Upper Bounds for Dynamic Memory Allocation

Yusuf Hasan, Wei-Mei Chen, *Member*, *IEEE*,
J. Morris Chang, *Senior Member*, *IEEE*, and Bashar M. Gharaibeh

Abstract—In this paper, we study the upper bounds of memory storage for two different allocators. In the first case, we consider a general allocator that can allocate memory blocks anywhere in the available heap space. In the second case, a more economical allocator constrained by the address-ordered first-fit allocation policy is considered. We derive the upper bound of memory usage for all allocators and present a systematic approach to search for allocation/deallocation patterns that might lead to the largest fragmentation. These results are beneficial in embedded systems where memory usage must be reduced and predictable because of lack of swapping facility. They are also useful in other types of computing systems.

Index Terms—Dynamic memory allocation, memory storage, storage allocation/deallocation policies, first-fit allocator, garbage collection.

1 Introduction

Dynamic memory allocation is an active area of research in computer science for decades. There are many algorithms proposed to reduce memory storage or improve performance [11], [22]. In this paper, we concentrate on the upper bound of the dynamic memory storage for all allocators and the allocation pattern that leads to the maximum of memory usage. The derived upper bounds are beneficial in embedded systems where memory usage must be predictable because of lack of swapping facility.

1.1 Background

A computer program may need varying amounts of memory to perform its tasks depending on changes in its input or interactions with other external programs. A memory allocator, or just allocator in short, is a collection of subroutines (mainly `malloc` and `free` in C/C++) contained in a system library (libc.a in Unix) that is linked in with each program executable. An allocator can be implemented using one of several known algorithms. The allocator obtains memory from the operating system and provides it to its program in blocks of the specified sizes when requested by the program by means of calls to `malloc`. While a heap memory block is in use by a program, the block cannot be relocated to a different address in the heap space. The allocator takes a memory block back when the program calls the `free` subroutine and keeps it for future requests for memory by the same program. It may merge the freed block with any adjacent free blocks to create a larger block of free memory space and it might split a large free block to allocate part of it when requested by the program via `malloc`. Thus, allocators support program need for unforeseen amounts of memory or *dynamic* memory. Dynamic memory is also called by terms such as heap memory, heap space, heap memory space, store, storage, or just heap for historical reasons.

For the same input and to perform the same tasks on that input, a program will usually require the same amount of heap memory. The amount of heap memory, i.e., the number of bytes of heap memory, a program uses during the course of its execution can rise and fall as the program allocates and frees blocks of memory. The maximum amount of heap memory a program uses during a single run is called its "high water mark." We will also refer to it as program memory requirement. The amount of memory the program's allocator obtains from the operating system during a single run of the program is often referred to as its memory usage. Initially, when a program starts, its heap space consists of one contiguous block of free linear memory space. Due to the program's repeated and random allocation (via `malloc`) and release of memory (via `free`), the program's heap can become fragmented with noncontiguous and interspersed blocks of allocated and free memory. This harmful but mostly unavoidable phenomenon called fragmentation causes memory usage to overshoot the program's high water mark by a lesser or higher degree depending mainly on the severity of fragmentation. This paper tries to discover the upper bound of memory usage of an allocator when the program memory requirement is M number of bytes. The memory usage of an allocator is the sum of M and the extra memory needed due to fragmentation. The upper bound of memory usage of an allocator is the sum of M and the extra memory needed due to the worst possible fragmentation.

A good allocator uses an allocation algorithm that minimizes memory usage and finishes each allocation or deallocation task in the least amount of time possible, i.e., maximizes performance. Reducing memory usage is achieved mainly by reducing fragmentation as much as possible. In order to reduce fragmentation, allocators use coalescing of free adjacent blocks and allocation policies

- *Y. Hasan, J.M. Chang, and B.M. Gharaibeh are with the Department of Electrical and Computer Engineering, Iowa Sate University, Ames, IA 50011.*
- *W.-M. Chen is with the Department of Electronic Engineering, National Taiwan University of Science and Technology, No. 43, Sec. 4, Keelung Rd., Taipei 106, Taiwan, R.O.C. E-mail: wmchen@mail.ntust.edu.tw.*

Manuscript received 21 Sept. 2008; revised 13 May 2009; accepted 11 Sept. 2009; published online 30 Sept. 2009.
Recommended for acceptance by A. Zomaya.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-09-0479.
Digital Object Identifier no. 10.1109/TC.2009.154.

such as best-fit, first-fit, and several other documented algorithms. However, it has been found that there is a trade-off between memory usage and performance. An important purpose of dynamic memory research is to discover better allocation algorithms that achieve greater reduction in memory usage, and at the same time, deliver a higher performance than any known algorithms.

1.2 Paper Goals and Organization

We investigate the worst case memory usage or upper bound of memory usage of two different allocators. The first allocator can allocate memory blocks anywhere in the heap, while the second one uses the address-ordered first-fit allocation policy. It should be clear that the upper bound of memory usage of the first allocator is also the upper bound for all allocators because this allocator operates with the least constraints or restrictions. The upper bound of memory usage of this allocator is derived using an elementary mathematical method. On the other hand, the upper bound for the first-fit allocator is found using a systematic approach aimed at maximizing fragmentation. We describe an allocation/deallocation pattern that leads to the worst fragmentation. According to one published research [21], for a first-fit allocator, a store of about $M \log_2 N$ is sufficient where the total amount of memory used by the program is up to M and largest possible block size is N [20]; however, the worst case allocation pattern is still not clear and it is hard to describe the worst case behavior.

The contribution of this paper is twofold. First, we derive the upper bound of the dynamic memory storage for all allocators; second, we describe a worst case allocation-deallocation pattern for the first-fit allocator and derive the resulting memory usage. The upper bound can be used for multiple purposes. In an embedded system without disks for swapping, the upper bound shows the amount of memory required for programs to run without any risk of running out of memory. For a system without garbage collection [12], the minimum memory space required by a particular program to be able to run in all scenarios without running out of memory can be determined by the upper bounds derived from this research. In general-purpose computers also, all sorts of programs can benefit from the knowledge of worst case memory usage.

