects live, number of allocation and deallocations, max and min block size requested by the program and overall storage allocated by the program in its life time.

The maxlive is obtained after rounding the allocation requests to next higher power of 2. Depending on the program and its size the maxlive varies from 411552 bytes for Check of size 1 to 15494888 bytes for Compress of size 10. We have not done many simulations for programs with size 100 because of the long time and storage they need.

Program (size)	Maxlive (bytes)	Maxlive (objects)	Allocations	Deallocations	Max Block size	Min Block Size	Overall Storage Allocated
check(1)	411552	4957	6296	1505	32768	8	559112
compress(1)	9972336	4605	5124	653	4194304	8	10021232
compress(10)	15494888	4657	5234	711	4194304	8	15553096
db(1)	500832	5482	7609	3111	32768	8	774696
db(10)	3064536	99626	122015	117521	524288	8	4958896
jack(1)	1955616	11314	209715	198890	32768	8	7050032
javac(1)	871432	17440	26115	8812	32768	8	1549808
javac(10)	2204600	55236	209715	155837	32768	8	8743504
jess(1)	638328	10132	45868	36377	32768	8	2355552
jess(10)	1177768	16910	106514	90036	32768	8	5510056
jess(100)	1377448	21428	209715	189410	32768	8	8974528
mpegaudio(1)	594232	7006	7551	679	32768	8	649768
mpegaudio(10)	742912	8487	9087	734	32768	8	817984
mtrt(1)	5341472	209435	209715	330	262144	8	5353776
raytrace(1)	5341704	209433	209715	332	262144	8	5354200

Figure 5.1: Java Test Program Characteristics.

Program	Maxlive	Max Objects Alive	Allocations	Deallocations	Max Block size	Min Block Size	Overall Storage Allocated
cfrac	34552	1521	1528	1230	512	8	36728
gawk	3680046	151465	874375	722920	8192	1	50344545
gs	673248	6195	108541	102384	32768	16	31689120
p2c	96802	1698	5479	4265	1024	2	222312
ptc	102304	2885	2885	0	1024	8	102304

Figure 5.2: C Test Program Characteristics.

5.1.2 C Test Programs

Similar to the above Java programs Figure 5.2 lists the properties of the C programs we used in our simulations.

5.2 Simulator

The simulator, implemented in Java programming language, reads the program trace — program's allocation and deallocation requests — from a file and executes each request. For example, an allocation request looks like "1 17 32", where '1' indicates it

is an allocation request, '17' is the identification number, in short *Id*, of the block being requested and '32' is the size of the block. Further down in the program trace this Id is used to request deallocation, if any, of the block. For example, the deallocation request for the block allocated above looks like "0 17", where '0' indicates it is a deallocation request and '17' is the Id of the block.

The simulator starts with a fixed size heap according to the given parameters. For example, for the experiments we use heap of size M and $2M$. It uses storage equal in size to the heap to keep track of density of the allocated blocks according to the Heap Manager Algorithm given in Section 4.4.1. It uses standard Java class HashMap to maintain mapping between the block Id and the address on the heap where it is actually located. If an allocation request is unsatisfiable due to fragmentation, then the simulator relocates part of the storage or compacts the heap to make space for the requested block. Its behavior is described in the following subsections.

5.2.1 Allocation

Allocation is according to the address-ordered best-fit buddy algorithm described in Section 2.4.2, where a block of smallest possible size equal to or greater than the required size is selected with preference to the lowest address block. When a block of required size is not available then a block of higher size is selected and split repeatedly until a block of required size is obtained.

5.2.2 Defragmentation

The algorithm used is the greedy algorithm given in Section 4.4.1. If there is no block of the requested size and larger, then it finds the minimally occupied block of the requested size and relocates the chunks to some other locations in the heap and uses the free block created to satisfy the allocation request.

