Scientia Iranica D (2016) 23(3), 1217–1227

Sharif University of Technology
Scientia Iranica
Transactions D: Computer Science & Engineering and Electrical Engineering
www.scientiairanica.com

ABT and SBT revisited: Efficient memory management techniques for object oriented and web-based applications

M. Rezaei[a,*] and K.M. Kavi[b]

a. *Department of Information Technology, Faculty of Computer Engineering, University of Isfahan, Isfahan, P.O. Box 81746 73441, Iran.*
b. *Department of Computer Science, University of North Texas, P.O. Box 311366, Denton, Texas 76203, USA.*

Received 18 January 2015; received in revised form 20 October 2015; accepted 8 February 2016

KEYWORDS
Dynamic memory management;
Storage utilization;
Sequential fit;
Segregated free lists;
Binary trees.

Abstract. Dynamic memory management is an important and essential part of computer systems design. Efficient memory allocation, garbage collection, and compaction are becoming critical in parallel and distributed applications using object oriented languages like C++ and Java. In addition to achieving fast allocation/de-allocation of memory objects and fragmentation, memory management techniques should strive to improve the overall execution performance of object oriented applications. In this paper, we introduce Address Ordered and Segregated Binary Trees, two memory management techniques particularly efficient for object oriented applications. Our empirical results manifest that both ABT and SBT, when accompanied by coalescing, outperform the existing allocators such as Segregated free lists in terms of storage utilization and execution performance. We also show that these new allocators perform well in terms of storage utilization, even without coalescing. This is in particular suitable for web-applications.

1. Introduction

The efficiency of memory management algorithms, particularly in object oriented environments, has drawn the attention of researchers. The need for more efficient memory management is currently being driven by the popularity of object oriented languages in general [1,2], and Java in particular [3]. A memory manager's task is to organize and track the free chunks of memory as well as the current memory used by the running process. The primary goals of any efficient memory manager are high storage utilization and execution performance [4]. Current implementations, however, have failed to achieve both aims at the same time. For example, Sequential Fit algorithms show high storage utilization but poor execution performance [5,6]. On the other hand, Segregated Free lists reveal higher memory fragmentations, yet their execution performances are among the best. Well-known placement policies, such as Best Fit and First Fit, have been explored with both Sequential Fit and Segregated free lists for either speed or storage utilization benefit.

In this paper, we propose variations to Binary Tree allocators and Address Ordered and Segregated Binary Tree memory managers that report reasonable execution performance while maintaining low fragmentation compared with existing allocators.

We strongly believe that the reason that ABT and SBT outperform the existing allocators is their better cache locality behavior. As it is shown by other researchers, the binary tree configuration of ABT, as well as the segregation nature of SBT, behaves

*. *Corresponding author.*
E-mail addresses: m.rezaei@eng.ui.ac.ir (M. Rezaei); kavi@cs.unt.edu (K.M. Kavi)

adjustably well with respect to both temporal and spatial locality outcomes of the applications [7,8].

In a study by Johnstone and Wilson [5], the authors thoroughly analyzed the memory fragmentation (i.e., the excess amount of memory used by an allocation method beyond what is actually requested by an application). They identify memory allocation policies as First Fit, Best Fit, Next Fit, Worst Fit, and Buddy System and investigate the memory fragmentation using different implementations of these policies. Two key conclusions of the paper are:

1. Less fragmentation results from a policy (and its implementation) immediately coalescing freed memory;
2. Since objects allocated at the same time tend to die at the same time, it is better not to allocate from recently freed chunks of memory. If a memory allocation policy allocates temporally neighboring requests in spatially neighboring memory addresses, it is possible for these blocks to be coalesced when freed.

Note that coalescence of freed chunks can adversely affect the execution efficiency. Our approaches, however, using address ordered binary trees, lead to an efficient implementation of immediate coalescence of free blocks [6].

In this paper, first, we describe most commonly used process memory managers (aka allocation techniques) and address their shortcomings, in both execution performance and storage utilization terms. Then, we present our implementations of user process memory manager that address the disadvantages of available allocation techniques. Last sections of this paper present empirical results and draw our conclusions.

2. Levels of memory management system

For fully comprehending and appreciating the memory management system, it is necessary to realize its role in a typical computer system. Figure 1 shows the two levels of computer system memory manager: Operating system and user process memory managers.

Operating System (OS) memory manager allocates large chunks of memory, called OS pages, to process memory management systems. The size of OS pages, allocated to process memory managers, is fixed. For example, Linux uses 4 KByte pages whereas most unix based operating systems use 8-KByte page sizes. During the entire period of its execution, a user process acquires no more than 300 OS pages. In contrast, runtime system memory managers (user-level processes) are responsible for allocating small chunks of memory to running processes [9]. A typical process allocates more than tens of thousands of objects of different sizes dynamically [6]. The separation of memory management is needed to eliminate too frequent kernel calls when the memory space of a running process grows dynamically.

