Report Number: WUCSE-2003-69

2003-10-10

Storage Coalescing

Delvin C. Defoe

Typically, when a program executes, it creates objects dynamically and requests storage for its objects from the underlying storage allocator. The patterns of such requests can potentially lead to internal fragmentation as well as external fragmentation. Internal fragmentation occurs when the storage allocator allocates a contiguous block of storage to a program, but the program uses only a fraction of that block to satisfy a request. The unused portion of that block is wasted since the allocator cannot use it to satisfy a subsequent allocation request. External fragmentation, on the other hand, concerns chunks of memory that reside between... Read complete abstract on page 2.

Storage Coalescing

Delvin C. Defoe

Complete Abstract:

Typically, when a program executes, it creates objects dynamically and requests storage for its objects from the underlying storage allocator. The patterns of such requests can potentially lead to internal fragmentation as well as external fragmentation. Internal fragmentation occurs when the storage allocator allocates a contiguous block of storage to a program, but the program uses only a fraction of that block to satisfy a request. The unused portion of that block is wasted since the allocator cannot use it to satisfy a subsequent allocation request. External fragmentation, on the other hand, concerns chunks of memory that reside between allocated blocks. External fragmentation becomes problematic when these chunks are not large enough to satisfy an allocation request individually. Consequently, these chunks exist as useless holes in the memory system. In this thesis, we present necessary and sufficient storage conditions for satisfying allocation and deallocation sequences for programs that run on systems that use a binary-buddy allocator. We show that these sequences can be serviced without the need for defragmentation. We also explore the effects of buddy-coalescing on defragmentation and on overall program performance when using a defragmentation algorithm that implements buddy system policies. Our approach involves experimenting with Sun's Java Virtual Machine and a buddy system simulator that embodies our defragmentation algorithm. We examine our algorithm in the presence of two approximate collection strategies, namely Reference Counting and Contaminated Garbage Collection, and one complete collection strategy - Mark and Sweep Garbage Collection. We analyze the effectiveness of these approaches with regards to how well they manage storage when we alter the coalescing strategy of our simulator. Our analysis indicates that prompt coalescing minimizes defragmentation and delayed coalescing minimizes number of coalescing in the three collection approaches.

Department of Computer Science & Engineering

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

2003-69

Storage Coalescing - Master's Thesis, December 2003

Authors: Defoe, Delvin C.

October 10, 2003

Abstract: Typically, when a program executes, it creates objects dynamically and requests storage for its objects from the underlying storage allocator. The patterns of such requests can potentially lead to internal fragmentation as well as external fragmentation. Internal fragmentation occurs when the storage allocator allocates a contiguous block of storage to a program, but the program uses only a fraction of that block to satisfy a request. The unused portion of that block is wasted since the allocator cannot use it to satisfy a subsequent allocation request. External fragmentation, on the other hand, concerns chunks of memory that reside between allocated blocks. External fragmentation becomes problematic when these chunks are not large enough to satisfy an allocation request individually. Consequently, these chunks exist as useless holes in the memory system.

In this thesis, we present necessary and sufficient storage conditions for satisfying allocation and deallocation sequences for programs that run on systems that use a binary-buddy allocator. We show that these sequences can be serviced without the need for defragmentation.

We also explore the effects of buddy-coalescing on defragmentation and on overall program performance when using a defragmentation algorithm that implements buddy system policies. Our approach involves experimenting with Sun's Java Virtual Machine and a buddy system simulator that embodies our defragmenation algorithm. We

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

EFFECTS OF COALESCING ON THE PERFORMANCE OF SEGREGATED SIZE STORAGE ALLOCATORS

by

Delvin C. Defoe

Prepared under the direction of Professor Ron K. Cytron

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

December, 2003

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

ABSTRACT

EFFECTS OF COALESCING ON THE PERFORMANCE OF SEGREGATED SIZE STORAGE ALLOCATORS

by Delvin C. Defoe

ADVISOR: Professor Ron K. Cytron

December, 2003
Saint Louis, Missouri

Typically, when a program executes, it creates objects dynamically and requests storage for its objects from the underlying storage allocator. The patterns of such requests can potentially lead to internal fragmentation as well as external fragmentation. Internal fragmentation occurs when the storage allocator allocates a contiguous block of storage to a program, but the program uses *only* a fraction of that block to satisfy a request. The unused portion of that block is wasted since the allocator cannot use it to satisfy a subsequent allocation request. External fragmentation, on the other hand, concerns chunks of memory that reside *between* allocated blocks. External fragmentation becomes problematic when these chunks are not large enough to satisfy an allocation request individually. Consequently, these chunks exist as useless holes in the memory system.

In this thesis, we present necessary and sufficient storage conditions for satisfying allocation and deallocation sequences for programs that run on systems that use a binary-buddy allocator. We show that these sequences can be serviced without the need for defragmentation.

We also explore the effects of buddy-coalescing on defragmentation and on overall program performance when using a defragmentation algorithm that implements buddy system policies. Our approach involves experimenting with Sun's Java Virtual Machine and a buddy system simulator that embodies our defragmenation algorithm. We examine our algorithm in the presence of two approximate collection strategies, namely Reference Counting and Contaminated Garbage Collection, and one complete collection strategy - Mark and Sweep Garbage Collection. We analyze the effectiveness of these approaches with regards to how well they manage storage when we alter the coalescing strategy of our simulator. Our analysis indicates that prompt coalescing minimizes defragmenation and delayed coalescing minimizes number of coalescings in the three collection approaches.

to my family and friends who always express confidence in my capacity to succeed

Contents

List of Tables

List of Figures

Acknowledgments

I am indebted to my advisor, Ron K. Cytron, for his supervision, support, guidance, and insight into the work that forms the substance of this thesis. I would also like to acknowledge the other members of my defense committee, Dr. Chris Gill and Dr. Aaron Stump, for offering their time and expertise in evaluating my research.

Thank you goes out to Reynold Bailey, Ronelle Bailey, Dante Cannarozzi, Sharath Cholleti, Morgan Deters, Matt Hampton, and Christine Julien, for their willingness to lend their time, support, and intellect in making this project a success.

I also extend my gratitude to the Chancellors Graduate Fellowship Program for financially supporting my education and for creating an environment that allows me to have fun while advancing my academic credentials.

I would especially like to thank my parents, Molly Williams and Morrel Defoe for their love, support, and care that they give so selflessly. Special thanks go out to my granny Aldith Joseph.

To all those who contributed in one way or the other in making this thesis a success, I thank you.

Delvin C. Defoe

Washington University in Saint Louis
December 2003

Chapter 1

Introduction

1.1 Memory Management

Typically, when a program executes on a machine it is associated with four *regions* of computer memory, namely the code segment, the global data area, the stack, and the heap [14].

Code Segment: The *code segment* contains the stored instructions of programs in execution. For the most part, instructions are preloaded in this area; hence, dynamic storage issues do not significantly affect this region.

Global Data Area: The *global data area* is used for allocating statically declared objects, such as global variables and constants [9]. These objects remain live for the duration of the executing program; thus their storage is not reclaimed until the program exits the runtime system. The static nature of these allocations means the storage allocator must reserve enough storage for these objects *a priori* and not pull storage from some dynamic source because dynamic storage is not guaranteed to be available when needed.

Stack: Typically, the *stack* is used to allocate and deallocate variables that are local to methods [9]. When a method is invoked in a single-threaded application, storage is allocated on the stack for its local variables. This allocation causes the stark to grow. When the method returns its local variables are no longer meaningful and can no longer be used, hence the storage associated with its variables are reclaimed and the result is the shrinkage of the stark. For a multi-threaded application, the storage allocation problem is not as simple,

since each thread may be associated with its own stack. This complicates the job of the storage allocator since it has to keep track of and respond to the storage behavior of each thread. Another issue multi-threading introduces is synchronization, especially if a single stack is used.

Heap: The *heap* is used to allocate objects dynamically using mechanisms such as the `malloc` function of the C programming language and the `new` operator of Java. These facilities allow for the setting apart of a contiguous block of storage in the heap for the dynamically allocated objects of an application written in these languages. Languages like C and C++ allow program developers the flexibility of explicitly deallocating objects and reclaiming the storage occupied by those objects using mechanisms such as `free` and `delete`, respectively. This process can be automated using garbage collection mechanisms and for languages like Java these are the only options.

As is the case with the stack, multi-threading also introduces issues with the heap that the storage allocator must address. Some of these issues involve synchronized access to the heap or to free storage of the heap, locking granularity, and overhead in updating the heap and in responding to allocation requests.

Typically, a process does not request that all its dynamic objects be allocated at the beginning of its execution and its dynamic objects are not all deallocated together some time later. Instead, allocations and deallocations are interspersed in time and space. This gives rise to the evolution of holes in the heap whenever deallocations occur in the middle of the heap. These holes may not be large enough individually to satisfy subsequent allocation requests since the objects that generate subsequent requests may require blocks of storage that are individually larger than each hole. A heap with holes as described above may become exhausted to the point where the next allocation request cannot be satisfied, even though there may be enough collective free storage to satisfy the request. Such a heap is considered *fragmented* and requires some form of *defragmentation* before an allocation request can be serviced. Defragmentation seeks to resolve the *fragmentation* problem by relocating live objects so that multiple adjacent holes can be merged into a larger contiguous block that can potentially be used to satisfy a subsequent allocation request.

1.2 Real-Time and Embedded Systems

Real-time systems model activities in the real world and, consequently, are concerned with keeping up with events that take place in real time in the outside world [17]. Some of the tasks of such systems are real-time tasks that have a degree of urgency associated with them. The correctness of these tasks depends not only on the results that follow from computations, but also on the time at which these results are made available. Communication networks, avionics, navigation, medicine, and multimedia are a few of the fields in which real-time systems are deployed.

Because these systems are time-critical systems, their allocation request sequences must be satisfiable in bounded time, both in terms of predictability and with respect to specific bounds derived from the application or its environmental characteristics. This leaves little allowance for defragmentation since defragmentation can be costly and can introduce unnecessary overhead, hence motivating the need for a defragmentation-free storage allocator.

Embedded systems are found in automobiles, microwave ovens, toys, telephones, robotics and mobile computing devices. Because of their particular characteristics and specific usage as components with specific functionality, embedded systems have stringent energy, power, and area constraints [10]. Embedded systems are also constrained with regards to cost and memory (of which the heap is a part) size. Typically, embedded systems are components of larger systems and exist mainly to perform specific functions, hence the storage requirements for these systems may be known *a priori* [5]. Notwithstanding their deployment, a decision must be made as to whether embedded systems should be equipped with the capabilities of static storage allocation or dynamic storage allocation. Static storage allocation is an inefficient storage management strategy if some allocated objects do not have a lifespan as long as the duration of program execution. Dynamic storage allocation also poses a problem since it tends to fragment the heap. We explore the fragmentation/defragmentation problem to gain insight into the extent to which it can be addressed.

1.3 Problem Definition

Fragmentation is one of the many serious issues with real-time and embedded systems when dynamic storage allocation is allowed. A fragmented heap may necessitate defragmentation in order to satisfy a subsequent allocation request. We seek to:

- provide necessary and sufficient storage conditions for satisfying allocation and deallocation sequences for programs that run on systems that use a binary-buddy allocator without the need for defragmentation, and to
- measure the effects of buddy-coalescing on defragmentation and on the overall performance of programs in execution.

We provide a detailed explanation of the experiments we have conducted, present our experimental results, and analyze these results in subsequent chapters.

1.4 Structure of Thesis

The rest of this thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 provides background information on storage allocators, defragmentation algorithms, and garbage collection techniques. Chapter 3 also introduces variations of the binary-buddy allocator. Chapter 4 discusses storage requirements for binary-buddy allocators and provides a proof for the general case of such allocators. Chapter 5 details the experiments we conducted in exploring the effects of buddy-coalescing on fragmentation and on program performance. Chapter 6 presents the results of our experiments and gives some analysis of the results. Finally, we offer conclusions and discuss future work in chapter 7.

Chapter 2

Related Work

2.1 The LINUX Kernel

The Linux Kernel employs a page memory system for which it must establish a robust and efficient strategy for allocating contiguous page frames [2]. In doing so, it has to deal with a well-known memory management problem called external fragmentation. External fragmentation is the idea that frequent allocations and deallocations of groups of contiguous page frames of different sizes may lead to a situation in which several small blocks of free page frames are "scattered" inside larger blocks of allocated page frames. One consequence of external fragmentation is that it may make it impossible to allocate a large block of contiguous page frames, even though there are enough free pages to satisfy the request.

To resolve the external fragmentation issue, an algorithm based on the binary buddy-system is employed. The kernel accesses physical memory through 4KB pages. All free page frames are grouped into lists of blocks that contain 1, 2, 4, 8, 16, 32, 64, 128, 265, or 512 contiguous page frames, respectively. The physical address of the first page frame of a block is a multiple of the group size - for example, the initial address of a 16-page-frame block is a multiple of $16 * 2^{12}$ ($16 * 4$KB).

Using a simple example, we show how the Linux Kernel uses the buddy algorithm. Assume there is a request for a group of 128 contiguous page frames, i.e., half a mega-byte. To satisfy this request, the algorithm first checks to see if a free block in the 128 page-frame list exists. If it finds one, it uses it to satisfy the request. If it fails to find one, the algorithm looks for the next larger block - a free block in the 256 page-frame list. If such a block exists, the kernel allocates 128 of the 256 page frames to satisfy the request and inserts the remaining 128 page frames into the list of 128

page-frame blocks. If a 256 page-frame block does not exist, the process continues to the list of 512 page-frame blocks. If that list is empty, the algorithm signals an error condition and gives up.

When the kernel releases blocks of page frames, it attempts to merge pairs of free buddy blocks of size b together into a single block of size $2b$. If there is no buddy block for the new block of size $2b$, the kernel puts it on the list of blocks of size $2b$. Otherwise, the merging of buddy pairs continues until there are no more buddy pairs to merge. The kernel puts the resulting larger block on the appropriate list of free blocks.

There are alternatives to the buddy-system algorithm, but the developers of the Linux Kernel chose that algorithm for the following reasons. In some cases, contiguous page frames are necessary to satisfy a request, since contiguous linear addresses may not always work. A typical example involves a memory request for buffers to be assigned to a Direct Memory Access (DMA) processor. DMA accesses the address bus directly while transferring several disk sectors in a single I/O operation, thus, the buffers requested must be located in contiguous page frames. Even when contiguous page frame allocation is not strictly necessary, it offers the kernel the big advantage of leaving the paging tables unchanged. This in turn minimizes average memory access times.

2.2 Controlling Fragmentation in Metronome

Bacon, Cheng, and Rajan [1] did work on controlling fragmentation and space consumption in Metronome, a real-time garbage collector for Java. Although the main focus of our work is not real-time garbage collection *per se*, we have similar interests in exploring fragmentation issues. For this reason and the fact that we use a number of garbage collectors in our experiments, we reference Bacon, Cheng, and Rajan [1] as related work.

Metronome is a mostly non-copying real-time garbage collector, which achieves worst-case pause times of 6 milliseconds while maintaining consistent mutator CPU utilization rates of 50% with only 1.5 to 2.1 times the maximum heap space required by the application. Metronome is incremental in nature and is targeted at embedded systems. It is a uni-processor collector that does copying only when defragmentation occurs.

Bacon, Cheng, and Rajan show that Metronome exhibits the following characteristics: provable real-time bounds, time-based and work-based scheduling, and control over internal fragmentation. They also show that Metronome corrects the external fragmentation problem by relocating objects from mostly empty to mostly full pages, if the number of free pages falls below a certain threshold. This is similar in spirit to the defragmentation algorithm we implemented in Buddy Simulator where minimally occupied blocks are freed to allocate larger objects.

2.3 Tailored-List and Recombination-Delaying Buddy Systems

Kaufman introduced two variations of the binary buddy-system that focus on *delayed coalescing* [11] some two decades ago. These variants are Tailored-List Buddy System (TLBS) and Recombination-Delaying Buddy-System (RDBS). We discuss these variants shortly, but we first introduce a few concepts.

- **k-block** - a block of memory of size 2^k ($1 \leq k \leq M$), 2^M is the size of the largest block.
- **k-list** - a list of available blocks of size 2^k.
- **k-buddies** - a pair of k-blocks formed when a (k + 1)-block is split.

Tailored-List Buddy System (TLBS): TLBS tries to maintain the number of blocks on each k-list in accordance with the proportion of requests that are expected to be served from each k-list. It is assumed that the actual requested size-distribution is known *a priori*, so the desired number of blocks on each k-list is established according to this distribution. As a result, the likelihood that the tailored k-list is not empty is increased, and thus fewer searches, splits, and recombinations of k-buddies are necessary. When a pair of k-buddies becomes free as a result of deallocation, the buddies are not necessarily recombined to form a (k + 1)-block. Instead, the k-list's block count is first examined. If the block count is greater than the desired number of blocks, the buddies are recombined to form a (k + 1)-block, which is put on the (k + 1)-list. This recombination process is recursive. Otherwise, no recombination takes place and the k-buddies are put on the rear end of the k-list. Allocations are served from the front end of the list.

Recombination-Delaying Buddy-System (RDBS): The allocation procedure for RDBS is similar to that of TLBS, but deallocation is simpler. Deallocation involves simply returning a freed k-block to its k-list, regardless of the k-block count and the anticipated number of requests from the k-list. RDBS does not recombine blocks during deallocation, but postpones recombination until it receives a request for which no k-block is directly available.

Although RDBS runs as fast as TLBS, when using a lightly loaded system TLBS runs slightly faster. RDBS's high performance is attributed to the dynamic manner in which it determines its k-block count from the size distribution of the blocks being released.

Both variants run faster than the traditional binary buddy-system and thus reduce program execution times. These variants are also as effective as the traditional binary buddy-system in minimizing external fragmentation.

Chapter 3

Background

Because of the significance of the memory management problem, researchers and industry leaders the world over have been expending resources in seeking solutions. There is much debate over whether garbage collection should be used to free up storage or whether program developers should be allowed the flexibility of manually reclaiming storage. Even in the arena of garbage-collected-languages, there is battle over the garbage collection strategies that should be used and the effectiveness of these strategies. Section 3.1 gives an overview of the garbage collection strategies with which we experiment in this project.

Another aspect of the memory management problem that we consider is the set of storage allocation policies that govern the behavior of memory managers. In Section 3.2 we describe some of the most widely used allocation policies [21] including buddy-system [12] policies upon which we base our research. We also discuss defragmentation in Section 3.3.

3.1 Overview of Garbage Collectors

Garbage collection is the process whereby the reclamation of a computer's memory is *automated*. In systems written with languages like C and C++, the programmer is responsible for explicitly reclaiming heap storage using mechanisms like `free` and `delete`, respectively. In some cases, the use of libraries and frameworks help to hide this; however, it is still the programmer's responsibility to deal with deallocation of application's objects. In languages like Java, on the other hand, garbage collection is used to alleviate this burden. The functions of a garbage collector are two-fold [20]. A garbage collector is responsible for finding data objects that are no longer reachable

through pointer traversal. That task involves distinguishing *live* objects from *dead* objects. A garbage collector is also responsible for reclaiming the storage of dead objects so it can be used by the running program. Live objects are objects that are reachable by the running program and therefore may be used in future computation. Dead objects, on the other hand, are those objects that are not reachable by the running program and are referred to as garbage. In the literature the first function of a garbage collector is referred to as *garbage detection*, and the second function is referred to as *garbage reclamation*. In practice, these functions are interleaved in time, and garbage reclamation is usually strongly dependent on the garbage detection mechanisms in use. In our experiments we use the following three garbage collectors: the Java Virtual Machine's [13] Mark and Sweep Garbage Collector [20] (packaged with Sun's 1.1.8 JVM) which we will call JVMGC, a Reference Counting Garbage Collector (RCGC) [20], and a Contaminated Garbage Collector (CGC) [4]. These collectors implement the two-fold functionality described above in their own ways. We give an overview of each collector in subsequent subsections.

