A Lazy Buddy System Bounded by Two Coalescing Delays per Class

R. E. Barkley
AT&T Bell Laboratories
Summit, NJ 07901

and

T. Paul Lee
AT&T Bell Laboratories
Holmdel, NJ 07733

ABSTRACT

The watermark-based lazy buddy system for dynamic memory management uses lazy coalescing rules controlled by watermark parameters to achieve low operational costs. The correctness of the watermark-based lazy buddy system is shown by defining a space of legal states called the lazy space and proving that the watermark-based lazy coalescing rules always keep the memory state within that space. In this paper we describe a different lazy coalescing policy, called the DELAY-2 algorithm, that focuses directly on keeping the memory state within the lazy space. The resulting implementation is simpler, and experimental data shows it to be up to 12% faster than the watermark-based buddy system and about 33% faster than the standard buddy system. Inexpensive operations make the DELAY-2 algorithm attractive as a memory manager for an operating system.

The watermark-based lazy buddy policy offers fine control over the coalescing policy of the buddy system. However, applications such as the UNIX® System kernel memory manager do not need such fine control. For these applications, the DELAY-2 buddy system provides an efficient memory manager with low operational costs and low request blocking probability. In the DELAY-2 buddy system, the worst-case time for a free operation is bounded by two coalescing delays per class, and when all blocks are returned to the system, the system memory is coalesced back to its original state. This ensures that the memory space can be completely shared.

2. *INTRODUCTION*

The buddy system [1, 2] is a dynamic memory manager that responds well to shifting memory demands. When requested to allocate a block of a particular size, the buddy system can split a larger piece of memory into smaller *buddies* to match the request. For ease of discussion in this paper, we assume a binary buddy system; the lazy coalescing control can also be applied to Fibonacci and weighted buddy systems. When a block is freed, if the block's buddy is also free the buddy system coalesces the memory immediately; coalescing buddies makes available a larger piece of memory that can be used later.

In the buddy system, the costs to allocate and free a block are low compared to those of best- or first-fit policies [1]. However, in some memory management applications such as in an operating system, memory allocation and free operations must be as fast as possible. In the buddy system the costs of these operations are dominated by the costs of fragmenting or coalescing memory. In this paper, we describe a variation of the buddy system that reduces the costs of memory operations by reducing the number of times we have to fragment and coalesce memory.

Studies of STREAMS buffer request/release characteristics [3] in UNIX® System V show that workloads often exhibit steady-state behavior in memory demand; that is, they maintain slow time-varying demand for blocks of particular classes. (We refer to blocks of the same size as a block *class*.) We have previously designed a watermark-based lazy buddy system that capitalizes on this steady-state behavior [4]. The watermark-based lazy buddy system has low operational costs, ensures that all memory space will be available to any block class, and guarantees bounds on the cost to allocate or free a particular block. The UNIX System kernel, however, does not require control as fine as that provided by the watermark-based lazy buddy system with

its two watermark parameters. In this paper, we present another lazy buddy system, called the DELAY-2 buddy system, that has slightly coarser controls but offers even lower operational costs. The new system is similar to the watermark-based buddy system in that it is based on the same notion of a *lazy space* and it still bounds the cost of free operations to two coalescing delays per class.

The body of this paper is organized as follows: in Section 2 we describe the DELAY-2 buddy system, its characteristics, and an important transitive closure property. In Section 3, we present a simple slack variable-based implementation of the algorithm. In Section 4, we describe measurements made on a prototype STREAMS buffer manager based on the DELAY-2 buddy algorithm and compare them with measurements made on the standard buddy system and on the watermark-based buddy system. In Section 5, we discuss memory efficiency, and in Section 6 we summarize the observations and results.

3. THE DELAY-2 BUDDY SYSTEM

To set the framework for discussing the DELAY-2 buddy system, we first briefly describe the standard buddy system.

The buddy system begins with a monolithic piece of free memory. In response to a block request, the system splits the free memory into smaller pieces until it obtains a block that matches the request. In a binary buddy system a request for a block of size s is satisfied with the smallest block of size 2^n such that $s \leq 2^n$. The unused or free *buddy* of the allocated block is placed on the free list corresponding to its class. The system responds to additional requests for blocks either by taking a properly sized block from the appropriate free list or by taking a larger block and splitting it until a correctly sized block is obtained. The state of every block, whether allocated or free, is kept in a system-wide bit-map. When a block is being freed, the system looks in the bit-map to see if the buddy of the block is also free. If it is, the system coalesces the two buddies to create a larger free block; the larger block is then coalesced with its buddy if that buddy is free, and so on until either the buddy of the newly coalesced block is not free or all the memory has been coalesced. Since the system recursively coalesces whenever any block is freed, the system memory returns to its original monolithic state when all the blocks have been freed. We refer to each of the buddy coalescing operations *for a particular class* as a "coalescing delay." Buddies in one class that coalesce (with a corresponding coalescing delay) will form a new free block in the next larger class; the new block may then coalesce (with a coalescing delay for its class), and so on. Thus, a particular block free operation may recursively cause several coalescing operations, but *each class* will have at most one coalescing delay.