5.2.3 Left-First Compaction

This is an alternative to defragmentation when there is no free block to satisfy an allocation request. It compacts all the storage to the lower end (left end as shown in the Figure 5.3), based on address, by filling up the holes with the closest block (to the right) of less than or equal size. An example is shown in Figure 5.3 where Figure 5.3(a)

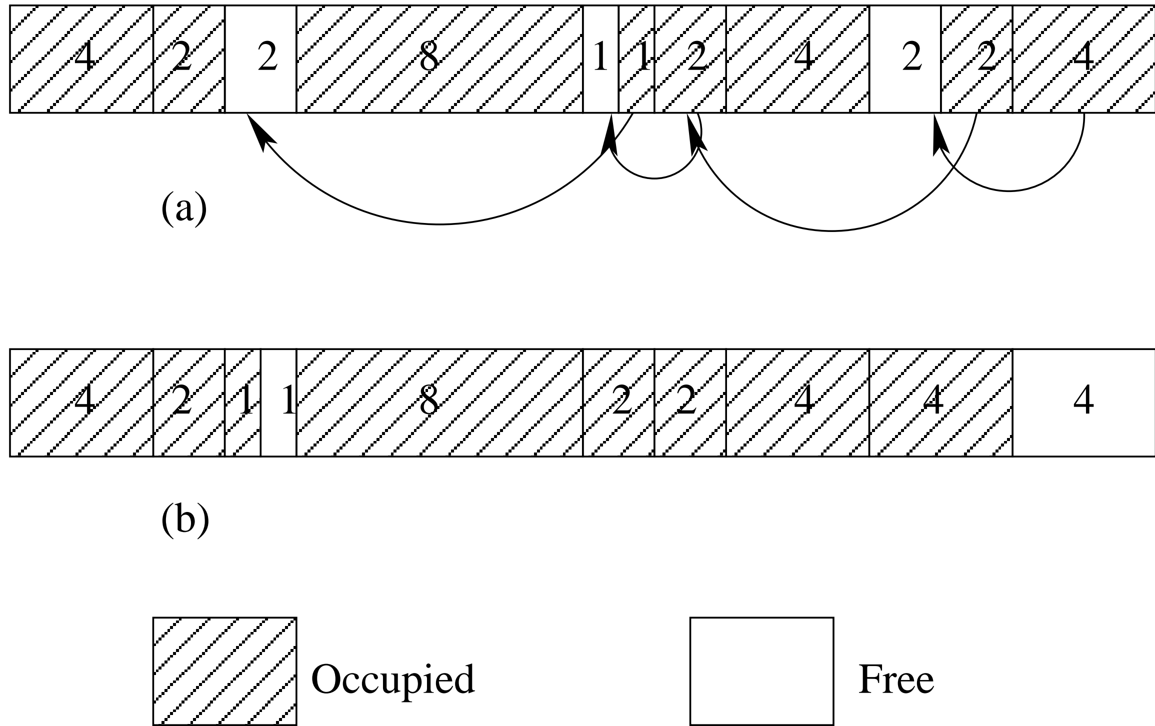


Figure 5.3: Left-First Compaction

and Figure 5.3 (b) gives the initial and final configuration. The movement of occupied blocks is indicated by arrows beginning from the *left*.

5.2.4 Right-First Compaction

Similar to left-first compaction in moving all the storage to the left end of the heap but it picks the blocks from the right end i.e., farthest addressed block of size less than or equal to the hole. An example is shown in Figure 5.4 where Figure 5.4(a) and Figure 5.4 (b) gives the initial and final configuration. The movement of occupied blocks is indicated by arrows beginning from the *left*. Since our buddy allocator uses address-ordered policy (left-first in the figures) the heap tends to be full on the left end of the heap. So compaction by moving blocks from the right end should move less memory than left-first compaction and hence be a tougher competitor to our defragmentation algorithm.

