A New Implementation Technique for Memory Management

Mehran Rezaei and Krishna M. Kavi[1]
The University of Alabama in Huntsville

Abstract

Dynamic memory management is an important and essential part of computer systems design. Efficient memory allocation, garbage collection and compaction are becoming increasingly more critical in parallel, distributed and real-time applications using object-oriented languages like C++ and Java. In this paper we present a technique that uses a Binary Tree for the list of available memory blocks and show how this method can manage memory more efficiently and facilitate easy implementation of well known garbage collection techniques.

Keywords: Dynamic Memory Management, Best Fit, First Fit, Buddy System, Memory Fragmentation, Garbage Collection, Generation Scavenging.

1. Introduction

The need for more efficient memory management is currently being driven by the popularity of object-oriented languages in general ([Chang 99], [Calder 95]), and Java in particular [Abdullahi 98]. The trend towards the use of Java in Internet, real-time and embedded systems requires predictable and/or reasonable execution times for memory allocation and garbage collection. Our research originally was motivated by the recent trends in the design of computer systems and memory units, leading to Intelligent RAM's (IRAM) or Processing In Memory (PIM) devices. We are studying the migration of memory management, including garbage collection to the intelligent memory devices. We are investigating algorithms and techniques that are not only efficient in execution time, but require less complex hardware logic to be included in the IRAM devices. During this research we studied the use of a Binary Tree for tracking AVAILable (chunks) blocks of memory. The tree is organized using the starting address of the memory blocks. In this paper we describe this approach for memory management and show how well known memory allocation and garbage collection techniques can be implemented more naturally within the context of the Binary Tree algorithm.

The remainder of the paper is organized as follow: Section 2 describes memory management policies, and related work. Section 3 presents our new implementation (Binary Tree) of Best Fit and First Fit techniques. Section 4 shows the results and compares Binary Tree implementation with Linear Linked list implementation of Best Fit and First Fit, in terms of execution performance. Section 5 addresses Garbage Collection using Binary Tree algorithm.

2. Background and Related Research

Dynamic memory allocation is a classic problem in computer systems. Typically we start with a large block of memory (sometimes called a heap). When a user process needs memory, the request is granted by carving a piece out of the large block of memory. The user process may free some of the allocated memory explicitly, or the system will reclaim the memory when the process terminates. At any time the large memory block is split into smaller blocks (or chunks), some of which are allocated to a process (live memory), some are freed (available for future allocations), and some are no-longer used by the process but are not available for allocation (garbage). A memory management system must keep track of these three types of memory blocks and attempt to efficiently satisfy as many of the process's requests for memory as possible.

Memory allocation schemes can be classified into Sequential Fit, Buddy System and Segregated Free List algorithms. The Sequential Fit approach (including First Fit, Best Fit) keeps track of available chunks of memory on a list. Known sequential techniques differ in how they track the memory blocks and how they allocate memory requests from the free blocks. Normally the chunks of memory (or at least the free chunks) are maintained as a Linear Linked list. When a process releases memory, these chunks are added to the free list, either at the end or in place if the list is sorted by addresses (Address Order [Wilson 95]); freed chunk may be **coalesced**

[1] Please direct all correspondence to Krishna M. Kavi, Dept. of Electrical and Computer Engineering, The University of Alabama in Huntsville, Huntsville, AL 35899; email: kavi@ece.uah.edu.
This research is supported in part by NSF grants: CCR 9796310, EIA 9729889, EIA 9820147.

with adjoining chunks to form larger chunks of free memory. When an allocation request arrives, the free list is searched until an appropriately sized chunk is found. The memory is allocated either by granting the entire chunk or by **splitting** the chunk (if the chunk is larger than the requested size). Best Fit methods [Knuth 73] try to find the smallest chunk that is at least as large as the request. First Fit [Knuth 73] methods will find the first chunk that is at least as large as the request. Best Fit method may involve delays in allocation while First Fit method may lead to more external fragmentation [Johnstone 98].

