공학박사학위논문

가상머신의 메모리 관리 최적화

Optimizing Memory Management in Virtual Machine

2014 년 2 월

서울대학교 대학원

전기 컴퓨터 공학부

최 형 규

공학박사학위논문

가상머신의 메모리 관리 최적화

Optimizing Memory Management in Virtual Machine

2014 년 2 월

서울대학교 대학원

전기 컴퓨터 공학부

최 형 규

가상머신의 메모리 관리 최적화

Optimizing Memory Management in Virtual Machine

지도교수 문 수 묵

이 논문을 공학박사 학위논문으로 제출함

2013 년 12 월

서울대학교 대학원

전기 컴퓨터 공학부

최 형 규

최형규의 공학박사 학위논문을 인준함

2013 년 12 월

위 원 장 백 윤 흥 (인)

부위원장 문 수 묵 (인)

위 원 이 혁 재 (인)

위 원 이 재 진 (인)

위 원 김 수 현 (인)

Abstract

Optimizing Memory Management in Virtual Machine

Hyung-Kyu Choi
School of Electrical Engineering and Computer Science
The Graduate School
Seoul National University

Memory management is one of key components in virtual machine and also affects overall performance of virtual machine itself. Modern programming languages for virtual machine use dynamic memory allocation and objects are allocated dynamically to heap at a higher rate, such as Java. These allocated objects are reclaimed later when objects are not used anymore to secure free room in the heap for future objects allocation. Many virtual machines adopt garbage collection technique to reclaim dead objects in the heap. The heap can be also expanded itself to allocate more objects instead. Therefore overall performance of memory management is determined by object allocation technique, garbage collection and heap management technique. In this paper, three optimizing techniques are proposed to improve overall performance of memory management in virtual machine. First, a lazy-worst-fit object allocator is suggested to allocate small objects with little overhead in virtual machine which has a garbage collector. Then a biased allocator is proposed to improve the performance of garbage collector itself by reducing extra overhead of garbage collector. Finally an ahead-of-time heap expansion technique is suggested to improve user responsiveness as well as overall performance of memory management by suppressing invocation of garbage collection. Proposed optimizations

are evaluated in various devices including desktop, embedded and mobile, with different virtual machines including Java virtual machine for Java runtime and Dalvik virtual machine for Android platform. A lazy-worst-fit allocator outperform other allocators including first-fit and lazy-first-fit allocator and shows good fragmentation as low as first-fit allocator which is known to have the lowest fragmentation. A biased allocator reduces 4.1% of pause time caused by garbage collections in average. Ahead-of-time heap expansion reduces both number of garbage collections and total pause time of garbage collections. Pause time of GC reduced up to 31% in default applications of Android platform.

Keywords: optimization, object allocation, garbage collection, heap management, virtual machine, memory management

Student Number: 2002-30447

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent decades, virtual machine are becoming more common and widely adopted in various environment from embedded to server environment and most of modern computing devices support virtual execution environment. Java virtual machine [1] is one of popular virtual machines and available for various computing devices including low-end smartcard, digital TV, computer and high performance enterprise server. Dalvik is another well known virtual machine recent years. Dalvik virtual machine [2] is a core execution engine of Android [3] operating system for mobile devices including smartphone and table computers. By the year 2013, over 900 million Android devices have been activated worldwide [4] and most of web servers employ Java virtual machine to support Java language for server programming. And more virtual machines are employed in consumer appliances, such as digital TV [5], to provide user interactive services for individual users and service providers. Therefore virtual machine is very common nowadays and most of users who use smartcards, smartphones, computers, televisions or any kind of computing devices are already using some

virtual machines directly or indirectly.

Although most of users do not recognize the presence of virtual machine, user experience on those computing devices is affected by virtual machine, because overall performance of the device is determined by virtual machine when virtual machine plays a essential role in running applications. Therefore performance of virtual machine is a very important aspect as well as functions that virtual machine provides.

1.1 The need of optimizing memory management

Memory management module is one of key components in virtual machine and affects overall performance of virtual machine. Modern programing languages for virtual machine use dynamic memory allocation and objects are allocated dynamically to heap at a higher rate, such as Java. These allocated objects are reclaimed later when objects are not used anymore to secure free room in the heap for future objects allocation and many virtual machines adopt garbage collection technique to reclaim dead objects in the heap. Instead the heap can be expanded itself to allocate more objects and the heap itself is also allocated from memory. Since all memory management discussed above occur at runtime, efficiency of memory management affects the performance of virtual machine directly.

Overall performance of memory management is basically determined by object allocation technique, garbage collection and heap management technique. However each memory management technique is intricately related with each other and it is very hard to predict combined performance of memory management. Even worse memory management itself is also affected by other components of virtual machine as well as underlying hardware such as memory

hierarchy including caches. Behavior of applications also affects performance of a specific memory management technique and we cannot always avoid worst case situation unless we can predict future behavior of applications. Therefore there is always a need for optimizing memory management to improve performance of virtual machine as environments change, including hardware, behavior of applications, virtual machine and etc. This paper will discuss memory management in widely used environments and will propose optimizations to improve memory management in real devices.

1.2 Outline of the Dissertation

The rest of this thesis is organized as follows. Virtual machine and memory management are described in detail and problems with existing memory management in virtual machine are discussed and defined in chapter 2. Three optimizing techniques are introduced to enhance overall performance of memory management in virtual machine. Chapter 3 addresses a fast and efficient object allocator and proposes a lazy worst fit allocator with evaluation. After addressing the overhead of generational garbage collector, biased allocator is introduced to relieve the overhead in chapter 4. In chapter 5, ahead-of-time heap expansion technique is proposed and evaluated to improve overall performance of memory management through carefully selected but aggressive heap management. Then I summarizes proposed techniques with conclusions and discusses future works in chapter 6

Chapter 2

Backgrounds

2.1 Virtual Machine

A virtual machine is a software program that implements a machine and is capable of running software programs. There are known to be two kinds virtual machine including system virtual machine and process virtual machine. [6] In this paper, I'm going to deal with only process virtual machine and I will use term *virtual machine* to refer process virtual machine. This kind of virtual machine provides a platform-independent programming environment by abstracting underlying hardware. Therefore programs written for the virtual machine can be ran on any devices where the virtual machine is available.

There are a variety of process virtual machine available but two of virtual machine are going to be described in this section to provide short backgrounds for the remaining of this paper. One is famous Java virtual machine (JVM) [1] and another is Dalvik [2] virtual machine in Android [3] platform. Although two virtual machine are totally different virtual machine, both virtual machine have

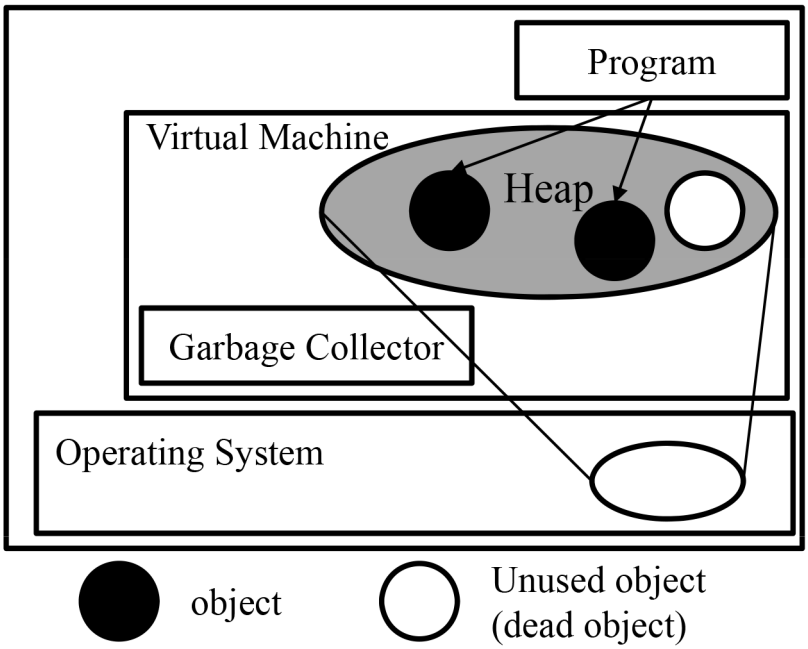

Figure 2.1 Virtual machine, heap and objects

some similarities in memory management, because applications are written in same language, i.e. Java language [7]. In the following section, we are going to describe memory management in virtual machine.

2.2 Memory management in virtual machine

Java is a class-based object-oriented programming language which allocates objects frequently. Java also adopts an automatic memory management technique called garbage collection [8] to reclaim unused objects automatically. Therefore a program written in Java allocates objects frequently and those objects are reclaimed automatically when they are not used anymore.

Figure 2.1 depicts abstract view of memory management in virtual machine. Virtual machine maintains a large pool of memory called the heap. The heap

could be a part of whole memory maintained by operating system. A program run on virtual machine allocates objects from the heap and uses them. Unused objects remains in the heap until being freed by garbage collector. Those unused objects, i.e. dead objects, are reclaimed later by a garbage collector which is an essential part of virtual machine. The size of heap can be increased if there is no room for new objects requested by the program. Then the heap grows to satisfy allocation request of the program by allocating more memory from operating system. In vice versa, the heap can also shrink if there is sufficient unused room. In short, memory management in virtual machine can be classified into three operation including object allocation, garbage collection and heap resizing. We will propose optimizing approaches for each operation in following sections.

Chapter 3

Lazy Worst Fit Allocator

3.1 Introduction

Modern programming languages use dynamic memory allocation [9]. As applications become more complex and use more of an object-oriented programming style, memory objects are allocated dynamically at a higher rate. This requires fast dynamic memory allocation.

Memory allocation should also be space efficient. A request for memory allocation cannot be satisfied when there is no free memory chunk that can accommodate the requested memory. This may happen even when the total amount of unused memory is larger than the amount of memory requested, due to fragmentation. In fact, fragmentation is the single most important reason for the wastage of memory in an explicitly managed heap or a heap managed by a non-moving garbage collector.

There are many approaches to implementing memory allocators, which exhibit different degrees of fragmentation and different allocation speeds. A com-

mon approach is maintaining a linked list of free memory chunks, called the free list, and searching the free list for a chunk that can satisfy a memory allocation request based on the fitting policy, such as first fit (FF), best fit or worst fit. Memory allocation using FF and best fit tends to have relatively low fragmentation [9], yet searching the free list has a worst-case linear time complexity.

In garbage-collected systems there are compacting garbage collection techniques such as copying collection [10] or mark-and-compact collection [11]. In such systems used and unused memory are not interleaved, so fragmentation does not exist. Thus, the obvious and fastest way to allocate memory is by simply incrementing an allocation pointer for each allocation.

There is a memory allocation approach for the free lists, motivated by the fast memory allocation of compacting collection, such that pointer increment is used as the primary allocation method, with FF, best fit or even worst fit as the backup allocation method [12]. This approach was called lazy fit, in the sense that finding a fitting memory chunk is delayed until really necessary. Preliminary experimental results simulating the traces of memory requests showed that the approach is promising since most memory allocations can be done via pointer increments.

This paper attempts to confirm the practical usefulness of lazy fits in the context of Java. We propose lazy worst fit (LWF) as a memory allocation method for a Java virtual machine with non-moving garbage collection. We implement LWF on a working Java virtual machine and evaluate its allocation speed and fragmentation, compared with lazy first fit (LFF) and FF. This chapter is organized as follows. Section 3.2 discusses memory allocation using conventional fits. Section 3.3 reviews memory allocation using lazy fits and proposes the LWF for Java. Section 3.4 presents our experimental results. Finally,

the paper is summarized in Section 3.5.

3.2 Allocation with fits

Before discussing memory allocation using lazy fits, we first discuss memory allocation using conventional fits.

In the simplest implementation of conventional fits, a single free list of free memory chunks is maintained. When a request for allocating memory is made, an appropriate free memory chunk is found from the free list after traversing the list from the head free chunk. The exact manner in which an appropriate free memory chunk is found depends on the fitting policy.

With first fit, the free list is searched sequentially and the first free memory chunk found that is able to satisfy the memory allocation request is used. This can be further divided into several types according to the order in which the free list is sorted: address-ordered, last-in-first-out (LIFO) and first-in-first-out (FIFO).

The address-ordered FF is known to have the least fragmentation, with the LIFO FF being noticeably worse. There is evidence that the FIFO FF has as little fragmentation as the address-ordered FF [13].

With best fit, the free memory chunk with the smallest size that is able to satisfy the memory allocation request is used. Along with FF, this policy is known to have little fragmentation in real programs.

In worst fit, the largest free memory chunk is used to satisfy the memory allocation request on the contrary to best fit. This policy alone is known to have much worse fragmentation than FF or best fit, so it is rarely used in actual memory allocators. However, worst fit can be useful when combined with lazy fit, which is explained in the next section.

The approach of using a single free list to keep track of the free memory chunks is very slow owing to a worst-case linear time complexity, especially if best fit or worst fit is done. So in actual implementations of modern memory allocators, more scalable implementations, such as segregated free lists, Cartesian trees and splay trees [9], are used for memory allocation.

Segregated free lists are the most common and simplest approach used in actual implementations [14, 8]. It divides memory allocation request sizes into size classes and maintains separate free lists containing free memory chunks in the size class. This approach, also called segregated fits, still has a worst-case linear time complexity, yet its allocation cost is known to be not much higher than that of a copying collector [8]. However, in our experiments unacceptably long search times for the segregated free lists do occur in practice (see Section 3.4), which indicates that the linear time complexity for accessing the free lists can be a real obstacle to fast allocation with fits.

3.3 Lazy fits

Memory allocation using lazy fit uses pointer increments[1] as the primary allocation method and conventional fits as the backup allocation method.

To be precise, an allocation pointer and a bound pointer are maintained for a current free space area. When a memory allocation request is made, the allocation pointer is incremented and it is checked against the bound pointer to see whether the memory allocation request can be satisfied. If it is satisfied, the memory that was pointed out by the allocation pointer before it was incremented is returned. Otherwise, conventional fit allocation is used to obtain a free memory chunk to be used as the new free space area, and the remainder of

[1]Pointer decrements can also be used for implementing lazy fits, but we assume pointer increments in this paper.

the former free space area is returned to the free list. The new free space area would then be used for allocating objects with pointer increments.

This is rather similar to the typical allocation algorithm used in systems with compacting garbage collectors, which also use pointer increments to allocate memory. The latter avoids a backup allocation method because there is no fragmentation, because compacting garbage collectors leave only one free chunk after compaction.

The fit method used for the backup allocation does not have to be any particular one. It could be first fit, best fit or even worst fit. These will be called lazy first fit (LFF), lazy best fit and lazy worst fit (LWF) respectively. In fact, it does not matter which approach is used for the backup allocation method as long as it is able to handle fit allocation. Using first fit or best fit would probably have the advantage of less fragmentation, while using worst fit would probably result in larger free space areas, which would result in more memory allocations using pointer increments for faster speed.

Figure 3.1 shows a simple example of how a lazy address ordered first fit would work.[2] Figure 3.1a shows the initial state when the LFF allocator starts allocating in a new free space area. The allocation and bound pointers point to the start and the end of the free space area respectively.

Allocation occurs within the given free space area, as in Figure 3.1b, incrementing the allocation pointer appropriately to accommodate each memory allocation request. This goes on until the free space area is no longer able to satisfy the memory allocation request, i.e. the space remaining in the free space area is smaller than that needed by the caller. Then, we put what remains of the current free space area back into the free list and search the free list for

[2]However, we used a segregated FF instead of an address-ordered FF in the experiment, because the address-ordered FF is very slow. The only difference is how to manage the free list.

a new free space area which can be used to allocate memory. The allocation and bound pointers are set to the start and the end of the new free space area respectively, and the cycle begins anew.

Figure 3.1c shows the state of the heap after the old free space area, marked as 'old', is put back into the free list, and the allocation and bound pointers point to the boundaries of the new free space area, marked as 'new', which had just been extracted from the free list using FF.

To speed up memory allocation using a lazy fit even more, the allocation and bound pointers could be held in two reserved global registers. This allows one to allocate memory without touching any other part of the memory, except for the memory we are allocating, in the common case. This is in contrast to many other allocation algorithms which usually require at least some manipulation of the data structure in the memory.

Lazy fit also has the potential to be faster than segregated storage since it

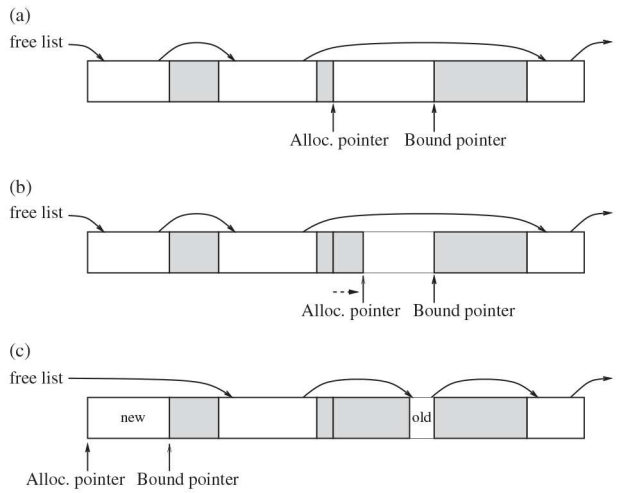

Figure 3.1 An example of a lazy address-ordered first fit (shaded areas denote used memory). (a) Initial state; (b) allocation using pointer increments; (c) the state after the new free space area was found by first fit.

has no need to decide size classes. Objects allocated closely together in time would probably be used together, so there could also be a beneficial effect on cache performance, since lazy fit would tend to group together objects that are consecutively allocated.

3.3.1 Lazy worst fit

In order to use lazy fit for garbage-collected systems such as Java, we made two engineering choices. First, we propose using worst fit in order to reduce the search time for the free lists. So, after each garbage collection we sort the free memory chunks in the free list in decreasing order of sizes. By using worst fit, a single comparison suffices to find out whether there is a chunk in the sorted free list which is able to accommodate the requested object, whereas alternative methods such as FF or best fit may require many comparisons to ascertain whether such a chunk exists.

Second, the previous free space area which had been unable to accommodate the requested object is discarded and not put back into the free list when using lazy worst fit. One reason is that inserting it into a sorted free list would introduce $O(n)$ time complexity [15] when we use a simple singly linked list, while all operations in LWF, including pointer increments and worst fits, can be done in $O(1)$ time. Giving up the previous free area will keep the O(1) allocation speed, which would obviate the worst-case linear time complexity of accessing the free list. Since the free list is constructed from scratch during garbage collection, there is no problem in discarding the old free space.

LWF is also expected to be faster by having more pointer incrementing memory allocation, since we can get larger free space areas, yet this would depend on the pattern of memory requests. On the other hand, LWF may result in more fragmentation and wastage of the discarded free spaces, which

might lead to larger heap sizes and more garbage collection cycles. All of these will be evaluated through experiments in the following Section 3.4.

3.4 Experimental results

Lazy fits are evaluated by generating traces of memory requests for a set of C programs and measuring the fragmentation and fit frequencies for both explicitly managed heaps and garbage collected heaps in previous work [12]. In this paper, we implemented lazy fits on a working Java virtual machine and evaluate whole Java system using non-trivial Java programs.

3.4.1 LWF implementation in the LaTTe Java virtual machine

A memory system using LWF was implemented on LaTTe, a freely available Java virtual machine with a just-in-time (JIT) compiler [16]. This subsection describes the implementation of lazy fits in LaTTe and outlines the LaTTe memory management system, which will be helpful in understanding the experimental results.

LaTTe manages a small object area and a large object area separately, and LWF is done only on the small object area which contains objects that are smaller than a kilobyte. One of the reasons for the separation is that sharing the same heap among large and small objects may result in high fragmentation, as the experimental results in [12] indicate. Large objects are allocated in separated area and best fit is used when allocating them.

LaTTe uses a partially mark and sweep garbage collector, in the sense that the runtime stack is scanned conservatively for pointers while all objects located in the heap are handled in a type accurate manner [17]. Pointers should be handled conservatively, since there is no accurate information for pointers in stacks. It is also possible to provide accurate type information to garbage

collector, but it requires more computation and memory space to maintain information at runtime and degrades overall performance. The separation of the small object area and the large object area also helps the garbage collector identify pointers more easily and efficiently, since we can make use of the fact that memory is separated when handling pointers.

LaTTe starts with an initial heap pool of 8 MB. Both the small object area and the large object area are allocated from this heap pool in units of 2 MB. If there is no memory available in the pool, LaTTe activates the garbage collection thread to reclaim unused memory.

After each garbage collection, LaTTe may decide to expand the heap depending on its capacity. The idea is that if the heap is too small, the garbage collection frequency would be unacceptably high. On the other hand, LaTTe does not expand the heap unnecessarily, since other applications will run out of memory in a multiprogramming environment.

LaTTe expands the heap only when the size of free memory is less than the size of live objects, meaning that the heap is less than half empty. Here, the free memory is estimated by the cumulative size of objects allocated between the previous garbage collection and the current garbage collection, instead of the heap size minus the size of live objects, which cannot be considered to be entirely free owing to fragmentation. So, LaTTe expands the heap only when the size of live objects exceeds the size of objects allocated, the expanded amount being the difference between these two quantities (rounded off to 2 MB). This appears to be a good compromise between the conflicting goals of keeping the size of the heap small and keeping the garbage collection frequency to a reasonable level.

Table 3.1 Benchmarks

Benchmarks	Description
_202_jess	A Java version of NASA's CLIPS expert shell system
_209_db	Data management software which performs multiple database functions on memory resident database
_213_javac	Java compiler from the JDK 1.0.2
_227_mtrt	Dual-threaded ray tracer that render the scene in the input file
_228_jack	A Java parser generator
EulerBench	Computational Fluid Dynamics
MonteCarloBench	Monte Carlo simulation
RayTracerBench	3D Ray Tracer
SearchBench	Alpha-beta pruned search

3.4.2 Experimental environment

We ran the experiments on a Sun Blade 1000 machine with a UltraSPARC-III microprocessor 750MHz with a 32 KB instruction cache and a 64 KB data cache. It also has an 8 MB second-level cache and a 1 GB memory.

Our benchmarks are composed of nine selected benchmark applications from the SPECjvm98 suites and section 3 of the Java Grande benchmark suites Version 2.0, which are listed in Table 3.1. The first five benchmarks of the table are selected from SPECjvm98 and remaining four benchmarks are chosen from Java Grande. We excluded those benchmarks that do not allocate enough small objects from the suites, such as _200_check, _201_compress and _222_mpegaudio in SPECjvm98, and MolDynBench in the Java Grande benchmarks. We used size A inputs for the Java Grande benchmarks during the experiments.

Table 3.2 Running Time Analysis

Benchmark	Total running time (seconds)			Allocation time (seconds)		
	LWF	LFF	FF	LWF	LFF	FF
_202_jess	12.359	15.352	212.641	1.676	4.520	198.466
_209_db	18.235	18.652	35.507	0.629	1.021	18.038
_213_javac	19.162	995.678	7855.287	1.779	972.019	7818.269
_227_mtrt	11.093	22.844	2846.460	1.326	11.996	2826.267
_228_jack	13.889	16.950	354.650	1.305	3.764	338.057
EulerBench	33.347	972.437	22600.980	1.198	937.126	22523.039
MonteCarloBench	107.530	118.061	132.451	0.106	10.404	24.163
RayTracerBench	21.378	21.882	74.493	1.059	1.471	54.213
SearchBench	19.740	20.506	106.550	1.383	2.178	88.089

3.4.3 Performance of LWF

We experimented with three different memory allocation policies: LWF, LFF and FF. The first fit algorithm used in LFF and FF uses segregated free lists segregated by a power of two distribution [18], with objects maintained in the FIFO order. This segregated free list is believed to reduce the allocation time compared with the traditional FF and may have less fragmentation [9].

