

國立交通大學

資訊工程系

碩士論文

使用依區塊大小排列的分離式串列的物件配置器

在垃圾蒐集環境下的尋訪之減少



Reducing the Traversals in Size-Order Segregated List
Object Allocator for Garbage Collection Environment

研究生：黃欽毓

指導教授：鍾崇斌 博士

中華民國九十四年八月

使用依區塊大小排列的分離式串列的物件配置器

在垃圾蒐集環境下的尋訪之減少

Reducing the Traversals in Size-Order Segregated List

Object Allocator for Garbage Collection Environment

研究生：黃欽毓

Student：Chin-Yu Huang

指導教授：鍾崇斌 博士

Advisor：Dr. Chung-Ping Chung



A Thesis

Submitted to Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science and Information Engineering

August 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年八月

使用依區塊大小排列的分離式串列的物件配置器 在垃圾蒐集環境下的尋訪之減少

學生：黃欽毓

指導教授：鍾崇斌 博士

國立交通大學資訊工程學系碩士班

摘要

使用依區塊大小排列的分離式串列之物件配置器是一種常見的物件配置器。使用依區塊大小排列的分離式串列的物件配置器，在作物件配置時，需要做尋訪以找出存放合適大小的區塊的串列。因為串列可能會變成空的，使得尋訪的串列增加，所以其效能仍然有改善的空間，這種情形在垃圾收集環境之下尤為顯著。

因此，我們在這份研究中提出了一個查表機制，以減少在配置物件時的尋訪動作。機制中的主要結構是一個要求區塊大小 - 最靠近的存放夠大區塊的非空串列的對映表。在配置物件之時，藉由查詢此對映表，只需要一次尋訪就能定位出存放有合適大小的區塊的串列，使得物件配置的速度大幅增快；我們並且以爪哇語言，一種使用自動垃圾回收機制的物件導向程式語言為例。結果顯示我們提出的機制能使管理堆積區的效能增加一倍。我們也研究了不同的區塊合併策略在自動垃圾回收環境之下，對堆積區管理的效能的影響；並且指出了在選擇區塊合併策略上的權衡，會在加入我們所提出的機制之後發生變化。原本會使得物件配置速度較慢而不傾向被使用的立即合併策略，在加入了我們的機制之後，成為較好的合併策略。這個結果也說明了在自動垃圾收集環境之下，只要物件配置速度夠快，空間破碎問題才是較重要的議題。

Reducing the Traversals in Size-Order Segregated List Object Allocator for Garbage Collection Environment

Student : Chin-Yu Huang

Advisor : Dr. Chung-Ping Chung

Department of Computer Science and Information Engineering
National Chiao-Tung University

Abstract

Size-ordered segregated list allocator is a widely-used object allocator. Upon object allocation, traversals are required to locate the free list holding large enough free chunks. Because some free lists might become empty, increasing the traversals upon object allocation, the performance of size-ordered segregated list allocator still has room for improvement. This is especially true in garbage collection environment.

Therefore, we propose a table lookup mechanism to reduce the traversals. The main structure in this mechanism is a table which is a mapping *from allocation request of each size to the closest non-empty free list holding large enough chunks*. By looking up the table, only one traversal is required to locate the closest non-empty free list, hence speeding up object allocation. We take Java, a popular object-oriented programming language with automatic garbage collection, as example. The result shows that our proposed mechanism can improve the heap management performance by 100%. We also studied the impact of using different coalescing strategies on heap management performance in garbage collection environment, and showed that the trade-off to choosing a coalescing strategy will change, in the presence of our proposed mechanism. Immediate coalescing strategy, which is not used because it results in slower allocation speed, becomes a better one after our proposed mechanism is added. The result also indicates that fragmentation problem is a more serious issue in garbage collection environment than allocation performance if the allocation speed is fast enough.

Acknowledgment

首先我要感謝我的指導教授 鍾崇斌老師。這兩年在交大的時光，老師在學業上面給了我非常嚴謹的指導和許多寶貴的建議，讓我成長了許多，並且得以順利完成碩士論文；老師除了在學業上的指導之外，也在做人做事的方法上給我很大的啟發。我也要感謝實驗室的另一位指導老師 單智君老師，除了擔任我的口試委員，並且在小組討論的時候給了我非常多的幫助。還要感謝另外一位口試委員 邱日清老師，在口試的時候給我的建議，讓我的論文更加完善。

我也要感謝實驗室的所有學長、同學們在我做碩士論文的期間給我的協助和鼓勵，讓我能得以克服遭遇到的種種困難，並且在沮喪與失意之時重燃鬥志。特別感謝喬偉豪學長，和 Java 研究群的黃俊諭、劉彥志、陳裕生同學。

最後，要感謝我的家人給我的支持，才讓我有信心和動力把碩士論文完成。



黃欽毓

2005/8/30

Table of contents

| | |
|---|------|
| 摘要..... | i |
| Abstract..... | ii |
| Acknowledgment..... | iii |
| Table of contents..... | iv |
| List of Figures..... | vi |
| List of Tables..... | viii |
| Chapter 1 Introduction..... | 1 |
| Chapter 2 Background..... | 4 |
| 2.1 Java Technology..... | 4 |
| 2.2 Heap Management..... | 5 |
| 2.3 Mark-Sweep Garbage Collector..... | 7 |
| 2.4 Free-List-Based Object Allocators..... | 8 |
| 2.4.1 Single List Object Allocator..... | 8 |
| 2.4.2 Size-Ordered Segregated List Object Allocator..... | 9 |
| 2.4.3 Coalescing Strategies of Allocators..... | 11 |
| 2.4.4 Summary of Free-List-Based Object Allocators in Garbage Collection Environment..... | 15 |
| Chapter 3 Design..... | 16 |
| 3.1 Observation on Size-Ordered Segregated List Fit Allocator in Garbage Collection Environment..... | 16 |
| 3.2 Next Hit Table Mechanism..... | 18 |
| 3.2.1 Object Allocation with Next Hit Table Mechanism..... | 19 |
| 3.2.2 Maintaining Next Hit Table..... | 20 |
| 3.2.3 Overhead..... | 23 |
| 3.3 Choosing a Coalescing Strategy..... | 24 |
| Chapter 4 Experiment..... | 26 |
| 4.1 Methodology..... | 26 |

| | |
|---|----|
| 4.2 Environment..... | 27 |
| 4.3 Experimental Results..... | 29 |
| 4.3.1 The Impact of Different Coalescing Strategies..... | 29 |
| 4.3.2 The Effectiveness of Next Hit Table Mechanism..... | 32 |
| 4.3.3 The Next Hit Table Mechanism Leads to a Different Choice of Coalescing Strategy..... | 35 |
| 4.3.4 Determining the Number of Free Lists..... | 36 |
| Chapter 5 Conclusion..... | 38 |
| References..... | 40 |
| Appendix: Complete Experimental Results..... | 41 |



List of Figures

| | |
|---|----|
| FIGURE 2-1: Java Virtual Machine internal architecture..... | 4 |
| FIGURE 2-2: Heap free space might be scattered if objects are never move..... | 5 |
| FIGURE 2-3: Heap can be compacted in heap management scheme where object can be moved..... | 6 |
| FIGURE 2-4: Mark phase..... | 7 |
| FIGURE 2-5: Sweep phase..... | 8 |
| FIGURE 2-6: Single list allocator..... | 9 |
| FIGURE 2-7: Size-ordered segregated list fit allocator..... | 10 |
| FIGURE 2-8: Coalescing of contiguous free chunks..... | 11 |
| FIGURE 2-9: Free chunk size distribution where small size chunks are sufficient..... | 13 |
| FIGURE 2-10: Free chunk size distribution where small chunks are insufficient..... | 14 |
| FIGURE 3-1: Nodes of traversals of an object allocation request in different point within program execution..... | 16 |
| FIGURE 3-2: Only the last free list holds a chunk in the beginning of program execution... | 17 |
| FIGURE 3-3: The change in free chunk size distribution during program execution..... | 18 |
| FIGURE 3-4: Next Hit Table..... | 19 |
| FIGURE 3-5: Initialization state of NHT..... | 20 |
| FIGURE 3-6: NHT update example 1..... | 21 |
| FIGURE 3-7: NHT update example 2..... | 22 |
| FIGURE 4-1: Object allocation time using different coalescing strategy..... | 31 |
| FIGURE 4-2: Garbage collection time using different coalescing strategy..... | 31 |
| FIGURE 4-3: Total heap management time using different coalescing strategy..... | 31 |
| FIGURE 4-4: Object allocation time of using deferred coalescing strategy plus NHT..... | 32 |
| FIGURE 4-5: Garbage collection time of using deferred coalescing strategy plus NHT..... | 33 |
| FIGURE 4-6: Total heap management time of using deferred coalescing strategy plus NHT.. | 33 |
| FIGURE 4-7: Total heap management time of using never coalescing strategy plus NHT.... | 33 |
| FIGURE 4-8: Object allocation time of using immediate coalescing strategy plus NHT..... | 34 |
| FIGURE 4-9: Garbage collection time of using immediate coalescing strategy plus NHT.... | 34 |
| FIGURE 4-10: Total heap management time of using immediate coalescing strategy plus | |

| | |
|---|----|
| NHT | 34 |
| FIGURE 4-11: Object allocation time of 4 configurations..... | 35 |
| FIGURE 4-12: Garbage collection time of 4 configurations..... | 36 |
| FIGURE 4-13: Total heap management time of 4 configurations..... | 36 |
| FIGURE 4-14: The number of free lists and heap management time | 37 |
| FIGURE A-1: Object allocation time of different allocators (_202_jess)..... | 41 |
| FIGURE A-2: Garbage collection time of different allocators (_202_jess)..... | 41 |
| FIGURE A-3: Total heap management time of different allocators (_202_jess)..... | 41 |
| FIGURE A-4: Object allocation time of different allocators (_205_raytrace)..... | 42 |
| FIGURE A-5: Garbage collection time of different allocators (_205_raytrace)..... | 42 |
| FIGURE A-6: Total heap management time of different allocators (_205_raytrace)..... | 42 |
| FIGURE A-7: Object allocation time of different allocators (_209_db)..... | 43 |
| FIGURE A-8: Garbage collection time of different allocators (_209_db)..... | 43 |
| FIGURE A-9: Total heap management time of different allocators (_209_db)..... | 43 |
| FIGURE A-10: Object allocation time of different allocators (_213_javac)..... | 44 |
| FIGURE A-11: Garbage collection time of different allocators (_213_javac)..... | 44 |
| FIGURE A-12: Total heap management time of different allocators (_213_javac)..... | 44 |
| FIGURE A-13: Object allocation time of different allocators (_227_mtrt)..... | 45 |
| FIGURE A-14: Garbage collection time of different allocators (_227_mtrt)..... | 45 |
| FIGURE A-15: Total heap management time of different allocators (_227_mtrt)..... | 45 |

List of Tables

| | |
|---|----|
| TABLE 2-1: Most objects are of a small number of sizes..... | 12 |
| TABLE 2-2: Comparison between free-list-based allocator..... | 13 |
| TABLE 3-1: NHT update frequency using deferred coalescing strategy..... | 24 |
| TABLE 4-1: Overview of SPECjvm98 benchmark..... | 28 |



Chapter 1

Introduction

A fundamental and important part of modern programming languages is dynamic object allocation. Many programming languages, such as C, C++, Java, and Smalltalk, allow programmers to dynamically allocate object in heap, and dynamic object allocation has become a powerful tool that enables programmers to develop applications with more flexibility [1]. In object-oriented programming languages such as Java and Smalltalk, since data are encapsulated in objects and objects are almost allocated dynamically, the use of dynamic memory allocation is especially extensive, and the concept of software engineering continues encouraging programmers to use more dynamic allocated object.

In languages with explicit heap management, de-allocation of objects is done by explicit program statements and thus programmers themselves are responsible for dealing with de-allocation. Because the vast amount of allocated objects, a programmer might forget de-allocating dead objects, or might de-allocate an object which should not be de-allocated to ensure correct program execution. This leads to memory leak and dangling reference problem. Therefore, some modern programming languages such as Lisp and Java use automatic garbage collection. Automatic garbage collection means that dynamically allocated objects are not de-allocated explicitly by the running program itself but the garbage collector is instead responsible for reclaiming the spaces occupied by dead objects, and typically, garbage collection is invoked when an object allocation request cannot be satisfied. With the power of automatic garbage collection, memory leak and dangling reference problem can be avoided, and it is easier to write applications that are more safe and reliable [2].

Because the extensive use of dynamically allocated object, time spent on object allocation plays an important role on program total execution time. Thus a fast object allocator is important for efficient execution. In automatic garbage collection environment, another issue about object allocator arises. Since garbage collection is typically invoked when object allocation request cannot be satisfied due to insufficient large contiguous free space, the fragmentation brought by the object allocator is also important.

Size-ordered segregated list allocator is generally believed a very fast object allocator. By the use of segregated list, the allocator rare needs to sequentially search the free list to find a large enough free chunk. However, traversals are still required to locate the closest non-empty free list which holds sufficient large free chunks. And we found that the time spent on traversals is especially much in automatic garbage collection environment. This motivates us to reduce the traversals upon allocation request of size-ordered segregated list allocator in automatic garbage collection environment. Furthermore, the coalescing of contiguous free chunks is also an important issue which will affect the performance of object allocation. Coalescing contiguous free chunks to a large free chunk tends to lower the fragmentation and thus has a positive impact on reducing the number of garbage collection invocation, but it will degrade the allocation speed of size-ordered segregated list allocator. The purpose of this thesis is to compose a size-ordered segregated list object allocator with reduced traversals and choosing a good coalescing strategy to use with this allocator.

The most popular object-oriented programming language which has garbage collection incorporated in is Java [3]. The use of Java is widely spread over many fields of computer system, ranging from high-end server applications to low-end embedded systems. Therefore, Java will be taken as example in this research.

The organization of the thesis is as follows. In chapter 2, the related background is

presented. In chapter 3, we first give an important observation on size-order segregated list allocators in garbage collection environment. Based on that, our design to reduce traversals upon object allocation is proposed. In chapter 4, the performance of the proposed mechanism is evaluated via experiments. Conclusion is presented in chapter 5. Complete experimental results will be presented in Appendix.



Chapter 2

Background

In this chapter, the related background about our research is presented. First, some important tips on Java are given. Second, the heap management is described. Finally, details about dynamic object allocation and garbage collection are depicted.

2.1 The Java Technology

Java is an object-oriented programming language whose syntax is similar to C++. However, Java is a complete system rather than only a programming language. Unlike most other programming languages such as C, C++, and Fortran, Java programs are not statically compiled to machine-dependent native code for execution. On the other hand, Java programs must be compiled to Java virtual machine bytecode, a machine-independent code. Any Java virtual machine conforming the Java virtual machine specification can execute bytecode. Hence, Java technology consists of the Java programming language and the Java virtual machine. Figure 2-1 illustrates the internal architecture of the Java virtual machine.