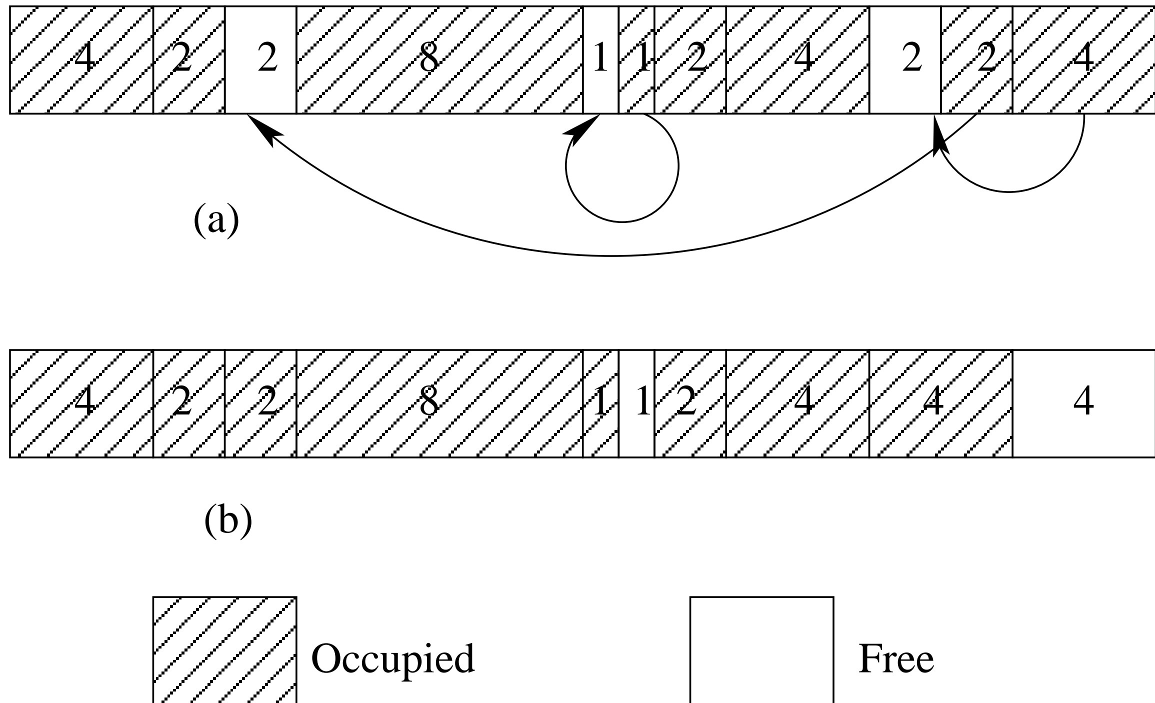


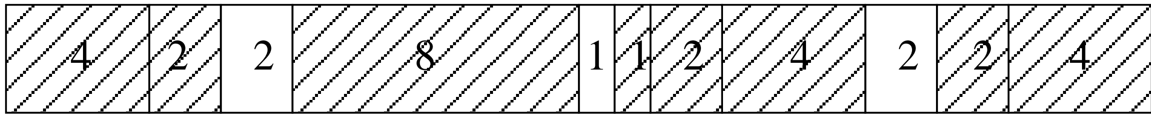
Figure 5.4: Right-First Compaction

5.2.5 Compaction Without Using Buddy Properties

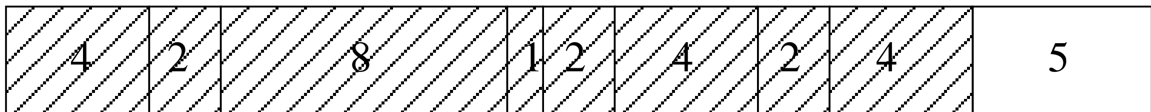
This method has been implemented to compare our defragmentation algorithm to a naive compaction method of sliding the blocks to one end of the heap, without following the buddy block properties similar to the general non-buddy allocators. An example is given in Figure 5.5.

5.3 Results

The most important part of this research is to provide bounded time guarantee for each and every allocation request. As we have seen in Section 4.4, the defragmentation algorithm does provide bounded time allocation and the amount of storage relocated during defragmentation is given for heaps of size M and $2M$ bytes. This section shows the results of the application of the defragmentation algorithm (Section 4.4.1) to see the overall amount of relocation that takes place in SPEC benchmark programs' life-time.



(a)



(b)

Figure 5.5: Non-Buddy Compaction

5.3.1 Defragmentation with $2M$-byte Heap

First we explored the defragmentation in various benchmarks with $2M$-byte heap using the algorithm described in Section 4.4.1, which is based on the theorem 4.4.3. To our surprise we found that none of the benchmarks needed any defragmentation when the heap size is $2M$ bytes! So having twice the maxlive storage, the address-ordered best-fit buddy allocator is able to avoid any relocation of the storage. Even some randomly generated program traces[2] did not need any relocation with a heap of size $2M$ bytes. Looking at these results, we wonder if fragmentation is a real problem. If having extra storage is not big issue, just by putting some extra storage, we might be able to get rid of fragmentation issue altogether. In a study [5], it has been shown that some well-known allocation algorithms have essentially zero fragmentation.