3.1.1 Mark and Sweep Garbage Collector

The Mark and Sweep Garbage Collector (MSGC) distinguishes live storage from garbage using a technique known as tracing [20]. Tracing commences from the *root set*[1] and traverses the graph of pointer relationships using either a depth-first or breadth-first approach. The objects that are reached are *marked* [7] in some way, either by altering bits within the object or using a bitmap or other data structure. This constitutes the garbage detection phase of the two-phase functionality described above.

The *sweep* phase of the collector is its garbage reclamation phase. Once the live objects have been determined, their storage is protected, but the storage of the dead objects is reclaimed. The reclamation process involves the exhaustive examination of memory to find all unmarked objects and to declare their storage as free storage that can be used to service other allocation requests. That free storage is linked to one or more free lists accessible to the allocation routine.

One advantage of MSGC is that it is an exact collector *i.e.*, it collects all objects that are unreachable by the running program. It accomplishes this by exhaustively

[1]The root set comprises the variables that are active when the garbage collector is invoked. This includes global variables, local variables in activation records on the activation stack, and variables currently in registers.

searching the heap for garbage. All garbage is swept during the collector's reclamation phase.

While it is a simple scheme, MSGC has some drawbacks. First, detecting live objects has a time complexity of $\Theta(n + l)$ where n is the number of live objects and l is the number of live references. n is usually not known prior to execution, so n is not bounded. The running time of MSGC is also not bounded since it depends on n. This situation is bad for real-time systems since the upper bound on the running time of the collector must be known *a priori*. Another problem with MSGC is that the cost of a collection is proportional to the size of the heap. All live objects must be marked and all dead objects must be collected, imposing serious limitations on the efficiency of MSGC.

3.1.2 Contaminated Garbage Collector

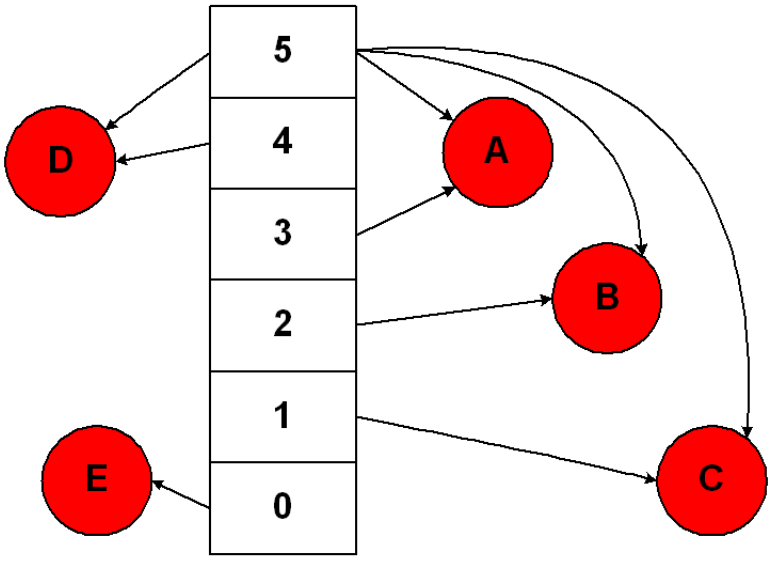

Figure 3.1: Contaminated Garbage Collection

Contaminated garbage collection (CGC) is a new technique for determining when an object can be garbage collected [4]. CGC does not mark live objects as MSGC does, but it dynamically associates each object X with a stack frame M such that X is collectible when M pops. CGC is based on the following property of single-threaded applications: the liveliness of each object X in the heap is due to references that ultimately begin in the program's runtime stack and static areas. X is live until

all frames containing references to it are popped. These references may be direct or indirect. A direct reference is a reference from a stack frame to a heap object as shown in Figure 3.1, but an indirect reference involves at least one intermediate object.

Because of the nature of a stack, the set of frames responsible for the liveness of X contains some frame M that pops last (the oldest frame). Consequently, the liveness of X is tied to frame M so that when M pops, X is collectible.

In Figure 3.1 and Table 3.1, the stack frames are numbered from 0 to 5 with frame 0 being the last and frame 5 being the first frame to pop. Each frame is associated with a method call, and the local variables of each method reside in the method's frame. Frame 0 contains static allocations. The objects labeled A to E reside in the heap and the arrows represent references from the method's local variables to the heap objects. Table 3.1 shows the liveness of the heap objects.

Table 3.1: Liveness of Heap Objects

Object	Referencing Frames	Dependent Frame
A	3, 5	3
B	2, 5	2
C	1, 5	1
D	4, 5	4
E	0	0

One of the features that Figure 3.1 fails to show is that each heap object X potentially has a field $X.x$ that is able to reference other heap objects. Suppose C references A so that $C.c \rightarrow A$. With this new reference A can be collected no earlier than C, and thus A's dependent frame is changed from frame 3 to frame 1. In this case we say C *contaminates* A by touching or referencing it. The effect of contamination is symmetric, *i.e.*, if C contaminates A or A contaminates C the dependent frame of $\{A, C\}$ is the older of the dependent frames of A and C. In Figure 3.1 the dependent frame of this set is frame 1, C's dependent frame.

In terms of the abstract two-phase collector described above, CGC does garbage detection when a heap object is referenced by a variable that resides in the runtime stack or in the static area, or when a heap object contaminates another. Garbage reclamation occurs when the dependent frame of a heap object pops.

CGC possesses some good qualities that are worth mentioning. First, it is feasible to bound the execution time of CGC for detecting live objects because CGC

executes incrementally over the lifespan of the running program. Second, the point of collection for CGC is a stack frame pop. After CGC garbage collects it simply hands over the pointers to the areas most recently collected to appropriate lists of free space. Third, collection of each object has a complexity of $O(1)$.

However, CGC also faces problems that can have serious implications on its efficiency. Overhead is one such problem. The execution of every instruction is associated with some overhead that adds to a program's overall execution time. Although the overhead per instruction is small, large programs can experience noticeable delays in execution. Another problem associated with CGC is its conservativeness in detecting dead objects. Objects may be dead several stack frames earlier but CGC detects them only when their dependent stack frame pops. This is due to the concept of *equilive sets*[2]. From the perspective of CGC, every object in an equilive set has the same lifespan, so every object is collected at the same time when their equilive set's dependent frame pops.

3.1.3 Reference Counting Garbage Collector

In Reference Counting Garbage Collection (RCGC), each object is associated with a count of references to it. When a reference, for example to object X, is created by copying a pointer from one place to another using assignment, X's reference count is incremented. Figure 3.2 illustrates this concept. When an existing pointer to X is eliminated, X's reference count is decremented. The storage X occupies can be reclaimed when X's reference count becomes 0. A reference count of 0 means there are no references pointing to X and, as such, X cannot be reached by the running program.

When an object's storage is reclaimed, its pointers are examined and any object to which it points has its reference count decremented appropriately since references from dead objects don't count in determining whether or not an object is live. Thus, reclaiming one object can potentially lead to reclaiming other objects.

In terms of the two-phase functionality described above, for RCGC the garbage detection phase is equivalent to incrementing and decrementing reference counts and

[2]An equilive set is a set of heap objects whose lifespan is tied to a dependent frame. Initially, every object has its own equilive set but when objects contaminate each other, their equilive sets are merged and are tied to their objects' oldest dependent frame. From CGC's perspective every object in an equilive set dies at the same time.

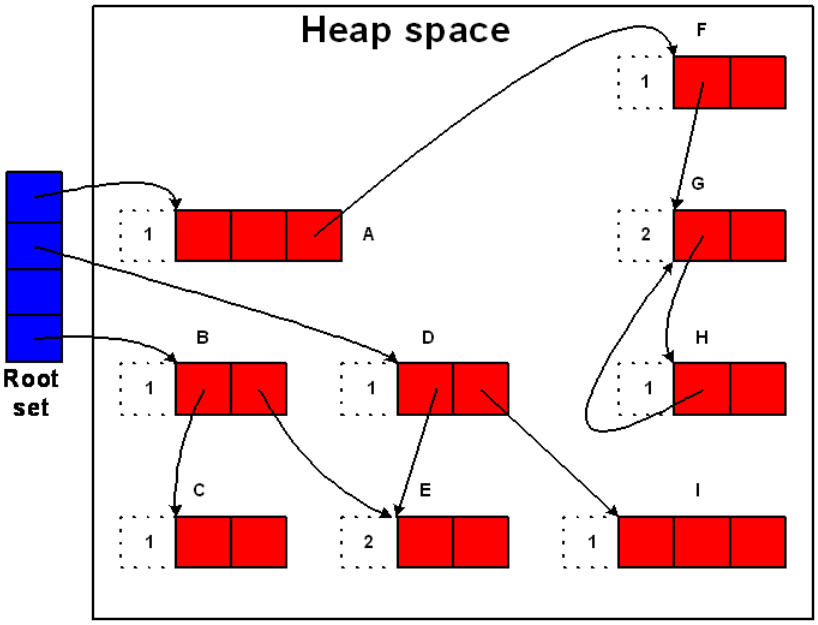

Figure 3.2: Reference Counting Garbage Collection

checking whether reference counts equal 0. The garbage reclamation phase occurs when an object's reference counts become 0.

Like CGC, RCGC's execution is incremental over the lifetime of the program so its running time can be bounded. This is good for real-time systems since RCGC can be utilized as a real-time collector. With RCGC collection is frequent (when stack frames pop) and can be done in constant time.

The main problem with RCGC is its inability to collect objects involved in reference cycles. Consider Figure 3.2. In the heap space, object G references object H and H references G, forming a cycle. Suppose F no longer references G. This causes G's reference count to decrease to one. From that instant neither G nor H is reachable by the program, but RCGC does not collect them because their reference counts are not 0.

3.2 Storage Allocation Mechanisms

The job of a storage allocator is to keep track of memory that is in use and memory that is free so that when a program requests allocation for its objects the allocator is able to satisfy the requests from free storage [21]. An allocator has no control over

the number of requests that are made, nor does it have any authority over the sizes of the blocks requested. These are all up to the program generating the requests. The allocator can only deal with memory that is free and choose where in the free memory an allocation request can be satisfied. It records the address and size of free memory in some hidden data structure, for example a linear list, an ordered tree, a bitmap, or some other data structure.

Since the allocator has no *off-line* knowledge of allocation requests it utilizes an *on-line* algorithm to deal with the requests. This means the allocator can only respond to requests sequentially. The decisions it makes must be immediate and irrevocable.

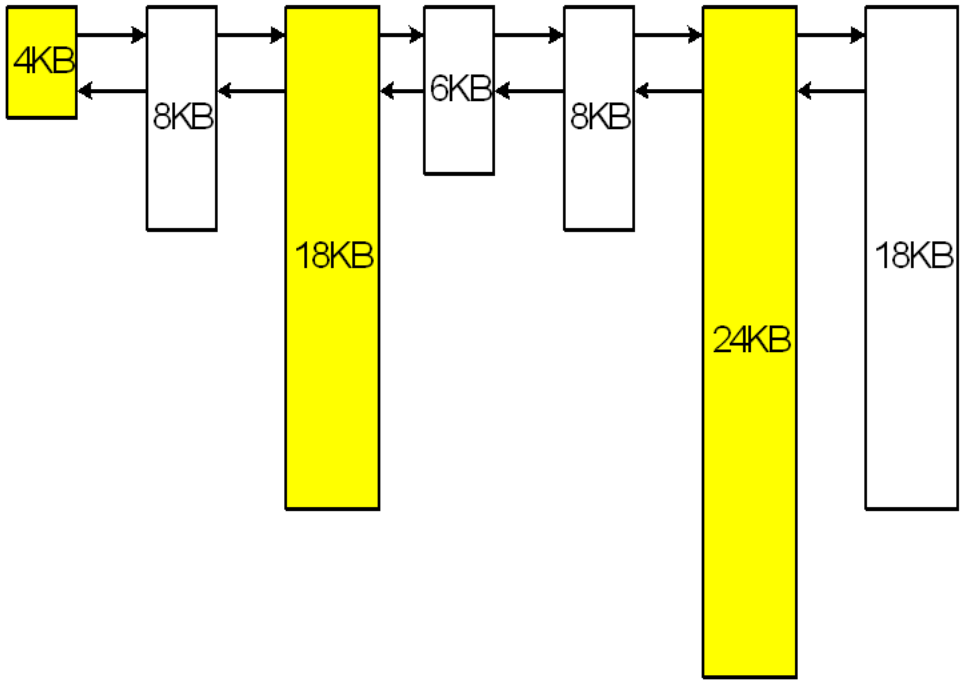

Figure 3.3: Sequential Fit Example: Unshaded blocks are free and shaded blocks contain live objects.

In the following subsections we give a brief overview of some conventional storage allocation mechanisms. The list of mechanisms is extensive but we limit our discussion to sequential fit, segregated free list, and the buddy-system [8] because they are the most common.

3.2.1 Sequential Fit Allocator

Typically, a sequential fit allocator has a single linear list of all free blocks of memory. The list is often doubly-linked or circularly-linked. Figure 3.3 illustrates the architecture of a sequential fit allocator that uses a doubly-linked list to organize its free blocks of memory. The label inside each block shows the actual size of the block in kilobytes. A shaded block is a block with live storage and an unshaded block is a free block that may be allocated. Variants of sequential fit allocators include first fit, next fit, best fit, and worst fit allocators. We describe these variants with the aid of the following example.

Consider Figure 3.3. Suppose a request is made for a 6KB block. We illustrate how each sequential fit allocator attempts to satisfy that request.

First Fit: *First fit* simply searches the list from the beginning until a free block large enough to satisfy the allocation request is found. If the block found is too large to satisfy the request, it is split into two sub-blocks so that one is used to satisfy the request and the other is put on the free list.

The behavior of a first fit allocator is easily demonstrated using the example given above. To satisfy the request a first fit allocator searches the list from the left end until it encounters the leftmost 8KB block, which is the first free block large enough to satisfy the 6KB request. It splits that block into a 6KB block and a 2KB block, uses the 6KB block to satisfy the request, and puts the remaining 2KB block on the list of free blocks.

One of the issues with first fit allocators is the observation that larger blocks near the beginning of the heap tend to be split first and the remaining fragments result in multiple small blocks near the beginning of the heap. This can cause an increase in search time since the search is sequential and these blocks would have to be passed each time a request is made for a larger block. First fit allocators also scale poorly for systems in which many objects are allocated and many different sizes of blocks accumulate. Weinstock discusses other issues with using first fit allocators in [18].

Next Fit: *Next fit* uses a roaming pointer for allocations. The pointer "remembers" the location where the last successful search was satisfied and begins the next search from there. Consequently, successive searches cycle through the free list so that searches do not always begin from the same location. This behavior

has the effect of decreasing average search time when using a linear list. Knuth regards next fit as an optimization over first fit [12].

With regards to the sample problem above, suppose the last successful request resulted in allocating the leftmost 18KB block. The roving pointer is currently at that location. The request to allocate a 6KB block does not cause the search to begin from the front of the heap but from the current location of the pointer even though there is an 8KB block available near the front (left side) of the heap. The search continues rightward until the available 6KB block is reached. That block satisfies the request exactly so no splitting is required.

Although next fit is an optimization over first fit, next fit is not problem-free. The roving pointer cycles through memory regularly causing objects from different *execution phases*[3] to be interspersed in memory. This may adversely affect fragmentation if objects from different phases have different lifetimes.

Best fit: *Best fit* involves exhaustively searching the list from the front until the smallest free block large enough to satisfy a request is found. The basic strategy of best fit allocators is to minimize wasted space so that fragmentation is as small as possible. A study of the worst case performance of these allocators is given in [16]. The request represented above can be satisfied by using the only free 6KB block in Figure 3.3. This block becomes live and the number of free blocks decreases.

One of the problems of best fit allocators is the exhaustive search required to find a block that fits best. Although the search stops if an exact fit is found, the exhaustive nature of the search can be a bottleneck to scalability. This is even more prevalent in heaps with many free blocks. Another problem that can prevail is fragmentation if the fits are very good but not perfect. In this case most of each block will be used, but the remainder might be so small that they might be unusable.

Worst fit: A *worst fit* allocator satisfies the request for a 6KB block by splitting the 18KB block at the end of the list into a 6KB block and a 12KB block. The 6KB block is used to satisfy the request and the 12KB block is put on the list of free storage. In essence, the worst fit allocator searches the list to find

[3] An execution phase occurs when a 'group' of objects is 'born' at the same time and 'dies' at about the same time.

the largest block that can satisfy the allocation request, splits that block to satisfy the request and puts the remaining storage on the list. This avoids the problem of having too many small, unusable fragments as in best fit allocation. One concern with worst fit allocation is the observation that it works poorly in synthetic trace studies, i.e. in studies using traces from simulations. This may be attributed to the tendency to ensure that no very large blocks are available.

3.2.2 Segregated Free List

The idea of segregated free list allocators is that an array of free lists is used to index free storage where each entry in the array points to a different list of blocks of the same size [21]. No two entries point to the same list, and blocks in different lists are of different sizes. Typically, the segregation in these allocators is logical for indexing purposes and not physical. When an executing program requests memory, the list of the requested size block is searched and a block of that size is used to satisfy the request providing that list is not empty. When a block of allocated memory becomes free, that block is returned to the free list of blocks of that size. Most variants of this class of allocators support general splitting and coalescing. We give a brief introduction of a few variants of segregated free list allocators.

Size Class: *Size classes* are used to cluster blocks of similar sizes together so that when a particular size block is requested, the request is serviced from the class containing that size. If the requested size is not on the boundary of its class, the common response of these allocators is to round up the requested size to the upper boundary of its class and to use that much storage to service the request. A variety of different size schemes have been experimented with, for example linear schemes and powers of two schemes, but schemes with smaller spacing are preferred for indexing purposes. Although this type of allocator has the potential to perform well, especially for classes with small spacing, it also has the potential to suffer from internal fragmentation.

Simple Segregated Storage: In *simple segregated storage* allocators, no splitting of larger blocks to satisfy requests for smaller blocks is allowed. Instead, when a free list becomes empty, to service requests for blocks of that size more storage is requested of the underlying operating system (OS). The OS responds by making one or two virtual memory pages available for use. These pages are then split

into blocks of the same size, strung together and put on the free list. Requests for blocks of that size are satisfied from the extended list. This also explains the nomenclature of this type of allocator: each large unit of storage contains blocks of only one size.

One advantage of using simple segregated storage allocators is the fact that no headers are required for each allocated block of a particular size since a per page header is used to record sizes and other relevant information. This saves storage overhead in cases where small objects are allocated. A significant disadvantage with this scheme is its vulnerability to external fragmentation since no attempt is made to split or coalesce blocks to satisfy requests for objects of other sizes.

Segregated Fit: A *segregated fit* allocator is a type of size class allocator that uses an array of free lists where each array holds blocks within a certain class. When a request for a block within a particular class is made, a sequential fit (typically first fit or next fit) search within that class is done to satisfy that request. If no fit for the requested size is available within that class, the list of next larger size blocks is searched. If a block is found, it is split to satisfy the request. If no block is found, however, the list of the next larger size block is searched. This process continues until a free block is found, or the list with blocks of the largest size is exhausted. In this case more memory is requested of the OS.

One of the advantages of segregated fit allocators is search speed. Using multiple lists makes searching for free blocks faster than using a single list. Although this is the case, coalescing of blocks can cause search time to increase. Consider the situation where two free blocks of size X are coalesced to form a larger free block of size $2X$ such that no more free blocks of size X are available. Consider a subsequent request for a block of size X. Instead of finding a free block of size X available, the allocator has to search the list that contains blocks of size $2X$, split one such block into two blocks of size X each, and use one to satisfy the request. Delayed coalescing can help in solving that problem.