LFF works in exactly the same way as LWF except that FF is used when the pointer-incrementing allocation fails. And the remainder of the previous free space is discarded as in LWF. Unlike LWF and LFF, FF always returns the remainder of the free space to the free lists.

By comparing LWF and LFF we can evaluate the impact of LWF's $O(1)$ access time for the free lists and how worst fit and first fit affect fragmentation in the context of lazy fits. By comparing LWF or LFF with FF, we can evaluate the impact of pointer-incrementing allocation of lazy fits.

For each benchmark, Table 3.2 shows the total running time and allocation time of each policy. The results indicate that LWF is always better than LFF, and LFF is always better than FF. In fact, there are several benchmarks

Table 3.3 Allocation Time Analysis

Benchmark	Allocation time (seconds)					
	Fit allocation			Other		
	LWF	LFF	FF	LWF	LFF	FF
_202_jess	0.164	2.974	195.117	1.512	1.546	3.349
_209_db	0.075	0.462	16.736	0.554	0.559	1.302
_213_javac	0.505	970.099	7814.567	1.274	1.920	3.702
_227_mtrt	0.153	10.805	2823.008	1.173	1.191	3.259
_228_jack	0.089	2.527	335.049	1.216	1.237	3.008
EulerBench	0.461	935.616	22518.786	1.526	1.510	4.253
MonteCarloBench	0.038	10.312	24.020	0.068	0.092	0.143
RayTracerBench	0.036	0.447	51.819	1.023	1.024	2.394
SearchBench	0.048	0.854	85.088	1.335	1.324	3.001

which show excessively high improvement when using LWF. When compared to LFF, _213_javac and EulerBench show much shorter running time with LWF. _202_jess, _213_javac, _227_mtrt and EulerBench also have been drastically improved when compared to FF.

In order to check whether the allocation policy really affects the running time, we measured the total memory allocation time for the small object area separately, which is shown in the second column of Table 3.2. The allocation time results are consistent with the running time results such that longer allocation time means longer running time.

The allocation time of each policy includes the time spent for fit allocation using worst fit or FF, which was also measured separately as shown in Table 3.3. The second column of the table shows time spent of fit allocation to find a new free chunk. The third column, i.e. other, are remaining time of total allocation other than fit allocation and it includes pointer incrementing allocation time.

Next, we analyze why LFF and FF have longer fit allocation time than LWF. There are two major differences between LWF and other LFF/FF, that affect the fit allocation time. The first is the frequency of fit allocation. Generally,

Table 3.4 Frequency (%) of small memory allocation via fit policy

Benchmarks	LWF	LFF
_202_jess	6.731	7.611
_209_db	0.691	0.661
_213_javac	11.815	22.372
_227_mtrt	3.688	2.815
_228_jack	2.760	2.665
EulerBench	21.017	9.408
FMonteCarloBench	10.696	27.892
FRayTracerBench	1.048	1.021
SearchBench	1.467	1.450
Geomean	3.867	4.101

LWF is expected to allocate more often via pointer increments than via fits, since worst fit would allow larger free spaces than LFF for pointer-incrementing allocation.

Table 3.4 shows the frequencies (%) of the fit allocation for LWF and LFF respectively. And the fit frequency for FF is obviously 100% which is not shown in the table. For _213_javac and MonteCarloBench, LFF has a much higher fit frequency than LWF. On the other hand, LWF has a much higher fit frequency than LFF in EulerBench. Therefore, contrary to our expectation we cannot see any definite relationship between the fit frequency and the fit policy.

Another major difference between LWF and LFF/FF is that the search time of the free lists for the fit allocation is $O(1)$ for LWF and $O(n)$ for LFF/FF. In order to check whether this difference really affects the fit allocation time, we measured the total number of 'link' operations, i.e. the operation to follow a single link in the free lists, for each policy.

Table 3.5 shows the number of link operations for LWF, LFF and FF respectively. It shows that LFF and FF execute many more link operations than LWF. Even for _227_mtrt and EulerBench where LFF has lower fit frequencies

Table 3.5 Comparision of 'link' operations

	Total number of operation (thousands)			Ratio	
Benchmarks	LWF	LFF	FF	LFF/LWF	FF/LWF
_202_jess	527.5	84,597.5	3,784,346.7	160.4	7173.5
_209_db	2.8	5,473.2	283,373,982.0	1955.4	101241.2
_213_javac	682.0	7,223,197.8	45,998,050.8	10590.8	67443.5
_227_mtrt	207.2	241,971.1	27,300,486.8	1167.7	131742.6
_228_jack	169.1	58,787.2	5,840,161.2	347.7	34545.7
EulerBench	1,361.4	10,469,150.4	194,860,155.4	7689.8	143128.1
MonteCarloBench	30.2	216,744.8	592,767.2	7184.6	19648.9
RayTracerBench	11.5	332.5	474,167.1	28.8	41103.3
SearchBench	10.1	589.8	1,937,796.8	58.2	191274.0

than LWF, LFF has a higher number of link operations than LWF. Since FF always uses fit allocations, it obviously executes more link operations than LFF. These results are consistent with the fit allocation time in Table 3.3, especially for those that have an excessively long fit allocation time. So, it is evident that the $O(n)$ search time is the dominant reason for the longer running time in LFF and FF. In fact, it can be seen that the linear time complexity of conventional fit allocation may cause an unacceptably high overhead, even with segregated implementations.

3.4.4 Fragmentation of LWF

Another important aspect of a memory allocator is fragmentation. It is generally believed that higher fragmentation requires larger heaps and causes more garbage collection, which may affect performance.

It is expected that LWF causes worse fragmentation than LFF and FF since worst fit is known to be poorer than FF in terms of fragmentation. Also, LWF and LFF have a disadvantage in fragmentation when compared to FF, since they discard the remainder of the previous free space area. Contrary to these expectations, our performance results in Table 3.2 indicate that the overall

Table 3.6 Average fragmentation ratio (%)

Benchmarks	LWF	LFF	FF
_202_jess	3.694	3.222	1.608
_209_db	0.240	0.264	0.252
_213_javac	14.203	5.400	5.427
_227_mtrt	2.092	2.231	2.713
_228_jack	2.454	2.504	1.097
EulerBench	0.395	4.513	8.326
MonteCarloBench	12.482	1.170	0.349
RayTracerBench	0.394	0.398	0.489
SearchBench	0.059	0.068	1.423
Geomean	1.249	1.155	1.332

performance of LWF is still better than LFF and FF, so these results need to be verified.

Tables 3.6 and 3.7 show the fragmentation ratio for the small object area for each policy. The fragmentation ratio was measured as follows. Whenever the memory allocator cannot satisfy a request for the small object area (so either 2 MB is allocated from the heap pool or garbage collection is invoked if the heap pool is empty), we measure the fragmentation ratio at that point.

Table 3.6 indicates that the average fragmentation ratio of LWF is not always higher than that of LFF and FF. In fact, we cannot see any definite correlation. For those benchmarks where the average fragmentation ratio of LWF is noticeably higher, such as _213_javac or MonteCarloBench, we found that the sequence of memory requests for the small object area occasionally includes requests for a relatively large object, e.g. > 100 bytes.

The problem with LWF is that larger free areas are consumed at the beginning of the allocation, such that by the time these large object requests arrive, their chance of being allocated in the current free space is lower, leading to the current free space being discarded, although it can still accommodate more

Table 3.7 Worst-case fragmentation ratio (%)

Benchmarks	LWF	LFF	FF
_202_jess	4.70	4.14	1.87
_209_db	0.38	0.38	0.27
_213_javac	47.21	7.64	8.27
_227_mtrt	21.85	21.49	21.35
_228_jack	6.03	5.47	1.92
EulerBench	2.17	6.56	12.18
MonteCarloBench	35.11	1.33	0.38
RayTracerBench	0.42	0.42	0.51
SearchBench	0.06	0.08	1.46

Table 3.8 Garbage collection data and size of small object area

	Garbage collection						Size of small area		
	Time (sec)			Count			(bytes)		
Benchmarks	LWF	LFF	FF	LWF	LFF	FF	LWF	LSFF	SFF
_202_jess	1.041	1.028	1.122	67	66	65	6,291,456	6,291,456	6,291,456
_209_db	0.865	0.855	0.856	8	8	8	20,971,520	20,971,520	20,971,520
_213_javac	4.042	3.910	3.901	16	16	17	44,040,192	35,651,584	37,748,736
_227_mtrt	1.817	1.800	1.853	18	18	18	18,874,368	18,874,368	18,874,368
_228_jack	0.458	0.452	0.468	42	42	42	6,291,456	6,291,456	6,291,456
EulerBench	3.074	2.749	3.176	43	38	41	14,680,064	16,777,216	16,777,216
MonteCarloBench	1.159	1.104	1.079	0	0	0	6,291,456	4,194,304	4,194,304
RayTracerBench	0.092	0.091	0.088	35	35	35	6,291,456	6,291,456	6,291,456
SearchBench	0.415	0.426	0.441	202	202	205	2,097,152	2,097,152	2,097,152
Geomean	0.906	0.884	0.911						

small objects.

On the other hand, LFF would have a relatively better chance of allocating the large object in the current free space. This would make LWF suffer more from fragmentation than LFF. Such cases may result in very high fragmentation for LWF as shown in Table 3.7 where the worst-case fragmentation ratio is measured.

In order to check the impact of fragmentation, we measured the garbage collection frequency, garbage collection time and the total size of the small object area, as shown in Table 3.8. The table shows that there is little difference in garbage collection time and frequency of garbage collection among the three

policies, which would explain why fragmentation did not have a major effect on performance.

As to the size of the small object area, LWF uses larger areas than LFF for _213_javac and MonteCarloBench where LWF suffers more from fragmentation, whereas LFF uses larger areas than LWF for EulerBench where LFF suffers more from fragmentation. However, there is no tangible impact on garbage collection time or frequency depending on allocation methods, as discussed.

3.5 Summary

We propose the use of lazy worst fit for memory allocation in Java, which exploits pointer-incrementing memory allocation with free lists. LWF avoids the linear time complexity of managing the free lists that may cause an unacceptably high memory allocation overhead, and it does not suffer much from fragmentation. One interesting question is whether these benefits may even allow a non-moving garbage collector to compete with compacting collectors, while avoiding their drawbacks. For example, copying collection has some problems such as half-availability of the heap space, exponential performance degradation as the object residency[3] increases [8] or poor locality [11]. Mark-and-compact collection is also known to be expensive to implement since compaction requires more than just copying objects or updating pointers [8]. It is left as a future work to evaluate non-moving garbage collectors with LWFs, compared with compacting garbage collectors.

[3]The ratio of live objects to garbage objects at any given time.

Chapter 4

Biased Allocator

4.1 Introduction

Virtual machine adopts automatic memory management to manage the heap. Automatic memory management reclaims objects which are not used anymore automatically from the heap, although object allocation is requested explicitly by a program. Garbage collection is a famous approach to find unnecessary objects, i.e. dead object, and reclaim them [8]. Allocation strategies and garbage collection should be considerate each other, since garbage collector is responsible for securing and managing free space which is used by allocator later to allocate objects. We can consider garbage collector a producer of free space and then allocator can be a consumer of free space. Therefore some garbage collectors enforce allocation methods considering fragmentation, performance and throughput. In vice versa, some allocation strategies are more efficient with specific garbage collectors. Various garbage collectors have been proposed by many researchers [19, 8, 20, 21, 22]. In detail, there have been different approaches

to find dead object and also there are various ways to secure free space. One of simplest way to find unnecessary objects is traversing pointers recursively from always live objects, which are called roots, to find reachable objects which are live and necessary. Another approach maintains counters for incoming references to each object at runtime to determine a liveness of object [23]. There are also several ways to secure free space after identifying dead objects. A simplest way maintains a list of free space by reclaiming dead objects. Another approach secures free space by moving live objects to different area, as a result previous area contains only dead objects and whole previous area can be considered to be free space. There are so many garbage collectors depending on how they identify dead objects and how they secure free space. Among them, a generational garbage collector is famous and widely adopted in virtual machines, e.g. Java Virtual Machine from Oracle [24].

A generational garbage collector manages the heap by splitting whole heap into several generations from young to older. With a generational garbage collector, a new object is always allocated from a nursery area which is one of generations and considered to contain young objects. Then later if a nursery is overpopulated and there is no room for new objects, a generational garbage collector secures free space from a nursery by moving live objects to older generations. We call this object copying a promotion, since an object is promoted to old generations. Such garbage collection on a nursery is called minor garbage collection. Later we have to reclaim all dead object in young and old generations too when old generations are also overpopulated. We call it a major garbage collection or a full garbage collection. Figure 4.1 depicts how a simple two generational garbage collector works. Due to various advantages, a generational GC is adopted in many virtual machines for various environments. First, a garbage collection can be completed in a short time when minor GC

is requested instead of full GC, because minor GC performs only on a nursery which is relatively smaller than whole heap. Although number of garbage collections increases relatively, each pause time caused by garbage collection is reduced and responsiveness of virtual machine is improved when compared to a garbage collector with only full GC. Furthermore secured free space from a nursery is continuous and fragmentation free since whole young generation is empty after all live objects are promoted. There are many variations of generational garbage collector depending on number of generations, size of young and old generations and etc. [19, 25, 21] However a generational garbage collection has unavoidable runtime overhead and it shows undesired behaviors in some cases.

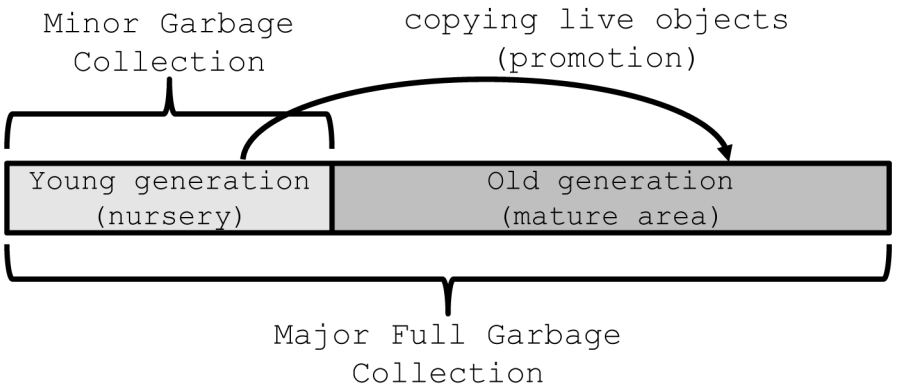

Figure 4.1 A generational garbage collector with two generations.

A generational garbage collector has to promote live objects to older generations to clear up a nursery. Each promotion contains not only copying an object but also updating pointers which refer to the object just moved to a new location. A generational garbage collector is beneficial when only few objects are live and most of objects in young generation are dead. However when many of objects in a nursery are live and is going to be promoted, the overhead

of promotion increases to hide advantages of a generational GC. In worst case when every object in a nursery is live, we have to promote all objects and minor GC does not reclaim dead object at all with overhead of minor GC and promotions. We suggest that such overhead can be avoided if we place promoted objects to old generations instead of young generation when those objects are being allocated at first. We are going to segregate objects in various ways to reduce number of objects allocated to a nursery. Rest of the paper is composed as follows. In the next Section 4.2, we address the problem in detail and propose an approach to exploit biased allocators to improve a generational garbage collector. Then we propose a way to invoke biased allocator and describe three analyses to identify objects to be allocated with biased allocators in Section 4.3. We describe how to combine proposed analyses and how we implemented proposed approaches in real environment in Section 4.4. In Section 4.5, proposed approaches are evaluated on a real embedded device. Section 4.6 summarizes the paper and discusses future works.

4.2 Motivation

As we discussed in the previous section, a generational garbage collector itself suffer from inherent overhead of promotion. As a result pause time of each garbage collection can be increased to compensate advantages of a generational GC. There have been many researches to improve a generational GC [8, 21, 22] , but most of them require modifying a garbage collector itself and a garbage collector is getting more complicated which is hard to predict the effect modification in various situation.

We propose an approach to exploit an allocation instead of a generational GC to overcome the undesired overhead of a generational GC. We already address that such an undesirable behavior of a generational GC is due to pro-

motion of many objects in a nursery. In other words, such objects live long to the time when minor GC is requested to reclaim dead objects. We are going to avoid the situation by simply locating such objects in old generations instead of a nursery when objects are allocated. Simply we can allocate all objects in old generations, but then it is not a generational GC anymore and may suffer from long pause time of full GC instead. Therefore we have to choose a set of objects and allocate them to old generation using biased allocators. In the following section, we propose a way to make use of biased allocators and describe how to identify objects to be biased in detail.

4.3 Biased allocator

With biased allocators, an object can be allocated to heap in different ways depending on various properties to improve the performance of heap management with a generational garbage collector. On the other hand, traditional virtual machine with a generational GC allocates an object to a nursery area of heap with a single same allocator. We propose that we affect the performance of heap management in a beneficial way by reducing copying overhead of generational garbage collection if we allocate an object to other than a nursery carefully with different object allocators. In this section, we will discuss when to choose an allocator and propose a way to make a decision with less runtime overhead. Then we will describe three analyses to select an allocator.

4.3.1 When to choose an allocator

We can choose an allocator every time when an object is being allocated to the heap. It would be best if we can perform fine-grain analysis for each object and decide allocation area for each object. However it is not easy to predict lifetime of each object precisely and there will be extra overhead if we choose

an allocator every time an object is being allocated. Usually an object allocation occurs very frequently and an additional computation could harm overall performance. Therefore it would beneficial to runtime performance if we can choose an allocator without extra overhead of an allocation itself.

A *new* bytecode in Java Virtual Machine always knows a type of an object to be allocated [1] and the *new* bytecode allocate objects of same type. Therefore we are going to exploit the property that each *new* bytecode always allocates isomorphic type of objects at runtime.

Also we try to reduce the overhead of decision making by making a decision once and use the same decision later. To achieve these, we choose an allocator when bytecode are analyzed and being translated into native machine code to improve overall performance. In other words, biased allocator can be applied to any kind of translators including just-in-time compiler (JITC), ahead-of-time compiler (AOTC) and install-time compiler (ITC). A Just-in-time compiler which translates bytecode into machine code at runtime is a famous acceleration technique [26, 16, 27, 28] . Ahead-of-time compiler [29, 30, 31] and install-time compiler [32] analyze and translate bytecode into native machine code before it is being executed.

A biased allocator is chosen when a *new* bytecode is being translated into machine code depending on the type of object to be allocated. Then the *new* bytecode is translated into a machine code which allocates an object with the selected allocator. In this way, we make a decision once and an allocation is done without additional overhead other than that the allocator allocates an object in a different way.

4.3.2 How to choose an allocator

Even though that a specific *new* bytecode accepts an isomorphic type, it is not easy to exploit the information to select an allocator wisely. We need whole type analysis on a Java program to make a correct decision and it is not eligible for a JITC or AOTC, because whole type analysis including class hierarchy analysis [33, 34] is not a simple problem and it takes much time. Therefore we consider three information as well as simple type information, i.e. class information which is known directly from the *new* bytecode itself.

First we identify a location where local-scoped objects are allocated. Also an allocation site within a loop is identified and being chosen to use a biased allocator. Finally we analyze the use of an allocated object which is assigned to static fields and identify locations where the object is allocated. Since an object can be allocated from multiple locations depending on control flow, we exploit traditional iterative data follow analysis. Of course, type information is always considered together with three properties.

Local-scoped objects

An object is known to be locally scoped if an object is live only within a specific scope. A scope can be anything such as a basic block, a super block, a trace, a method or even a program. There have been many researches to identify locally scoped objects and escape analysis is one of famous technique to identify locally scoped objects. Escape analysis has been used in Java to make use of stack allocation [35, 36] to relieve memory pressure on the heap and adopted in various JVM such as Java Standard Edition 6 [37]. We use an escape analysis to identify an allocation site where objects being allocated are locally scoped. We expect that such an allocation site can make use of traditional allocator or even stack allocator which uses a stack instead of the heap, because locally

scope objects are only live within a specific scope and liveness is limited to the scope which can be considered being relatively shorter than other objects which escape the scope. As a result, we don't have to consider such allocation sites for being a candidate for biased allocation to improve the performance of heap management with garbage collection.

Objects allocated inside loops

Loops have been a famous target for an optimization, because many programs spends most of the time in loops and small improvement in a loop can be result in large runtime improvement of the performance due to its repetition. We also look into loops, because an allocation in loops will continue allocate same type of objects until loop stops and quite large amount of objects are allocated inside of the loops.

We expect that objects allocate inside loops are relatively short lived compared to objects allocated outside of loops, because loops usually perform same computation repetitively and many objects allocated within loops are for temporary use. We decide objects allocated inside loop to be possibly short-lived at first. However we find that some objects, which are allocated in a loop but have relatively small size, are long-lived. Therefore allocation sites within loops are chosen when smaller objects are allocated. We can easily compute the size of objects, because the type of object being allocated is identified directly from *new* bytecode as we described before. We don't have to worry about leaving large objects behind in young area, because promotion overhead is more dominated by number of objects being promoted than size of objects as we discussed in previous sections.

Objects assigned to static fields

There are two types of objects in Java, i.e. an instance object and a class object. An instance object is an instance of a specific class which are usually allocated with *new* keyword of Java language and object we talked before in this paper are all instance objects. A class object is a unique object of a specific class and they are usually created implicitly by Java virtual machine when the class is being resolved. A static field is a field not related to an instance object but class object itself. Since a static field looks like a global variable, researches have shown that an object assigned to a static field tend to be immortal, i.e. never dead till the program ends [38].

We decide to make use of this property and use biased allocators for such allocation sites where any object allocated can be assigned to static fields. We make use of traditional analysis of reaching definition to identify allocation sites on the compilation unit. Candidate allocations sites can be one or more and even we can't find a site, because we perform analyses only within the compilation unit.

Of course, some candidate allocation sites can be duplicated with the previous analysis, i.e. allocation sites within loop. We will discuss how we arrange three analyses we discussed here to make a decision for biased allocation in the following section.

4.4 Analyses and implementation

Each allocation site can have three properties, i.e. local, loop and static. Local means this allocation site allocates objects which are live only within the scope. Loop means this allocation site is located within loops and size of allocation is larger than threshold. Static means this allocation site allocate objects which

can be possibly assigned to static fields. Only allocation sites which is neither local nor loop are selected for biased allocation. Then we find allocation sites with static property and add them to candidates and we are going to describe how we make use of three analyses.

At first, we assume that all allocation sites are candidate for biased allocation. We find locally scoped object with escape analysis. After we identify allocation sites which only allocate locally scoped objects, we remove those sites from candidates. We do not discard the list of allocation sites that are local and keep the list for later use.

We continue to identify allocation sites within loop and this analysis can be done with other traditional loop optimizations as well. However this analysis should be done after any control flow changes or code motions are made to loop, because the location of an allocation site can be changed with those optimizations and even allocation sites can be eliminated after optimizations. Furthermore we do not analyze and skip allocation sites which are already identified to allocate only locally scoped objects from previous escape analysis. Then we reduce candidate allocation sites with results of loop analysis. We find out that some allocation sites within loop allocate only locally scoped objects and it is obvious that these objects have relatively shorter lifetime than other objects which escape the same scope.