3.1 Lazy Coalescing Rules

In the standard buddy system, the cost of memory operations is dominated by bookkeeping costs and by the time spent splitting and coalescing memory. To reduce the operational costs, we can design buddy systems with more relaxed coalescing rules. Relaxed rules work well when the workload (i.e., the dynamic block request/release characteristics) is steady and the request rates for blocks of various sizes change slowly.

The watermark-based lazy buddy policy (θ_l,θ_s) [4] is an example of a buddy system with relaxed coalescing rules. The coalescing rules are governed by the policy parameters θ_l (the lazy ratio) and θ_s (the speedup ratio). The watermark-based lazy buddy policy differs from the buddy system in that it modulates the buddy coalescing process depending on the mode of each block class.

In the watermark-based lazy buddy system, free blocks can be either "globally-free" or "locally-free." A "globally-free" block is a free block that is marked as free in the bit-map but is placed on the free list for its class because its buddy is marked busy in the bit-map. All free blocks in the standard buddy system could be considered "globally-free." A "locally-free" block, on the other hand, is not marked as free in the bit-map; it is placed directly on the free list, and the current state of its buddy is disregarded. When a block is returned to the system, the watermark-based lazy buddy system does not always free the block "globally" by marking it as free in the system-wide bit-map and trying to coalesce the block with its buddy. Instead, it sometimes frees the block only "locally" by putting it on the free list for its class and labeling it "locally-free." A "locally-free" block is only free in the free list, and it can be accessed only through the corresponding list head; the system-wide bit-map shows the block as still allocated and unavailable for coalescing even if its buddy is free.

Whether a returning block is freed locally or globally depends on the *mode* of the class, that is, on how heavily blocks in that class are being used relative to the two watermark parameters. The block usage for a particular class is denoted θ and is defined as the proportion of blocks of that class currently in the free list relative to the total number of blocks of that class in existence. At any time each block class can be in one of three modes: *lazy* mode ($\theta \leq \theta_l$, blocks are being heavily used and block free operations are local), *reclaiming* mode ($\theta_l < \theta \leq \theta_s$, blocks are not being heavily used and block free operations are global), or *accelerated* mode ($\theta > \theta_s$, requests for blocks of this class are drying up; block free operations are global and when a block is freed the system also globally frees a

second "locally-free" block in that class if there is one). A free block that coalesces with its buddy in one class is then treated as a free block returning to the next class, and that free operation is controlled by the mode of that class. Block releases in a class in reclaiming mode may have one coalescing delay for that class; block releases in a class in accelerated mode may have two coalescing delays.

The system normally operates in the lazy mode, which has minimal bookkeeping and low operational costs. The reclaiming and accelerated modes allow the system to detect when block usage patterns change and to recover blocks that are uncoalesced because they were freed in the lazy mode. In essence, the lazy coalescing rules add a damping mechanism to the buddy system's coalescing process by anticipating that the block will soon be reused. The acceleration mechanism speeds up the block coalescing process to respond to shifting memory demands and guarantees that when all blocks are released all memory will be coalesced.

The correctness of the watermark-based lazy buddy system for $0 \le \theta_l \le \theta_s \le ½$ is shown by defining for each class of blocks a space of legal states called the *lazy space* and proving that the watermark-based lazy coalescing rules always keep the block class within that space. In this paper we describe a different lazy coalescing policy, called the DELAY-2 algorithm, that does away with the watermark device and focuses directly on maintaining the state of each block class within the lazy space. The resulting implementation is much simpler, and the DELAY-2 system, by concentrating directly on keeping block classes in the lazy space, can be more aggressive in making local instead of global block free operations, thus reducing the cost of memory operations. The DELAY-2 system is lazier than the watermark-based system, causing some sluggishness when memory demand patterns change abruptly. However, this is not a major concern in environments such as the UNIX kernel. The DELAY-2 buddy system is described in more detail in the following sections.

3.2 The State-Driven DELAY-2 Algorithm

To describe formally the DELAY-2 algorithm, we first define the lazy space.

The state S of a class of blocks can be described by a non-negative triplet (N,F_l,F_g), where N is the total number of blocks in that class in existence, F_l is the number of "locally-free" blocks in the free list, and F_g is the number of "globally-free" blocks in the free list. To be able to coalesce all the original memory and also to bound the cost of a freeing operation, we need to restrict the state of each block class in the lazy space.

The **lazy space** L_N for integers $N \ge 0$ is defined to be the set of states (N,F_l,F_g) satisfying the following two conditions:

(i). $0 \le F_g \le \lceil N/2 \rceil$,
(ii). $F_g \le N - 2F_l$.

Condition (i) is a necessary condition of the coalescing rules for the "globally-free" blocks. Condition (ii) reflects the effect of boundedness (coalescing delays) on the number of "locally-free" blocks. Given condition (ii), we have $F_l \le (N-F_g)/2$ and $F_l \le N/2$ since $0 \le F_g$. This leads to $0 \le F_l \le \lfloor N/2 \rfloor$, which is similar to condition (i) for F_g.

When the DELAY-2 buddy system frees blocks, it frees them either locally or globally depending on what it must do to keep its state in the lazy space. When a block is being freed, if the system can make it a local free operation and still be in the lazy space, it will free the block locally. Otherwise, it will free the block globally. If the new state after the global free operation is still outside the lazy space, the system will take a "locally-free" block from its free list and free it globally to get back into the lazy space. The DELAY-2 algorithm is described in detail in Figure 1.

