Fast Allocation and Deallocation with an Improved Buddy System*

Erik D. Demaine and J. Ian Munro

Department of Computer Science, University of Waterloo,
Waterloo, Ontario N2L 3G1, Canada,
{eddemaine, imunro}@uwaterloo.ca

Abstract. We propose several modifications to the binary buddy system for managing dynamic allocation of memory blocks whose sizes are powers of two. The standard buddy system allocates and deallocates blocks in $\Theta(\lg n)$ time in the worst case (and on an amortized basis), where n is the size of the memory. We present two schemes that improve the running time to $O(1)$ time, where the time bound for deallocation is amortized. The first scheme uses one word of extra storage compared to the standard buddy system, but may fragment memory more than necessary. The second scheme has essentially the same fragmentation as the standard buddy system, and uses $O(2^{(1+\sqrt{\lg n}) \lg \lg n})$ bits of auxiliary storage, which is $\omega(\lg^k n)$ but $o(n^\varepsilon)$ for all $k \geq 1$ and $\varepsilon > 0$. Finally, we present simulation results estimating the effect of the excess fragmentation in the first scheme.

1 Introduction

The *binary buddy system* [13] is a well-known system for maintaining a dynamic collection of memory blocks. Its main feature is the use of suitably aligned blocks whose sizes are powers of two. This makes it easy to check whether a newly deallocated block can be merged with an adjacent (unused) block, using bit operations on the block addresses. See Section 1.1 for a more detailed description of the method.

While the buddy system is generally recognized as fast, we argue that it is much slower than it has to be. Specifically, the time to allocate or deallocate a block of size 2^k is $\Theta(1 - k + \lg n)$ in the worst case, where n is the size of the memory in bytes. Not only is this a worst-case lower bound, but this much time can also be necessary on an amortized basis. Once we encounter a block whose allocation requires $\Theta(1 - k + \lg n)$ time, we can repeatedly deallocate and reallocate that block, for a total cost of $\Theta(m(1 - k + \lg n))$ over m operations. Such allocations and deallocations are also not rare; for example, if the memory is completely free and we allocate a constant-size block, then the buddy system uses $\Theta(\lg n)$ time. Throughout this paper we assume standard operations on a word of size $1 + \lg n$ or so bits can be performed in constant time.

* This work was supported by the Natural Science and Engineering Research Council of Canada (NSERC).

1.1 Buddy System

The (binary) buddy system was originally described by Knowlton [11, 12]. It is much faster than other heuristics for dynamic memory allocation, such as first-fit and best-fit. Its only disadvantage being that blocks must be powers of two in size, the buddy system is used in many modern operating systems, in particular most versions of UNIX, for small block sizes. For example, BSD [18] uses the buddy system for blocks smaller than a page, i.e., 4 kilobytes.

The classic description of the buddy system is Knuth's [13]. Because our work is based on the standard buddy system, we review the basic ideas now.

At any point in time, the memory consists of a collection of *blocks* of consecutive memory, each of which is a power of two in size. Each block is marked either *occupied* or *free*, depending on whether it is allocated to the user. For each block we also know its size (or the logarithm of its size). The system provides two operations for supporting dynamic memory allocation:

1. Allocate (2^k): Finds a free block of size 2^k, marks it as occupied, and returns a pointer to it.
2. Deallocate (B): Marks the previously allocated block B as free and may merge it with others to form a larger free block.

The buddy system maintains a list of the free blocks of each size (called a *free list*), so that it is easy to find a block of the desired size, if one is available. If no block of the requested size is available, Allocate searches for the first nonempty list for blocks of at least the size requested. In either case, a block is removed from the free list. This process of *finding a large enough free block* will indeed be the most difficult operation for us to perform quickly.

If the found block is larger than the requested size, say 2^k instead of the desired 2^i, then the block is split in half, making two blocks of size 2^{k-1}. If this is still too large ($k - 1 > i$), then one of the blocks of size 2^{k-1} is split in half. This process is repeated until we have blocks of size 2^{k-1}, 2^{k-2}, $\ldots$, 2^{i+1}, 2^i, and 2^i. Then one of the blocks of size 2^i is marked as occupied and returned to the user. We will modify this *splitting process* as the first step in speeding up the buddy system.