3.2.3 Buddy-System

The buddy-system is a variant of segregated free lists that supports restrictive but efficient splitting and coalescing of memory blocks. In the simplest buddy-systems the heap is split into two large portions which are each split into two smaller portions,

and so on. This creates a hierarchical division of memory that constrains where in the heap objects can be placed. Each block size has its unique free list. Thus, when a block of a particular size is requested, a free block from the list with the smallest size blocks large enough to satisfy the request is used. If that block is large enough to be split in two smaller blocks, one of the smaller blocks is used to satisfy the request and the other is put on its free list. In a sense the buddy-system allocator functions like a segregated fit allocator. There are several versions of the buddy system; however, we focus only on the binary buddy-system. The buddy-system has the speed advantage of the segregated fit allocator but internal fragmentation is a concern. For references to resources that describe the buddy-system in more detail, Peterson and Norman [15] serves as a good guide.

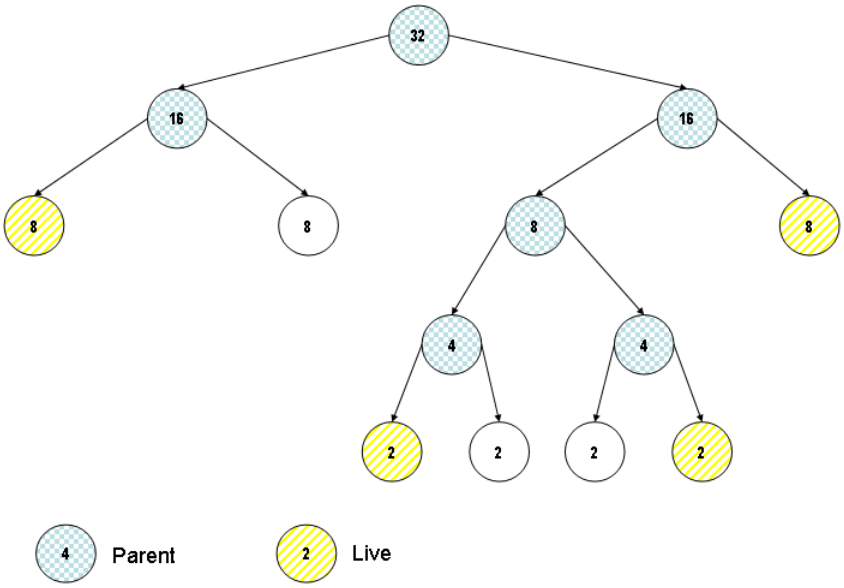

Figure 3.4: Tree Configuration of Free List for Binary Buddy-System

A binary buddy allocator uses an array, a binary tree, or other data structure to maintain free lists of available block sizes 2^i, $0 \leq i \leq m$ where $2^m \leq$ the heap size. When a block of size 2^j is split into two smaller blocks, each small block is of size 2^{j-1}. These small blocks are called buddies and can be merged with each other to form a larger block when they are both free at the same time. It is possible for two adjacent blocks of the same size to fail to be buddies, as Figure 3.4 illustrates. Hence, a means for determining whether two adjacent blocks of the same size are buddies is needed.

Fortunately, a straightforward method is available. Given the memory address and size of a block, only a bit flip is necessary to compute the address of its buddy [12].

Figure 3.4 illustrates the use of a binary tree to maintain the free lists of each block size. Let 2^m be the size of the heap where the unit of storage is a *byte*. Let the root of the tree be at the m^{th} level of the hierarchical division of the heap. Thus, blocks at depth 1 of the tree are at the $(m-1)^{th}$ level of the hierarchy, and blocks at depth i are at the $(m-i)^{th}$ hierarchical division. Given the size of blocks at a level, the level number can be found by taking the base two logarithm of the size. So in Figure 3.4 the root of the tree is at level 5.

We use shadings to differentiate the types of blocks used in the configuration. The unshaded ovals represent blocks that are free and can be allocated. The *live* ovals signify blocks that contain live objects and cannot be allocated. The *parent* ovals stand in place of blocks that once existed but were split into smaller blocks. We use this figure to show the allocation behaviors of the address-ordered binary buddy allocator and the address-ordered best-fit binary buddy allocator.

Address-Ordered Binary Buddy Allocator: An address-ordered binary buddy allocator (AOBBA) is a binary buddy allocator that uses a first fit allocation mechanism as described in Section 3.2.1. Given the heap configuration in Figure 3.4, suppose an allocation request is made for a block of size 2. Either of three blocks, namely two blocks of size 2 at level 1 and one block of size 8 at level 3, can be used to satisfy the request. However, because AOBBA uses a first fit mechanism, the 8 byte block is split repeatedly to satisfy the request. The 2 byte block with the smallest address is used to service that request.

One observation that can be made from this behavior is that the lower end of the heap is heavily used, and the upper end is used only if a request cannot be satisfied from the lower end. Thus, address-ordered binary buddy allocators give preference to the lower end of the heap. In Chapter 4, our presentation of the heap storage requirements for the binary buddy-system is based on the analysis of this mechanism [5].

Address-Ordered Best-Fit Binary Buddy Allocator: Instead of using a first fit mechanism as AOBBA does, an address-ordered best-fit binary buddy allocator (AOBFBBA) uses a best fit algorithm to satisfy a request. Thus, the smallest block large enough to service a request will always be used, regardless of its location in the heap. Notice that preference is given to the lower end of

the heap, hence the address-ordered nature of the mechanism. Finding such a block is not necessarily too expensive since a free list for each block size is maintained. To satisfy the request mentioned above the leftmost 2 byte free block at level 1 is used. This means that the 8 byte block whose address is smaller than that of this 2 byte block is not split. Thus, it can potentially be used to satisfy a subsequent request. Our implementation of the buddy allocator [5] is based on this mechanism.

3.3 Defragmentation

Defragmentation may be defined as the relocation of live objects from their current location in the heap to some other location so that vacated blocks can be coalesced to form larger contiguous blocks. This is a function of the underlying memory allocator, typically performed in response to an allocation request for which no single block of storage is large enough to satisfy, even though there may be sufficient interleaved free memory to satisfy the request. Memory in that condition is fragmented and if no coalescing of free blocks is done, storage would be wasted. Defragmentation seeks to minimize such waste, but not without cost.

One of the cost factors associated with defragmentation is processor time. The less processor time expended on relocating live storage, the more effective the defragmentation algorithm. For real-time and embedded systems defragmentation is discouraged, but for systems that are not in these classes, effective defragmentation algorithms are explored.

The defragmentation algorithm we use in our experiments is a bounded-time algorithm described in [5] that uses a greedy heuristic, which we explain shortly. Suppose allocation for a block of size Z bytes is requested, but there is not enough contiguous free storage to satisfy that request. In an attempt to minimize the cost associated with the defragmentation process, the greedy algorithm searches the heap for a continuous block of storage that is minimally occupied. The objects in this minimally occupied block of storage, the *target block*, are relocated to the smallest blocks outside of that block that can accommodate them. Objects are never relocated from one location within the target block to another location within the same target block. This process is recursive and leads to the coalescing of smaller blocks to form larger blocks. Ultimately the target block becomes a contiguous block of available storage that can be used to satisfy the request.

One reason this algorithm is bounded in time is that the search for a minimally occupied block is not sequential. Rather, it is a tree traversal that commences at the hierarchical division of the requested size. More details on this algorithm can be found in Section 4.4.1 of [5].

3.3.1 Compaction

There are a number of other defragmentation algorithms in use in research and in industry. One of these is compaction. Compaction, as the name suggests, involves compacting one *end*[4] of the heap with live storage while the other end of the heap remains free for allocation. This algorithm does not necessarily model buddy-system properties, although some variants do. Essentially, all the live storage in a fragmented heap slides to one end of the heap, leaving a large continuous region of free storage on the other end. The individual blocks that form this region are coalesced into the largest free block possible that can then be used to satisfy the pending and/or subsequent requests. Compaction can be rather costly since it displaces a significant amount of storage.

[4]Despite the dynamic representaion of the heap via binary tree, etc., the heap itself is ultimately a range of contiguous memory addresses with a lower address bound and an upper address bound. These bounds are considered the ends of the heap.

Chapter 4

Storage Requirement for Binary Buddy Allocator

In his Master's thesis, *Storage Allocation in Bounded Time* [5], Sharath Cholleti proved that for an *Address-Ordered Binary Buddy Allocator* that favors free blocks in lower addresses of the *heap*, $M(\log n + 2)/2$ bytes of storage is necessary and sufficient for that allocator to satisfy an allocation and deallocation sequence without the need for defragmentation (Theorem 3.3.3) [5]. In the expression, $M(\log n + 2)/2$, M represents the *maxlive*: the maximum storage the program requires at any instant during its execution, and n denotes the *max-blocksize*. The max-blocksize is the size of the largest block the program can allocate at any instant during execution. We seek to extend this proof to include the class of binary buddy allocators. Our approach is to show that every variant of the binary buddy allocator can be made to behave like an *Address-Ordered Binary Buddy Allocator* ($AOBBA$) that favors the lower addresses of the heap and has the same storage requirements.

4.1 Notations

We use the notations described below throughout this chapter and the rest of the thesis to discuss the storage usage and behavior of applications. All storage sizes are in bytes.

1. $allocate(i, \mathrm{n}_i)$ denotes the i^{th} allocation request for the application where n_i signifies the block size requested.

2. $deallocate(i)$ signifies the deallocation request for the i^{th} object allocated.

3. $\theta = \{\theta_1, \theta_2, \ldots, \theta_s\}$ denotes an allocation and deallocation request sequence where θ_j represents:

   - $allocate(i, \mathrm{n}_i)$, or
   - $deallocate(i)$.

4. $\log N$ represents $\log_2 N$.

5. $\theta \hookrightarrow \theta'$ signifies an into mapping of θ to θ' such that $|\theta| \leq |\theta'|$ and either of the following is true:

   - $\theta_i \rightarrow \theta'_i$
   - $\theta_i \rightarrow \theta'_j$, $i < j$

6. 'Allocator α satisfies sequence θ' means α is able to find storage for every $allocate(i, \mathrm{n}_i) \in \theta$ and allows the underlying garbage collector to reclaim the storage occupied by $allocate(i, \mathrm{n}_i)$ after $deallocate(i) \in \theta$.

7. Allocation policy, as used in this thesis, refers to a set of rules governing the behavior of a storage allocator. Variants of a particular class of allocators have distinct policies since they emphasize varying behavioral patterns.

4.2 Intuitive Upper Bound

The following theorem shows an intuitive upper bound for the storage requirement for binary buddy allocators.

Theorem 4.2.1 *For every binary buddy allocator α with allocation policy ψ and for every allocation and deallocation sequence θ, there exists θ' such that $\theta \hookrightarrow \theta'$ and α satisfies θ' without defragmentation with no more than $M \log M$ bytes of storage, where M = maxlive, n = max-blocksize, both maxlive and max-blocksize are powers of 2, and $n < M$.*

Proof: This theorem is very similar in spirit to Theorem 3.1.1 from [5] since they both present an upper bound heap size for storage allocators. The difference between this theorem and Theorem 3.1.1 is that this theorem is more general since it targets all binary buddy allocators and not just AOBBA.

Let $K = \log M$ denote the largest number of distinct block sizes that can be allocated. We use proof by contradiction to show that KM bytes of storage is sufficient for satisfying an allocation and deallocation sequence without the need for defragmentation. Suppose KM bytes of storage is not sufficient for satisfying an allocation and deallocation sequence without the need for defragmentation. Commit to each size $k \in \{2^i | 0 \leq i \leq K\}$ a contiguous block of M bytes of storage. For each request for a block of size k bytes the memory manager allocates storage from that size's pool of M bytes. Since the total number of bytes that can be allocated cannot exceed M, i.e., $\sum_{k=2^0}^{2^K} k \leq M$, no more storage is needed by the allocator to satisfy a request of size k bytes. Hence a contradiction. ∎

Theorem 4.2.1 gives an upper bound for the storage that is required by a binary buddy allocator to satisfy allocation and deallocation sequences without the need for defragmentation, but the bound is not a tight upper bound. Programs do not require that much storage and allocators do not make all that storage available to programs. We present storage requirements that are more realistic in subsequent sections, but before we do so, we describe a necessary translation algorithm in Section 4.3.

4.3 Translation Algorithm

Let F be an algorithm that translates allocation and deallocation sequence θ to allocation and deallocation sequence θ' such that $\theta' = \{\theta'_1, \theta'_2, \ldots, \theta'_t\}$, $s \leq t$, $\theta \hookrightarrow \theta'$, and an allocator with allocation policy ψ can satisfy θ' with the same heap size that an allocator with allocation policy β would use to satisfy θ. We give some intuition into what F does in Section 4.3.1.

4.3.1 Intuition for Translation Algorithm

In the description for F, β represents an $AOBBA$ allocation policy and ψ represents any variant of the the class of binary buddy allocation policies. To facilitate our discussion of F, let β signify an $AOBBA$ and let ψ signify a variant of the class of binary buddy allocators. Using these latter representations, we describe the functionality of F.

What F does is the following: F uses the storage needed for β, $f(n)$, to constrain ψ by incrementally generating an allocation and deallocation sequence, θ', that forces ψ to behave as β. Since F is knowledgeable of θ, β, ψ, and $f(n)$, F is able

to generate θ' by using address mapping and pseudo objects. We use an example to illustrate how this is done.

Suppose the next $\theta_i \in \theta$ is a request to allocate a 4 byte block in a partially occupied heap with at least two free blocks large enough to satisfy that request. To satisfy θ_i, β would use the free block at the lower end of the heap. Suppose ψ uses another free block in the middle or other end of the head. F forces ψ to use the same free block as β by creating a pseudo object and allocating it where ψ would allocate θ_i, thus leaving the free block on the lower end of the heap as the only block from which ψ can satisfy θ_i. The pseudo object is then deallocated. The allocation request for the pseudo object, the allocation request for θ_i, and the deallocation request for the pseudo object are all added to θ' in that order. If additional pseudo objects were needed, requests for their allocation and deallocation would also be added to θ' in the same order. The pseudo code detailing the behavior of F follows. Notice that F generates θ' without violating $f(n)$.

4.3.2 Pseudo Code for Translation Algorithm

1. *Input to F*
   θ is an allocation and deallocation sequence
   $s = |\theta|$
   $\beta()$ is a function that takes $\theta_i \in \theta$, $0 \leq i \leq s$, as input and returns its address under allocation policy β.
   $\psi()$ is a function that takes θ_j' as input and returns its address under allocation policy ψ.
   $f(n)$ is the storage needed for β to satisfy θ.

2. *Output from F*
   θ' is an allocation and deallocation sequence
   $t = |\theta'|$

3. *Pseudo Code for F*
   $i = 0$
   $j = 0$
   $bud = f(n)$ /* memory left to satisfy remaining allocation requests */
   while $i \leq s$ and bud !≤ 0 do
   address = $\beta(\theta_i)$

if address == $\psi(\theta_i)$ then
  /* mapping is established */
  $\theta_i \rightarrow \theta_j^{'}$
  /* *bud* is updated by subtracting the size of θ_i if θ_i is an allocation request or by adding the size of θ_i if θ_i is a deallocation request. */
  update *bud*
else
  /* mapping is not established because $\beta(\theta_i)$ and $\psi(\theta_i)$ yield different addresses. We need to allocate "pseudo-objects" to change the allocation policy from ψ to β or to force ψ to mimic the behavior of β . "Pseudo-object" sequence begins at current value of j. */
  pseudo_start = j
  while address $\neq \psi(\theta_i)$ and *bud* $!\leq 0$
    /* add a new allocation request ("pseudo object") to $\theta^{'}$ */
    $\theta_j^{'}$ = new allocation request of size θ_i
    update *bud*
    $j = j + 1$
  end while
  /* mapping is now established */
  $\theta_i \rightarrow \theta_j^{'}$
  update *bud*
  /* "pseudo object" sequence ends at j - 1 */
  pseudo_end = j - 1
  /* now deallocate "pseudo objects"*/
  k = pseudo_start
  while $k \leq$ pseudo_end do
    $j = j + 1$
    $\theta_j^{'}$ = deallocation request for "pseudo object" allocated at $\theta_k^{'}$
    update *bud*
    $k = k + 1$
  end while
  /* all "pseudo objects" are deallocated and $\theta_i \rightarrow \theta_j^{'}$ */
end if
$i = i + 1$

$j = j + 1$
end while
/* the size of θ' is one less than j */
$t = j$ - 1

4.4 Sufficient Storage Requirement

Lemma 4.4.1 *For every binary buddy allocator α with allocation policy ψ and for every allocation and deallocation sequence θ, there exists an allocation and deallocation sequence θ' such that $\theta \hookrightarrow \theta'$ and α satisfies θ' without defragmentation, and with no more than $I(n) = M(\log n + 2)/2$ bytes of storage, where M = maxlive, n = max-blocksize, both maxlive and max-blocksize are powers of 2, and $n < M$.*

Proof: We now prove that $I(n) = M(\log n + 2)/2$ bytes of storage is sufficient to satisfy θ without the need for defragmentation when a binary buddy storage allocator is employed. This proof uses the technique of mapping θ to some θ' using translation algorithm F described in Section 4.3 so that ψ uses the same storage that an $AOBBA$ allocator would use.

Let P be a program that yields an allocation and deallocation sequence $\theta = \{\theta_1, \theta_2, \ldots, \theta_s\}$ when using an $AOBBA$. Suppose P is run on allocator α with allocation policy ψ. Knowing ψ, let F be a translation algorithm, described in Section 4.3, that takes an $AOBBA$ as β and translates θ to θ' such that $\theta' = \{\theta'_1, \theta'_2, \ldots, \theta'_t\}$, $s \leq t$, and $\theta \hookrightarrow \theta'$. F preserves the behavior of P since F is concerned only with allocation and deallocation requests and F does not alter source code. Since each θ_i, and each θ'_j represents an allocation or deallocation request and $\theta \hookrightarrow \theta'$, every $\theta_i \in \theta$, $1 \leq i \leq s$ maps to some $\theta'_j \in \theta'$, $1 \leq j \leq t$. Thus, F possesses the following features.

- Each request contains an object ID; F uses that ID and allocation and/or deallocation address to map θ_i to θ'_j. This follows from Section 4.3.

- In program P, if the allocation of object with ID_k depends on the address of object with ID_{k-r}, $0 < r < k$, $k \leq \mathrm{ID}_{lastObject}$, or on some other address A, F captures that address dependency in a data structure, for example a heap data structure of size $H \leq M$ bytes. We attribute H to overhead since every allocator uses some storage overhead.

- F potentially allocates and reclaims "pseudo-objects" but their storage does not count toward maxlive since P is ignorant of those potential requests. Their storage is regarded, instead, as overhead.

Suppose an $AOBBA$ can satisfy the sequence θ without the need for defragmentation. Then, according to [5], no more than $M(\log n + 2)/2$ bytes of storage is needed since $I(n) = M(\log n + 2)/2$ bytes of storage is sufficient for an $AOBBA$ to satisfy allocation and deallocation request sequence θ without the need for defragmentation, where M is the maxlive and $n < M$ is the max-blocksize. But binary buddy allocator α with allocation policy ψ satisfies θ' without the need for defragmentation with no more than $I(n) = M(\log n + 2)/2$ bytes of storage since $\psi()$ is constrained by $f(n)$, the storage needed for β to satisfy θ and β is an $AOBBA$ policy. Thus, Lemma 4.4.1 holds. ∎

4.5 Necessary Storage Requirement

Lemma 4.5.1 *For every binary buddy allocator α with allocation policy ψ and for every allocation and deallocation sequence θ, there exists an allocation and deallocation sequence θ' such that $\theta \hookrightarrow \theta'$ and α satisfies θ' without defragmentation, and with at least $I(n) = M(\log n + 2)/2$ bytes of storage, where M = maxlive, n = max-blocksize, both maxlive and max-blocksize are powers of 2, and $n < M$.*

Proof: We prove that $I(n) = M(\log n + 2)/2$ bytes of storage is necessary to satisfy the allocation and deallocation sequence θ without the need for defragmentation when a binary buddy storage allocator is used. The technique we use in this proof involves showing the existence of an allocation and deallocation sequence θ that uses $I(n)$ bytes of storage when serviced by an $AOBBA$.