Usually, runtime system (memory manager of runtime system is in our interest) uses different primitives to increase and decrease the address space of "heap" and "stack"; "heap" is the memory space that houses dynamically allocated objects, whereas "stack" is the memory space for keeping the local variables of functions when they are activated. Figure 2 depicts how the address space of a running process, especially "heap" and "stack", grows. For example, "**gcc**", the gnu c compiler, provides "obstack" (object stack) routines for resizing the stack space and allocation libraries for resizing the "heap". When a user program needs more space, it issues an allocation request. User process memory manager, in response, returns the address of the block of memory as large as the requested size. If the process memory manager cannot successfully respond to the request, it will acquire more memory from Operating System (OS) memory manager via kernel system calls. Unix like OSs provides two families of system calls for such purposes: "**brk, sbrk**" and "**mmap, munmap**". "brk" returns so-called break point of the user process address space. Via "sbrk", user process memory manager is able to either acquire more memory from OS or release some of the unused portion of its available memory back to the OS. One of the main concerns with "sbrk" is that the caller is responsible for page alignment of the returned addresses. The functionality of the other

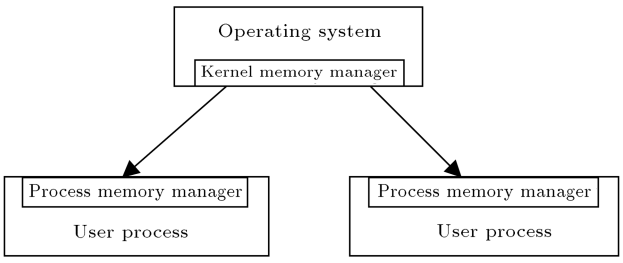

Figure 1. Memory management hierarchy.

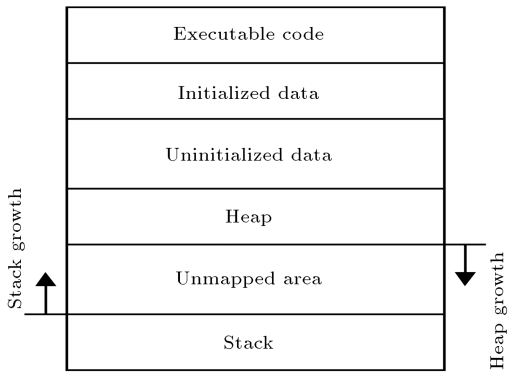

Figure 2. Memory area of a user process.

family of OS memory manager system calls, "**mmap**" and "**munmap**", is similar to that of "sbrk" and "brk"; but the returned addresses are page aligned and the caller does not need to maintain the page alignment afterwards. They are also supported in other Operating Systems like Windows families.

3. Known allocation techniques

Currently, used memory allocation schemes can be classified into *Sequential Fit* algorithms, *Buddy Systems*, *Segregated Free Lists*, and *Binary Tree* techniques.

Sequential Fit approach (including First Fit and Best Fit) keeps track of available chunks of memory in a doubly linked list. Known Sequential Fit techniques differ in how they track the memory blocks, how they allocate memory requests from the free blocks, and how they place newly freed objects back into the free list. When a process releases memory, these chunks are added to the free list, either at front or in place, if the list is sorted by addresses (Address Order [4]). When an allocation request arrives, the free list is searched until an appropriate sized chunk is found. The memory is allocated either by granting the entire chunk or by splitting the chunk (if it is larger than the requested size). Best Fit methods try to find the smallest chunk that is at least as large as the request, whereas First Fit methods find the first chunk that is at least as large as the request [10,11]. Best Fit method may involve delays in allocation while First Fit method leads to more external fragmentation [5]. If the free list is in address order, newly freed chunks may be combined with their surrounding blocks. Such practice, referred to as coalescing, is made possible by employing boundary tags in doubly linked list of address ordered free chunks [10].

In Buddy Systems, the size of any memory chunk (live, free, or garbage) is 2^k for some k [10]. Two chunks of the same size that are next to each other, in terms of their memory addresses, are called buddies. If a newly freed chunk finds its buddy among free chunks, two buddies can be combined into a larger chunk of size 2^{k+1}. During allocation, larger blocks are split into equal sized buddies until a small 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 reported that as much as 25% of memory is wasted due to internal fragmentation in buddy systems [5]. An alternate implementation, Double Buddy, which creates buddies of equal size but does not require the sizes to be 2^k, has shown to reduce the fragmentation by half [5,12].

Segregated free list approaches maintain multiple linked lists, one for each different sized chunk of available memory. Allocation and de-allocation requests are directed to their associated lists based upon the size of the requests. Segregated Free Lists are further classified into two categories: *Simple Segregated Storage* and *Segregated Fits* [4]. No coalescing or splitting is performed in Simple Segregated Storage and the size of chunks remains unaltered. If a request cannot be satisfied by its associated sized list, additional memory from operating system is acquired via sbrk or mmap system calls. In contrast, Segregated Fit allocator attempts to satisfy the request from a list containing larger sized chunks - a larger chunk is split into several smaller chunks if required. Coalescing is also employed in Segregated Fit allocators for further improvement of storage utilization. Simple Segregated Storage allocators are best known for their high execution performance while Segregated Fit allocators' edge is their high storage utilization.

In Binary Tree allocators, free chunks of memory are kept in a binary search tree whose search key is the address of the free chunks of memory. **Cartesian Tree**, which was proposed almost three decades ago, is one of the known Binary Tree Allocators [13]. This allocator is an address ordered binary search tree that forces its tree of free chunks to form a heap in terms of chunk sizes. In other words, Cartesian Tree allocator maintains a binary tree whose nodes are the free chunks of memory with the following conditions:

a. Address of descendants on left (if any) $\leq$ address of parent $\leq$ address of descendants on right (if any);
b. Size of descendants on left (if any) $\leq$ size of parent $\geq$ size of descendants on right (if any).

The latter that mandates Cartesian Tree to have its largest node at the root, causes the tree to become usually unbalanced and possibly degrade into a linearly linked list.