Now when a block is deallocated, the buddy system checks whether the block can be merged with any others, or more precisely whether we can undo any splits that were performed to make this block. This is where the buddy system gets its name. Each block B_1 (except the initial blocks) was created by splitting another block into two halves, call them B_1 (with the same address but half the size) and B_2. The other block, B_2, created from this split is called the *buddy* of B_1, and vice versa. The *merging process* checks whether the buddy of a deallocated block is also free, in which case the two blocks are merged; then it checks whether the buddy of the resulting block is also free, in which case they are merged; and so on.

One of the main features of the buddy system is that buddies are very easy to compute on a binary computer. First note that because of the way we split and merge blocks, blocks stay *aligned*. More precisely, the address of a block of size 2^k (which we always consider to be written in binary) ends with k zeros. As a result, to find the address of the buddy of a block of size 2^k we simply flip the $(k+1)$st bit from the right.

Thus it is crucial for performance purposes to know, given a block address, the size of the block and whether it is occupied. This is usually done by storing a *block header* in the first few bits of the block. More precisely, we use headers in which the first bit is the *occupied bit*, and the remaining bits specify the size of the block. Thus, for example, to determine whether the buddy of a block is free, we compute the buddy's address, look at the first bit at this address, and also check that the two sizes match.

Because block sizes are always powers of two, we can just encode their logarithms in the block headers. This uses only $\lg \lg n$ bits, where n is the number of (smallest) blocks that can be allocated. As a result, the smallest practical header of one byte long is sufficient to address up to $2^{128} \approx 3.4 \cdot 10^{38}$ blocks. Indeed, if we want to use another bit of the header to store some other information, the remaining six bits suffice to encode up to $2^{64} \approx 1.8 \cdot 10^{19}$ blocks, which should be large enough for any practical purposes.

1.2 Related Work

Several other buddy systems have been proposed, which we briefly survey now. Of general interest are the Fibonacci and weighted buddy systems, but none of the proposals theoretically improve the running time of the Allocate and Deallocate operations.

In Exercise 2.5.31 of his book, Knuth [13] proposed the use of Fibonacci numbers as block sizes instead of powers of two, resulting in the *Fibonacci buddy system*. This idea was detailed by Hirschberg [9], and was optimized by Hinds [8] and Cranston and Thomas [7] to locate buddies in time similar to the binary buddy system. Both the binary and Fibonacci buddy systems are special cases of a generalization proposed by Burton [5].

Shen and Peterson [23] proposed the *weighted buddy system* which allows blocks of sizes 2^k and $3 \cdot 2^k$ for all k. All of the above schemes are special cases of the generalization proposed by Peterson and Norman [20] and a further generalization proposed by Russell [22]. Page and Hagins [19] proposed an improvement to the weighted buddy system, called the *dual buddy system*, which reduces the amount of fragmentation to nearly that of the binary buddy system. Another slight modification to the weighted buddy system was described by Bromley [3, 4]. Koch [14] proposed another variant of the buddy system that is designed for disk-file layout with high storage utilization.

The fragmentation of these various buddy systems has been studied both experimentally and analytically by several papers [4, 6, 17, 20–22].

1.3 Outline

Sections 2 and 3 describe our primary modifications to the Allocate and Deallocate operations of the binary buddy system. Finding an appropriate free block for Allocate is the hardest part, so our initial description of Allocate assumes that such a block has been found, and only worries about the splitting part. In

Sections 4 and 5 we present two methods for finding a block to use for allocation. Finally, Section 6 gives simulation results comparing the fragmentation in the two methods.

2 Lazy Splitting

Recall that if we allocate a small block out of a large block, say 2^i out of 2^k units, then the standard buddy system splits the large block $k-i$ times, resulting in subblocks of sizes 2^{k-1}, 2^{k-2}, …, 2^{i+1}, 2^i, and 2^i, and then uses one of the blocks of size 2^i. The problem is that if we immediately deallocate the block of size 2^i, then all $k-i+1$ blocks must be remerged into the large block of size 2^k. This is truly necessary in order to discover that a block of size 2^k is available; the next allocation request may be for such a block.

To solve this problem, we do not explicitly perform the first $k-i-1$ splits, and instead jump directly to the last split at the bottom level. That is, the large block of size 2^k is split into two blocks, one of size 2^i and one of size $2^k - 2^i$. Note that the latter block has size not equal to a power of two. We call it a *superblock*, and it contains allocatable blocks of sizes 2^i, 2^{i+1}, …, 2^{k-2}, and 2^{k-1} (which sum to the total size $2^k - 2^i$). For simplicity of the algorithms, we always remove the small block of size 2^i from the left side of the large block of size 2^k, and hence the allocatable blocks contained in a superblock are always in order of increasing size.