If the free list is in Address Order [Wilson95], newly freed chunks may be combined with its surrounding blocks, leading to larger chunks. However, this requires a "linear" search through the free list when inserting a newly freed block of memory (or when searching for a suitable chunk of memory).

In Buddy System ([Knowlton 65], [Knuth 73]), the size of any memory chunk (live, free or garbage) is 2^k for some k. Two chunks of the same size that are next to each other in terms of their memory addresses, are known as buddies. If a newly freed chunk finds its buddy among the free chunks, the two buddies can be combined into a larger chunk of size 2^{k+1}. During allocation, larger chunks are split into equal sized buddies, until a chunk that is at least as large as the request is created. Large internal fragmentation is the main disadvantage of this technique. It has been shown that as much as 25% of memory is wasted due to fragmentation in buddy systems [Johnstone 98]. An alternate implementation, Double Buddy ([Johnstone 98], [Wise78]), which create buddies of equal size, but does not require the sizes to be 2^k, is shown to reduce the fragmentation by half.

The Segregated list approach maintains multiple linked lists, one for each different sized chunk of available lists. Returning a free chunk from one of the lists satisfies allocation requests (by selecting a list containing chunks, which are at least as large as the request). Freeing memory, likewise, will simply add the chunk to the appropriate list. No coalescing or splitting is performed and the size of chunks remains unaltered. The main advantage of Segregated lists is the execution efficiency in allocating and freeing memory chunks. The disadvantage is the inefficient usage of memory. The memory is divided into regions based on the different sized blocks. Since the number and frequency of requests for different sized chunks depends on the application, improper division of memory into regions can lead to poor memory utilization, and even inability to satisfy all requests from the application. Some recent methods fine-tune the Segregated list methods using execution profiles of programs [Chang 99].

3. Binary Trees For AVAILable Lists

3.1. Overview Of The Algorithm

In our approach to memory management, we maintain the free chunks of memory (i.e., AVAILable chunks) on a Binary Tree, using the starting address of the chunk. In addition to maintaining the size of the chunk, each node in the Binary Tree also keeps track of the size of the largest chunk of memory available in its left and right sub-trees. This information can be used during allocation to minimize the search time. This information can also be used for an efficient implementation of the Best Fit techniques (more precisely, Better Fits). By creating a separate tree for different sized chunks, our approach can be adapted to segregated lists techniques. However, in this paper we will assume sequential list approaches only.

While inserting a newly freed chunk of memory, our algorithm checks to see if the new chunk of memory can be combined (or coalesced) with existing nodes in the tree to create larger chunks. Inserting a new free chunk will require searching the Binary Tree with a complexity of $O(l)$ where l is the number of levels in the Binary Tree, and $\log_2(n) \leq l \leq n$, where n is the number of nodes in the Binary Tree. It is possible that the Binary Tree de-generates into a linear list (ordered by the address of the chunk) leading to a linear O(n) insertion complexity. We advocate periodic balancing of the tree to minimize the insertion complexity. As a heuristic we can assume that $\log_2(n) \leq l \leq 2 * \log_2(n)$. The re-balancing can be aided by maintaining the total number of nodes in the left and right sub-trees with each node. These values can be used to identify the root for the balanced tree. *Note, however, the coalescing of chunks described above already helps in keeping the tree from being unbalanced.* In other words, the number of times a tree must be balanced, although depends on a specific application, will be relatively small in our approach. We will not rebalance trees in this paper.