There exists a variety of ad hoc allocators in literature that are not included in this work for several reasons. First and foremost, our study is directed towards general purpose allocators. Secondly, it is not our intention to concentrate on allocators, rather we would like to form a smarter allocator which possesses reasonable performance, high storage utilization, and good locality behavior. More thorough taxonomy of different allocators can be found in the survey written by Wilson et al. [4].

4. Proposed allocation techniques: Address ordered and segregated binary trees

In Address Ordered Binary Tree (ABT), the free chunks of memory are maintained in a binary search tree like in Cartesian Tree [6]. To overcome the inefficiency forced by the size condition of Cartesian Tree allocator (condition b), we not only remove this restriction entirely from our implementation, but also

replace it with a strategy that enhances the allocation speed of ABT technique. Similar to Segregated Fit allocator, Segregated Binary Tree keeps several ABTs, one for each class size.

4.1. Address Ordered Binary Tree (ABT)

In this specific implementation of Binary Tree algorithms, each node of the tree contains the sizes of the largest memory chunks available in its left and right sub-trees. This information can be utilized to improve the response time of allocation requests by directing search for an appropriate sized chunk of memory, thus implementing Better Fit strategy [6,4]. The Binary Tree algorithms whose trees are address ordered are ideally suited for coalescing the free chunks; hence, storage utilization is further improved. In our Address Ordered Binary Tree, while inserting a newly freed chunk of memory, we check if it can be coalesced with existing nodes in the tree. Inserting a new free chunk will require searching the tree with $O(l)$ complexity where l is the tree level bounded by $\log_2(n)$ and n; n is the number of nodes in the tree. It is possible that the tree de-generates into a linear list, leading to a linear $O(n)$ insertion complexity. To minimize the insertion complexity, we advocate periodic tree re-balancing, which can be aided by keeping the information about the levels and number of nodes of the left and right sub-trees. Note that coalescing of the chunks described above already helps in keeping the tree from being unbalanced. Thus, the number of times a tree should be re-balanced, although it depends on specific application, will be relatively infrequent in our approach.

4.1.1. Algorithm for inserting a newly freed memory chunk

The following algorithm shows how a newly freed chunk can be added to Address Ordered Binary Tree of available chunks. The data structure of each node representing the free chunk of memory contains chunk's size, pointers to its left and right children, a pointer to its parent, and the sizes of largest chunks in its right and left sub-trees. The structure used for each node of the tree:

```
struct node{
   size_t Size;
   size_t MaxLeft;
   size_t MaxRight;
   struct node *Left;
   struct node *Right;
   struct node *Parent;
};
```

Insertion and Coalescing algorithms.

```
void INSERT(void *ChunkAddress, size_t
    ChunkSize, node *Travel){
  if(((void *)Travel+ Travel->size ==
  ChunkAddress) ||
    (ChunkAddress + ChunkSize == (void *)Travel))
      COALESCE(ChunkAddress,ChunkSize,Travel);
  else{
    if (ChunkAddress < (void *)Travel){
      if (Travel->Left == NULL){
          Travel->Left=CREATE(ChunkAdress,
          ChunkSize);
          Travel->MaxLeft=ChunkSize;
          ADJUSTSIZE (Travel);
      }
      else
          INSERT (ChunkAddress,ChunkSize,Travel-
          >Left);
    }
    else{
      if (Travel->Right == NULL){
          Travel->Right=CREATE
         (ChunkAddress, Chunksize);
          Travel->MaxRight=ChunkSize;
          ADJUSTSIZE (Travel);
      }
      else
            INSERT (ChunkAddress, ChunkSize,
Travel-
          >Right);
    }
  }
}

void ADJUSTSIZE (node *Travel){
  if (Travel->Parent == NULL)
    return NULL;
  if (Travel->Parent->Left == Travel)
    Travel->Parent->MaxLeft=
          MAX (Travel->Size, Travel- >MaxLeft,
         Travel->MaxRight);
  else
    Travel->Parent->MaxRight=
          MAX (Travel->Size, Travel->MaxLeft,
         Travel->MaxRight);
  ADJUSTSIZE (Travel->Parent);
}

void COALESCE (void *ChunkAddress,size_t Chunk-
Size, node *Travel){
  if (ChunkAddress > (void *) Travel)
    Travel->size+=ChunkSize;
  else{
    ChunkAddress=(node *)Travel;
    ChunkAddress->size+=ChunkSize;
  }
  ADJUSTSIZE (Travel);
}
```

4.1.2. Complexity analysis

INSERT is very similar to binary tree traversal and its time complexity depends on l, levels of the tree. COALESCE's complexity depends on ADJUSTSIZE function that traverses the tree upwards; therefore, both COALESCE and ADJUSTSIZE possess $O(l)$ time complexity. The other functions used in the implementation of ABT are SEARCH and DELETE. SEARCH is a search binary tree that is improved with keeping MaxLeft and MaxRight; hence, its upper bound time complexity is $O(l)$. DELETE function only visits one node but it calls ADJUSTSIZE and therefore its time complexity is also $O(l)$.

4.2. Segregated Binary Tree (SBT)

In a manner similar to Segregated Fit technique, Segregated Binary Tree keeps several Address Ordered Binary Trees, one for each chunk size [14]. Each tree is typically small, thus reducing the search time while retaining the memory utilization advantage of Address Ordered Binary Tree. In our implementation, SBT contains 8 binary trees; Memory chunks less than 64 bytes and greater than 512 bytes are kept in the first and the last binary trees, respectively. Each binary tree is responsible for keeping chunks of a size range, and sizes range in 64 byte intervals. For example, the second binary tree's range is [64,128) (viz., if a chunk's size is x then $64 \leq x < 128$).