In general, we maintain the size of each allocated block as a power of two, while the size of a free block is either a power of two or a difference of two powers of two. Indeed, we can view a power of two as the difference of two consecutive powers of two. Thus, every free block can be viewed as a superblock, containing one or more allocatable blocks each of a power of two in size.

To see how the free superblocks behave, let us consider what happens when we allocate a subblock of size 2^j out of a free superblock of size $2^k - 2^i$. The free block is the union of allocatable blocks of sizes $2^i, 2^{i+1}, \ldots, 2^{k-1}$, and hence $i \le j \le k-1$. Removing the allocatable block of size 2^j leaves two consecutive sequences: $2^i, 2^{i+1}, \ldots, 2^{j-1}$ and $2^{j+1}, 2^{j+2}, \ldots, 2^{k-1}$. Thus, we split the superblock into the desired block of size 2^j and two new superblocks of sizes $2^j - 2^i$ and $2^k - 2^{j+1}$.

2.1 Block Headers

As described in Section 1.1, to support fast access to information about a block given just its address (for example when the user requests that a block be deallocated) it is common to have a header on every block that contains basic information about the block. Recall that block headers in the standard buddy system store a single bit specifying whether the block is occupied (which is used to test whether a buddy is occupied), together with a number specifying the logarithm of the size of the block (which is used to find the buddy).

With our modifications, superblocks are no longer powers of two in size. The obvious encoding that uses $\Theta(\lg n)$ bits causes the header to be quite large—a single byte is insufficient even for the levels at which the buddy system is applied today (for example, smaller than 4,096 bytes in BSD 4.4 UNIX [18]). Fortunately, there are two observations which allow us to leave the header at its current size. The first is that allocated blocks are a power of two in size, and hence the standard header suffices. The second is that free blocks are a difference of two powers of two in size, and hence two bytes suffice; the second byte can be stored in the data area of the block (which is unused because the block is free).

2.2 Split Algorithm

To allocate a block of size 2^j, we first require a free superblock containing a properly aligned block of at least that size, that is, a superblock of size $2^k - 2^i$ where $k > i$ and $k > j$. Finding this superblock will be addressed in Sections 4 and 5. The second half of the Allocate algorithm is to *split* that superblock down to the appropriate size, and works as follows. Assume the superblock B at address a has an appropriate size. First examine the header of the block at address a. By assumption, the occupied bit must be clear, i.e., B must be free. The next two numbers of the header are k and i and specify that B has size $2^k - 2^i$. In other words, B is a superblock containing allocatable blocks of size $2^i, 2^{i+1}, \ldots, 2^{k-1}$, in that order.

There are two cases to consider. The first case is that one of the blocks in B has size 2^j, that is, $i \leq j \leq k-1$. The address of this block is $a + \sum_{m=i}^{j-1} 2^m = a + 2^j - 2^i$. First we initialize the header of this block, by setting the occupied bit and initializing the logarithm of the size to j. The address of this block is also returned as the result of the Allocate operation. Next B has to be split into the allocated block and, potentially, a superblock on either side of it. If $j < k-1$ (in other words, we did not allocate the last block in the superblock), then we need a superblock on the right side. Thus, we must initialize the block header at address $a + 2^{j+1} - 2^i$ with a clear occupied bit followed by k and $j+1$. (That is, the block has size $2^k - 2^{j+1}$.) Similarly, if $j > i$ (in other words, we did not allocate the first block in the superblock), then we need a superblock on the left side. Thus, we modify the first number in the header at address a from k to j, thereby specifying that the block now has size $2^j - 2^i$.

The second case is the undesirable case in which $i > j$, in other words we must subdivide one of the blocks in superblock B to make the allocation. The smallest block in B, of size 2^i, is adequate. It is broken off and the remainder of B is initialized as a free superblock of size $2^k - 2^{i+1}$. The block of size 2^i is broken into the allocated block of size 2^j, and a superblock of size $2^i - 2^j$ which is returned to the structure for future use.