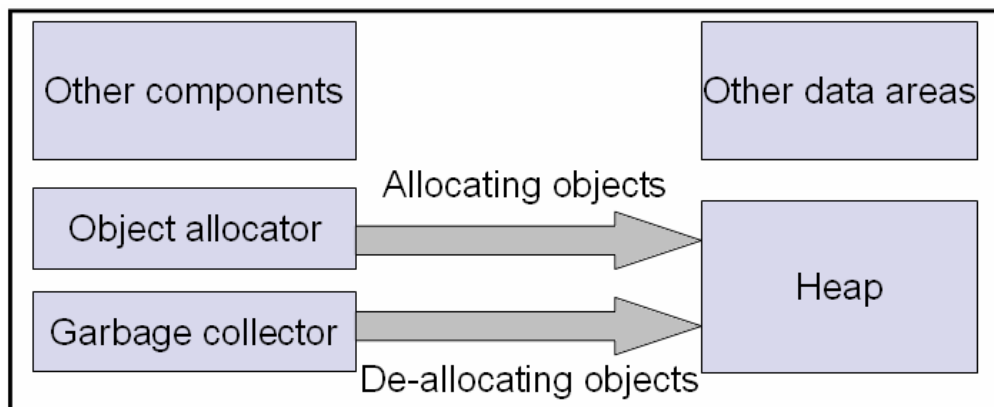


Figure 2-1: Java Virtual Machine Internal Architectures

2.2 Heap Management

Inside the Java virtual machine, there is a Java heap which stores dynamically allocated objects [4]. Java does not allow programs to explicitly de-allocate objects, and it relies on automatic garbage collection to de-allocate dead objects. An object no longer referenced by the running program is said to be dead. To identify and reclaim dead objects in heap are both time consuming thus fewer invocations of garbage collection are preferred. As a result, garbage collection is typically triggered after an object allocation request cannot be satisfied.

The components responsible for managing the heap are the object allocator and the garbage collector, and there are two types of heap management scheme. The first scheme is that allocated objects are never moved (Figure 2-2), and the other is that allocated objects can be moved to make the heap compacted (Figure 2-3).

In heap management scheme where objects are never moved, garbage collection is less costly, since garbage collector only needs to identify and reclaim dead objects. However, free spaces will be scattered around the heap, thus an efficient data structure must be used to manage these scattered free chunks. Object allocation speed thus depends on the underlying data structure and allocation policy.

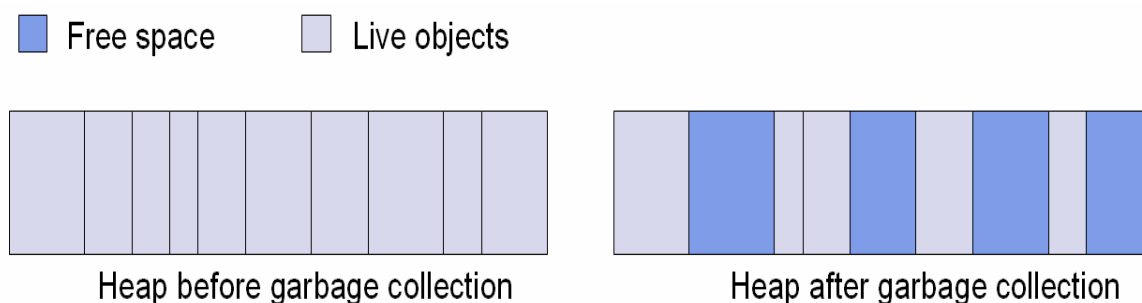


Figure 2-2: Heap free space will be scattered if objects are never move

In heap management scheme where objects can be moved, live objects can be moved to one end of the heap to make the heap compacted. This will result in a single contiguous free space in the heap, thus very simple data structure can be used to management free space. And it makes object allocation almost a non-issue, since upon object allocation request there is only one free chunk to be considered.

Nevertheless, compacting objects bring extra costs to garbage collection, which are copying objects and updating references inside objects. Thus, some systems adopt moving heap management scheme but heap is not always compacted. This means free spaces might also be scattered around the heap like the situation in non-moving heap management scheme.

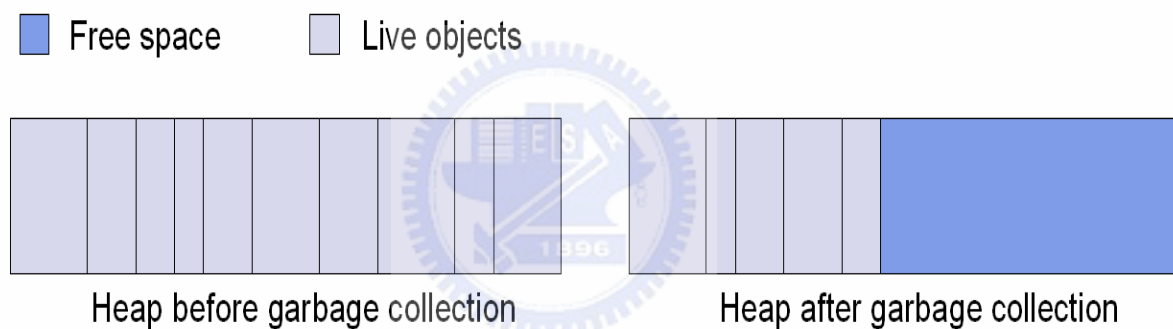


Figure 2-3: Heap can be compacted if object can be moved

Much research has been focused on non-moving heap management scheme and moving heap management schemes. However, it is impossible to pick up a winning strategy, since different strategy leads to different performance tradeoffs [5]. We will focus on heap management scheme where heap is not always compacted.

For heap management scheme where heap is not always compacted, mark-sweep and reference counting garbage collectors are most widely know, but there is rare use of reference counting garbage collector in modern system because of its overhead to count the number of references is too high.

Mark-Sweep Garbage Collector

Mark-sweep garbage collector collects dead objects by two phases, the mark phase and the sweep phase. In mark phase, dead objects are identified, and in sweep phase, dead objects are reclaimed.

Reference variables and dynamically allocated objects form a directed graph, called reference graph. Those variables are roots of this graph. A node n is reachable if there is a path of directed edges $r \rightarrow \dots \rightarrow n$ starting at some root r . In mark phase (Figure 2-4), the reference graph is traversed and each node visited during traversal is marked. Any node not marked is not reachable thus no longer referenced by the running program. These unmarked objects must be garbage, and should be reclaimed.

Then in sweep phase (Figure 2-5), it scans from starting address to the last address of heap, looking for objects that are not marked. These non-marked nodes are garbage and will be linked into the underlying data structure which maintains the free space.

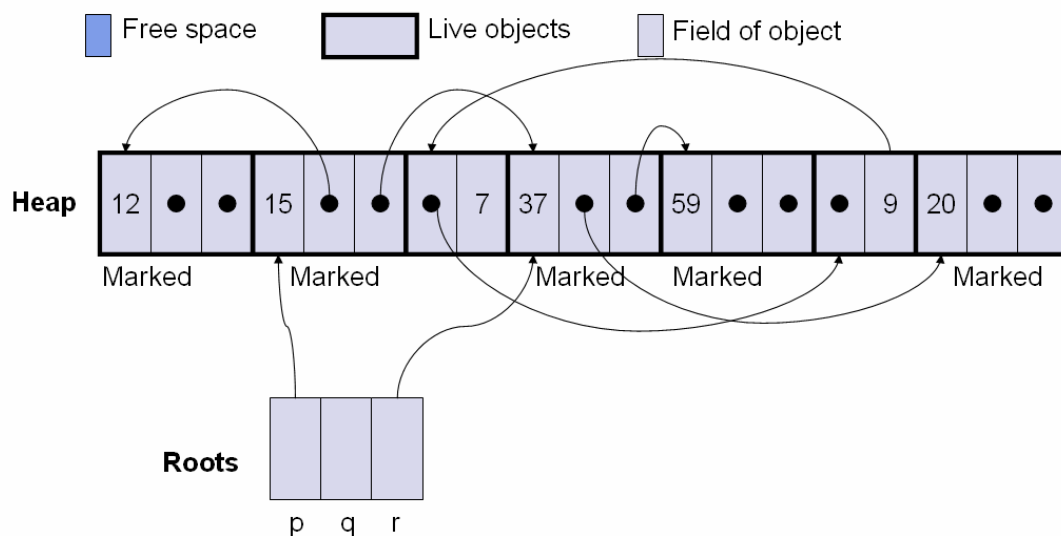


Figure 2-4: Mark phase

(Information source: Figure 13.4(a) of [6])

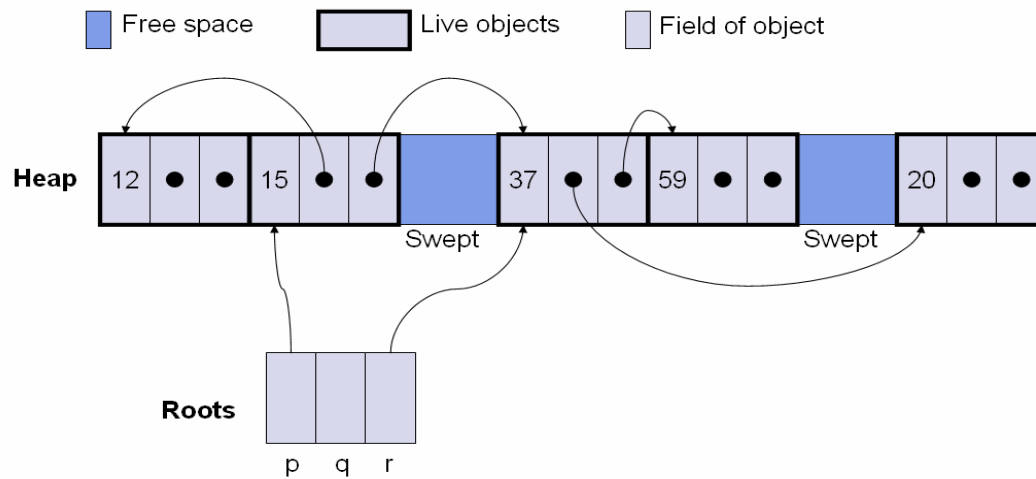


Figure 2-5: Sweep phase

(Information source: Figure 13.4(b) of [6])

2.4 Free-List-Based Object Allocator

Because free space might be scattered around the heap in non-moving heap management scheme, the underlying data structure used to manage free space is generally a graph-like data structure, and in practice, most allocators use linked-list-based data structure to maintain the free space in heap and are conventionally called free-list-based object allocators. Many free-list-based allocators have been proposed and the important free-list-based allocators are described as follows.

2.4.1 Single List Object Allocator

Typically, a contiguous memory space in heap is called a free chunk. In a single list object allocator, free chunks of all sizes are chained a single linked list (Figure 2-6) and thus it must sequentially search the list to find a chunk of appropriate size. Single list first fit, single list best fit, single list worse fit, and single list next fit allocators are widely known allocators in this category.

List Header

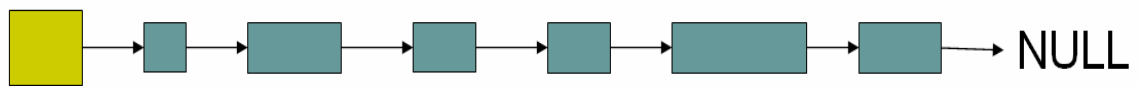


Figure 2-6: Single list object allocator

Single list best fit allocator has to search the linked list to find the smallest free block large enough to satisfy a request. A best fit search is generally exhaustive, although it might stop when a perfect fit is found. Single list best fit allocator generally exhibits low fragmentation but the sequential search does not scale well to large heap, which might have many free chunks.

Single list worst fit requires exhaustive search and it also has high fragmentation, thus it is rarely used.

Single list first fit allocator is the most notable single list allocator, since its fragmentation is pretty low and allocation speed is better than single list best fit and single list worst fit. Upon allocating an object, the allocator searches the free list for the first free chunk that is sufficient large to satisfy the request, and if the remainder after object allocation is larger than minimum object size, the remainder is put back and linked to the end of the free list. Similarly, during garbage collection, contiguous free chunks will always be merged to a larger free chunk. A free chunk is simply added to the end of the free list.

2.4.2 Size-Ordered Segregated List Allocator

Single list first fit allocator has a very high worst case allocation cost, which requires linear search through a single list until finding a large enough free chunk, and average allocation cost in practice is also poor, especially when the heap is large and allocated objects

are of many kinds of sizes.

Hence, segregated list allocators are proposed to reduce the search time of single list allocators. As the name of segregated list allocator, multiple free lists are used to maintain the free space. There are many possible variations of allocators using segregated list, and among all the most notable and efficient one is size-ordered segregated list allocator.

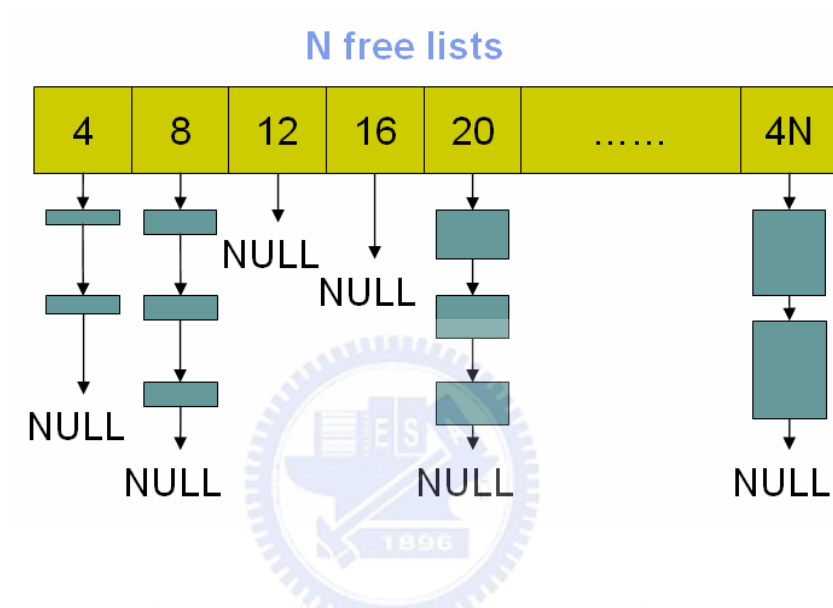


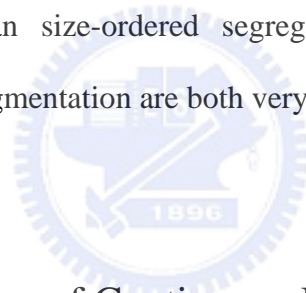
Figure 2-7: Size-ordered segregated list allocator

In size-ordered segregated list allocator (Figure 2-7), each size is associated with a corresponding free list. Each list holds free chunks of exact size, except for the last free list. An integer N is selected such that most free chunks are smaller than $4N$ because there are very few large objects. It is observed that 95% objects are smaller than 256 bytes and 99% objects are smaller than 1K bytes [7], and thus the value N is chosen between 64 and 256 in practice.

Suppose a heap where free chunks are multiple of 4-byte. All free chunks whose sizes are equal to $4N$ or greater than $4N$ are linked in the last free list. Thus, size-ordered segregated list fit allocator has N free lists rather than an unbounded numbers of free lists. Upon allocating an M -byte object, the allocator first indexes into the free list holding size M free

chunks. If $M \geq 4N$, the allocation request is satisfied by best-fit allocation in the last free list. Otherwise, the allocation request processes as follow. If there is an M -byte free chunk chained in the corresponding free list, the allocation request is satisfied by taking the first free chunk linked in the free list. If the free list corresponding to M -byte chunk is empty, the allocator traverses up the free list header array until a non-empty free list holding chunks larger than M is visited. Suppose that the non-empty free list holds free chunks of size K . If $K < 4N$, the allocation request is satisfied by taking the first free chunk in the free list. Otherwise, if $K \geq 4N$, best-fit allocation is tried at the last free list. If there is remainder after allocation, the remainder is put back and inserted into the corresponding free list holds that size.

A notable variant of segregated list allocator is the buddy systems. Buddy systems can provide faster allocation than size-ordered segregated list allocator, but the internal fragmentation and external fragmentation are both very high, thus it is not suitable for garbage collection environment.



2.4.3 Coalescing Strategy of Contiguous Free Chunks

The basic operation of size-ordered segregated list allocator has been described above. And there is an important issue that affects the performance of size-ordered segregated list allocator, “Will contiguous free chunks be coalesced? And when?”