5. Empirical results

In order to evaluate the benefits of our approach to memory management, we developed simulators that accepted requests for memory allocation and de-allocation. In this work, four major allocator algorithms were implemented and studied, i.e. Address Ordered Binary Tree (ABT), Sequential Fit (SqF), Segregated Binary Tree (SBT), and Segregated Fit (SgF). We have investigated the impact of different placement policies such as First Fit, Best Fit, and Better Fit on the proposed allocators. SqF is a resemblance of Sequential Fit algorithm mentioned in the previous sections and SgF resembles Doug Lea's well known allocator [15]. We have conducted our experiments with and without coalescing to investigate its impact on different allocators, especially ABT and SBT. In this section, we will first explain our framework and then show the data collected while performing our experiments.

5.1. Experimental framework

For our experiments, we have used Java Spec98 benchmarks since java programs are allocation-intensive [16]. Applications with large amount of live data (dynamically allocated) are worthy benchmark candidates for memory management algorithms, because they clearly expose the memory allocation speed and memory fragmentation.

Java Spec98 benchmarks are instrumented on Unix to collect traces indicating memory allocation and de-allocation requests. Table 1 summarizes the benchmarks' descriptions while Table 2 reports the benchmarks' statistics. Table 2 shows that:

- On average, Java applications allocate more than 70,000 objects;
- About ten thousand objects are not de-allocated; they lead to the memory leaks. Note that Java run time system uses an automatic memory manager so that programmer is not responsible for de-allocating the objects. Automatic memory manager deals with three types of objects: live (application is currently using them), free (which are in the free list of the allocator), or garbage (application is not using them anymore, but they have not been freed yet). Every so often, when memory is exhausted, garbage collector, which is a part of the automatic memory manager, in some way, identifies the garbage objects. If any object is marked as garbage, the garbage collector will then issue a de-allocation request to move them to the free list;
- Java Virtual Machine (JVM) memory management system consists of a garbage collector and an allocator. JVM garbage collector issues allocation requests for new objects and decides when an object is no longer used by the application (garbage object).

Table 1. Benchmark description.

Benchmark	Description
check	A simple program that tests various features of JVM
compress	Modified Lempel-Ziv method (LZW)
db	Performs data base functions
jack	A Java parser generator
javac	Java compiler for JDK 1.0.2
jess	Java expert shell system
mpegaudio	Decompresses audio files that conform MPEG3
mtrt	A threaded raytracer

Table 2. Benchmark statistics.

Benchmark	Total no. allocation requests	Total no. de-allocation requests	Average request (bytes)	Average live (Mbytes)
check	46,665	40,991	96	0.82
compress	41,239	35,849	94	0.77
db	43,176	37,727	94	0.79
jack	80,814	73,854	86	1.12
javac	132,745	123,222	87	1.68
jess	81,857	74,298	93	1.29
mpegaudio	97,430	91,095	77	1.23
mtrt	54,682	48,662	91	0.92
average	71,148	64,603	89.2	1.05

It then automatically issues de-allocation requests for garbage objects. Garbage collector directs the allocation and de-allocation requests to the allocator that is responsible for managing the heap (allocating the objects from the heap and returning the freed objects to the heap);

- The average amount of live memory is 1 MBytes - 128 pages if page size is 8 KBytes. Normally, about 50% of memory is wasted; hence, an allocator needs 260 pages of memory to perform reasonably in terms of fragmentation;
- Average request size is 90 Bytes, which means there are 12,000 live objects at any given time during the application's run.

For comparing different allocators, we collected statistical data for the following items:

Average number of free chunks measures the memory overhead of a memory management algorithm. It is also an indication of how the memory space is fragmented; the more the number of the free chunks, the higher the memory fragmentation.

OS pages consumed shows how many pages of memory are acquired by the allocator. Each page is 8 KBytes; hence, this figure multiplied by 8 equals the amount of memory an allocator has consumed in KBytes. This number also reports how many times the operating system kernel is called to increase the heap size. Kernel system calls are expensive and cost a lot of execution cycles; therefore, *OS Pages Consumed* is an indirect indication of execution performance.

Internal fragmentation measures the excess memory allocated by an allocator as compared to the actual memory requested by the user program.

Average numbers of nodes searched at allocation and de-allocation measure the execution complexity for each algorithm while allocating or de-allocating a chunk of memory. These numbers provide a measure of the allocator's execution efficiency.

Maximum numbers of nodes searched at allocation and de-allocation are the worst-case execution time for allocation and de-allocation.

Coalescence frequency measures how often a newly freed chunk of memory can be combined with other free nodes. "Less fragmentation will result if an implementation immediately coalesces free chunks" [4].

5.2. Observations

For better illustration, observation, and analysis of the data garnered based on this work, we use boxplot, a plot which shows first quartile, median, and third quartile along with minimum and maximum on distribution of statistical data for each metric. The boxes on the plots illustrate the area in which the central 50% of distribution lies [17].

In this research, three classes of allocation algorithms were studied, i.e. simple techniques (doubly linked lists and binary trees - simple data structures to keep track of free chunks of memory), coalescence-permitted techniques (doubly link lists and binary trees of free chunks with the possibility of coalescence and concatenation of adjacent free chunks), and segregation (segregated free lists and segregated binary trees of free chunks). For example, ABT-FFWC stands for Address Ordered Binary Tree - First Fit With Coalescing - whereas ABT-FF is simply the same allocator without coalescing.