Finally we look into every assignment of an object to static fields and try to identify one or more allocation sites where the object was allocated. This analysis should be done just before the code generation, because any control and data flow changes can affect the result of this analysis. After we identify allocation sites, we add those allocation sites to candidates for biased allocation. In short, we can formulate above sequences as in Figure 4.2. We should keep the order of local, loop and static analysis, because there can be an allocation site

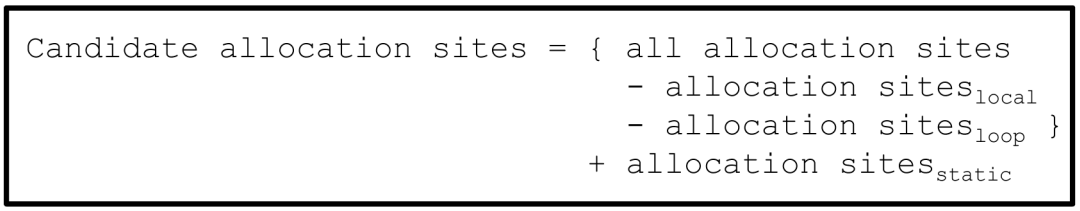

Figure 4.2 Candidate selection with three analyses

which reside in loop and allocate large objects, but allocates objects which can be assigned to static fields. Of course there is no allocation site which allocates locally scoped object and allocates objects assigned to static fields, because an object is not locally scoped if there is any assignments of an object to static fields.

We implemented these analyses on Oracle's phoneME Advanced MR2 version. This phoneME advanced MR2 is Java virtual machine for embedded devices and can run Java applications via interpreter and just-in-time compiler (JITC). Ahead-of-time compiler (AOTC) [29, 30, 31] is also available for translating Java bytecode to native machine code with optimizations where proposed approaches had been inserted.[1] Our analyses were also done within a method scope, since a translation unit of the AOTC is a method. Figure 4.3 depicts a implementation of biased allocator in virtual machine with AOTC. Analyses are implemented in AOTC and we generate hints at static time as shown in the figure. A biased allocator itself is available in virtual macine and allocates objects regarding hints at runtime.

[1] As we mentioned before, analyses can be implemented in any translator which translates code, such as JITC, AOTC and ITC.

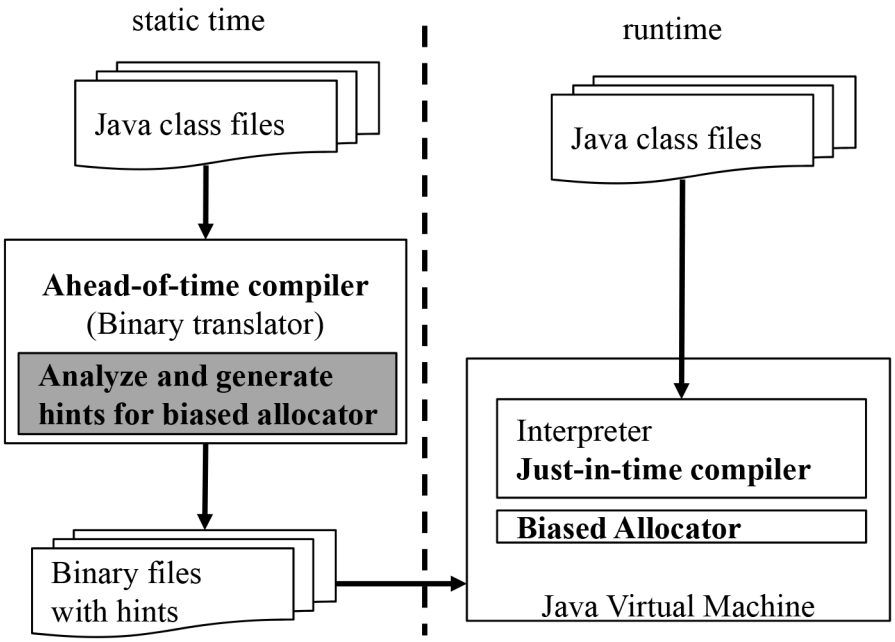

Figure 4.3 Implementation of biased allocator

4.5 Evaluation

We evaluate our proposed analyses on phoneME Advanced MR2 [39] with digital TV (DTV)[5] set-top box which includes MIPS based core with 128MB main memory. This software platform in digital TV supports advanced common application platform (ACAP) middleware and is running on the Linux with kernel 2.6.12.

We make use of AOTC to perform proposed optimization and observed the effect of biased allocation without runtime overhead of analyses. Java applications have been translated by AOTC before running and stored in set-top box for evaluation. We use six micro benchmarks from specjvm98 [40] to evaluate our approaches. We choose a generic generational garbage collector with two generations in phoneME Advanced MR2 to reclaim objects while running

specjvm98. Since total pause time due to garbage collections is relatively small compared to total running time, we compared total pause time separately instead of total running time and measured the amount of promotions occurred in generation garbage collections.

4.5.1 Total pause time of garbage collections

We measured total pause time of garbage collections before and after applying proposed approaches and compared them in Figure 4.4. About up to 12.2% of total pause time caused by garbage collection has been reduced and about 4.1% of pause time is removed in average. Figure 4.5 depicts the size of biased objects compared to total size of objects allocated. We identify lots of objects from _209_db where pause time has been reduced most. However even we biased more than 10% of objects from _228_jack, total pause time is not reduced much as we expected compared to other programs and we can't find direct correlations between the size of biased objects and total pause time. After careful examination, we find out that total pause time of generational garbage collector is affected by various factors and it is very hard to predict. For example, size and number of objects allocated in nursery area affect pause time. When promotion occurs, more factors affect pause time of generational garbage collection, because a promotion includes copying an object and updating pointers to copied object. Even worse promotions may incur a full major garbage collection when there is no sufficient space in a mature area.

On the other hand, our approaches may consume a mature area more aggressively due to false detection. Three analyses we proposed are all based on static analysis without runtime information. Therefore we can't predict exact life time of objects and availability of the heap is not concerned at all. As a result proposed approach may induce side effects in unexpected ways due to

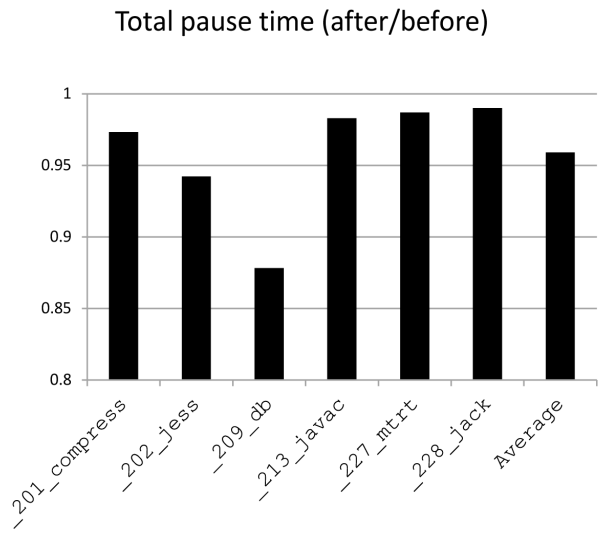

Figure 4.4 Ratio of total pause time after applying biased allocation compared to non-biased allocation

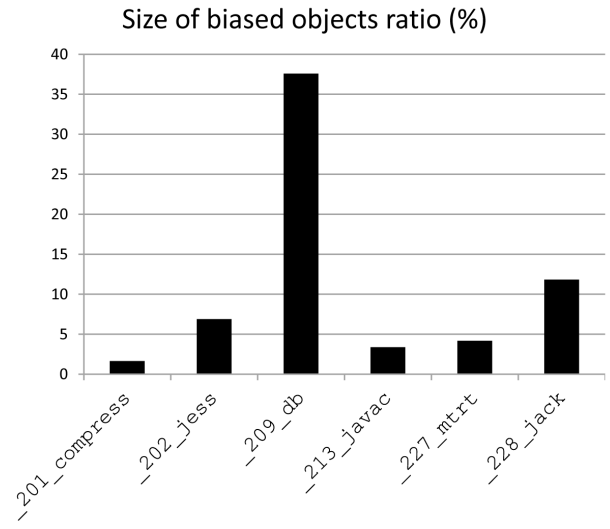

Figure 4.5 Ratio of biased objects size compared to total objects

exploiting a mature area much more than a nursery area. However it is not easy to calculate lifetime of objects exactly and our research is a start point to exploit different allocation based on analyses. We will discuss these matters in the last section again with future works.

4.5.2 Effect of each analysis

We also evaluated the effect of each proposed analysis in Figure 4.6. When we choose objects with an escape analysis, we can't reduce total pause time of garbage collections effectively. We found that total pause time has been reduced much after analyzing loops. Even though we decide to bias objects which are allocated to static fields towards old generation, Figure 4.6 shows that there is only a little improvement with this optimization. However it is expected, because objects assigned to static fields are rarely overwritten and few allocations are related to static fields. Of course, there are some allocation sites where few objects are assigned to static fields and other objects are discarded soon. A proposed analysis may decide those allocation sites to be candidate for biased allocation but those are not desirable choices, because we want to allocate objects that live long. Therefore those candidates can be false-positive. Nevertheless it reduces pause time slightly in average.

4.5.3 Pause time of each garbage collection

We also examine each garbage collection to evaluate biased allocation. Since behavior of garbage collections is changed after applying proposed optimizations, it is not reasonable to compare each garbage collections one-to-one. For example, garbage collections are invoked at different phase of a program and each garbage collection may reclaim different objects after applying optimizations. Therefore we choose the first five garbage collections of _209_db where

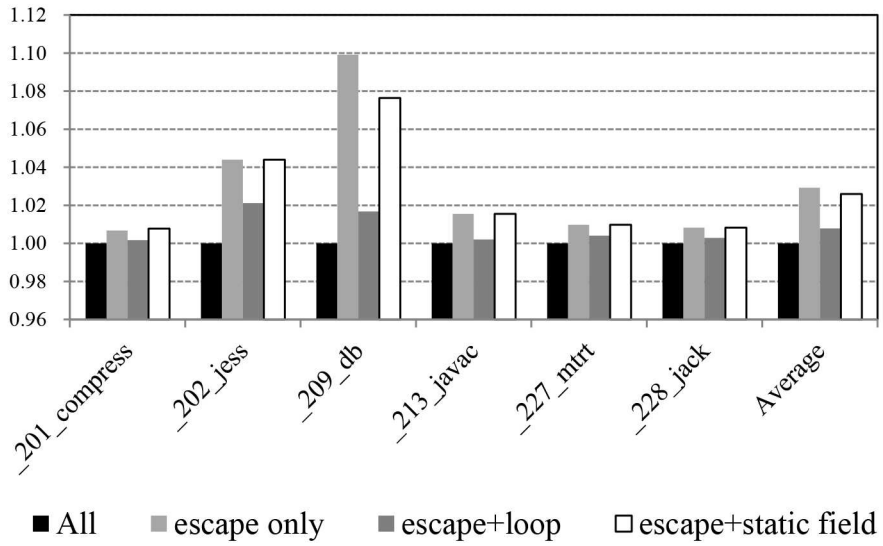

Figure 4.6 Ratio of total pause time of garbage collections compared to all analyses enabled. Therefore All is always one.

promotion occurs and compared number of promotions to original garbage collections as in Figure 4.7. We choose these five garbage collections, because they behaves different but not totally different. Even though it is not fair to compare them one-to-one, it is easily noticed that total number of promotion occurred in the first five garbage collections have been reduced about 25%. All five garbage collections have less number of promotions than original garbage collections. This is expected results, since biased allocator try to allocate objects in a mature area other than in a nursery where some objects should be promoted later. The first garbage collection has been invoked more lately than before, because a nursery is less populated after applying biased allocation.

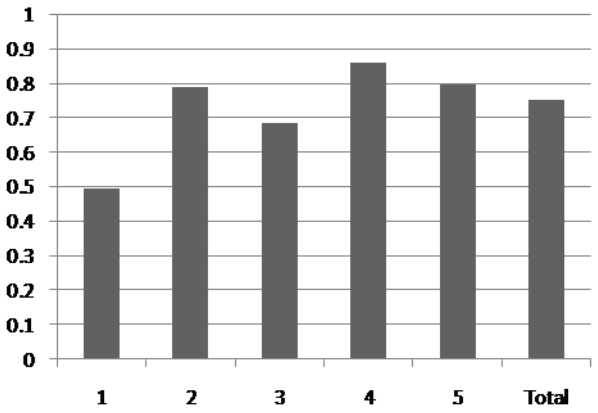

Figure 4.7 Ratio of promotions occurred for the first five garbage collections with biased allocator compared to original in _209_db.

4.6 Summary

We proposed a way that different allocators can cooperate with garbage collectors which have a critical role in memory management of virtual machine. For a generational garbage collector, we proposed approaches which make use of existing analysis techniques to relieve the side effect of generational garbage collector. Allocation sites have been chosen and biased with three analyses and each biased allocation site uses new biased allocators instead of original allocator. We implement a proposed approach in real embedded Java device and evaluate the effectiveness. Total pause time of garbage collections has been reduced and promotion overhead of generational garbage collection has been also reduced in overall.

However we can't guarantee correctness of biased allocation with analyses discussed in this paper. Furthermore analyses discussed in this paper are done at static-time and does not make use of any runtime information. We expect that analyses can be more accurate if runtime information is provided. Each

allocation site use same allocator after decision had made. We expect allocators can be chosen adaptively or allocator itself can evolve for further improvement. Also we use only single biased allocator to bias objects but more allocators can be used for various garbage collectors. We are also expecting that there are opportunities for biased allocation to improve other garbage collectors as well as generational garbage collector.

Chapter 5

Ahead-of-time Heap Management

5.1 Introduction

Automatic memory management improves productivity of programming and secures the stability of a program, since it frees the programmer from various memory management concerns including memory leakage problem. A variety of virtual machines adopts automatic memory management techniques such as garbage collection. For example Java virtual machine [1], JavaScriptCore in webkit [41, 42] and Dalvik virtual machine in Android [2] make use of garbage collector to reclaim dead objects automatically.

Garbage collection (GC) which automatically finds and reclaims dead objects, i.e. objects which are not used anymore, is a famous automatic memory management technique. [8, 23, 21, 22, 20] With garbage collection, programmers don't have to concern tedious implementation of memory management when writing programs. Numerous techniques about garbage collection have been proposed regarding diverse software environments and purposes. Reclaim-

ing dead objects at runtime incurs inevitable runtime overhead and many approaches have been proposed to reduce the overhead, because finding dead object requires a certain amount of computation to make a decision.

We can totally avoid those runtime overhead if infinite memory resources are available and no object is needed to be reclaimed. However in real world, memory resource is limited by hardware and multiple programs share the memory. Even worse, programs are competing for the memory in multitasking environments. A program allocates objects on a heap which is also allocated on total memory for private use of the program. When certain conditions are met, garbage collection starts to reclaim dead objects and secures free space in the heap for future object allocations. Usually garbage collection reclaims dead object when there is no sufficient space in the heap to satisfy a new object allocation request. However, garbage collection is not always successful to secure free space due to various reasons. In such cases, virtual machine tries to complete an object allocation by expanding the heap itself to make a room for new objects, i.e. allocating more heap space on the memory.

As we discussed before, it is obvious that we can avoid garbage collection overhead, if virtual machine chooses to expand the heap instead of performing garbage collection. But it may result in the very large heap and it is only feasible with infinite memory as discussed before. Therefore most virtual machine tries to secure free space in the heap by reclaiming dead object with garbage collection before expanding the heap when there is no sufficient free space in the heap for a new object allocation request.

While it is reasonable to choose garbage collection before expanding the heap to avoid excessive memory use, it is also true that expanding the heap can hide garbage collection overhead. Consequently virtual machine should make a choice carefully between garbage collection and heap expansion considering

overall performance and heap use. A choice of garbage collection and heap expansion does not guarantee the same results and the result is greatly affected by memory behavior of an application. For example, if an application allocate objects which are always live, garbage collection is almost useless because it cannot reclaim objects at all. In such case, expanding heap is better choice than garbage collection considering the performance and heap use, because the heap use is always same regardless of the choice but the performance differs with the choice. A variety of approaches has been introduced to compromise the performance and heap use by speculating the memory behavior of applications. [43, 44, 45, 46, 47] Previous researches shows that it is very hard to predict behavior of applications exactly and there are some ways to speculate the behavior indirectly and we are also inspired by those approaches.

We propose a heuristic for choosing heap expansion carefully to improve overall performance and to provide better user experience with runtime information observed from real applications. Runtime temporal information is taken into consideration as well as runtime spatial information when making decision between garbage collection and heap expansion. Then we try to expand heap ahead-of-time to fully avoid garbage collection overhead with temporal and spatial information.

In the following Section 5.2, we describe our motivation based on observations of real applications. We explain a existing heuristic for garbage collection and heap expansion in Android system in Section 5.3. Then we propose our heuristics based on spatial and temporal information in Section 5.4. We evaluate proposed heuristic in real device in Section 5.5 and summarize this chapter in Section 5.6.

5.2 Motivation

Android employs mark-and-sweep based garbage collector with a concurrent GC approach which is invoked periodically when certain conditions are met to secure sufficient free memory space in time. However it seems that many GC invocations failed to secure sufficient memory and even worse too many GC invocations are requested in a short time interval. Such behaviors of GC result in bad user experiences.

Figure 5.1 depicts GC distribution based on the secured free memory amount in Android. We look into six applications running on Galaxy Nexus to observe garbage collection behavior. Black indicates allocation GC which is invoked due to allocation failure and grey indicates concurrent GC which is invoked periodically.

We observed that more than 50% of GC invocations secure only small amount of free memory, i.e. less than 10 kilobytes, in a `Gallery` application and most of those GC are requested due to allocation failure, i.e. black. For `Camera` and `Maps`, more than 20% of GC invocations reclaim less than 10KB dead objects. Allocation GC tends to secure less amount of free memory than concurrent GC in many applications. Therefore we can infer that allocation GC is not successful to collect lots of dead objects and secure large free memory. In the such situation, Android is forced to expand the heap after garbage collection to secure additional free space when garbage collection secure relatively small amount of free space by reclaiming dead objects.

We also observed that in some applications many GC are requested in a short time. Figure 5.2 shows distribution of time interval depending on the number of GC invocation in the interval where each time interval is one second. Among 30 time intervals, 14 time intervals do not suffer GC overhead at all,

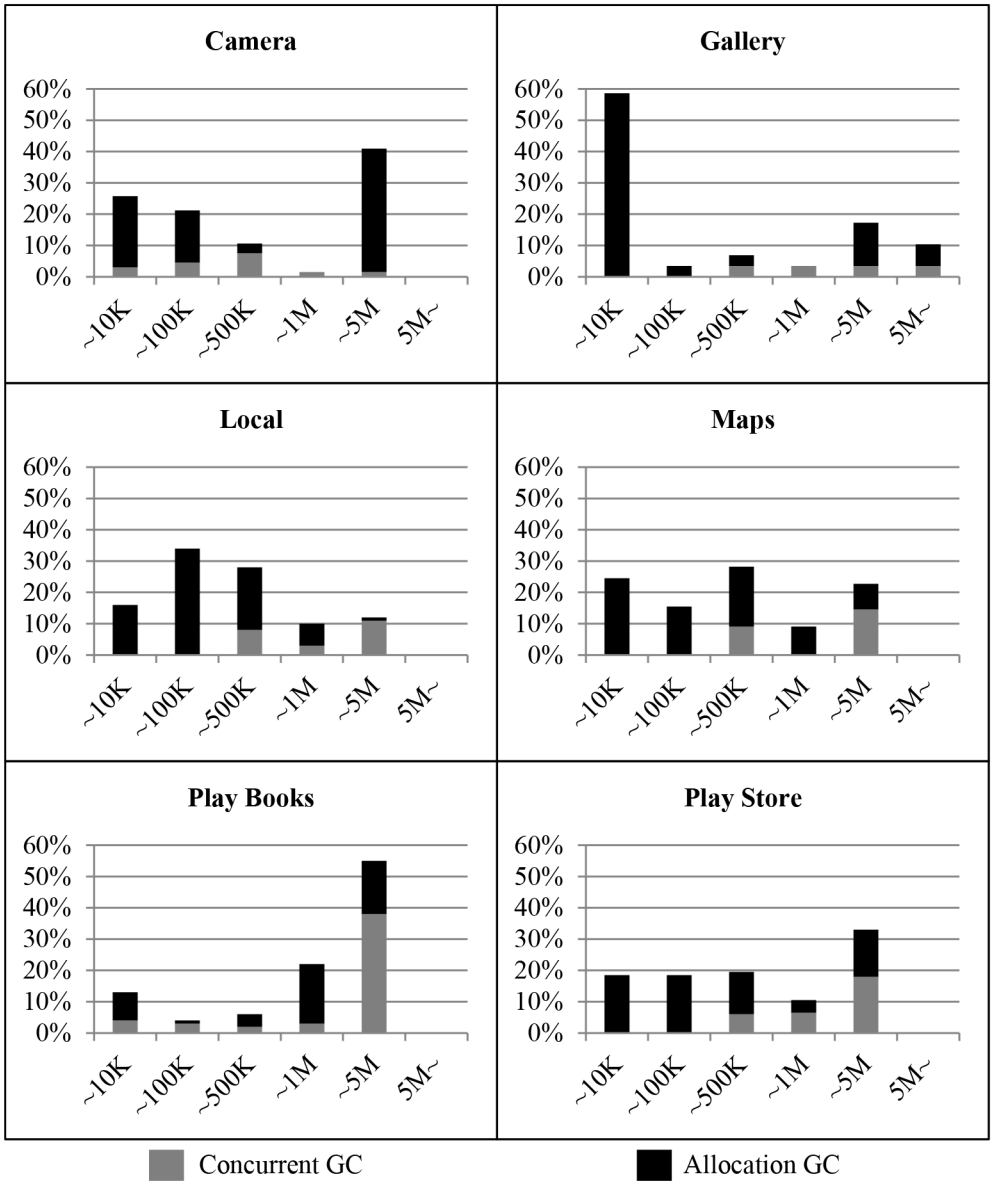

Figure 5.1 GC distribution by secured free memory amount

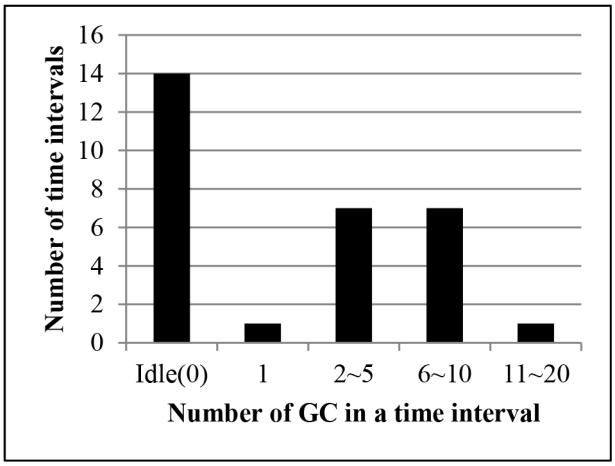

Figure 5.2 Number of time intervals depending on the number of GC in `Maps` application

whereas there is a interval where more than 10 garbage collection are requested in a second. Even with concurrent GC which is invoked periodically to secure free space before allocation GC is invoked, we can conclude from the observation that garbage collection is invoked excessively in a relatively short time interval.

From the first observation, we found that many garbage collections failed to secure sufficient free space in some applications where the heap is forced to be expanded as a consequence. From the latter observation, we observed that distribution of garbage collection is biased and there is a situation where excessively many garbage collections are invoked in a short time interval, resulting in bad user experiences. We are going to propose heuristics to make a choice between heap expansion and garbage collection to avoid such undesirable situation.

5.3 Android

Android adopts Dalvik virtual machine as core execution engine and Dalvik allocates memory from operating system as a heap and manages this heap. As we describe in previous sections, objects are allocated on the heap when an application requests new objects to be allocated. Dalvik employs garbage collection to reclaim dead objects automatically at runtime and secures free spaces for future object allocations. Consequently application programmers can rely on garbage collection and don't have to worry about memory management. Dalvik may expand the heap by allocating a new memory space from operating system when there is no sufficient free space after reclaiming dead objects. In the following subsections, we are going to describe heuristics used in Dalvik to reclaim dead object, i.e. garbage collection heuristic, and to expand the heap after the garbage collection, i.e. heap expansion heuristic.