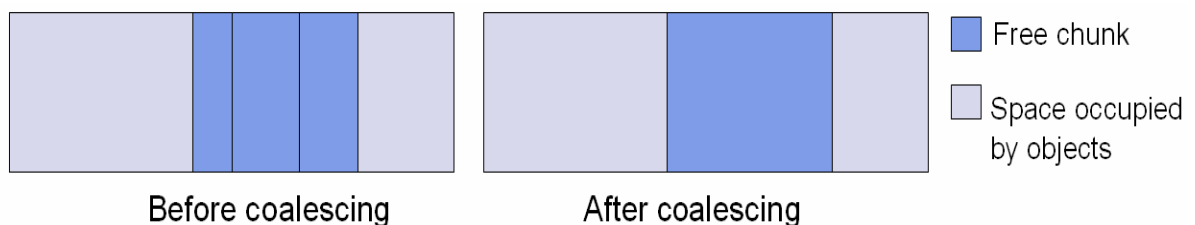


Figure 2-8: Coalescing of contiguous free chunks

Never Coalescing Strategy

A characteristic of object allocation activities is the locality of object size, which says if previously many allocated objects are of some sizes, then objects of these sizes will be allocated extensively. This concept can be supported by the fact that 90% of objects are of 10% of sizes (Table 2-1). Therefore, never coalescing strategy is used. Using never coalescing strategy, newly identified contiguous free chunks are never coalesced but on the other hand each free chunk is directly chained back into the corresponding free list holding that size. This makes small size chunk very sufficient and enables the allocator to cyclic allocate small size chunk very quickly (Figure 2-9).

Table 2-1: Most objects are of a small number of sizes

| Benchmark | Total number of size classes | Number of size classes of 90% objects | Number of size classes of 90% objects / Total number of size classes | Number of size classes of 99% objects | Number of size classes of 99% objects / Total number of size classes |
|---------------|------------------------------|---------------------------------------|--|---------------------------------------|--|
| _201_compress | 73 | 19 | 26.0% | 36 | 49.3% |
| _202_jess | 158 | 12 | 7.6% | 26 | 16.5% |
| _205_raytrace | 72 | 5 | 6.9% | 10 | 13.9% |
| _209_db | 78 | 7 | 9.0% | 16 | 20.5% |
| _213_javac | 166 | 12 | 7.2% | 27 | 16.3% |
| _227_mtrt | 72 | 5 | 7.0% | 10 | 14.0% |
| _228_jack | 448 | 6 | 1.3% | 18 | 4.0% |
| Average | 166 | 10 | 10% | 22 | 13% |

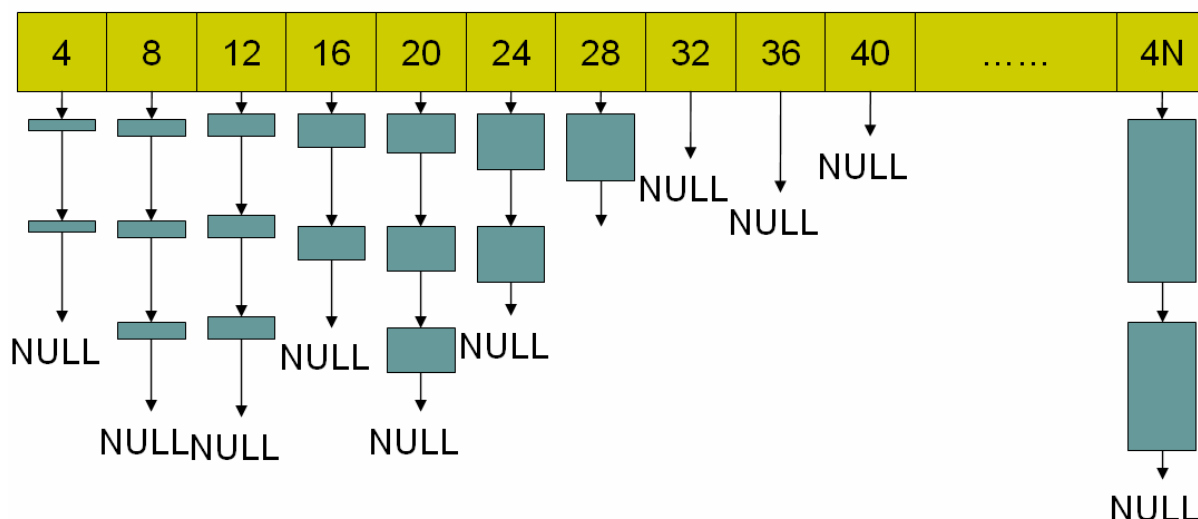


Figure 2-9: Free chunk size distribution where small size chunks are sufficient

Because contiguous free chunks are never coalesced into a larger chunk, it is likely to produce many small chunks in heap, and larger free chunks are difficult to find. This fragmentation problem might lead to more frequent invocation of garbage collection, since an object allocation request for large size might fail when there is still considerable total free space in heap.

Deferred Coalescing Strategy

Another strategy is to defer the coalescing operation. Its goal is similar to never coalescing, but has lower fragmentation. Using deferred coalescing strategy, contiguous free chunks are not coalesced upon garbage collection but on the other hand each free chunk is directly chained into the corresponding free list holding that size. Coalescing only occurs upon request a free chunk in the last free list or when allocation request still fails after garbage collection. When searching the last free list for a best-fit chunk, contiguous large chunks are merged to a larger chunk. When allocation request still fails after garbage collection, each group of contiguous free chunks will be coalesced.

Immediate Coalescing Strategy

Never coalescing strategy does not try to merge any contiguous free chunk, and deferred coalescing strategy only merges contiguous free chunk lazily. Hence, another strategy which has a fundamental different criteria is used – immediate coalescing.

Immediate coalesce strategy will coalesce each group of contiguous free chunks upon garbage collection. This can be done during the sweep phase of mark-sweep garbage collector, thus the coalescing operation incurs little overhead. After coalescing, the resulting bigger free chunks are added back to the corresponding free list. Thus, the free chunk size distribution using immediate coalescing strategy is very different from that using deferred coalescing and never coalescing strategy – smaller chunks are insufficient while larger chunks are more (Figure 2-10). And this results worse allocation performance, since the allocator is more likely to traverse more headers to locate the closest non-empty free list. On the other hand, since coalescing is performing aggressively, it results in less fragmentation.

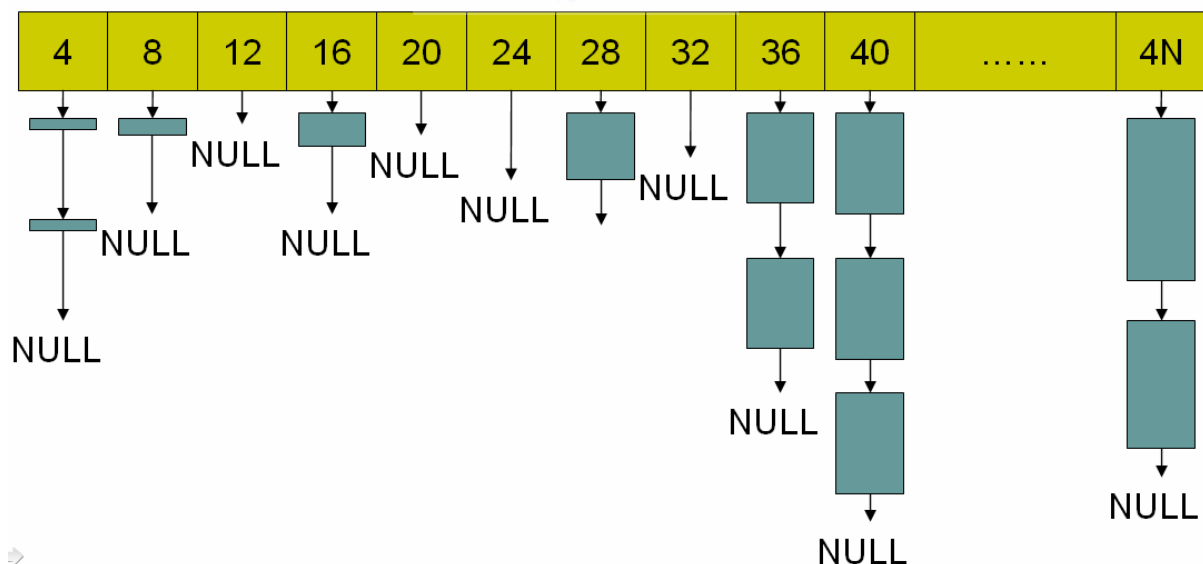


Figure 2-10: Free chunk size distribution where small chunks are insufficient

2.4.4 Summary of Free-List-Based Object Allocators in Garbage Collection Environment

We have described two important categories of object allocator, single list first fit allocator and size-ordered segregated list allocator. Single list first fit allocator requires searching a single linked list upon allocation, thus performs badly in large heap size because there are potentially more free chunks. Hence single list first fit allocator is only suitable for memory-constrained embedded system. On the other hand, size-ordered segregated list allocator is more scalable than single list first fit allocator. Moreover, three important coalescing strategies used in size-ordered segregated list allocator have also presented.

Table 2-2 is the comparison among these free-list-based allocators.

Table 2-2: Comparison among free-list-based allocators (SPECjvm98 with 6M heap size)

| Allocator | Worst case cost per allocation | Average cost per allocation | External Fragmentation |
|---|---|-------------------------------------|------------------------|
| Single List First Fit | $O(n)$, n = the number of free chunks | Several hundreds of node traversals | Low |
| Size-Ordered Segregated List + Never Coalescing | $O(m+k)$, m = the number of free lists, | 15 ~ 40 node traversals | High |
| Size-Ordered Segregated List + Deferred Coalescing | The same as above | 17 ~ 40 node traversals | High |
| Size-Ordered Segregated List + Immediate Coalescing | The same as above | 50 node traversals | Low |

Chapter 3

Design

In this chapter, we will first present an important observation on the change of allocation performance with time, which inspires our design. Then, our design for accelerating object allocation is presented. Finally, we will give a discussion on what might be the best coalescing strategy to incorporate into design.

3.1 Observation on Size-Ordered Segregated List Allocator in Garbage Collection Environment

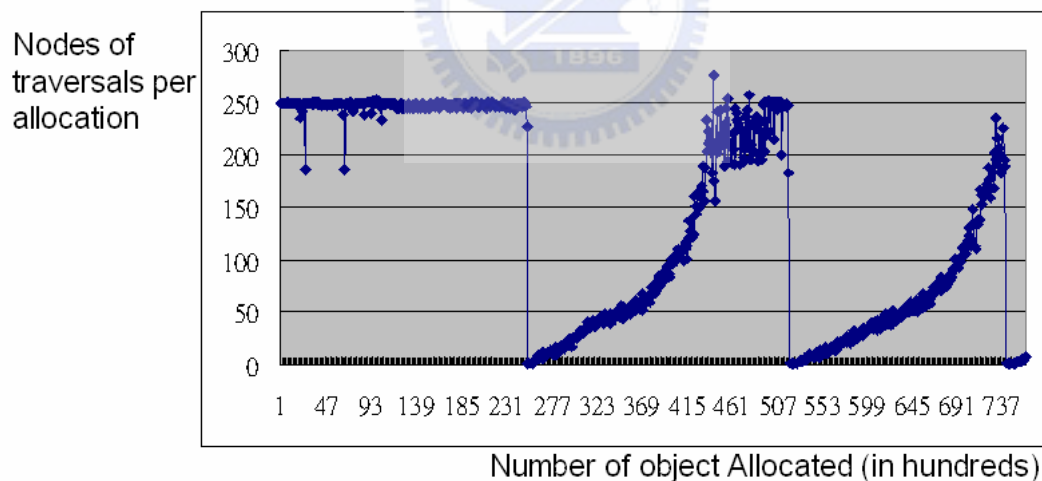


Figure 3-1: Nodes of traversals of an object allocation request in different point within program execution (SPECjvm98 with 6M heap size)

From the Figure 3-1, it is observed that before the first invocation of garbage collection, the size-ordered segregated list allocator pays excessive traversals upon object allocation

request. This is because the heap is itself a big free chunk in the beginning of program execution, and until the first invocation of garbage collection, there is only a single free chunk in heap and this chunk is linked in the last free list while other free lists are empty. Any allocation request for small sizes must be satisfied at the last free list, after traversing through the free list headers. (Figure 3-2)

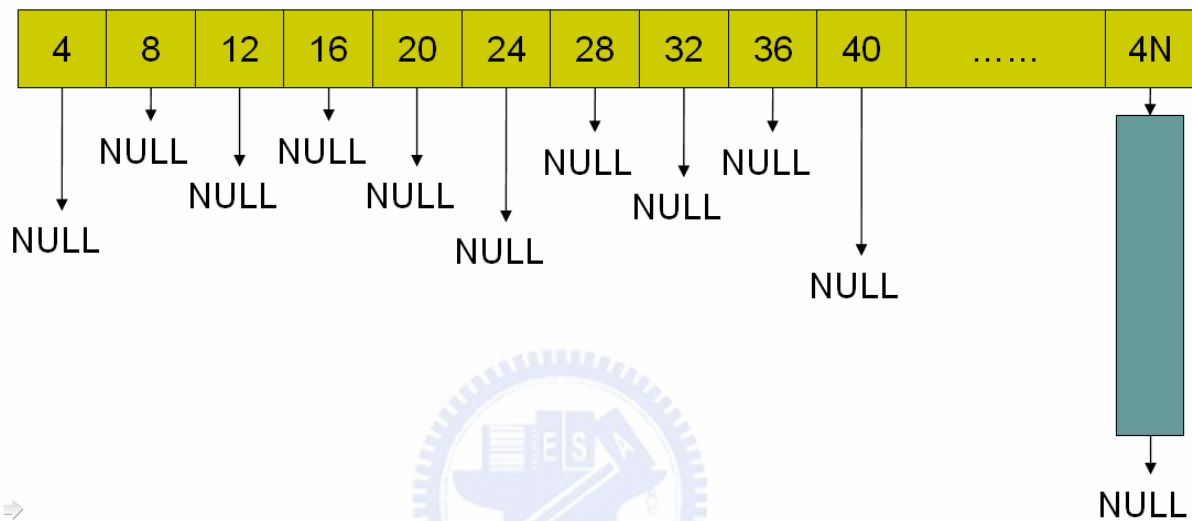


Figure 3-2: Only the last free list holds a chunk in the beginning of program execution

Another observation from the figure is that allocation performance is very good after garbage collection just finishing, however, the allocation performance is dropping as the next invocation of garbage collection is approaching. The explanation of this phenomenon is that during the execution of a Java program, dead objects can be de-allocated and the spaces occupied by them are returned to free lists only during garbage collection. This indicates that free chunks are continuously consumed upon allocation request and free spaces cannot be reclaimed until garbage collection. Therefore, free chunks of some sizes become exhausted as the program executes. As the program continues its execution and garbage collection is approaching, more free lists become empty and thus the speed performance of object allocation degrades sharply. (Figure 3-3)