Following subsections are devoted to discussions about our findings in two directions, execution performance and storage utilization.

5.2.1. Execution performance

The dominant metric in evaluating execution performance of allocation algorithms is *The Average Number*

of Nodes Searched at Allocation (AN^{2}SA). Figure 3 illustrates a comparison between all of the allocation techniques with respect to AN^{2}SA. SqF is a LIFO (Last In First Out) list; newly freed chunks are put at the front of the list. This means that SqF pays only one node search for de-allocation. For a fair comparison, then, one needs to consider both nodes searched at allocation and de-allocation of ABT (add them up) and compare the total number with the number of nodes searched only at allocation of SqF. For the beginning, we consider the allocation overhead, since with the advent of multicore and multithreaded architectures, a thread can be devoted to performing memory management tasks. If that is the case, then only the library calls for allocating memory via application thread will be blocking calls, whereas de-allocation will become a background activity - we will get back on this issue later on in this paper.

The first glance at Figure 3 indicates that keeping free chunks in a binary search tree can be well compared to the existing allocation techniques in terms of speed. Secondly, but more importantly, both coalescing and segregation are more pronounced when enforced on SqF than ABT. Note that the second four boxplots present the impact of coalescing and the third four show the influence of segregation.

From among twelve allocation algorithms studied in this work, we selected eight better performed ones and represented their allocation behavior in Figure 4. These data show that six out of eight best performed allocators in this research are from the family of binary search tree implementation (either ABT or SBT). Secondly, both Better Fit and First Fit flavor of Segregated Binary Tree Allocator behave the best. Therefore, we suggest both of these allocation techniques for general purpose environments.

It is worth mentioning that number of times an

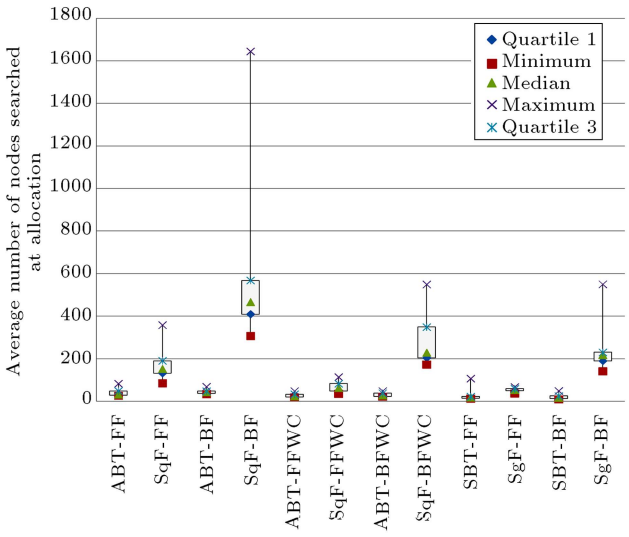

Figure 3. Execution performance comparison.

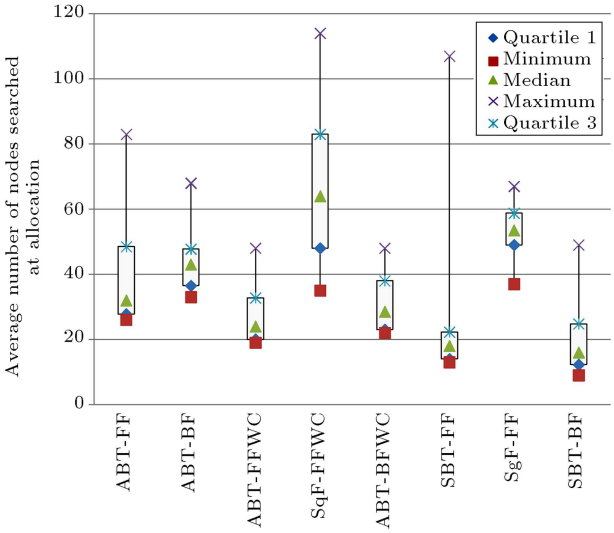

Figure 4. Execution performance comparison of the selected allocators.

allocator issues a system call also reffects (indirectly) the execution performance. Figure 5 illustrates total number of times each allocation technique uses "sbrk" or "mmap" system call for using more memory resources. Kernel system calls are expensive and hence more "sbrk" system calls degrade the performance. Although segregation equalizes the behavior of allocators in terms of "sbrk" system call, the family of allocation techniques that we studied in this work tend to behave almost the same with respect to number of "sbrk" system calls.

Finally, Figure 6 depicts the allocators' behavior in terms of allocation speed for the eight selected techniques when the number of nodes searched at de-allocation is added to AN^{2}SA for the sake of a fair comparison. The value of ABT and SBT allocators is

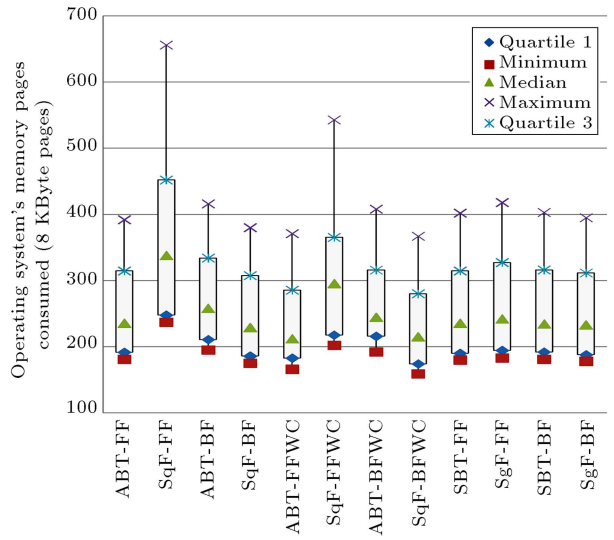

Figure 5. Total number of "sbrk" system calls during the execution of applications.