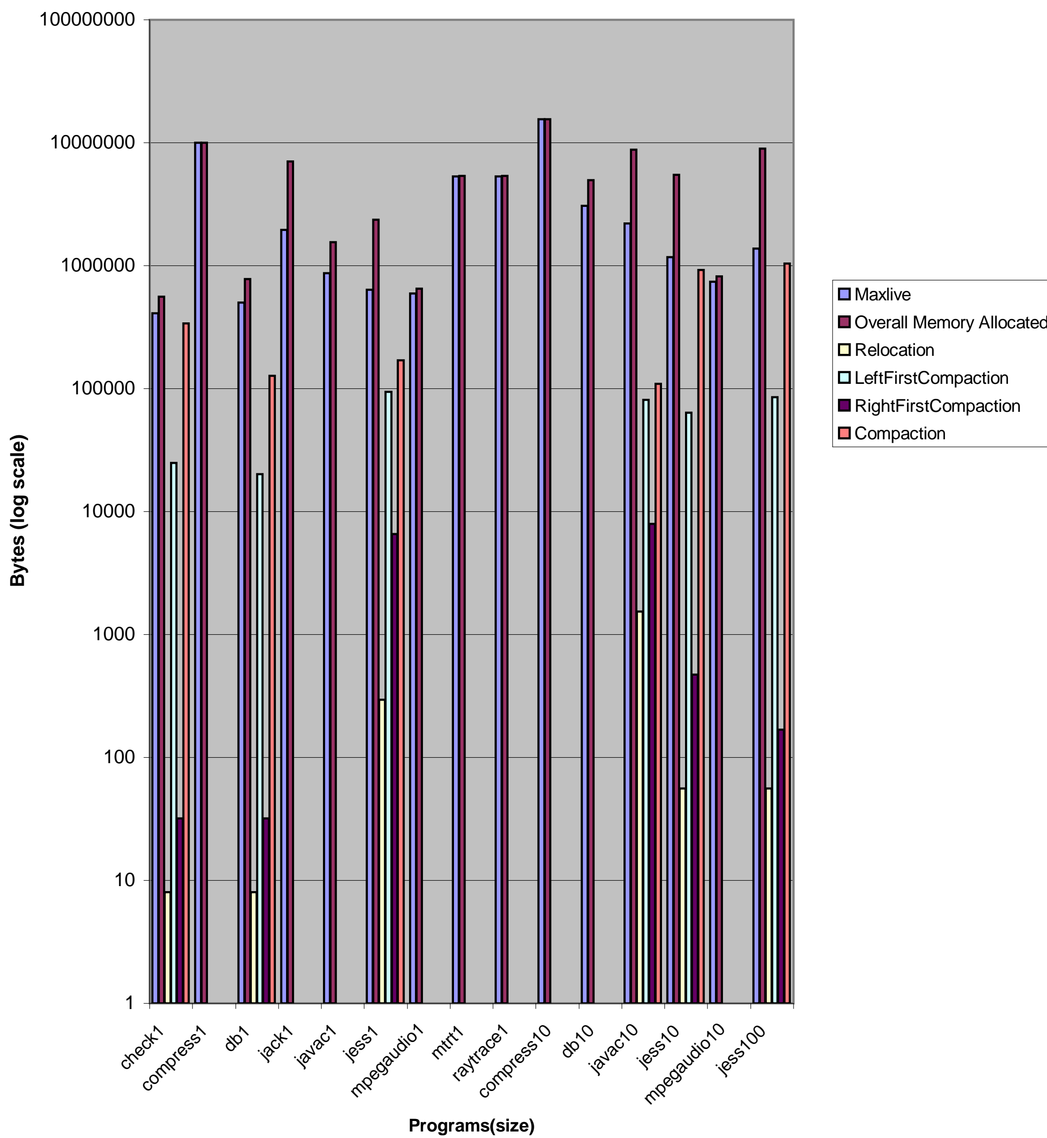


Figure 5.6: Java Test Programs – Amount of Relocation

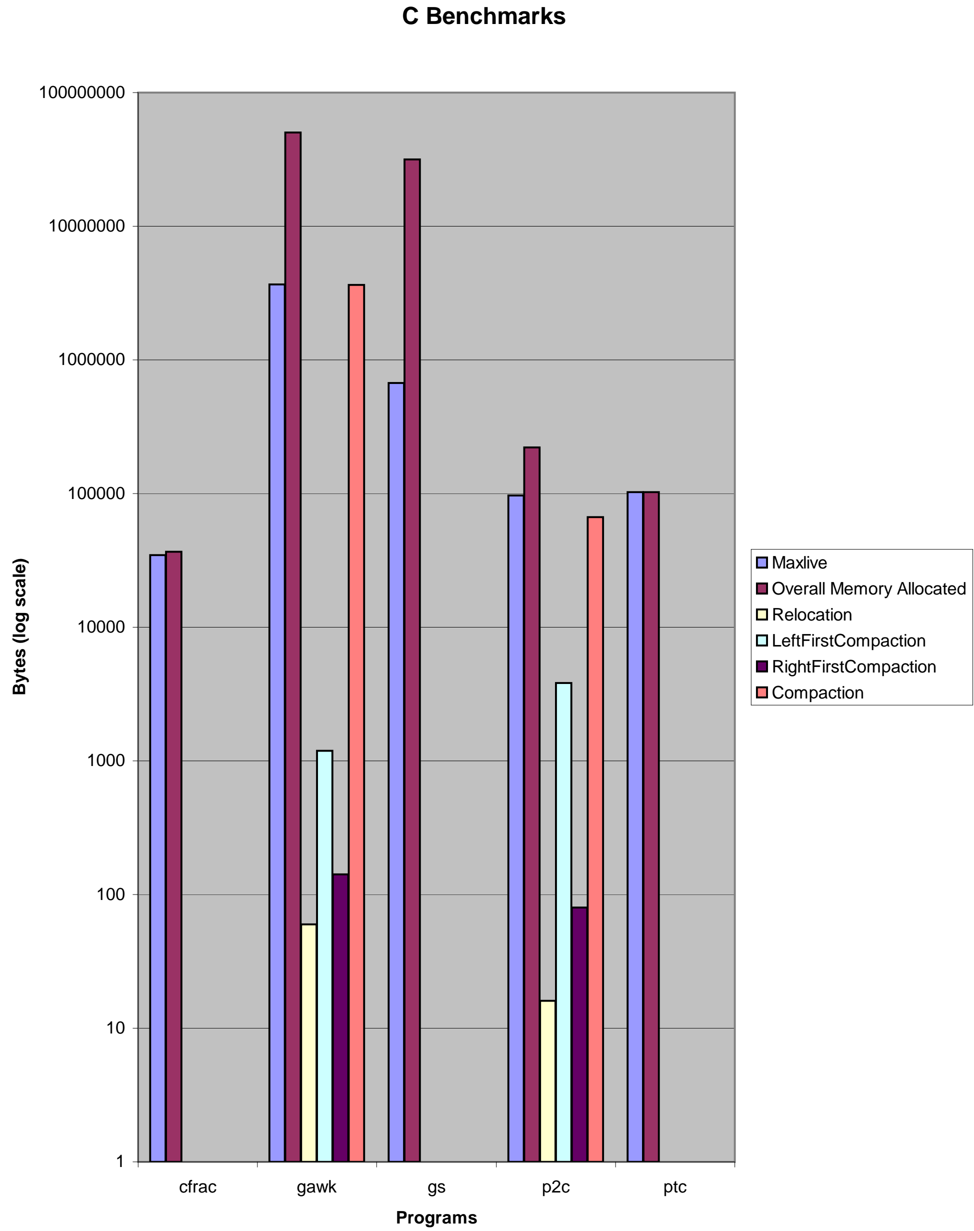


Figure 5.7: C Test Programs – Amount of Relocation

5.3.2 Defragmentation with M-byte Heap

After seeing that none of the benchmarks needed any defragmentation with $2M$-byte heap, we experimented with M-byte heap to see the effect of fragmentation and the performance of our defragmentation. But with only M-byte heap there is no guarantee about how much storage is relocated when compared to the optimal. From Figure 5.6 and Figure 5.7 we can see that very few programs required defragmentation! Among the programs which needed defragmentation, except for Jess of size 1 and Javac of size 10, the amount of storage relocated by our defragmentation algorithm for other programs is very insignificant. But compared to our algorithm which relocates storage only to satisfy a particular request without defragmenting the whole heap, all the compaction methods perform badly. The amount of storage relocated by our defragmentation algorithm is summed over all the relocations necessary, whereas for the compaction methods the amount is only for one compaction which is done when the allocator fails to satisfy an allocation request for the first time. Among all the compaction methods only right-first compaction (Section 5.2.4) performed reasonably well. The other two methods – left-first compaction (Section 5.2.3) and naive compaction of sliding all the live storage to one end (Section 5.2.5) – relocated lot more storage, which some times is almost equal to the maxlive of the program.