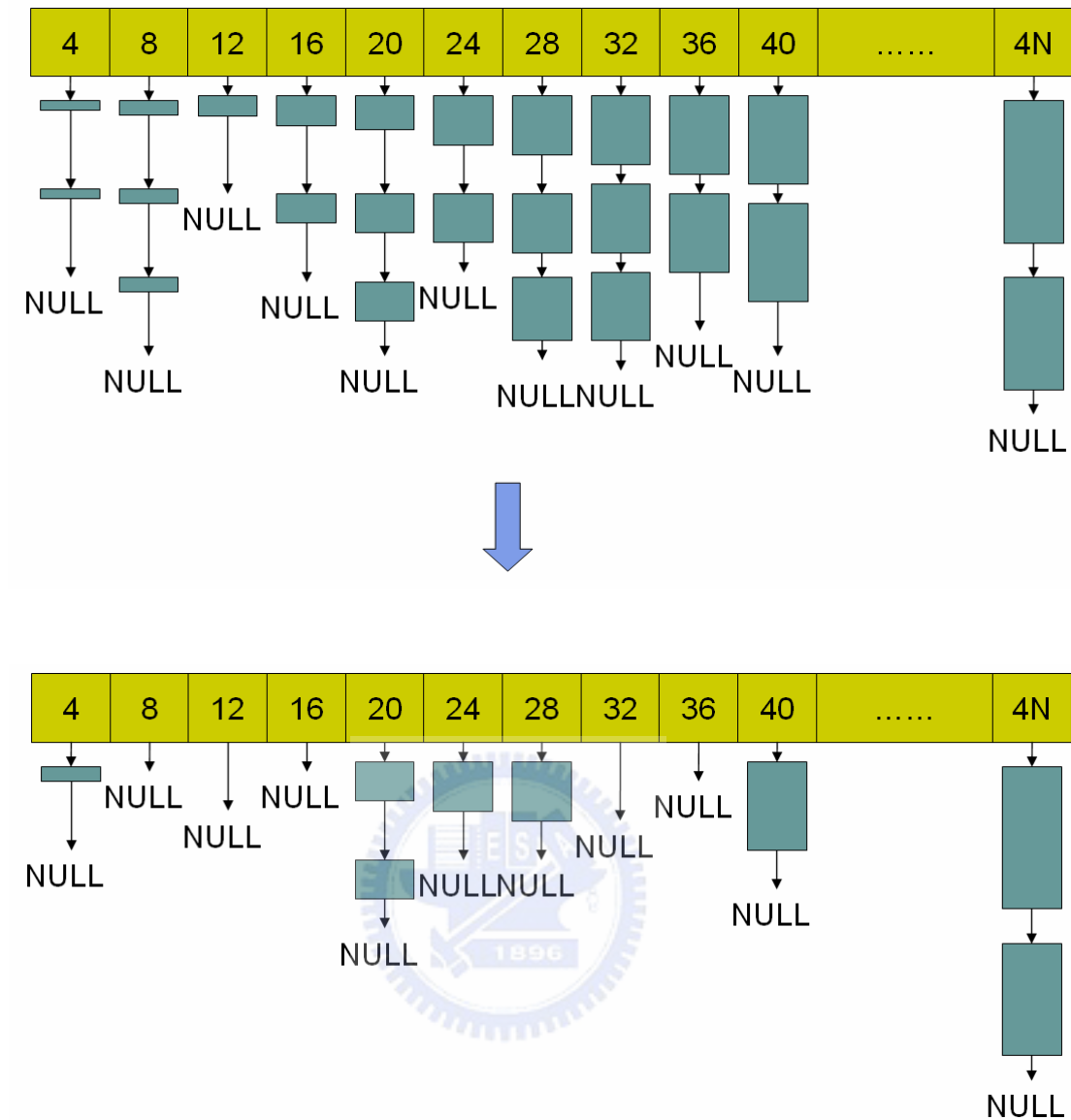


Figure 3-3: The change in free chunk size distribution during program execution

3.2 Reducing Traversal during Object Allocation – Next Hit

Table Mechanism

By observing the object allocation process of size-ordered segregated list allocator, we know that it must traverse up the free list headers if the free list corresponding to the requested size contains no chunk and a sufficient large chunk is found until the closest

non-empty free list is visited. Since many free lists become empty as the program execution, why not to remember allocation request of each size can be satisfied at which free list?

We can add a table to direct allocation request of each size to the closest non-empty free list holding sufficient large size. The table is essentially a mapping from requested size to the right free list, thus each size has a corresponding table entry. We called the table Next Hit Table (NHT) since by looking up the table, the right free list is located without traversing up. In the following subsections, we suppose that heap chunks are multiple of 4 bytes because most Java virtual machines use 4-byte as the unit of heap spaces.

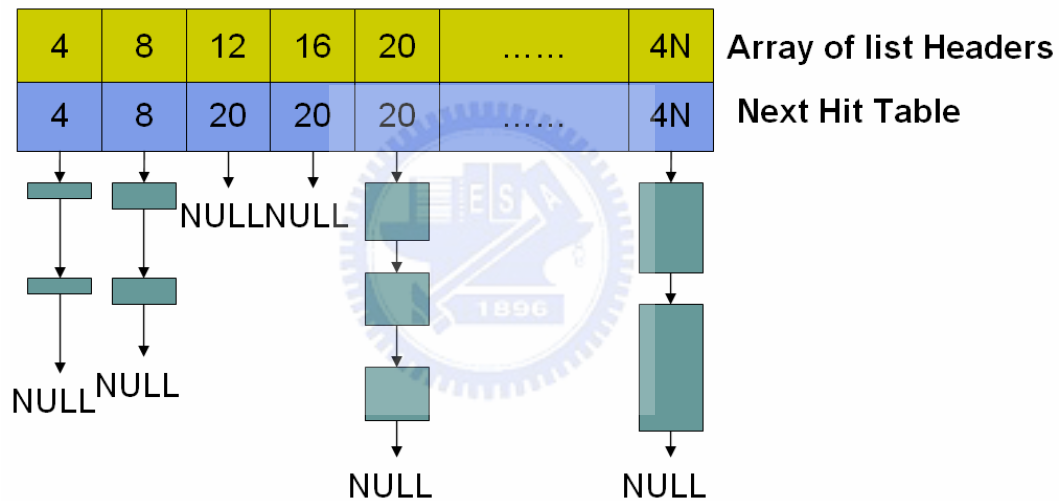


Figure 3-4: Next Hit Table

3.2.1 Object Allocation with Next Hit Table

Object allocation with NHT support is done as follow. Upon allocation request of a size S object, the allocator first lookup the table to know that a sufficient large chunk can be found in free list F , holding $4F$ -byte objects. If $F < N$, the allocation request is satisfied by taking the first free chunk in free list F . Otherwise, it searches a best-fitting chunk in the last free list (free list N) to satisfy the allocation request.

3.2.2 Maintaining Next Hit Table

NHT will be initialized to N , which indicates the last free list because in the beginning of execution, the heap is itself a single large free chunk. Note that because there is only a free chunk in the heap until the first invocation of garbage collection, the initially bad allocation performance before the first invocation of garbage collection becomes very quick, a table lookup plus direct allocating the only chunk without searching.

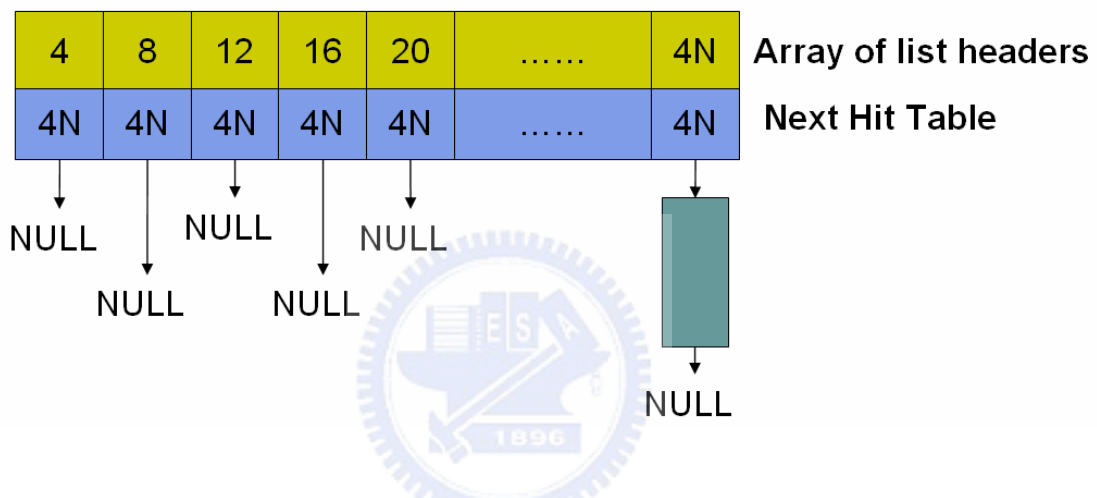


Figure 3-5: Initialization state of NHT

For maintaining the right mapping, NHT must be updated when a free list becomes empty and when a free list becomes non-empty. A free list will become empty if it runs out of chunk in list after object allocation request, and there are two cases that a free list will become non-empty. The first case is when the remainder of an allocated chunk is linked back to a free list and the free list is previous empty. The second case occurs during garbage collection - when a free chunk is swept and then linked back to a free list and the free list has been empty before garbage collection.

When a free list F becomes empty after object allocation, the update process is as follow. Suppose the NHT entries corresponding to free lists $F-1, F-2, \dots, F-i$ are of value F , and the

NHT entry corresponding to $F - i - 1$ is not of value F . Then the NHT entries corresponding to free lists $F, F-1, F-2, \dots, F - i$ are updated to the value of NHT entry corresponding to free list $F + 1$. This is to reflect the fact that afterwards allocation requests of size $4F, 4F-4, \dots, 4F - 4i$ can no longer be satisfied at free list F . Figure 3-6 depicts the update operation when the free list holding 20-byte chunk becomes empty.

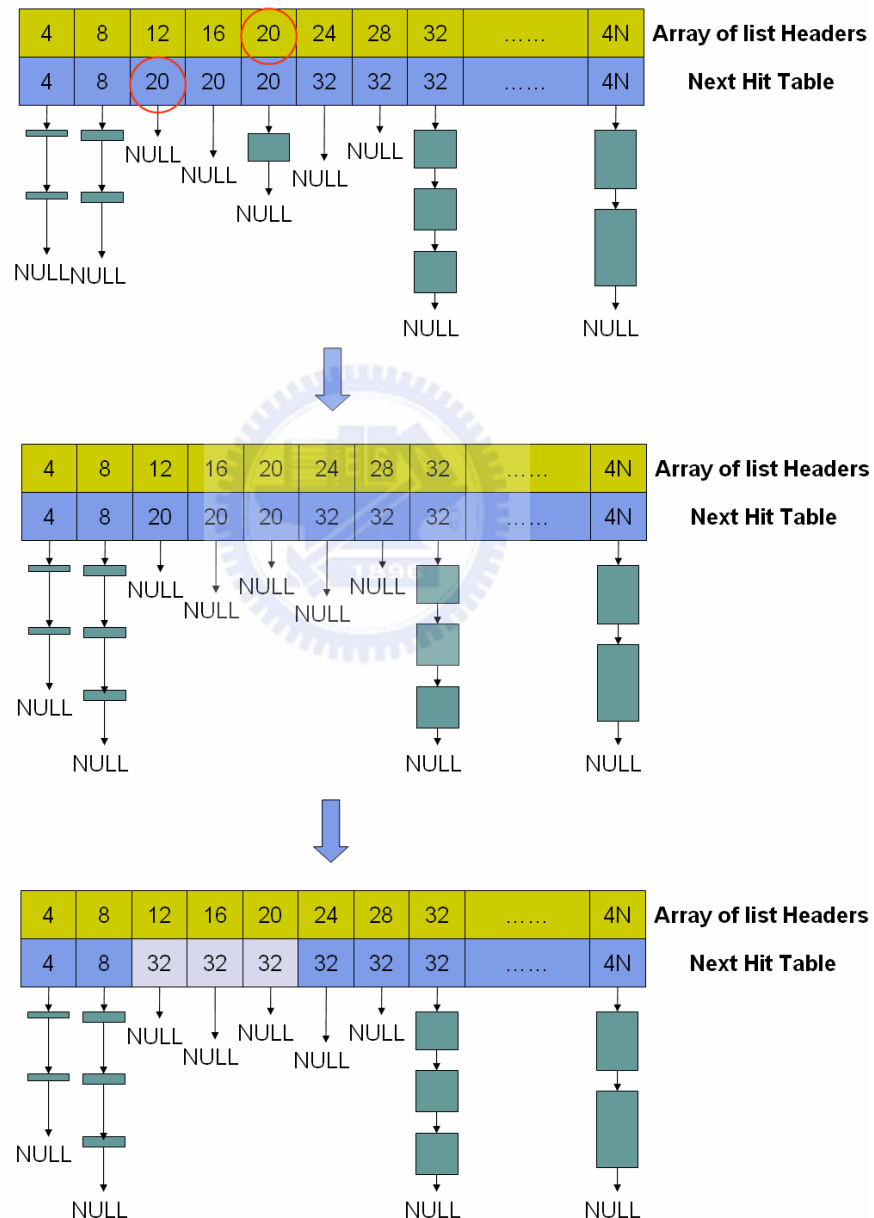


Figure 3-6: NHT update example 1

When a free list F becomes non-empty, the update process is as follow. Suppose the NHT entry corresponding to free list $F - 1$ is not of value $F - 1$, the NHT entry corresponding to free list $F - 2$ is not of value $F - 2$, ..., and the NHT entry corresponding to free list $F - i$ is not of value $F - i$, but the NHT entry corresponding to free list $F - i - 1$ is of value $F - i - 1$, then the NHT entries corresponding to free lists $F, F-1, F-2, \dots, F - i$ are updated to F to reflect that next allocation request of size $4F, 4F-4, \dots, 4F - 4i$ can be satisfied at free list F . Figure 3-7 depicts the update operation when the free list holding 16-byte chunk becomes non-empty due to the remainder being added back.

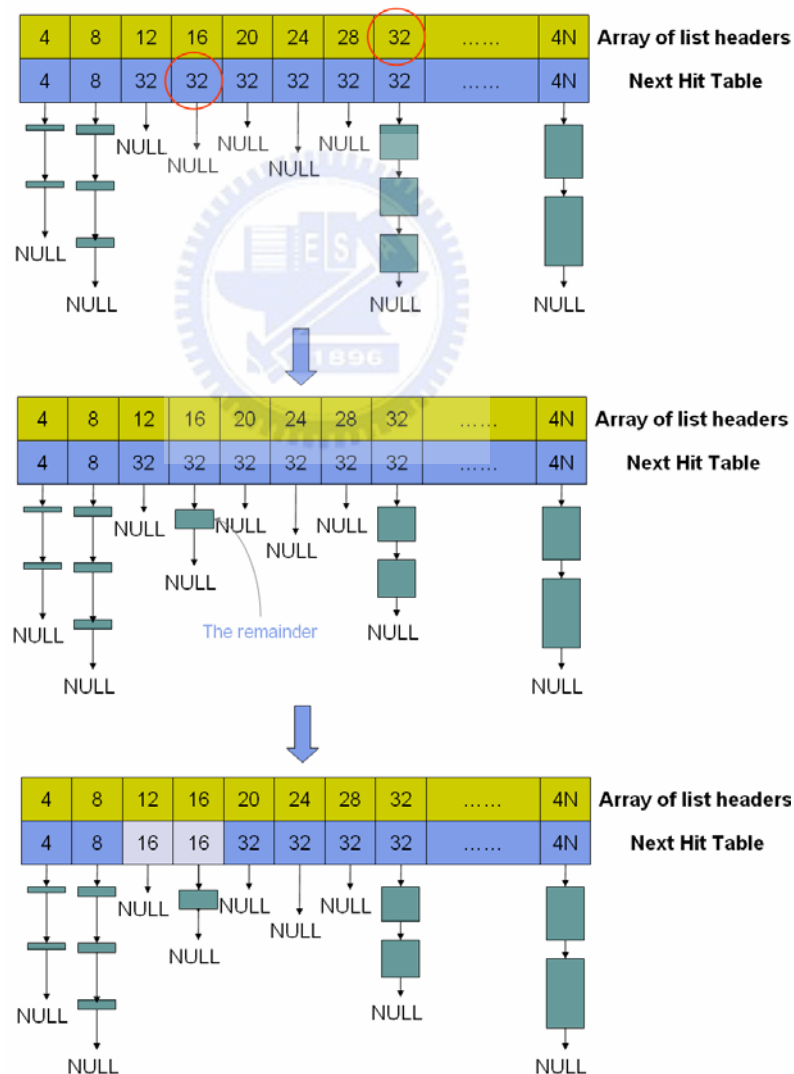


Figure 3-7: NHT update example 2

If the update occurs during garbage collection, we use special handling. Because there is no object allocation request during garbage collection, we can postpone the update to the end of garbage collection process. By such treatment, the table only needs to be updated once during garbage collection.