An immediate consequence of the above modification to the split operation is the following:

Lemma 1. *The cost of a split is constant.*

3 Unaggressive Merging

This section describes how merging works in combination with blocks whose sizes are not powers of two. Our goal is for merges to undo already performed splits, because the conditions that caused the split no longer hold. However, we are not too aggressive about merging: we do not merge adjacent superblocks into larger superblocks. Instead, we wait until a collection of superblocks can be

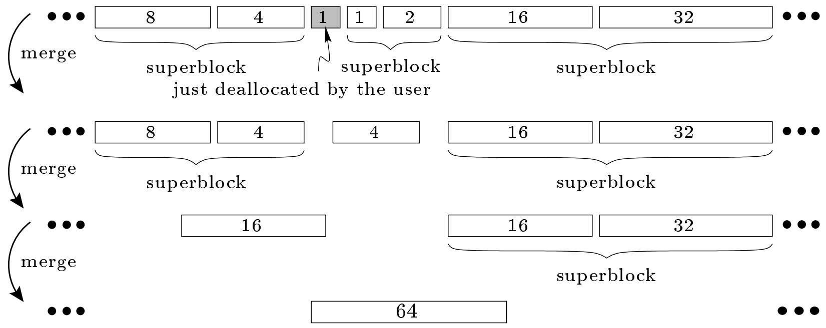

Fig. 1. An example in which blocks have been aggressively merged into superblocks, but a single deallocation causes $\Theta(\lg n)$ merges.

merged into a usual block of a power of two in size. This is because we will only use superblocks to speed up splits. An amortized time bound for merges follows immediately, and unfortunately this kind of "aggressive merging" is not enough to obtain a worst-case time bound; see Fig. 1.

Hence, our problem reduces to detecting mergeable buddies in the standard sense, except that buddies may not match in size: the left or right buddy may be a much larger superblock. This can be done as follows. Suppose we have just deallocated a block B and want to merge it with any available buddies. First we clear the occupied bit in the header of B. Next we read the logarithm of the size of the block, call it i, and check whether B can be merged with any adjacent blocks, or in other words whether it can be merged with its buddy, as follows.

Because of the alignment of allocated blocks, the last i bits of the address of B must be zeros. If the $(i+1)$st bit from the right is a zero, then our block is a left buddy of some other block; otherwise, it is a right buddy. In either case, we can obtain the address of B's buddy by flipping the $(i+1)$st bit of B's address, that is, by taking a bitwise exclusive-or applied to 1 shifted left i times.

If the header of B's buddy has the occupied bit clear, we read its size $2^k - 2^j$. If B's size equals the lacking size 2^j (i.e, $i = j$), we merge the buddies and update the header to specify a size of 2^j. In this case, we repeat the process to see whether the buddy of the merged block is also free.

Lemma 2. *The cost of a sequence of merges is constant amortized.*

Proof. The total number of individual merges is at most twice the number of already performed (and not remerged) splits, and hence each sequence of merge operations takes $O(1)$ amortized time. □

4 Finding a Large Enough Free Block: Fragmentation

This section presents our first approach to the remaining part of the Allocate algorithm, which is to find a block to return (if it is of the correct size) or split (if it is too large). More precisely, we need to find a block that is at least as large as the desired size. The standard buddy system maintains a doubly linked list of free blocks of each size for this purpose. Indeed, the free list is usually stored as a doubly linked list whose nodes are the free blocks themselves (since they have free space to use). The list must be doubly linked to support removal of a block in the middle of the list as the result of a merge.

We do the same, where a superblock of size $2^k - 2^i$ is placed on the free list for blocks of size 2^{k-1}, corresponding to the largest allocatable block contained within it. This will give us the smallest superblock that is large enough to handle the request. However, it may result in splitting a block when unnecessary; we shall readdress this issue in the next section.

The difficulty in finding the smallest large-enough superblock is that when (for example) there is a single, large block and we request the smallest possible block, it takes $\Theta(\lg n)$ time to do a linear scan for the appropriate free list. To find the appropriate list in $O(1)$ worst-case time, we maintain a bitvector of length $\lfloor \lg n \rfloor$, whose $(i+1)$st bit from the right is set precisely if the list for blocks of size 2^i is nonempty. Then the next nonempty list after or at a particular size 2^k can be found by first shifting the bitvector right by k, and then computing the least-significant set bit.