5.3.1 Garbage Collection

Garbage collection in Dalvik adopts a mark-and-sweep strategy to find and reclaim dead objects. Mark-and-sweep garbage collection has two phases including a mark phase and a sweep phase. The first mark phase traverses all reachable objects recursively from objects in root set which is a predefined by the virtual machine, Dalvik itself in this case. All reachable objects are marked in the mark phase and we can consider all unmarked objects dead because those objects cannot be used from anywhere. We reclaim all unmarked objects and secure new free space by sweeping all unmarked objects in the sweep phase. [8]

Dalvik invokes the garbage collection in two ways. First, there is a dedicated thread for the garbage collection and this thread wakes periodically to reclaim dead objects when certain conditions are met, i.e. concurrent garbage collection. Secondly, a garbage collection starts when there is no sufficient free space

to satisfy the new object allocation request, i.e. allocation garbage collection. In the concurrent GC, a mark-phase of GC and application threads runs concurrently for a time being. Then a GC thread waits all application threads to be stopped and continues to complete remaining mark phases and the whole sweep phase. In the allocation GC, the garbage collection waits all other threads to be stopped and then continues to mark-phase and sweep-phase, often known as a stop-the-world approach.

5.3.2 Heap expansion heuristic

Dalvik decides to expand the heap in two conditions after the garbage collection. If android fails to secure free space which is less than preferred ratio of the total heap size, android chooses to expand the heap. Dalvik also expands the heap when an object allocation request failed to find room for allocation after the garbage collection which is invoked by the allocation request, because the garbage collection already reclaimed all known dead objects but still there is no free space suitable for a new object.

Even when garbage collection is successful to reclaim sufficient dead objects and secures free space larger than the size of allocation request, allocation request may not be satisfied due to fragmentation problem. Fragmentation problem occurs when there is sufficient free space in total but no continuous free space is available to satisfies the allocation request, because free space is fragmented in small pieces. [18] The fragmentation problem can be avoided with more complicated garbage collection, such as mark-and-compact GC and generational GC [8], but it is unavoidable with mark-and-sweep garbage collector used in Android. Those complicated garbage collection requires more computation than mark-and-sweep and may incur other performance problems. There have been approaches that various garbage collection is adaptively chosen [44]

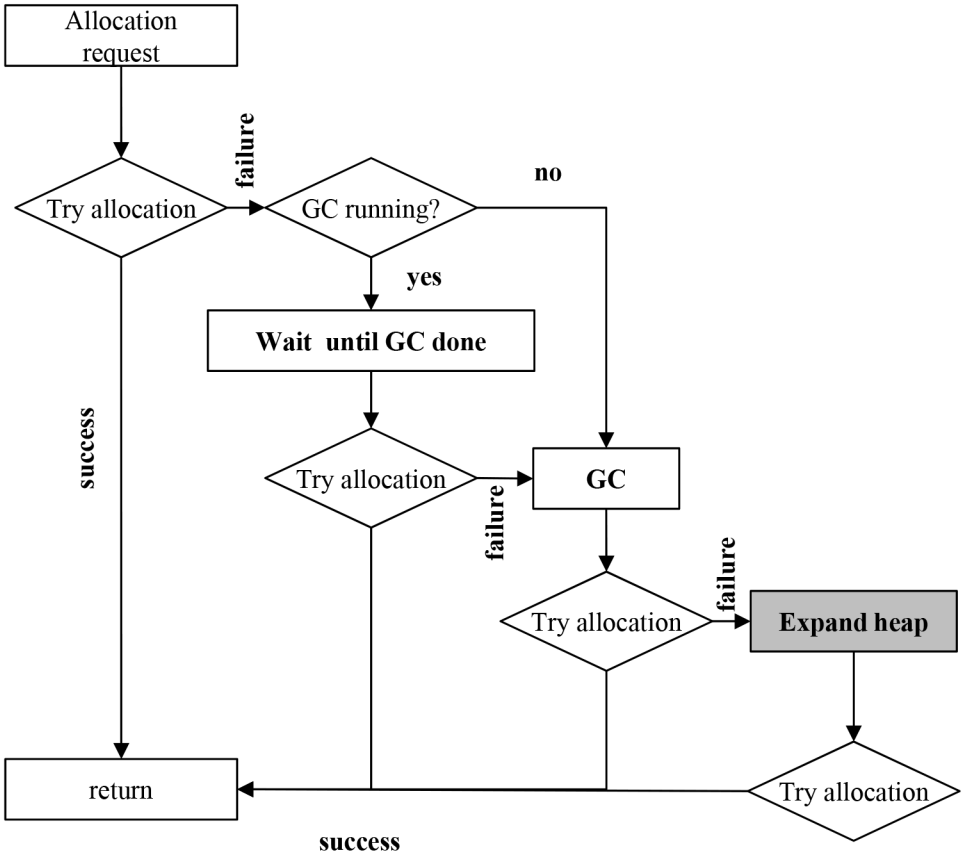

Figure 5.3 Flow of heap management in Android 4.1.2

but the topic is beyond the scope here. In this chapter, we will discuss how to make a choice between garbage collection and heap expansion when garbage collection technique is fixed.

Figure 5.3 depicts the flow of heap management in Android 4.1.2. Allocation trial, garbage collection and heap expansion caused by an allocation request is shown in the figure. We found that Android chooses to expand heap after three allocation trials and two garbage collections in the worst cases. We expect that by expanding heap wisely beforehand we can satisfy the allocation request with

less allocation trials and less garbage collection, i.e. avoiding the worst case scenario. We can also avoid future garbage collections, if we expand the heap more aggressively when expanding the heap in advance. In following sections, we are going to propose an ahead-of-time heap expansion heuristic to achieve less runtime overhead with less garbage collections by exploiting heap expansion aggressively in advance.

5.4 Ahead-of-time heap expansion

We have to consider several issues when expanding the heap. We can avoid every garbage collections except concurrent garbage collection, if we always expand the heap without limitation to satisfy object allocation requests. However size of the heap will grow too large for memory resource available in a device where multiple applications and services run altogether. As a result memory utilization will not be effective in such multi-programming environment, if one application solely consumes large amount of memory. Furthermore we can't avoid concurrent garbage collection in Android and it may incur unaffordable runtime overhead, because garbage collection, especially mark-and-sweep based one, has to traverse all objects to sweep unmarked objects in the whole heap which might be very large. As a result runtime overhead of each garbage collection will be increased as the heap grows, although total number of garbage collection is reduced by always expanding the heap. In other words, user experiences will be getting worse with such heavy runtime overhead of each concurrent garbage collection.

On the contrary we can also suppress the size of heap being increased, if we choose to expand the heap only when garbage collection cannot secure sufficient free space to satisfy an allocation request. Each garbage collection can be completed in a shorter time, because the size of heap is maintained as small as it

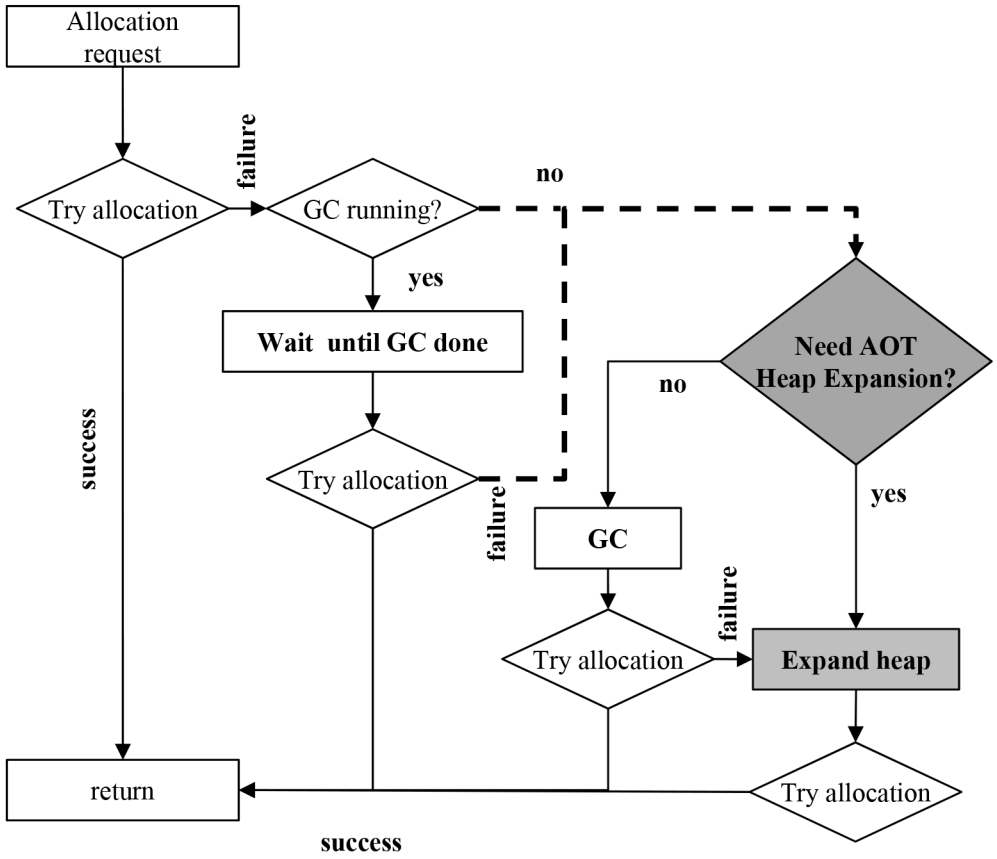

Figure 5.4 Flow of heap management with ahead-of-time heap expansion

can be, while garbage collection is invoked more frequently. Consequently total number of garbage collection will be increased and overall runtime overhead of garbage collections will be also increased, resulting in bad performance of whole Android system.

As we discussed, we have to choose heap expansion heuristic carefully, because heap expansion affects not only total heap size but also performance of whole system. In this chapter, we take into account the runtime spatial information which has been also exploited in other previous researches [44, 45, 47]

and propose a new heuristic to improve the existing heuristic in Android. Then we are going to exploit runtime temporal information to propose heuristics for better user experiences. With these spatial and temporal information, we try to expand heap in advance when there is no need to expand heap right away, i.e. ahead-of-time heap expansion.

Figure 5.4 shows how ahead-of-time heap expansion works in Android when an allocation request made. Unlike original flow in Figure 5.3, there is an additional computation to make a decision between garbage collection and heap expansion. Compared to original flow of Figure 5.3, we can avoid a garbage collection with heap expansion if certain conditions are met. In the following subsections, we will discuss what kind of information is used to make a decision.

5.4.1 Spatial heap expansion

We are going to exploit spatial information to expand the heap in ahead-of-time. There is a lot of spatial information available at runtime regarding object allocation, garbage collection and the heap. For example, size of allocated objects after the last garbage collection, size of reclaimed objects from current garbage collection and size of used heap have been exploited in other researches. [47, 17, 45, 44] Furthermore crafted information with such spatial information, such as ratio, has been also used in various ways.

Among spatial information, we choose information directly related to garbage collection to determine whether the GC is successful or not. The total size of reclaimed objects can be calculated right after the garbage collection. In Section 5.2, we measured the size of reclaimed objects and found out that garbage collection often secures relatively small free space.

We suspected that those garbage collections try to reclaim dead object repeatedly even when there are only few dead objects available. The problem is

that amount of dead objects cannot be determined before the garbage collection. We decide to expand heap when current garbage collection secures relatively small free space. By expanding the heap now, we can reduce the chance of invoking future garbage collections with few dead objects due to allocation failure. If it works, we can reduce the number of garbage collection which secures small free space and total number of garbage collection will be reduced as well. With mark-and-sweep garbage collection, we can reduce overall overhead of garbage collection by reducing the number of garbage collection. As a result, heap management with less GC overhead provide better user experiences and better overall performance.

Spatial information other than size of reclaimed objects can be used as well. We also make use of other information in ahead-of-time heap expansion framework. First, size of total free space available after the current GC is used to make a decision regarding heap expansion, because size of available free space reflects how much amount of new objects can be allocated on the heap before next allocation failure. However the size of free space is not reliable information, since it is useless if the ratio of fragmentation is getting high. Furthermore the size of free space is not flexible and sufficient information, because the size of required memory and the size of working set differ from an application to an application.

To consider different memory requirement of applications, we tried to exploit ratio of free space compared to the heap. We can adaptively consider working set of applications with the ratio instead of the size of free space. However this information is turned out to be unreliable in heap management with mark-and-sweep garbage collection. We will discuss these other spatial information in Section 5.5.

5.4.2 Temporal heap expansion

Although spatial information is very useful and provides valuable insights, it is very hard to figure out correlation between spatial information and performance, especially user experiences. [20, 48] Therefore we try to exploit temporal information in addition to spatial information, because we think that temporal information reflects performance and user experiences directly.

Like spatial information, there are a variety of temporal information with memory management such as object allocation, garbage collection, page fault and etc. Among them, we try to exploit temporal information regarding garbage collection to reduce garbage collection overhead, because garbage collection in Android adopts a stop-the-world approach which stops the whole program execution when garbage collection is running. This strategy affects user experience directly in a bad way when the pause time is getting longer.

The simple and intuitive temporal information related to the garbage collection is garbage collection pause time. However the pause time alone is not enough to determine heap expansion, because pause time of mark-and-sweep collection depends on the number of objects and the number of objects in the heap is totally determined by applications. Therefore using pause time to determine heap expansion can mislead us and cannot be applied to various applications with different set of working objects in general. If we want to reduce the pause time for an application with large working set of objects, we have to change garbage collection itself and this problem is beyond the scope of this paper as we mentioned before.

In fact, as well as garbage collection with long pause time, garbage collection with short pause time can also cause a bad user experience if such short garbage collection is invoked frequently in a short period of time. In Section 5.2, we

observed that many garbage collections were requested in a short time. Based on the observation, we are going to propose a way to expand the heap in advance when garbage collection is called multiple times in a short time interval. The simplest temporal information is an interval between garbage collections and it can be measured directly. However this information only reflects the last two garbage collection and it is not enough to determine whether many garbage collections have been invoked frequently in a short time interval.

Instead we count up the number of consecutive garbage collections only when an interval between last two garbage collections is shorter than threshold and reset the counter if the interval is longer than threshold. When the counter meets predefined number of garbage collections, we ascertain that the last consecutive garbage collections have been invoked in a limited of time. Based on the information, we predict that there will be a upcoming garbage collection in a short time again. So we decide to expand heap in advance and we anticipate no more invocation of garbage collection in a short time.

5.4.3 Launch-time heap expansion

A user does not care about user responsiveness when an application is just being launched and there is no way to interact with the application. Instead what a user expected is a fast launching of the application. Therefore we can apply a completely different heuristic for garbage collection and heap expansion when an application is being started.

First, we don't have to rely on concurrent garbage collection, because few user input is required and responsiveness doesn't matter. Unlike the previous approaches, we exploit temporal information to suppress concurrent GC instead of allocation GC. When a signal wakes up a thread for a concurrent garbage collection, we compute the time since the last garbage collection including both

allocation and concurrent garbage collection. If the time interval is shorter than a threshold, we skip a concurrent garbage collection and the thread is being slept again.

We can also expand the heap more aggressively without concerning over expanding the heap, since the heap should grow to a certain size to satisfy a minimum memory requirement of the application when the application is being started. We exploit a spatial information and temporal information to make a decision on an aggressive heap expansion. We calculate the size of free space secured by the collection after an allocation garbage collection. If the size is less than a threshold and the time since the last collection, including both concurrent and allocation, is short, we decide to grow the heap to meet a certain utilization ratio before an allocation trial.

Unlike the ahead-of-time heap expansion in previous sections, we suppress a concurrent garbage collection and we do not avoid an allocation garbage collection but expand the heap more aggressively after the garbage collection. Without considering the responsiveness, we expect less concurrent garbage collections as well as less allocation GC in overall. Since we don't skip an allocation garbage collection, we assert that we can reclaim dead objects in time when relcaiming is really necessary, and therefore we can ease the side effect of an aggressive heap expansion.

5.5 Evaluation

We evaluated proposed heuristics on Galaxy Nexus with Android 4.1.2 Jelly Bean. Galaxy Nexus is a Android smartphone with touchscreen co-developed by Google and Samsung Electronics. It contains 1GB RAM and TI OMAP 4460 which have dual-core 1.2GHz Cortex-A9 supporting ARMv7 instruction set. Android 4.1.2 supports trace-based just-in-time compiler (JITC) to acceler-

ate application execution and manages the heap with mark-and-sweep garbage collector.

Default applications of Android have been used to observe the effect of heuristics. We choose three applications to evaluate proposed approaches while running with user inputs from those default applications, e.g. `Camera`, `Gallery` and `Maps`. `Camera` and `Gallery` invoke many garbage collections but reclaims few dead objects as shown in Section 5.2. On the other hand, `Maps` provided bad user experiences, because garbage collections are invoked a lot in a short time when a user interacts with the maps application. We are going to evaluate the effect of spatial heap expansion and temporal heap expansion with these applications.

To evaluate heuristic for application launching, we use 11 applications including above three applications. These applications include very simple applications as well as complex ones, i.e. `Gallery`, `Calculator`, `MMS`, `Settings`, `Deskclock`, `email`, `Browser`, `Maps`, `Calendar`, `Contacts` and `youtube`.

5.5.1 Spatial heap expansion

We choose threshold to be 10 kilobytes for spatial heap expansion heuristic considering size of reclaimed objects to compare behavior of garbage collection with original one shown in Section 5.2.

Figure 5.5 and Figure 5.6 depict garbage collections distribution depending on the size of reclaimed objects in `Camera` and `Gallery`. About one fourth of garbage collection in camera and about half of collection in gallery secured free space less than 10 kilobytes with original Android heuristic. After applying ahead-of-time heap expansion with spatial information, garbage collection distribution is changed. Ratio of garbage collections which reclaimed less than 10KB of objects has been reduced in both applications. Most of the reduction

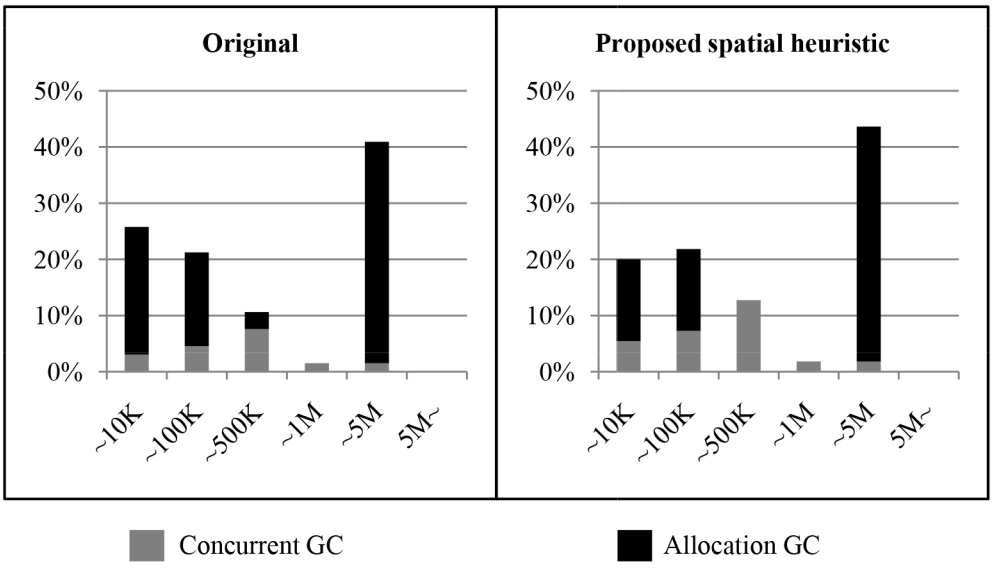

Figure 5.5 GC distribution by the size of reclaimed objects in `Camera`

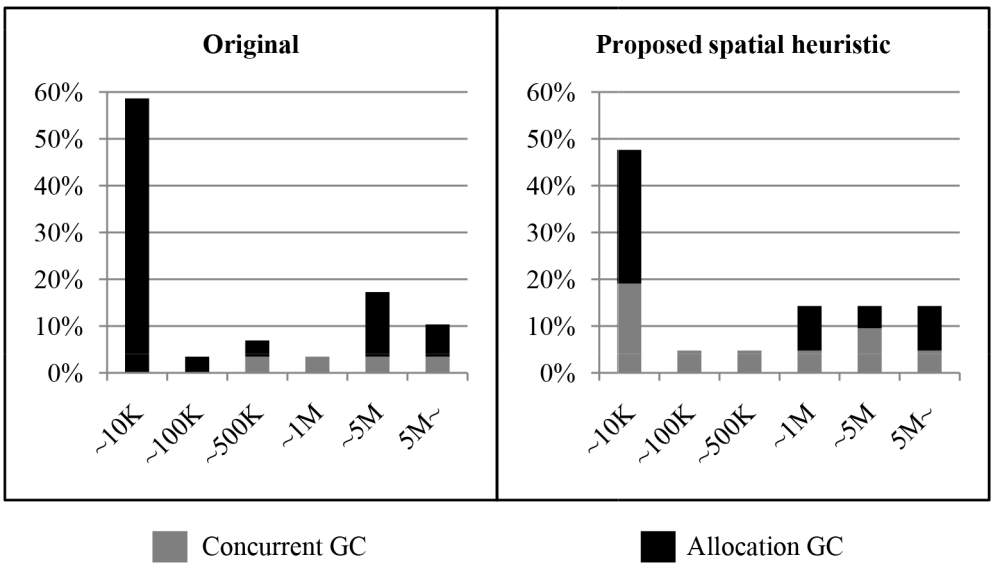

Figure 5.6 GC distribution by the size of reclaimed objects in `Gallery`

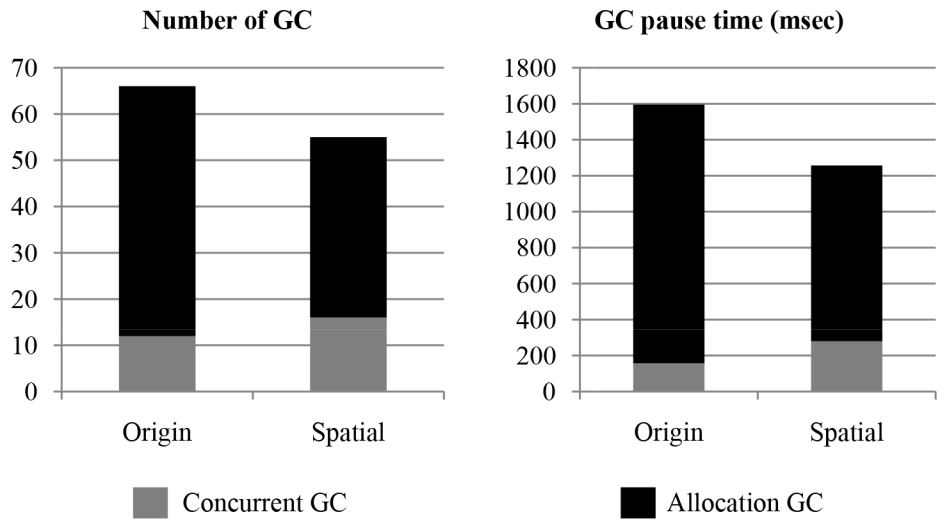

Figure 5.7 Changes of GC behavior in `Camera` after applying spatial heuristic

is due to reduction of allocation GC, while ratio of concurrent GC has been increased. This is expected consequences, because proposed heuristic avoids allocation GC and concurrent GC has more opportunities to be invoke due to less invocation of allocation GC.