3.2.3 Overhead

We have described how our proposed mechanism, NHT, works to reduce the traversals upon object allocation in size-ordered segregated list allocator. We discuss the overhead using NHT now.

NHT mechanism incurs both space and speed overhead. The space overhead comes from the data structure implementing NHT, which can be simply an array of integer. And because typically size-ordered segregated list allocator uses no more than hundreds of free lists which mean hundreds of 4 bytes, the overhead only counts up several KB.

The speed overhead might be more critical than space overhead, since if the speedup gained by fast object allocation can not amortize the time cost to maintain NHT, our ambition will simply fail. Remember that the update only occurs in three cases, and these occur infrequently. A free list might become empty only when it runs out of free chunks, and free chunks are reclaimed only during garbage collection, thus this case rarely occurs. However, the speed overhead of NHT when using immediate coalescing strategy might be higher, because it is more potential to have the remainder after allocating a larger chunk and the remainder will be linked back into a free list. Our simulation result shows that update does not occur frequently – only one update occurs averagely during 1000 allocation request using deferred coalescing strategy (Table 3-1).

Table 3-1: NHT update frequency using deferred coalescing strategy

| Benchmark | A free list becomes non-empty | A free list becomes empty | Total number of update | Total number of object allocation | Number of allocation / Number of update |
|---------------|-------------------------------|---------------------------|------------------------|-----------------------------------|---|
| _201_compress | 145 | 116 | 261 | 6479 | 25 |
| _202_jess | 254 | 225 | 479 | 110471 | 231 |
| _205_raytrace | 118 | 93 | 211 | 575724 | 2729 |
| _209_db | 138 | 113 | 251 | 123166 | 491 |
| _213_javac | 152 | 122 | 274 | 222891 | 813 |
| _227_mtrt | 111 | 81 | 192 | 677785 | 3530 |
| average | 153 | 125 | 278 | 286086 | 1029 |

3.3 Choosing a Coalescing Strategy

Another interesting question is, “Is our mechanism more suitable to combine with deferred coalescing, never coalescing, or immediate coalescing strategy?”

Remember that in automatic garbage collection environment, an allocator can affect two parts of program execution time, which are time for object allocation and time for garbage collection. An allocator using deferred coalescing or never coalescing strategy can provide very fast allocation while introducing longer garbage collection time. In contrast, an allocator using immediate coalescing strategy has slower allocation speed but result in shorter time for garbage collection.

To our knowledge, no previous research figured out that in garbage collection environment, whether fast allocation is more important or low fragmentation. Although deferred coalescing is generally considered the best strategy among the three, to what extent it wins over never coalescing or immediate coalescing strategy is not clear. Moreover, with our proposed Next Hit Table mechanism, many allocation request can be satisfied by a table lookup and then direct get the chunk from the right free list, even if immediate coalescing strategy is used. This let us to pose an ambitious assumption, immediate coalescing is the best coalescing strategy in the presence of Next Hit Table's support. We will prove this by experiment.

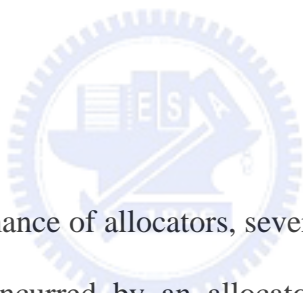


Chapter 4

Experiment

The goal of this chapter is to set up some experiment to see the following things. First, the comparison between different coalescing strategies using in size-ordered segregated list allocators is shown. Second, we want to see to what extent the performance will gain after adding NHT. Third, it is interesting that whether adding Next Try Table will really change the feasible coalescing strategy.

4.1 Methodology



For evaluating the performance of allocators, several methodologies have been proposed. For study the fragmentation incurred by an allocator, traditionally memory- trace-driven simulation is used. For study the allocation speed of an allocator, some research do simulation of the number of node being searched during object allocation and uses the number of node being searched to judge an allocator, while other research direct implement their allocators and uses execution time to evaluate the speed performance.

For our research, we do not look at fragmentation and speed separately like many previous research, since we are interested in the allocator's speed performance and the interaction between the allocator and garbage collector. In a programming language with automatic garbage collection, the seriousness of fragmentation is converted to execution time, that the higher the fragmentation, the greater the time will be spent on garbage collection since higher fragmentation results in more frequent invocation of garbage collection. Whether

an allocator is good should be judged by the time spent on object allocation plus garbage collection, since garbage collection is a very sophisticated process and thus difficult to simulate its speed performance. Therefore, we implement our allocator and measure the execution time spent on object allocation and garbage collection.

Since we want to use execution time to judge the performance of allocator, we must use some way to collect the time information. A traditional technique to measure time is to use operating system's system calls, but system calls are so expensive and object allocation activities occur very frequently, so the overhead using system calls will disturb the accuracy. Thus we decide to use alternative approach to collect execution time information.

Some processor have built in hardware event counter which counts elapsed clock cycles and special registers which can be used to record clock cycles information. And using a single assembly instruction can read the content of the register. Thus, we decide to use cycles as the measure of time.

4.2 Environment

Sun's CVM is chosen as the based Java virtual machine in our implementation because it is open-source and widely known. CVM is pure interpreter-based virtual machine intended to be used in embedded systems [8]. The underlying hardware is Pentium 4 and the chosen operating system is Debian Linux.

We have gathered many benchmarks, including SPECjvm98 and Embedded CaffeineMark. Embedded CaffeineMark is aimed for testing the performance of Java virtual machine in embedded environments. It a synthetic benchmark suite and only do very simple operation, such as looping, calling method, and allocated String. Hence it is not a

representative benchmark to reflect the object allocation behavior in real Java programs.

SPECjvm98 is an industry-standard benchmark suite for evaluating the performance of Java virtual machines [9]. Benchmarks in SPECjvm98 are for solving real-world problem and several of them are commercial applications. Thus, SPECjvm98 is chosen as our test benchmark suite, since we are interesting in the performance of dynamic object allocation. Among the benchmarks, `_222_mpegaudio` and `_228_jack` are not tested because of porting problems.

Table 4-1: Overview of SPECjvm98 benchmark

| Benchmarks | Short Description |
|----------------------------|---|
| <code>_201_compress</code> | Modified Lempel-Ziv method (LZW). Basically finds common substrings and replaces them with a variable size code. |
| <code>_202_jess</code> | Java Expert Shell System is based on NASA's CLIPS expert shell system. |
| <code>_205_raytrace</code> | A raytracer that works on a scene depicting a dinosaur. |
| <code>_209_db</code> | Performs multiple database functions on memory resident database. |
| <code>_213_javac</code> | This is the Java compiler from the JDK 1.0.2. |
| <code>_227_mtrt</code> | A raytracer that works on a scene depicting a dinosaur, where two threads each renders the scene in the input file time-test model. |
| <code>_228_jack</code> | A Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS). |

4.3 Experimental Results

We now give the experimental results and discuss on them. We will first clarify the impact of different coalescing strategies on size-ordered segregated list allocator. Second, we will show how well NHT improve object allocation performance, using different coalescing strategies. Then, we will prove our former assumption that with the power of NHT, immediate coalescing strategy comes back to flavor via empirically results. In the end, we will use experiment to find the number of free lists of size-ordered segregated list allocator resulting in the best performance using NHT. We will only show the results of `_205_raytrace` in this chapter because it is representative, and the results of other benchmarks are presented in the appendix.

4.3.1 Impact of Different Coalescing Strategies

Three important coalescing strategies, never coalescing, deferred coalescing, and immediate coalescing strategy, have been described in chapter 2. Generally speaking, never coalescing has the fastest allocation speed, but results in most serious fragmentation. Deferred coalescing strategy is believed to have lower fragmentation than never coalescing strategy and it still has fast allocation speed. Immediate coalescing strategy has poor allocation performance but on the other hand, it has the lowest fragmentation.

From Figure 4-1, we can see that immediate coalescing strategy has the worst allocation performance among the three strategies, and deferred coalescing has comparable allocation performance to never coalescing strategy. The allocation performance of deferred coalescing and never coalescing strategy are 2.5 time the allocation performance of immediate coalescing until the heap becomes too small. When heap size is less than about 2 times minimum heap

size, the allocation performance of deferred coalescing and never coalescing degrade sharply. This is because garbage collection is invoked excessively, and each last allocation before triggering the garbage collection is an unsatisfied allocation request, indicating that the allocator must traverse up to the last free list. On the other hand, the allocation performance of immediate coalescing strategy does not have much degradation.

Considering the influence of coalescing strategy on time spent on garbage collection, we can see from Figure 4-2 that immediate coalescing constantly result in far shorter garbage collection time, especially when heap size becomes too small. Never coalescing and deferred coalescing strategy bring unacceptable long garbage collection time because of excessive invocation of garbage collection.

Another interesting trend observed from Figure 4-2 in the figure is that within some range of heap size, the garbage collection performance does not improve as the heap size grows. This results from the way mark-sweep garbage collector to reclaim free space. In the sweep phase, it scans from one end of the heap to the other end, thus if the number of garbage collection invocation is the same, smaller heap size could have short garbage collection time.

When taking both object allocation time and garbage collection time into accounts (see Figure 4-3), deferred coalescing and never coalescing strategy have 2 times speedup over immediate coalescing strategy if heap is sufficient large. However, in very constrained heap size, immediate coalescing strategy is the only usable strategy.

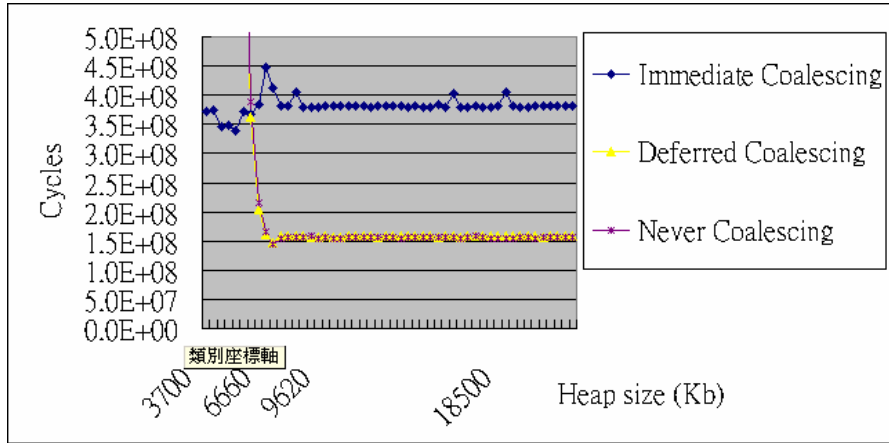


Figure 4-1: Object allocation time using different coalescing strategy

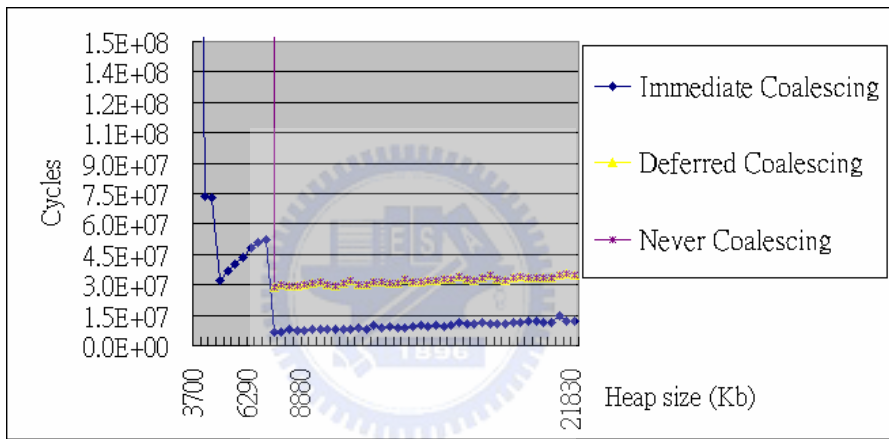


Figure 4-2: Garbage collection time using different coalescing strategy

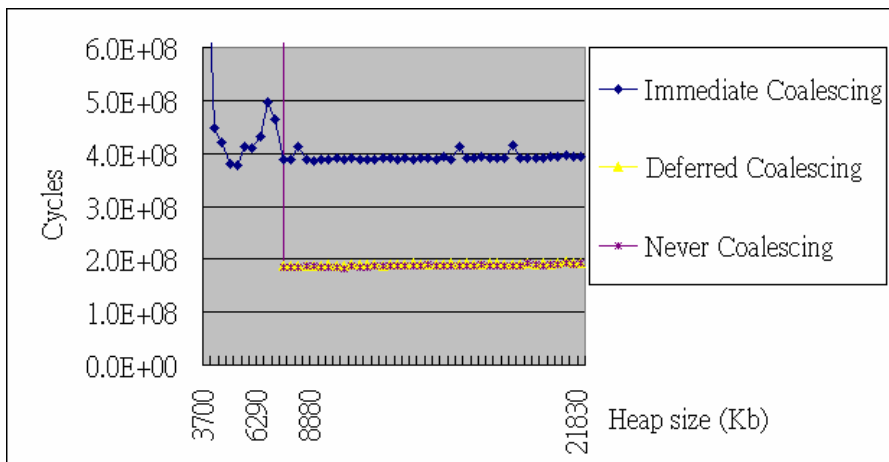


Figure 4-3: Heap management time using different coalescing strategy

4.3.2 The Effectiveness of Next Hit Table Mechanism

After our NHT is added, size-order segregated list allocator using either deferred coalescing, never coalescing, and immediate coalescing strategy has improvement in allocation performance and the improvement is most significant when applying NHT with immediate coalescing strategy (see Figure 4-4 and 4-8). This is because using immediate coalescing strategy makes many free lists holding small chunks more likely to become insufficient or empty after garbage collection, thus originally many allocation requests spend considerable traversal to locate the right free list to get a large enough chunk. The allocation performance using deferred coalescing and never coalescing strategy are still comparable after adding NHT. And, as expected, the presence of NHT does not effect garbage collection time (see Figure 4-5 and 4-9).

We can also observe that when heap size is less than a threshold, which is about 2 times minimum heap size, the allocation performance gap between the allocator using immediate coalescing with NHT and the allocator using immediate coalescing without NHT is closer in smaller heap size. This is because in such a constrained heap size, the time to update NHT occupies more fraction of heap management time.