The latter operation is included as an instruction in many modern machines. Newer Pentium chips do it as quickly as an integer addition. It can also be computed in constant time using boolean and basic arithmetic operations [2]. Another very simple method is to store a lookup table of the solutions for all bitstrings of length $\Theta(n^\varepsilon)$ for some constant $\varepsilon > 0$, using $\Theta(n^\varepsilon)$ words of space; cut the bitvector into $\lceil 1/\varepsilon \rceil$ chunks; and check for a set bit in each chunk from right to left. This $\Theta(n^\varepsilon)$ extra space is justified because many data structures require this operation, and it is perfectly reasonable for the operating system to provide a common static table for all processes to access.

Theorem 1. *The described modifications to the buddy system cause* Allocate *[*Deallocate*] to run in constant worst-case [amortized] time.*

5 Finding a Large Enough Free Block: Extra Storage

One unfortunate property of the method described above is that even if a block of the desired size is available as part of a superblock, it may not be used because preference is given to a larger block. The reason is that our method prefers a superblock whose largest allocatable block is minimal. Unfortunately, such a superblock may not contain an allocatable block of exactly (or even close to) the desired size, whereas a superblock containing a larger largest block might. Furthermore, even if there is no block of exactly the desired size, our method will not find the smallest one to split. As a result, unnecessary splits may be performed, slightly increasing memory fragmentation. We have not performed a statistical analysis of the effect in fragmentation as a result of this property, but simulation results are presented in Section 6.

In this section, we present a further modification that solves this problem and leaves the fragmentation in essentially the same state as does the standard buddy system. Specifically, we abstract the important properties of the standard buddy system's procedure for finding a large enough free block into the following *minimum-splitting requirement*: the free block chosen must be the smallest block that is at least the desired size. In particular, if there is a block of exactly the desired size, it will be chosen. This requirement is achieved by the standard buddy system, and the amount of block splitting is locally minimized by any method achieving it.

Of course, there may be ties in "the smallest block that is at least the desired size," so different "minimum-splitting" methods may result in different pairs of blocks becoming available for remerging, and indeed do a different amount of splitting on the same input sequence. However, we observe that even different implementations of the "standard buddy system" will make different choices. Furthermore, if we view all blocks of the same size as being equally likely to be deallocated at any given time, then all minimum-splitting systems will have identical distributions of fragmentation.

In the context of the method described so far, the minimum-splitting requirement specifies that we must find a superblock containing a block of the appropriate size if one exists, and if none exists, it must find the superblock containing the smallest block that is large enough. This section describes how to solve the following more difficult problem in constant time: find the superblock whose smallest contained block is smallest, over all superblocks whose largest contained block is large enough to serve the query.

Recall that superblocks have size $2^k - 2^j$ for $1 \le k \le \lceil \lg n \rceil$ and $0 \le j \le k-1$. For each of the possible (k, j) pairs, we maintain a doubly linked list of all free superblocks of size $2^k - 2^j$ (where "superblock" includes the special case of "block"). By storing the linked list in the free superblocks themselves, the auxiliary storage required is only $\Theta(\lg^2 n)$ pointers or $\Theta(\lg^3 n)$ bits.

For each value of k, we also maintain a bitvector V_k of length $\lfloor \lg n \rfloor$, whose jth bit indicates whether there is at least one superblock of size $2^k - 2^j$. This vector can clearly be maintained in constant time subject to superblock allocations and deallocations. By finding the least-significant set bit, we can also maintain the minimum set j in V_k for each k, in constant time per update.

The remaining problem is to find some superblock of size $2^k - 2^j$ for which, subject to the constraint $k > i$, j is minimized. This way, if $j \le i$, this superblock contains a block of exactly the desired size; and otherwise, it contains the smallest block adequate for our needs. The problem is now abstracted into having a vector $V_{\min}$, whose kth element $(1 \le k \le \lfloor \lg n \rfloor)$ is the minimum j for which there is a free superblock of size $2^k - 2^j$; in other words, $V_{\min}[k] = \min\{j \mid V_k[j] > 0\}$. Given a value i, we are to find the smallest value of j in the last $\lfloor \lg n \rfloor - i$ positions of $V_{\min}$; in other words, we must find $\min\{V_{\min}[k] \mid k > i\}$.