The rest of the paper is organized as follows: In Section 2, related work and background information are provided. Next, in Section 3, we describe the problem in detail. Then, in Section 4, we study the upper bound of memory usage for the allocator which is free to place the requested block anywhere in memory. In Section 5, we estimate the upper bound of memory usage for the addressed-ordered first-fit allocator and use a set of benchmark programs with different request patterns to validate the upper bound. Finally, Section 6 contains the conclusion of the paper.

2 Related Work

Several previously published allocators focus on the real-time applications. A dynamic memory allocator as part of the Ada runtime is proposed in [17]. The main goal of this algorithm is to provide allocation and deallocation of memory in a bounded time. The experiments conducted in [7] also show that segregating the free list by size can offer a reasonable bound on the allocation and deallocation time. It has been reported in [3] that a general-purpose allocator (the Lea's allocator [16]) performs as good as the custom allocators. This work also presented an implementation of region-based allocator which leads to higher performance. The aforementioned research papers deal with the speed issues of allocators. The issues of memory space usage are discussed in this paper.

Analyses of many different dynamic memory allocation algorithms have been performed [8], [19], [21]. As mentioned before, the lower bound of the worst case of memory usage is proportional to the amount of allocated data multiplied by the logarithm of the ratio between the largest and smallest block sizes, i.e., $O(\log_2(N/n))$, where N and n are the sizes of the largest and smallest allocated memory blocks, respectively [21]. In one publication [4], it has been shown that an allocator that achieves this lower bound is the pure or simple segregated storage allocator. A pure segregated allocator does not coalesce or split memory blocks once they are allocated. Therefore, once allocated a block's size cannot change and when freed, it cannot be split or merged with adjacent free blocks and used to satisfy a future program request for a memory block of different size. This allocator keeps a separate free list of freed blocks for each block size and the allowed block sizes are powers of 2 only. The number of such lists then is $\log_2(N/n) + 1$ and since each list can contain a maximum of M bytes only, the allocator's upper bound of memory usage is $M(\log_2(N/n) + 1)$. The upper bound will be reached if a program using such an allocator alternately allocates and frees M bytes worth of blocks of each allowed size from n through N.

The goals of a memory allocator are to minimize memory usage and maximize performance. Fragmentation is the chief problem of memory allocation because it leads to increased memory usage by a memory allocator [22]. The heap is fragmented when memory blocks freed by a program are noncontiguous. Two types of fragmentation are defined in literature: internal and external [13], [14]. Internal fragmentation refers to the difference between the requested block size and the allocated block size where the latter is slightly larger (4-8 bytes in efficient allocators). The allocated block may include a "header" for storing *metadata* about the block such as its size and allocated/free status. On the other hand, external fragmentation refers to the proliferation of free memory blocks that are not contiguous. Internal fragmentation is a property of the allocator and can be reduced to a large extent by efficient allocation algorithms [10], [16]. For example, an allocator with block sizes that are multiples of eight will have much lower internal fragmentation than an allocator such as the binary-buddy allocator [6] with block sizes that are powers of 2. All references to fragmentation in this paper mean external fragmentation. Fortunately, for most (but not all) practical allocation/free patterns observed in actual programs, good allocators such as first-fit and best-fit allocators display low fragmentation, and consequently, low memory usage [1], [11]. Moreover, the first-fit allocator tends to have the best speed performance due to its simplicity. Therefore, the first-fit allocator is among the most commonly used ones today and is discussed further in this paper.

Allocators manage the heap space of a program. There are many strategies developed for allocators to improve their speed and memory usage. More details on these algorithms are available in literature [14], [22]. Each allocation algorithm has its own limitation. Most theoretical analyses and empirical evaluations for allocators are derived with assumptions of randomness which are based on the behavior of real applications [5], [8], [15], [18], [19], [20], [21]. While the theoretical upper bounds are of great interest in the research community, the mathematical analysis of allocation algorithms has proved to be quite challenging [14], [18]. This paper focuses on the analysis of memory space usage for allocators. More specifically, we intend to derive the upper bound of first-fit allocator.

3 Problem Statement

If a program heap memory requirement is M bytes and the size of the smallest memory block allocated is n bytes, what is the amount of memory that is necessary and sufficient to satisfy the program's memory requirement even in the worst case of fragmentation possible? The answer depends on the memory request and release pattern generated by the program and allocation policy of the memory allocator. Clearly, a memory allocator for this program will need at least M bytes of memory. But due to fragmentation of the heap space, it will usually need more as mentioned earlier. The amount of memory used by an allocator when a particular program's memory requirement is M bytes and the size of the smallest memory block that can be allocated is n bytes is called its memory usage. The maximum amount of memory that can be used by an allocator for any program when the program memory requirement is M bytes and the size of the smallest memory block that can be allocated is n bytes is called the upper bound on memory usage of the allocator.

Finding lower and upper bounds of algorithms whether in terms of memory usage or performance is a classical and fundamental area of research in computer science. Memory usage and performance properties of many algorithms such as searching and sorting algorithms (binary search, quick sort, shell sort, bubble sort, etc.) have been proven through mathematical analysis as well as experimental verification. These results have enabled researchers to devise better and better algorithms and empowered programmers to easily identify the best known algorithm for a given task. In this paper, our purpose is to discover the upper bounds of memory usage for different types of allocators. We don't consider the lower bound because that is already quite obvious and equal to the program's memory requirement.

We study upper bounds of memory usage for two different allocators. In the first case, we consider a general allocator that can allocate memory blocks anywhere in the available heap space. Since this allocator is free to place the block anywhere in the heap, it can cause the maximum fragmentation possible. And the memory usage obtained in this case implies that the amount is sufficient for all allocators. In the second case, we work on the more economical allocator constrained by first-fit allocation policy. The first-fit allocation policy was found to be among the most effective in minimizing memory usage as well as in increasing performance [11], [22].

Most modern allocators use 16 bytes for the minimum block size. The header of the memory block needs 4 bytes to store the block-size, 4 bytes to point to the next free-block in the free linked list, and another 4 bytes to point to the previous free-block in the free linked list. So, we need at least 12 bytes. For portability, we align it on a 8-byte boundary (as required by processors such as the Sun SPARC processor), and therefore, the minimum block size becomes 16. The value of n is usually 8 which allows for block sizes of 16, 24, 32, 40, and all other multiples of 8. Windows XP memory allocator and Lea's allocator [16] are examples of modern allocators that allocate memory blocks in sizes that are multiples of 8 [2], [3]. Linux dynamic memory allocator is based on Lea's allocator [9].