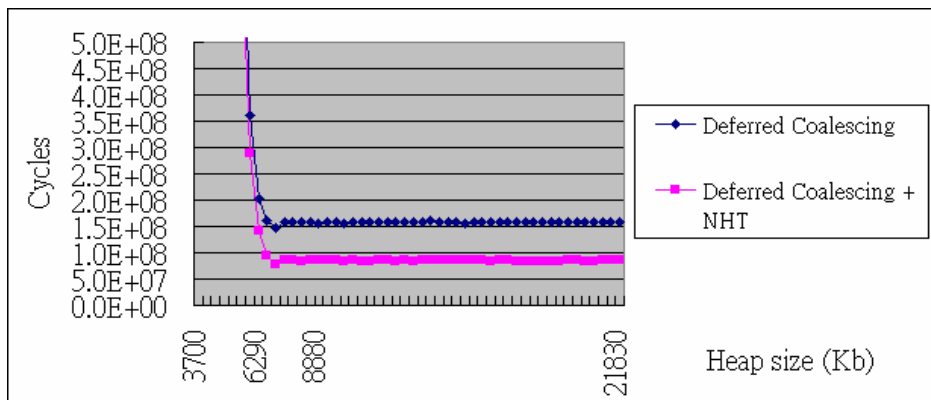


Figure 4-4: Object allocation time of using deferred coalescing strategy plus NHT

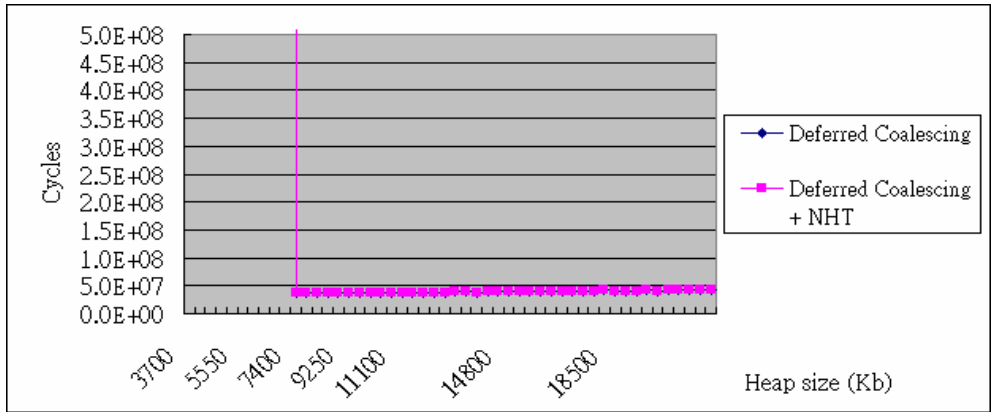


Figure 4-5: Garbage collection time of using deferred coalescing strategy plus NHT

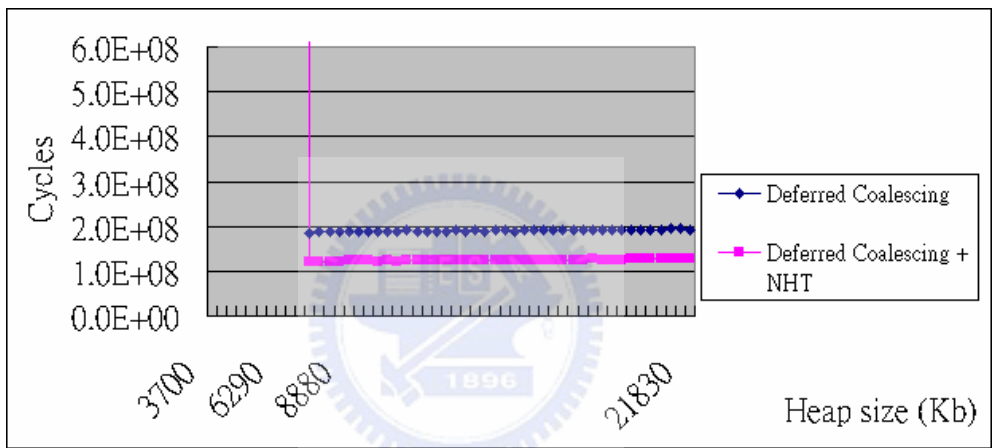


Figure 4-6: Heap management time of using deferred coalescing strategy plus NHT

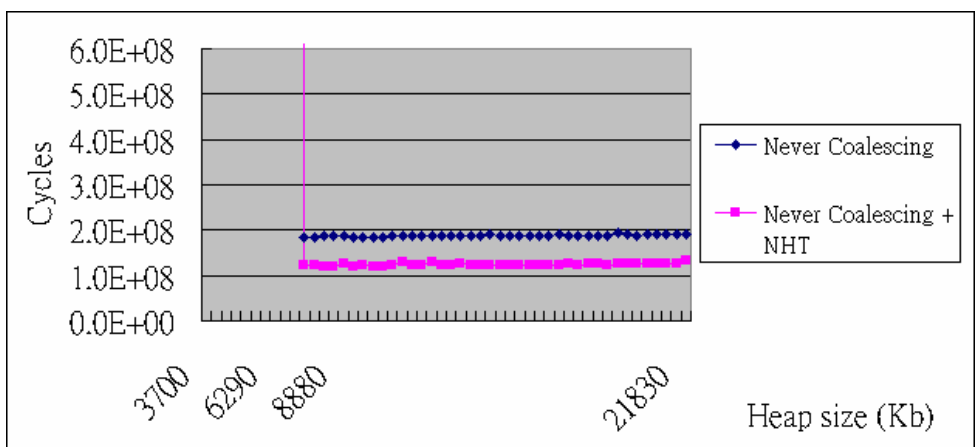


Figure 4-7: Heap management time of using never coalescing strategy plus NHT

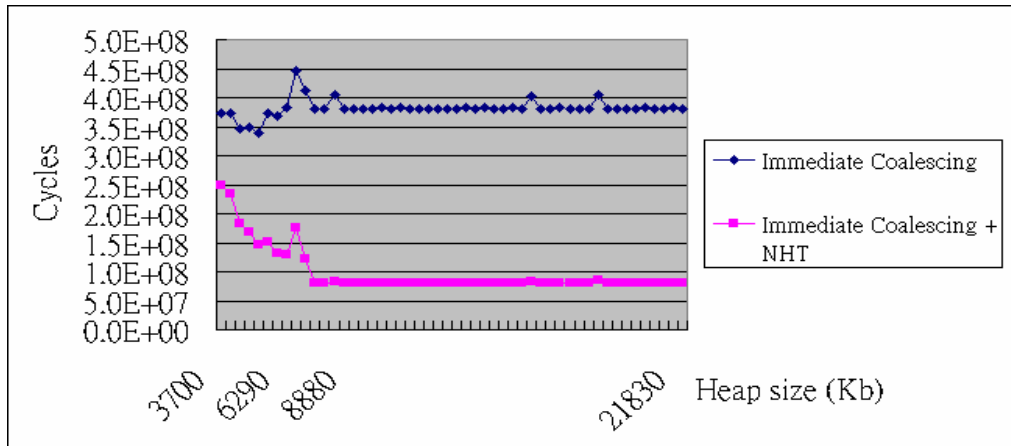


Figure 4-8: Object allocation time of using immediate coalescing strategy plus NHT

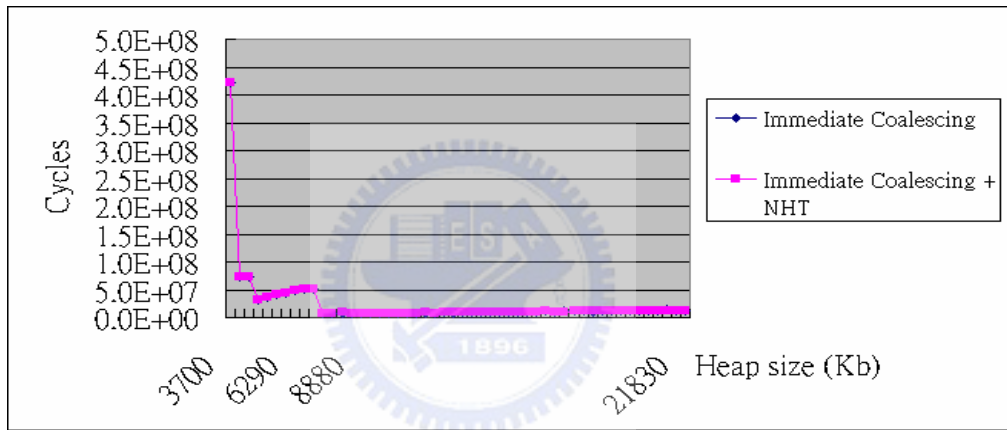


Figure 4-9: Garbage collection time of using immediate coalescing strategy plus NHT

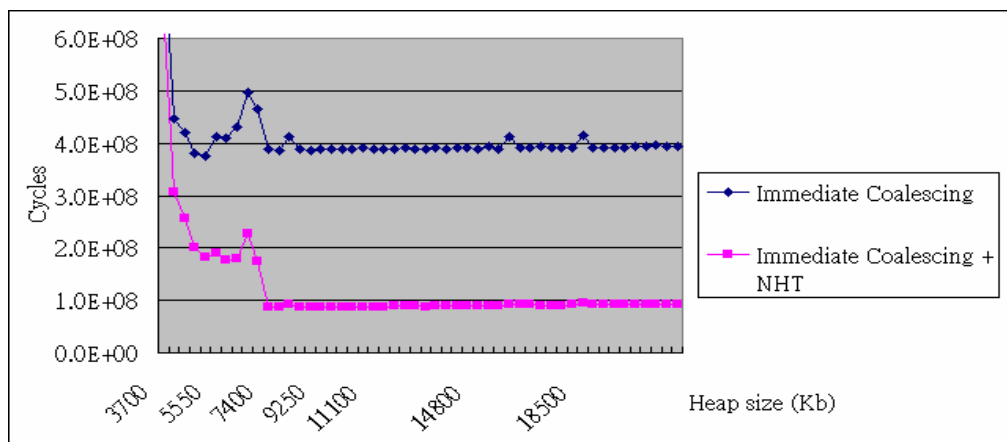


Figure 4-10: Heap management time of using immediate coalescing strategy plus NHT

4.3.3 NHT Leads to a Different Choice of Coalescing Strategy

As the experimental results shown in subsection 4.3.1, we already know that size-order segregated list allocator using deferred coalescing and never coalescing strategy outperform that using immediate coalescing, without our NHT mechanism. However, after adding NHT, the order changes dramatically. No matter the heap size is constrained or more sufficient, the allocator with NHT using immediate coalescing strategy constantly beats the other two strategies with NHT (see Figure 4-13). Although using deferred coalescing plus NHT has the fastest allocation speed because of the update of NHT is most infrequent if using deferred coalescing strategy, the allocation speed gap between using immediate coalescing strategy with Next Hit Table and using deferred coalescing strategy with Next Hit Table are too small to be observable. The advantage of low fragmentation of immediate coalescing strategy becomes a dominate factor because of NHT's ability to reduce many traversals during allocation.

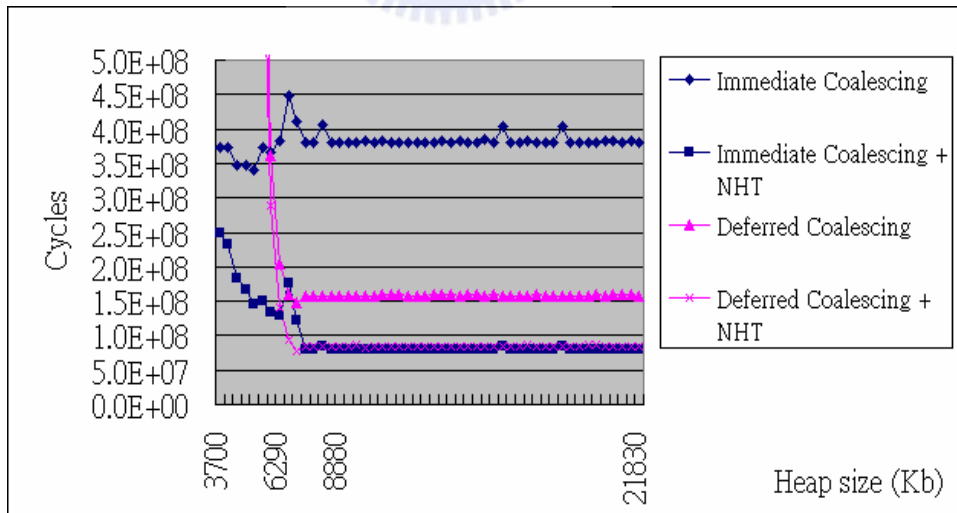


Figure 4-11: Object allocation time of 4 configurations

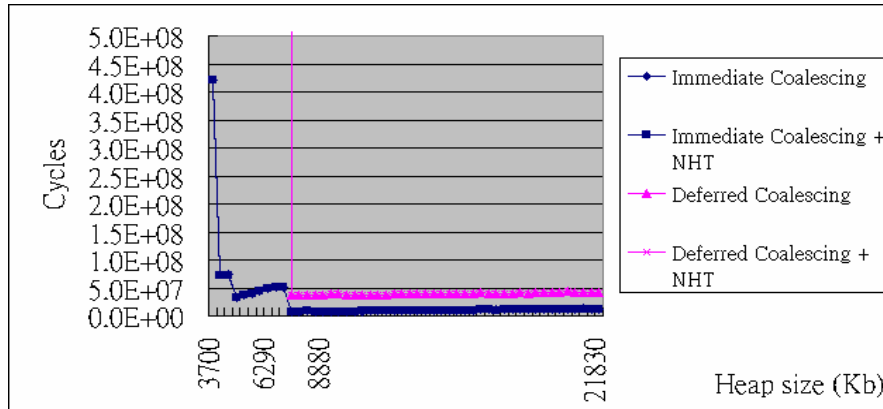


Figure 4-12: Garbage collection time of 4 configurations

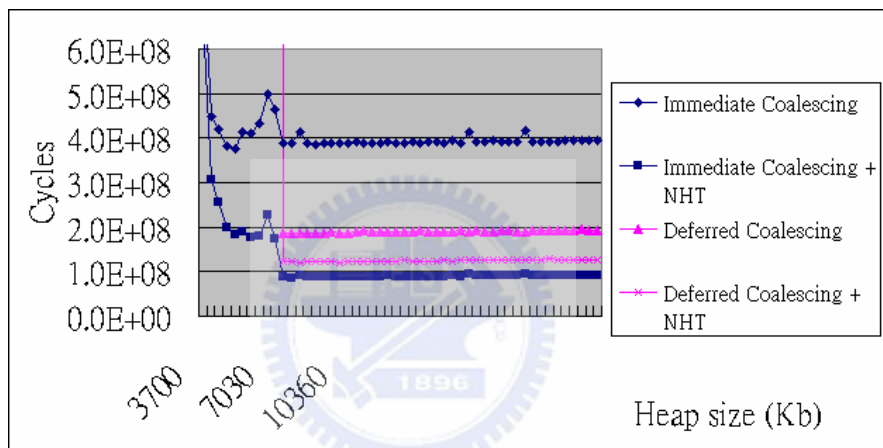


Figure 4-13: Total heap management time of 4 configurations

4.3.4 Determining the Number of Free Lists

The above result has told us size-order segregated list allocator with NHT plus immediate coalescing strategy is the best configuration. Another import parameter for size-ordered segregated list fit allocator is the number of free lists. The number of free lists determines a size such that all free chunks larger than this size are chained in the last free list. For our experimental environment where heap objects are multiple of 4-byte, having N free lists means all free chunks of sizes equal to or larger than 4N are chained in the last free list. A

good value N should be chosen such that it can result in a balance between time spent on header traversals plus NHT updates, and time spent on best-fitting search in the last free lists. We determine N by experiment.

As the result shown in Figure 4-14, we can observe that using more free lists gives better performance, until the number of free lists reaches 256. Using fewer free lists indicates that more time will be spent on best-fitting search in the last free list, and because immediate coalescing strategy tends to produce larger free chunks, N should not be chosen too small. Also, using more than 256 free lists does not give further improvement because there is rare free chunk larger than 1K bytes using immediate coalescing strategy. We can conclude that 256 free lists are enough.

