Report Number: WUCSE-2004-58

2004-10-07

Heap Defragmentation in Bounded Time

Sharath R. Cholleti, Delvin Defoe, and Ron K. Cytron

Knuth's buddy system is an attractive algorithm for managing storage allocation, and it can be made to operate in real time. However, the is-sue of defragmentation for heaps that are managed by the buddy system has not been studied. In this paper, we present strong bounds on the amount of storage necessary to avoid defragmentation. We then present an algorithm for defragmenting buddy heaps and present experiments from applying that algorithm to real and syn-thetic benchmarks. Our algorithm is within a factor of two of optimal in terms of the time re-quired to defragment the heap so as to... Read complete abstract on page 2.

Heap Defragmentation in Bounded Time

Sharath R. Cholleti, Delvin Defoe, and Ron K. Cytron

Complete Abstract:

Knuth's buddy system is an attractive algorithm for managing storage allocation, and it can be made to operate in real time. However, the is-sue of defragmentation for heaps that are managed by the buddy system has not been studied. In this paper, we present strong bounds on the amount of storage necessary to avoid defragmentation. We then present an algorithm for defragmenting buddy heaps and present experiments from applying that algorithm to real and syn-thetic benchmarks. Our algorithm is within a factor of two of optimal in terms of the time re-quired to defragment the heap so as to respond to a single allocation request. Our experiments show our algorithm to be much more efficient than extant defragmentation algorithms.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2004-58

Heap Defragmentation in Bounded Time (Extended Abstract)

Authors: Cholleti, Sharath; Defoe, Delvin; Cytron, Ron K.

October 7, 2004

Abstract: Knuth's buddy system is an attractive algorithm for managing storage allocation, and it can be made to operate in real time. However, the issue of defragmentation for heaps that are managed by the buddy system has not been studied. In this paper, we present strong bounds on the amount of storage necessary to avoid defragmentation. We then present an algorithm for defragmenting buddy heaps and present experiments from applying that algorithm to real and synthetic benchmarks.
Our algorithm is
within a factor of two of optimal in terms of the time required to
defragment the heap so as to respond to a single allocation request.
Our experiments show
our algorithm to be much more efficient than extant defragmentation
algorithms.

Heap Defragmentation in Bounded Time*(Extended Abstract)

Sharath R. Cholleti, Delvin Defoe and Ron K. Cytron
Department of Computer Science and Engineering
Washington University
Saint Louis, MO 63130

Abstract

Knuth's buddy system is an attractive algorithm for managing storage allocation, and it can be made to operate in real time. However, the issue of defragmentation for heaps that are managed by the buddy system has not been studied. In this paper, we present strong bounds on the amount of storage necessary to avoid defragmentation. We then present an algorithm for defragmenting buddy heaps and present experiments from applying that algorithm to real and synthetic benchmarks. Our algorithm is within a factor of two of optimal in terms of the time required to defragment the heap so as to respond to a single allocation request. Our experiments show our algorithm to be much more efficient than extant defragmentation algorithms.

1 Introduction

When an application starts, the storage allocator usually obtains a large block of storage, called the *heap*, from the operating system, which the allocator uses to satisfy the application's allocation requests. In real-time or embedded systems the heap size is usually fixed *a priori*, as the application's needs are known. Storage allocators are characterized by how they allocate and keep track of free storage blocks. There are various types of allocators including unstructured lists, segragated lists and buddy allocators [9]. In this paper we study defragmentation of the buddy allocator, whose allocation time is otherwise reasonably bounded [5] and thus suitable for real-time applications.

Over time, the heap becomes *fragmented* so that the allocator might fail to satisfy an allocation request for lack of sufficient, contiguous storage. As a remedy, one can either start with a sufficiently large heap so as to avoid fragmentation problems, or devise an algorithm that can rearrange storage in bounded time to satisfy the allocationrequest. We consider both of those options in this paper, presenting the first tight bounds on the necessary storage and the first algorithm that defragments a buddy heap in reasonably bounded time.