4 An Upper Bound of Dynamic Memory Allocation

In this section, we consider an allocator that can allocate and deallocate memory blocks anywhere in the heap space so that it causes maximum fragmentation and reaches the upper bound of memory usage. Let M be the number of bytes of dynamic memory used by a program, n be the smallest allocated block size, and the memory usage be the sum of M and the extra memory needed due to fragmentation. We assume that all allocation blocks are multiples of n in sizes (as most modern allocators round up requested size for portability; see [2], [3], [16]) and the largest possible allocation block has a size of M, where M is also a multiple of n. If n is small (8 or 16), as it is in most modern allocators, the last assumption changes M by an insignificant amount (maximum $n-1$) for convenience in analysis without detracting from the practical utility of the derived results.

Initially, the allocator obtains contiguous heap memory space from the operating system. Let this be S bytes initially equal to M. Then, the heap memory space S is fragmented by the allocated and the freed memory blocks. We intend to find the largest possible memory usage that can result from all possible allocation/deallocation patterns under the M and n constraints. The allocator can place a requested memory block anywhere within S if there is a free block large enough within S. Only when there is no free block large enough to accommodate, the requested block can the allocator obtain more memory from the OS thereby increasing S which is the same as memory usage here. For this allocator, we need to find a pattern for allocation and deallocation that will repeatedly cause S to increase because of fragmentation until it can be increased no further. At that maximized value of S, we will have found our upper bound of memory usage for this allocator as well as for all allocators.

Intuitively, the maximum fragmentation may be caused by many small freed blocks. One way to achieve this is to have many small allocated blocks scattered across the heap in a noncontiguous fashion. The strategy used by the allocator to maximize the amount of memory usage is the following. We suppose that there are p noncontiguous allocated blocks each of size n. This implies that the given memory space is broken up into $p+1$ fragmented free blocks and the memory amount left to be allocated by the

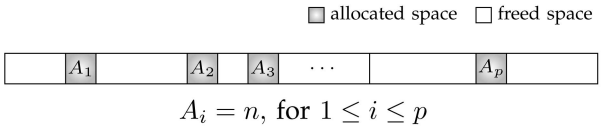

Fig. 1. Memory space layout.

running program is $M - pn$. The layout of memory space used is shown in Fig. 1. If the size of the next allocation request is $M - pn$ and the size of the largest freed chunk is less than $M - pn$, we must obtain more memory from the OS to extend the used memory by the requested size.

A scenario called *insufficient* is defined as one where the size of the next allocation request is greater than the size of the largest free heap memory block. Assume that every allocation is to be aligned on an address which is multiple of n. Next, we will find the maximum amount for memory storage needed in an insufficient scenario.

Lemma 1. *Let M be the maximum amount of memory space used by a program and n be the smallest allocation block size. If T is the total amount of memory storage in an insufficient scenario, then*

$$T \le \frac{M^2}{4n} + \frac{M}{2} - \frac{3n}{4},$$

for all allocators.

Proof. Suppose that an insufficient scenario occurs after a sequence of allocations and deallocations. Fig. 2 is the snapshot of the memory space just before the allocation request whose size is larger than those of all available blocks.

Suppose that there are p allocated sections. Let A_i be the size of the ith allocated block for $1 \le i \le p$ and B_j be the size of the jth freed block for $1 \le j \le p+1$. Then, we have

1. $n \le A_i \le M$ for $1 \le i \le p$;
2. $B_1, B_{p+1} \ge 0$ and $B_j \ge 1$ for $2 \le i \le p$; and
3. $T = \sum_{i=1}^{p} A_i + \sum_{j=1}^{p+1} B_j$.

Since the size of the coming allocation request can be up to $M - \sum_{i=1}^{p} A_i$, we have $B_j \le M - \sum_{i=1}^{p} A_i - n$. Then,

$$\begin{aligned} T &\le \sum_{i=1}^{p} A_i + (p+1)\left(M - \sum_{i=1}^{p} A_i - n\right) \\ &= (p+1)M - p\sum_{i=1}^{p} A_i - np - n \\ &\le (p+1)M - p(pn) - np - n, \end{aligned}$$

since $A_i \ge n$ for $1 \le i \le p$. This corresponds to the case of the obtained memory space which is divided into $p+1$ pieces by p blocks each of size n. So, $T \le Mp + M - np^2 - np - n$. Let $f(x) = Mx + M - nx^2 - nx - n$, then we have $f'(x) = M - 2nx - n$ and $f''(x) = -2n$. And the maximum of $f(x)$ occurs at $x = \frac{M-n}{2n}$ since $f'(x) = 0$ and $f''(x) < 0$ for all $n > 0$. Thus,

$$T \le f\left(\frac{M-n}{2n}\right) = \frac{M^2 + 2nM - 3n^2}{4n} = \frac{M^2}{4n} + \frac{M}{2} - \frac{3n}{4}.$$

Hence, we find the maximum amount for memory usage under an insufficient scenario. □

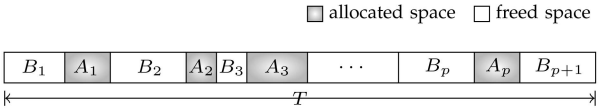

Fig. 2. Memory space layout under an insufficient scenario.

Theorem 1. *Let M be the maximum amount of memory space used by the program and n be the smallest allocation block size. If S is the minimum amount of memory storage sufficient for all allocators, then*

$$S = \frac{M^2}{4n} + \frac{M}{2} + \frac{n}{4}.$$

Proof. By Lemma 1, the worst case occurs when the memory space is broken into $\frac{M-n}{2n} + 1$ available blocks by $\frac{M-n}{2n}$ freed blocks of size n. And the size of the largest free memory chunk is less than the size of the newly arrived allocation request (i.e., $\frac{M+n}{2}$) by n. If we could expand one of largest free memory chunks at least by size n, the next allocation request would be met. Thus,

$$S = \frac{M^2}{4n} + \frac{M}{2} - \frac{3n}{4} + n = \frac{M^2}{4n} + \frac{M}{2} + \frac{n}{4}.$$

This completes the proof. □

Hence, the upper bound of memory storage for all allocators used is $\frac{M^2}{4n} + \frac{M}{2} + \frac{n}{4}$, where M is the maximum amount of memory used by the program at any time and n is the size of the smallest allocated memory block and all blocks are multiples of n.

