Storage Allocation for Real-Time, Embedded Systems [⋆]

Steven M. Donahue[1], Matthew P. Hampton[1], Morgan Deters[1], Jonathan M. Nye[1], Ron K. Cytron[1], and Krishna M. Kavi[2]

[1] Washington University
Department of Computer Science
Saint Louis, MO 63130, USA,
cytron@cs.wustl.edu,
WWW home page: http://www.cs.wustl.edu/~doc/
[2] University of North Texas
P.O. Box 311277
Denton, Texas 76203

Abstract. Dynamic storage allocation and automatic garbage collection are among the most popular features that high-level languages can offer. However, time-critical applications cannot be written in such languages unless the time taken to allocate and deallocate storage can be reasonably bounded. In this paper, we present algorithms for automatic storage allocation that are appropriate for real-time and embedded systems. We have implemented these algorithms, and results are presented that validate the predictability and efficiency of our approach.

1 Introduction

Languages featuring dynamic storage allocation and automatic garbage collection continue to grow in popularity—such languages include Java (*Java* is a registered trademark of Sun Microsystems) and ML. Following standard terminology, storage requests are satisfied by *allocating* storage from a *storage heap.* Such storage is *live* or *busy* until such time as the storage is declared *dead.* For languages like Java, an automatic *garbage collection* algorithm can detect dead objects. Other languages offer primitives for dynamically asserting the death of an object. In any case, once the object is declared dead, the storage associated with the object can be *deallocated*, which makes that storage available for subsequent reallocation.

Developers of real-time and embedded systems have been slow to embrace automatic storage management for the following reasons.

– *Real-time applications require predictable execution.* Automatic storage management incurs overhead that can be difficult or impossible to predict. In this paper, we examine this issue with respect to storage allocation. For real-time applications, allocation time must be bounded.

[⋆] This work is supported by the National Science Foundation under grant 0081214 and by DARPA under contract F33615–00–C–1697

– *Embedded systems require predictable storage bounds.* Embedded systems are typically deployed without the advantages of a virtual memory backing store. Thus, the heap cannot be extended without physically inserting more RAM into the system.
  As a result, the storage requirements for these kinds of applications must be known in advance. For our purposes, this implies that the size of the run-time heap is fixed and known *a priori*.

Unfortunately, the performance of automatic storage management continues to be problematic for developers of embedded or real-time systems. While there are currently several reasons for this, one concern is that the time taken to perform storage-management functions cannot easily be bounded. Time-critical applications cannot abide such behavior. In our view, the term *time-critical* applies to both of the following.

– An application may be time-critical in the sense that some instruction sequences must execute in a reliable timeframe. Embedded and real-time applications often have this property.
– Hardware support for storage-management functions can be time-critical in the sense that such hardware must be clocked at a predetermined rate for synchronous operation. In this situation, it is better to perform a little work at each storage-management operation than to have some operations execute for-free and others take a very long time.

Ironically, Java was initially designed as a language for embedded-systems applications, but its storage management remains problematic for the following reasons.

– Storage allocation [9] usually involves searching a *free-list* of storage blocks. For garbage-collected programs, the free-list tends to diminish until the point of collection. Following collection, the free-list typically contains a large number of (former) objects. To satisfy a single storage request, searching the free-list could take time proportional to the size of the list—unacceptable for time-critical situations.
– Garbage collection [8] techniques usually require marking (a subset of) the program's live objects; the storage for the unmarked objects can then be returned to the storage manager for reallocation. Collection cycles can happen unexpectedly and can take considerable time; moreover, negative effects on the data and instruction caches are drastic [4, 2, 1].
– As the program executes, the heap becomes *fragmented*: relatively small holes develop in the heap storage area. To defragment the heap, objects are either copied or compacted during the collection cycle, taking more execution time and toll on the cache.

If the time taken to execute storage-management functions can be reasonably bounded, then languages such as Java could better support time-critical applications. Moreover, it then becomes possible to relegate storage-management activities to hardware where better performance might be obtained.

Our paper's contributions can be summarized as follows.

- Results are presented using Knuth's *buddy system* [6], which bounds the time taken to satisfy a storage allocation request.
- A variation of the buddy system is presented that delays recombination. We present results on the effectiveness and efficiency of this variation.
- An algorithm is presented for defragmenting a buddy system heap. The algorithm is specialized toward satisfying a single request, rather than a mass defragmentation of the heap which can be disruptive to program execution.