3.3 State Transition Diagram

Although we might expect the DELAY-2 buddy system to be similar to the laziest policy(½,½) of the watermark-based lazy buddy policy, there is one subtle difference because the DELAY-2 buddy system concentrates directly on maintaining its state in the proper lazy space. The DELAY-2 buddy system decrements the F_l variable first, and is more aggressive than the watermark-based lazy buddy policy(½,½) about staying in the lazy space without coalescing. Thus, the state transition diagram for the DELAY-2 buddy system is more compact and less complicated. Furthermore, because the DELAY-2 buddy system coalesces less often than the watermark-based lazy buddy policy(½,½), intuitively the DELAY-2 buddy system should be less expensive. Figure 2 shows the state transition diagram for lazy space L_6. To simplify the diagram, we do not show transitions for states in different lazy spaces.

3.4 All Blocks Coalesced When All Returned

An important property of any dynamic memory management policy is that all blocks are coalesced when they are all returned. Obviously the buddy system has this property since it laboriously coalesces whenever possible. For the DELAY-2 buddy system, we have to verify that this property holds such that no block becomes inaccessible.

Given the system is in state $S(N,F_l,F_g)$, the next state S_{next} after an operation is as follows:

(I). If the next operation is a block request,
  If $F_l>0$, then S_{next} becomes (N,F_l-1,F_g),
  Else if $F_g>0$, then S_{next} becomes $(N, 0,F_g-1)$.
  When $F_l=F_g=0$, first get two blocks by splitting a larger one, and then S_{next} becomes $(N+2,1,0)$.

(II). If the next operation is a block free operation,
  If (N,F_l+1,F_g) is in L_N, then
    mark it "locally-free" and free the block locally,
    S_{next} becomes (N,F_l+1,F_g).
  Else
    mark it "globally-free" and free the block globally; then, one of the following two cases will hold:
    (1). If buddy is not free, block cannot coalesce and
      S_{next} becomes (N,F_l,F_g+1).
      if S_{next} is not in L_N, then
        free one "locally-free" block globally,
        S_{next} becomes (N,F_l-1,F_g+2) without coalescing, or
        S_{next} becomes $(N-2,F_l-1,F_g)$ with coalescing.
    (2). If buddy is free, block coalesces and
      S_{next} becomes $(N-2,F_l,F_g-1)$.
      if S_{next} is not in L_{N-2}, then
        free one "locally-free" block globally,
        S_{next} becomes $(N-2,F_l-1,F_g)$ without coalescing, or
        S_{next} becomes $(N-4,F_l-1,F_g-2)$ with coalescing.

Figure 1. The DELAY-2 Buddy System

In general, the amount of memory managed by the buddy or lazy buddy systems does not have to be exactly 2^k [1]. Thus, each block class is initialized with either $N=0$ or $N=1$, i.e., the initial state is either (0,0,0) or (1,0,1). Those classes starting with $N=1$ have a block that is non-coalesceable. By definition, we always have $N\geq F_l+F_g$. The design of the buddy system dictates that N change by increments or decrements of 2. Thus, as the state changes, classes that start with $N=1$ always have odd integer N and the classes that start with $N=0$ always have even integer N.

Now, to show that all coalesceable blocks are coalesced when all are returned, we need to show that $N=F_l+F_g$ implies $N=F_l=F_g=0$ if N is an even integer or $N=F_g=1$, $F_l=0$ if N is an odd integer. Without loss of generality, we assume that all requests can be satisfied if necessary by splitting a block of the next larger size; the state transition from $(N, 0,0)$ to $(N+2,1,0)$ or $(N+2,0,1)$ can be viewed as a block free operation taking place when state is $(N+2,0,0)$.

Before proving the main theorem, we first prove two lemmas.

Lemma 1. For any state $S(N,F_l,F_g)$ under the DELAY-2 buddy system, $0\leq F_g\leq \lceil N/2\rceil$.

Proof: Since all blocks are paired as buddies (except one when N is odd and the class must start with $N=1$) and buddies are coalesced when both are "globally-free," we cannot have more than half of the total coalesceable block population in the "globally-free" category. That is, $F_g\leq N/2$ if N is even and $F_g\leq (N+1)/2$ if N is odd. Combining the two, we have $0\leq F_g\leq \lceil N/2\rceil$. ■

Lemma 1 leads to condition (i) of the lazy space definition.

Lemma 2. For a given state S in L_N ($N\geq 0$), a block request or free operation under the DELAY-2 buddy system brings the next state S_{next} to one of the four lazy spaces L_{N+2}, L_N, L_{N-2} (only when $N\geq 2$), or L_{N-4} (only when $N\geq 4$).