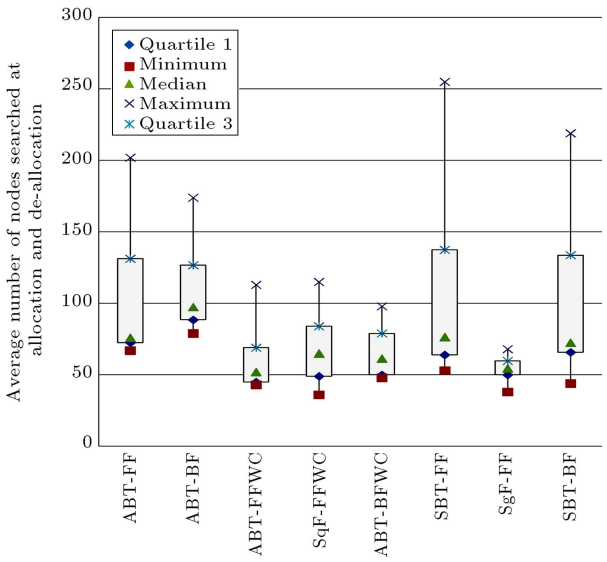

Figure 6. Execution performance comparison of the selected allocators, with the de-allocation overhead.

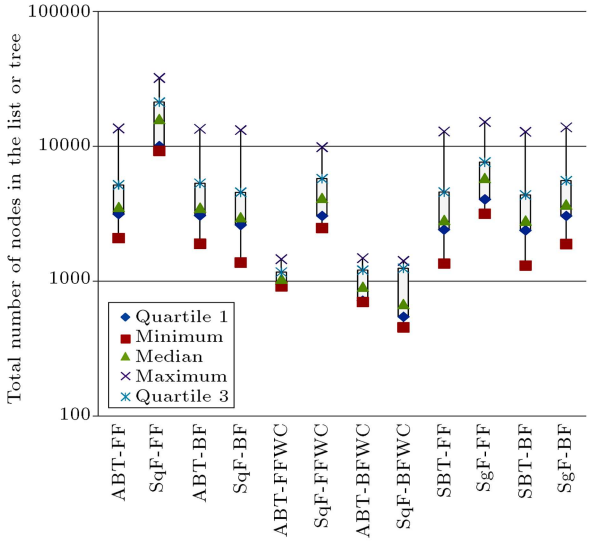

Figure 7. Total number of nodes used by allocation techniques.

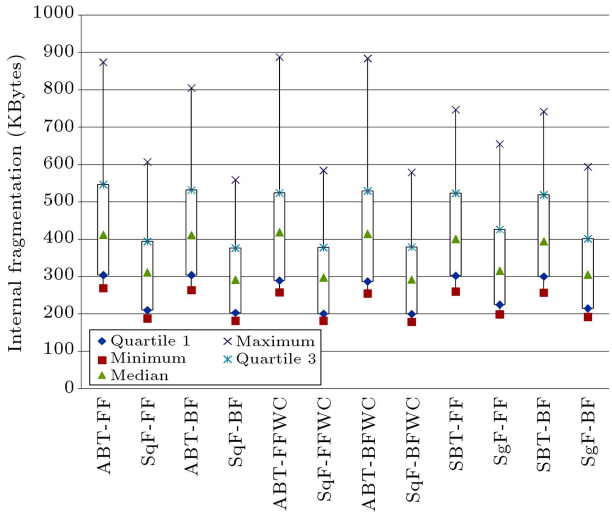

Figure 8. Internal fragmentation.

due to the property of off-loading de-allocation portion of memory management overhead; nevertheless, Figure 6 convinces us that all flavors of ABT and SBT can be well compared to their counterparts in SqF and SgF.

5.2.2. Storage utilization

Every time an allocator issues an "sbrk" system call, kernel returns the beginning address of 8 KByte free space back to the allocator. Therefore, more "sbrk" system calls during the execution of a program means more external fragmentation. It is evident from Figure 5 that two best performed allocators are ABT-FF and SqF-BF in terms of external fragmentation. It can also be inferred that segregation tends to equalize allocators' external fragmentation behaviors.

Furthermore, total number of free chunks in the lists or binary trees is also an indication of fragmented memory. Figure 7 depicts the boxplot of total number of nodes in free lists for the allocation techniques of this study. It is interesting to note that still ABT-FF and SqF-BF perform well and segregation impact shows the consistency with what we concluded from Figure 5 for external fragmentation.