Our paper is organized as follows. Section 1.1 explains the buddy allocator and defragmentation; Section 2 shows how much storage is necessary for an application-agnostic, defragmentation-free buddy allocator; Section 3 gives the worst-case storage relocation necessary for defragmetnation; and Section 4 presents an algorithm that performs within twice the cost of an optimal defragmentation algorithm. Section 5 presents various experimental results of our defragmentation algorithm and compares its efficiency with extant defragmentation approaches.

1.1 Buddy Allocator

In the binary buddy system [7], separate lists are maintained for available blocks of size 2^k bytes, $0 \leq k \leq m$, where 2^m bytes is the heap size.

*Sponsored by DARPA under contract F3333333; contact author cytron@cs.wustl.edu

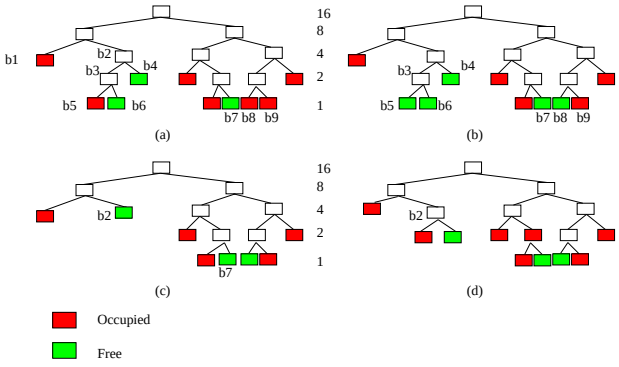


Figure 1: Buddy Example

Initially the entire block of 2^m bytes is available. When a block of 2^k bytes is requested, and if no blocks of that size are available, then a larger block is split into two equal parts repeatedly until a block of 2^k bytes is obtained.

When a block is split into two equal sized blocks, these blocks are called *buddies* (of each other). If these buddies become free at a later time, then they can be *coalesced* into a larger block. The most useful feature of this method is that given the address and size of a block, the address of its buddy can be computed very easily, with just a bit flip. For example, the buddy of the block of size 16 beginning in binary location $xx \ldots x10000$ is $xx \ldots x00000$ (where the x's represent either 0 or 1).

Address-Ordered Buddy Allocator The address-ordered policy selects a block of requested or greater size with the lowest address. If there are no free blocks of the requested size then the allocator searches for a larger free block, starting from the lowest address, and continuing until it finds a sufficiently large block to divide to get a required-size block. For example, if a 1-byte block is requested in Figure 1(c), block b_2 (of 4 bytes) is split to obtain a 1-byte block even though there is a 1-byte block available in the heap. Using this policy the lower addresses of the heap tend to get preference leaving the other end unused unless the heap gets full or fragmented a lot. Our analysis of the binary-buddy heap requirement is based on an allocator that uses this policy, which we call an Address-Ordered Binary Buddy Allocator ($AOBBA$).

Address-Ordered Best-Fit Buddy Allocator In this policy a block of smallest possible size equal to or greater than the required size is selected with preference to the lowest address block. When a block of required size is not available then a block of higher size is selected and split repeatedly until a block of required size is obtained. For example, if a 1-byte block is requested in Figure 1(c), block b_7 is chosen. If a 2-byte block is requested then block b_2 is chosen. Our implementation of the binary-buddy is based on this policy.

1.2 Defragmentation

Defragmentation can be defined as moving already allocated storage blocks to some other address so as to create contiguous free blocks that can be coalesced to form a larger free block. Generally, defragmentation is performed when the allocator cannot satisfy a request for a block. This can be performed by garbage collection [9]. A garbage collector tries to separate the live and deallocated or unused blocks, and by combining the unused blocks, if possible, usually a larger block is formed, which the allocator can use to satisfy the allocation requests. Some programming languages like C and C++ need the program to specify when to deallocate an object, whereas the programming languages, like Java, find which objects are live and which are not by using some garbage collection technique. Generally garbage collection is done by either identifying all the live objects and assuming all the unreachable objects to be dead as in mark and sweep collectors [9] or by tracking the objects when they die as in reference counting [9] and contaminated garbage collection techniques [1].