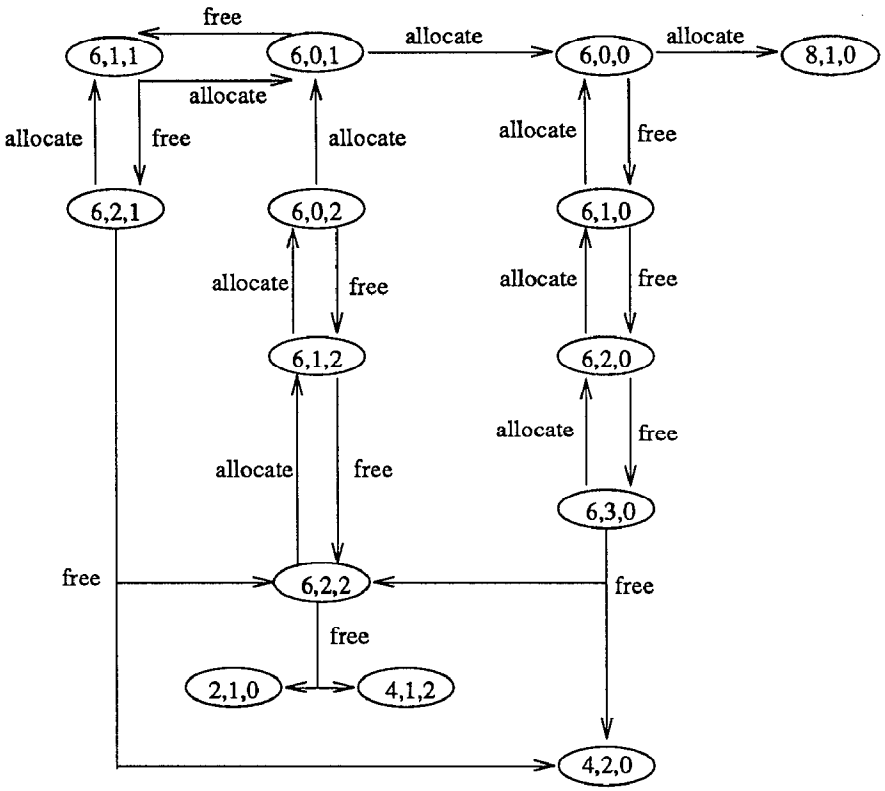

Figure 2. State Transition Diagram for Lazy Space L_6

Lemma 2 states the transitive closure property of the lazy space with respect to the DELAY-2 buddy system. The proof, too long to be included here, is given in Appendix A. With these two lemmas, the main theorem for the DELAY-2 buddy system can be proven.

> **Theorem.** For any state (N,F_l,F_g) under the DELAY-2 buddy system, $N=F_l+F_g$ implies $N=F_l=F_g=0$ if N is even, or $N=F_g=1$, $F_l=0$ if N is odd.
>
> *Proof:* Given the system starts at (0,0,0) or (1,0,1), we know from Lemma 2 that the state (N,F_l,F_g) reached by the DELAY-2 buddy system satisfies conditions (i) and (ii) of the definition. Re-arranging condition (ii), we have $F_l+F_g \le N-F_l$. Since $N=F_l+F_g$, we have $N \le N-F_l$ or $F_l \le 0$. Note that F_l is non-negative; thus, $F_l=0$. Substituting F_l, $N=0+F_g \le \lceil N/2 \rceil$. Depending on whether N is even or odd, we have $N \le 0$ and $N \le 1$, respectively. If N is even, we conclude $N=F_l=F_g=0$. If N is odd, we conclude $N=F_g=1, F_l=0$. ■

4. A SLACK VARIABLE-BASED IMPLEMENTATION

Figure 1 shows one way that the DELAY-2 buddy system can be implemented, that is, by maintaining a 3-tuple state variable for each block class. However, condition (ii) of the lazy space definition suggests a simpler implementation. For each class of block we define a non-negative slack variable $D=N-2F_l-F_g$. For either of the two possible initial states, (0,0,0) or (1,0,1), D is initially 0. The DELAY-2 buddy algorithm given in Figure 1 shows how to increment and decrement N, F_l, and F_g values when allocating and freeing blocks, and it is simple to calculate the equivalent change to the slack variable D. This leads to an efficient implementation based only on the slack variable D, shown in Figure 3. We do not need to maintain the three variables N, F_l, and F_g. Instead, a single variable will suffice. Intuitively, the slack variable reflects the slackness of the memory state within the corresponding lazy space. When D is greater than 2, the system is well within the lazy space and can afford to make a block "locally-free." When D is 1, the system is near the edge of the lazy space and must free a block globally to stay in the lazy space. When D is 0, the system is on the edge and must globally free an additional "locally-free" block to force the state back into the lazy space. The correctness of the implementation using the slack variable D follows the same proof in Appendix A and is not repeated. The critical part of the proof is to show that when D is 0, there is at least one "locally-free" block.

We can keep both "locally-free" and "globally-free" blocks on the same doubly-linked free list by partitioning the list such that "locally-free" blocks are always at the

(Initial value of D is 0)
Given current value D, the slack variable D_{next} after an operation is as follows:

(I). if the next operation is a block request,
    If there is any free block, select one to allocate
      if the allocated block is "locally-free,"
        then D_{next} becomes $D+2$,
        else D_{next} becomes $D+1$.
    Otherwise,
        first get two blocks by splitting a larger one into two
          (which implies a recursive use of this algorithm);
        allocate one and mark the other one "locally-free";
        D_{next} remains unchanged for this class (but may change for
            other classes because of the recursive call).