3.2. Algorithm for Insertion Of A Newly Freed Memory Chunks

The following algorithm shows how a newly freed chunk can be added to the Binary tree of AVAILable chunks. The data structure for each node representing a free chunk of memory contains the starting address of the chunk, the size of the chunk, a pointers to its left and right child nodes, the size of the largest chunk of memory in its left and right sub-trees.

```
Structure used as tree's nodes:
Struct node {
    Int          Start;          // Starting Address Of the Chunk
    Int          Size;           // Size of the Chunk
    Int          Max_Left;       // Size of largest chunk in the left sub-trees
    Int          Max_Right;      // Size of largest chunk in the right sub-trees
    Struct node  *Left_Child;    // Pointer to Left  Subtree
    Struct node  *Right_Child;   // pointer to Right Subtree
    Struct node  *Parent;        // pointer to Parent – Need to adjust Max_Left and Max_Righ
};
Algorithms and Functions:
Void INSERT(int Chunk_Start, int Chunk_Size, node *Root) {
    // Chunk_Start and Chunk_Size are associated with the newly freed block of memory
    // Check if the new chunk can be coalesced with the Root node
    If ((Root→Start + Root→Size == Chunk_Start) || (Chunk_Start + Chunk_Size == Root→Start))
        COALESCE (Chunk_Start, Chunk_Size, Root);
    Else {
    If (Chunk_Start < Root->Start)
        If (Root→Left_Child == NULL)                          // Create a new left child
            Root→Left_Child = CREATE_NEW(Chunk_Start,Chunk_Size);
        Else
            INSERT (Chunk_Start, Chunk_Size, Root→Left_Child);
        Else {
            If (Root→Right_Child == NULL)                     // Create a new Right Child
                Root→Right_Child = CREATE_NEW(Chunk_Start, Chunk_Size);
            Else
                INSERT(Chunk_Start, Chunk_Size, Root→Right_Child);
        }
    }
    ADJUST_SIZES (Root);
}
Void ADJUST_SIZES ( node *Root) {
    // This function traverses up the tree from the current node, adjusting the sizes of the largest
    // chunks on the left and right sub-trees
}
Void COALESCE (int Chunk_Start, int Chunk_Size, node  *Root) {
    If (Root->Start + Root ->Size = Chunk_Start){
        Root→Size = Root→Size + Chunk_Size;
        Check-If-Right-Child-Can-Be-Coalesced;                      // Case I.1
        Check-If-Leftmost-Child-Of-Right-Child-Can-Be-Coalesced;    // Case I.2
    }
    Else If (Chunk_Start + Chunk_Size = Root→Start) {
        Root->Start = Chunk_Start;
        Root->Size = Root->Size + Chunk_Size;
        Check-If-Left-Child-Can-Be-Coalesced;                       // Case II.1
        Check-If-Rightmost-Child-Of-Left-Child-Can-Be-Coalesced;    // Case II.2
    }
}
```

3.3. Complexity Analysis

Algorithm INSERT is very similar to Binary Tree traversal and the execution time complexity depends on the number of levels l in the tree. In a balanced tree, $l \leq \log_2(n)$ where n is the number of nodes in the tree. The function COALESCE, as can be seen, checks at most 4 other nodes that are candidates for further coalescing. Note that only one of the 4 cases will lead to a coalescing. Case I.2 and Case II.2 may require a traversal down the tree – from the current node to the leaves. This computation has a $O(l)$ execution time complexity. The additional step involved in inserting a newly freed chunk is in adjusting the sizes of the largest chunks in the left and right sub-trees as indicated by the function ADJUST_SIZES. This requires a traversal up the tree to the root of the tree. Once again the complexity is $O(l)$. Thus, the overall complexity of the INSERT algorithm is $O(l)$.

Allocation requests are satisfied using traditional Binary Search algorithms with a complexity of $O(l)$. In our case, the search is speeded-up by the Max_Left and Max_Right that guide the search to the appropriate sub-tree.