Internal fragmentation, however, is a different story. Figure 8 shows internal fragmentation behavior of the allocation techniques under investigation in this research. Overall, this figure simply conveys that variation of binary tree allocators reveals more internal fragmentation due to the overhead of more information kept in each node. Each free chunk in Address Ordered Binary Tree keeps three sizes and MaxRight and three addresses in its header. If the computer system is 32 bits, each of these items will be 4 bytes; consequently, 24 bytes is needed for keeping each chunk in the tree ("3 addresses + 3 sizes" *4 = 24). This means that any allocation request smaller than 24 bytes should be rounded up to 24. If the request sizes are small (which is the case for Java benchmarks), allocating bigger chunks for small sized requests causes higher internal fragmentation. On the other hand, Sequential Fit allocators need two addresses (previous and next pointers) and a size (chunk's size), which add up to 12 Bytes overhead.

5.3. Analysis

For the big picture analysis of this work, we categorized the allocators' behavior considering two main objectives of memory management techniques, i.e. execution performance and storage utilization. While examining Figures 4 and 5, one can conclude that ABT-FFWC, SBT-FF, SBT-BF, and ABT-FF perform the best in terms of speed. The two figures that also represent storage utilization behavior of allocators are

Figures 8 and 5, which illustrate that SqF-BFWC, SqF-BF, SBT-FF, and ABT-FFWC perform the best, respectively. However, the performance of allocators is more comparable via their execution time. In other words, allocators perform almost the same in terms of storage utilization and almost quite differently in terms of speed. Having conveyed this, when we consider the two above sets and conduct a simple union, it is indeed inferred that ABT-FFWC and SBT-FF are two candidates for the memory management techniques in general purpose environments. It is also worth noting that the execution overhead of coalescing is negligible when compared with the search time. Each coalescence, at most, needs two comparisons, three pointer modifications, and an addition - a total of six integer operations. If there is no chance of coalescence, though, the extra overhead is only a comparison. In our study, we calculated the frequency of coalescing, too, which turned out to be 46 to 59% for all allocators. This means that on average, about four integer operations $((0.59 * 6) + ((1 - 0.59) * 1) = 3.95)$ are added on each de-allocation, nonetheless it could all be offloaded to an separate thread in multithreaded or multicore environment.

Consider a set of applications whose memory objects live a short period of time. This is the case for web-based applications where memory objects belong to transactions. Inoue et al. have shown that allocators which tend to perform coalescing on each case of freeing an object behave poorly in multicore environments [18]. The allocation technique that they suggested eliminates coalescing, which in turn reduces bus traffic in multicore processors; it makes 11.4% to 51.5% performance improvement compared to default allocators which do perform coalescing [18]. Utilizing this result, we strongly believe that SBT-FF will be a good candidate for web applications.

We observe a great improvement on *Maximum Number of Nodes Searched at Allocation and De-Allocation*, which can be viewed in Figure 9. These numbers reflect the worst-case execution time.

Finally and for parallel and distributed systems, we should mention that in many OpenMP applications, memory is allocated by one thread and shared by other threads. Thus, our technique is useful in managing memory allocation by the main thread. In case of MPI, since the same code is run on all processes, memory allocation and reallocation are replicated in nodes and our technique can be applied in parallel for each node.

6. Conclusions

We have proposed new memory management algorithms, ABT and SBT, that maintain the available chunks of memory in binary search trees. The search

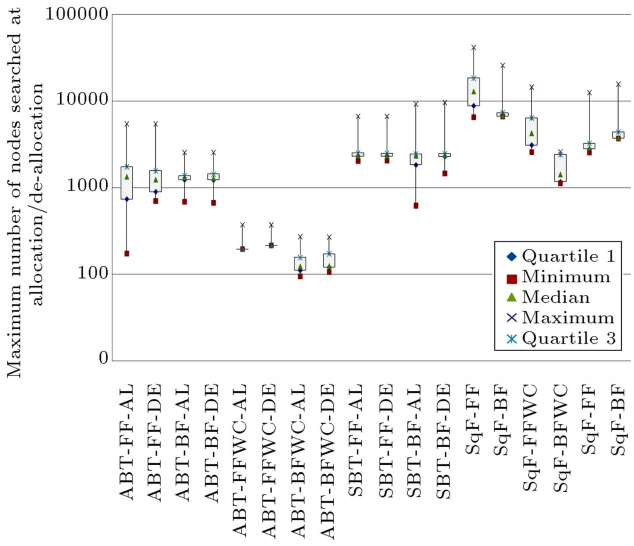

Figure 9. Worst-case execution time.

key in ABT and SBT is the starting address of the free chunks of memory. In addition, we keep track of the sizes of largest chunk of memory in the left and right sub-trees. This information is used to speed up the search phase of allocation.

We have used Java applications to compare ABT and SBT with the existing allocators, Sequential and Segregated Fit algorithms, since Java applications allocate several tens of thousands of objects of varied sizes. From Java Spec98 Benchmarks, we have collected the allocation and de-allocation traces and fed them to the memory management simulators. We have designed memory management simulators that report data on memory fragmentation and search time of allocation and de-allocation requests.

In general, allocators perform the best when they are allowed to explore coalescing. ABT and SBT are address ordered; hence, coalescing, specifically, helps these allocators outperform others in allocation time. In today's multithreaded architecture, one thread can be scheduled to execute the application's code (application thread) while another runs allocator's code (memory management thread). The application thread can fully run in parallel with the memory management thread when processing the de-allocation requests; therefore, what matters the most is allocation search time when the execution of the application thread is blocked. Since the fastest allocation search time among all allocators studied in this paper has been achieved by SBT with coalescing, it is the best candidate for the memory management thread.

Maximum Number of Nodes Searched at Allocation and De-allocation.

As shown in the tables, SBT with coalescing reports the worst case search time (Max number of Nodes

Searched at Allocation is 110 for Better Fit SBT with coalescing).