Our paper is organized as follows: Section 2 explains our approach and implementation using simple examples. Section 3 presents experiments based on this implementation. Section 4 presents conclusions and ideas for future work in this area.

2 Approach

In this section, we describe our approach for obtaining free blocks of storage in bounded time. Wilson presents an excellent survey of storage allocation [9], and the buddy system upon which we base our work is described well by Knuth [6].

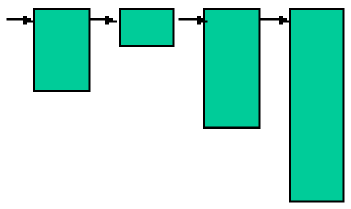


Fig. 1. Unstructured list allocator

2.1 Motivation for Segregated Free-Lists

We first consider a common storage allocation technique, based on maintaining a relatively unstructured list of available blocks, as shown in Figure 1. A storage request is satisfied by searching that free-list for (typically) the first block that is sufficiently large to satisfy the request. While this approach works well in practice, it is possible that the only block that can satisfy a given request is at the far end of the free-list. Because the free-list structure is a function of the allocating program's behavior, it is not easy to bound the time needed to satisfy an allocation request, even if we assume the free-list contains a block of suitable size.

By contrast, consider a storage allocator whose free-lists are *segregated* by size, as shown in Figure 2. For efficiency in obtaining a block of the desired size,

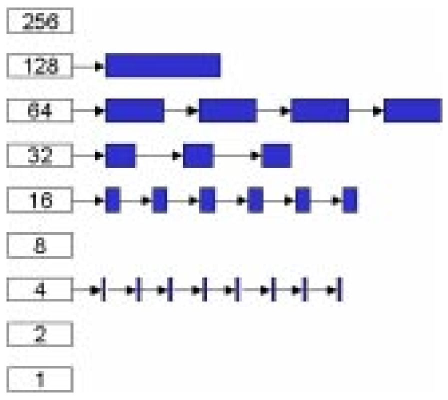


Fig. 2. Data structure for buddy system; objects are segregated by size, and each list of equally-sized blocks is referenced from the *list display*, shown at the left

an indexable slot is reserved to head a linked list for each possible block size. A request for a block of a given size can be satisfied by simply returning a block from the appropriate list. This takes constant time, assuming that a block is available on the appropriate list.

2.2 Buddy System Allocation

Our approach for storage allocation is a segregated-by-size technique, based on Knuth's buddy system [6]. Each storage request is resolved to a block of size 2^k for some positive, integral value of k. Figure 3(a) shows the buddy system's structure in its initial state, assuming the heap is 256 bytes.

The buddy system operates as follows:

1. When the program requests memory, the allocator first calculates the smallest power of 2 that is larger than or equal to the size requested. More specifically, a request of size s is translated into a request of size $2^k, k = \lceil \log_2 s \rceil$.
2. The free-list at index k is consulted for an available block.
3. If a block of size 2^k is not available, then two such blocks can be obtained through bisection of a block of size 2^{k+1}. Figure 3(b) shows the result of subdividing the initial heap into two sub-blocks.
4. Applying this strategy recursively, increasingly larger blocks can be subdivided until a block of size 2^k can be obtained.

For example, the heap in Figure 2 has blocks available of size 16. Thus, a request for a block of size 10 can be satisfied immediately, with the resulting block returned in the time it takes to unlink a block from the size-16 free-list.

Further, consider a request for a block of size 8. Because the list of blocks of size 8 is empty, the buddy system hunts upwards for a larger block that can be subdivided to obtain the desired size. The time necessary for that search is

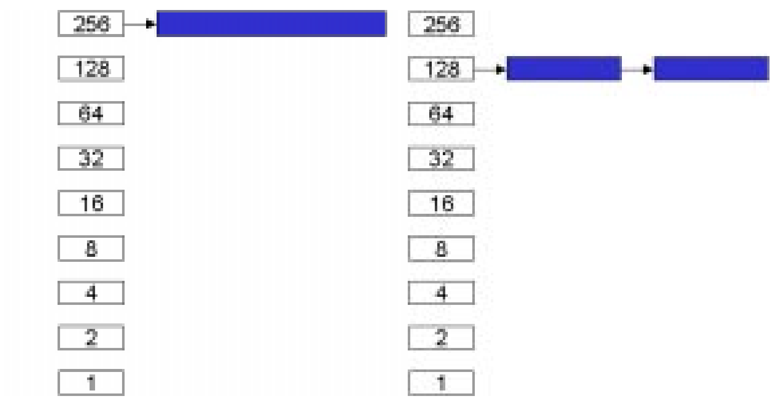


Fig. 3. A block can be divided into two sub-blocks

$O(log(M))$ where M is the size of the storage heap. For embedded and real-time systems, we assume that the heap size is fixed and that $O(log(M))$ time is considered efficient.