In our study, we assume we are given both the allocation and deallocation requests. For this study it does not matter how the deallocated

blocks are found – whether it is by explicit deallocation request using `free()` for C programs or by using some garbage collection technique for Java programs. For real-time purposes we get the traces for Java programs using contaminated garbage collection [1], which keeps track objects when they die, instead of mark and sweep collectors which might take unreasonably long time. Our work makes use of the following definitions:

1. *Maxlive*, M, is defined as the maximum number of bytes alice at any instant during the program's execution.

2. *Max-blocksize*, n, is the size of the largest block the program can allocate.

2 Defragmentation-Free Buddy Allocator

We examine the storage requirements for a binary-buddy allocator so that heap defragmentation is *never* necessary to satisfy an allocation request. If the amount of storage a program can use at any instant is known, assuming worst-case fragmentation, then a system with that much storage suffices.

Some previous work on bounding the storage requirement is found in [8]. Even though the bound given in that paper is for a system in which blocks allocated are always a power of 2, the allocator assumed is not a buddy allocator.

Theorem 2.1 *$M(\log M + 1)/2$ bytes of storage are necessary and sufficient for a defragmentation-free buddy allocator, where M is the maxlive and the max-blocksize.*

Proof: See [2] for a detailed proof. The necessary part is proven by constructing a program that requires that number of bytes to avoid defragmentation. The sufficiency part is proven by observing that if each buddy list has M bytes total, then sufficient storage of any size is available without defragmentation. ∎

Theorem 2.2 *The tight bound of $M(\log M + 1)/2$ bytes holds for* any *buddy-style storage mananger (i.e., not just those that are address-ordered).*

Proof: There is insufficient room to present the proof in this paper; see [4]. The proof shows that however an allocator picks the next block to allocate, there is a program that can force the allocator to behave like an address-ordered allocator. ∎

Discussion While a bound of $O(M \log M)$ is usually quite satisfactory for most problems, consider its ramifications for embedded, real-time systems. For a system that has at most M bytes of live storage at any time, the bound implies that a factor of $\log M$ extra storage is required to avoid defragmentation. Even if M is only the Kilobytes, $\log M$ is a factor of 10, meaning that the system needs 10x as much storage as it really uses to avoid defragmentation. Inflating the RAM or other storage requirements of an embedded system by that amount could make the resulting product noncompetitive in terms of cost.

3 Worst case relocation with heap of M bytes

Given that it is unlikely that suffient storage would be deployed to avoid defragmentation, we next attack this problem from the opposite side. We find the amount of storage that has to be relocated in the worst case, if the allocator has M bytes and the maxlive of the program is also exactly M bytes. By defintion, a program can run in its maxlive storage; however, having only that much storage places as much pressure as possible on a defragmentor. The work that must be done is stated in the theorems below, with the proofs found in [2].

Theorem 3.1 *With heap of M bytes and maxlive M bytes, to allocate a s-byte block, where*

$s < M$, $\frac{s}{2} \log s$ *bytes must be relocated, in worst case.*

Theorem 3.2 *With heap of M bytes and maxlive M bytes, to allocate a s-byte block, $s-1$ blocks must be relocated, in worst case.*

Discussion With the smallest possible heap, defragmentation may have to move $O(s \log s)$ bytes by moving $O(s)$ blocks to satisfy an allocation of size s. If an application reasonably bounds its maximum storage request, then the minimum work required to satisfy an allocation is also reasonably bounded in s. Unfortunately, finding the "right" blocks to move in this manner is not easy, and has been shown to be NP-hard [8]. Thus, a heap of just M bytes is not suitable for real-time, embedded applications, and a heap of $M \log M$ bytes, while avoiding defragmentation, is too costly for embedded applications.