The basic idea is that, because each element of $V_{\min}$ value takes only $\lg \lg n$ bits to represent, "many" can be packed into a single $(1 + \lg n)$-bit word. Indeed, we will maintain a dynamic multiway search tree of height 2. The $\lfloor \lg n \rfloor$ elements of $V_{\min}$ are split into roughly $\sqrt{\lg n}$ groups of roughly $\sqrt{\lg n}$ elements each. The pth child of the root stores the elements in the pth group. The root contains roughly $\sqrt{\lg n}$ elements, the pth of which is the minimum element in the pth group. As a consequence, each node occupies $\sqrt{\lg n} \lg \lg n$ bits.

A query for a given i is answered in two parts. First we find the minimum of the first $\lfloor i/\sqrt{\lg n}\rfloor$ elements of the root node, by setting the remaining elements to infinities, and using a table of answers for all possible root nodes. The second part in determining the answer comes from inspecting the $\lceil i/\sqrt{\lg n}\rceil$th branch of the tree, which in general will contain some superblocks (or j values) that are valid for our query and some that are not. We must, then, make a similar query there, and take the smallest j value of the two. The extra space required is dominated by the table that gives the value and position of the smallest element, for all possible $\sqrt{\lg n}$ tuples of $\lg\lg n$ bits each. There are $2^{\sqrt{\lg n}\lg\lg n}$ entries in this table, and each entry requires $2\lg n$ bits, for a total of $2^{(1+\sqrt{\lg n})\lg\lg n+1}$ bits. As a consequence, the total space required beyond the storage we are managing is $o(n^{\varepsilon})$ but $\omega(\lg^k n)$. Updating the structure to keep track of the minimum j for each k, in constant time after each memory allocation or deallocation, is straightforward.

Theorem 2. *The described modifications to the buddy system satisfy the minimum-splitting requirement, and cause* Allocate *[*Deallocate*] to run in constant worst-case [amortized] time.*

6 Simulation

To help understand the effect of the excess block splitting in the first method (Section 4), we simulated it together with the standard buddy system for various memory sizes and "load factors." Our simulation attempts to capture the spirit of the classic study by Knuth [13] that compares various dynamic allocation schemes. In each time unit of his and our simulations, a new block is allocated with randomly chosen size and lifetime according to various distributions, and old blocks are checked for lifetime expiry which causes deallocations. If there is ever insufficient memory to perform an allocation, the simulation halts.

While Knuth periodically examines memory snapshots by hand to gain qualitative insight, we compute various statistics to help quantify the difference between the two buddy systems. To reduce the effect of the choice of random numbers, we run the two buddy schemes on exactly the same input sequence, repeatedly for various sequences. We also simulate an "optimal" memory allocation scheme, which continually compacts all allocated blocks into the left fraction of memory, in order to measure the "difficulty" of the input sequence.

Few experimental results seem to be available on typical block-size and lifetime distributions, so any choice is unfortunately guesswork. Knuth's block sizes are either uniformly distributed, exponentially distributed, or distributed according to a hand-selection of probabilities. We used the second distribution (choosing size 2^i with probability $1/(1+\lfloor\lg n\rfloor)$), and what we guessed to be a generalization of the third distribution (choosing size 2^i with probability 2^{-i-1}, roughly). We dropped the first distribution because we believe it weights large blocks too heavily—blocks are typically quite small. Note that because the two main memory-allocation methods we simulate are buddy systems, we assume that all allocations ask for sizes that are powers of two. Also, to avoid rapid overflow in the second distribution, we only allow block sizes up to $n^{3/4}$, i.e., logarithms of block sizes up to $\frac{3}{4} \lg n$.

Knuth's lifetimes are uniformly distributed according to one of three ranges. We also use uniform distribution but choose our range based on a given parameter called the *load factor*. The load factor L represents the fraction of memory that tends to be used by the system. Given one of the distributions above on block size, we can compute the expected block size E, and therefore compute a lifetime Ln/E that will on average keep the amount of memory used equal to Ln (where n is the size of memory). To randomize the situation, we choose a lifetime uniformly between 1 and $2Ln/E - 1$, which has the same expected value Ln/E.

The next issue is what to measure. To address this it is useful to define a notion of the system reaching an "equilibrium." Because the simulation starts with an empty memory, it will start by mostly allocating blocks until it reaches the expected memory occupancy, Ln. Suppose it takes t time steps to reach that occupancy. After t more steps (a total of $2t$), we say that the system has reached an *equilibrium*; at that point, it is likely to stay in a similar configuration. (Of course, it is possible for the simulation to halt before reaching an equilibrium, in which case we discard that run.)