(II). if the next operation is a block free operation,
    Case $D \geq 2$:
        mark it "locally-free" and free it locally;
        D_{next} becomes $D-2$.
    Case $D=1$:
        mark it "globally-free" and free it globally;
        D_{next} becomes 0.
    Case $D=0$:
        mark it "globally-free" and free it globally;
        select one "locally-free" block, mark it "globally-free" and free it globally,
        D_{next} remains 0.

Figure 3. A Slack Variable-based Implementation of DELAY-2 Buddy System

head of the list. When blocks are being freed, "locally-free" blocks are inserted at the head of the list and "globally-free" blocks (if their buddies are not free) are inserted at the tail of the list. With these rules and with an initially empty list, it is straightforward to show that the list is always composed of two distinct parts (with one or both possibly empty). This strategy makes it simple to reclaim an extra "locally-free" block when we need to free it globally—we merely take the block from the head of the list. If the head of the list is not a "locally-free" block, there are no "locally-free" blocks in the list. Note that we always allocate blocks from the head of the list because those are the "locally-free" blocks and are least expensive to allocate.

5. *MEASUREMENT AND COMPARISON*

To gather empirical performance data for the DELAY-2 algorithm and compare it with the watermark-based lazy buddy policy(½,½) and the standard buddy system, we implemented a prototype STREAMS buffer manager [5, 6, 7] based on the DELAY-2 buddy system. We drive the memory system with a simulated multi-user timesharing workload [8] of three different stochastic structures. Memory demand traffic is characterized by a vector of pairs that gives the mean time between requests and mean block holding times for each class of block. Table 1 shows the vector for each simulated heavy user.

Table 1. Memory Load Parameters

Block Size (bytes)	Interarrival Time (ms)	Holding Time (ms)
16	113	54
64	112	5
128	138	654
256	528	12
512	10142	14
1024	4804	17
2048	60	275

For our experiments we chose the vector corresponding to three simulated users, which imposes a load equivalent to that seen under normal operating conditions on our microcomputer.

Table 2. Means (Standard Deviations) for *allocb()* and *freeb*() Calls (in microseconds)

Memory Policies		Workload Scenarios		
		D/D/m	M/M/m	M/M/m Batch(4)
allocb ()	Buddy System	117.0 (15.8)	110.9 (31.9)	136.8 (35.5)
	Lazy Buddy (½,½)	82.1 (10.4)	86.9 (15.5)	91.5 (26.5)
	DELAY-2 Buddy System	79.0 (11.0)	81.9 (14.8)	85.5 (23.4)
freeb ()	Buddy System	160.0 (14.4)	173.0 (25.3)	183.9 (31.0)
	Lazy Buddy (½,½)	106.3 (12.5)	131.1 (72.8)	150.4 (92.3)
	DELAY-2 Buddy System	100.2 (15.3)	110.0 (45.5)	129.9 (72.4)

Table 3. Blocking Probability Statistics

Memory Policies	Workload Scenarios		
	D/D/m	M/M/m	M/M/m Batch(4)
Buddy System	0.0	0.000135	0.0218
Lazy Buddy (½,½)	0.0	0.000257	0.0328
DELAY-2 Buddy System	0.0	0.000297	0.0318

Given these parameters, we can generate three types of workloads with different stochastic structures. The first workload is that of the D/D/m [9] queue, which assumes for each class of blocks deterministic arrivals, deterministic service time, and multiple servers. In this context, a *server* is a block and *service time* corresponds to the block holding time. This type of workload is frequently used for testing and evaluating memory management systems. The second workload is the M/M/m queue, which assumes Poisson arrivals and exponential service times. If the goodness of the DELAY-2 buddy system depends on deterministic or steady-state usage, the M/M/m workload will allow us to analyze the algorithm in a non-optimal environment. The third workload has the same M/M/m structure, but is modified to have batched arrivals; the service times for customers arriving in the same batch are not the same, but are drawn from the same distribution. This workload is an approximation of the traffic observed in database transaction systems; the block requests arrive in bursts corresponding to transactions in the database applications.

The primary performance metrics for memory management systems in our work are the mean and standard deviation of the costs to allocate (*allocb*) or free (*freeb*) a block. We also measure the *blocking probability* or allocation failure rate. During the experiments, each system was given the same fixed amount of memory to manage. If a particular request for memory could not be satisfied from the fixed arena, the system did not increase the arena but simply failed the request. Thus, systems with lower failure rates are systems that manage their memory more efficiently. We present these statistics for the three memory managers under the different workloads and also chart the cost distributions for the M/M/m workload. All measurements were made on an AT&T microcomputer running System V Release 3.2 using CASPER, a high resolution timing and tracing tool [10].

The means and the standard deviations for the three workloads for the standard buddy system, watermark-based lazy buddy policy(½,½), and DELAY-2 buddy system are summarized in Table 2. The differences in *freeb*() costs are more dramatic than those in *allocb*() costs since most of the overhead is absorbed in the release and coalesce phase. Considering the combined *transaction* cost of an *allocb*() and a *freeb*() pair, and comparing the DELAY-2 buddy system with the (½,½) policy, the DELAY-2 buddy system is 5% better for D/D/m workload, 12% better for M/M/m workload, and 11% better for M/M/m batch(4) workload. Relative to the standard buddy system, the DELAY-2 buddy system is about 35%, 32%, and 33% faster, respectively.