Best Fit Sequential Fit with Coalescing behaves the best among allocators in terms of Storage Utilization. It shows about 14% improvement in terms of fragmentation when compared with Better Fit SBT with Coalescing. However, the execution performance improvement of Better Fit SBT compared with Best Fit SqF is 90% (17 + 12 = 29 compared with 286).

On the whole, the data represented in this paper shows that Address Ordered and Segregated Binary Trees' execution performance is far better than Sequential and Segregated Fits', while in terms of Storage Utilization all the allocators perform almost the same.

References

1. Berger, E.D., Zorn, B.G. and McKinley, K.S. "Composing high performance memory allocators", *The ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation*, Snowbird, UTA, pp. 114-124 (2001).
2. Chen, T.F. and Baer, J.L. "Effective hardware-based data prefetching for high- performance processors", *IEEE Transactions on Computers*, **44**(5), pp. 609-623 (1995).
3. Abdullahi, S.E. and Ringwook, G.A. "Garbage collecting the internet: A survey of distributed garbage collection", *ACM Computing Surveys*, **30**(3), pp. 330-373 (1998).
4. Wilson, P.R., Johnstone, M.S., Neely, M. and Boles, D. "Dynamic storage allo-cation: A survey and critical review", *Int. Workshop on Memory Management*, Kinross, Scotland, pp. 1-116 (1995).
5. Jonhnstone, M.S. and Wilson, P.R. "The memory fragmentation problem: Solved?", *Int. Symposium on Memory Management*, Vancouver, BC, pp. 26-36 (1998).
6. Rezaei, M. and Kavi, K.M. "A new implementation technique for memory management", *2000 IEEE Southeast Conf.*, Nashville, TN, pp. 332-339 (2000).
7. Jula, A. and Rauchwerger, L. "Fast allocation speed, low memory fragmentation, and high speed locality: Pick two", *Int. Symposium on Memory Management*, Dublin, Ireland, pp. 109-118 (2009).
8. Rezaei, M. and Kavi, K.M. "Intelligent memory manager: Reducing cache polutions due to memory management functions", *Elsevier Journal of Systems Architecture*, **52**(1), pp. 41-55 (2006).
9. Crowley, C. *Operating System: A Design - Oriented Approach*, Ed., 1st, McGraw-Hill Publishing, Boston, MA (1997).
10. Knuth, D.E. *The Art of Computer Programming*, 1st Ed., *Fundamental Algorithm*, 3rd Ed., Addison-Wesley, Boston, MA (1997).
11. Standish, T. *Data Structure Techniques*, 1st Ed., Addison-Wesley, Boston, MA (1980).
12. Wise, D.S. "The double buddy - system", Technical Report 79, Computer Science Department, Indian University, Bloomington, IN (1979).
13. Stephenson, C.J. "Fast Fit: New methods for dynamic storage allocation", *9th Symposium on Operating Systems Principles*, Bretton Woods, NH, pp. 30-32 (1983).
14. Rezaei, M. and Cytron, R.K. "Segregated binary tree: Decoupling memory manager", *Technical Committee on Computer Architecture Newsletter* (2001).
15. Lea, D. *A Memory Allocator*, http//gee.cs.oswego.edu/dl/html/malloc.html
16. *Java Spec98 and its Documents*, http://www.spec.org/osg/jvm98.
17. Benjamini, Y. "Opening the box of a boxplot", *The American Statistician*, **42**(4), pp. 257-262 (1988).
18. Inoue, H., Komatsu, H. and Nakatani, T. "A study of memory management for web-based applications on multicore processors", *The ACM SIGPLAN 2009 Conf. on Programming Language Design and Implementation*, Dublin, Irland, pp. 386-396 (2009).

Biographies

Mehran Rezaei is currently a faculty member and head of Information Technology Department at University of Isfahan, Iran. Before returning to his home country, from 2006 to 2010, he was a software consultant in Wells Fargo and FDIC, where he designed and developed data bases and data warehouse systems. During 2004-2006 he served as a faculty member in the department of Computer Engineering at University of Texas at Arlington.

His research is currently on application of text and data mining in Decision Support Systems and Business Intelligence, but since the focus of his PhD degree work was on computer architecture and mainly on memory side, he still retains some interests in computer systems architecture such as the cache configuration of approximate computing.

He received his PhD degree in Computer Sciences from the University of North Texas, MS in Electrical Engineering from University of Alabama in Huntsville, and BS degree in Electronics Engineering from Isfahan University of Technology.

Krishna Kavi is currently a Professor of Computer Science and Engineering and the Director of the NSF Industry/University Cooperative Research Center for Net-Centric Software and Systems at the University of North Texas. During 2001-2009, he held the Chair of the department. He also held an Endowed Chair Professorship in Computer Engineering at the University of Alabama in Huntsville, and served in the

faculty of the University Texas at Arlington. He was a Scientific Program Manager at US National Science Foundation during 1993-1995. He served in several editorial boards and program committees.

His research is primarily on Computer Systems Architecture including multi-threaded and multicore processors, cache memories, and hardware assisted memory managers. He also conducted research in the area of formal methods, parallel processing, and real-time systems. He has published more than 150 technical papers in these areas. He received more than US $5 M in research grants. He has supervised 14 PhDs and more than 35 MS students. He received his PhD from Southern Methodist University in Dallas Texas and a BS degree in EE from the Indian Institute of Science in Bangalore, India.