4 Greedy Heuristic with 2M Heap

In this section we consider a practical solution to the above problems. We consider a heap slightly bigger than optimal—twice maxlive—and consider a heuristic for defragmentation that is linear in the amount of work required to relocate storage. Here, we use the terms *chunk* and *block* (of storage). A 2^k-byte *chunk* is a contiguous storage in the heap which consists of either free or occupied *blocks* (objects). Our heuristic defragmentation algorithm is based on the following lemma [2]:

Lemma 4.1 *With $2M$-byte heap, when allocation for a block of 2^k bytes is requested, there is a 2^k-byte chunk with less than 2^{k-1} bytes live storage.*

If there is an allocation request for a 2^k-byte block and there is no free block of 2^k bytes then, from the Lemma 4.1, less than 2^{k-1} bytes have to be relocated to create a free 2^k-byte block.

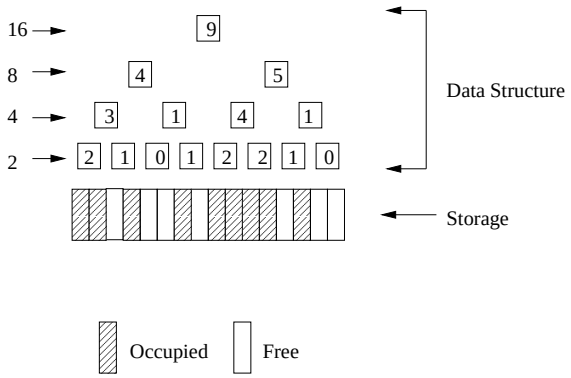


Figure 2: Data Structure

But to relocate these blocks we might have to empty some other blocks by repeated relocations if there are no appropriate free blocks.

How much storage must these recursive relocations move? For example, consider an allocation request for a 256-byte block but there is no such free block. According Lemma 4.1, there is a 256-byte chunk in which less than 128 bytes are live. Assume there is a 64-byte live block that has to be moved out of 256-byte chunk. But now suppose there is no free block of 64 bytes. Now, a 64-byte chunk has to be emptied. Let that contain a block of 16 bytes. This 16-byte block cannot be moved in to either the 256 or the 64-byte chunk, as it is being relocated in the first place to empty those chunks. The result of accounting for the above work is captured by the follwoing [2]:

Theorem 4.2 *With $2M$-byte heap, the greedy approach of selecting a chunk with minimum amount of live storage for relocation, relocates* less than *twice the amount of storage relocated by an optimal strategy.*

Corollary 4.3 *With a $2M$-byte heap, the amount of storage relocated to allocate a s-byte block is less than s bytes.*

Heap Manager Algorithm A naive method for finding the minimally occupied chunk involves processing the entire storage pool, but

the complexity of such a search is unacceptable at $O(M)$. We therefore use the data structure shown in Figure 2 to speed our search. Each level of the tree has a node associated with a chunks of storage that could be allocatable at that level. In each node, an integer shows the number of live bytes available in the subtree rooted at that node. In Figure 2, the heap is 16 bytes. The level labeled with "$2 \rightarrow$" keeps track of the number of live bytes in each $2 - byte$ chunk. Above that, each node at the $4 \rightarrow$ level is the sum of its children, showing how many bytes are free at the 4-byte level, and so on.

When a 2-byte block is allocated, the appropriate node at the $2 \rightarrow$ level is updated, but that information must propagate up the tree using parent-pointers, taking $O(\log M)$ time. To find a minimally occupied block of the required size, only that particular level is searched in the data structure, decreasing the time complexity of the search to $O(M/s)$, where s is the requested block size (it has to go through $2M/s$ numbers to find a minimum). We have used hardware to reduce this kind of search to near-constant time [5] and we expect a similar result could obtain here.