Table 3 shows the differences in blocking probabilities among the different policies. The DELAY-2 buddy system differs only slightly from the watermark-based lazy buddy policy(½,½). The reduced operational costs are gained without any significant penalty in increased blocking probabilities.

Figure 4 compares the cost distributions of *allocb*() and *freeb*() for the buddy system, watermark-based lazy buddy policy(½,½), and DELAY-2 buddy system with the M/M/m workload. The small spikes on the tail of the distribution for the *freeb*() operation reflect the cost incurred when the system recursively coalesces small blocks into much larger blocks.

The measurement results presented in this section were obtained in a carefully controlled environment and with a synthetic memory workload so that we could compare the

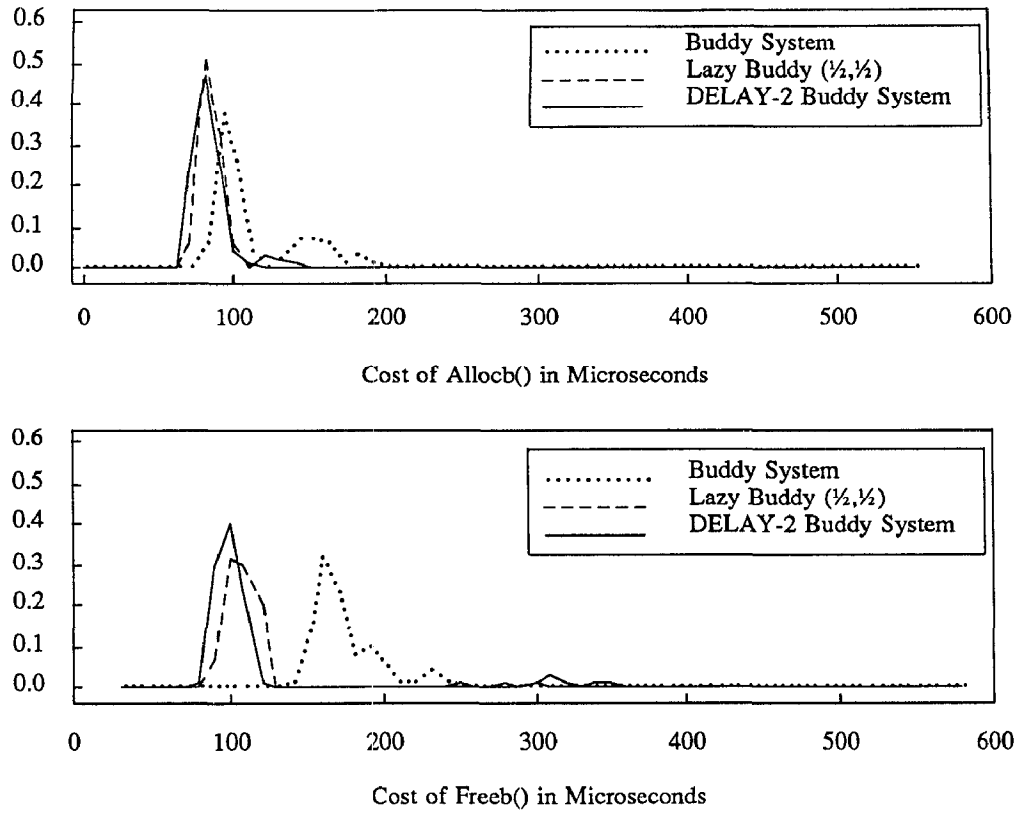

Figure 4. Probability Density Functions for Exponential Interarrival and Service Times

different systems. Under a live load, we measured operational costs similar to those seen with the D/D/m workload. Also, we observed as many as 200 memory allocation and free operations per second. Compared with the standard buddy system, the DELAY-2 buddy systems saves almost 100μs per allocate and free transaction, and thus saves up to 2% in system capacity.

6. MEMORY EFFICIENCY

Fast memory management is crucial in an operating system environment, but efficient memory utilization is also important. Memory efficiency is reduced by both *internal* and *external* fragmentation. Binary buddy systems always allocate memory in sizes of powers of 2, and the difference between the size requested and the size allocated is unused. This problem, called the internal fragmentation, has been studied elsewhere [1]. Although buddy systems have worse internal fragmentation than first-, better-, or best-fit systems, we were willing to pay that price to get the lower operational costs. Furthermore, operating system data structures are frequently small and are often close to a power of 2 in size, both of which help reduce the amount of memory lost to internal fragmentation.

External fragmentation measures the proportion of memory allocated relative to the amount of memory being managed. In the experimental system, the amount of memory managed by the system was held constant and requests were failed if there was not enough free memory left in the arena to satisfy the request. The request failure rate is thus an indirect measure of memory utilization or external fragmentation.