5 Estimations of Memory Usage for Addressed-Ordered First-Fit Allocators

In this section, we consider an allocator that must allocate freed memory blocks at lower addresses before freed blocks at higher addresses, if multiples blocks of the requested or larger sizes exist in the heap space. As in the previous section, let M be the number of bytes of dynamic memory used by the program and n be the smallest allocation block size. All allocation block sizes are multiples of n and the maximum allocation block size is M, where M is also a multiple of n.

Initially, the allocator obtains some heap memory space from the operating system. Then, the memory is fragmented by allocation and deallocation requests without violating the constraint that the allocator must allocate available memory blocks with lower addresses before those with higher addresses. We try to find an allocation/deallocation pattern that leads to the worst fragmentation under the M and n constraints. The pattern is based on the heuristic that each freed space may not be reused in future allocations, thus effecting incremental increases in the size of heap.

The problem of finding the upper bound of memory usage may also be seen as a game between the program and the allocator. The program's memory requirement is fixed at M bytes and the allocator's allocation policy is also fixed and known to the program. The program's aim in the game is to force the allocator to increase its memory usage as much as possible. The program achieves its aim by

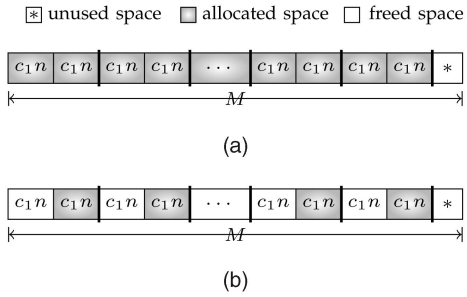

Fig. 3. Memory space layouts during the first cycle. (a) The allocation phase. (b) The deallocation phase.

maximizing fragmentation through the best allocation and deallocation pattern it can find for this purpose. The allocator, on the other hand, aims at keeping memory usage as low, i.e., close to M as possible using it allocation policy, and coalescing and splitting of memory blocks to keep fragmentation in check. Supposing that to achieve its aim, the program always succeeds in finding the best allocation/deallocation pattern that exists, the game is really a test of the allocator's allocation policy. The resulting memory usage is the allocator's upper bound on memory usage for there exists no other program or allocation/deallocation pattern that can force its memory usage beyond the upper bound. The upper bound is expressed in terms of M and n, the two variables.

5.1 A Simple Approach to Increasing Fragmentation

Consider a program that tries to fragment the heap by allocating and deallocating blocks in the following pattern for the purpose of maximizing memory usage. The allocation/deallocation request pattern extends memory usage cycle by cycle. Each cycle is composed of two phases: the allocation phase and deallocation phase. The program first allocates a pair of blocks of size c_1n (for an integer c_1) repeatedly when the total amount is not greater than M, as shown in Fig. 3a. To extend the memory usage, it then frees all the blocks that are the first of each pair. The heap snapshot after the deallocations is shown in Fig. 3b. The unused and freed space in the Fig. 3b is available for allocation in the next cycle. In each of the following cycles, memory usage is increased again with allocation blocks whose sizes are greater than the blocks freed in the previous cycles. The allocation and deallocation requests are generated by the pseudocode in Fig. 4.

It should be noted that the allocator does its utmost to minimize memory usage by following its first-fit allocation policy and splitting and coalescing memory blocks whenever possible but the program's allocation/deallocation pattern forces the increase in memory usage in spite of that. In terms of game playing, both the program and the allocator are playing their very best moves possible under the given constraints.

Suppose that c_in is the size of allocated/free blocks in the ith cycle where $\{c_i\}_{i\geq 1}$ be a sequence of integers where $c_1 < c_2 < c_3 < \cdots$. Let δ_i be the size of the unused space after the last allocation in the ith cycle, that is, $\delta_i =$ $M -$ (the sum of the amount of allocated blocks). Thus, after the first cycle, the extended amount is $M - \delta_1$ and

```
i ← 1            //i: the ith cycle
S_t ← 0          //S_t: the amount of memory usage
M_avail ← M
repeat
    p_i ← ⌊M_avail/(2c_in)⌋
    for k = 1 to p_i do //p_i: the number of pairs
        allocate two blocks of size c_in
    end
    S_t ← S_t + 2c_inp_i
    for k = 1 to p_i do
        deallocate the (2k − 1)th block of size c_in
    end
M_avail ← M_avail − c_inp_i
i ← i + 1            // for the next cycle
until (M_avail < 2c_in )
S_t ← S_t + M_avail
```

Fig. 4. A basic strategy.

the freed amount is $\frac{M-\delta_1}{2}$. So, the available amount becomes $\frac{M-\delta_1}{2} + \delta_1 = \frac{M+\delta_1}{2}$. Similarly, after the second cycle, we have that the extended amount is $\frac{M+\delta_1}{2} - \delta_2$ and the available amount is $\frac{M+\delta_1+2\delta_2}{4}$. The memory usage after the second cycle is shown in Fig. 5. And the program extends the memory usage repeatedly m times if the available space is not enough for the $m+1$ cycle, that is,

$$\frac{M + \delta_1 + 2\delta_2 + 4\delta_3 + \cdots + 2^{m-1}\delta_m}{2^m} < 2c_{m+1}n.$$

Hence, the total amount of memory usage is the sum of the amounts of the extended space in each cycle, that is,

$$\begin{aligned} M - \delta_1 &+ \frac{M + \delta_1 - 2\delta_2}{2} + \frac{M + \delta_1 + 2\delta_2 - 4\delta_3}{4} \\ &+ \frac{M + \delta_1 + 2\delta_2 + 4\delta_3 - 8\delta_4}{8} + \cdots \\ &+ \frac{M + \delta_1 + 2\delta_2 + \cdots + 2^{m-2}\delta_{m-1} - 2^{m-1}\delta_m}{2^{m-1}} \\ &= \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{m-1}}\right)M - \frac{\delta_1}{2^{m-1}} - \frac{\delta_2}{2^{m-2}} \\ &- \cdots - \delta_m < 2M. \end{aligned}$$

To increase the memory usage as much as possible, c_i should start from the minimum number (i.e., $c_1 = 1$) and increase very slowly so that more cycles will be needed.