The above results, which showed the weakness of the compaction methods when compared to our defragmentation algorithm, indicate that it might be a good idea to go for localized defragmentation to satisfy a single allocation request instead of defragmenting the whole heap. If the defragmentation is needed for very few allocations (according to the amount of storage relocated as shown in Figure 5.6 and Figure 5.7 and the number of relocations as shown in Figure 5.9) there is no point in doing extra work by compacting the whole heap, either in anticipation of satisfying the future allocation requests without any defragmentation or for some other reason.

5.3.3 Minimally Occupied vs Random Block Selection

Here we compare the two different heuristics — selecting minimally occupied block and selecting a random block for relocation. They are compared using the amount of relocation and the number of relocations required, for various programs. Since not all the benchmarks need defragmentation as seen in Figure 5.6 and Figure 5.7,

[2] The program to generate a random allocation and deallocation sequence with a particular maxlive was implemented by Delvin Defoe

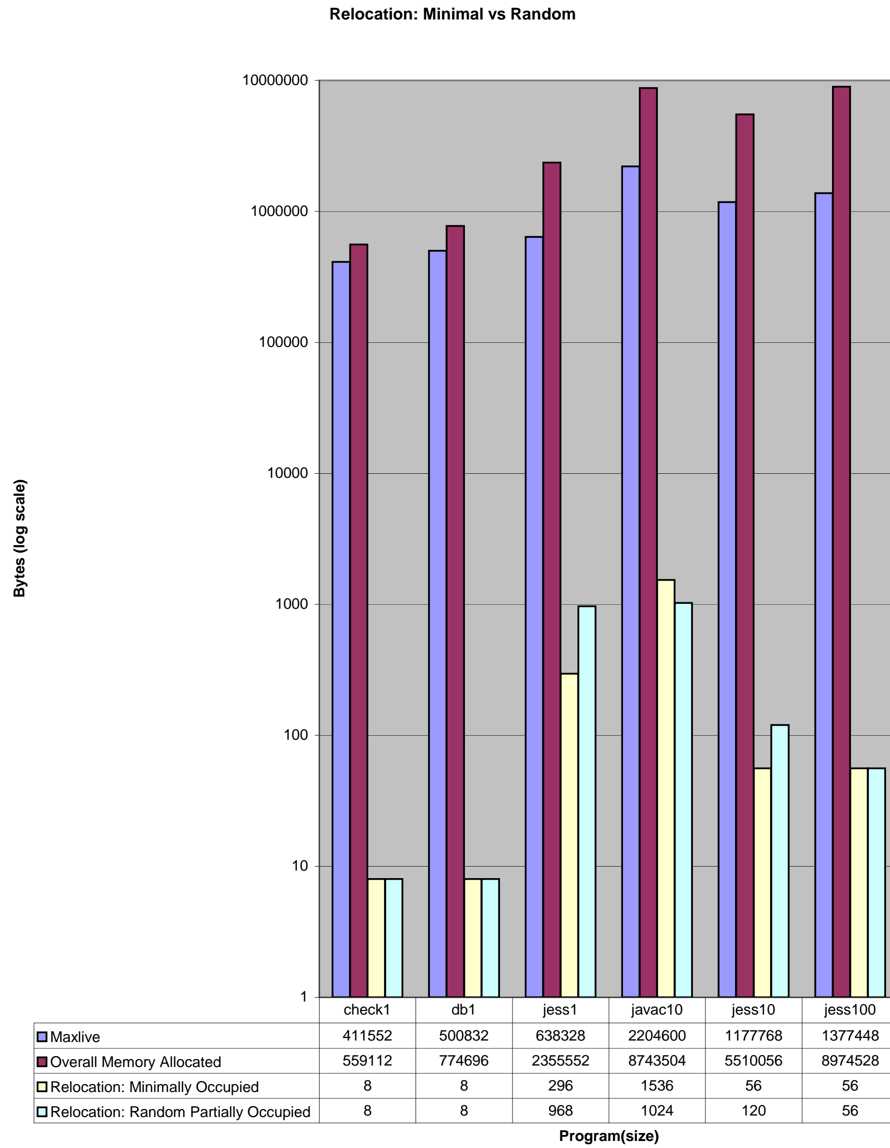
the random block selection heuristic is applied only on the programs that needed defragmentation.