We also observed changes of garbage collection behavior as shown in Figure 5.7 and 5.8. Total number of garbage collections is also reduced after applying spatial heap expansion in both applications. Especially allocation GC which is requested when an object allocation failure occurs has been invoked less than original. As discussed before this was expected, because spatial heap expansion has been proposed to avoid allocation garbage collection by expanding heap aggressively. Total pause time of garbage collection has been also reduced as the total number of garbage collection is reduced, although concurrent GC spends more time than before. We shorten the pause time 21.2% in camera and 31% in gallery.

While we reduced the pause time, max size of heap has been increased

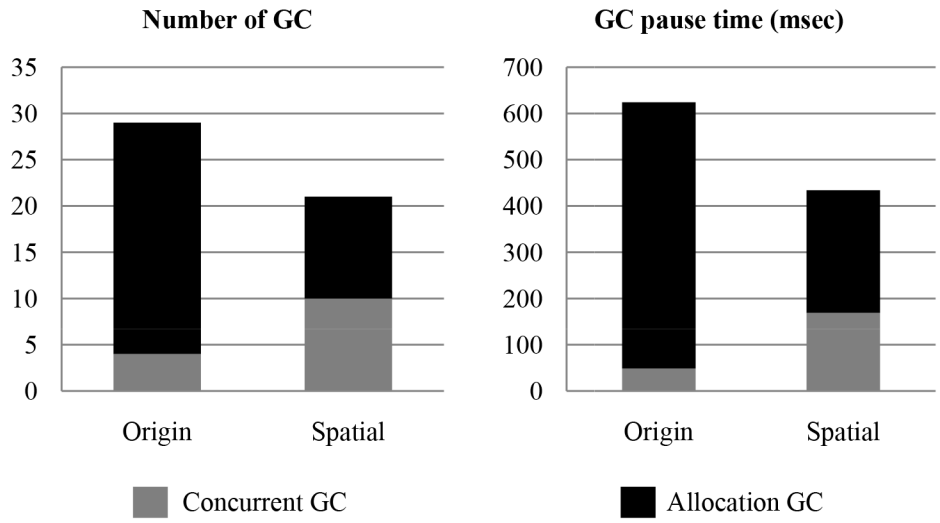

Figure 5.8 Changes of GC behavior in `Gallery` after applying spatial heuristic

somewhat as side effect due to aggressive heap expansion. With ahead-of-time heap expansion, camera requires 18.8% more heap, i.e. from 25.6MB to 30.4MB, and gallery allocates 3.5% more heap , i.e. from 37.6MB to 38.9MB.

5.5.2 Comparision of spatial heap expansion

We evaluate spatial heuristics with size of reclaimed objects in previous section. We also implemented and evaluated ahead-of-time heap expansion with other spatial information, such as size of free space and ratio of free space. Figure 5.9, 5.10 and 5.11 compares all four spatial heuristics, including original, size of reclaimed objects, size of free space and ratio of free space. `Cameara` application is used for the comparison.

Figure 5.9 describes the GC distribution after applying each heuristic. We found out that two spatial heuristic, i.e. size of reclaimed objects and size of free space, are effective in reducing the number of garbage collection with small size of reclaimed objects. Therefore it is reasonable to use those two spatial

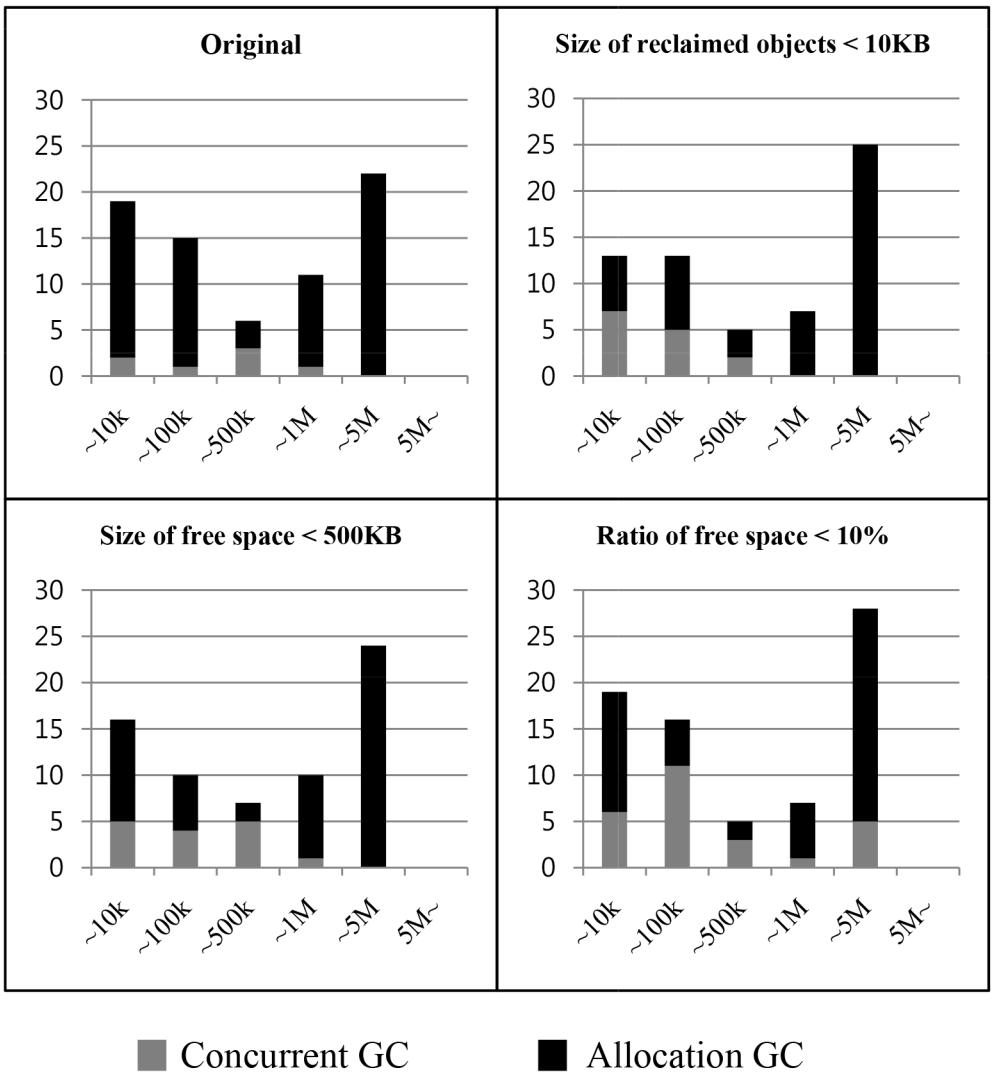

Figure 5.9 GC distribution depending on size of reclaimed objects in `Camera`

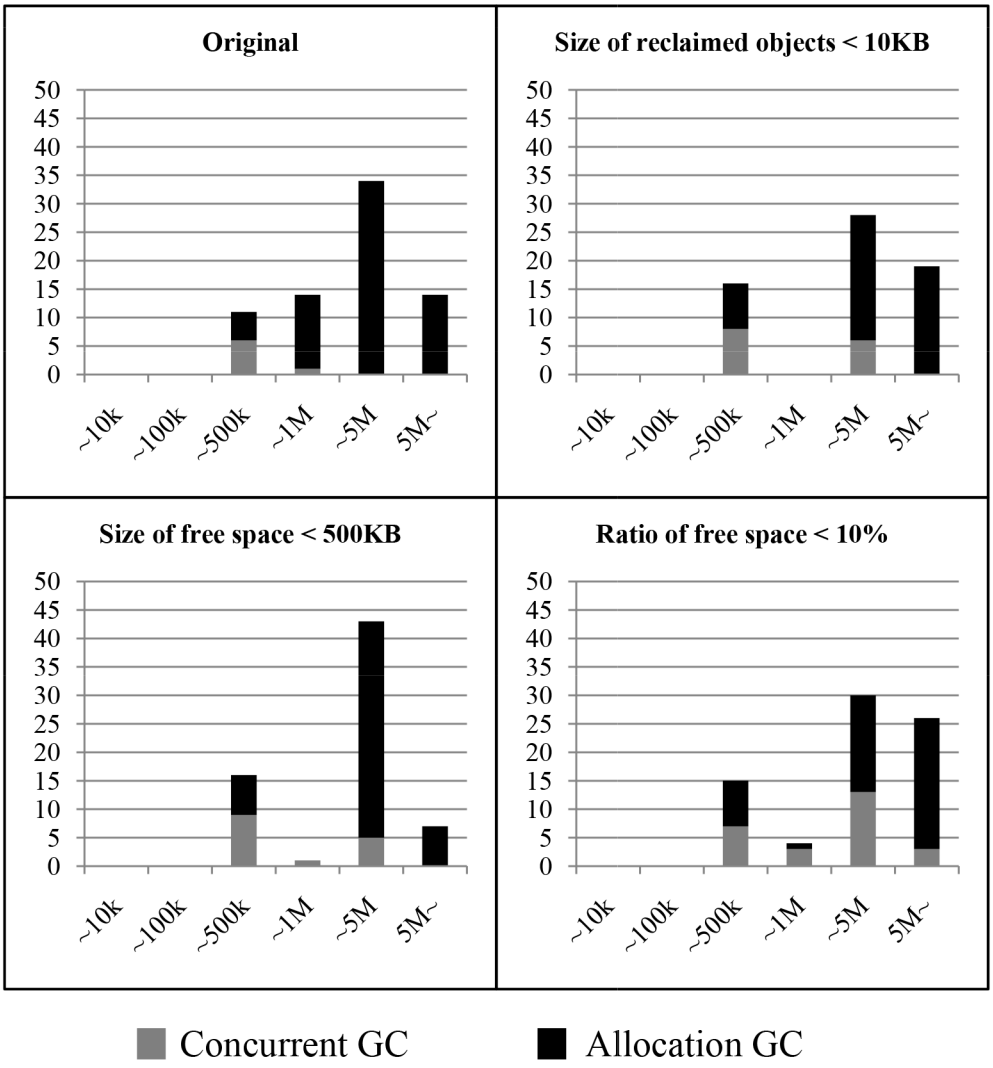

Figure 5.10 GC distribution depending on size of free space in `Camera`

information to predict future behavior of garbage collections.

When we examine the GC distribution by the size of free space as in Figure 5.10, we didn't find meaningful changes except slight changes in distribution. Even with the spatial heuristic with size of free space, there are still allocation

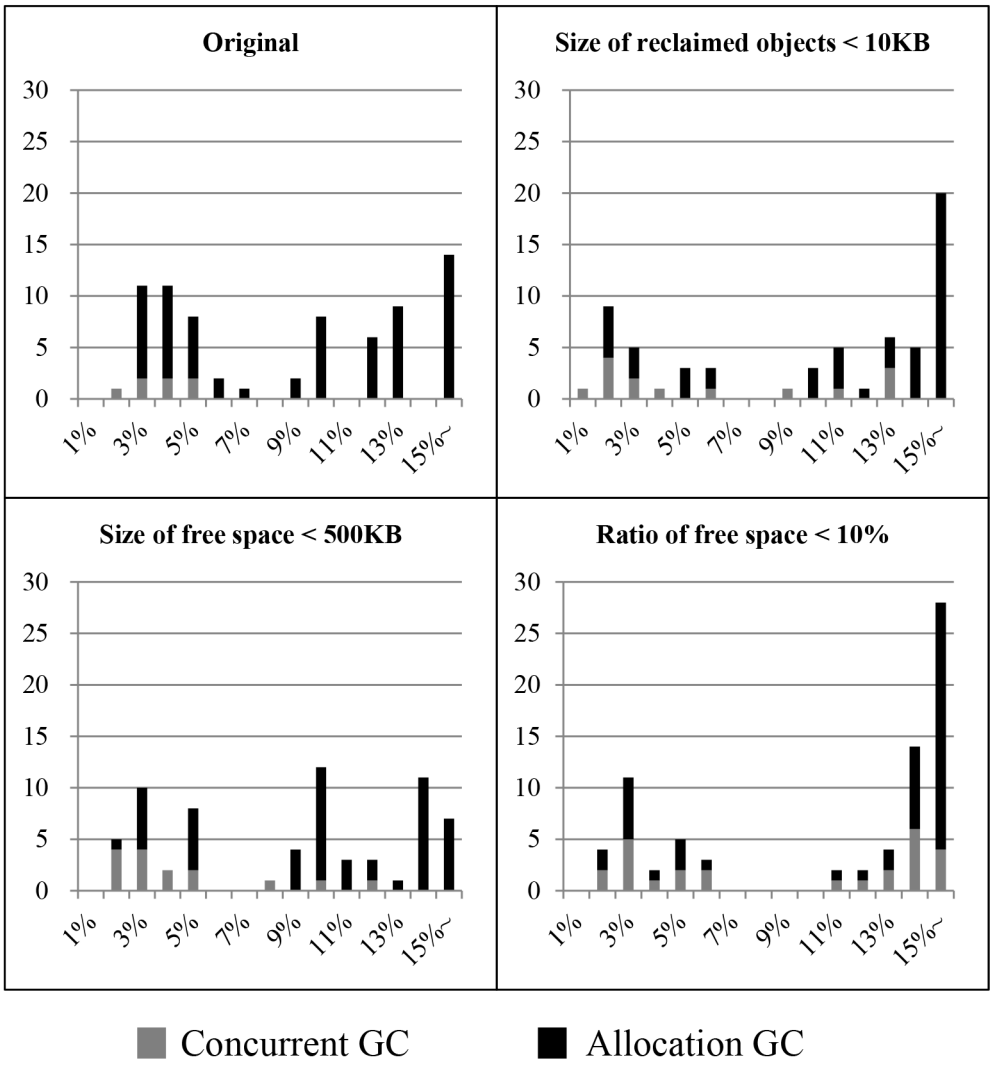

Figure 5.11 GC distribution depending on ratio of free space in `Camera`

GCs which produces less than 500KB free space. From the result, we suspect that size of total free space after the current garbage collection does not guarantee future behaviors of garbage collections.

Finally we look into the ratio of free space after applying four spatial heuristics as shown in Figure 5.11. Two spatial heuristics have changed the distribu-

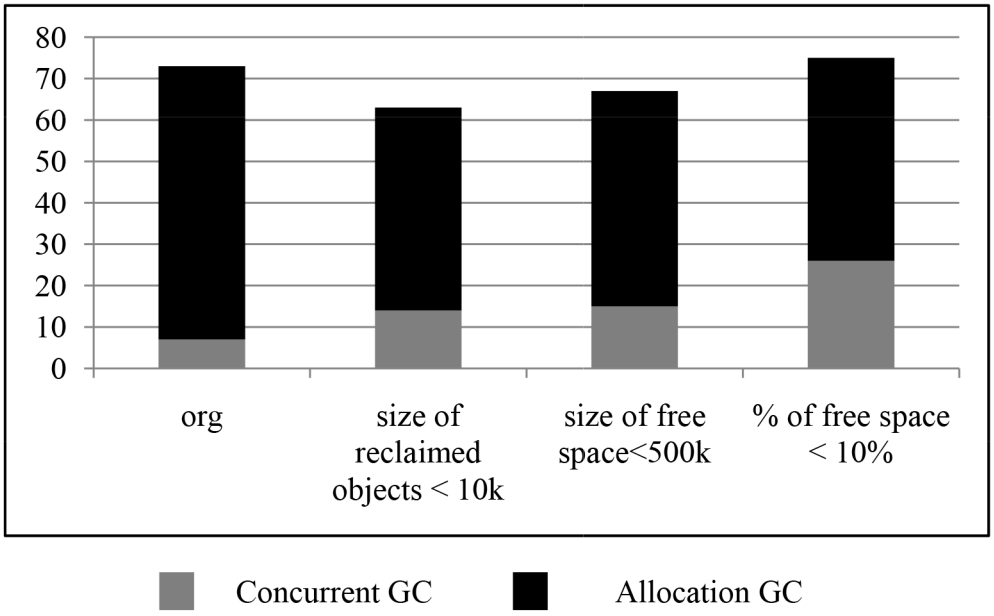

Figure 5.12 Total number of garbage collections of `Camera` with different heuristics

tion of GC. Heuristics with size of reclaimed objects and ratio of free space secures relatively more free space than before. It was expected that number of garbage collections which secures relatively less free space has been reduced with a heuristic with ratio of free space. However we are not convinced whether this changes is beneficial or not, because securing more free space does not promise better performance.

To evaluate the performance of each spatial heuristic, we measured the number and total pause time of GC in Figure 5.12 and 5.13. All three spatial heuristic reduce the number of allocation GC while number of concurrent GC increased. A heuristic with ratio of free space results in more number of GC when considering both allocation GC and concurrent GC. A proposed spatial heuristic with reclaimed object shows the least number of GC overall. Same

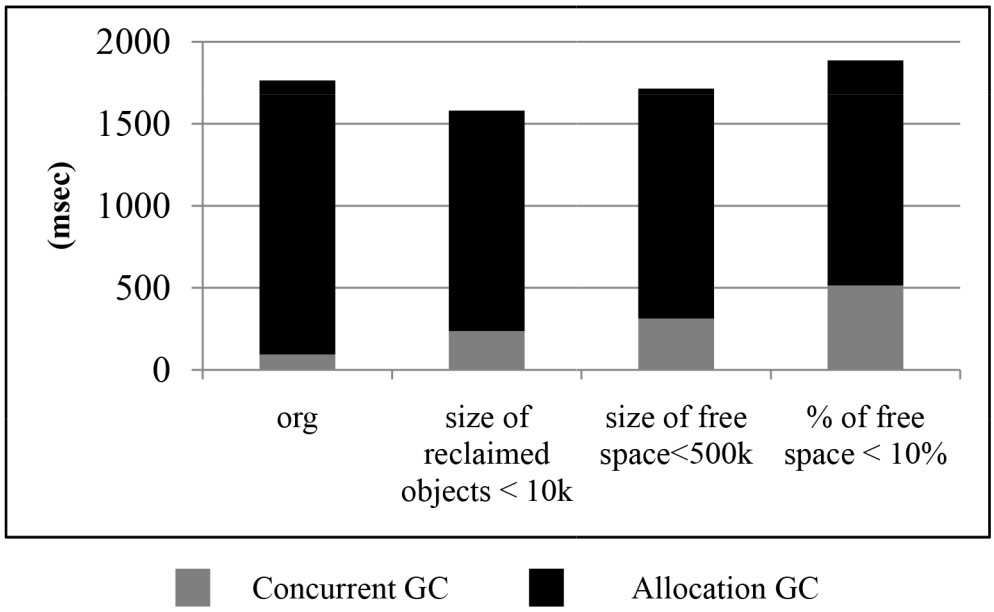

Figure 5.13 GC pause time of `Camera` with different heuristics

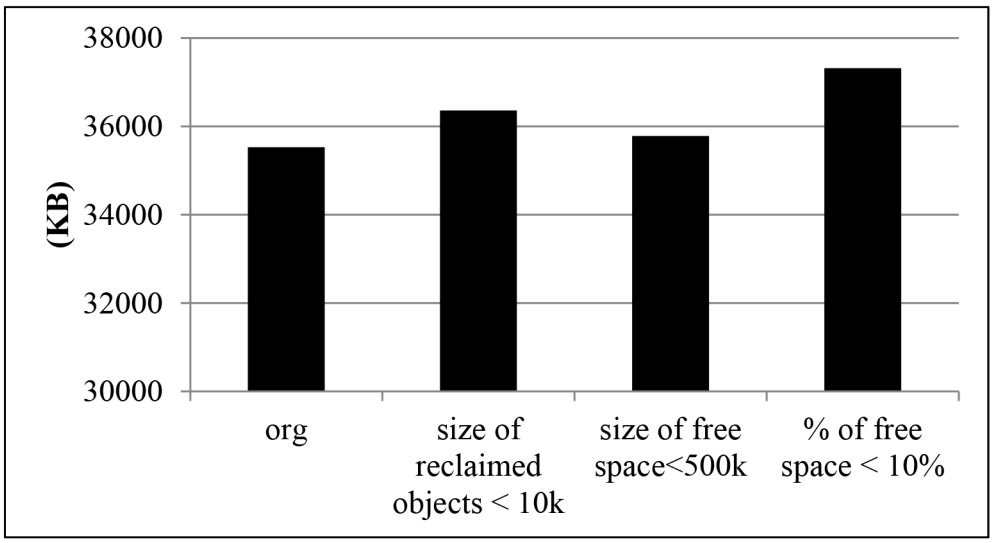

Figure 5.14 Size of max heap in `Camera` with different heuristics

result can be found with pause time of GC as in Figure 5.13, since number of GC and pause time of GC are strongly correlated when mark-and-sweep GC is used.

Size of max heap is also measured in Figure 5.14 to check the side effect of aggressive heap expansion. Original heuristic without ahead-of-time heap expansion shows the smallest size of max heap and it is expected as well, because it always invokes GC before expanding heap. The heuristic with size of reclaimed objects shows the best performance but requires more heap as discussed before.

We decide to track overall behavior of heap to analyze the effect of each heap expansion approach in more detail. During the execution of an application, we traced the size of heap and live objects when each garbage collection completed. The size of live objects is computed during mark phase of garbage collection and the size of heap is measured after heap expansion occurred. We also calculate the ratio of free space compared to total heap. Figure 5.15 and 5.16 show these values regarding each spatial heap expansion heuristic.

All four heuristics show that heap grows as time goes and the size of heap converges to the size of max heap. With ahead-of-time heap expansion, heap grows more rapidly than original in early time. Size of live objects also increases and converges at some point, and this should be same regardless of heuristics because size of live objects is solely depends on the behavior of the application. Therefore we can easily infer that size of free space may increase at first and converges to some point, since size of free space can be directly computed by subtracting size of live objects from size of total heap. Therefore a heuristic with size of free space may not work correctly after some point and threshold should be adaptively changed to cope with such application behavior. Finally ratio of free space is also increasing as time goes and we find out this was mainly due to fragmentation problem in mark-and-sweep garbage collector. Therefore we

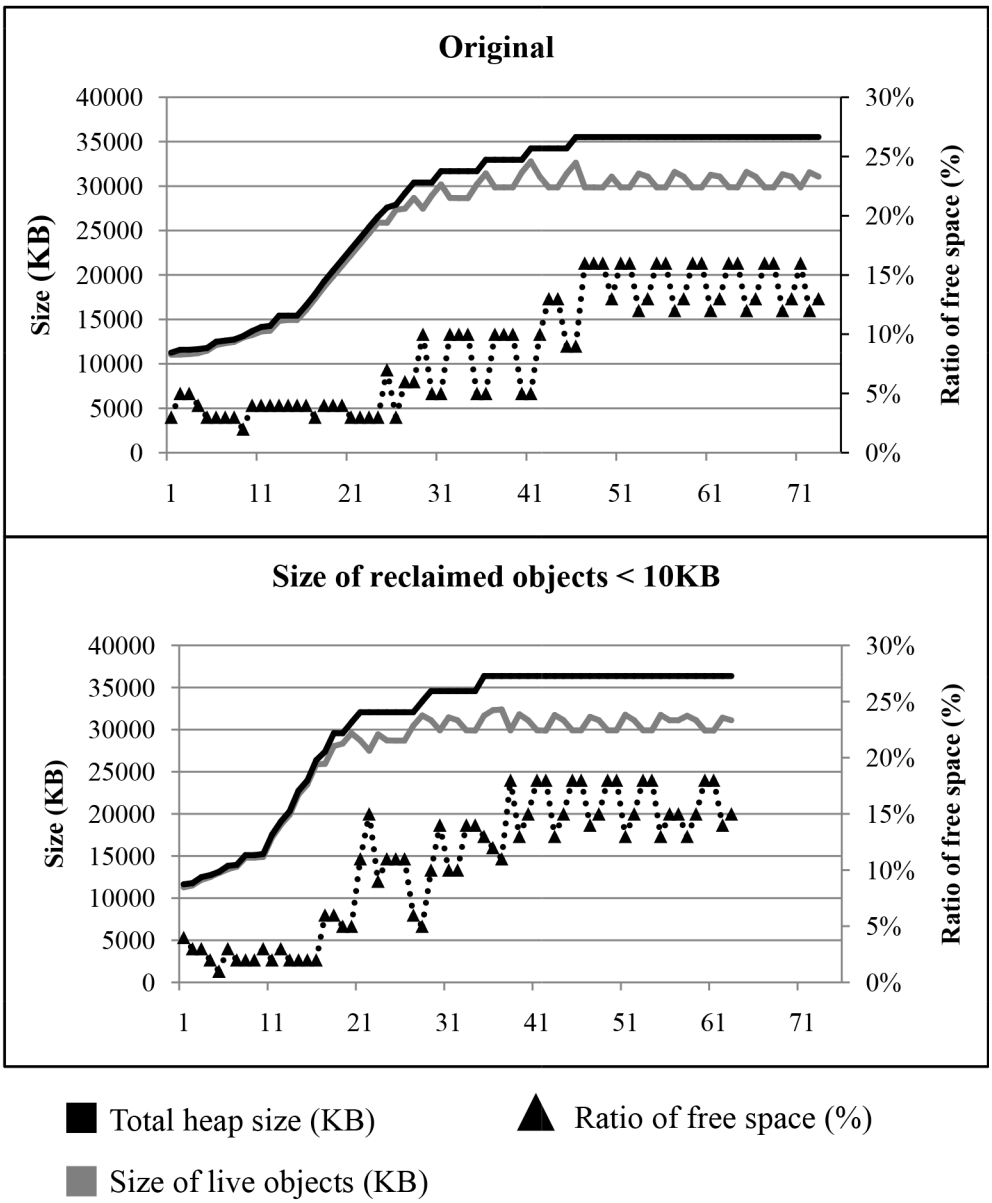

Figure 5.15 Heap behavior of `Camera` with original and proposed heuristics. X-axis denotes each garbage collection and left y-axis depicts the total heap size and the size of live objects in kilobytes, while right y-axis shows ratio of free space in percentage.