Instead of failing requests, we can allow the pool of managed memory to grow. Less efficient managers will require more memory to satisfy a given load; thus, we can use the maximum amount of memory taken by the different managers under varying loads as a measure of their memory efficiency. Figure 5 shows these data for M/M/m workload. As expected, the standard buddy system is the most efficient. All block requests in our experiment were in sizes of powers of two and thus the buddy system could achieve close to 100% efficiency. In the worst case the DELAY-2 buddy system requires an additional 4.25 kbyte or 14% more memory, but on average it needs only 6% more. We also show the memory requirements of the fast fits memory manager [11, 12] to show the efficiency of a non-buddy-based system. In our experiments the fast fits manager fell between the standard buddy and the watermark-based buddy systems.

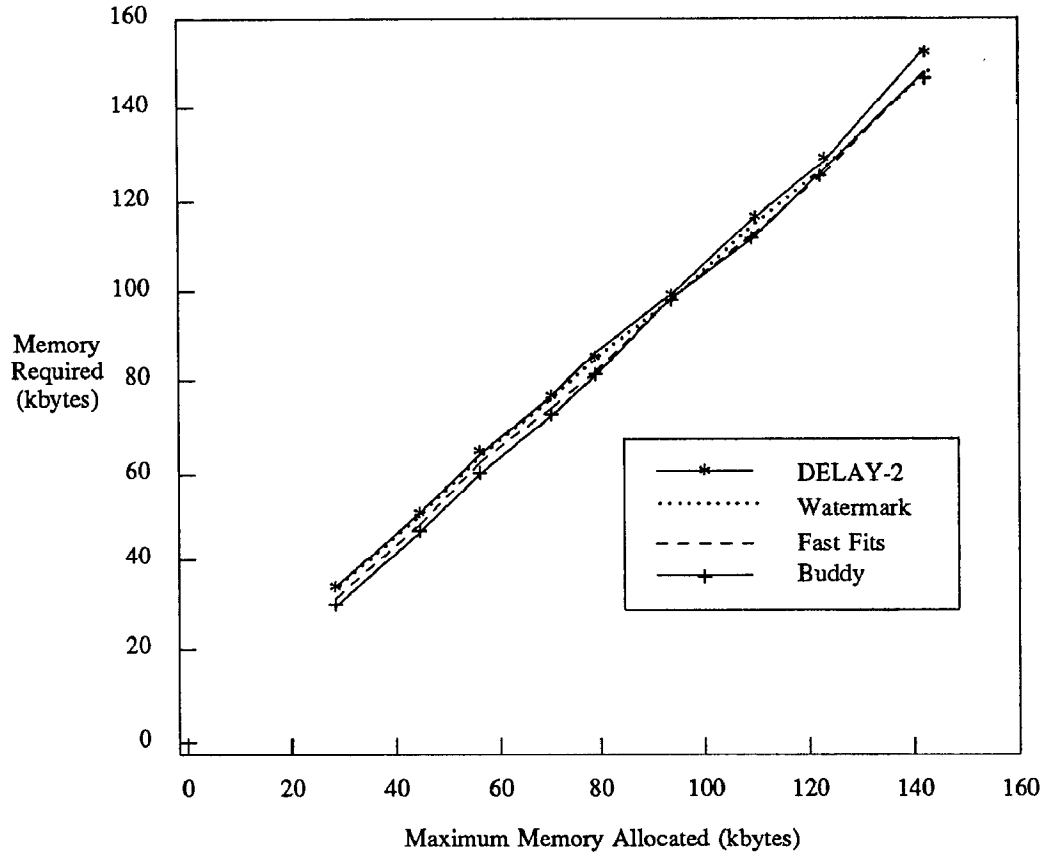

Figure 5. Memory Requirements for Dynamic Memory Managers

7. *CONCLUSION*

The design of the DELAY-2 buddy system was inspired by the formal proof of the transitive closure within the lazy space for the watermark-based buddy system. Instead of relying on the watermark device to keep the memory state in the lazy space indirectly, we focus directly on keeping the state within the lazy space. The algorithm can be implemented using a single variable that measures the distance the current state is from the lazy space boundary. The experimental data shows that this simpler implementation gives a faster algorithm.

The watermark-based lazy buddy policy(θ_l,θ_s) offers fine control over the coalescing policy of the buddy system. However, applications such as the UNIX System kernel do not need such fine control; furthermore, it is difficult to tune the parameters. For these applications, the DELAY-2 buddy system provides an efficient memory manager with low operational costs and low request blocking probability. In the DELAY-2 buddy system, the worst-case time for a free operation is bounded by two coalescing operations per class (an important feature for an operating system environment), and when all blocks are returned to the system, the system memory is coalesced back to its original state. This ensures that the memory space can be completely shared.

The dynamic kernel memory manager supplied in UNIX System V Release 4.0 is based on the DELAY-2 buddy system.

8. *ACKNOWLEDGEMENTS*

Don Milos and Nancy Mintz made significant contributions and provided much practical support during this study. We would also like to thank the SOSP referees for their insightful comments and valuable suggestions.

REFERENCES

1. D. E. Knuth, *The Art of Computer Programming, Vol I, Fundamental Algorithms*, Addison-Wesley, Reading, Mass. 1968.

2. J. L. Peterson and T. A. Norman, "Buddy Systems," *Commun. of the ACM*, Vol. 20, No. 6, June 1977, pp. 421-431.

3. T. P. Lee and R. E. Barkley, "Configuring UNIX® STREAMS Communication Buffers Based on an Erlang Traffic Model," *Proc. of the 1988 Computer Networking Symposium*, April 1988, pp. 230-234.