One obvious candidate for a quantity to measure is the amount of fragmentation (i.e., the number of free blocks) for each method, once every method has reached an equilibrium. However, this is not really of interest to the user: the user wants to know whether her/his block can be allocated, or whether the system will *fail* by being unable to service the allocation. This suggests a more useful metric, the *time to failure*, frequently used in the area of fault tolerance.

A related metric is to wait until all systems reach an equilibrium (ignoring the results if the system halts before that), and then measure the largest free allocatable block in each system. For the standard buddy system, this is the largest block of size a power of two; for our modified buddy system, it is the largest block in any superblock; and for the optimal system, it is simply the amount of free memory. This measures, at the more-or-less arbitrary time of all systems reaching equilibrium, the maximum-size block that could be allocated.

We feel that these two metrics capture some notion of what users of a memory-allocation system are interested in. By evaluating them for all three systems under the same inputs, we can measure the difference between the two buddy systems, relative to the optimal system. This kind of "relative error" was measured for 100 runs and then averaged, for each case. Memory size ranges between 2^4 (the smallest power-of-two size for which a difference between the two buddy systems is noticeable) and 2^{12} (the size used by BSD 4.4 UNIX [18]). The tested load factors are 50%, 75%, and 90%.

The results are shown in Fig. 2. The relative errors are for the most part quite small (typically under 5%). Indeed, our first method occasionally does somewhat better than the standard buddy system, because its different choices of blocks to split cause some fortunate mergings. Further evidence is that, for the exponential distribution, less than 10% of the runs showed any difference in

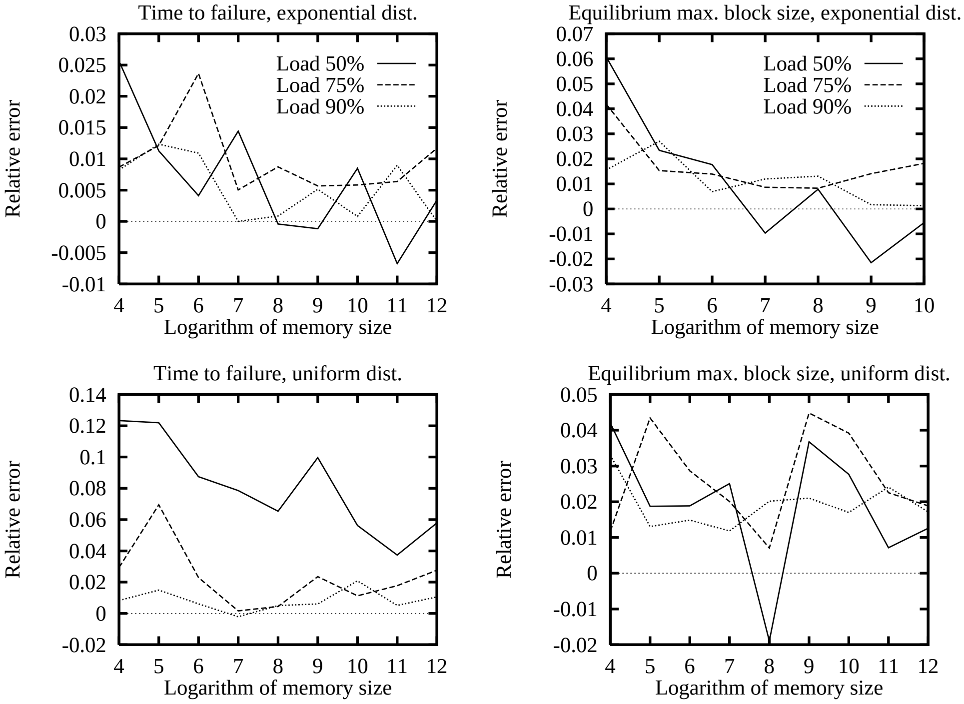

Fig. 2. Simulation results. The distributions refer to the distributions of the logarithms of the block sizes, and "load" refers to the load factor.

time-to-failure between the two systems. (However, the number of differences is greater for the uniform distribution.)

Thus, the difference in distributions of fragmentation between the two buddy systems seems reasonably small. The simplicity of our first method may make it attractive for implementation.