3.4. Restructuring The Tree.

When a node in the AVAILable tree is deleted either due to an allocation or due to coalescing, the tree must be adjusted. This process requires changes to at most two pointers. When a node is deleted we consider the following cases.

a. *If the deleted node has a right child, the right child will replace the deleted node. The left child of the deleted node (if one exists) becomes the left child of the new node. Otherwise, if the replacing node (the right child of the deleted node) has a left sub-tree, we traverse the left sub-tree and the leftmost child of the replacing node becomes the parent of the left child of the deleted node.*
b. *If the deleted node has no right child then the left child replaces the deleted node.*

Since the Case "a" above may involve traversing down the tree (from the right child of the deleted node), the worst case complexity is $O(l)$.

4. Empirical Results

In order to evaluate the benefits of our approach to memory management, we developed simulators that accept requests for memory allocation and de-allocation. We studied 3 different implementations for tracking memory chunks: Binary tree based on starting addresses (our method), Linear Linked lists (more traditional method), Binary Buddy system. For each we studied two memory allocation policies: Best-Fit and First-Fit. In all implementations, we included coalescing of freed objects whenever possible. We do not show the results for Buddy system since its performance is very poor in terms of fragmentation.

4.1. The Selected Test Benchmarks

For our tests, we used Java Spec98 benchmarks [Javaspec 98], because Java programs are allocation intensive. Applications with large amount of live data are worthy benchmarks for memory management techniques, because such benchmarks stress the memory overhead incurred by the management algorithms. The programs were instrumented using ATOM [Eustance 94] on Digital Unix, to collect traces indicating memory allocation and de-allocation requests. The general statistics of the benchmarks used are shown in Table 1.

Table 1: Benchmark Statistics

Benchmark	Total # Allocation Requests	Total # Deallocaton Requests	Total Bytes of Memory Requested	Average Size Of Request Bytes	Average Size Of Live Memory/Bytes
Simple	12,110	8,069	4,555,549	376	699,443
Check	46,666	41,009	4,472,192	96	860,498
Jess	81,858	74,309	7,615,801	93	1,350,750
Jack	80,815	73,863	7,173,707	89	1,173,134
MPEG	97,431	91,104	7,546,025	77	1,292,207
Average	63,776	57,671	6,272,655	146	1,075,206

The first benchmark (Simple) is not a Java Spec98 program but a collection of simple programs (including Fibonacci) written by UAH students. Check is a collection of programs that exercise the error checking capabilities of JVM on a machine. Jess is an expert system shell based on NASA's CLIPS system. Jack is a Java parser generator, and MPEG is an MPEG Layer-3 audio program. We ran into difficulties in running other Java Spec98 programs (including the Javac and db) due to an older JVM translator on DEC Unix available to us.

4.2. Execution Performance

In this subsection we compare the execution performance of our algorithm with Linear Linked list techniques using both First Fit and Best Fit approaches. For this comparison we collected statistical data on the following items:

Average Number of free-chunks. This will measure the memory overhead of a memory management algorithm. The memory needed for each node in our Binary Tree method must include: a starting address of the chunk, 3 pointers (left, right and parent) and 3 sizes (size of the chunk, maximum sized chunks in the left and right sub-trees) for a total of 28 bytes. In traditional Linked list methods, we need a starting address, 2 pointers (next and previous) and one size (size of the node) for a total of 16 bytes. The average number of nodes also measures the execution time needed for searching the Binary Tree (and Linear Linked list).

Maximum Number of free-chunks. This measures worst case memory overhead and execution times for searching the tree (as well as the Linear Linked list).

Average number of nodes searched for allocation and free. These measure the execution complexity while searching the tree for allocation and for inserting a freed node.

Maximum number of nodes searched for allocation and free. These measure the worst case execution time for allocation and de-allocation.

Frequency of Coalescing. This measures how often a newly freed chunk of memory can be combined with other free nodes. As pointed out in [Johnstone 98] "less fragmentation results if a policy (and an implementation) immediately coalesces freed memory". Moreover, coalescing also helps in reducing the number of nodes in the AVAILable list.

The first set of data in Table-2 compares the Binary Tree based implementation of AVAILable lists with that using Linear Linked lists.