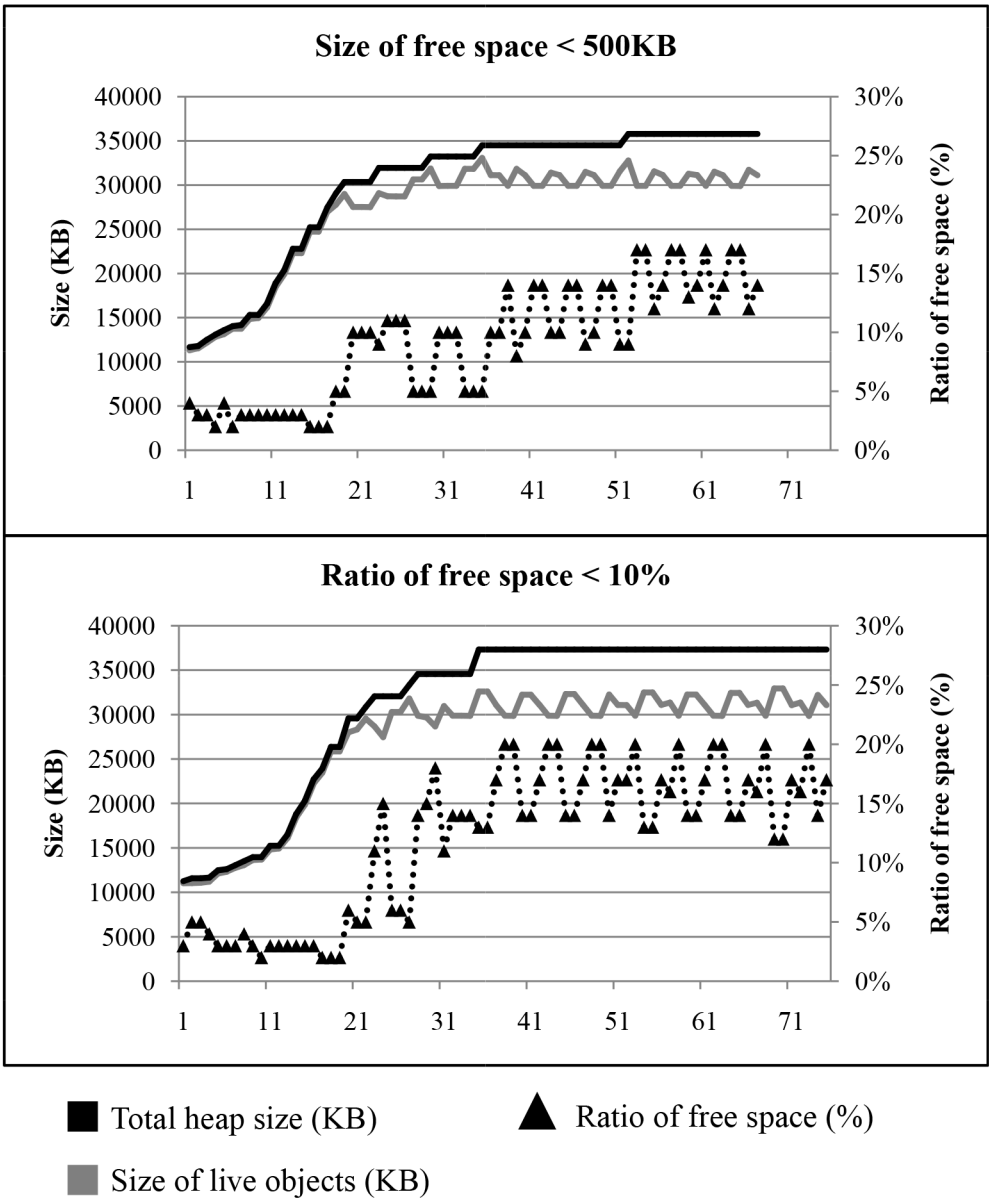

Figure 5.16 Heap behavior of `Camera` with other spatial heuristics. X-axis denotes each garbage collection and left y-axis depicts the total heap size and the size of live objects in kilobytes, while right y-axis shows ratio of free space in percentage.

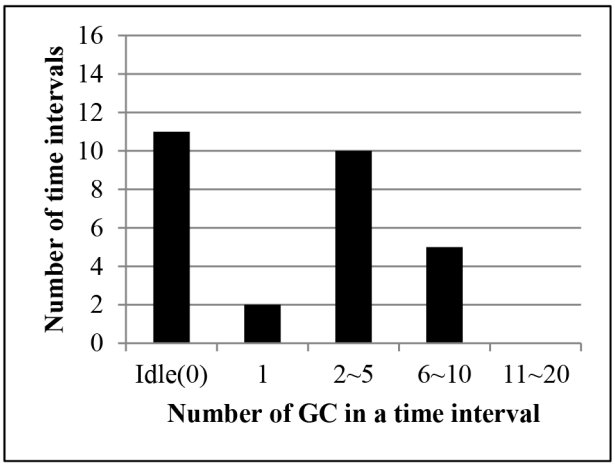

Figure 5.17 Number of time intervals depending on the number of GC in a time interval after applying temporal heap expansion in `Maps`

can conclude that fixed size of free space or ratio of free space are not reliable information to determine ahead-of-time heap expansion with mark-and-sweep garbage collection here, while size of reclaimed object is reliable information to predict future behavior of garbage collection.

5.5.3 Temporal heap expansion

We also evaluate ahead-of-time heap expansion with temporal information. Figure 5.2 in the section 5.2 shows time interval distribution depending on the number of garbage collections invoked within a time interval, where each time interval is one second. We count up the number of garbage collection if time interval between two garbage collection is less than 300ms. Then we expand heap ahead-of-time when counter exceed the threshold. A histogram of time interval after applying ahead-of-time temporal heap expansion is shown in Figure 5.17. Compared to Figure 5.2, we can easily observe that we completely removed time intervals where garbage collection is invoked more than 10 times

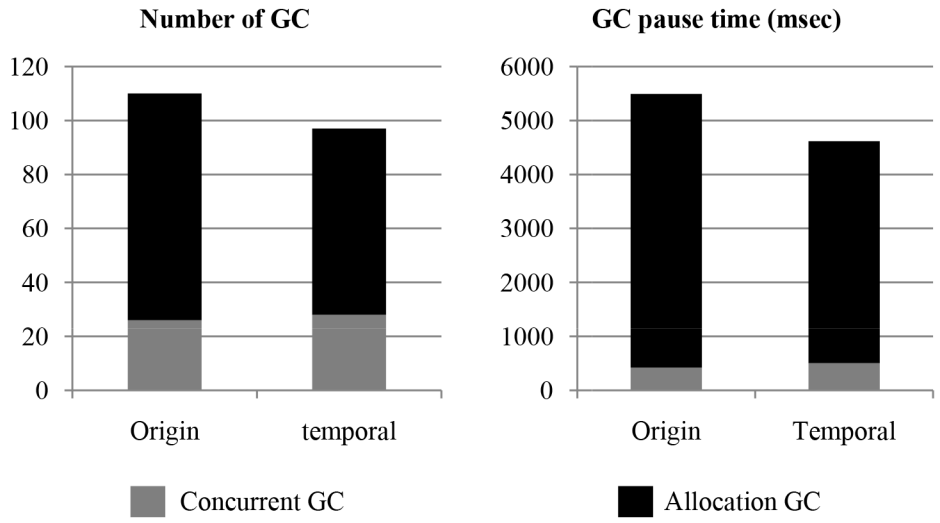

Figure 5.18 Changes of GC behavior in `Maps` after applying temporal heuristic

in a second. We also observed that much less lags were observed when a user interacts with the `Maps` application but it cannot be measured quantitatively. We figure out the improvement qualitatively by recording the behavior of maps application in video and comparing them.

Temporal heap expansion also reduces total number of garbage collections by avoiding garbage collection with timely heap expansion, especially allocation garbage collection. In consequences, number of GC and pause time of GC are reduced in meaningful amount as shown in Figure 5.18. Like spatial heap expansion, allocation GC is avoided with temporal heap expansion, because we expands heap when allocation failure occurred and garbage collection has been invoked too much in a short time.

Although we expand heap based on temporal information other than spatial information, max size of heap has been increased with temporal heuristic. Because we expand the heap even when garbage collection can secure sufficient free space, heap expansion occurred more frequently than before. In `Maps` appli-

Table 5.1 Number of garbage collection and heap expansion. Only heap expansion due to allocation failure after the GC has been counted

Benchmarks	Before	After
Allocation GC	109	111
Concurrent GC	135	112
Heap Expansion	26	16
Total	270	239

Table 5.2 Pause time of garbage collections

Pause time(msec)	Before	After
Allocation GC	5144	5065
Concurrent GC	1353	1077
Total	6497	6142

cation, we require 10.9% more heap than before, e.g. from 27.4MB to 30.4MB.

5.5.4 Launch-time heap expansion

We evaluated a launch-time heuristic with spatial and temporal information when applications start to run. Total 11 applications are launched and applications have been launched explicitly in serial manner five times.

We measured number of garbage collections and number of heap expansion due to allocation failure as in Table 5.1. Concurrent GC has been invoked much less than before, because we avoid the concurrent GC with the heuristic as well as allocation GC. When the last garbage collection, regardless of concurrent or allocation, has already reclaimed objects shortly before, we skip concurrent GC. The number of heap expansion due to allocation failure has been reduced, since we expand heap aggressively to secure sufficient free space after allocation garbage collection when temporal and spatial thresholds are met.

We also measured pause time caused by garbage collections in Table 5.2. Overall pause time has been reduced 5.5% and most of the improvement has been from the concurrent garbage collection as we already expected, because

the number of concurrent garbage collection have been reduced.

5.6 Summary

In this chapter, we propose ahead-of-time heap expansion heuristics to avoid bad garbage collection behavior in Android with temporal and spatial heuristic.

We proposed an ahead-of-time heap expansion framework to enhance existing Android heap management heuristic. Then size of reclaimed objects is considered to determine ahead-of-time heap expansion in addition to existing utilization information. Two more kinds of spatial information are exploited and evaluated with size of reclaimed object. We also exploited temporal information to detect bad garbage collection behavior when many GCs are invoked in a short time and to apply ahead-of-time heap expansion. In such case, we skip next GC invocation by expanding heap ahead-of-time instead of GC. Finally we also propose a heuristic when an application is being launched where the responsiveness doesn't matter. We evaluated proposed heuristics with default key applications in Android. Results show that we can relieve the situation where GCs are invoked many times but reclaim relatively few objects and too many GCs are invoked in a short time. Also we reduce total pause time caused by garbage collections when an application is launched by a user.

We exploit three spatial information and one temporal information in this paper. We can refine these information more carefully and there can be more kinds of information which might be useful for ahead-of-time heap expansion. We use a totally different heuristic when an application starts, but we expect that more improvement can be achieved if we can apply different heuristics of ahead-of-time heap expansion adaptively as an application behavior changes.[47]

Chapter 6

Conculsion

In this paper, I propose three optimizing approaches for memory management in virtual machine. Proposed approaches address memory management issues including object allocation, garbage collection and heap management. Memory management issues of a variety of virtual machine including Dalvik virtual machine in Android platform which is widely spread recently as well as famous Java virtual machine are considered. Also wide range of virtual machine environment is considered including embedded, mobile and server environment.

First, I've proposed a lazy worst fit allocator which is a fast object allocator with low fragmentation. Proposed allocation has been implemented in Java virtual machine and has been evaluated on desktop and server environment. A lazy worst fit allocator outperforms other allocators including segregated first fit and lazy first fit and shows good fragmentation as low as first fit allocator which is known to have the lowest fragmentation.

Secondly, a biased allocator is suggested to address extra overhead of generational garbage collector. A proposed approach has been implemented in embed-

ded Java virtual machine and evaluated on embedded device including digital TV. With three analyses, a biased allocator reduces 4.1% of pause time caused by generational garbage collections in average.

Finally, ahead-of-time heap expansion framework is introduced to avoid worst-case behavior of garbage collection. The proposed approach has been implemented in Dalvik virtual machine of Android platform and evaluated on mobile device, i.e. smartphone, with real applications. Ahead-of-time heap expansion reduces both number of garbage collections and total pause time of garbage collections. Pause time of GC reduced up to 31% in default applications of Android platform.

Memory management deals with a variety of issues and new problems are raised as new devices and software environment are being introduced. These problems are complicated, because several issues are interconnected each other, including object allocation, garbage collection and heap management. I've addressed problems of object allocation, garbage collection and heap management separately, but also tried to address garbage collection overhead by introducing new allocator and new heap management technique. I hope such approaches is useful to deal with future problems in memory management.

Bibliography

[1] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] D. Ehringer, "The dalvik virtual machine architecture," *Techn. report (March 2010)*, 2010, http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf.

[3] "Android official website." [Online]. Available: http://www.android.com

[4] "900 million Android activations!" May 2013, Google I/O 2013. [Online]. Available: https://developers.google.com/events/io/2013/, http://www.youtube.com/watch?v=1CVbQttKUIk

[5] "Interactive tv web," http://www.interactivetvweb.org.

[6] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[7] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Addison-Wesley, 1996.

[8] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, 1st ed. New York, NY, USA: John Wiley and Sons, Inc., 1996.

[9] P. Wilson, M. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Memory Management*, ser. Lecture Notes in Computer Science, H. Baler, Ed. Springer Berlin Heidelberg, 1995, vol. 986, pp. 1–116. [Online]. Available: http://dx.doi.org/10.1007/3-540-60368-9_19

[10] C. J. Cheney, "A nonrecursive list compacting algorithm," *Commun. ACM*, vol. 13, no. 11, pp. 677–678, Nov. 1970. [Online]. Available: http://doi.acm.org/10.1145/362790.362798

[11] P. Wilson, "Uniprocessor garbage collection techniques," in *Memory Management*, ser. Lecture Notes in Computer Science, Y. Bekkers and J. Cohen, Eds. Springer Berlin Heidelberg, 1992, vol. 637, pp. 1–42. [Online]. Available: http://dx.doi.org/10.1007/BFb0017182

[12] Y. Chung and S.-M. Moon, "Memory allocation with lazy fits," in *Proceedings of the 2Nd International Symposium on Memory Management*, ser. ISMM '00. New York, NY, USA: ACM, 2000, pp. 65–70. [Online]. Available: http://doi.acm.org/10.1145/362422.362457

[13] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Memory allocation policies reconsidered," Technical report, University of Texas at Austin Department of Computer Sciences, Tech. Rep., 1995.

[14] W. T. Comfort, "Multiword list items," *Commun. ACM*, vol. 7, no. 6, pp. 357–362, Jun. 1964. [Online]. Available: http://doi.acm.org/10.1145/512274.512288

[15] D. E. Knuth, *The art of computer programming, Volumn 1: Fundamental algorithms*, 3rd ed. Boston, MA, USA: Addison-Wesley Professional, 1997.

[16] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman, "Latte: A Java VM just-in-time compiler with fast and efficient register allocation," in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 128–. [Online]. Available: http://dl.acm.org/citation.cfm?id=520793.825720

[17] Y. C. Chung, S.-M. Moon, K. Ebcioğlu, and D. Sahlin, "Reducing sweep time for a nearly empty heap," in *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '00. New York, NY, USA: ACM, 2000, pp. 378–389. [Online]. Available: http://doi.acm.org/10.1145/325694.325744

[18] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: Solved?" in *Proceedings of the 1st International Symposium on Memory Management*, ser. ISMM '98. New York, NY, USA: ACM, 1998, pp. 26–36. [Online]. Available: http://doi.acm.org/10.1145/286860.286864

[19] A. W. Appel, "Simple generational garbage collection and fast allocation," *Software: Practice and Experience*, vol. 19, no. 2, pp. 171–183, 1989. [Online]. Available: http://dx.doi.org/10.1002/spe.4380190206

[20] M. Hertz, Y. Feng, and E. D. Berger, "Garbage collection without paging," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05.

New York, NY, USA: ACM, 2005, pp. 143–153. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065028

[21] F. Xian, W. Srisa-an, C. Jia, and H. Jiang, "AS-GC: An efficient generational garbage collector for Java application servers," in *Proceedings of the 21st European Conference on Object-Oriented Programming*, ser. ECOOP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 126–150. [Online]. Available: http://dl.acm.org/citation.cfm?id=2394758.2394768

[22] P. Reames and G. Necula, "Towards hinted collection: Annotations for decreasing garbage collector pause times," in *Proceedings of the 2013 International Symposium on Memory Management*, ser. ISMM '13. New York, NY, USA: ACM, 2013, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2464157.2464158

[23] Y. Levanoni and E. Petrank, "An on-the-fly reference counting garbage collector for Java," in *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '01. New York, NY, USA: ACM, 2001, pp. 367–380. [Online]. Available: http://doi.acm.org/10.1145/504282.504309

[24] "Memory management in the Java HotSpot virtual machine," Apr. 2006, http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf.

[25] D. Doligez and X. Leroy, "A concurrent, generational garbage collector for a multithreaded implementation of ml," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '93. New York, NY, USA: ACM, 1993, pp. 113–123. [Online]. Available: http://doi.acm.org/10.1145/158511.158611

[26] A. Krall, "Efficient JavaVM just-in-time compilation," in *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 205–. [Online]. Available: http://dl.acm.org/citation.cfm?id=522344.825703

[27] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "Adaptive optimization in the Jalapeno JVM," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '00. New York, NY, USA: ACM, 2000, pp. 47–65. [Online]. Available: http://doi.acm.org/10.1145/353171.353175

[28] J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 97–113, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/857076.857077

[29] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson, "Toba: Java for applications a way ahead of time (wat) compiler," in *Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3*, ser. COOTS'97. Berkeley, CA, USA: USENIX Association, 1997, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268028.1268031

[30] A. Varma and S. S. Bhattacharyya, "Java-through-C compilation: An enabling technology for Java in embedded systems," in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 3*, ser. DATE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 30 161–. [Online]. Available: http://dl.acm.org/citation.cfm?id=968880.969233

[31] A. Nilsson and S. Robertz, "On real-time performance of ahead-of-time compiled Java," in *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 372–381.

[32] H.-K. Choi, D.-H. Jung, and S.-M. Moon, "Install-time compiler for embedded mobile devices," in *Proceedings of Workshop on Interaction between Compilers and Computer Architectures*, ser. INTERACT-12, 2008.

[33] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proceedings of the 9th European Conference on Object-Oriented Programming*, ser. ECOOP '95. London, UK, UK: Springer-Verlag, 1995, pp. 77–101. [Online]. Available: http://dl.acm.org/citation.cfm?id=646153.679523

[34] G. Snelting and F. Tip, "Understanding class hierarchies using concept analysis," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 3, pp. 540–582, May 2000. [Online]. Available: http://doi.acm.org/10.1145/353926.353940

[35] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for Java," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '99. New York, NY, USA: ACM, 1999, pp. 1–19. [Online]. Available: http://doi.acm.org/10.1145/320384.320386

[36] D. Gay and B. Steensgaard, "Stack allocating objects in Java," Microsoft Research, Tech. Rep., 1999.

[37] "Java HotSpot^{TM} virtual machine performance enhancements," 2013, http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html.

[38] M. Hirzel, J. Henkel, A. Diwan, and M. Hind, "Understanding the connectivity of heap objects," in *Proceedings of the 3rd International Symposium on Memory Management*, ser. ISMM '02. New York, NY, USA: ACM, 2002, pp. 36–49. [Online]. Available: http://doi.acm.org/10.1145/512429.512435

[39] "Phoneme project," https://java.net/projects/phoneme.

[40] "SPECjvm98 documentation," 1999, http://www.spec.org/osg/jvm98/jvm98/doc/index.html.

[41] "JavaScriptCore," http://trac.webkit.org/wiki/JavaScriptCore.

[42] "JS Core Garbage Collector," http://trac.webkit.org/wiki/JS%20Core%20Garbage%20Collector.

[43] E. Andreasson, F. Hoffmann, and O. Lindholm, "To collect or not to collect? machine learning for memory management." in *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, S. P. Midkiff, Ed. Berkeley, CA, USA: USENIX Association, 2002, pp. 27–39.

[44] S. Soman, C. Krintz, and D. F. Bacon, "Dynamic selection of application-specific garbage collectors," in *Proceedings of the 4th International Symposium on Memory Management*, ser. ISMM '04. New York, NY, USA: ACM, 2004, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/1029873.1029880

[45] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Automatic heap sizing: Taking real memory into account," in *Proceedings of the 4th International Symposium on Memory Management*, ser. ISMM

'04. New York, NY, USA: ACM, 2004, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/1029873.1029881

[46] D. Buytaert, K. Venstermans, L. Eeckhout, and K. De Bosschere, "Garbage collection hints," in *Proceedings of the First International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 233–248. [Online]. Available: http://dx.doi.org/10.1007/11587514_16

[47] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara, "Program-level adaptive memory management," in *Proceedings of the 5th International Symposium on Memory Management*, ser. ISMM '06. New York, NY, USA: ACM, 2006, pp. 174–183. [Online]. Available: http://doi.acm.org/10.1145/1133956.1133979

[48] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI. New York, NY, USA: ACM, 2004, pp. 177–188. [Online]. Available: http://doi.acm.org/10.1145/1024393.1024415

요약

메모리 관리는 가상머신의 핵심 기능 중 하나이며 가상머신의 성능에 큰 영향을 준다. 자바와 같은 가상머신을 위한 최신의 프로그래밍 언어들은 동적 메모리 할당 기법을 사용하며 객체를 heap에서 자주 할당한다. 이렇게 할당된 객체들은 추후 더 이상 사용되지 않게 되면 추후 할당할 객체들을 위한 빈 공간을 확보하기 위해 회수된다. 많은 가상 머신들이 쓰레기 수집기라 불리는 기법을 채택하여 heap에서 사용하지 않는 죽은 객체들을 회수한다. 반면에 heap 자체의 크기를 늘려서 더 많은 객체를 할당하도록 할 수도 있다. 이처럼 메모리 관리의 성능은 객체 할당기법, 쓰레기 수집기 그리고 heap 관리 기법에 의해서 결정된다.