2.3 Buddy System Deallocation

When blocks are deallocated, a common problem for most storage-management algorithms is the coalescing of free blocks that happen to lie consecutively into larger blocks. The buddy system greatly simplifies this task. When a block of size 2^{k+1} is bisected into two blocks of size 2^k, the resulting blocks are said to be *buddies* of each other. A buddy of size 2^k can implicitly compute its buddy's address by flipping a predetermined bit of its own address—typically bit k where bit 0 is the rightmost bit.[1]

Figure 4 shows the result of requesting a block of size 16 given the initial condition shown in Figure 2. The initial block is recursively subdivided until two blocks of size 16 are obtained. One of those blocks is returned to satisfy the allocation request, and the other block remains on the free-list for blocks of size 16.

When storage is returned, the buddy system eagerly joins buddies to create ever larger blocks. Thus, if the block allocated in Figure 4 is immediately deallocated, buddies are joined together repeatedly until the heap is returned to the state shown in Figure 2.

2.4 Buddy System Implementation and Behavior

As shown in Figure 2, the buddy system normally keeps an array of linked lists of identically-sized blocks. Each element of the array heads the linked list for blocks

[1] Without loss of generality, we assume the heap's origin is address 0.

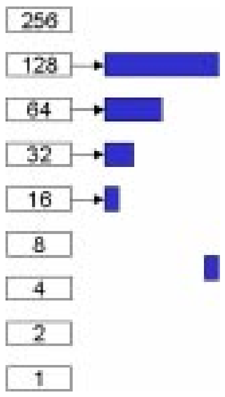


Fig. 4. Allocation using the buddy system

of a given power of two. When a block is deallocated, a check is performed to see if that block's buddy is also free. In support of this test, each block is equipped with one bit to reflect whether it is busy. The bit must be present when the block is actually in use by the allocating program.

When a block is not in use, it is kept on a linked list as shown in Figure 2. The blocks are actually maintained in a *doubly-linked* list, which facilitates quick deletion of a block from its linked list. Space for these links can come from the block itself, since while on the free-list, the space is not otherwise in use. In addition, it is necessary for the block to keep track of its size, which implies that the smallest block that can be managed using the buddy system must be able to accommodate two pointers, along with the size of the block.

Although the buddy system has the advantages stated in this paper, it has not been widely used for storage allocation in programming-language systems for the following reasons.

1. The buddy system tends to fragment storage [6]. This means that the heap may contain sufficient storage to satisfy a request, but the storage is not contiguous within a buddy's boundary. There are two sources of fragmentation:
   Internal fragmentation occurs within a block, when a request for storage is rounded up to a power of two. Consider a request for a block of size s that is translated into a request for size 2^k. For such a request, the number of wasted bytes w must satisfy $0 \leq w < 2^{k-1}$.
   External fragmentation occurs because free storage can be distributed among blocks whose buddies are *busy*—still in use by the allocating program. Such free blocks cannot (at present) be combined.
   From the above, we see that internal fragmentation could waste up to half of a program's data store. We claim that the advantages of the buddy system mitigate such waste. Moreover, we assume that whoever runs a program can establish a reasonable size for the heap, based on program behavior and input data—this must be true for embedded systems which cannot typically

afford a backing store. The bound for the heap's size could be multiplied by 1.5 to obtain a heap where internal fragmentation need not be a concern.
Alternatively, if the size of data types is known *a priori*, then worst-case analysis can identify a (perhaps better) bound on internal fragmentation. This bound can be calculated by finding the data type that maximizes the amount of internal fragmentation. Clearly, the worst-case run of the program would be one in which all allocations are of this type. Thus, the worst-case bound on internal fragmentation would be the level of internal fragmentation created by allocating the worst-case data type.
External fragmentation remains problematic, because the allocating program can cause the heap to reach a state where sufficient storage exists but is unallocatable due to its position in the heap. In this paper we present an algorithm for defragmenting a buddy heap. The algorithm operates on-demand, and defragments just enough storage to satisfy a given request.