Table 2: Exact Allocation

Allocated Size – Requested size = 0

		Binary Tree Implementaton						Linked List Implementation			
Bench-Mark Name	Policy	Avg. No. Nodes	Coale-scence Frq.	Nodes Searched At Allocation		Nodes Searched At De-Allocation		Average No. Nodes	Coale-scense Frq.	Nodes Searched At Allocation	
				Avg.	Max	Avg	Max			Avg.	Max
Simple	BF	59.42	0.86	6.89	35	9.67	34	436.55	0.33	368.6	1005
	FF	119.4	0.95	1	1	74.46	210	309.36	0.85	1	1
Check	BF	127.48	0.88	28.23	84	41.72	195	1689.18	0.29	1392.41	3175
	FF	557.35	0.97	1	1	417.18	840	2635.72	0.87	1	1
Jess	BF	218.02	0.85	71.2	155	87.82	252	2892.26	0.17	2365.24	4916
	FF	1106.97	0.97	1	1	831.23	1874	4852.51	0.86	1	1
Jack	BF	171.27	0.89	39.43	95	49.53	169	3236.11	0.3	2611.11	4938
	FF	796.5	0.98	1	1	622.72	1455	4249.59	0.87	1	1
MPEG	BF	154.7	0.92	30.27	88	46.02	490	3060.86	0.5	2475.3	4896
	FF	691.85	0.98	1	1	520.77	1155	3746.86	0.91	1	1
Avg	BF	146.18	0.88	35.20	91.40	46.95	228.00	2262.99	0.32	1842.53	3786.00
	FF	654.414	0.97	1.00	1.00	493.27	1106.80	3158.81	0.87	1.00	1.00

In this run, we allocated exactly the number of bytes requested. This policy can lead to many very small chunks of memory, cluttering the AVAILable lists and causing longer search delays. We show the data for both First Fit and Best Fit allocation policies. This table shows that the Binary Tree implementation consistently outperforms Linear Linked list implementation in every aspect: average number of nodes in AVAILable list, Maximum number of nodes, Frequency of a freed-node being coalesced, Average (and Maximum) number of nodes searched to satisfy an allocation request. It is also interesting to observe that Best Fit (or Better Fit) can be more efficiently implemented using the Binary Tree, although the allocation performance is adversely impacted (same is true for Linear Linked list implementation also). The number of nodes and the frequency of coalescence for Linear Linked lists using First Fit policy compare very poorly with the Binary Tree implementation. The Best Fit policy using Linear Linked list require 15.5 times as many nodes as the Binary Tree approach (averaged across all benchmarks)-- 2263 nodes vs. 146 in Binary Tree. The high frequency of coalescence is primarily due to the birth and death behavior of Java objects: "objects allocated together tend to die together". The table also shows that the Binary Tree implementation has fewer nodes (on AVAILable list) leading to smaller memory overhead for the tree. Since each node in Binary Tree requires 28 bytes per node, the average bytes used are 28*146 =4088 (for Best Fit) as compared to 16*2263 =36208 in Linear Linked list.

On average the Binary Tree searches 35 nodes (in Best-Fit policy) to find a suitable node and 47 nodes during de-allocation; while the Linear Linked list searches 1843 nodes on allocation. This is a significant performance enhancement. The differences are even more pronounced for First Fit policies. *In the rest of the paper, we will compare only Best Fit policies, since our goal is to show that the Binary Tree method can implement Best Fit policy very efficiently.* Binary Tree implementation outperforms Address Ordered and Linked Lists even better for First Fit policy.

In the next set of experiments, we avoided keeping very small chunks of memory, by allocating more than the requested number of bytes. In Table 3, we will not keep chunks that are smaller than 16 bytes and in Table 4 we do not keep chunks smaller than 32 bytes. The data shows a significant improvement in performance (for both Binary Tree and Linear Linked list implementations), although the improvements are more dramatic with Binary Tree implementations.