5.2 The Relation between the Increment of Allocation Request Size and Memory Usage

To study the effect of increasing allocation request size to the memory usage, we simulate different patterns of size

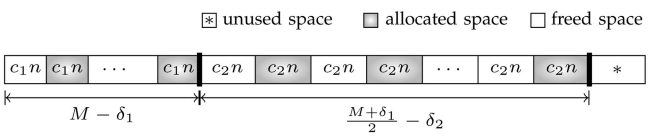

Fig. 5. Memory space layout after the second cycle.

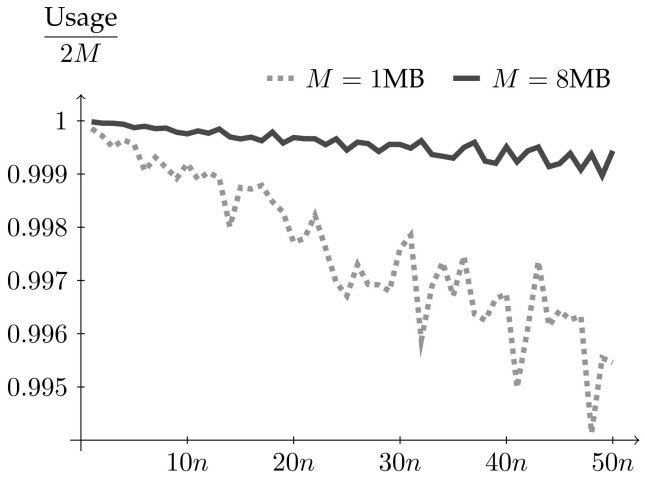

Fig. 6. The relation of increasing allocation request size and the memory usage where n is 8 bytes.

increment. It starts with n which means in each cycle, the requested block size increases by n (i.e., $n, 2n, 3n, 4n, \ldots$, etc.). The next increment pattern is $2n$ which means that in each cycle, the requested block size increases by $2n$ (i.e., $n, 3n, 5n, 7n, \ldots$, etc.), and so forth.

Fig. 6 shows the empirical results from a software simulation of the memory model using the simple strategy described in previous subsection. The program starts by allocating M/n blocks of size n and ends when the memory remaining to be allocated cannot be made greater than the largest contiguous free space, and therefore, no further increase in memory usage is possible. In every new cycle, the block sizes are increased by $n, 2n, 3n, \ldots$, respectively. From the ratio of the memory usage to $2M$ shown in Fig. 6, we observe that the memory usage for the increment of size n is slightly greater than the other cases, while the differences are insignificant.

It is worth noting that the ratio of deallocated memory to allocated memory is exactly $1/2$. With this ratio, after deallocation, each freed block is equal in size to the allocated block in the memory layout (see Fig. 3b). In the next section, we will consider the memory usage when a different ratio of deallocation to allocation is used.

5.3 The Relation between the Ratio of Deallocation to Allocation and Memory Usage

The strategy we have applied so far is based on the idea that the next allocation request size must be bigger than the sizes of the previously deallocated blocks. Thus, the next allocation request will not fit in any of the previously freed space and the memory usage will grow greater in each cycle. The more cycles we have, the more will the memory usage grow. In order to maximize the number of cycles, the block size in the $(i+1)$th cycle will be made greater than the freed block size in the ith cycle by no more than n bytes, the smallest increment possible since block size must be a multiple of n.

In this section, we investigate the deallocation to allocation ratio as $(r-1)$ to r, where r is greater than or equal to 2. Apparently, as the ratio gets larger, the amount of memory available to allocate gets larger in each cycle.

TABLE 1
The Relation of the Ratio of Deallocation to Allocation and Memory Usage Where M Is 1 MB and n Is 8 Bytes

ratio	number of cycles	memory usage
1 : 2	14	2097000
2 : 3	10	3103728
3 : 4	8	3838312
4 : 5	7	4296864
5 : 6	6	4480936
6 : 7	5	4383136
7 : 8	5	4546504
8 : 9	5	4478464
9 : 10	5	4782064
10 : 11	4	4360176
11 : 12	4	4342152
12 : 13	4	4446496
13 : 14	4	4504912
14 : 15	4	4452144
15 : 16	4	4225816
16 : 17	4	4395264
17 : 18	4	4560640
18 : 19	3	3869392
19 : 20	3	3858752
20 : 21	3	3873696

However, the number of the cycles will be less when the sizes of deallocated blocks increase. The final value of memory usage will be determined by the ratio and the number of cycles. We present the relation between ratio, number of cycles, and final memory usage in Table 1.

For any ratio $t = (r-1)/r$, the memory usage will be

$$\begin{aligned} S &< M(1 + t + t^2 + t^3 + \cdots + t^{\log_r M - 1}) \\ &< \frac{M(1 - t^{\log_r M})}{1 - t}. \end{aligned}$$

For t values of $1/2$, $2/3$, $3/4$, or $4/5$, it can be calculated that $S < 6M$. And that maximum value of S occurs when t is $8/9, 9/10$, or $10/11$ for M between 1 and 120 MB.