From Figure 5.8 we see that out of the 6 Java SPEC benchmark programs which needed some defragmentation, for 3 programs (namely check(1), db(1) and jess(100)) the same amount of relocation is required for both the heuristics – selecting minimally occupied block and selecting a random block, for 2 programs (namely jess(1) and jess(10)) selecting minimally occupied block is better and for 1 program (namely javac(10)) selecting random block is better.

From Figure 5.9, 2 Java SPEC benchmark programs (namely check(1) and db(1)) needed same number of relocations while other 4 (namely jess(1), javac(10), jess(10) and jess(100)) needed different number of relocations. For all those 4 programs selecting minimally occupied block is better. Note that even though random selection needed more relocations for javac(10), the amount of storage relocated is less.

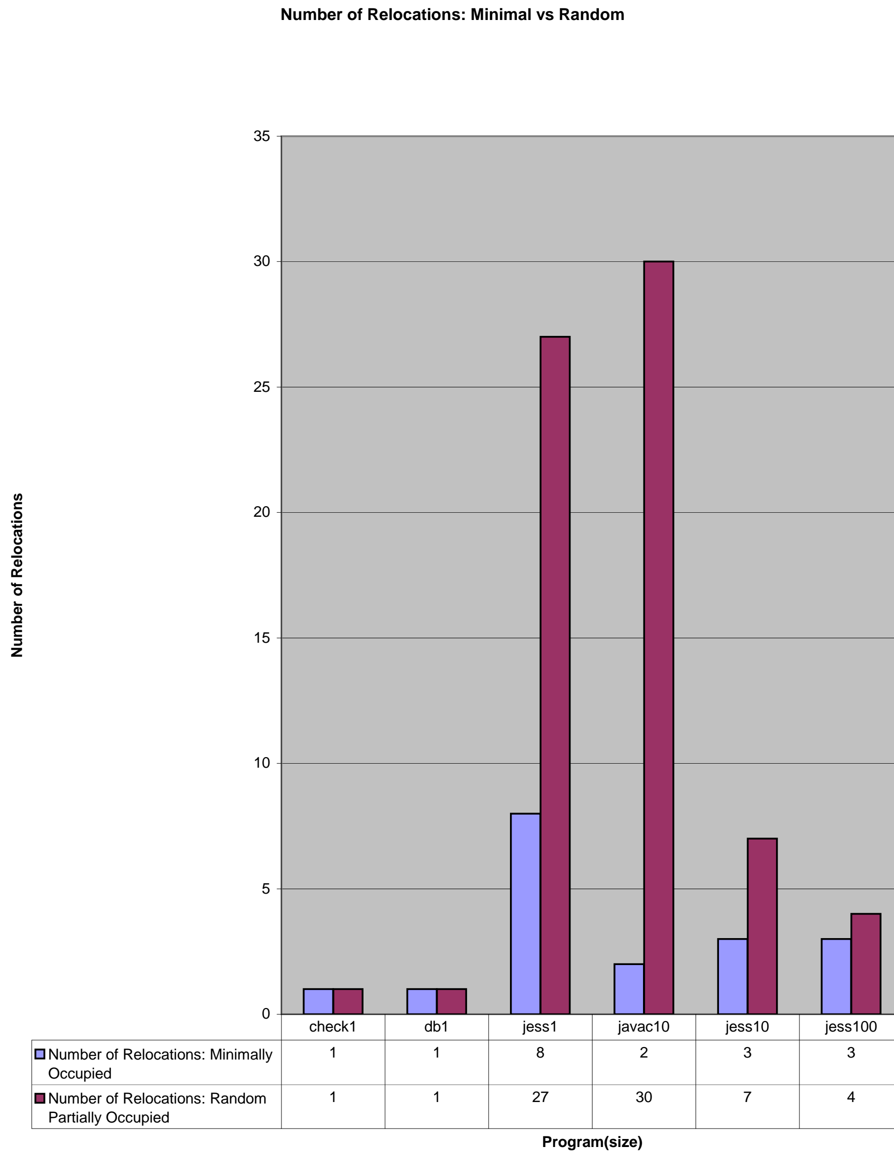
For the C benchmarks there is no significant difference between the two heuristics. So we have not listed them here.