Let P be a program with allocation and deallocation sequence θ when executed on a system with an $AOBBA$. Suppose θ is the result of the following steps.

1. Allocate blocks of size 2^i, $i = 0$, such that $\sum 2^i \leq M$
2. Deallocate every other block
3. Repeat steps 1. and 2. $\forall i$, $0 < i \leq \log n$.

But an $AOBBA$ requires exactly $M(\log n + 2)/2$ bytes of storage to satisfy the sequence θ [5]. If translation algorithm F, described in Section 4.3, is used to translate

θ to some θ' with β being an $AOBBA$ policy, ψ uses exactly $M(\log n + 2)/2$ bytes of storage to satisfy θ'. Thus, there exists a sequence that requires $M(\log n + 2)/2$ bytes of storage. So, for every binary buddy allocator α with allocation policy ψ and for every allocation and deallocation sequence θ, there exists an allocation and deallocation sequence θ' such that $\theta \hookrightarrow \theta'$ and α satisfies θ' without defragmentation with at least $I(n) = M(\log n + 2)/2$ bytes of storage, where M = maxlive, n = max-blocksize, both maxlive and max-blocksize are powers of 2, and $n < M$. ∎

4.6 Necessary and Sufficient Storage Requirement

Theorem 4.6.1 *For every binary buddy allocator α with allocation policy ψ and for every allocation and deallocation sequence θ, there exists an allocation and deallocation sequence θ' such that $\theta \hookrightarrow \theta'$ and α satisfies θ' without defragmentation, and with exactly $I(n) = M(\log n + 2)/2$ bytes of storage, where M = maxlive, n = max-blocksize, both maxlive and max-blocksize are powers of 2, and $n < M$.*

Proof: The proof follows directly from Lemma 4.4.1 and Lemma 4.5.1. ∎

Theorem 4.6.1 provides a tight bound for the storage requirement of any binary buddy allocator. Typically, most applications do not consume that much storage for the allocation of their objects, but for real-time and embedded systems where *defragmentation* should be minimal, $I(n) = M(\log n + 2)/2$ is required. Defragmentation can be costly and unpredictable, thus, the feasibility of its occurrence should be low to guarantee the correctness of these systems.

Chapter 5

Experiments on Coalescing

When a program executes, the underlying memory manager uses a finite heap to allocate storage for objects the program creates dynamically and to reclaim the storage when those objects are no longer reachable. If little or no fragmentation is allowed, according to Theorem 4.6.1 a heap of size $\Theta(M \log M)$ bytes is required. The $\log M$ factor makes the memory system very costly as M grows [5]. To keep the cost of the system from getting too high, defragmentation can be allowed and the heap can be $\Theta(M)$ bytes. We explore the effects of coalescing on defragmentation and program performance using a heap of size M bytes. The defragmentation algorithm we use in this study is presented in [5]. To facilitate experimentation, a few simulators, which are described in subsequent sections are used. One of the key simulators is *Sharath's BuddySystem*, which implements the defragmentation algorithm and coalescing strategies we use in this study.

This chapter is organized as follows. Section 5.1 describes the simulators we employ in conducting experiments. Section 5.2 summarizes the Java benchmarks for which we generate program execution-trace information. Section 5.3 details the methodologies we employ for the experiments we perform with the Java benchmarks, and Section 5.4 explains how we utilize an application that generates a random allocation and deallocation request sequence in our experiments.

5.1 Simulators

The simulators we use to facilitate experimentation are discussed in the subsections below. A flow chart that shows the order in which the simulators are used is given in a subsequent section.

5.1.1 Trace Reader

When the JVM executes a Java program, for example a Java benchmark, and the logfile flag is enabled, a binary output file is saved in the path specified. That file comprises information on allocation and deallocation requests, time analysis, and other pertinent concepts. The *Trace Reader*[1] application extracts specific information from the binary output file and stores it in an ASCII file. For this series of experiments we employ a modified form of the Trace Reader application called *RequestReader*[2]. RequestReader generates two ASCII files: one file consists of allocation and deallocation request information and the other file consists of a summary of the allocation and deallocation request behavior.

5.1.2 Buddy Simulator

Buddy Simulator[3] serves as the key simulator in the experimentation process. The input parameters to Buddy Simulator are listed below.

1. file with the allocation and deallocation request sequence
2. *maxlive* (in bytes)
3. $heapIncrFactor$
4. $[STRATEGY]$
5. $[APPROP\_BLOCK\_HEURISTIC]$
6. $[COALESCING\_COND]$
7. $[COALESCING\_LEVEL]$
8. $[DEBUG]$

$heapIncrFactor$ is multiplied by *maxlive* to determine the size of the heap that is used to service the allocation and deallocation request sequence. The $[COALESCING\_COND]$ parameter is used to indicate whether the coalescing policy should be *prompt-coalescing* or *delayed-coalescing*. Prompt-coalescing is coalescing

[1]The Trace Reader was implemented by Dante Cannarozzi, David Olliges and Conrad Warmbold, and was modified by Dante Cannarozzi, Morgan Deters and Matthew Hampton

[2]This modification was done by Sharath Cholleti

[3]Buddy Simulator is the work of Sharath Cholleti.

right away, coalescing immediately after a deallocation as long as a pair of buddies is free. If delayed-coalescing is the policy specified, then [$COALESCING\_LEVEL$] indicates the level of aggressiveness with which coalescing is to be done. We are concerned with two levels of delayed-coalescing in these experiments: *demand-coalescing* and *thorough-coalescing*. The idea of demand-coalescing is summarized accordingly: whenever there is not a large enough contiguous free block to satisfy an allocation request, coalescing is done *only* to satisfy that request. Thorough-coalescing, on the other hand, is coalescing of the entire heap whenever a contiguous free block is not available to satisfy an allocation request. The [$DEBUG$] parameter is similar in spirit to a verbose output flag that is used to display extra status information.

With these command-line parameters, Buddy Simulator uses an Address-Ordered Best-Fit Allocation strategy to service the allocation and deallocation request sequence contained in the input file. If the heap becomes sufficiently fragmented to the extent that the heap contains enough storage to accommodate an allocation request but no contiguous block is large enough to satisfy that request, the heap is then defragmented. The defragmentation algorithm implemented in Buddy Simulator runs in bounded time and the amount of storage relocated is minimal [5]. A more thorough discussion of the defragmentaion algorithm is found in [5].

Buddy Simulator displays allocation, deallocation, relocation, and coalescing information on standard I/O, and stores the summary of this information in an ASCII output file.

5.1.3 Random Allocation Trace

Random Allocation Trace is an program that simulates the allocation and deallocation behavior of applications. It functions as a trace generator given the complexity of generating as many possible traces as possible with the wish to attain some reasonable coverage on variety of program traces. Figure 5.1 gives an idea of some of the parameters that must be specified when using this application.

The *maxlive* is the sole constraint on the size of the heap and is expected to be a power of two integer in the range $16 \leq maxlive < 2^{31} - 1$. The number of allocations is indicative of the cumulative sum of all the allocation requests the 'application' makes of the memory manager and the percentage bias toward allocation is honored as long as storage is available for satisfying allocation requests.

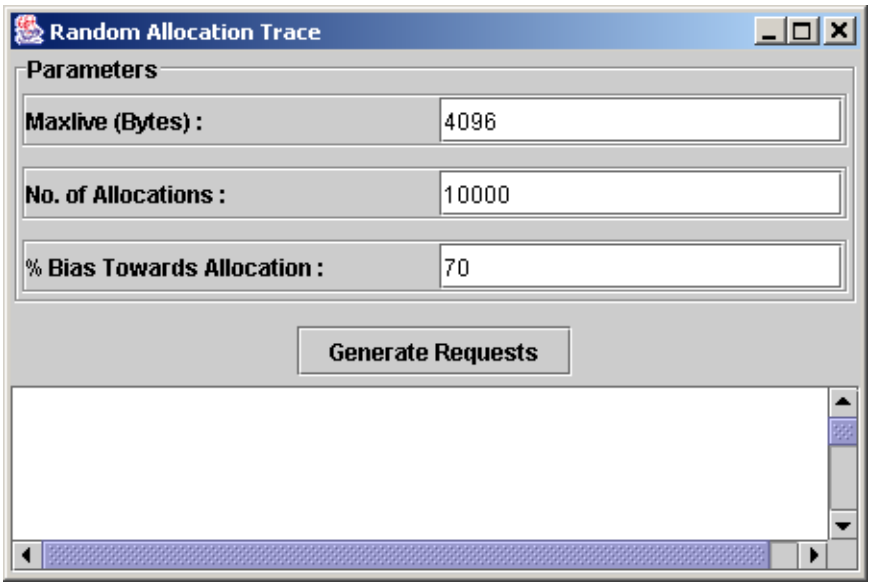

Figure 5.1: Random Allocation Trace User Interface

When the Generate Requests button is clicked, a file dialog pops up to allow the user to specify the path to the file in which the allocation and deallocation request sequence should be stored. After the path is specified, the request sequence is generated and stored in the output file.

Random Allocation Trace emulates an $AOBBA$ but uses randomness in deciding whether the next request should be an allocation request or a deallocation request. The amount of storage to be allocated to the next object and the next object to deallocate are determined randomly also. During execution, a progress monitor is used to aid the user in monitoring the progress of requests generation. At the end of execution, not only is the request sequence stored in the output file, but a summary of the input parameters and the frequency with which the maxlive is used up are also reported.

5.2 Java Benchmarks

The programs we use to obtain allocation and deallocation request sequences are the SPEC JVM98 benchmark suite [6]. There are nine applications in that benchmark suite and we give a brief description of each. More discussion on these applications can be found in [6, 3].

200_check: This benchmark is a simple application that tests and verifies the validity of the Java Virtual Machine (JVM). 200_check's time is not used in the performance metrics, but its output must be validated before performance metrics can be generated. Some of the tests included in 200_check are tests for:

- logical, arithmetic, branch, shift and many other operations
- array indexing beyond its bounds
- creating super classes and their sub classes and checking for method and field access violations

201_compress: 201_compress uses Modified Lempel-Ziv method (LZW). Basically, it finds common substrings and replaces them with a variable size code. This is a deterministic process that can be done on the fly. Thus, the decompression procedure needs no input table, but tracks the way the table was built. The algorithm for this technique is from [19].

201_compress is a Java port of the 129.compress benchmark from CPU95, but improves upon that benchmark in that it compresses real data from files instead of synthetically generated data as in 129.compress.

209_db: This benchmark performs multiple database functions on a memory resident database. It reads in a 1 MB file which contains records with names, addresses and phone numbers of entities, and a 19KB file called *scr6* which contains a stream of operations to perform on the records in the file. The program loops and reads commands until it hits the 'q' command. The commands it performs on the file include, among others:

- add an address
- delete an address
- find an address
- sort addresses

228_jack: 228_jack is a Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS). This is an early version of what is now called JavaCC. The workload consists of a file named jack.jack, which contains instructions for the generation of jack itself. This file is fed to jack so that the parser generates itself multiple times.

213_javac: This is the Java compiler from the JDK 1.0.2.

202_jess: Jess is the Java Expert Shell System based on NASA's CLIPS expert shell system. Jess supports the development of rule-based expert systems (like Eliza) that can be tightly coupled to code written in Java. In simplest terms, an expert shell system continuously applies a set of if-then statements, called rules, to a set of data, called the fact list. The benchmark workload solves a set of puzzles commonly used with CLIPS. To increase run time the benchmark problem iteratively asserts a new set of facts representing the same puzzle but with different literals. The older sets of facts are not retracted. Thus the inference engine must search through progressively larger rule sets as execution proceeds.

222_mpegaudio 222_mpegaudio is an application that decompresses audio files that conform to the ISO MPEG Layer-3 audio specification (developed by Fraunhofer IIS). MP3 is an encoding technique that allows data compression of digital signals up to a factor of 12 without losing sound quality as perceived by the human ear. The workload consists of about 4MB of audio data.

227_mtrt 227_mtrt is a variant of 205_raytrace, a raytracer that works on a pictorial scene depicting a dinosaur. This program uses a multi-threaded driver where the threads render the scene in an input file of size 340KB.

5.3 Experiments with Java Benchmarks

The experiments with the SPEC JVM98 benchmark suite consists of several phases and each phase is illustrated in a figure below. We discuss each phase in its subsection.

5.3.1 Generating Binary Log File

The version of the JVM we use for these experiments is JDK 1.1.8. This virtual machine was instrumented to support the JVMGC, CGC, and RCGC garbage collectors. Figure 5.2 shows the input to the JVM and the output from running benchmark applications on the JVM. By altering the values of ms and mx while keeping them equal, we are able to use a sum-of-binaries technique to determine the minimum heap size (with KB precision) for each benchmark/garbage collector combination. This is

significant since we need the minimum heap size to accurately measure defragmentation. The binary logfile contains, as a minimum, the information requested in the logconf file. For these experiments the requested information consists of allocation and deallocation requests. The logfile parameter is the path to the binary logfile where the JVM stores the output. The sizes associated with each run of each benchmark on each garbage collector are 1% and 100%.

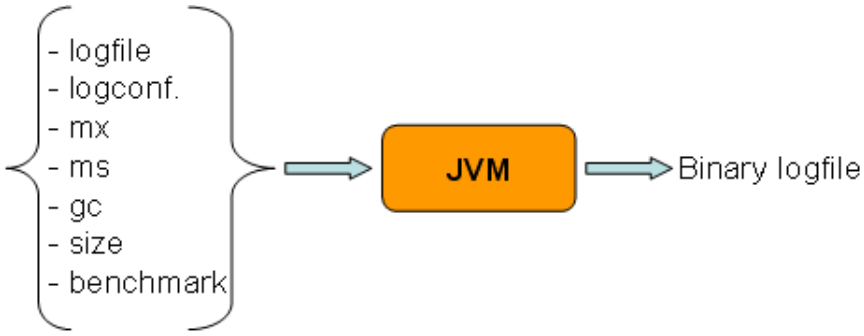

Figure 5.2: Generating Binary Log File

5.3.2 Extracting Allocation and Deallocation Requests

The binary output file generated by the JVM is fed to the RequestReader application, as indicated in Figure 5.3. RequestReader parses the file, extracts the allocation and deallocation requests, and stores them in a 'request' file. Each line of the request file contains an allocation request or a deallocation request. The former is of the the form "1 6 256" where 1 indicates the request is an allocation request, 6 is an example object ID, and 256 is the storage on the heap, in bytes, needed to store that object. The latter is of the form "0 5" where 0 signals a deallocation request and 5 is the ID of the object whose storage is being reclaimed. A deallocation request should not be interpreted as the application requesting of the memory manager that its objects be deallocated because Java applications do not do such. It is, however, the job of the garbage collector to determine when an object is collectible and to reclaim storage from that object. RequestReader also outputs a 'statistics' file with a summary of the allocation behavior of the benchmark.

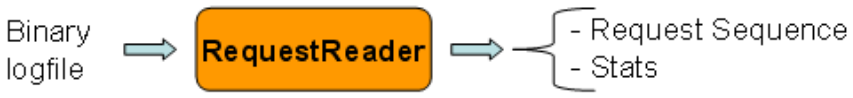

Figure 5.3: Extracting Allocation/Deallocation Requests

5.3.3 Simulating Memory Manager Behavior

Before the request file from RequestReader is fed to Buddy Simulator, the maxlive has to be measured. A Java application named *Maxlive*[4] is used to measure the actual maxlive and the power-of-two maxlive for the requests in the request file. We are concerned with the power-of-two maxlive because Buddy Simulator uses a power-of-two allocation algorithm. With the maxlive determined, the request file, maxlive, and all the parameters outlined in Section 5.1.2 are fed to the simulator as its input parameters. The simulator uses them to model the behavior of the storage allocator mentioned in Section 5.1.2 and presented in [5]. The simulation results are stored in an appropriate ASCII file. We study the results meticulously to gain insight into the effects of coalescing on defragmentation and program performance. Our findings are presented in chapter 6.

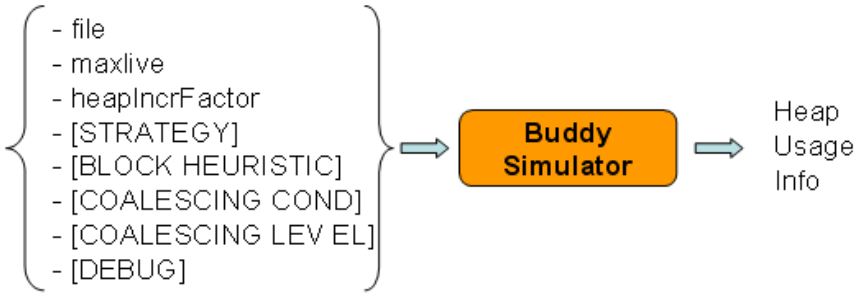

Figure 5.4: Simulating Memory Management

5.4 Experiments with Random Allocation Trace

When executing the Random Allocation Trace application, the maxlive, number of allocations, and percentage bias toward allocation are all entered in their appropriate text fields. The maxlive is expected to be a power-of-two maxlive since Random

[4]Maxlive was implemented by Sharath Cholleti.

Allocation Trace does not verify that it is. The absence of this check is intentional since the user may want to experiment with different types of maxlive. Clicking the 'Generate Requests' button prompts the user to specify the path to the file where the allocation and deallocation request sequence is to be stored. Being satisfied that it has all the necessary parameters to execute, Random Allocation Trace simulates the allocation behavior of a typical application and saves its allocation and deallocation requests in the specified location. That request file, along with the maxlive, and other parameters described above are fed to Buddy Simulator to yield allocation and relocation statistics as described in Section 5.3.3. These results are presented in chapter 6.