The algorithm's details are straightforward and appear in [4], wherein proofs can be found of the following theorems:

Theorem 4.4 *With $2M$-byte heap, defragmentation according to the Heap Manager Algorithm(Section 4) to satisfy a single allocation request of a s-byte block takes $O(Ms^{0.695})$ time.*

Theorem 4.5 *With M-byte heap, defragmentation according to the Heap Manager Algorithm(Section 4) to satisfy an allocation request for a s-byte block takes $O(Ms)$ time.*

Discussion A constant increase in heap size (from M to $2M$) allows the heuristic to operate polynomially, but its complexity may be unsuitable for real-time systems. The work that must be done is reasonably bounded, in terms of the allocation-request size s, but more research is needed to find heuristics that operate in similar time.

5 Experimental Results

Storage requirements as well allocation and deallocation patterns vary from one program to another. Hence, storage fragmentation and the need for defragmentation vary as well. To facilitate experimentation, we implemented a simulator that takes the allocation and deallocation information from a program trace and simulates the effects of allocation, deallocation, and defragmentation using buddy algorithm and our heuristic described in Section 4.

For benchmarks, we used the Java SPEC benchmarks [3]. To illustrate the efficiency of our defragmentation algorithm, we compare our results with the following approaches currently in common practice.

Left-First Compaction This approach compacts all the storage to the lower end, based on address, by filling up the holes with the closest block (to the right) of less than or equal size.

Right-First Compaction This is similar to left-first compaction in moving all the storage to the left end of the heap, but it picks the blocks by scanning from the *right end*, where we expect storage is less congested due to our address-ordered policy for allocation.

Compaction Without Using Buddy Properties This method has been implemented to compare our defragmentation algorithm to a naive compaction method of sliding the blocks to one end of the heap, without following the buddy block properties similar to the general non-buddy allocators.

5.1 Defragmentation with $2M$-byte Heap

We explored the defragmentation in various benchmarks with $2M$-byte heap using the algorithm described in Section 4, which is based on the theorem 4.2. To our surprise we found that

none of the benchmarks needed any defragmentation when the heap size is $2M$ bytes! So having twice the maxlive storage, the address-ordered best-fit buddy allocator is able to avoid any relocation of the storage. Even some randomly generated program traces did not need any relocation with a heap of size $2M$ bytes. These results actually confirm what is known in practice: most allocation algorithms do not need defragmentation [6].

However, our theoretical results show that there are some programs for which defragmentation will be problematic, and real-time systems must be concerned with accomodating worst-case behavior.

5.2 Defragmentation with M-byte Heap

A $2M$-byte heap induced no defragmentation for our benchmarks, so we next experiment wtih an exact-sized heap of M bytes. Note that with an M-byte heap, there is no guarantee about how much storage is relocated when compared to the optimal. From Figure 3 we see that very few programs required defragmentation. Among the programs that needed defragmentation, except for Jess of size 1 and Javac of size 10, the amount of storage relocated by our defragmentation algorithm for other programs is very insignificant. But compared to our algorithm which relocates storage only to satisfy a particular request without defragmenting the whole heap, all other compaction methods perform badly. The amount of storage relocated by our defragmentation algorithm is summed over all the relocations necessary, whereas for the compaction methods the amount is only for one compaction which is done when the allocator fails to satisfy an allocation request for the first time. Among all the compaction methods, only right-first compaction performed reasonably well. The other two methods—left-first compaction and naive compaction—relocated significantly more storage, sometimes close to M bytes.

The above results, which showed the weakness of the compaction methods when compared to our defragmentation algorithm, indicate the value of localized defragmentation to satisfy a single allocation request instead of defragmenting the whole heap. If the defragmentation is needed for very few allocations (according to the amount of storage relocated as shown in Figure 3, and given the number of relocations as shown in Figure 5), there is no point in doing extra work by compacting the whole heap, either in anticipation of satisfying the future allocation requests without any defragmentation or for some other reason.