Table 3. Overallocation-16

Allocated size – Requested Size <= 16

	Binary Tree Implementaton							Address-Ordered List Implementation				
Bench-Mark Name	Avg. No. Nodes	Avg. Intern Frag-ment	Coale-scence Frq	Nodes Searched At Allocation		Nodes Searched At De-allocation		Avg. No. Nodes	Avg. Intern Frag-ment	Coale-scense Frq	Nodes Searched At Allocation	
				Avg.	Max	Avg.	Max				Avg.	Max
Simple	45.01	0.54	0.79	3.45	28	5.5	27	89.08	1.1	0.39	67.19	397
Check	29.13	0.97	0.79	4.36	22	7.44	132	231.29	1.97	0.33	167.89	661
Jess	39.72	1.34	0.72	5.08	36	8.04	135	170.04	2.10	0.35	140.41	634
Jack	46.22	0.95	0.80	5.03	38	8.91	129	267.90	1.96	0.33	236.68	998
MPEG	60.00	0.83	0.83	5.27	28	9.95	453	315.16	1.85	0.40	240.49	986
Avg.	44.02	0.93	0.79	4.64	30.4	7.97	175	214.69	1.80	0.36	170.53	735.2

Table 4. Overallocation – 32
Allocated Size – Requested Size <= 32

	Binary Tree Implementaton							Linear List Implementation				
Bench-Mark Name	Avg. No. Nodes	Avg. Intern Frag-ment	Coale-scence Frq	Nodes Searched At Allocation		Nodes Searched At De-allocation		Avg. No. Nodes	Avg Intern Frag-ment	Coale-scense Frq.	Nodes Searched At Allocation	
				Avg.	Max	Avg	Max				Avg.	Max
Simple	37.09	2.20	0.75	2.89	30	6.08	44	74.6	3.87	0.44	60.42	336
Check	24.94	2.92	0.78	3.62	25	7.01	128	117.35	5.72	0.43	77.71	494
Jess	26.60	3.70	0.70	3.92	29	7.47	128	78.50	6.83	0.33	75.42	478
Jack	53.26	2.99	0.79	4.87	42	10.92	128	158.37	6.10	0.44	133.50	905
MPEG	45.81	2.38	0.82	4.10	31	9.00	451	213.46	5.30	0.50	144.34	864
Avg	37.54	2.84	0.77	3.88	31.40	8.10	175.80	128.46	5.56	0.43	98.28	615.40

In Table 3, the average number of nodes searched at allocation (and de-allocation) is substantially smaller, while the actual amount of fragmentation due to "over-allocation" is very small: less than one byte. In the Binary Tree implementation, the average number of nodes searched at allocation, over all programs is only 4.64 (as compared to 35.20 in Table 2), and the number of nodes searched on de-allocation drops to 8 (from 47). Linear Linked list method also shows improvements in searching: dropping to 171 from 1843. The coalescence frequency dropped slightly for all programs and implementations. The memory overhead in Binary Tree averages to 1232 bytes as compared to 3435 bytes in Linear Linked list implementation

From Table 4 it can be seen that the improvement over Table 3 is very modest, while the fragmentation due to "over-allocations" increases. We believe that since Java requests are often very small (average number of bytes for our test data is about 100 bytes as shown in Table 1), we feel that it is not very beneficial to "over-allocate" large chunks.

5. Garbage Collection Using Binary AVAILable Tree

In this section we will outline how the Binary Tree can be used for Generation Scavenging techniques for Garbage Collection [Unger 84, 92]. In such techniques, the heap is divided into two spaces: to-space and from-space. Initially, all allocations will be in one of the spaces (say from-space). When the space is exhausted, the system will start allocation from the other space (i.e., to-space). The system will transfer only the live objects in from-space over to-space. At the completion of this copying, the roles of the two spaces are reversed. In our Binary Tree implementation of AVAILable tree, we can simulate the to-space and from-space by selecting either the left sub-tree or right sub-tree for allocation. Traversing left sub-tree allocates from low addresses while traversing right sub-tree allocates from high addresses. Allocating from one end of the memory (either low or high addresses) also achieves better locality for allocated data. In this paper we have not implemented garbage collection suggested here, but we are in the process of exploring various collection and compaction policies within the context of our Binary Tree implementation.