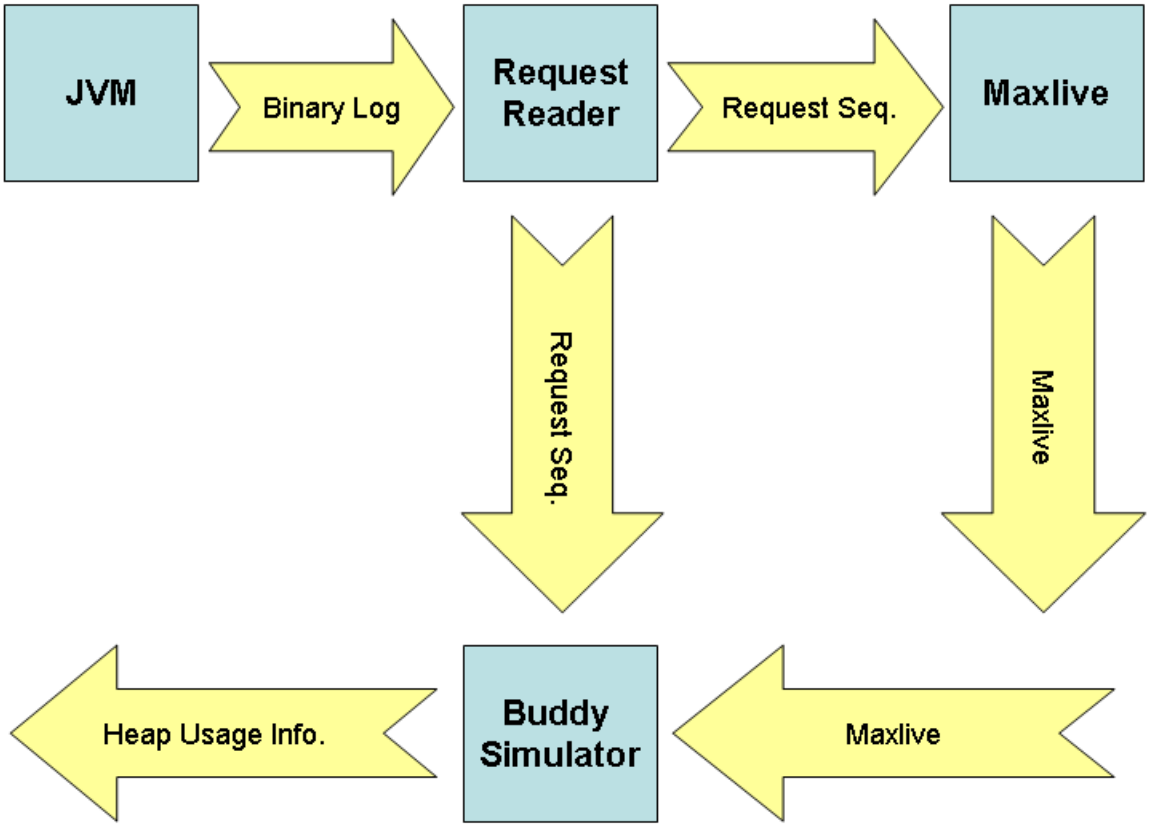

Figure 5.5: Flow Chart of Experiments with Java Benchmarks

Chapter 6

Experimental Results

In chapter 5 we thoroughly described the experiments we performed. We present the results of these experiments and analyze them in this chapter.

6.1 Minimum Heap

Table 6.1: Minimum Heap Size (in KB) for Size 1 of Java Benchmarks

Benchmark	JVMGC	CGC	RCGC
200_check	384	421	344
201_compress	9687	9697	9697
209_db	451	551	407
228_jack	1987	8317	5141
213_javac	894	961	847
202_jess	684	1221	714
222_mpegaudio	507	501	481
227_mtrt	6184	6881	6267
205_raytrace	6184	6881	6267

If an allocator has an unbounded heap to satisfy allocation requests, fragmentation is not an issue and the underlying garbage collector does not need to run. However, the reality is such that memory is a limited resource that must be managed properly. Our objectives for this research were to investigate the effects of coalescing on defragmentation and on programs' performance; as such we believe it was necessary to determine the minimum heap size the Java benchmarks needed when they

were executed on systems with different garbage collectors. Using larger heap sizes would not have allowed us to measure defragmentation accurately since a heap that is too large typically has a free chunk of memory large enough to satisfy an allocation request; hence, requiring no defragmentation. We present the minimum heap sizes for size 1 and size 100 of the Java Benchmarks in Table 6.1 and Table 6.2, respectively. These heap sizes are given with kilobyte precision.

Table 6.2: Minimum Heap Size (in KB) for Size 100 of Java Benchmarks

Benchmark	JVMGC	CGC	RCGC
200_check	384	421	344
201_compress	134367	134367	134367
209_db	11587	127456	11597
228_jack	2000	64520	9041
213_javac	19530	130988	86607
202_jess	1968	22128	5850
222_mpegaudio	507	507	507
227_mtrt	6184	14776	12904
205_raytrace	6184	20400	7747

With the exception of 201_compress size 100 and 222_mpegaudio size 100, for each benchmark the minimum heap size varies from garbage collector to garbage collector. For the most part, JVMGC produced the smallest minimum heap size for the benchmarks. However, RCGC outperformed JVMGC for 209_db size 1, 200_check (both sizes), 213_javac size 1, and 222_mpegaudio size 1. CGC produced the largest minimum heap sizes.

These differences in heap size can be attributed to the varying characteristics of the garbage collectors. JVMGC is the least conservative of the collectors since it has the potential to collect every dead object. However, JVMGC runs least frequently. RCGC runs incrementally but has difficulty reclaiming objects involved in reference cycles. CGC is the most conservative collector since object contamination causes younger equilive sets to shrink and older equilive sets to grow, resulting in more objects being collected later.

6.2 Number of Coalescings

Coalescing buddies consumes system resources just as any other memory operation. Some of the costs associated with coalescing buddies include verifying that two neighboring blocks of the same size are buddies, verifying that the buddies are indeed free, merging free buddy pairs, adjusting the lists that buddies belonged to before merging, and adjusting the lists that resulting larger blocks belong to after merging. These costs are overhead and should be kept at a minimum.

Figure 6.1, Figure 6.2, and Figure 6.3 show the effects of delayed coalescing on number of coalescings for size 100 of Java benchmarks. These figures reveal that regardless of which garbage collector is used, for each Java benchmark prompt coalescing does the most coalescings of buddies. Thorough coalescing reduces number of coalescings significantly; thus, indicating that the number of times prompt coalescing coalesces buddies is unnecessary.

The savings derived from using thorough coalescing with JVMGC instead of prompt coalescing varies in the range of 3.8 to 2.3 million. For 222_mpegaudio, prompt coalescing merged buddies 3.8 times as many as did thorough coalescing and for 209_db, prompt coalescing merged buddies as many as 2.3 million times as many as thorough coalescing did. Demand coalescing yielded even more savings for most of the benchmarks. However, for 200_check and 222_mpegaudio thorough coalescing and demand coalescing coalesced buddies the same number of times. 201_compress and 209_db experienced at least 1000 coalescings with prompt coalescing but zero with both thorough coalescing and demand coalescing.

The results for CGC are similar to those for JVMCG but there is no difference in savings between thorough coalescing and demand coalescing. The largest savings experienced by the delayed coalescing strategies was 652000 , and the benchmark for which these savings were observed was 202_jess. The smallest savings, 3.6, was recorded for 200_check. 201_compress and 209_db did not coalesce buddies when delayed coalescing strategies were used.

RCGC produced results that differ in several respects to the results from both JVMGC and CGC. While most of the benchmarks witnessed no difference between thorough coalescing and demand coalescing, 228_jack, 213_javac, and 202_jess did. No benchmark experienced zero coalescing with the delayed coalescing strategies and the best savings CGC received from delayed coalescing was limited to 59000.

From these three garbage collectors, 209_db and 201_compress benefited most from delayed coalescing and 222_mpegaudio and 200_check benefited least. The findings for size 1 of the benchmarks are similar to those of size 100 and are captured in Figure A.1, Figure A.2, and Figure A.3.

6.3 Memory Relocation - Defragmentation

Defragmentation - relocation of live objects can be very expensive and has the potential to be problematic. Some of the costs involve searching the address space for blocks large enough to accommodate objects subject to relocation, deallocating those objects, and reallocating them. These actions can result in cache inconsistency if the caching mechanism is *write back* and not *write through*. To avert these consequences, defragmentation should be minimal. Table 6.3 shows the effects of delayed coalescing on defragmentation for size 100 of Java benchmarks.

From among the three garbage collectors we employed, JVMGC had the largest margin of percentage of memory relocated for prompt coalescing, followed by CGC, then RCGC. JVMGC also recorded the largest margin for thorough coalescing and demand coalescing. Typically, the percentage of memory relocated increased from prompt coalescing to thorough coalescing and from thorough coalescing to demand coalescing, but the increase is not significant. The largest increase witnessed was 0.38532% and it was witnessed by 228_jack. It is interesting to note, also, that for some benchmarks, delayed coalescing introduced no defragmentation across garbage collectors and for another group, no defragmentation with a particluar garbage collector. 209_db and 201_compress were not adversely affected by delayed coalescing, but 228_jack was affected worst since delayed coalescing caused 228_jack to relocate the largest proportion of live storage.

The results for size 1 of the of Java benchmarks are slightly different from the results for size 100, but for the most part, they are comparable. Some of the differences observed are the following: CGC recorded the largest margin of percentage of memory relocated for thorough coalescing and demand coalescing, the largest percentage increase was almost 1%, and no benchmark enjoyed zero relocation across garbage collectors. These results can be verified from Table A.7, Figure 6.4, Figure 6.5, and Figure 6.6.

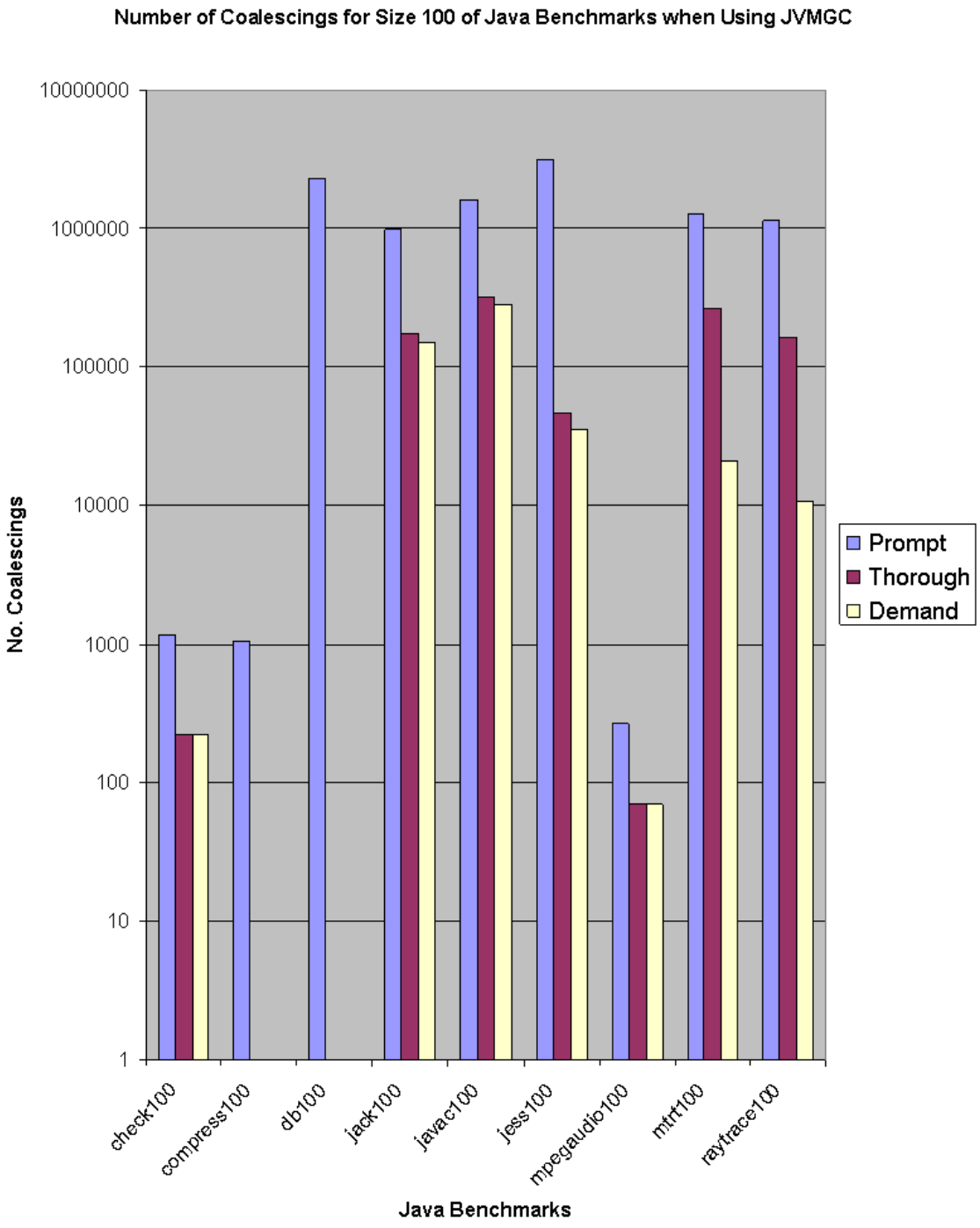

Figure 6.1: Number of Coalescings vs Coalescing Strategy when Using JVMGC - Benchmarks are of Size 100

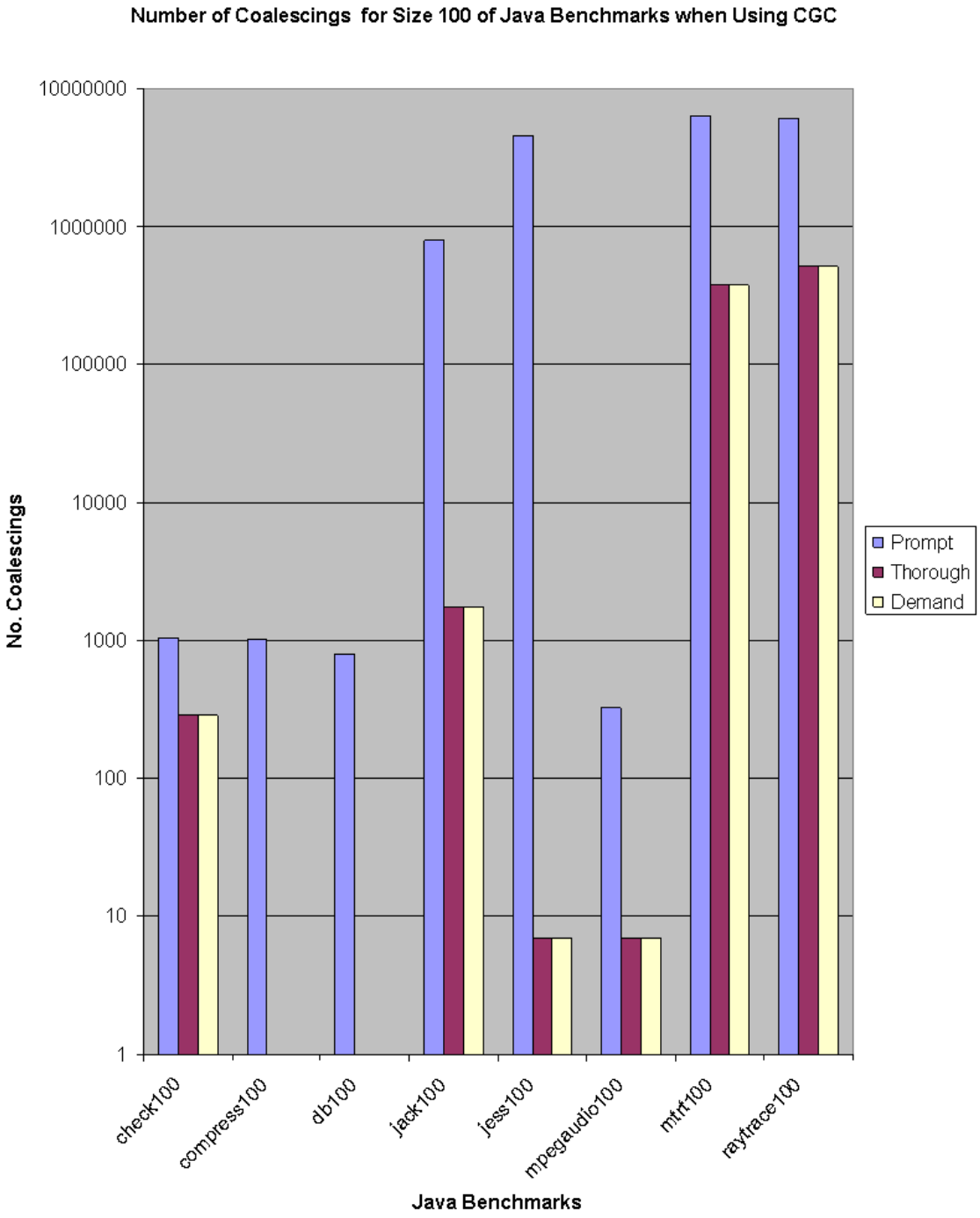

Figure 6.2: Number of Coalescings vs Coalescing Strategy when Using CGC - Benchmarks are of Size 100

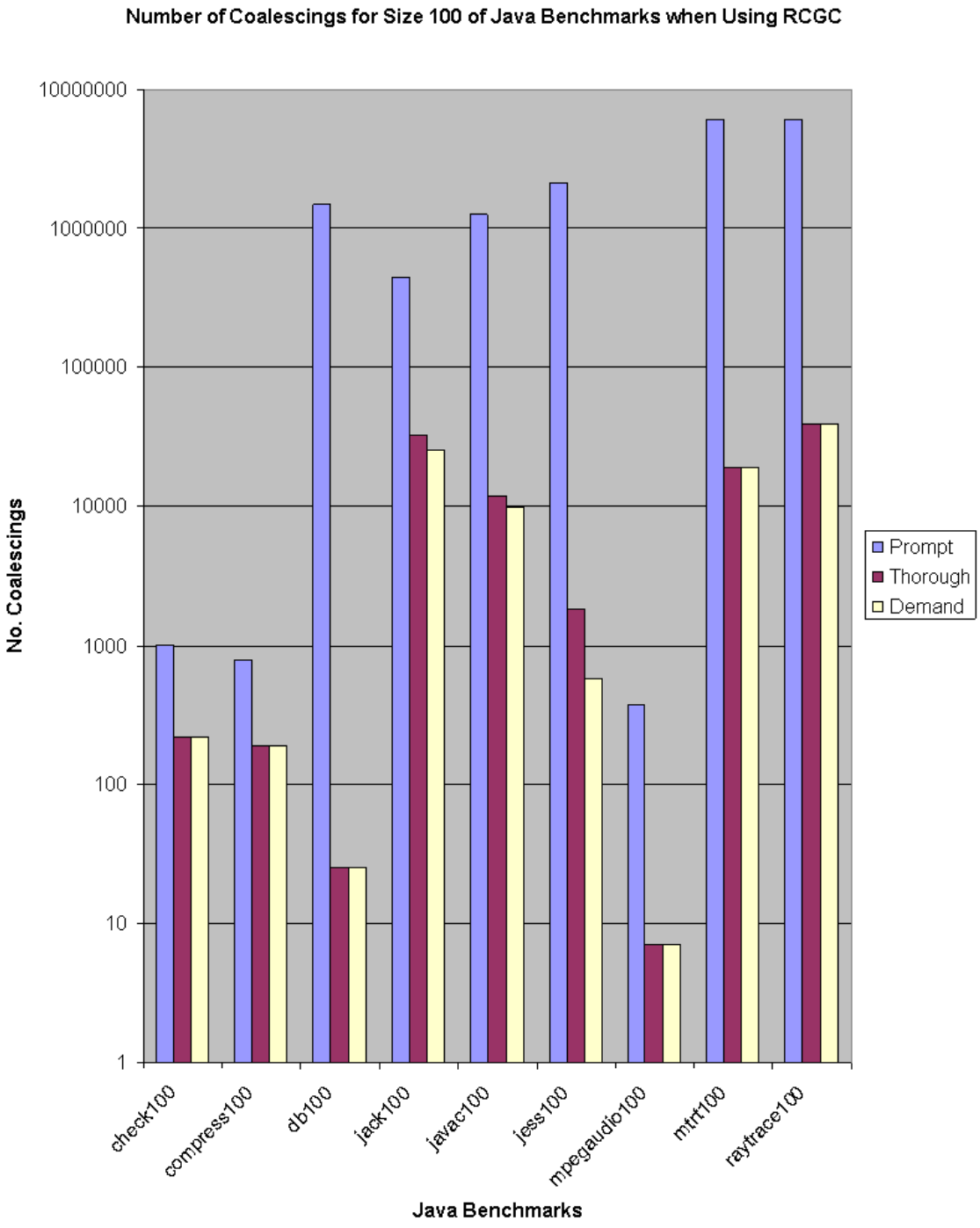

Figure 6.3: Number of Coalescings vs Coalescing Strategy when Using RCGC - Benchmarks are of Size 100