5.3 Minimally Occupied vs Random Block Selection

Our heuristic selects the minimally occupied block for relocation. In this section we compare that strategy against what happens when the relocated block is chose at random and the heap is sufficiently small so as to cause defragmentation (M bytes). We report results only for those benchmarks that needed defragmentation under those conditions.

From Figure 4 we see that out of the 6 Java SPEC benchmark programs which needed some defragmentation, for 3 programs (check(1), db(1) and jess(100)) the same amount of relocation is required. For 2 programs (jess(1) and jess(10)), selecting the minimally occupied block is better and for 1 program (javac(10)), selecting a random block is better.

From Figure 5, 2 Java SPEC benchmark programs (check(1) and db(1)) needed the same number of relocations while 4 others (jess(1), javac(10), jess(10) and jess(100)) needed a different number of relocations. For all those programs, selecting the minimally occupied block is better. Note that even though random selection needed more relocations for javac(10), the amount of storage relocated is less.

The above results indicate that using the random block selection might be a good alternative, and it avoids searching for the minimaly occupied block. We have not as yet determined the

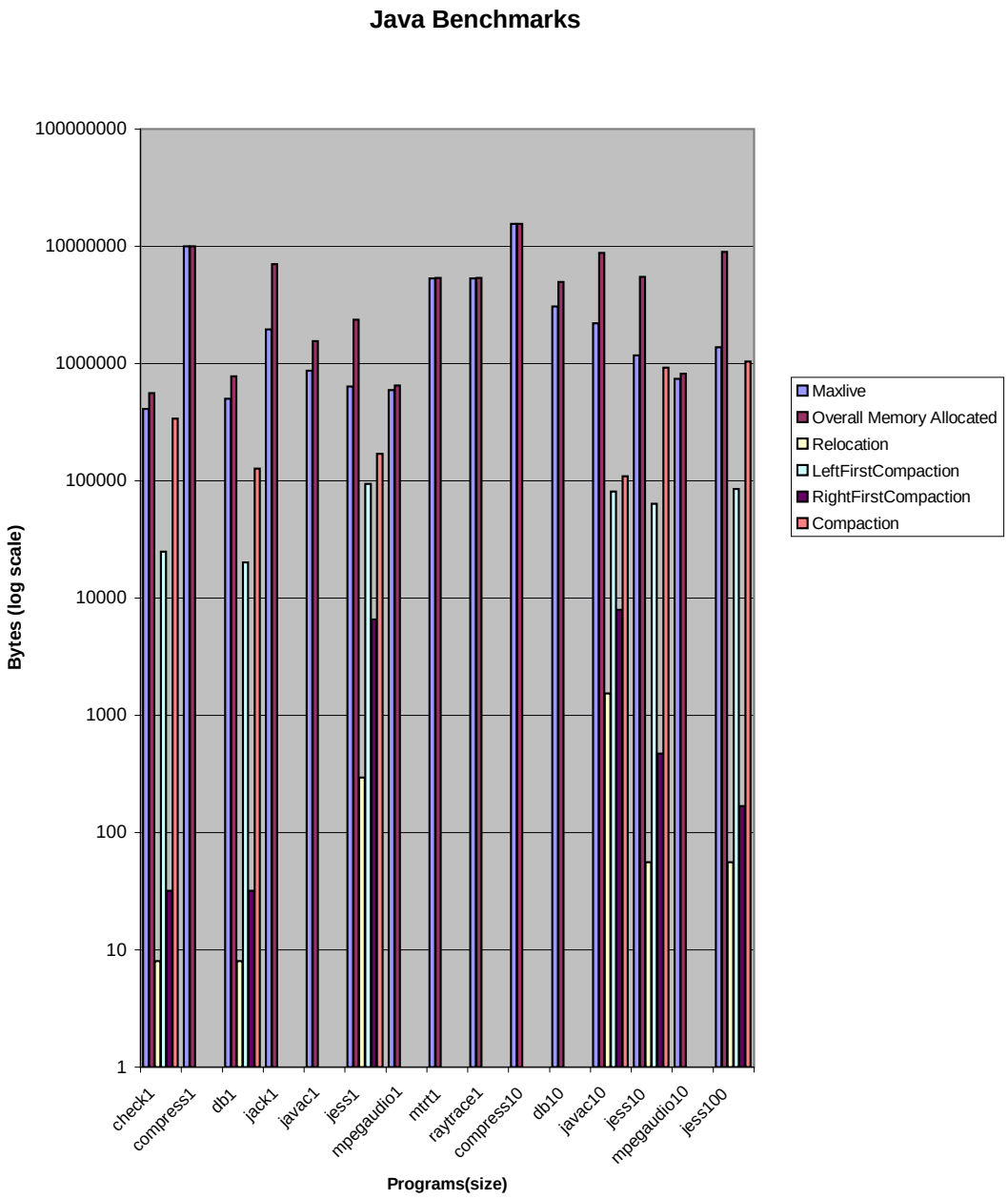
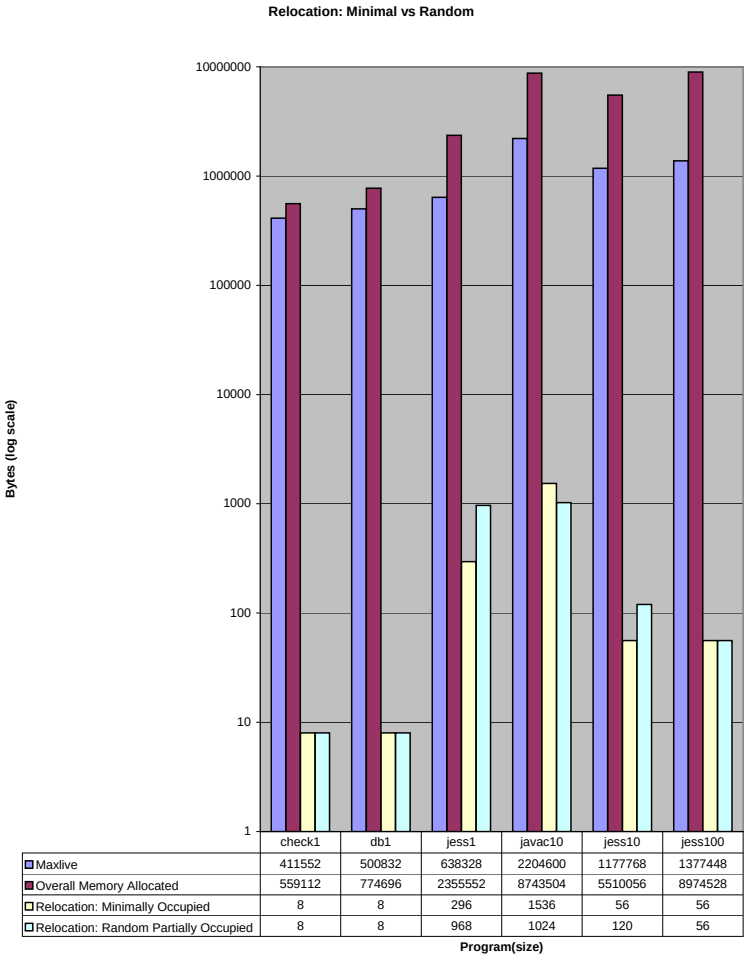


Figure 3: Java Test Programs – Amount of Relocation



	check1	db1	jess1	javac10	jess10	jess100
Maxlive	411552	500832	638328	2204600	1177768	1377448
Overall Memory Allocated	559112	774696	2355552	8743504	5510056	8974528
Relocation: Minimally Occupied	8	8	296	1536	56	56
Relocation: Random Partially Occupied	8	8	968	1024	120	56