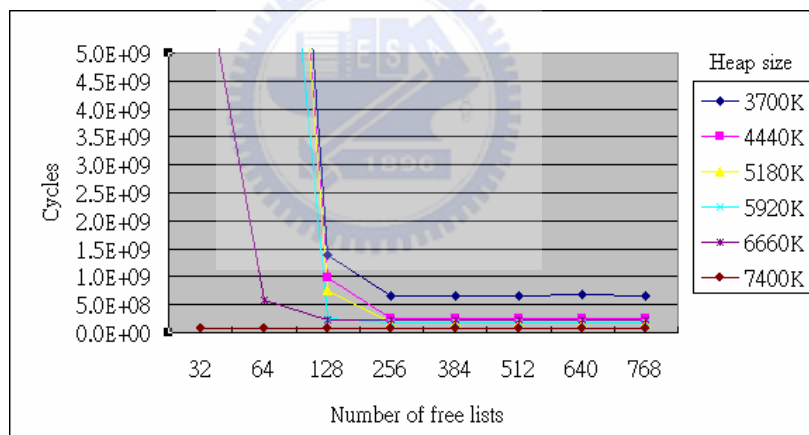


Figure 4-14: Using 256 free lists are good enough

Chapter 5

Conclusion

In this thesis, we proposed a mechanism, Next Hit Table, to accelerate object allocation for size-ordered segregated list object allocator in automatic garbage collection environment, with negligible space overhead. Our proposed mechanism is based on the observation that many free lists will become empty within the interval between two invocations of garbage collection. The concept of Next Hit Table is very simple, a mapping from allocation request for each size to the closest non-empty free list holding sufficient large chunks. The mechanism is very effective that the result shows that our proposed mechanism can improve the overall heap management performance by 100%.

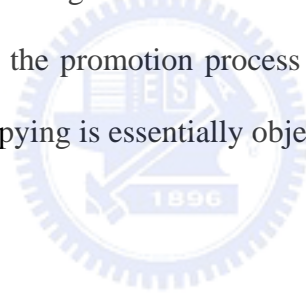
Another issue we studied is the coalescing strategies of contiguous free chunks. Although deferred coalescing is generally considered a better strategy than never coalescing strategy and immediate coalescing strategy, we showed that with the power of Next Hit Table mechanism, immediate coalescing is better, rather than deferred coalescing strategy. Although the update of Next Hit Table is more frequent if using immediate coalescing strategy, the allocation speed gap between using immediate coalescing strategy with Next Hit Table and using deferred coalescing strategy with Next Hit Table are very small. Such an upside-down result tells us that if allocation can be done very fast then fragmentation is the most important problem in garbage collection environment.

Explicitly managing heap and incremental garbage collection environment also have dynamic object allocation, thus they can benefit from the Next Hit Table mechanism. However, because dead objects are de-allocated using explicit de-allocation statement in

explicit heap management environment, free chunks of each sizes are less likely to become insufficient due to the locality of object size. Therefore, the benefit of using Next Hit Table will be less.

Moreover, since using incremental garbage collector means the heap space might still be sufficient at garbage collection, free chunks of each sizes are less likely to become insufficient. Furthermore, because not all dead objects at garbage collection triggering point will be collected at once, more time is spent on garbage collection due to more frequent garbage collection. These two factors make the benefit of Next Hit Table less if using incremental garbage collector.

Our proposed mechanism might also be incorporated in generational mark-sweep garbage collector to accelerate the promotion process which copies live objects from young space to old space, since the copying is essentially object allocation in old space.



References

- [1] Wilson, P. R., et al. “Dynamic Storage Allocation – A Survey and Critical Review”. In *Proceedings of 1995 International Workshop on Memory Management*, Kinross, Scotland, UK, September 27-29, 1995.
- [2] Jones, R., Lins, R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley, 1996.
- [3] Arnold, K., et al. *The Java Programming Language*, Addison Wesley, 3rd edition, 2000.
- [4] Lindholm T., Yelling, F., *The Java Virtual Machine Specification*, Addison Wesley, 2nd edition, 1999.
- [5] Johnstone, M. S. *Non-Compacting Memory Allocation and Real-Time garbage Collection*, University of Texas in Austin, PhD’s Dissertation, 1996.
- [6] Appel, A. W., Palsberg J. *Modern Compiler Implementation in Java*, Cambridge University Press, 2nd edition, 2002.
- [7] Fong, A. S., Li, R. C. L. “Dynamic Memory Allocation/Deallocation Behavior in Java Programs”. In *Proceedings of 2002 IEEE Region 10 Conference on Computers, Communications, Control and Power Engineering*, October 28-31, 2002.
- [8] Sun Microsystems Inc. CVM. <http://java.sun.com>.
- [9] Standard Performance Evaluation Corporation. SPECjvm98. <http://www.spec.org>.

Appendix: Complete Experimental Results

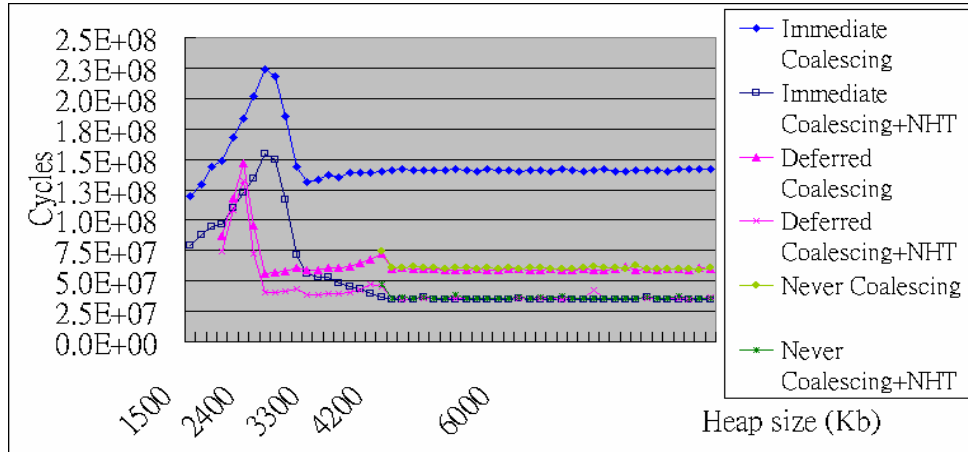


Figure A-1: Object allocation time of different allocators (_202_jess)

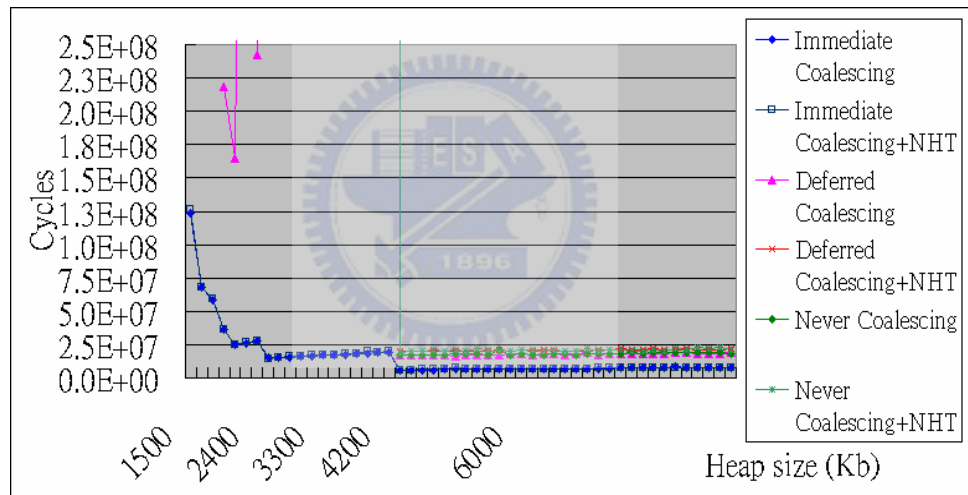


Figure A-2: Garbage collection time of different allocators (_202_jess)

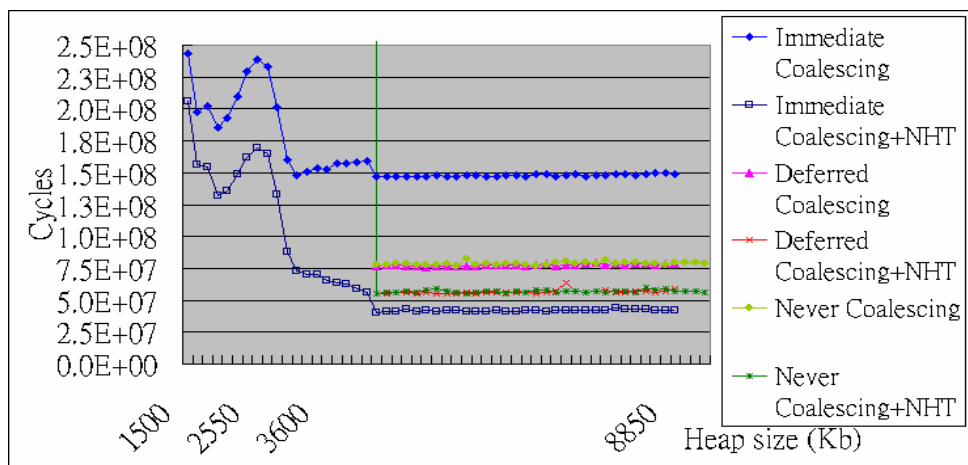


Figure A-3: Total heap management time of different allocators (_202_jess)

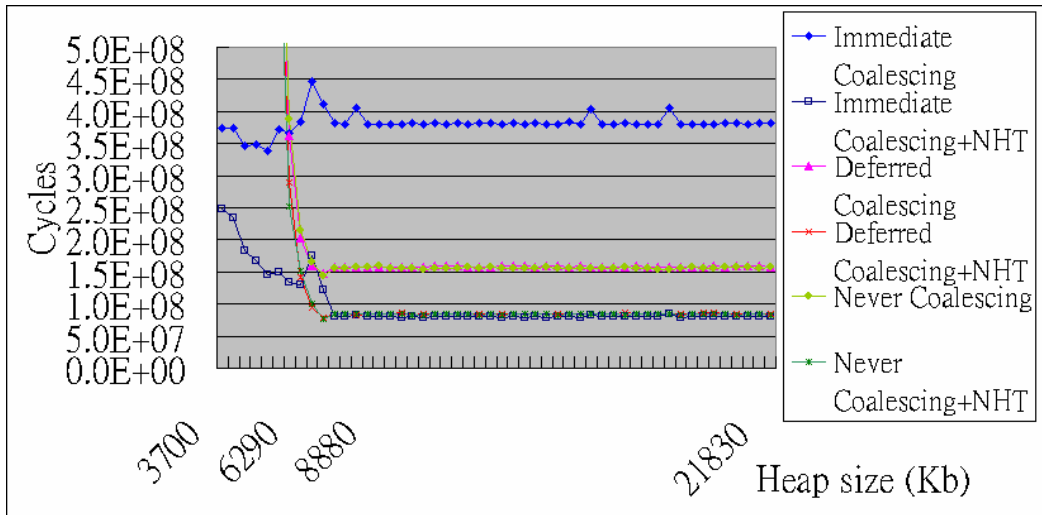


Figure A-4: Object allocation time of different allocators (_205_raytrace)

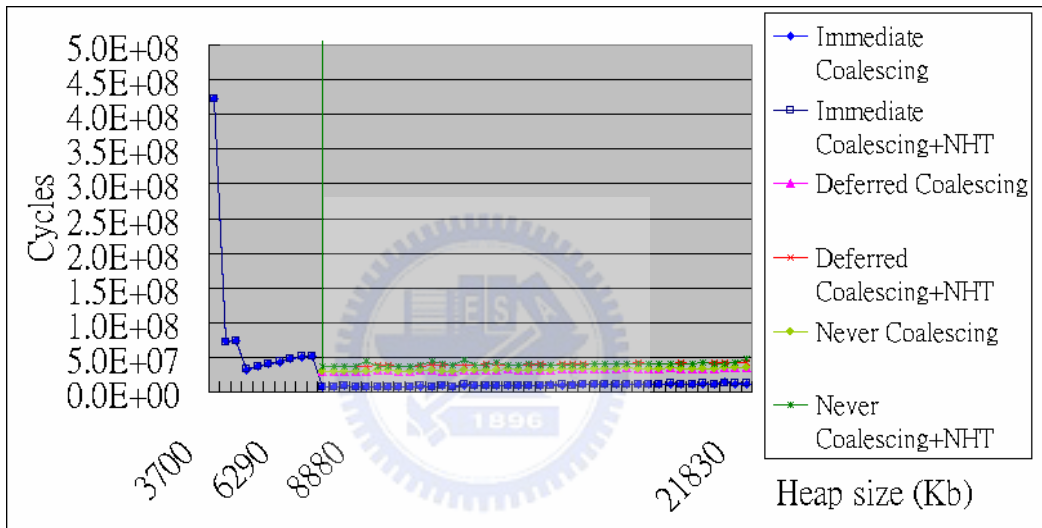


Figure A-5: Garbage collection time of different allocators (_205_raytrace)

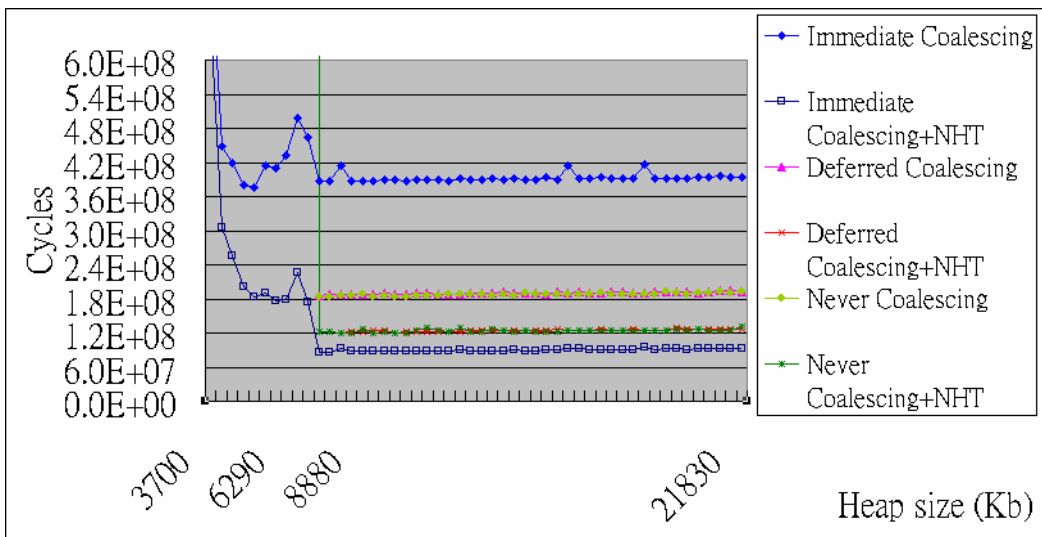


Figure A-6: Total heap management time of different allocators (_205_raytrace)

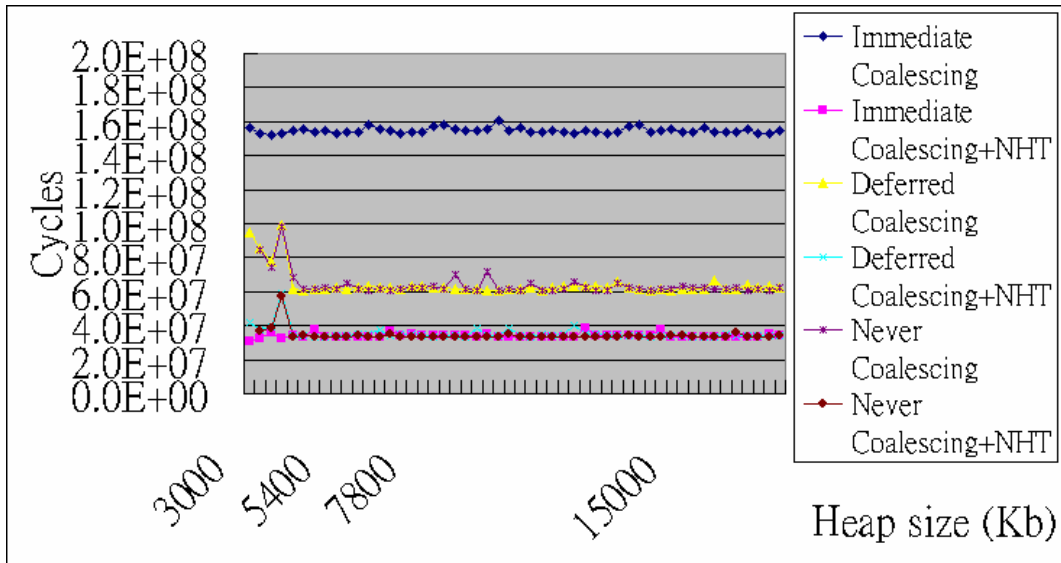


Figure A-7: Object allocation time of different allocators (_209_db)