The above results indicate that using the random block selection might be a good alternative. That would avoid the search time for the minimally occupied block. Since the search time is significant the system designer should determine if it is worth going for the minimally occupied block for a particular system depending on its needs.



	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

Figure 5.8: Minimally Occupied vs Random – Amount of Relocation



	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

Figure 5.9: Minimally Occupied vs Random – Number of Relocations

Chapter 6

Conclusion and Future Work

First we showed kM storage is sufficient for a defragmentation-free buddy allocator, where k is the number of different block sizes the program can request. We gave examples to show that $O(M)$ is not sufficient for a defragmentation-free buddy allocator. Then we proved a tight bound of $M(\log m + 2)/2$ for a defragmentation-free address-ordered buddy allocator.

We proved various theorems showing allocation of blocks in non-increasing order of sizes will never lock the allocator because of fragmentation, and that optimal relocation sequence can be sorted in increasing order of the block sizes.

Designed a greedy algorithm, with heap-like datastructure with bounded time guarantee of $O(Ms^{0.695})$ to allocate a s-byte block and less than twice optimal relocation, with $2M$-byte heap. We also gave the bounds with M-byte heap. Even though the programs' traces we experimented with did not need any relocation with $2M$, there is some relocation involved when heap size is M. The results show better performance of a localized defragmentation algorithm, which defragments just a small portion of the heap to satisfy a single request, over the conventional compaction algorithms.

Also, we compared the greedy algorithm with random selection heuristic to find that our greedy algorithm performed better, as expected. But since the overall fragmentation is low, random heuristic that takes lesser time might be good enough for some systems.

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

better data structures and algorithms to improve on the current ones. For example, instead of using linear search in each level of the data structure we can use heap sort to maintain each level, hence making the cost of searching a minimally occupied block $O(1)$ and cost of updating each level $O(\log N)$, where N is the number of nodes in that level.

Our experiments are based on program traces using Contaminated Garbage Collection [1] only. It would useful to conduct experiments with traces collected from other garbage collection methods like Reference Counting [12] and the mark and sweep garbage collectors [12]. The future work could involve the implementation of the defragmentation algorithm in JVM. That will give a better idea of actual time it takes and see if we can give exact bounds in micro or milli seconds it takes to allocate a block of particular size (for some specific machine configuration).

Also, studying the effects on cache when such a localized defragmentation algorithm is used would be interesting.

References

[1] Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 264–273. ACM Press, 2000.

[2] Benjamin Chelf. Dynamic memory allocation – part ii. *Linux Magazine*, July 2001.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[4] SPEC Corporation. Java SPEC benchmarks. Technical report, SPEC, 1999. Available by purchase from SPEC.

[5] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *International Symposium on Memory Mangement*, October 1998.

[6] Donald E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts, 1973.

[7] C. L. Liu and J. W. Layland. Scheduling algorithms for multi-programming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 1:40–61, January 1973.

[8] J.M. Robson. Bounds for Some Functions Concerning Dynamic Storage Allocation. *Journal of ACM*, 21(3):491–499, July 1974.

[9] Kenneth H. Rosen. *Discrete Mathematics and Its Applications.* McGraw-Hill, New York, NY, 4th edition, 1999.

[10] John A. Stankovic. Strategic Directions in Real-Time and Embedded Systems. *ACM Computing Surveys*, 28:751–763, December 1996.

[11] David Stepner, Nagarajan Rajan, and David Hui. Embedded application design using a real-time os. In *Proceedings of the 36th ACM/IEEE conference on Design automation conference*, pages 151–156, New York, NY, 1999. ACM Press.

[12] Paul R. Wilson. Uniprocessor garbage collection techniques (Long Version). Submitted to ACM Computing Surveys, 1994.

[13] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *International Workshop on Memory Management*, Kinross, Scotland, UK, September 1995.

Vita

Sharath Reddy Cholleti

Date of Birth	May 19, 1979
Place of Birth	Jangaon, Andhra Pradesh, India
Degrees	B.Tech. Computer Science and Engineering, 2000, from Indian Institute of Technology, Guwahati, India.
Publications	Sharath Reddy Cholleti and Sudeshna Sarkar. "Correlation based Neural Net Construction" in *Proceedings of 5th International Conference on Cognitive Systems*, New Delhi, India, Dec 1999.

December 2002