7 Conclusion

We have presented two enhancements to the buddy system that improve the running time of Allocate to constant worst-case time, and Deallocate to constant amortized time. The more complex method keeps the distribution of fragmentation essentially the same as the standard method, while the simpler approach leads to a different and slightly worse distribution. It would be of interest to specify this difference mathematically.

We note that it is crucial for Allocate to execute as quickly as possible (and in particular fast in the worst case), because the executing process cannot proceed until the block allocation is complete. In contrast, it is reasonable for the Deallocate time bound to be amortized, because the result of the operation is not important and the actual work can be delayed until the CPU is idle (or the memory becomes full). Indeed, this delay idea has been used to improve the cost of the standard buddy system's Deallocate [1, 10, 15, 16]. On the other hand, for the purposes of theoretical results, it would be of interest to obtain a constant worst-case time bound for both Allocate and Deallocate.

References

1. R. E. Barkley and T. Paul Lee. A lazy buddy system bounded by two coalescing delays per class. *Operating Systems Review*, pages 167–176, Dec. 1989.
2. Andrej Brodnik. Computation of the least significant set bit. In *Proceedings of the 2nd Electrotechnical and Computer Science Conference*, Portoroz, Slovenia, 1993.
3. Allan G. Bromley. An improved buddy method for dynamic storage allocation. In *Proceedings of the 7th Australian Computer Conference*, pages 708–715, 1976.
4. Allan G. Bromley. Memory fragmentation in buddy methods for dynamic storage allocation. *Acta Informatica*, 14:107–117, 1980.
5. Warren Burton. A buddy system variation for disk storage allocation. *Communications of the ACM*, 19(7):416–417, July 1976.
6. Shyamal K. Chowdhury and Pradip K. Srimani. Worst case performance of weighted buddy systems. *Acta Informatica*, 24(5):555–564, 1987.
7. Ben Cranston and Rick Thomas. A simplified recombination scheme for the Fibonacci buddy system. *Communications of the ACM*, 18(6):331–332, June 1975.
8. James A. Hinds. Algorithm for locating adjacent storage blocks in the buddy system. *Communications of the ACM*, 18(4):221–222, Apr. 1975.
9. Daniel S. Hirschberg. A class of dynamic memory allocation algorithms. *Communications of the ACM*, 16(10):615–618, Oct. 1973.
10. Arie Kaufman. Tailored-list and recombination-delaying buddy systems. *ACM Transactions on Programming Languages and Systems*, 6(1):118–125, Jan. 1984.
11. Kenneth C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–625, Oct. 1965.
12. Kenneth C. Knowlton. A programmer's description of L6. *Communications of the ACM*, 9(8):616–625, Aug. 1966.
13. Donald E. Knuth. Dynamic storage allocation. In *The Art of Computer Programming*, volume 1, section 2.5, pages 435–455. Addison-Wesley, 1968.
14. Philip D. L. Koch. Disk file allocation based on the buddy system. *ACM Transactions on Computer Systems*, 5(4):352–370, Nov. 1987.
15. T. Paul Lee and R. E. Barkley. Design and evaluation of a watermark-based lazy buddy system. *Performance Evaluation Review*, 17(1):230, May 1989.
16. T. Paul Lee and R. E. Barkley. A watermark-based lazy buddy system for kernel memory allocation. In *Proceedings of the 1989 Summer USENIX Conference*, pages 1–13, June 1989.
17. Errol L. Lloyd and Michael C. Loui. On the worst case performance of buddy systems. *Acta Informatica*, 22(4):451–473, 1985.
18. Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.
19. Ivor P. Page and Jeff Hagins. Improving the performance of buddy systems. *IEEE Transactions on Computers*, C-35(5):441–447, May 1986.
20. James L. Peterson and Theodore A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, June 1977.
21. Paul W. Purdom, Jr. and Stephen M. Stigler. Statistical properties of the buddy system. *Journal of the ACM*, 17(4):683–697, Oct. 1970.
22. David L. Russell. Internal fragmentation in a class of buddy systems. *SIAM Journal on Computing*, 6(4):607–621, Dec. 1977.
23. Kenneth K. Shen and James L. Peterson. A weighted buddy method for dynamic storage allocation. *Communications of the ACM*, 17(10):558–562, Oct. 1974. See also the corrigendum in 18(4):202, Apr. 1975.