Table 6.3: Percentage of Memory Relocated for Size 100 of Java Benchmarks

Benchmark	Garbage Collector	% Memory Relocated	Garbage Collector	% Memory Relocated	Garbage Collector	% Memory Relocated	Coalescing Strategy
200_check	JVMGC	0.0	CGC	0.0	RCGC	0.00143	Prompt
	JVMGC	0.21274	CGC	0.08138	RCGC	0.08567	Thorough
	JVMGC	0.21274	CGC	0.08138	RCGC	0.08567	Demand
201_compress	JVMGC	0.0	CGC	0.0	RCGC	0.0	Prompt
	JVMGC	0.0	CGC	0.0	RCGC	0.0	Thorough
	JVMGC	0.0	CGC	0.0	RCGC	0.0	Demand
209_db	JVMGC	0.0	CGC	0.0	RCGC	0.0	Prompt
	JVMGC	0.0	CGC	0.0	RCGC	0.00002	Thorough
	JVMGC	0.0	CGC	0.0	RCGC	0.00002	Demand
228_jack	JVMGC	0.21050	CGC	0.0	RCGC	0.01269	Prompt
	JVMGC	0.20455	CGC	0.05370	RCGC	0.09148	Thorough
	JVMGC	0.58987	CGC	0.05370	RCGC	0.12976	Demand
213_javac	JVMGC	0.0	CGC	N/A	RCGC	0.00230	Prompt
	JVMGC	0.17458	CGC	N/A	RCGC	0.05478	Thorough
	JVMGC	0.17879	CGC	N/A	RCGC	0.05437	Demand
202_jess	JVMGC	0.00018	CGC	0.0	RCGC	0.00002	Prompt
	JVMGC	0.01626	CGC	0.0	RCGC	0.00012	Thorough
	JVMGC	0.01890	CGC	0.0	RCGC	0.00012	Demand
222_mpegaudio	JVMGC	0.0	CGC	0.03941	RCGC	0.0	Prompt
	JVMGC	0.07020	CGC	0.03941	RCGC	0.00246	Thorough
	JVMGC	0.07020	CGC	0.03941	RCGC	0.00246	Demand
227_mtrt	JVMGC	0.0	CGC	0.00019	RCGC	0.00011	Prompt
	JVMGC	0.0	CGC	0.00026	RCGC	0.00037	Thorough
	JVMGC	0.0	CGC	0.00026	RCGC	0.00037	Demand
205_raytrace	JVMGC	0.0	CGC	0.00009	RCGC	0.0	Prompt
	JVMGC	0.00404	CGC	0.00010	RCGC	0.00031	Thorough
	JVMGC	0.00404	CGC	0.00010	RCGC	0.00031	Demand

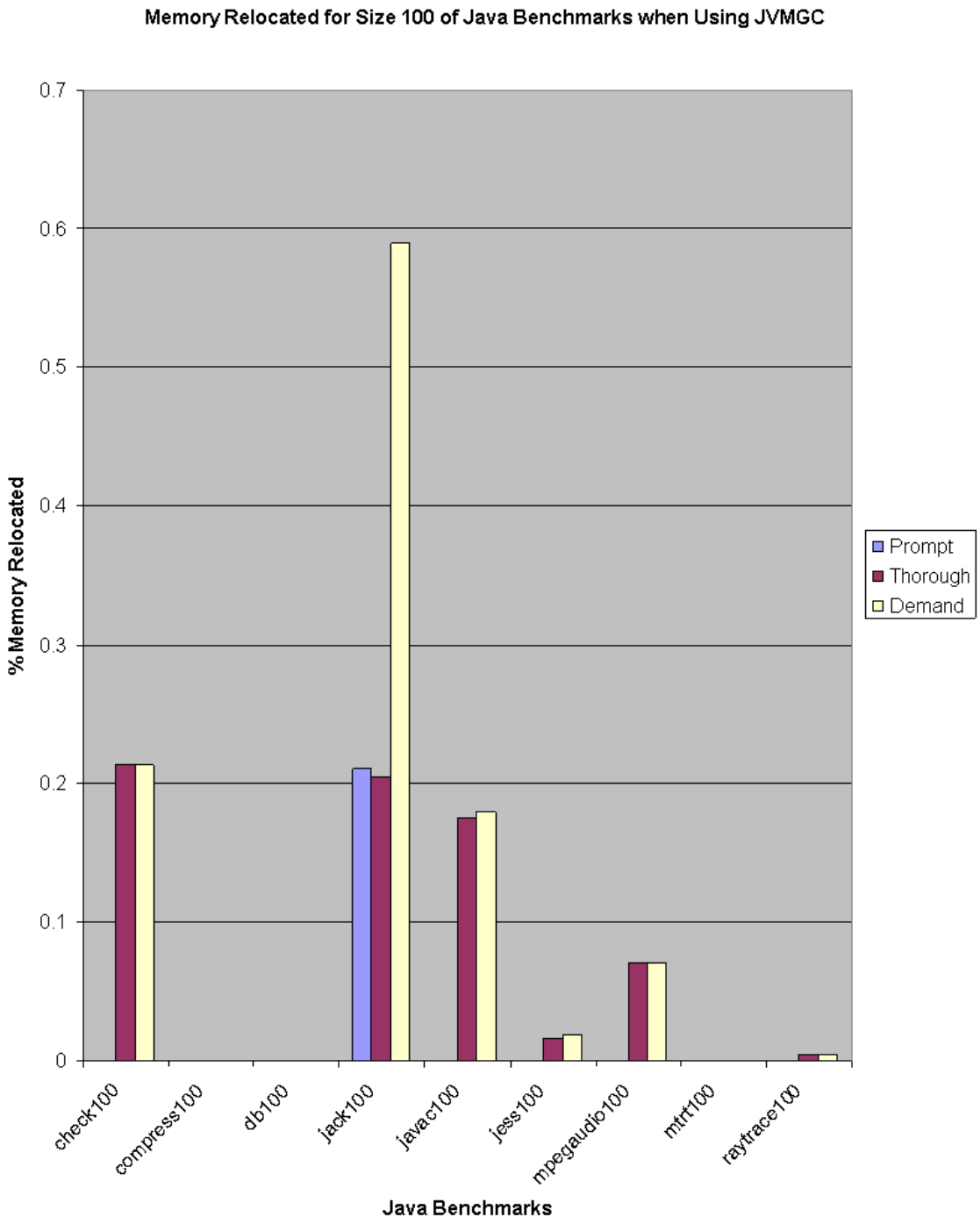

Figure 6.4: Percentage of Memory Relocated for Size 100 of Java Benchmarks when Using JVMGC

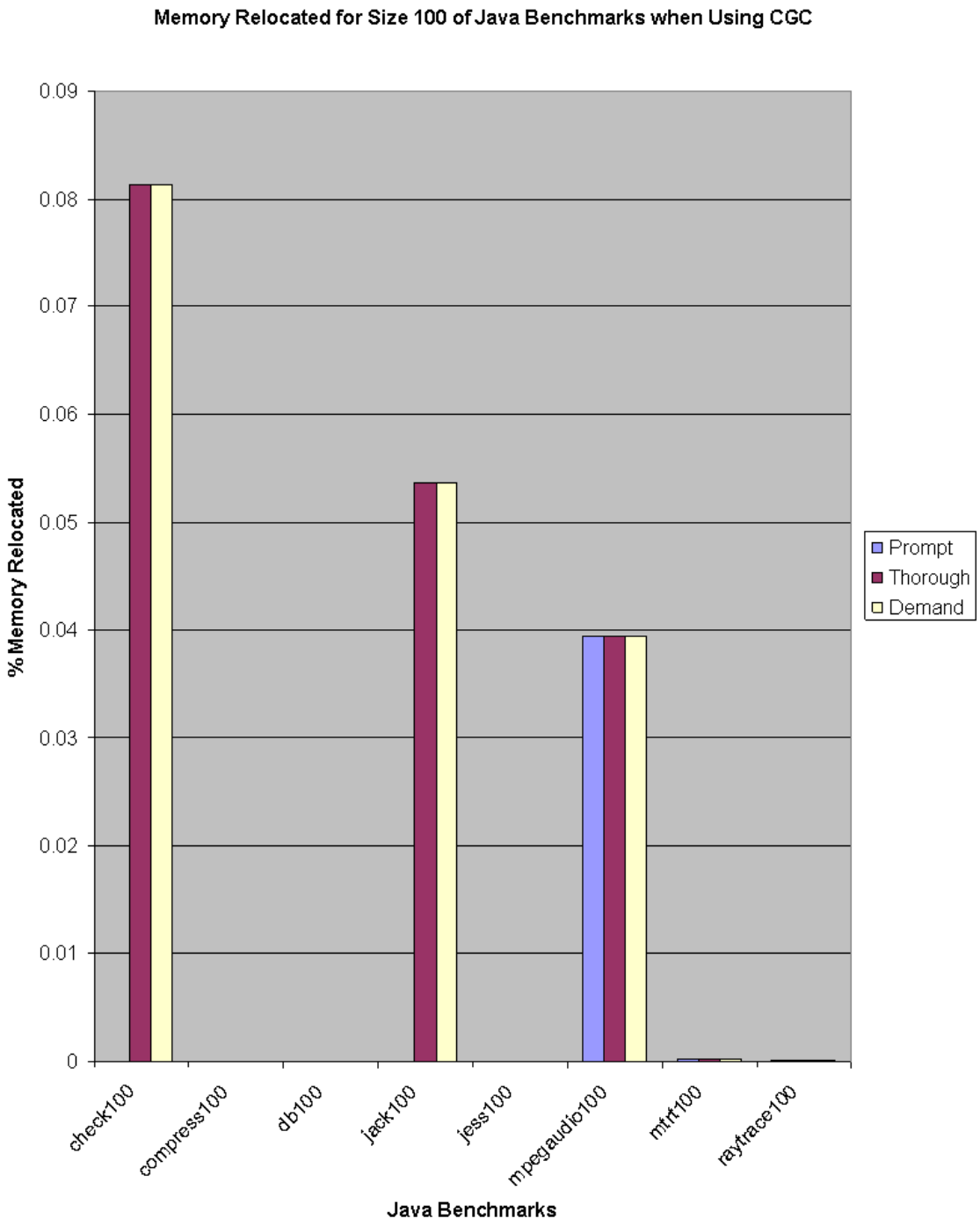

Figure 6.5: Percentage of Memory Relocated for Size 100 of Java Benchmarks when Using CGC

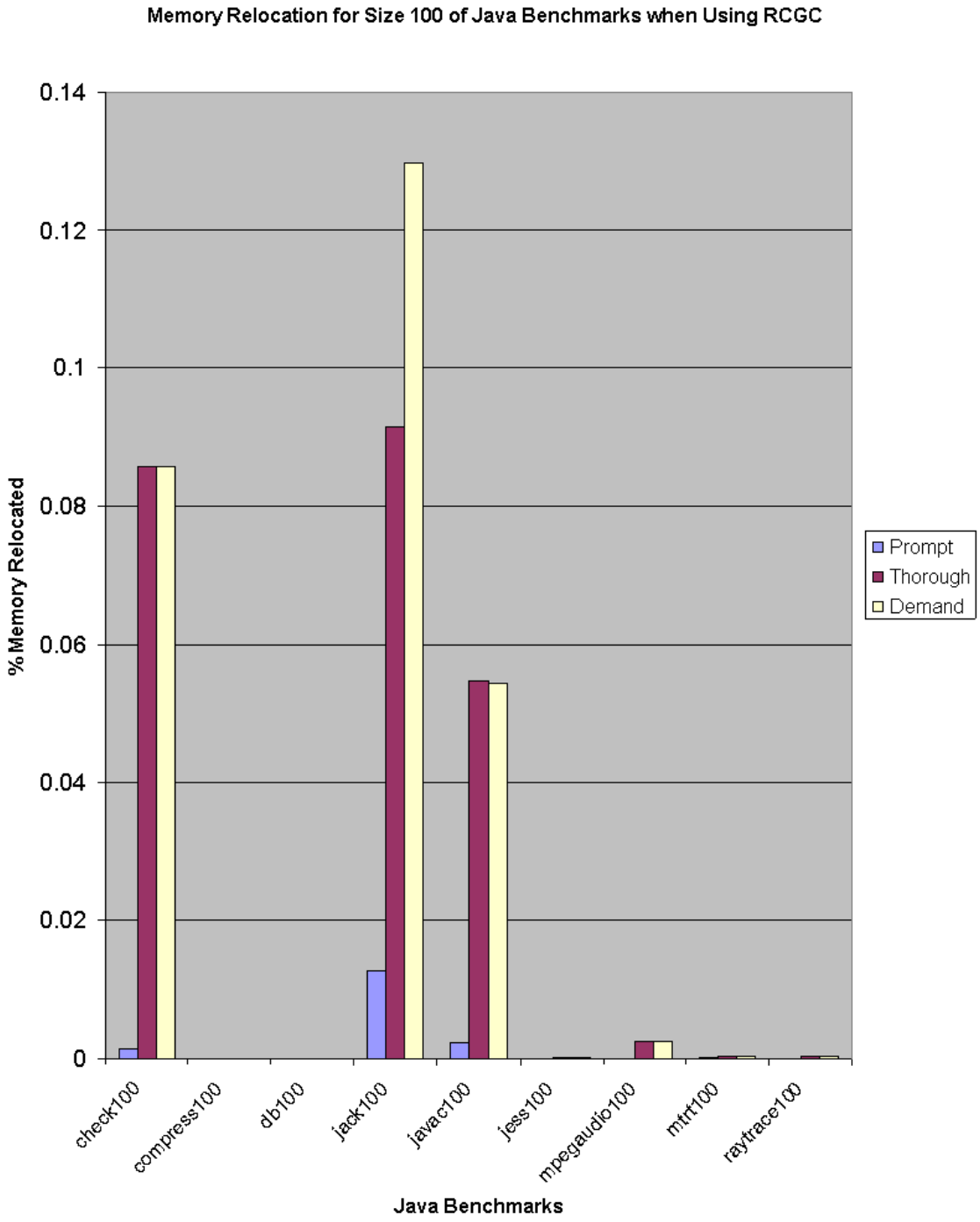

Figure 6.6: Percentage of Memory Relocated for Size 100 of Java Benchmarks when Using RCGC

6.4 Results from Synthetic Traces

We were suspicious about the results we obtained from the Java benchmarks, Section 6.3, because those programs seemed to be very 'well-behaved'. They allocate a majority of 'small' objects and few 'large' objects. Their allocations are many and their deallocations are few. These programs also allocate objects in groups and deallocate them in groups. The 'well-behaved' nature of those programs caused them to not significantly fragment the heap. This prompted us to use synthetic traces to study the allocation behavior of programs that are not so 'well-behaved'. In this venture we utilized the Random Allocation Trace Application (RATA).

RATA was used to simulate the execution of real programs. We used a maxlive of 4194304 bytes (4 MB) and an allocation bias of 70% for generating those traces. Figure 6.7 shows the number of objects created for each run of RATA. Table A.8 summarizes the memory usage statictics for those traces. FIgure 6.8 highlights the effects of the coalescing strategies on the number of coalescings and Figure 6.9 demonstrates how the coalescing strategies affect defragmentation.

Those traces show significantly more defragmentation that the Java benchmarks. The percentages of memory relocated for the Java benchmarks are upper bounded by 1% but for those traces they are upper bounded by 6%. Besides this observation, the effects of delayed coalescing on defragmentaion for those traces are similar to what we observed for the Java benchmarks.

Another observation from using those traces we found interesting was that delayed coalescing had negligible effects on number of coalescings. This suggests that the coalescings that were done when using prompt coalescing were necessary since the delayed coalescing strategies do not coalesce buddy-pairs until they have to.

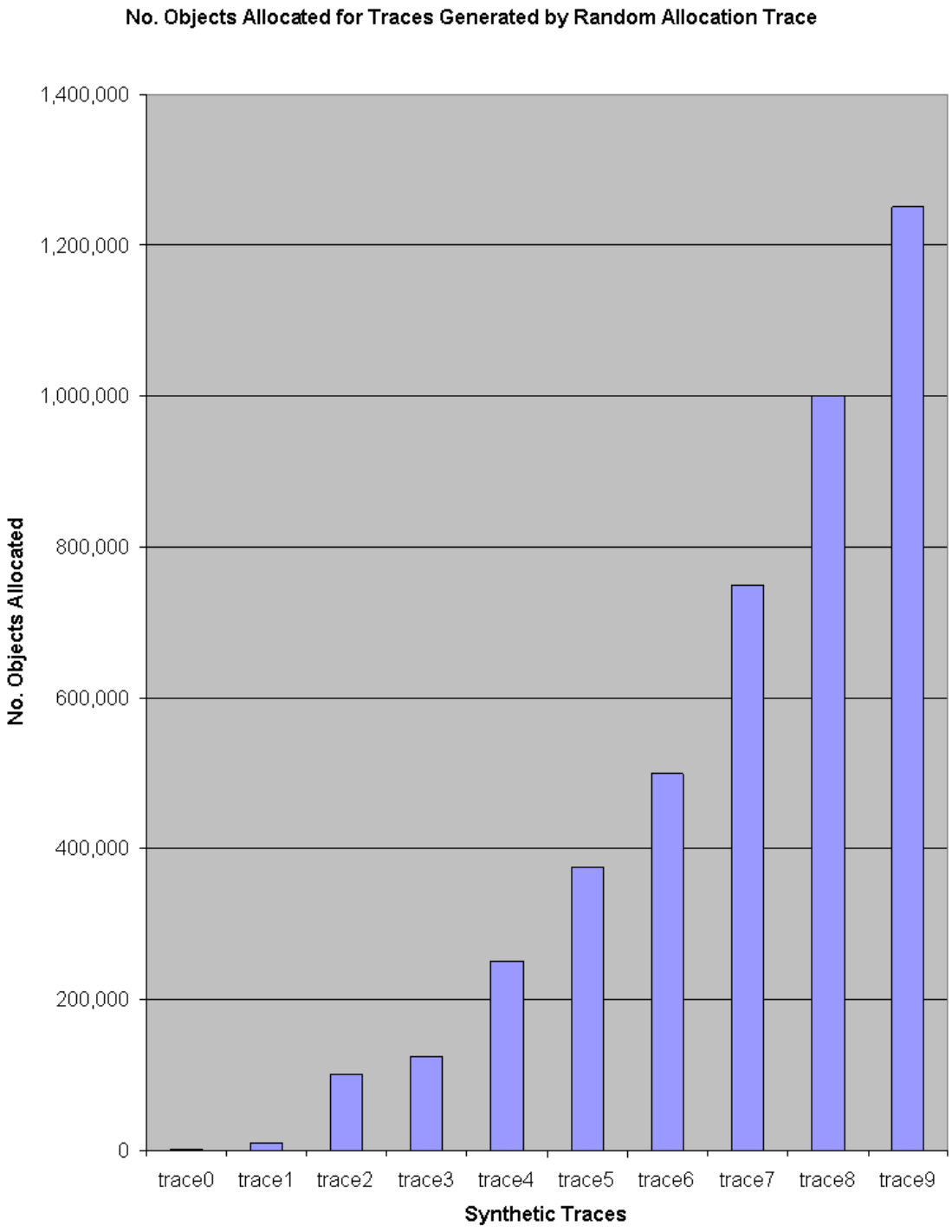

Figure 6.7: Total Number of Objects Created by Each Run of The Synthetic Trace Application