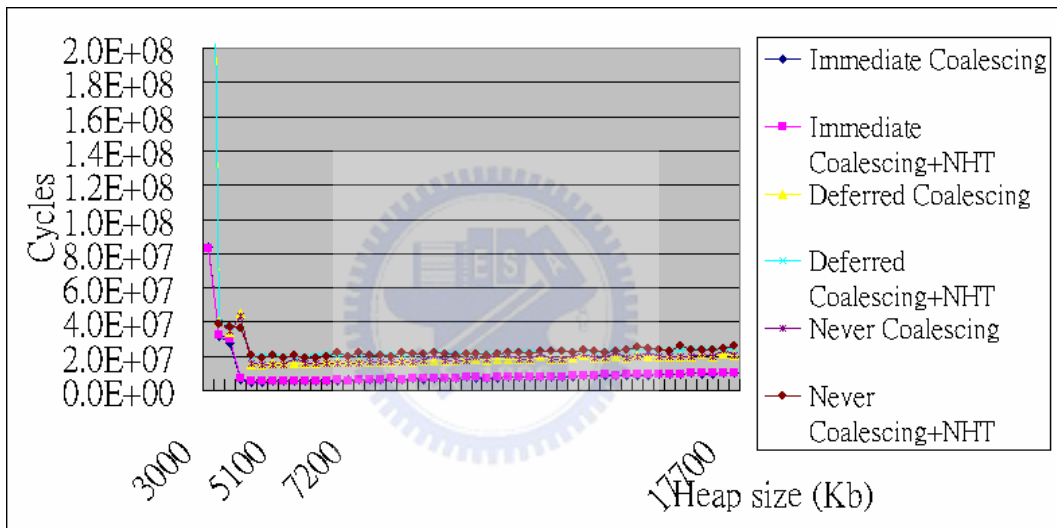


Figure A-8: Garbage collection time of different allocators (_209_db)

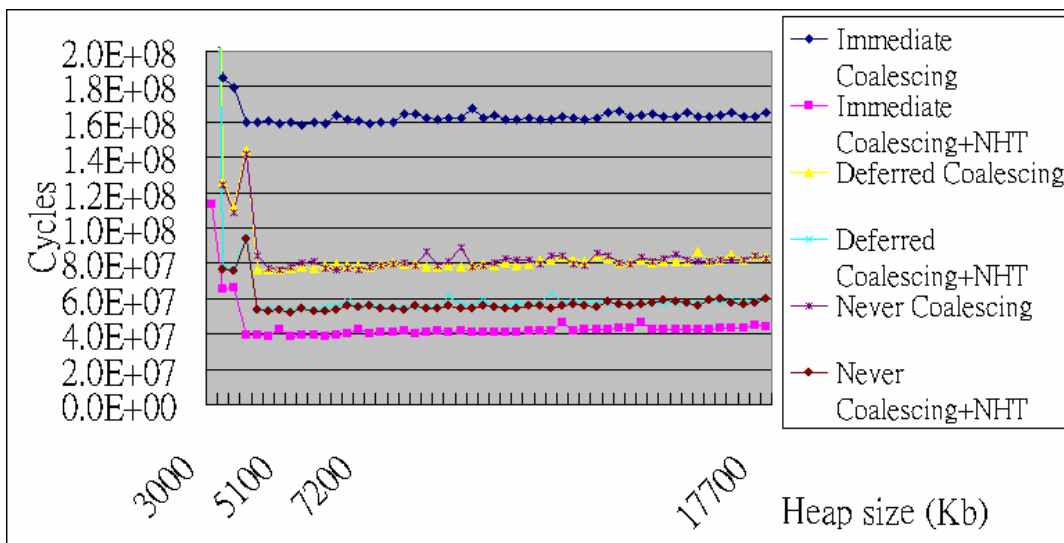


Figure A-9: Total heap management time of different allocators (_209_db)

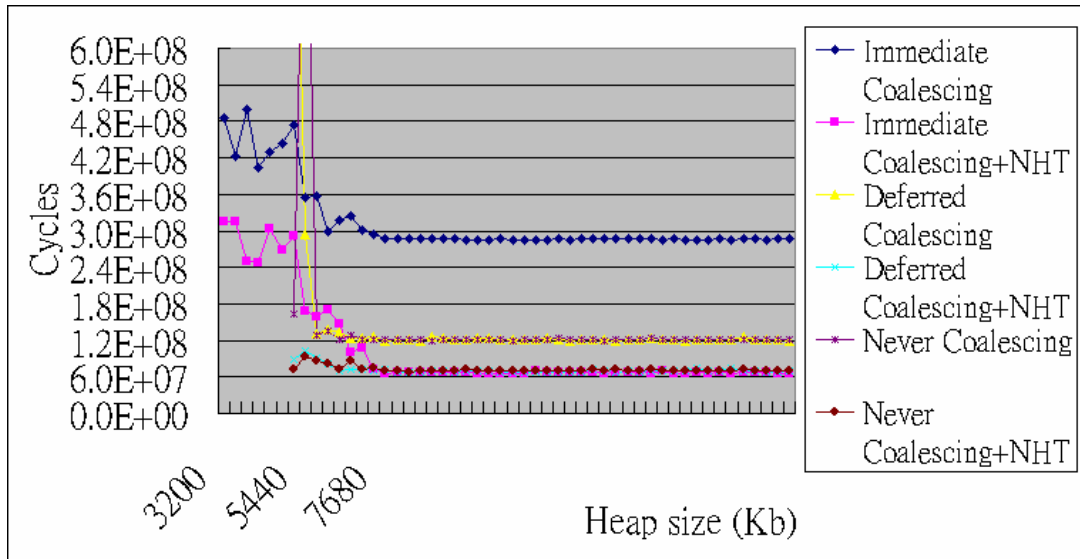


Figure A-10: Object allocation time of different allocators (_213_javac)

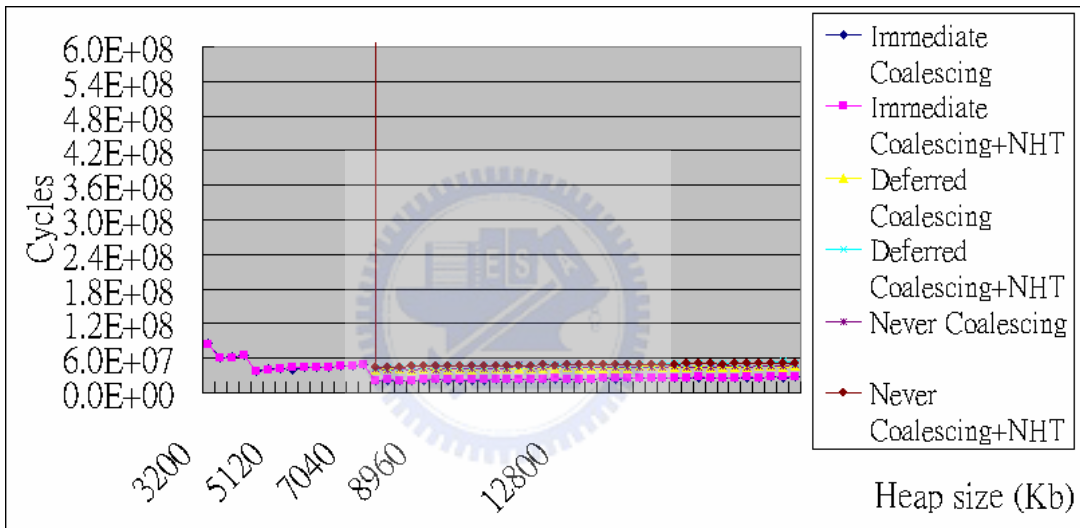


Figure A-11: Garbage collection time of different allocators (_213_javac)

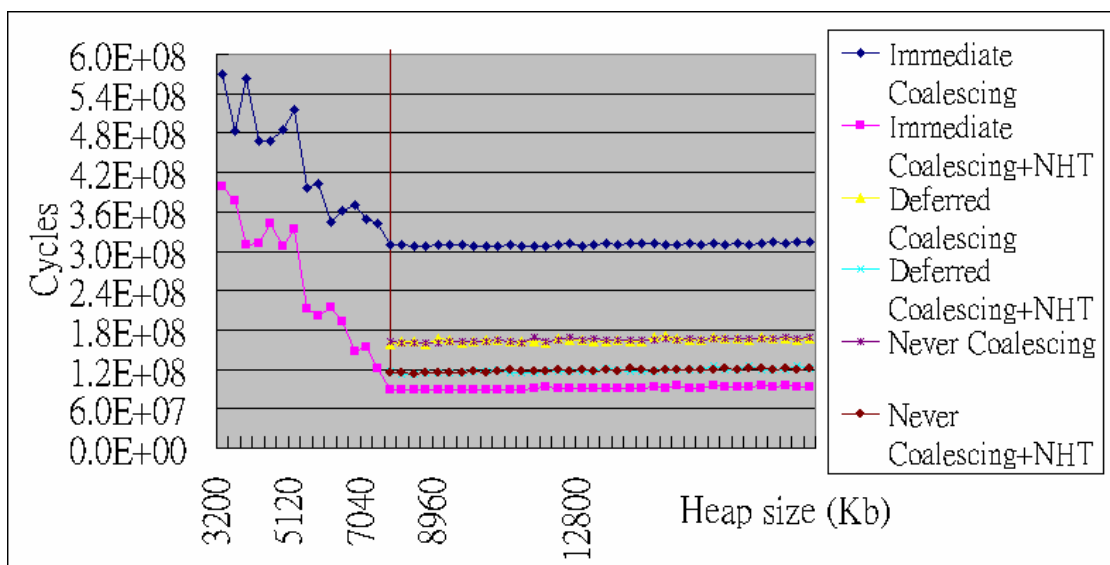


Figure A-12: Total heap management time of different allocators (_213_javac)

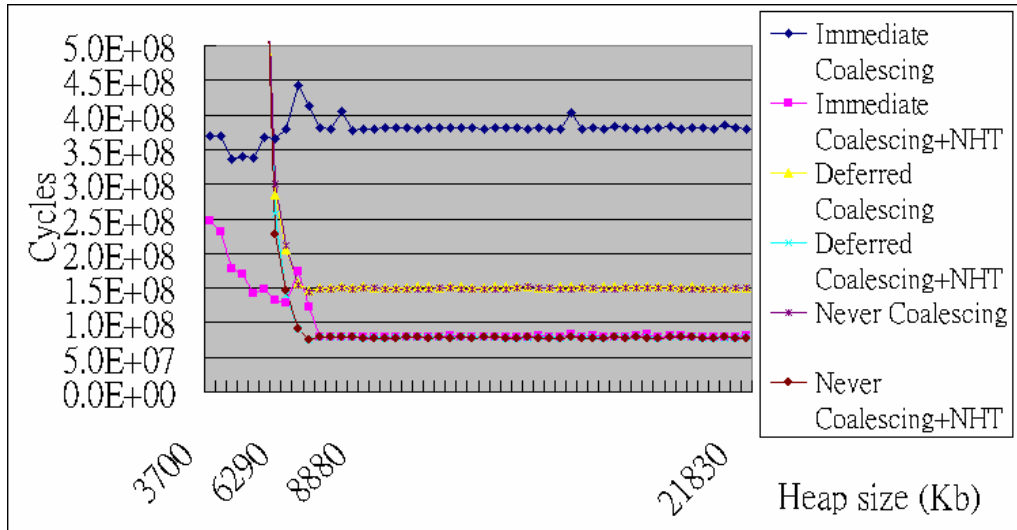


Figure A-13: Object allocation time of different allocators (_227_mtrt)

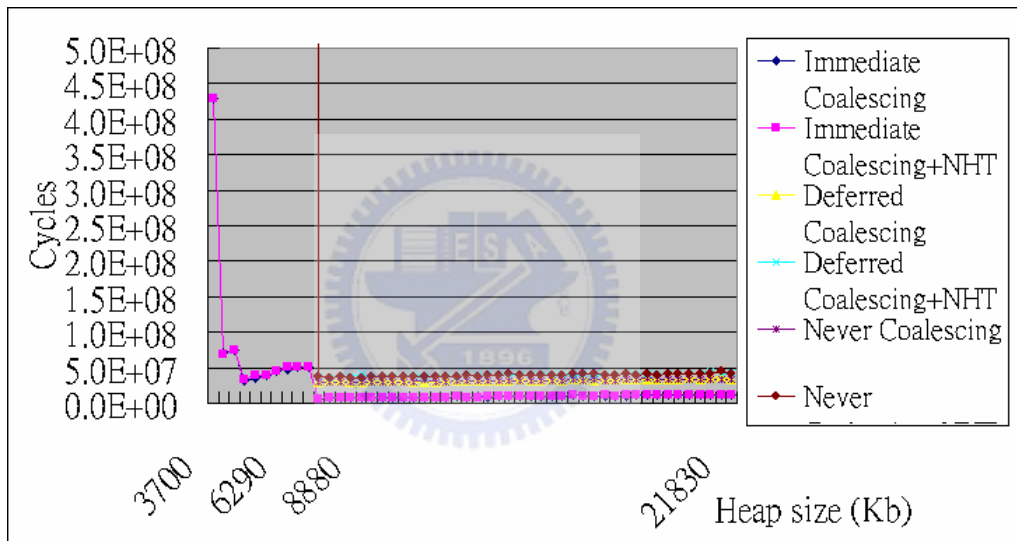


Figure A-14: Garbage collection time of different allocators (_227_mtrt)

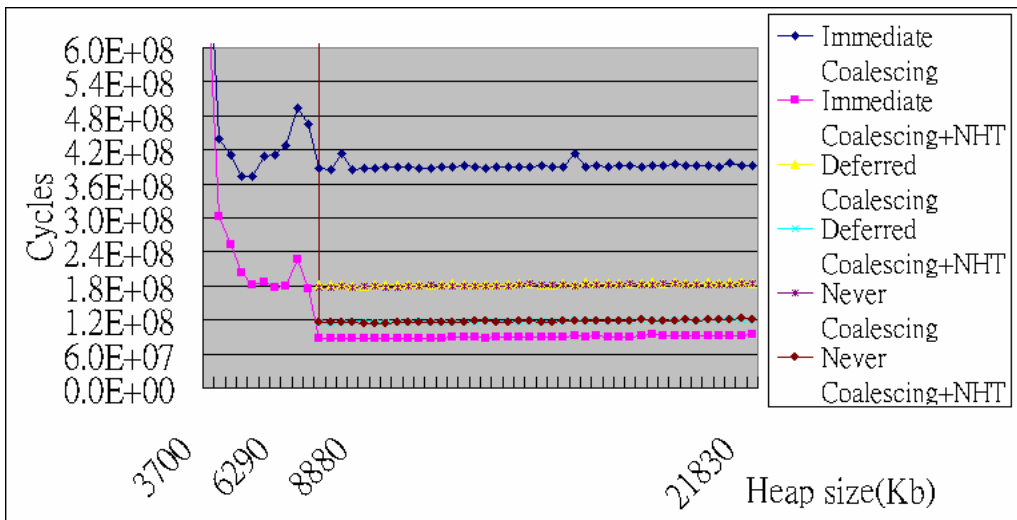


Figure A-15: Total heap management time of different allocators (_227_mtrt)