From Table 1 and Fig. 7, we can see that the memory usage increases when the ratio is in the range from $1/2$ to $9/10$, then the memory usage oscillates when the ratio is

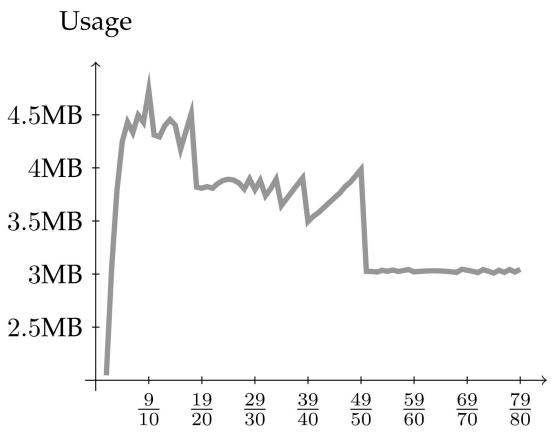

Fig. 7. The ratio of deallocation to allocation: n is 8 bytes and M is 1 MB.

```
p_1 ← M/n   // the first cycle
allocate p_1 blocks of size n
for k = 1 to M/n do
    if (k ∤ 2) then
        deallocate the kth block
end
for i = 2 to ℓ − 2 do   // the ith cycle, i ≥ 2
    p_i ← M/(2^i n)
    allocate p_i blocks of size 2^{i−1} n
    for j = 1 to i do
        for k = 1 to p_j do
            if (k ∤ 2^{i−j+1}) then
                deallocate the kth block of size 2^{j−1} n
        end
    end
end
allocate one block with size 2^{ℓ−1} n (= M/2)
```

Fig. 8. A strategy with blocks with sizes of n, $2n$, $4n$, $8n$, $16n$, ...

between $10/11$ and $17/18$. But when the ratio is greater than $18/19$, the memory usage starts to become smaller. This is because the number of cycles gets smaller. There are not enough cycles to generate more fragmentation, and thus, the memory usage is lower. We can easily find the maximum memory usage at the ratio of 9 to 10 (the memory usage is much greater than the ratio of 1 to 2) when M is 1 MB and n is 8 bytes. Moreover, it shows a similar behavior for different values of M and n, though the optimal ratio is not the same. In the next section, we will examine other approaches to achieve even greater memory usage.

5.4 Release of Memory Allocated in Previous Cycles

In the previous section, we made the size of next allocation request just greater than the size of the freed space left by the last deallocation. This approach frees half (or less or more depending on the ratio) of the allocated memory space in the immediately preceding cycle. In this section, we introduce a method that tries to free more allocated space in all the previous cycles. Again, to achieve the maximum memory usage, the allocation request size should be larger than any freed space in any previous cycle.

Suppose that the size of the next allocated block is x. To extend the memory usage, x must be larger than the sizes of any freed blocks so far. Examining the memory layout carefully, we observe that there is still more allocated space in previous cycles, besides the immediately preceding cycle, that can be freed without allowing the resulting sizes of the contiguous freed spaces to become more than or equal to x. Freeing this memory increases the size of the memory left to allocate in the present cycle, and thus, forces a larger increase in memory usage. The number of allocation/deallocation cycles is also increased.

Let $M = 2^\ell n$ for an integer $\ell > 0$. Next, we consider memory requests that only involve the blocks with sizes of $n, 2n, 4n, 8n, 16n, \ldots, 2^\ell n$. We describe the allocation and deallocation patterns explicitly in Fig. 8 though a similar one is also discussed in [20]. An example for the case of $M = 16$ and $n = 1$ is shown in Fig. 9.

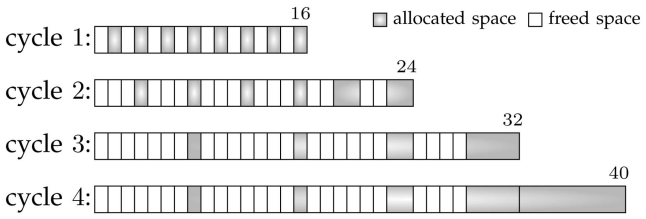

Fig. 9. The progression of memory usage for $M = 16$ and $n = 1$.

In the first cycle, the allocated amount is M and the freed amount is $M/2$. The freed amount in the ith cycle for $2 \leq i \leq \ell - 2$ is

$$\sum_{j=2}^{i} \frac{M}{2^{i+1}n} 2^{j-1}n + \frac{M}{2^i n} n = \frac{M}{2^i} + \frac{M}{2^{i-1}} + \frac{M}{2^{i-2}} + \cdots + \frac{M}{4} + \frac{M}{2^i} = \frac{M}{2}.$$

And the allocated amount in the ith cycle for $2 \leq i \leq \ell - 2$ is $\frac{M}{2^i n} \cdot 2^{i-1}n = M/2$. Because the size of the next allocation request is always greater than all freed blocks, the memory usage is extended when every new block is placed. Thus, the amount of the memory usage is the total allocated amount in all cycles, that is,

$$M + \frac{(\ell - 1)M}{2} = \frac{(\ell + 1)M}{2} = \frac{M(\log_2 M - \log_2 n + 1)}{2}.$$

5.5 Experimental Results

Table 2 shows the summary of empirical results from a software simulation of the memory model using a first-fit memory allocator and a driver program that tries to maximize the memory usage by freeing up memory from previous cycles whenever possible as described above. In the simulation runs, we increase the block size of allocation request in each new cycle, by $n, 2n, 3n, 4n, \ldots$, etc., or we double the freed block size of the previous cycle. In a given run, the increment is the same in each cycle. All blocks in a given cycle are of the same size as this was found to maximize memory usage. The resulting memory usage is expressed in terms of a ratio to $M \log_2 M$. The program ends when the memory remaining to be allocated cannot be made greater than the largest contiguous free space. In the simulation, we went up to 8 MB for the largest value for

TABLE 2
The Ratio of Memory Usage to $M \log_2 M$

M	increment n	increment $2n$	double
2^{17}	0.485	0.454	0.412
2^{18}	0.478	0.471	0.417
2^{19}	0.461	0.477	0.421
2^{20}	0.474	0.446	0.425
2^{21}	0.457	0.446	0.429
2^{22}	0.461	0.450	0.432
2^{23}	0.451	0.455	0.435

TABLE 3
General Information about the Test Programs

Benchmark	Description
GCBench	This is adapted from a benchmark written by John Ellis and Pete Kovac of Post Communications. It attempts to model those properties of allocation requests that are important to current GC techniques.
OpenSSL	OpenSSL is an open-source toolkit that implements SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols. This test measures the RSA 4096-bit performance of OpenSSL.
SciMark	This test runs the ANSI C version of SciMark 2.0, which is a benchmark for scientific and numerical computing developed by programmers at the National Institute of Standards and Technology. This test is made up of Fast Fourier Transform, Jacobi Successive Over-relaxation, Monte Carlo, Sparse Matrix Multiply, and dense LU matrix factorization benchmarks.
SQLite	This is a simple benchmark of SQLite. At present this test profile just measures the time to perform a pre-defined number of insertions on an indexed database.
Xalan	XML Parser
Encode-mp3	LAME is an MP3 encoder licensed under the LGPL. This test measures the time required to encode a WAV file to MP3 format.
Compress-7zip	This is a test of 7-Zip using p7zip with its integrated benchmark feature.
GnuPG	This test times how long it takes to encrypt a file using GnuPG of a file.