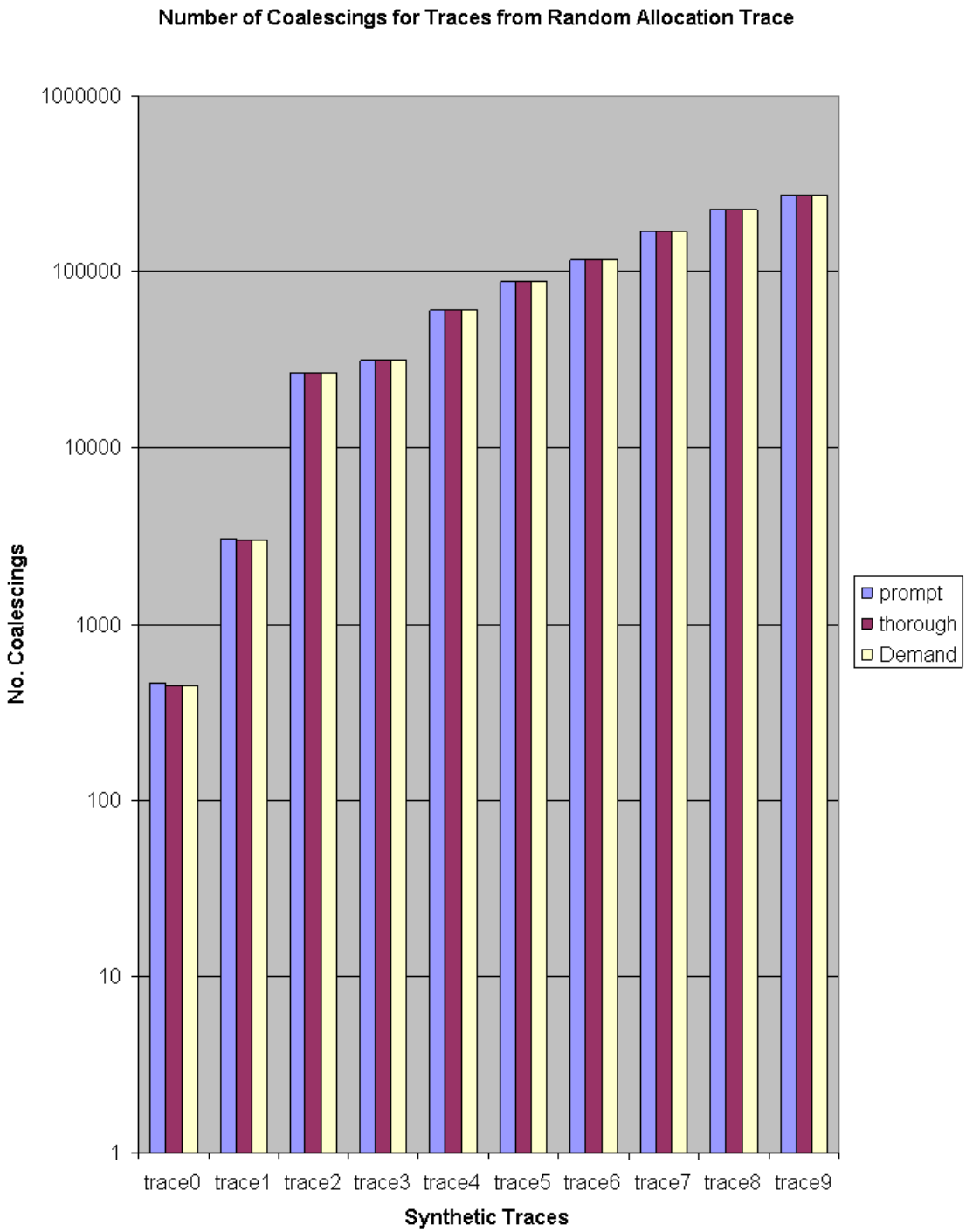

Figure 6.8: Number of Coalescings vs Coalescing Strategy when Using Synthetic Traces

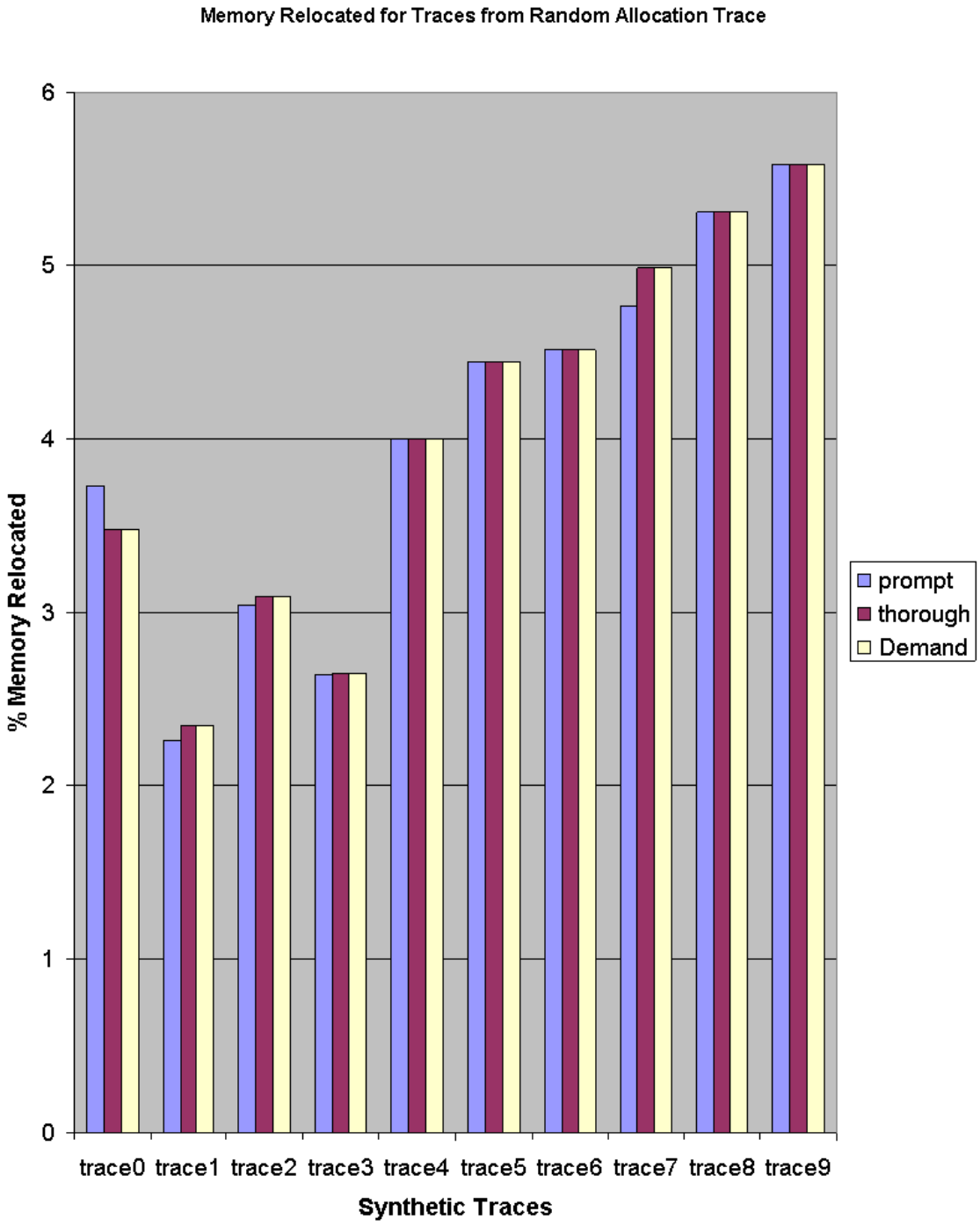

Figure 6.9: Percentage of Memory Relocated for Each Run of The Synthetic Trace Application

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In his thesis, *Storage Allocation in Bounded Time* [5], Sharath Cholleti proved that $M(\log n + 2)/2$ bytes of storage is necessary and sufficient for an Address-Ordered Binary Buddy Allocator to satisfy an allocation and deallocation sequence without the need for defragmentation. We expanded this proof to the class of Binary Buddy Allocators by showing that the behavior of any such allocator can be modified to mimic the behavior of an Address-Ordered Binary Buddy Allocator. Our approach involved designing an algorithm that translates allocation and deallocation request sequences generated by an Address-Ordered Binary Buddy Allocator to allocation and deallocation request sequences that would require no more storage if used with some other Binary Buddy Allocators. Our algorithm would need to know the other allocator in order to tailor the sequence to that allocator.

We also instrumented Sun's Java Virtual Machine to use a Contaminated Garbage Collector and a Reference Counting Garbage Collector, in addition to Sun's default Mark and Sweep Garbage Collector. We ran the SPEC JVM98 benchmark suite [6] on the instrumented JVM to obtain traces that allowed us to study the effects of delayed coalescing on defragmentation and on the overall performance of programs. From the results of Buddy.Simulator we discovered that delayed coalescing reduced number of coalescings remarkably over prompt coalescing (by a factor of the order of 2.6 million), resulting in significant reduction in overhead. While this finding is motivating, it is not always free. Delayed coalescing can also introduce a slight increase in defragmentation. For the SPEC JVM98 benchmarks, we noticed that the largest increase is less than 1%. We also experimented with some synthetic traces and

our findings were different from what we observed for the SPEC JVM98 benchmark suite. We found that delayed coalescing did not reduce the number of coalescings and significantly more defragmentation was observed in those traces.

7.2 Future Work

From analyzing the results from our experiments, we observed that in some cases demand coalescing reduces number of coalescings more than thorough coalescing does, while in other cases the reduction is the same. Demand coalescing also has the potential to defragment more than thorough coalescing. We seek to use this knowledge to enhance Buddy.Simulator look for allocation and deallocation patterns and to use those patterns to optimize its performance. Performance optimization would involve finding a balance between reducing number of coalescings and minimizing defragmentation.

We would also like to implement Buddy.Simulator in the JVM so we can thoroughly test it to determine how efficient it functions as a storage allocator. Testing it would allow us to measure how long it takes to coalesce buddies and how long it takes to relocate live objects. These measurements would expose any unforeseen factors.

We would also be interested in exporting these ideas to file systems. Much research has gone into developing file systems. However, we believe we can make some contribution to that research by offering an alternative to the systems already deployed.

Appendix A

Supporting Data for Experiments

The tables and figures below contain either raw data or supporting data for the experiments performed for this thesis. The captions and/or legends describe the data contained in each table or figure.

Section A.1 contains data for size 100 of the Java benchmarks; Section A.2 contains data for size 1 of the Java benchmarks, Section A.3 contains Memory Relocation data for size 1 of the Java benchmarks, and data for the traces from the Random Allocation Trace application are stored in Table A.8 of Section A.4.

A.1 Data for Size 100 of Java Benchmarks

A.1.1 JVM Garbage Collector

Table A.1: Memory Usage Statistics for Size 100 of Java Benchmarks when Using JVMGC

Benchmark	maxlive	Memory Allocated	Memory Relocated	No. of Relocations	No. of coalescings	Coalescing Strategy
200_check	481720	560296	0	0	1180	Prompt
	481720	560296	1192	73	219	Thorough
	481720	560296	1192	73	219	Demand
201_compress	166520200	166520200	0	0	1052	Prompt
	166520200	166582520	0	0	0	Thorough
	166520200	166582520	0	0	0	Demand
209_db	9403296	72632816	0	0	2299943	Prompt
	9403296	72632816	0	0	0	Thorough
	9403296	72632816	0	0	0	Demand
228_jack	2277880	37517608	78976	3674	984634	Prompt
	2277880	37517608	76744	1788	172537	Thorough
	2277880	37517608	221304	6904	148536	Demand
213_javac	18096880	147267016	0	0	1636547	Prompt
	18096880	147267016	257104	8772	319967	Thorough
	18096880	147267016	263296	9039	281274	Demand
202_jess	2244624	283031528	512	6	3154507	Prompt
	2244624	283031528	46032	1287	46355	Thorough
	2244624	283031528	53488	1511	35651	Demand
222_mpegaudio	638976	649592	0	0	265	Prompt
	638976	649592	456	27	69	Thorough
	638976	649592	456	27	69	Demand
227_mtrt	4377872	118076944	0	0	1258232	Prompt
	4377872	118076944	0	0	264288	Thorough
	4377872	118076944	0	0	21098	Demand
205_raytrace	4378064	112743896	0	0	1134884	Prompt
	4378064	112743896	4560	243	163236	Thorough
	4378064	112743896	4560	243	10745	Demand

A.1.2 Contaminated Garbage Collector

Table A.2: Memory Usage Statistics for Size 100 of Java Benchmarks when Using CGC

Benchmark	maxlive	Memory Allocated	Memory Relocated	No. of Relocations	No. of coalescings	Coalescing Strategy
200_check	515280	560360	0	0	1038	Prompt
	515280	560360	456	16	286	Thorough
	515280	560360	456	16	286	Demand
201_compress	166485968	166582648	0	0	1028	Prompt
	166485968	166582648	0	0	0	Thorough
	166485968	166582648	0	0	0	Demand
209_db	70642104	72765200	0	0	795	Prompt
	70642104	72765200	0	0	0	Thorough
	70642104	72765200	0	0	0	Demand
228_jack	8023736	37542424	0	0	783804	Prompt
	8023736	37542424	20160	1182	1748	Thorough
	8023736	37542424	20160	1182	1748	Demand
202_jess	4523992	283154664	0	0	4561983	Prompt
	4523992	283154664	16	1	7	Thorough
	4523992	283154664	16	1	7	Demand
222_mpegaudio	617984	649624	256	2	324	Prompt
	617984	649624	256	6	7	Thorough
	617984	649624	256	6	7	Demand
227_mtrt	9221488	118076944	224	4	6359732	Prompt
	9221488	118076944	304	5	373357	Thorough
	9221488	118076944	304	5	373357	Demand
205_raytrace	12837392	112743896	96	3	6136837	Prompt
	12837392	112743896	112	3	517372	Thorough
	12837392	112743896	112	3	517372	Demand

A.1.3 Reference Counting Garbage Collector

Table A.3: Memory Usage Statistics for Size 100 of Java Benchmarks when Using RCGC

Benchmark	maxlive	Memory Allocated	Memory Relocated	No. of Relocations	No. of coalescings	Coalescing Strategy
200_check	403080	560296	8	1	1021	Prompt
	403080	560296	480	18	217	Thorough
	403080	560296	480	18	217	Demand
201_compress	166426944	166582648	0	0	795	Prompt
	166426944	166582648	8	1	187	Thorough
	166426944	166582648	8	1	187	Demand
209_db	9096664	72632816	0	0	1486494	Prompt
	9096664	72632816	16	1	25	Thorough
	9096664	72632816	16	1	25	Demand
228_jack	2574848	37516296	4760	130	438138	Prompt
	2574848	37516296	34320	896	32679	Thorough
	2574848	37516296	48680	1268	25193	Demand
213_javac	29733936	145647976	3352	137	1253813	Prompt
	29733936	145647976	79792	3522	11780	Thorough
	29733936	145647976	79192	3496	9829	Demand
202_jess	1849000	283037192	56	3	2144614	Prompt
	1849000	283037192	328	17	1844	Thorough
	1849000	283037192	328	17	587	Demand
222_mpegaudio	595000	649624	0	0	372	Prompt
	595000	649624	16	1	7	Thorough
	595000	649624	16	1	7	Demand
227_mtrt	6960048	118076944	128	1	6060775	Prompt
	6960048	118076944	432	20	18989	Thorough
	6960048	118076944	432	20	18989	Demand
205_raytrace	4304992	112743896	0	0	6087513	Prompt
	4304992	112743896	352	22	39127	Thorough
	4304992	112743896	352	22	39127	Demand

A.2 Data for Size 1 of Java Benchmarks

A.2.1 JVM Garbage Collector

Table A.4: Memory Usage Statistics for Size 1 of Java Benchmarks when Using JVMGC

Benchmark	maxlive	Memory Allocated	Memory Relocated	No. of Relocations	No. of coalescings	Coalescing Strategy
200_check	481688	560264	0	0	1169	Prompt
	481688	560264	1216	72	220	Thorough
	481688	560264	1200	72	220	Demand
201_compress	9984112	10020984	0	0	238	Prompt
	9984112	10020984	1248	45	137	Thorough
	9984112	10020984	1248	45	137	Demand
209_db	579968	778056	0	0	2764	Prompt
	579968	778056	0	0	0	Thorough
	579968	778056	0	0	0	Demand
228_jack	2267336	11785824	0	0	317982	Prompt
	2267336	11785824	26528	841	27454	Thorough
	2267336	11785824	26528	841	27454	Demand
213_javac	1056904	1549448	0	0	8113	Prompt
	1056904	1549448	0	0	0	Thorough
	1056904	1549448	0	0	0	Demand
202_jess	846640	2357616	0	0	24166	Prompt
	846640	2357616	0	0	0	Thorough
	846640	2357616	0	0	0	Demand
222_mpegaudio	638904	649512	0	0	258	Prompt
	638904	649512	424	26	67	Thorough
	638904	649512	424	26	67	Demand
227_mtrt	4377728	6627768	0	0	209523	Prompt
	4377728	6627768	0	0	0	Thorough
	4377728	6627768	0	0	0	Demand
205_raytrace	4378040	6676872	0	0	208831	Prompt
	4378040	6676872	0	0	0	Thorough
	4378040	6676872	0	0	0	Demand

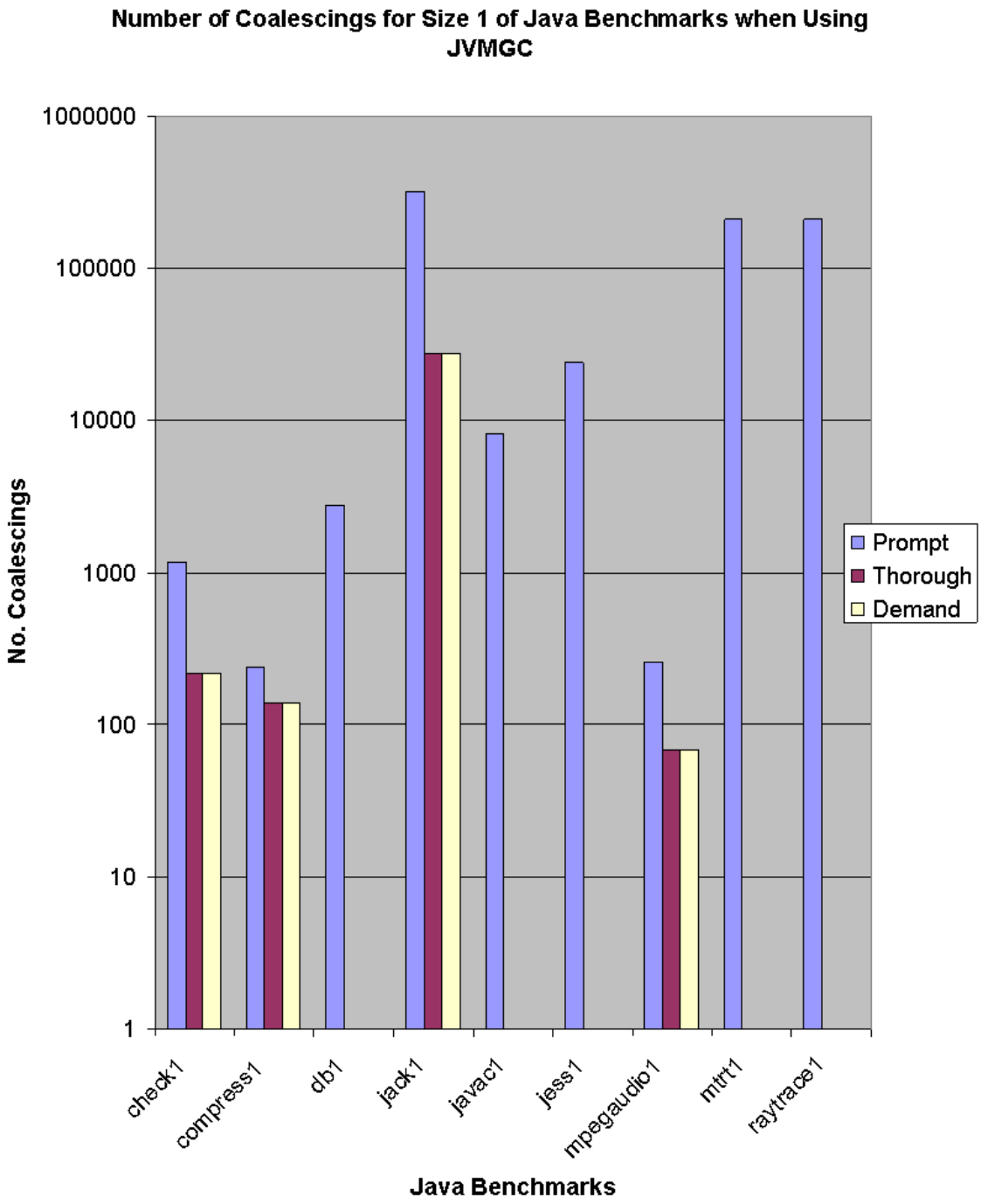

Figure A.1: Number of Coalescings vs Coalescing Strategy when Using JVMGC - Benchmarks are of Size 1