2. Performance can be poor for programs that create objects with very short lifetimes. In fact, a typical assumption of Lisp-like programs is that new objects will die soon. In the limiting case, the state of the free-list could oscillate between Figure 2 and Figure 4; if this is repeated many times, the buddy system suffers from overhead as blocks are joined together only to be split again by the next allocation request.
   We present a variation of the buddy system in this paper. Dubbed the *estranged buddy system*, we delay block recombination [5] until large storage blocks are needed.

For the purposes of this paper, it is important to understand the extent to which the buddy system operates within bounded time. There are essentially three steps to allocate a block of size 2^k.

1. Starting at index k, search upwards for an available block.
2. Recursively bisect the discovered block until a block of size 2^k is obtained.
3. Return the address of that block.

The first two steps can take time proportional to the size of the list display shown in Figure 2. For a heap of size M, the list display is $\theta(M)$. The final step takes constant time. Most programming language systems insist that any storage returned by an allocator be properly initialized, typically to all-zeros. As reported in Section 3, it is expected that most storage requests are for blocks of relatively small size—16 bytes. For a 16 Mbyte heap, assuming 16 is the smallest request, at most 20 slots could be inspected before a suitable block is found. This bound is quite reasonable when compared with the unknown length of an unstructured free-list.

2.5 Estranged Buddy System

As stated in Section 1, if blocks are allocated and immediately deallocated, then the buddy system could oscillate between subdividing blocks and reuniting buddies. Given our assumptions, such behavior causes no asymptotic difficulties.

As a practical concern, particularly for continuous garbage collection, we investigated the extent to which such wasteful behavior can be eliminated and we report on those results in this paper. The basic idea is to avoid unnecessary block recombination. Although delayed recombination has been previously proposed [5], our goal was to obtain an implementation efficient in terms of its use by embedded and real-time Java applications.

2.6 Motivation for Delayed Recombination

We next examine the conditions under which blocks of storage are deallocated.

Occasional collection is the traditional approach for garbage collection. The collector runs on-demand, when storage becomes scarce or in response to the application program requesting a collection cycle. From the deallocator's point of view, objects are returned in relatively large bursts rather than in a continuous stream.

Continuous collection returns objects in anticipation of future demands. Such a collector could run as a low-priority thread, collecting objects continuously as allowed by available CPU resources. Other techniques include contaminated garbage collection [1], which returns objects upon method return.

For occasional collection, programs appear to alternate between allocating and deallocating states. Objects are not returned in a trickle, but seemingly all at once when the collector runs. For such an approach, it may not be possible to deallocate a given block *immediately* after the block is allocated. The deallocation would have to wait until the next collection cycle.

On the other hand, a continuous collector can exhibit the behavior that calls for delayed recombination. For this paper, our experiments used occasional rather than continuous collection. Future research will investigate the results of the estranged buddy system for such collectors.

2.7 Estranged Buddy Allocation

The estranged buddy system is a variation of Knuth's buddy system that delays recombination of free blocks. When a block is deallocated, it is viewed as *estranged* from its buddy and thus reluctant to rejoin. Although this idea was first proposed by Kaufmann [5], no implementation details were provided. Below, we describe our implementation which is biased toward the behavior we expect from Java programs:

- Programs tend to request many blocks of the same size. In Java, equal type implies equal size, except for arrays. Since programs typically instantiate many objects of the same type, a request for a block of size s implies the likelihood of similar requests in the future.
- Programs tend to allocate relatively small blocks. Good object-oriented design prescribes simple objects with relatively few fields. Thus, most objects in Java are small.

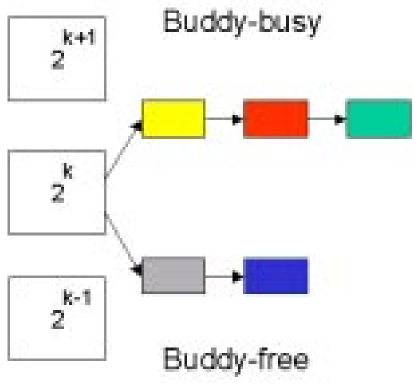


Fig. 5. Estranged buddy list structure

In our implementation of delayed recombination, the estranged buddy system maintains *two* free-lists per size, as shown in Figure 5.