M (the largest allocation request) since it is large enough, in practice, but the result is the same for greater values of M. The lowest value of M we show is 2^{17} bytes, but there are similar patterns even for lower values of M. Hence, they are not included in Table 2. Despite the oscillation behavior of the data seen in Table 2, we observe that the ratios of memory usage to $M \log_2 M$ are always below 0.5, that is, the memory usage is always close to $\frac{1}{2} M \log_2 M$. The upper bound derived in the previous section could not be surpassed by any of the allocation/deallocation patterns used to drive up memory usage.

5.6 Simulations with Benchmark Programs

In this paper, we focus on finding the upper bond of memory usage. As we have discussed above, the memory request and release pattern directly affects fragmentation and consequently memory usage. In the research community, synthetic memory request distributions have long been discarded and replaced by request patterns derived from actual programs. Thus, we next use a variety of benchmark programs with different request patterns to validate the theoretical upper bound.

In the simulation, we collected eight publicly available benchmark programs with a wide range of application. The functionality of these programs is summarized in Table 3. We first generate `malloc` and `free` traces for each program. Then, our simulator takes these traces as input and simulate three allocators. Other than the first-fit allocator mentioned earlier in this paper, we also simulate the binary-buddy and best-fit allocators. The simulation results are expressed in terms of allocator's memory usage (the highest amount of memory used during the entire run) and M (the actual program memory requirement) in Table 4.

From Table 4 (See allocation count column), it can be seen that some of the benchmark programs invoke the allocator much more frequently than others, such as GCBench. The reason of having the same value in first-fit and best-fit allocators for SciMark, GCBench, and Encode-mp3 is mainly due to its fixed-size allocation patterns. The buddy system tends to use more memory space than the other two schemes due to higher internal fragmentation. The memory usage of first-fit allocator is very close to that of the best-fit allocator [10], [11]. The proposed theoretical upper bound is about 6.77 to 10.43 times higher than the storages for the first-fit allocator. This confirms our earlier hypothesis that our upper bound is based on a worst case allocation/free pattern that is not commonly observed in the real-world applications.

TABLE 4
The Memory Usage of Best-Fit, First-Fit and Binary-Buddy Allocators

Benchmark	Allocation count	M	Best-fit	First-fit	Binary-buddy system	$\frac{M \log_2 M}{2}$
GCBench	15,333,864	12,582,888	16,582,888	16,582,888	20,971,488	148,383,454
OpenSSL	343,739	127,536	145,840	162,192	228,264	1,081,540
SciMark	3045	16,777,352	24,777,352	24,777,352	25,165,984	201,328,322
SQlite	115,443	206,228	280,863	284,095	333,584	1,820,362
Xalan	77,650	6,333,612	6,565,771	7,163,341	9,316,616	71,552,706
Encode-mp3	31	398,470	398,497	398,497	720,896	3,706,590
Compress-7zip	573	443,632,070	649,956,821	649,957,545	1,045,037,056	6,371,618,674
GnuPG	67	221,590	221,686	221,686	229,376	1,967,446

5.7 Discussions on Popular Allocation Policies

Unbounded or unpredictably high memory usage is a potential problem in multithreaded allocation schemes such as regions as well as in general-purpose allocators such as simple segregated fit. We have found an upper bound of memory usage for an allocator, the addressed-order first-fit allocator with immediate coalescing that is known to be among the best in keeping memory usage low.

Using a strategy similar to the one used in this section, memory allocation and release patterns causing worst case fragmentation and memory usage can be found for other types of allocators. For example, the worst case memory usage for the binary-buddy system would be similar to the one presented in this section for the case where the size increment in each cycles is twice the previous size. Thus, if cycle i uses a block size of x, then cycle $i+1$ uses a block size of $2x$ and the following cycle a block size of $4x$, etc. This is so because the binary-buddy allocator always allocates blocks in sizes that are powers of 2; moreover, coalesced and split blocks are also always powers of 2 in size. Thus, there are only $\log_2(M/n)$ block sizes possible if minimum block size is n, and consequently, there are only $\log_2(M/n)$ cycles possible. In each cycle, we can have a maximum of $M/2$ bytes of memory available for allocation including memory freed from all previous cycles. This can be seen by following a line of reasoning similar to the one presented for the first-fit allocator earlier. Thus, memory usage can be as high as $M + 0.5M\log_2(M/n)$. However, there is also a very large internal fragmentation in the binary-buddy system. A request for a block of size $p+1$, where p is power of 2 will be rounded up to $2p$, the next power of 2. The internal fragmentation in the worst case can be as high as 50 percent, so an actual program need for $M/2$ bytes can result in a program requirement for M bytes. This can double the actual upper bound of memory usage of binary-buddy allocators compared to first-fit allocators.

The worst case fragmentation pattern used for deriving the upper bound for first-fit allocators in this paper can be applied to best-fit allocators as well. In every cycle, none of the previously freed blocks are reused and all allocated blocks in a cycle are served from more memory obtained from the OS. Best-fit allocation policy cannot reduce the upper bound of memory usage forced by the worst-case allocation and release pattern described for first-fit allocators in this section. The above can be seen to be true by considering that the request size in each cycle is larger than the size of the largest free block in all the previous cycles even after all contiguous free blocks have been merged. Hence, the upper bound of memory usage for best-fit allocators must be at least as high as that obtained for first-fit allocators.

6 Conclusions

Memory usage has always been the important performance factor of a computer system. Memory space is managed dynamically in the heap region via allocation and free functions. The memory usage of a program is equal to the sum of M and the extra memory needed by the allocator due to fragmentation caused by unpredictable allocation and free patterns. This paper attempts to determine the upper bounds of two allocation schemes. First, we derive the upper bound of memory usage applicable to all allocators. Then, we find the upper bound of memory usage for the efficient first-fit allocator.