Figure 4: Minimally Occupied vs Random – Amount of Relocation

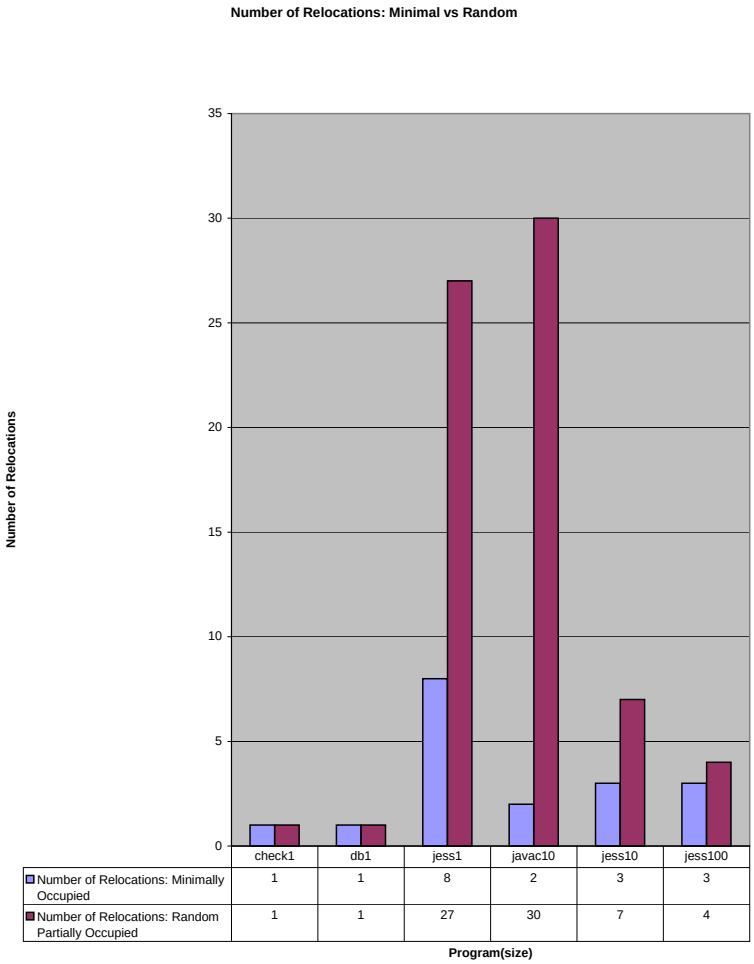


Figure 5: Minimally Occupied vs Random – Number of Relocations

theoretical time bound when random blocks are chosen for relocation.

6 Conclusions and Future Work

We presented a tight bound for the storage required for a defragmentation-free buddy allocator. While the bound could be appropriate for desktop computers, for embedded systems there is an inflationary factor of $\log M$ for the storage required to avoid defragmentation. While a factor of 2 might be tolerable, a factor of $\log M$ is most likely prohibative for embedded applications.

We presnted a greedy algorithm that allocates an s-byte block using less than twice the optimal relocation, on a $2M$-byte (twice maxlive) heap. Even though our benchmark programs did not need any relocation with $2M$, there is some relocation involved when heap size is M. The results show better performance of a localized defragmentation algorithm, which defragments just a small portion of the heap to satisfy a single request, over the conventional compaction algorithms.

We compared our greedy algorithm with random selection heuristic to find that our greedy algorithm performed better, as expected. But since the overall fragmentation is low, the random heuristic takes less time and might work well in practice; a theoretical bound on its time complexity is needed.

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

References

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

[2] Sharath Reddy Cholleti. Storage allocation in bounded time. Master's thesis, Washington University, 2002.

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

[4] Delvin Defoe, Sharath Cholleti, and Ron K. Cytron. Heap defragmentation in bounded time. Technical report, Washington University, 2004.

[5] Steven M. Donahue, Matthew P. Hampton, Morgan Deters, Jonathan M. Nye, Ron K. Cytron, and Krishna M. Kavi. Storage allocation for real-time, embedded systems. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software: Proceedings of the First International Workshop*, pages 131–147. Springer Verlag, 2001.

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

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

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

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