Buddy-busy contains blocks whose buddies are busy. The objects in such blocks are presumably still in use by the allocating program.

Buddy-free contains blocks whose buddies are free. Note that only one of the two free buddies appears on any list. The other buddy's availability is implied by its buddy's presence on the buddy-free list.

The two lists are used so that buddy-busy blocks can be allocated in preference to buddy-free blocks, the latter being saved for recombination should larger blocks be scarce.

Delayed recombination allows increased flexibility in satisfying allocation requests. We implemented the following heuristic to satisfy a request of size 2^k:

1. The buddy-busy list at index k is examined.
2. The buddy-free list at index k is examined.
3. We examine the buddy-free list at index $k-1$, so that two blocks of half the necessary size can be combined. The recombination of such blocks was delayed in the estranged buddy system.
4. We apply the usual buddy algorithm and search above for a large block that can be subdivided.
5. We try to glue from the buddy-free lists of the lowest level up to level k.

Essentially we favor constant-time strategies over searches of the list display. Also, by favoring buddy-busy over buddy-free we tend to preserve opportunities for recombination.

2.8 Defragmenting a Buddy Heap

In this section we examine an algorithm for defragmenting a buddy heap. The algorithm is appropriate for Knuth's buddy system as well as our estranged buddy system. We assume defragmentation becomes necessary when the heap contains sufficient storage to satisfy a request, but such space is the sum of storage "holes" that are not joinable in the buddy sense. We do not penalize the heap for internal fragmentation—we assume all storage requests are expressed as powers of two, and that wasted space within a block is not allocatable.

With our focus on real-time, embedded systems, a defragmentation algorithm must have the following properties:

- The heap cannot be extended. For an embedded system, the heap size is fixed when the product is delivered. Thus, defragmentation must happen in place.
- The defragmentation must occur in bounded time, since the need for defragmentation cannot be anticipated by the allocating program.

In fact, fragmentation can plague any allocator if blocks of storage can be returned *ad hoc*. Most allocators enter a compaction phase, during which storage is massively reorganized. All live objects are pushed to one end of the heap, and all "holes" pushed toward the other end. The holes are then combined into one large block that is suitable for subsequent allocations. For real-time systems, this approach cannot be reasonably bounded. We therefore developed an approach that liberates sufficient storage to satisfy only the request at hand.

Consider Knuth's buddy system in a situation where an allocation request of size 2^k cannot be satisfied yet space is available. The following must be true:

- There is no block available in any list of size $2^l, l \geq k$. Otherwise, such a block could be bisected until a block of size 2^k is obtained.
- There are no free blocks anywhere that can be combined. This follows from the eager recombination of Knuth's buddy system.

Thus, the only space that is available must be below level k. Defragmentation must then consist of relocating objects so as to free buddies that can be combined to obtain a block of size 2^k.

The key observation is shown in Figure 6. If two blocks are on some free-list at level j, then they are necessarily *not* buddies, as described above. However, each must have a buddy that is currently busy. By exchanging one block's busy buddy with the other free block, two joinable blocks result. This approach can be applied recursively down the size display to obtain a block at level k.

Application of the above defragmentation algorithm to our estranged buddy system is straightforward.

3 Experiments

Based on the implementation described above, we present experiments to investigate the following:

1. How does the performance of our bounded-time allocator compare with a standard, unstructured-list allocator? We are interested in worst-case as well as average performance on standard benchmarks.

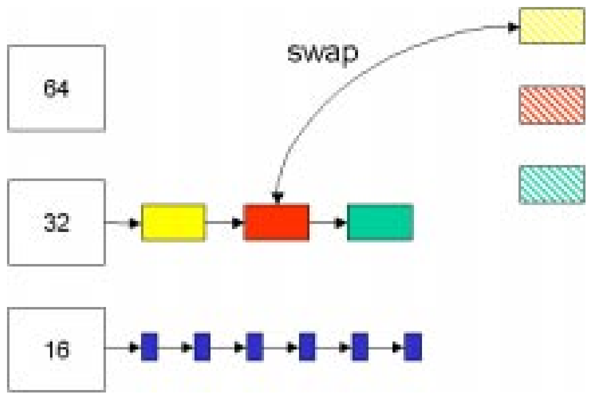


Fig. 6. Defragmentation algorithm

2. How does the performance of our estranged buddy system compare with Knuth's buddy system?
3. What size objects are typically allocated by Java programs?