6. Summary And Conclusions

In this paper we described the use of Binary Trees for maintaining the available chunks of memory. The Binary Tree is based on the starting address of memory chunks. In addition, we keep track of the sizes of largest blocks of memory in the left and right sub-trees. This information is used during allocation to find a suitable chunk of memory. Our data shows that Best Fit (or Better Fit) allocation policies can easily be implemented using the chunk sizes in the left and right sub-trees. The Binary Tree implementation permits immediate coalescing of newly freed memory with other free chunks of memory. Binary Tree naturally improves the search for appropriate size blocks of memory over Linear Linked lists.

We have empirically compared the Binary Tree method with traditional Linked lists for implementing AVAILable lists, using Java Spec98 benchmarks. The data shows that the Binary Tree approach consistently outperforms the Linked list method. The data also shows that since, the average number of nodes in the tree is substantially smaller than those in the linked list, the memory overhead in maintaining the AVAILable list is actually smaller for Binary Tree, although each node maintains more information.

We plan to extend these experiments in a variety of ways: use non-Java applications with larger allocation requests; implement garbage collection methods based on Generation Scavenging; develop a hardware implementation of our memory allocation method for the purpose of embedding it in a IRAM device.

7. References

[Abdullahi 98] S.E. Abdullahi and G.A. Ringwood. "Garbage collecting the Internet: A survey of distributed garbage collection", *ACM Computing Surveys*, Sept. 1998, pp 330-373.

[Calder 95] B. Calder, D. Grunwald and B. Zorn. "Quantifying behavioral differences between C and C++ programs", *Tech Rept. CU-CS-698-95*, Dept of CS, University of Colorado, Boulder, CO, Jan 1995.

[Chang 99] J. M. Chang, W. Lee and Y. Hasan. "Measuring dynamic memory invocations in object-oriented programs", *Proc. of 18th IEEE International Performance Conference on Computers and Communications (IPCCC-1999)*, Feb. 1999

[Eustance 94] A. Eustance and A. Srivastava. "ATOM: A flexible interface for building high performance program analysis tools" *DEC Western Research Laboratory, TN-44*, 1994.

[Javaspec 98] Java Spec98 Benchmarks and documentation can be obtained from http://www.spec.org/osg/jvm98/.

[Johnstone 98] M.S. Johnstone and P.R. Wilson. "The memory fragmentation problem: Solved?" *Proc. Of the International Symposium on Memory Management*, Vancouver, British Columbia, Canada, October 1998, pp 26-36.

[Knuth 73] D.E. Knuth. *The Art Of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, 1973.

[Knowlton 65] K.C. Knowlton. "A fast storage allocator", *Communications of the ACM*, Oct. 1965, pp 623-625.

[Unger 84] D.M.Unger. "Generation scavenging: A non-disruptive high performance storage reclamation algorithm", *Proc. Of ACM SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, pp 157-167.

[Unger 92] D.M. Unger and F. Jackson. "An adaptive tenuring policy for generating scanvengers", *ACM Transaction On Programming Languages and Systems*, Jan. 1992, pp. 1-27.

[Wise 78] D.S. Wise. "The double buddy-system". Technical Report 79, Computer Science Department, Indiana University, Bloomington, IN, Dec. 1979.

[Wilson 95] P.R. Wilson, et. Al. "Dynamic storage allocation: A survey and critical review" *Proc. Of 1995 International Workshop on Memory Management*, Kinross, Scotland, Springer-Verlag LNCS 986, pp1-116.