4. T. P. Lee and R. E. Barkley, "A Watermark-based Lazy Buddy System for Kernel Memory Allocation," *Proc. of the 1989 Summer USENIX Conference*, June 1989, pp. 1-13.

5. D. M. Richie, "A Stream Input-Output System," *AT&T Bell Lab. Tech. Journal*, Vol 63, No. 8, Part 2, October 1984, pp. 1897-1910.

6. *UNIX System V: STREAMS Primer*, AT&T Select Code 307-229, 1986.

7. *UNIX System V: STREAMS Programmer's Guide*, AT&T Select Code 307-227, 1986.

8. S. Gaede, "A Scaling Technique for Comparing Interactive System Capacities," *Conference Proc. of CMG XIII*, December 1982, pp. 62-67.

9. L. Kleinrock, *Queuing Systems Volume I: Theory*, John Wiley & Sons, 1975.

10. R. E. Barkley and D. Chen, "CASPER the Friendly Daemon," *Proc. of the 1988 Summer USENIX Conference*, June 1988, pp. 251-260.

11. C. J. Stephenson, "Fast Fits: New Methods for Dynamic Storage Allocation," *Proc. of the Ninth ACM Symposium of Operating Systems Principles*, October 1983, pp. 30-32 (extended abstract).

12. C. J. Stephenson, "Fast Fits: New Methods for Dynamic Storage Allocation," Research Report, T. J. Watson Research Center, Yorktown Heights, NY.

APPENDIX A - Proof of Transitive Closure

Lemma 2. For a given state S in L_N ($N \geq 0$), a block request or free operation under the DELAY-2 buddy system brings the next state S_{next} to one of the four lazy spaces L_{N+2}, L_N, L_{N-2} (only when $N \geq 2$), or L_{N-4} (only when $N \geq 4$).

Proof: From Lemma 1 we know that all S_{next} states satisfy condition (i) of the lazy space definition. To show condition (ii), the proof follows the steps of the DELAY-2 buddy system depicted in Figure 1. We assume that before the block request or free operation, $S(N,F_l,F_g)$ is in L_N, i.e., $F_g \leq N-2F_l$. We shall show that after the operation, the resulting state S_{next} is in one of the L_N's.

Case I. Block Request. It is easy to show that all possible three states S_{next} are in their respective L_N's.

1. $S_{next}=(N,F_l-1,F_g)$ in L_N is implied by $F_g \leq N-2(F_l-1) \leq N-2F_l$.
2. $S_{next}=(N,0,F_g-1)$ in L_N is implied by $F_g-1<F_g \leq N$.
3. $S_{next}=(N+2,1,0)$ in L_{N+2} is implied by $F_g=0<N-0=(N+2)-2$.

Case II. Block Free.

We only have to deal with the case that (N,F_l+1,F_g) is not in L_N and the block is marked "globally-free" and freed globally. One of the following two cases will hold:

1. S_{next} becomes (N,F_l,F_g+1) without coalescing. We only have to deal with the case that S_{next} is in not L_N, i.e., $F_g+1>N-2F_l$. Here, the algorithm will free one locally-free block globally, S_{next} becomes (N,F_l-1,F_g+2) without coalescing, or S_{next} becomes $(N-2,F_l-1,F_g)$ with coalescing. In either case, the resulting S_{next} is in L_N and L_{N-2}, respectively. This is easily shown by $F_g+2 \leq N-2F_l+2 \leq N-2(F_l-1)$ and $F_g \leq N-2F_l \leq (N-2)-2(F_l-1)$. However, we have to show that there exists at least one "locally-free" free block. If there is no "locally-free" block, we have $F_l=0$ or $F_g+1>N$. From condition (i), $\lceil N/2 \rceil \geq F_g$, we have an inequality $\lceil N/2 \rceil+1>N$. This is true only when $N=0$ or $N=1$. First, it is impossible to have a free operation when $N=0$. If $N=1$ and recall that $F_l=0$, the next state $(1,0,1)$ is in L_1 which contradicts the assumption. Thus, we have shown that F_l cannot be 0 in question.

2. S_{next} becomes $(N-2,F_l,F_g-1)$ with coalescing. Again, we only have to deal with the case that S_{next} is in not L_{N-2}, i.e., $F_g-1>(N-2)-2F_l$. Here, the algorithm will free one locally-free block globally, S_{next} becomes $(N-2,F_l-1,F_g)$ without coalescing, or S_{next} becomes $(N-4,F_l-1,F_g-2)$ with coalescing. In either case, the resulting S_{next} is in L_{N-2} and L_{N-4}, respectively. This is easily shown by $F_g \leq N-2F_l \leq (N-2)-2(F_l-1)$ and $F_g-2 \leq N-2F_l-2 \leq (N-4)-2(F_l-1)$. Similar to earlier treatment, we have to show that there exists at least one "locally-free" free block. If there is no "locally-free" block, we have $F_l=0$ or $F_g-1>(N-2)$ or $F_g+1>N$. From condition (i), $\lceil N/2 \rceil \geq F_g$, we have an inequality $\lceil N/2 \rceil+1>N$. This is the same inequality dealt above and same argument applies.