We implemented our approach in the context of Sun's Java system, JDK 1.1.8. Our changes were confined to those portions of its Java Virtual Machine (JVM) [7] that deal with storage allocation, in particular the `realObjAlloc` method of the `gc` module. Sun's 1.1.8 system offers the following JVM interpreters:

– A reference interpreter is provided, written entirely in C.
– A more efficient interpreter implements the most frequently executed portions in (Sparc) assembly language.

To facilitate our implementation, we based our work on the C version. However, the changes we made are compatible with the architecture of the (speedier) assembly version.

Sun's JVM interpreter manages objects using *handles*. Each handle contains a pointer to the object's current location as well as a reference to an appropriate method table for (virtual) method-lookup. One object can reference another only indirectly through the handles. Thus, if objects are relocated (during garbage collection, for example), only the handle's pointer to the object needs to be updated. To simplify our work, we retained the Sun JVM's use of handles even though our approach avoids relocating objects.

The timings were obtained on a Sparc Ultra 1 running at 167MHz with 128 megabytes of RAM. Our benchmarks consisted of the SPEC benchmarks [3], using their "large" problem size. Figure 7 summarizes the properties of these benchmarks, including the number of objects created and the execution time on the standard JDK 1.1.8 system. The times reported are for the unstructured-list allocator, as shipped with JDK 1.1.8. As shown in Figure 7, the `mpegaudio` and `compress` benchmarks take significant computational time without allocating many objects. We therefore do not report further results for those benchmarks, but concentrate instead on the others, which do allocate a substantial number of objects.

Name	Description	Lines of source	Objects created	Execution Time (sec)
compress	Modified Lempel-Ziv	6,396	10129	
jess	Expert System	570	7,923,782	1802
raytrace	Ray Tracer	3750	6,346,487	2101
db	Database Manager	1020	3,210,520	3766
javac	Java Compiler	9485	5,902,305	1969
mpegaudio	MPEG-3 decompressor	N/A	7,555	8519
mtrt	Ray Tracer, threaded	3750	6,586,584	2223
jack	PCCTS tool	N/A	6,579,042	2336

Fig. 7. SPEC benchmark properties

3.1 Worst-Case Performance of Buddy over Unstructured-List

A buddy system works well for real-time and embedded systems, not especially due to its average performance, but more due to the bounded nature of its worst-case allocation performance. We compared the JDK 1.1.8 (unstructured-list) allocator against the buddy system on the following contrived example:

1. N objects of constant size k are allocated, filling most of the heap.
2. References to every other object are purged, rendering $N/2$ objects dead.
3. A garbage collection cycle is performed. At the conclusion of the cycle the free-list contains $N/2$ blocks of size k, and a larger "remainder" block at the end.
4. An object of larger size is then requested.

The above scenario causes the unstructured-list allocator to search to the end of its free-list to find a block of suitable size. By contrast, the buddy system would have such blocks segregated by size so they can be found quickly. In Figure 8 we show the results for this example: the buddy system is 72 times faster than the unstructured-list allocator. Because that speedup depends on the length of the unstructured list, it is possible to make the speedup arbitrarily high for a contrived example.

While a contrived example may be unfair, it is important to note that a real program *could* misbehave and that the unstructured-list allocator can provide no *reasonable* bound on allocation time.