본 논문에서는 가상머신에서 메모리 관리 성능을 향상시키기 위한 세가지 기법을 제안하려고 한다. 우선 lazy worst fit이라는 객체 할당기법을 제안하여 쓰레기 수집기가 있는 가상머신에서 작은 객체들을 빠르게 할당할 수 있도록 하였다. 다음으로 biased allocator를 제안하여 쓰레기 수집기의 추가적인 시간 소모를 줄여 쓰레기 수집기의 수행 시간을 줄일 수 있도록 하였다. 마지막으로 ahead-of-time heap expansion기법을 제안하여 쓰레기 수집기의 호출을 억제하여 사용자 반응성과 메모리 관리 성능을 개선시키도록 하였다.

이렇게 제안된 기법들은 데스크톱, 내장형 그리고 모바일 기기 등과 같은 다양한 환경에서 구현되어 평가되었으며, Java 수행환경을 위한 자바 가상 머신과 Android 환경을 위한 Dalvik 가상머신에 적용되었다. Lazy worst fit 객체 할당기는 다른 할당 기법들과 비교해서 압도적인 성능을 보였으며, 가장 좋은 단편화 현상을 보이는 first fit과 비슷한 수준의 단편화 현상을 보여주었다. Biased allocator는 쓰레기 수집기의 수행시간을 평균적으로 4.1%의 개선하였다. Ahead-of-time heap expansion 기법은 쓰레기 수집기의 수행 횟수와 시간을

모두 줄일 수 있었다. Android 환경의 기본 응용 프로그램들을 이용하여 평가하였을 때, 쓰레기 수집기의 수행 시간은 최대 31% 줄일 수 있었다.

주요어: 최적화, 가상머신, 메모리 관리, 객체 할당, 쓰레기 수집기, 힙 관리
학번: 2002-30447

Acknowledgements

대학원을 시작하면서 알게 된 가상 머신이 최근에는 일반인들에게도 널리 쓰이고 있어 시간이 많이 흘렀음을 느끼게 되며 지금까지 옆에서 기다리면서 언제나 응원을 해 준 가족들 특히 아내 윤경이에게 고마운 마음을 전합니다. 또한 언제나 밝은 모습으로 삶의 활력을 불어 넣어준 종원이와 지민에게도 고맙다는 말을 하고 싶습니다. 또한 마음 고생 많이 시켜드렸는데도 묵묵히 응원해 주신 부모님에게도 감사 드리고 동생에게도 고맙다는 말을 전하고 싶습니다.

다양한 연구 경험을 제공해 주시고 필요한 조언을 해주시며 지도해 주신 지도교수님께 감사 드립니다. 또한 바쁘신 중에도 박사 논문 지도를 위해 시간을 내어주신 백윤흥 교수님, 이재진 교수님, 이혁재 교수님에게 감사 드립니다. 그리고 마지막으로 박사 심사에 위원으로 참여하여 시간을 쪼개어 여러 조언을 아끼지 않은 김수현 선배님에게 감사하다는 말을 전하고 싶습니다.

언제 봐도 반가운 친구들, 성엽, 동희, 준석, 성수, 철오 등에게도 덕분에 어려운 일이나 좋은 일이 있을 때 힘을 얻을 수 있었다고 말을 전하고 싶고 앞으로도 계속 변치 않기를 바라며 대학에서 엔지니어로서의 고민 그리고 이제는 삶에 대한 고민까지 나눌 수 있는 친구들인 영균, 용하, 재목, 정환, 성국, 용식, 기린, 영규, 재영, 효진 등에게도 같은 말을 전하고 싶다.

그리고 연구실에서 매일 얼굴을 보면서 시간을 보냈던 여러 분들에게도 인사의 말을 전하고 싶습니다. 우선 연구실에서 오랜 시간을 같이 보내며 연구실

생활에 활력을 준 이제형 선배님, 홍성현, 정동헌, 오형석에게도 고마웠다고 말을 전하고 싶습니다. 또한 연구실에 처음 들어와서 많은 것을 가르쳐 주셨던 박진표 선배님을 비롯하여 여러 선배님들에게 많은 도움을 받았던 기억이 납니다. 또한 벤처창업이라는 경험과 추억을 같이 쌓았던 양병선, 이준표, 이승일, 이흥복 선배님들 그리고 동기 하영에게도 덕분에 좋은 경험을 할 수 있었다는 말을 하고 싶습니다. 그 외에도 연구실에서 수학하며 서로를 알게 된 정홍집, 이상규, 문민수, 김정래, 유준민, 최선일, 배성환, 박종국, 김진철, 김성무 등도 기억에 남습니다. 마지막으로 최근 알게 된 성원, 원기, 진석, 혁우, 지환, 진우 등 후배들에게도 덕분에 연구실 생활이 즐거웠다고 전하고 싶습니다. 모두 하나하나 언급하지 못하지만 덕분에 좋은 추억을 가지고 졸업한다고 전하고 싶습니다.

공학박사학위논문

가상머신의 메모리 관리 최적화

Optimizing Memory Management in Virtual Machine

2014 년 2 월

서울대학교 대학원

전기 컴퓨터 공학부

최 형 규

공학박사학위논문

가상머신의 메모리 관리 최적화

Optimizing Memory Management in Virtual Machine

2014 년 2 월

서울대학교 대학원

전기 컴퓨터 공학부

최 형 규

가상머신의 메모리 관리 최적화

Optimizing Memory Management in Virtual Machine

지도교수 문 수 묵

이 논문을 공학박사 학위논문으로 제출함

2013 년 12 월

서울대학교 대학원

전기 컴퓨터 공학부

최 형 규

최형규의 공학박사 학위논문을 인준함

2013 년 12 월

위 원 장 백 윤 흥 (인)

부위원장 문 수 묵 (인)

위 원 이 혁 재 (인)

위 원 이 재 진 (인)

위 원 김 수 현 (인)

Abstract

Optimizing Memory Management in Virtual Machine

Hyung-Kyu Choi
School of Electrical Engineering and Computer Science
The Graduate School
Seoul National University

Memory management is one of key components in virtual machine and also affects overall performance of virtual machine itself. Modern programming languages for virtual machine use dynamic memory allocation and objects are allocated dynamically to heap at a higher rate, such as Java. These allocated objects are reclaimed later when objects are not used anymore to secure free room in the heap for future objects allocation. Many virtual machines adopt garbage collection technique to reclaim dead objects in the heap. The heap can be also expanded itself to allocate more objects instead. Therefore overall performance of memory management is determined by object allocation technique, garbage collection and heap management technique. In this paper, three optimizing techniques are proposed to improve overall performance of memory management in virtual machine. First, a lazy-worst-fit object allocator is suggested to allocate small objects with little overhead in virtual machine which has a garbage collector. Then a biased allocator is proposed to improve the performance of garbage collector itself by reducing extra overhead of garbage collector. Finally an ahead-of-time heap expansion technique is suggested to improve user responsiveness as well as overall performance of memory management by suppressing invocation of garbage collection. Proposed optimizations

are evaluated in various devices including desktop, embedded and mobile, with different virtual machines including Java virtual machine for Java runtime and Dalvik virtual machine for Android platform. A lazy-worst-fit allocator outperform other allocators including first-fit and lazy-first-fit allocator and shows good fragmentation as low as first-fit allocator which is known to have the lowest fragmentation. A biased allocator reduces 4.1% of pause time caused by garbage collections in average. Ahead-of-time heap expansion reduces both number of garbage collections and total pause time of garbage collections. Pause time of GC reduced up to 31% in default applications of Android platform.

Keywords: optimization, object allocation, garbage collection, heap management, virtual machine, memory management

Student Number: 2002-30447

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent decades, virtual machine are becoming more common and widely adopted in various environment from embedded to server environment and most of modern computing devices support virtual execution environment. Java virtual machine [1] is one of popular virtual machines and available for various computing devices including low-end smartcard, digital TV, computer and high performance enterprise server. Dalvik is another well known virtual machine recent years. Dalvik virtual machine [2] is a core execution engine of Android [3] operating system for mobile devices including smartphone and table computers. By the year 2013, over 900 million Android devices have been activated worldwide [4] and most of web servers employ Java virtual machine to support Java language for server programming. And more virtual machines are employed in consumer appliances, such as digital TV [5], to provide user interactive services for individual users and service providers. Therefore virtual machine is very common nowadays and most of users who use smartcards, smartphones, computers, televisions or any kind of computing devices are already using some

virtual machines directly or indirectly.

Although most of users do not recognize the presence of virtual machine, user experience on those computing devices is affected by virtual machine, because overall performance of the device is determined by virtual machine when virtual machine plays a essential role in running applications. Therefore performance of virtual machine is a very important aspect as well as functions that virtual machine provides.

1.1 The need of optimizing memory management

Memory management module is one of key components in virtual machine and affects overall performance of virtual machine. Modern programing languages for virtual machine use dynamic memory allocation and objects are allocated dynamically to heap at a higher rate, such as Java. These allocated objects are reclaimed later when objects are not used anymore to secure free room in the heap for future objects allocation and many virtual machines adopt garbage collection technique to reclaim dead objects in the heap. Instead the heap can be expanded itself to allocate more objects and the heap itself is also allocated from memory. Since all memory management discussed above occur at runtime, efficiency of memory management affects the performance of virtual machine directly.

Overall performance of memory management is basically determined by object allocation technique, garbage collection and heap management technique. However each memory management technique is intricately related with each other and it is very hard to predict combined performance of memory management. Even worse memory management itself is also affected by other components of virtual machine as well as underlying hardware such as memory

hierarchy including caches. Behavior of applications also affects performance of a specific memory management technique and we cannot always avoid worst case situation unless we can predict future behavior of applications. Therefore there is always a need for optimizing memory management to improve performance of virtual machine as environments change, including hardware, behavior of applications, virtual machine and etc. This paper will discuss memory management in widely used environments and will propose optimizations to improve memory management in real devices.

1.2 Outline of the Dissertation

The rest of this thesis is organized as follows. Virtual machine and memory management are described in detail and problems with existing memory management in virtual machine are discussed and defined in chapter 2. Three optimizing techniques are introduced to enhance overall performance of memory management in virtual machine. Chapter 3 addresses a fast and efficient object allocator and proposes a lazy worst fit allocator with evaluation. After addressing the overhead of generational garbage collector, biased allocator is introduced to relieve the overhead in chapter 4. In chapter 5, ahead-of-time heap expansion technique is proposed and evaluated to improve overall performance of memory management through carefully selected but aggressive heap management. Then I summarizes proposed techniques with conclusions and discusses future works in chapter 6

Chapter 2

Backgrounds

2.1 Virtual Machine

A virtual machine is a software program that implements a machine and is capable of running software programs. There are known to be two kinds virtual machine including system virtual machine and process virtual machine. [6] In this paper, I'm going to deal with only process virtual machine and I will use term *virtual machine* to refer process virtual machine. This kind of virtual machine provides a platform-independent programming environment by abstracting underlying hardware. Therefore programs written for the virtual machine can be ran on any devices where the virtual machine is available.

There are a variety of process virtual machine available but two of virtual machine are going to be described in this section to provide short backgrounds for the remaining of this paper. One is famous Java virtual machine (JVM) [1] and another is Dalvik [2] virtual machine in Android [3] platform. Although two virtual machine are totally different virtual machine, both virtual machine have

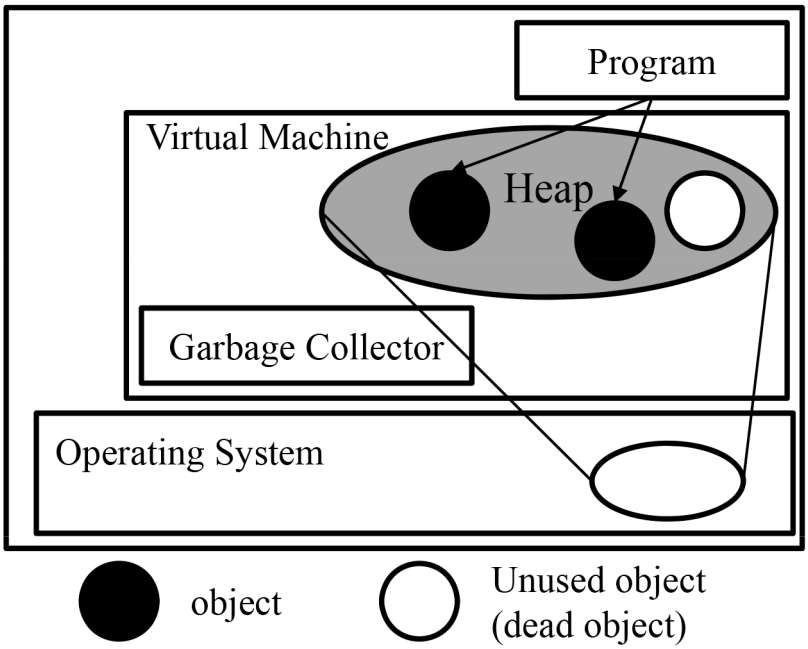

Figure 2.1 Virtual machine, heap and objects

some similarities in memory management, because applications are written in same language, i.e. Java language [7]. In the following section, we are going to describe memory management in virtual machine.

2.2 Memory management in virtual machine

Java is a class-based object-oriented programming language which allocates objects frequently. Java also adopts an automatic memory management technique called garbage collection [8] to reclaim unused objects automatically. Therefore a program written in Java allocates objects frequently and those objects are reclaimed automatically when they are not used anymore.

Figure 2.1 depicts abstract view of memory management in virtual machine. Virtual machine maintains a large pool of memory called the heap. The heap

could be a part of whole memory maintained by operating system. A program run on virtual machine allocates objects from the heap and uses them. Unused objects remains in the heap until being freed by garbage collector. Those unused objects, i.e. dead objects, are reclaimed later by a garbage collector which is an essential part of virtual machine. The size of heap can be increased if there is no room for new objects requested by the program. Then the heap grows to satisfy allocation request of the program by allocating more memory from operating system. In vice versa, the heap can also shrink if there is sufficient unused room. In short, memory management in virtual machine can be classified into three operation including object allocation, garbage collection and heap resizing. We will propose optimizing approaches for each operation in following sections.

Chapter 3

Lazy Worst Fit Allocator

3.1 Introduction

Modern programming languages use dynamic memory allocation [9]. As applications become more complex and use more of an object-oriented programming style, memory objects are allocated dynamically at a higher rate. This requires fast dynamic memory allocation.

Memory allocation should also be space efficient. A request for memory allocation cannot be satisfied when there is no free memory chunk that can accommodate the requested memory. This may happen even when the total amount of unused memory is larger than the amount of memory requested, due to fragmentation. In fact, fragmentation is the single most important reason for the wastage of memory in an explicitly managed heap or a heap managed by a non-moving garbage collector.

There are many approaches to implementing memory allocators, which exhibit different degrees of fragmentation and different allocation speeds. A com-

mon approach is maintaining a linked list of free memory chunks, called the free list, and searching the free list for a chunk that can satisfy a memory allocation request based on the fitting policy, such as first fit (FF), best fit or worst fit. Memory allocation using FF and best fit tends to have relatively low fragmentation [9], yet searching the free list has a worst-case linear time complexity.

In garbage-collected systems there are compacting garbage collection techniques such as copying collection [10] or mark-and-compact collection [11]. In such systems used and unused memory are not interleaved, so fragmentation does not exist. Thus, the obvious and fastest way to allocate memory is by simply incrementing an allocation pointer for each allocation.

There is a memory allocation approach for the free lists, motivated by the fast memory allocation of compacting collection, such that pointer increment is used as the primary allocation method, with FF, best fit or even worst fit as the backup allocation method [12]. This approach was called lazy fit, in the sense that finding a fitting memory chunk is delayed until really necessary. Preliminary experimental results simulating the traces of memory requests showed that the approach is promising since most memory allocations can be done via pointer increments.

This paper attempts to confirm the practical usefulness of lazy fits in the context of Java. We propose lazy worst fit (LWF) as a memory allocation method for a Java virtual machine with non-moving garbage collection. We implement LWF on a working Java virtual machine and evaluate its allocation speed and fragmentation, compared with lazy first fit (LFF) and FF. This chapter is organized as follows. Section 3.2 discusses memory allocation using conventional fits. Section 3.3 reviews memory allocation using lazy fits and proposes the LWF for Java. Section 3.4 presents our experimental results. Finally,

the paper is summarized in Section 3.5.

3.2 Allocation with fits

Before discussing memory allocation using lazy fits, we first discuss memory allocation using conventional fits.

In the simplest implementation of conventional fits, a single free list of free memory chunks is maintained. When a request for allocating memory is made, an appropriate free memory chunk is found from the free list after traversing the list from the head free chunk. The exact manner in which an appropriate free memory chunk is found depends on the fitting policy.

With first fit, the free list is searched sequentially and the first free memory chunk found that is able to satisfy the memory allocation request is used. This can be further divided into several types according to the order in which the free list is sorted: address-ordered, last-in-first-out (LIFO) and first-in-first-out (FIFO).

The address-ordered FF is known to have the least fragmentation, with the LIFO FF being noticeably worse. There is evidence that the FIFO FF has as little fragmentation as the address-ordered FF [13].

With best fit, the free memory chunk with the smallest size that is able to satisfy the memory allocation request is used. Along with FF, this policy is known to have little fragmentation in real programs.

In worst fit, the largest free memory chunk is used to satisfy the memory allocation request on the contrary to best fit. This policy alone is known to have much worse fragmentation than FF or best fit, so it is rarely used in actual memory allocators. However, worst fit can be useful when combined with lazy fit, which is explained in the next section.

The approach of using a single free list to keep track of the free memory chunks is very slow owing to a worst-case linear time complexity, especially if best fit or worst fit is done. So in actual implementations of modern memory allocators, more scalable implementations, such as segregated free lists, Cartesian trees and splay trees [9], are used for memory allocation.

Segregated free lists are the most common and simplest approach used in actual implementations [14, 8]. It divides memory allocation request sizes into size classes and maintains separate free lists containing free memory chunks in the size class. This approach, also called segregated fits, still has a worst-case linear time complexity, yet its allocation cost is known to be not much higher than that of a copying collector [8]. However, in our experiments unacceptably long search times for the segregated free lists do occur in practice (see Section 3.4), which indicates that the linear time complexity for accessing the free lists can be a real obstacle to fast allocation with fits.

3.3 Lazy fits

Memory allocation using lazy fit uses pointer increments[1] as the primary allocation method and conventional fits as the backup allocation method.

To be precise, an allocation pointer and a bound pointer are maintained for a current free space area. When a memory allocation request is made, the allocation pointer is incremented and it is checked against the bound pointer to see whether the memory allocation request can be satisfied. If it is satisfied, the memory that was pointed out by the allocation pointer before it was incremented is returned. Otherwise, conventional fit allocation is used to obtain a free memory chunk to be used as the new free space area, and the remainder of

[1]Pointer decrements can also be used for implementing lazy fits, but we assume pointer increments in this paper.

the former free space area is returned to the free list. The new free space area would then be used for allocating objects with pointer increments.

This is rather similar to the typical allocation algorithm used in systems with compacting garbage collectors, which also use pointer increments to allocate memory. The latter avoids a backup allocation method because there is no fragmentation, because compacting garbage collectors leave only one free chunk after compaction.

The fit method used for the backup allocation does not have to be any particular one. It could be first fit, best fit or even worst fit. These will be called lazy first fit (LFF), lazy best fit and lazy worst fit (LWF) respectively. In fact, it does not matter which approach is used for the backup allocation method as long as it is able to handle fit allocation. Using first fit or best fit would probably have the advantage of less fragmentation, while using worst fit would probably result in larger free space areas, which would result in more memory allocations using pointer increments for faster speed.

Figure 3.1 shows a simple example of how a lazy address ordered first fit would work.[2] Figure 3.1a shows the initial state when the LFF allocator starts allocating in a new free space area. The allocation and bound pointers point to the start and the end of the free space area respectively.

Allocation occurs within the given free space area, as in Figure 3.1b, incrementing the allocation pointer appropriately to accommodate each memory allocation request. This goes on until the free space area is no longer able to satisfy the memory allocation request, i.e. the space remaining in the free space area is smaller than that needed by the caller. Then, we put what remains of the current free space area back into the free list and search the free list for

[2]However, we used a segregated FF instead of an address-ordered FF in the experiment, because the address-ordered FF is very slow. The only difference is how to manage the free list.

a new free space area which can be used to allocate memory. The allocation and bound pointers are set to the start and the end of the new free space area respectively, and the cycle begins anew.

Figure 3.1c shows the state of the heap after the old free space area, marked as 'old', is put back into the free list, and the allocation and bound pointers point to the boundaries of the new free space area, marked as 'new', which had just been extracted from the free list using FF.

To speed up memory allocation using a lazy fit even more, the allocation and bound pointers could be held in two reserved global registers. This allows one to allocate memory without touching any other part of the memory, except for the memory we are allocating, in the common case. This is in contrast to many other allocation algorithms which usually require at least some manipulation of the data structure in the memory.

Lazy fit also has the potential to be faster than segregated storage since it

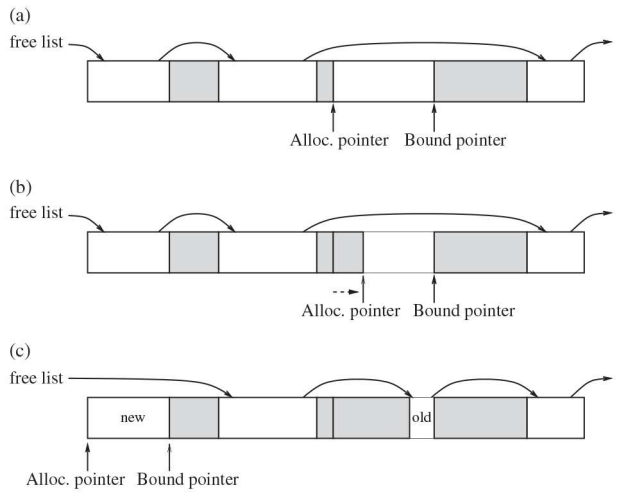

Figure 3.1 An example of a lazy address-ordered first fit (shaded areas denote used memory). (a) Initial state; (b) allocation using pointer increments; (c) the state after the new free space area was found by first fit.

has no need to decide size classes. Objects allocated closely together in time would probably be used together, so there could also be a beneficial effect on cache performance, since lazy fit would tend to group together objects that are consecutively allocated.

3.3.1 Lazy worst fit

In order to use lazy fit for garbage-collected systems such as Java, we made two engineering choices. First, we propose using worst fit in order to reduce the search time for the free lists. So, after each garbage collection we sort the free memory chunks in the free list in decreasing order of sizes. By using worst fit, a single comparison suffices to find out whether there is a chunk in the sorted free list which is able to accommodate the requested object, whereas alternative methods such as FF or best fit may require many comparisons to ascertain whether such a chunk exists.

Second, the previous free space area which had been unable to accommodate the requested object is discarded and not put back into the free list when using lazy worst fit. One reason is that inserting it into a sorted free list would introduce $O(n)$ time complexity [15] when we use a simple singly linked list, while all operations in LWF, including pointer increments and worst fits, can be done in $O(1)$ time. Giving up the previous free area will keep the O(1) allocation speed, which would obviate the worst-case linear time complexity of accessing the free list. Since the free list is constructed from scratch during garbage collection, there is no problem in discarding the old free space.