A.2.2 Contaminated Garbage Collector

Table A.5: Memory Usage Statistics for Size 1 of Java Benchmarks when Using CGC

Benchmark	maxlive	Memory Allocated	Memory Relocated	No. of Relocations	No. of coalescings	Coalescing Strategy
200_check	515280	560328	0	0	1030	Prompt
	515280	560328	456	16	286	Thorough
	515280	515280	456	16	286	Demand
201_compress	9985504	10020984	0	0	391	Prompt
	9985504	10020984	0	0	0	Thorough
	9985504	10020984	0	0	0	Demand
209_db	716424	778088	0	0	2413	Prompt
	716424	778088	5720	324	1381	Thorough
	716424	778088	5848	332	1384	Demand
228_jack	2668952	11785824	0	0	253960	Prompt
	2668952	11785824	112	4	2874	Thorough
	2668952	11785824	112	4	1340	Demand
213_javac	977696	1549448	200	6	2596	Prompt
	977696	1549448	200	6	9	Thorough
	977696	1549448	200	6	9	Demand
202_jess	1101696	2357232	56	5	24711	Prompt
	1101696	2357232	56	5	278	Thorough
	1101696	2357232	56	5	278	Demand
222_mpegaudio	618144	649512	256	2	343	Prompt
	618144	649512	256	6	7	Thorough
	618144	649512	256	6	7	Demand
227_mtrt	3991904	6627720	96	3	231769	Prompt
	3991904	6627720	112	3	171385	Thorough
	3991904	6627720	112	3	171385	Demand
205_raytrace	3992328	6676776	96	3	232435	Prompt
	3992328	6676776	112	3	171385	Thorough
	3992328	6676776	112	3	171385	Demand

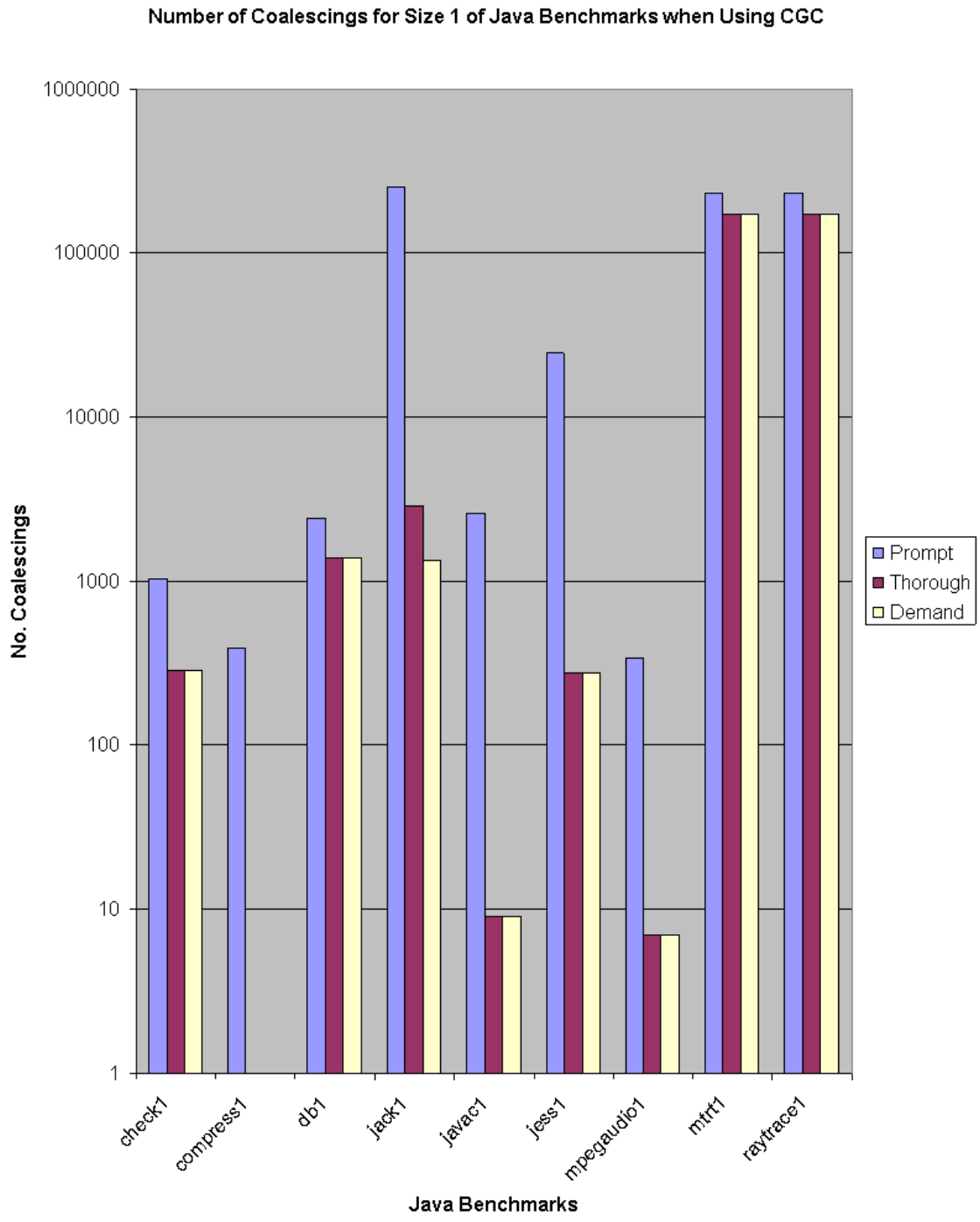

Figure A.2: Number of Coalescings vs Coalescing Strategy when Using CGC - Benchmarks are of Size 1

A.2.3 Reference Counting Garbage Collector

Table A.6: Memory Usage Statistics for Size 1 of Java Benchmarks when Using RCGC

Benchmark	maxlive	Memory Allocated	Memory Relocated	No. of Relocations	No. of coalescings	Coalescing Strategy
200_check	403080	560264	8	1	1028	Prompt
	403080	560264	480	18	217	Thorough
	403080	560264	480	18	217	Demand
201_compress	9972320	10020984	0	0	332	Prompt
	9972320	10020984	32	2	7	Thorough
	9972320	10020984	32	2	7	Demand
209_db	501248	778056	8	1	2346	Prompt
	501248	778056	3512	162	774	Thorough
	501248	778056	3800	180	762	Demand
228_jack	1955600	11785824	0	0	165505	Prompt
	1955600	11785824	16	1	264	Thorough
	1955600	11785824	16	1	28	Demand
213_javac	856024	1549448	0	0	2692	Prompt
	856024	1549448	16	1	2	Thorough
	856024	1549448	16	1	2	Demand
202_jess	639144	2357200	296	8	17047	Prompt
	639144	2357200	520	10	108	Thorough
	639144	2357200	712	13	89	Demand
222_mpegaudio	595032	649512	0	0	395	Prompt
	595032	649512	16	1	7	Thorough
	595032	649512	16	1	7	Demand
227_mtrt	3697472	6627768	0	0	42313	Prompt
	3697472	6627768	0	0	1168	Thorough
	3697472	6627768	0	0	145	Demand
205_raytrace	3697704	6676920	0	0	42679	Prompt
	3697704	6676920	0	0	1168	Thorough
	3697704	6676920	0	0	145	Demand

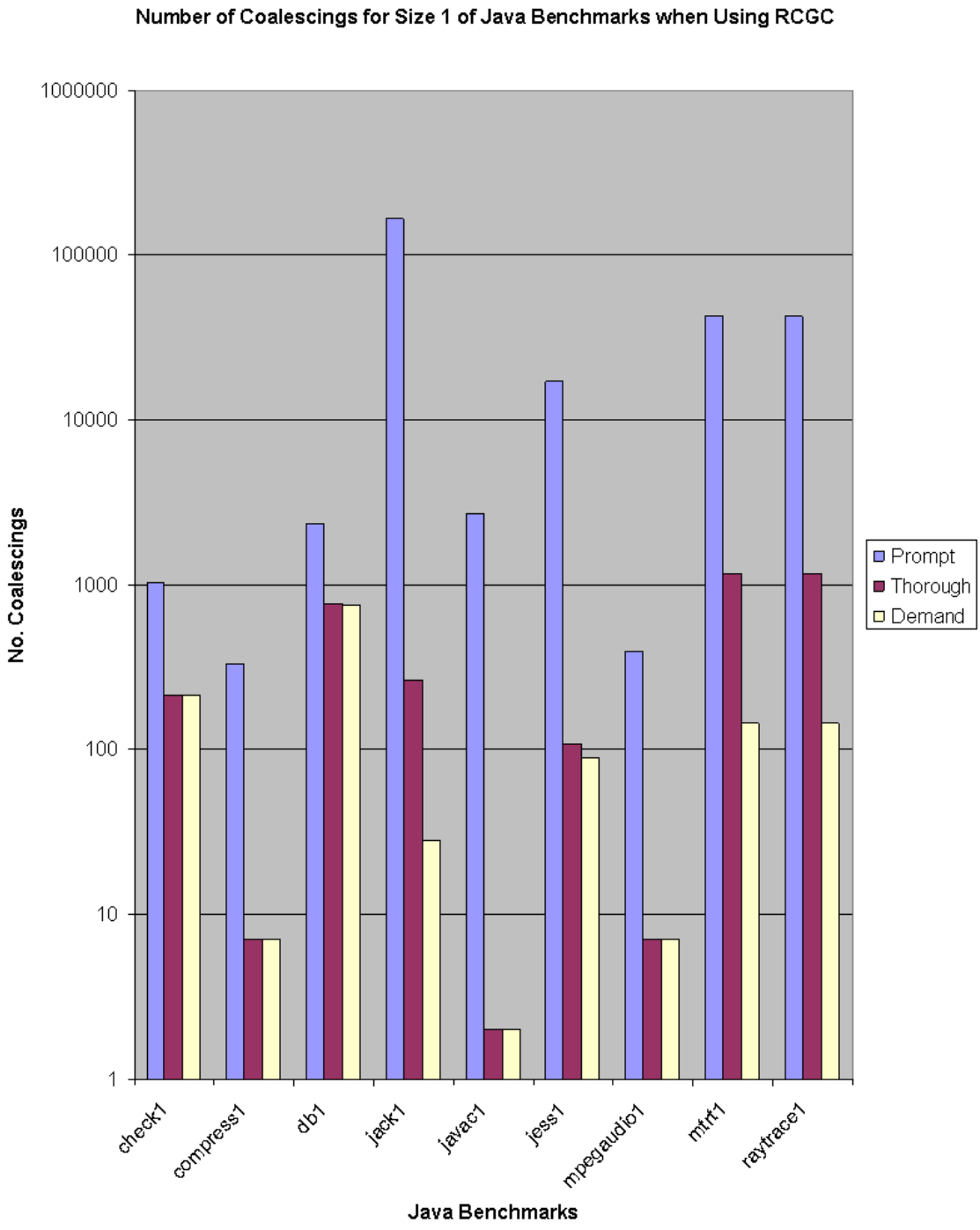

Figure A.3: Number of Coalescings vs Coalescing Strategy when Using RCGC - Benchmarks are of Size 1

A.3 Memory Relocation - Defragmentation

Table A.7: Percentage of Memory Relocated for Size 1 of Java Benchmarks

Benchmark	Garbage Collector	% Memory Relocated	Garbage Collector	% Memory Relocated	Garbage Collector	% Memory Relocated	Coalescing Strategy
200_check	JVMGC	0.0	CGC	0.0	RCGC	0.00143	Prompt
	JVMGC	0.21704	CGC	0.08138	RCGC	0.08567	Thorough
	JVMGC	0.21418	CGC	0.08138	RCGC	0.08567	Demand
201_compress	JVMGC	0.0	CGC	0.0	RCGC	0.0	Prompt
	JVMGC	0.01245	CGC	0.0	RCGC	0.00032	Thorough
	JVMGC	0.01245	CGC	0.0	RCGC	0.00032	Demand
209_db	JVMGC	0.0	CGC	0.0	RCGC	0.00103	Prompt
	JVMGC	0.0	CGC	0.73514	RCGC	0.45138	Thorough
	JVMGC	0.0	CGC	0.75159	RCGC	0.48840	Demand
228_jack	JVMGC	0.0	CGC	0.0	RCGC	0.0	Prompt
	JVMGC	0.22508	CGC	0.00095	RCGC	0.00014	Thorough
	JVMGC	0.22508	CGC	0.00095	RCGC	0.00014	Demand
213_javac	JVMGC	0.0	CGC	0.01291	RCGC	0.0	Prompt
	JVMGC	0.0	CGC	0.01291	RCGC	0.0010	Thorough
	JVMGC	0.0	CGC	0.01291	RCGC	0.00103	Demand
202_jess	JVMGC	0.0	CGC	0.00238	RCGC	0.01256	Prompt
	JVMGC	0.0	CGC	0.00238	RCGC	0.02206	Thorough
	JVMGC	0.0	CGC	0.00238	RCGC	0.03021	Demand
222_mpegaudio	JVMGC	0.0	CGC	0.03941	RCGC	0.0	Prompt
	JVMGC	0.06528	CGC	0.03941	RCGC	0.00246	Thorough
	JVMGC	0.06528	CGC	0.03941	RCGC	0.00246	Demand
227_mtrt	JVMGC	0.0	CGC	0.00145	RCGC	0.0	Prompt
	JVMGC	0.0	CGC	0.00169	RCGC	0.0	Thorough
	JVMGC	0.0	CGC	0.00169	RCGC	0.0	Demand
205_raytrace	JVMGC	0.0	CGC	0.00145	RCGC	0.0	Prompt
	JVMGC	0.0	CGC	0.00169	RCGC	0.0	Thorough
	JVMGC	0.0	CGC	0.00169	RCGC	0.0	Demand

A.4 Synthetic Traces

Table A.8: Memory Usage Statistics for Synthetic Traces - 70% Allocation Bias

Trace	No. of Objects	maxlive	Memory Allocated	Memory Relocated	No. of Relocations	No. of coalescings	Coalescing Strategy
trace0	1000	4194304	19085760	711808	204	466	Prompt
	1000	4194304	19085760	663280	279	453	Thorough
	1000	4194304	19085760	663280	280	453	Demand
trace1	10000	4194304	37191760	838640	2052	3070	Prompt
	10000	4194304	37191760	871728	2021	3021	Thorough
	10000	4194304	37191760	871728	2021	3019	Demand
trace2	100000	4194304	64427408	1958928	19532	26590	Prompt
	100000	4194304	64427408	1991504	19552	26591	Thorough
	100000	4194304	64427408	1991504	19552	26590	Demand
trace3	125000	4194304	65256272	1726720	23445	31551	Prompt
	125000	4194304	65256272	1728368	23457	31555	Thorough
	125000	4194304	65256272	1728368	23457	31555	Demand
trace4	250000	4194304	85822560	3432944	46937	60170	Prompt
	250000	4194304	85822560	3432944	46935	60160	Thorough
	250000	4194304	85822560	3432944	46935	60160	Demand
trace5	375000	4194304	98498608	4373136	70629	88403	Prompt
	375000	4194304	98498608	4373712	70612	88369	Thorough
	375000	4194304	98498608	4373712	70612	88369	Demand
trace6	500000	4194304	108367376	4893184	94352	116136	Prompt
	500000	4194304	108367376	4892336	94312	116068	Thorough
	500000	4194304	108367376	4892336	94312	116068	Demand
trace7	750000	4194304	127819520	6088592	140390	169551	Prompt
	750000	4194304	127819520	6370624	140623	169729	Thorough
	750000	4194304	127819520	6370624	140623	169729	Demand
trace8	1000000	4194304	141283648	7495760	187868	222793	Prompt
	1000000	4194304	141283648	7503040	187999	222867	Thorough
	1000000	4194304	141283648	7503040	187999	222867	Demand
trace9	1250000	4194304	156710096	8754512	235295	275906	Prompt
	1250000	4194304	156710096	8756704	234922	275504	Thorough
	1250000	4194304	156710096	8756704	234922	275504	Demand

References

[1] David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 81–92. ACM Press, 2003.

[2] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel: +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel: +1 617 354 5800, second edition.

[3] K. Bowers and D. Kaeli. Characterizing the SPEC JVM98 benchmarks on the Java Virtual Machine. Technical report, Northeastern University, Dept. of ECE, Computer Architecture Group.

[4] 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.

[5] Sharath R. Cholleti. Storage allocation in bounded time. Master's thesis, Washington University, Saint Louis, MO, 2002.

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

[7] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of High Performance Computing and Networking (SC'97)*, San Jose, California, November 1997.

[8] Dirk Grunwald, Benjamin G. Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–186, Albuquerque, New Mexico, June 1993. ACM.

[9] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2002.

[10] Mahmut T. Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Design Automation Conference*, pages 304–307, Los Angeles, CA, June 2000. ACM.

[11] Arie Kaufman. Tailored-list and recombination-delaying buddy systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):118–125, 1984.

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

[13] Tom Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1997.

[14] Chris H. Pappas and William H. Murray III. *The Complete Reference Visual C++ 6*. Osborne/McGraw-Hill, 1998.

[15] James L. Peterson and Theodore A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[16] J. M. Robson. Worst Case Fragmentation of First-fit and Best-fit Storage Allocation Strategies. *Computer Journal*, 20(3):242–244, 1977.

[17] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 4th edition, 2001.

[18] Charles B. Weinstock. *Dynamic Storage Allocation Techniques*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, April 1976.

[19] Terry A. Welch. A technique for high performance data compression. *j-COMPUTER*, 17(6):8–20, June 1984.

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

[21] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, Kinross, Scotland, UK, September 1995.

Vita

Delvin C. Defoe

Date of Birth November 28, 1973

Place of Birth La Plaine, Dominica

Degrees B.S. Magna Cum Laude, Computer Science and Mathematics, May 2001, from MidWestern State University, Wichita Falls, Texas.

Publications Delvin Defoe, Ranette Halverson, Nelson Passos, Richard Simpson, and Reynold Bailey. "A Study of Software Pipelining for Multi-dimensional Problems", in *Proceedings of the AeroSense-Aerospace/Defense Sensing, Simulation and Controls*, Orlando, FL, April 2001.

Delvin Defoe, Ranette Halverson, Nelson Passos, Richard Simpson, and Reynold Bailey. "Theoretical Constraints on Multi-Dimensional Retiming Design Techniques", in *Proceedings of the 13th International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, August 2000.

December 2003