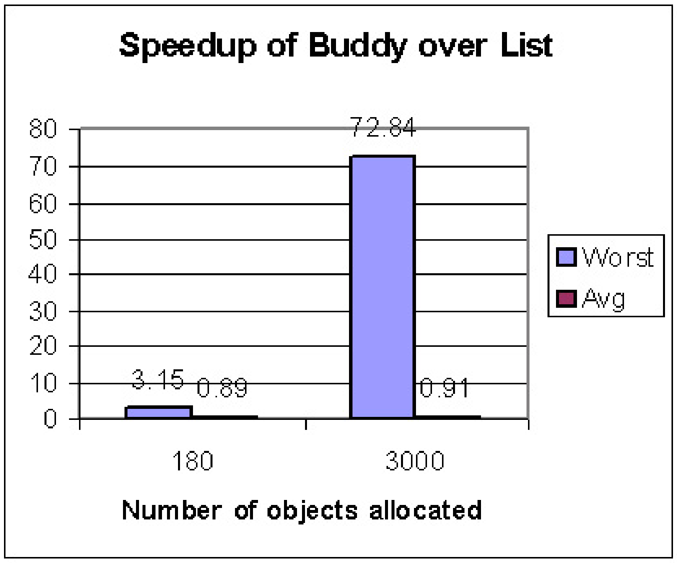


Fig. 8. Speedup of buddy system on a contrived example

Benchmark	Estranged	Knuth
jess	1.04	1.02
raytrace	1.03	0.92
db	1.01	1.01
javac	1.00	1.00
mtrt	1.10	1.02
jack	1.04	1.03

Fig. 9. Speedup of the buddy systems over JDK 1.1.8's unstructured-list allocator

3.2 Average performance of Knuth's Buddy and Estranged Buddy Systems

We next compare the efficiency of the unstructured-list allocator, Knuth's buddy system, and our estranged buddy system on the SPEC benchmarks. We ran the SPEC benchmarks, large size (100) under the following conditions:

- The JDK 1.1.8 system is equipped with a standard, unstructured-list allocator.
- Knuth's buddy system eagerly recombines blocks.
- The estranged buddy system delays recombination as described in Section 2.

Figure 9 shows that Knuth's buddy system operates well on these benchmarks, but sometimes loses performance. On the other hand, the estranged buddy system can be up to 10 percent faster than the (JDK 1.1.8) unstructured-list allocator.

Admittedly, neither approach offers tremendous improvement. However, these results show that the advantages of both systems in terms of worst-case performance do not come with a loss in average performance, especially when recombination is delayed.

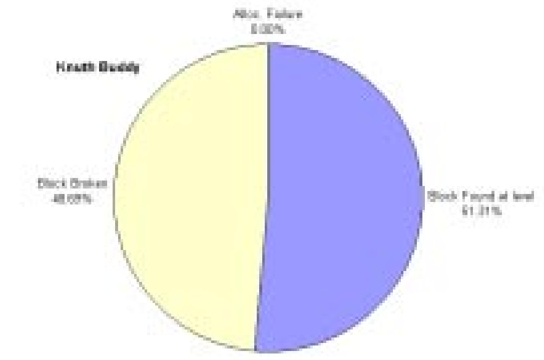
3.3 Qualitative Analysis of Estranged Buddy

We implemented the estranged buddy system described in Section 2. While the execution times reported above are an overall indication of our allocator's performance, we investigated qualitatively *how* requests are satisfied. The shaded portions of Figure 10 show the percentage of allocation requests that were satisfied immediately, by finding a block of size 2^k available in slot k of the size display. Figure 10 shows that by delaying recombination, significantly more blocks can be found without having to break larger blocks or glue smaller blocks. The estranged buddy system finds a block available in the buddy-free or buddy-busy list almost 90% of the time, while the Knuth implementation finds a block immediately only 50% of the time.

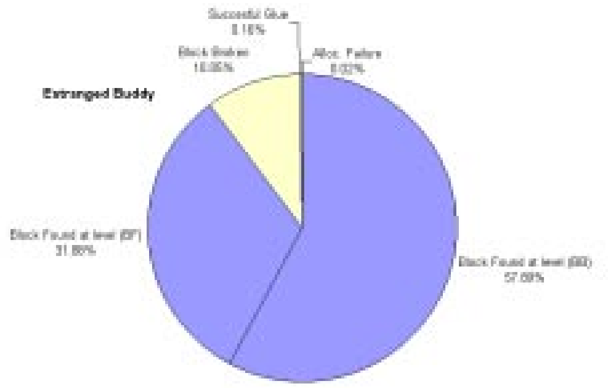
3.4 Object Size

We show in this section that one reason for the estranged buddy system's success is that most allocation requests by Java programs are for blocks of relatively small size. In fact, JDK 1.1.8 cannot allocate fewer than 16 bytes for an object, and a large number of requests are for size-16 blocks. We next examine two of the SPEC benchmarks in detail, showing their distribution of allocation requests.

The `raytrace` application is typical of those SPEC benchmarks that allocate many objects. Figure 11(a) shows the distribution of allocation requests for that `raytrace`. On the other hand, the `compress` benchmark allocates relatively few objects, but Figure 11(b) shows that some of those objects are large, presumably used for holding the data for compression.