Probably, most industrial and commercial application programs will not reach the upper bounds derived in this paper although this assumption cannot be guaranteed. Application programs do not have maximization of fragmentation as their primary goal. But minimization of fragmentation is not their goal either. The actual fragmentation of a given program using a given allocator cannot be predicted. However, it is very useful to know that no program will ever run out of memory if the amount of memory determined by the upper bound of an allocator for the specified program memory requirement is made available to it. More importantly, we have shown that there exists an allocator whose worst case memory usage or upper bound is lower than any other published upper bound for any other allocator that we are aware of.

Acknowledgments

This material is partially based upon work supported by the US National Science Foundation under Grant No. 0296131 (ITR) 0219870 (ITR) and 0098235. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the US National Science Foundation.

References

[1] C. Bays, "A Comparison of Next-Fit, First-Fit, and Best-Fit," *Comm. ACM,* vol. 20, no. 3, pp. 191-192, 1977.

[2] E.D. Berger, B.G. Zorn, and K.S. McKinley, "Composing High-Performance Memory Allocators," *Proc. 2001 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI),* pp. 114-124, 2001.

[3] E.D. Berger, B.G. Zorn, and K.S. McKinley, "Reconsidering Custom Memory Allocation," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02),* pp. 1-12, 2002.

[4] H.-J. Boehm, "The Space Cost of Lazy Reference Counting," *Proc. 31st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 210-219, 2004.

[5] E.G. Coffman, "An Introduction to Combinatorial Models of Dynamic Storage Allocation," *SIAM Rev.,* vol. 25, no. 3, pp. 311-325, 1983.

[6] J.M. Chang and E.F. Gehringer, "A High-Performance Memory Allocator for Object-Oriented Systems," *IEEE Trans. Computers,* vol. 45, no. 3, pp. 357-366, Mar. 1996.

[7] S.M. Donahue, M.P. Hampton, M. Deters, J.M. Nye, R.K. Cytron, and K.M. Kavi, "Storage Allocation for Real-Time, Embedded Systems," *Lecture Notes in Computer Science,* pp. 131-147, Springer, 2001.

[8] M.R. Garey, R.L. Graham, and J.D. Ullman, "Worst Case Analysis of Memory Allocation Algorithms," *Proc. Fourth Ann. ACM Symp. Theory of Computing,* 1972.

[9] W. Gloger, Dynamic Memory Allocator Implementations in Linux System Libraries, Poliklinik für Zahnerhaltung, http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html, 2009.

[10] Y. Hasan and J.M. Chang, "A Tunable Hybrid Memory Allocator," *J. Systems and Software,* vol. 79, no. 8, pp. 1051-1063, 2006.

[11] M.S. Johnstone and P.R. Wilson, "The Memory Fragmentation Problem Solved," *Proc. Int'l Symp. Memory Management (ISMM),* pp. 26-36, 1998.

[12] R.E. Jones and R.D. Lins, *Garbage Collection.* John Wiley and Sons, 1996.

[13] K.C. Knowlton, "A Fast Storage Allocator," *Comm. ACM,* vol. 8, no. 10, pp. 623-625, 1977.
[14] D.E. Knuth, *The Art of Computer Programming,* vol. 1. Addison-Wesley, 1998.
[15] M. Luby, J. Naor, and A. Orda, "Tight Bounds for Dynamic Storage Allocation," *J. on Discrete Math.,* vol. 9, no. 1, pp. 155-166, 1996.
[16] D. Lea, A Memory Allocator, http://gee.cs.oswego.edu/dl/html/malloc.html, 2009.
[17] M. Masmano, J. Real, I. Ripoll, and A. Crespo, "Running Ada on Real-Time Linux," *Lecture Notes in Computer Science,* pp. 322-333, Springer, 2003.
[18] G. Ch. Pflug, "Dynamic Memory Allocation—A Markovian Analysis," *Computer J.,* vol. 27, no. 4, pp. 328-333, 1984.
[19] J.M. Robson, "An Estimate of the Store Size Necessary for Dynamic Storage Allocation," *J. ACM,* vol. 18, no. 3, pp. 416-423, 1971.
[20] J.M. Robson, "Bounds for Some Functions Concerning Dynamic Storage Allocation," *J. ACM,* vol. 21, no. 3, pp. 491-499, 1974.
[21] J.M. Robson, "Worst Case Fragmentation of First Fit and Best Fit Storage Allocation Strategies," *Computer J.,* vol. 20, no. 3, pp. 242-244, 1977.
[22] P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review," *Proc. Int'l Workshop Memory Management,* vol. 986, pp. 1-116, 1995.

Yusuf Hasan received the BS and MS degrees in computer science and mathematical computer science, respectively, from the University of Illinois. He received the PhD degree in computer science from Illinois Institute of Technology. He has worked in the software and telecom industry for nearly two decades. His professional experience includes positions at Sybase, MCI, and Motorola and teaching programming in a community college. His research and professional interests include memory management, performance, SW architecture, telecom, and mathematical computer science. He is a member of the ACM and has reviewed submitted papers for various journals.

Wei-Mei Chen received the PhD degree in computer science and information engineering from the National Taiwan University in 2000. She is currently an associate professor at the National Taiwan University of Science and Technology. Her research interests include algorithm design and analysis, automatic memory management, and mobile computing. She is a member of the IEEE.

J. Morris Chang is an associate professor at Iowa State University. He received the PhD degree in computer engineering from the North Carolina State University. His industrial experience includes positions at Texas Instruments, Microelectronic Center of North Carolina, and AT&T Bell Laboratories. He received the University Excellence in Teaching Award at Illinois Institute of Technology in 1999. His research interests include wireless networks, performance study of Java virtual machines (JVM), and computer architecture. Currently, he is a handling editor of the *Journal of Microprocessors and Microsystems* and the Middleware & Wireless Networks subject area editor of the *IEEE IT Professional*. He is a senior member of the IEEE.

Bashar M. Gharaibeh received the master's degree from Iowa State University in 2006 and the bachelor's degree from Jordan University of Science and Technology in 2003. He is currently a PhD student in the Department of Electrical and Computer Engineering at Iowa State University. He works now with the Computer Systems and Languages group in the Department of Computer Engineering at Iowa State University. His research interests include automatic memory management, performance, and Java virtual machine.