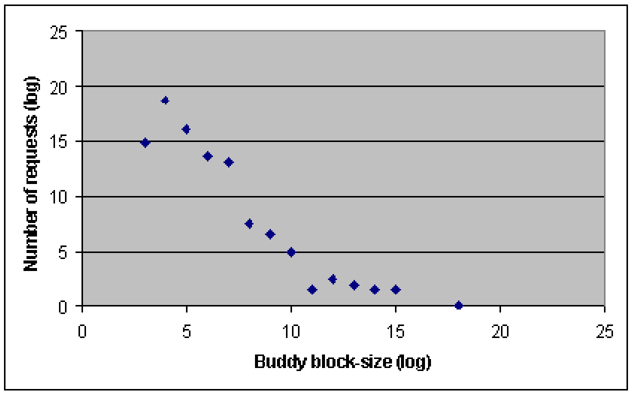


(a)

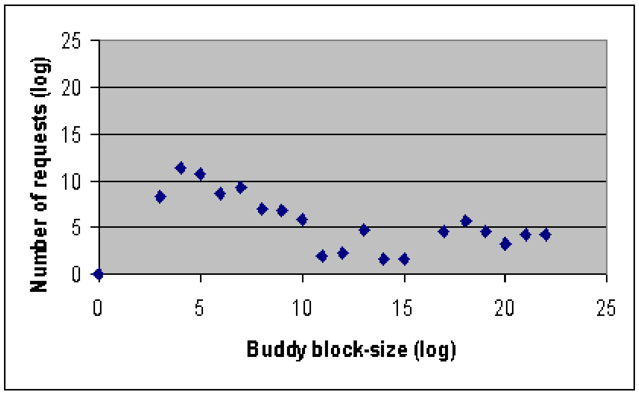


(b)

Fig. 10. Handling of requests by (a) Knuth's buddy system allocator and (b) the estranged buddy system allocator



(a) `raytrace`



(b) `comrpess`

Fig. 11. Allocation requests

4 Conclusions

From the experiments we have conducted, we have shown the following:

1. An allocator hat segregates the free-list by size offers a reasonable bound on the amount of time required for memory allocation and deallocation.
2. Delayed recombination of memory blocks (as in estranged buddy) increases significantly the chances that memory blocks will be available immediately upon an allocation request.
3. The estranged buddy system offers performance gains over Knuth's buddy system and the standard list allocator.

Arguably, the structured-list allocator is bounded, but its bound is $O(M)$—in fact, it is difficult to imagine an allocator whose performance is worst than $O(M)$. For real-time applications, cost analysis of an operation must consider worst-case behavior. A worst-case assumption of $O(M)$ for object instantiations causes gross overprovisioning for most allocations but is a necessarily conservative bound. Asymptotically, both Knuth's buddy system and the estranged buddy system operate in $O(\log M)$ time, where M is the size of the heap. For real-time and embedded systems, this bound should be sufficient to obtain reasonable performance without overly provisioning for allocation times.

For future work, we will investigate the extent to which the time for responding to an allocation request could be effectively *constant*. To obtain such an improvement, the allocator must go beyond segregation by size.

Acknowledgements

We thank Sun Microsystems for access to their Java interpreter. We thank Conrad E. Warmbold of Washington University for his help in preparing figures for this paper.

References

1. Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. *Programming Language Design and Implementation*, 2000.
2. Trishul Chilimbi and James Larus. Using generational garbage collection to implement cache-conscious data placement. *Proceedings of the International Symposium on Memory Management*, 1998.
3. SPEC Corporation. Java spec benchmarks. Technical report, SPEC, 1999. Available by purchase from SPEC.
4. Scott Haug. Automatic storage optimization via garbage collection. Master's thesis, Washington University, 1999.
5. Arie Kaufman. Tailored-list and recombination-delaying buddy systems. *ACM Transactions on Programming Languages and Systems*, 6(1):118–125, January 1984.
6. Donald E. Knuth. *Fundamental Algorithms, Volume 1, The Art of Computer Programming, Second Edition.* Addison-Wesley, 1973.

7. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.
8. Paul R. Wilson. Uniprocessor garbage collection techniques (Long Version). Submitted to ACM Computing Surveys, 1994.
9. Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.