LWF is also expected to be faster by having more pointer incrementing memory allocation, since we can get larger free space areas, yet this would depend on the pattern of memory requests. On the other hand, LWF may result in more fragmentation and wastage of the discarded free spaces, which

might lead to larger heap sizes and more garbage collection cycles. All of these will be evaluated through experiments in the following Section 3.4.

3.4 Experimental results

Lazy fits are evaluated by generating traces of memory requests for a set of C programs and measuring the fragmentation and fit frequencies for both explicitly managed heaps and garbage collected heaps in previous work [12]. In this paper, we implemented lazy fits on a working Java virtual machine and evaluate whole Java system using non-trivial Java programs.

3.4.1 LWF implementation in the LaTTe Java virtual machine

A memory system using LWF was implemented on LaTTe, a freely available Java virtual machine with a just-in-time (JIT) compiler [16]. This subsection describes the implementation of lazy fits in LaTTe and outlines the LaTTe memory management system, which will be helpful in understanding the experimental results.

LaTTe manages a small object area and a large object area separately, and LWF is done only on the small object area which contains objects that are smaller than a kilobyte. One of the reasons for the separation is that sharing the same heap among large and small objects may result in high fragmentation, as the experimental results in [12] indicate. Large objects are allocated in separated area and best fit is used when allocating them.

LaTTe uses a partially mark and sweep garbage collector, in the sense that the runtime stack is scanned conservatively for pointers while all objects located in the heap are handled in a type accurate manner [17]. Pointers should be handled conservatively, since there is no accurate information for pointers in stacks. It is also possible to provide accurate type information to garbage

collector, but it requires more computation and memory space to maintain information at runtime and degrades overall performance. The separation of the small object area and the large object area also helps the garbage collector identify pointers more easily and efficiently, since we can make use of the fact that memory is separated when handling pointers.

LaTTe starts with an initial heap pool of 8 MB. Both the small object area and the large object area are allocated from this heap pool in units of 2 MB. If there is no memory available in the pool, LaTTe activates the garbage collection thread to reclaim unused memory.

After each garbage collection, LaTTe may decide to expand the heap depending on its capacity. The idea is that if the heap is too small, the garbage collection frequency would be unacceptably high. On the other hand, LaTTe does not expand the heap unnecessarily, since other applications will run out of memory in a multiprogramming environment.

LaTTe expands the heap only when the size of free memory is less than the size of live objects, meaning that the heap is less than half empty. Here, the free memory is estimated by the cumulative size of objects allocated between the previous garbage collection and the current garbage collection, instead of the heap size minus the size of live objects, which cannot be considered to be entirely free owing to fragmentation. So, LaTTe expands the heap only when the size of live objects exceeds the size of objects allocated, the expanded amount being the difference between these two quantities (rounded off to 2 MB). This appears to be a good compromise between the conflicting goals of keeping the size of the heap small and keeping the garbage collection frequency to a reasonable level.

Table 3.1 Benchmarks

Benchmarks	Description
_202_jess	A Java version of NASA's CLIPS expert shell system
_209_db	Data management software which performs multiple database functions on memory resident database
_213_javac	Java compiler from the JDK 1.0.2
_227_mtrt	Dual-threaded ray tracer that render the scene in the input file
_228_jack	A Java parser generator
EulerBench	Computational Fluid Dynamics
MonteCarloBench	Monte Carlo simulation
RayTracerBench	3D Ray Tracer
SearchBench	Alpha-beta pruned search

3.4.2 Experimental environment

We ran the experiments on a Sun Blade 1000 machine with a UltraSPARC-III microprocessor 750MHz with a 32 KB instruction cache and a 64 KB data cache. It also has an 8 MB second-level cache and a 1 GB memory.

Our benchmarks are composed of nine selected benchmark applications from the SPECjvm98 suites and section 3 of the Java Grande benchmark suites Version 2.0, which are listed in Table 3.1. The first five benchmarks of the table are selected from SPECjvm98 and remaining four benchmarks are chosen from Java Grande. We excluded those benchmarks that do not allocate enough small objects from the suites, such as _200_check, _201_compress and _222_mpegaudio in SPECjvm98, and MolDynBench in the Java Grande benchmarks. We used size A inputs for the Java Grande benchmarks during the experiments.

Table 3.2 Running Time Analysis

Benchmark	Total running time (seconds)			Allocation time (seconds)		
	LWF	LFF	FF	LWF	LFF	FF
_202_jess	12.359	15.352	212.641	1.676	4.520	198.466
_209_db	18.235	18.652	35.507	0.629	1.021	18.038
_213_javac	19.162	995.678	7855.287	1.779	972.019	7818.269
_227_mtrt	11.093	22.844	2846.460	1.326	11.996	2826.267
_228_jack	13.889	16.950	354.650	1.305	3.764	338.057
EulerBench	33.347	972.437	22600.980	1.198	937.126	22523.039
MonteCarloBench	107.530	118.061	132.451	0.106	10.404	24.163
RayTracerBench	21.378	21.882	74.493	1.059	1.471	54.213
SearchBench	19.740	20.506	106.550	1.383	2.178	88.089

3.4.3 Performance of LWF

We experimented with three different memory allocation policies: LWF, LFF and FF. The first fit algorithm used in LFF and FF uses segregated free lists segregated by a power of two distribution [18], with objects maintained in the FIFO order. This segregated free list is believed to reduce the allocation time compared with the traditional FF and may have less fragmentation [9].

LFF works in exactly the same way as LWF except that FF is used when the pointer-incrementing allocation fails. And the remainder of the previous free space is discarded as in LWF. Unlike LWF and LFF, FF always returns the remainder of the free space to the free lists.

By comparing LWF and LFF we can evaluate the impact of LWF's $O(1)$ access time for the free lists and how worst fit and first fit affect fragmentation in the context of lazy fits. By comparing LWF or LFF with FF, we can evaluate the impact of pointer-incrementing allocation of lazy fits.

For each benchmark, Table 3.2 shows the total running time and allocation time of each policy. The results indicate that LWF is always better than LFF, and LFF is always better than FF. In fact, there are several benchmarks

Table 3.3 Allocation Time Analysis

Benchmark	Allocation time (seconds)					
	Fit allocation			Other		
	LWF	LFF	FF	LWF	LFF	FF
_202_jess	0.164	2.974	195.117	1.512	1.546	3.349
_209_db	0.075	0.462	16.736	0.554	0.559	1.302
_213_javac	0.505	970.099	7814.567	1.274	1.920	3.702
_227_mtrt	0.153	10.805	2823.008	1.173	1.191	3.259
_228_jack	0.089	2.527	335.049	1.216	1.237	3.008
EulerBench	0.461	935.616	22518.786	1.526	1.510	4.253
MonteCarloBench	0.038	10.312	24.020	0.068	0.092	0.143
RayTracerBench	0.036	0.447	51.819	1.023	1.024	2.394
SearchBench	0.048	0.854	85.088	1.335	1.324	3.001

which show excessively high improvement when using LWF. When compared to LFF, _213_javac and EulerBench show much shorter running time with LWF. _202_jess, _213_javac, _227_mtrt and EulerBench also have been drastically improved when compared to FF.

In order to check whether the allocation policy really affects the running time, we measured the total memory allocation time for the small object area separately, which is shown in the second column of Table 3.2. The allocation time results are consistent with the running time results such that longer allocation time means longer running time.

The allocation time of each policy includes the time spent for fit allocation using worst fit or FF, which was also measured separately as shown in Table 3.3. The second column of the table shows time spent of fit allocation to find a new free chunk. The third column, i.e. other, are remaining time of total allocation other than fit allocation and it includes pointer incrementing allocation time.

Next, we analyze why LFF and FF have longer fit allocation time than LWF. There are two major differences between LWF and other LFF/FF, that affect the fit allocation time. The first is the frequency of fit allocation. Generally,

Table 3.4 Frequency (%) of small memory allocation via fit policy

Benchmarks	LWF	LFF
_202_jess	6.731	7.611
_209_db	0.691	0.661
_213_javac	11.815	22.372
_227_mtrt	3.688	2.815
_228_jack	2.760	2.665
EulerBench	21.017	9.408
FMonteCarloBench	10.696	27.892
FRayTracerBench	1.048	1.021
SearchBench	1.467	1.450
Geomean	3.867	4.101

LWF is expected to allocate more often via pointer increments than via fits, since worst fit would allow larger free spaces than LFF for pointer-incrementing allocation.

Table 3.4 shows the frequencies (%) of the fit allocation for LWF and LFF respectively. And the fit frequency for FF is obviously 100% which is not shown in the table. For _213_javac and MonteCarloBench, LFF has a much higher fit frequency than LWF. On the other hand, LWF has a much higher fit frequency than LFF in EulerBench. Therefore, contrary to our expectation we cannot see any definite relationship between the fit frequency and the fit policy.

Another major difference between LWF and LFF/FF is that the search time of the free lists for the fit allocation is $O(1)$ for LWF and $O(n)$ for LFF/FF. In order to check whether this difference really affects the fit allocation time, we measured the total number of 'link' operations, i.e. the operation to follow a single link in the free lists, for each policy.

Table 3.5 shows the number of link operations for LWF, LFF and FF respectively. It shows that LFF and FF execute many more link operations than LWF. Even for _227_mtrt and EulerBench where LFF has lower fit frequencies

Table 3.5 Comparision of 'link' operations

	Total number of operation (thousands)			Ratio	
Benchmarks	LWF	LFF	FF	LFF/LWF	FF/LWF
_202_jess	527.5	84,597.5	3,784,346.7	160.4	7173.5
_209_db	2.8	5,473.2	283,373,982.0	1955.4	101241.2
_213_javac	682.0	7,223,197.8	45,998,050.8	10590.8	67443.5
_227_mtrt	207.2	241,971.1	27,300,486.8	1167.7	131742.6
_228_jack	169.1	58,787.2	5,840,161.2	347.7	34545.7
EulerBench	1,361.4	10,469,150.4	194,860,155.4	7689.8	143128.1
MonteCarloBench	30.2	216,744.8	592,767.2	7184.6	19648.9
RayTracerBench	11.5	332.5	474,167.1	28.8	41103.3
SearchBench	10.1	589.8	1,937,796.8	58.2	191274.0

than LWF, LFF has a higher number of link operations than LWF. Since FF always uses fit allocations, it obviously executes more link operations than LFF. These results are consistent with the fit allocation time in Table 3.3, especially for those that have an excessively long fit allocation time. So, it is evident that the $O(n)$ search time is the dominant reason for the longer running time in LFF and FF. In fact, it can be seen that the linear time complexity of conventional fit allocation may cause an unacceptably high overhead, even with segregated implementations.

3.4.4 Fragmentation of LWF

Another important aspect of a memory allocator is fragmentation. It is generally believed that higher fragmentation requires larger heaps and causes more garbage collection, which may affect performance.

It is expected that LWF causes worse fragmentation than LFF and FF since worst fit is known to be poorer than FF in terms of fragmentation. Also, LWF and LFF have a disadvantage in fragmentation when compared to FF, since they discard the remainder of the previous free space area. Contrary to these expectations, our performance results in Table 3.2 indicate that the overall

Table 3.6 Average fragmentation ratio (%)

Benchmarks	LWF	LFF	FF
_202_jess	3.694	3.222	1.608
_209_db	0.240	0.264	0.252
_213_javac	14.203	5.400	5.427
_227_mtrt	2.092	2.231	2.713
_228_jack	2.454	2.504	1.097
EulerBench	0.395	4.513	8.326
MonteCarloBench	12.482	1.170	0.349
RayTracerBench	0.394	0.398	0.489
SearchBench	0.059	0.068	1.423
Geomean	1.249	1.155	1.332

performance of LWF is still better than LFF and FF, so these results need to be verified.

Tables 3.6 and 3.7 show the fragmentation ratio for the small object area for each policy. The fragmentation ratio was measured as follows. Whenever the memory allocator cannot satisfy a request for the small object area (so either 2 MB is allocated from the heap pool or garbage collection is invoked if the heap pool is empty), we measure the fragmentation ratio at that point.

Table 3.6 indicates that the average fragmentation ratio of LWF is not always higher than that of LFF and FF. In fact, we cannot see any definite correlation. For those benchmarks where the average fragmentation ratio of LWF is noticeably higher, such as _213_javac or MonteCarloBench, we found that the sequence of memory requests for the small object area occasionally includes requests for a relatively large object, e.g. > 100 bytes.

The problem with LWF is that larger free areas are consumed at the beginning of the allocation, such that by the time these large object requests arrive, their chance of being allocated in the current free space is lower, leading to the current free space being discarded, although it can still accommodate more

Table 3.7 Worst-case fragmentation ratio (%)

Benchmarks	LWF	LFF	FF
_202_jess	4.70	4.14	1.87
_209_db	0.38	0.38	0.27
_213_javac	47.21	7.64	8.27
_227_mtrt	21.85	21.49	21.35
_228_jack	6.03	5.47	1.92
EulerBench	2.17	6.56	12.18
MonteCarloBench	35.11	1.33	0.38
RayTracerBench	0.42	0.42	0.51
SearchBench	0.06	0.08	1.46

Table 3.8 Garbage collection data and size of small object area

	Garbage collection						Size of small area		
	Time (sec)			Count			(bytes)		
Benchmarks	LWF	LFF	FF	LWF	LFF	FF	LWF	LSFF	SFF
_202_jess	1.041	1.028	1.122	67	66	65	6,291,456	6,291,456	6,291,456
_209_db	0.865	0.855	0.856	8	8	8	20,971,520	20,971,520	20,971,520
_213_javac	4.042	3.910	3.901	16	16	17	44,040,192	35,651,584	37,748,736
_227_mtrt	1.817	1.800	1.853	18	18	18	18,874,368	18,874,368	18,874,368
_228_jack	0.458	0.452	0.468	42	42	42	6,291,456	6,291,456	6,291,456
EulerBench	3.074	2.749	3.176	43	38	41	14,680,064	16,777,216	16,777,216
MonteCarloBench	1.159	1.104	1.079	0	0	0	6,291,456	4,194,304	4,194,304
RayTracerBench	0.092	0.091	0.088	35	35	35	6,291,456	6,291,456	6,291,456
SearchBench	0.415	0.426	0.441	202	202	205	2,097,152	2,097,152	2,097,152
Geomean	0.906	0.884	0.911						

small objects.

On the other hand, LFF would have a relatively better chance of allocating the large object in the current free space. This would make LWF suffer more from fragmentation than LFF. Such cases may result in very high fragmentation for LWF as shown in Table 3.7 where the worst-case fragmentation ratio is measured.

In order to check the impact of fragmentation, we measured the garbage collection frequency, garbage collection time and the total size of the small object area, as shown in Table 3.8. The table shows that there is little difference in garbage collection time and frequency of garbage collection among the three

policies, which would explain why fragmentation did not have a major effect on performance.

As to the size of the small object area, LWF uses larger areas than LFF for _213_javac and MonteCarloBench where LWF suffers more from fragmentation, whereas LFF uses larger areas than LWF for EulerBench where LFF suffers more from fragmentation. However, there is no tangible impact on garbage collection time or frequency depending on allocation methods, as discussed.

3.5 Summary

We propose the use of lazy worst fit for memory allocation in Java, which exploits pointer-incrementing memory allocation with free lists. LWF avoids the linear time complexity of managing the free lists that may cause an unacceptably high memory allocation overhead, and it does not suffer much from fragmentation. One interesting question is whether these benefits may even allow a non-moving garbage collector to compete with compacting collectors, while avoiding their drawbacks. For example, copying collection has some problems such as half-availability of the heap space, exponential performance degradation as the object residency[3] increases [8] or poor locality [11]. Mark-and-compact collection is also known to be expensive to implement since compaction requires more than just copying objects or updating pointers [8]. It is left as a future work to evaluate non-moving garbage collectors with LWFs, compared with compacting garbage collectors.

[3]The ratio of live objects to garbage objects at any given time.

Chapter 4

Biased Allocator

4.1 Introduction

Virtual machine adopts automatic memory management to manage the heap. Automatic memory management reclaims objects which are not used anymore automatically from the heap, although object allocation is requested explicitly by a program. Garbage collection is a famous approach to find unnecessary objects, i.e. dead object, and reclaim them [8]. Allocation strategies and garbage collection should be considerate each other, since garbage collector is responsible for securing and managing free space which is used by allocator later to allocate objects. We can consider garbage collector a producer of free space and then allocator can be a consumer of free space. Therefore some garbage collectors enforce allocation methods considering fragmentation, performance and throughput. In vice versa, some allocation strategies are more efficient with specific garbage collectors. Various garbage collectors have been proposed by many researchers [19, 8, 20, 21, 22]. In detail, there have been different approaches

to find dead object and also there are various ways to secure free space. One of simplest way to find unnecessary objects is traversing pointers recursively from always live objects, which are called roots, to find reachable objects which are live and necessary. Another approach maintains counters for incoming references to each object at runtime to determine a liveness of object [23]. There are also several ways to secure free space after identifying dead objects. A simplest way maintains a list of free space by reclaiming dead objects. Another approach secures free space by moving live objects to different area, as a result previous area contains only dead objects and whole previous area can be considered to be free space. There are so many garbage collectors depending on how they identify dead objects and how they secure free space. Among them, a generational garbage collector is famous and widely adopted in virtual machines, e.g. Java Virtual Machine from Oracle [24].

A generational garbage collector manages the heap by splitting whole heap into several generations from young to older. With a generational garbage collector, a new object is always allocated from a nursery area which is one of generations and considered to contain young objects. Then later if a nursery is overpopulated and there is no room for new objects, a generational garbage collector secures free space from a nursery by moving live objects to older generations. We call this object copying a promotion, since an object is promoted to old generations. Such garbage collection on a nursery is called minor garbage collection. Later we have to reclaim all dead object in young and old generations too when old generations are also overpopulated. We call it a major garbage collection or a full garbage collection. Figure 4.1 depicts how a simple two generational garbage collector works. Due to various advantages, a generational GC is adopted in many virtual machines for various environments. First, a garbage collection can be completed in a short time when minor GC

is requested instead of full GC, because minor GC performs only on a nursery which is relatively smaller than whole heap. Although number of garbage collections increases relatively, each pause time caused by garbage collection is reduced and responsiveness of virtual machine is improved when compared to a garbage collector with only full GC. Furthermore secured free space from a nursery is continuous and fragmentation free since whole young generation is empty after all live objects are promoted. There are many variations of generational garbage collector depending on number of generations, size of young and old generations and etc. [19, 25, 21] However a generational garbage collection has unavoidable runtime overhead and it shows undesired behaviors in some cases.

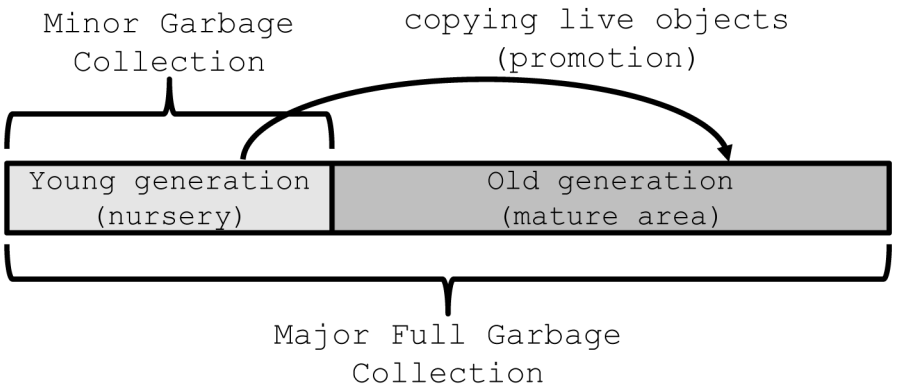

Figure 4.1 A generational garbage collector with two generations.

A generational garbage collector has to promote live objects to older generations to clear up a nursery. Each promotion contains not only copying an object but also updating pointers which refer to the object just moved to a new location. A generational garbage collector is beneficial when only few objects are live and most of objects in young generation are dead. However when many of objects in a nursery are live and is going to be promoted, the overhead

of promotion increases to hide advantages of a generational GC. In worst case when every object in a nursery is live, we have to promote all objects and minor GC does not reclaim dead object at all with overhead of minor GC and promotions. We suggest that such overhead can be avoided if we place promoted objects to old generations instead of young generation when those objects are being allocated at first. We are going to segregate objects in various ways to reduce number of objects allocated to a nursery. Rest of the paper is composed as follows. In the next Section 4.2, we address the problem in detail and propose an approach to exploit biased allocators to improve a generational garbage collector. Then we propose a way to invoke biased allocator and describe three analyses to identify objects to be allocated with biased allocators in Section 4.3. We describe how to combine proposed analyses and how we implemented proposed approaches in real environment in Section 4.4. In Section 4.5, proposed approaches are evaluated on a real embedded device. Section 4.6 summarizes the paper and discusses future works.

4.2 Motivation

As we discussed in the previous section, a generational garbage collector itself suffer from inherent overhead of promotion. As a result pause time of each garbage collection can be increased to compensate advantages of a generational GC. There have been many researches to improve a generational GC [8, 21, 22] , but most of them require modifying a garbage collector itself and a garbage collector is getting more complicated which is hard to predict the effect modification in various situation.

We propose an approach to exploit an allocation instead of a generational GC to overcome the undesired overhead of a generational GC. We already address that such an undesirable behavior of a generational GC is due to pro-

motion of many objects in a nursery. In other words, such objects live long to the time when minor GC is requested to reclaim dead objects. We are going to avoid the situation by simply locating such objects in old generations instead of a nursery when objects are allocated. Simply we can allocate all objects in old generations, but then it is not a generational GC anymore and may suffer from long pause time of full GC instead. Therefore we have to choose a set of objects and allocate them to old generation using biased allocators. In the following section, we propose a way to make use of biased allocators and describe how to identify objects to be biased in detail.

4.3 Biased allocator

With biased allocators, an object can be allocated to heap in different ways depending on various properties to improve the performance of heap management with a generational garbage collector. On the other hand, traditional virtual machine with a generational GC allocates an object to a nursery area of heap with a single same allocator. We propose that we affect the performance of heap management in a beneficial way by reducing copying overhead of generational garbage collection if we allocate an object to other than a nursery carefully with different object allocators. In this section, we will discuss when to choose an allocator and propose a way to make a decision with less runtime overhead. Then we will describe three analyses to select an allocator.

4.3.1 When to choose an allocator

We can choose an allocator every time when an object is being allocated to the heap. It would be best if we can perform fine-grain analysis for each object and decide allocation area for each object. However it is not easy to predict lifetime of each object precisely and there will be extra overhead if we choose

an allocator every time an object is being allocated. Usually an object allocation occurs very frequently and an additional computation could harm overall performance. Therefore it would beneficial to runtime performance if we can choose an allocator without extra overhead of an allocation itself.

A *new* bytecode in Java Virtual Machine always knows a type of an object to be allocated [1] and the *new* bytecode allocate objects of same type. Therefore we are going to exploit the property that each *new* bytecode always allocates isomorphic type of objects at runtime.

Also we try to reduce the overhead of decision making by making a decision once and use the same decision later. To achieve these, we choose an allocator when bytecode are analyzed and being translated into native machine code to improve overall performance. In other words, biased allocator can be applied to any kind of translators including just-in-time compiler (JITC), ahead-of-time compiler (AOTC) and install-time compiler (ITC). A Just-in-time compiler which translates bytecode into machine code at runtime is a famous acceleration technique [26, 16, 27, 28] . Ahead-of-time compiler [29, 30, 31] and install-time compiler [32] analyze and translate bytecode into native machine code before it is being executed.

A biased allocator is chosen when a *new* bytecode is being translated into machine code depending on the type of object to be allocated. Then the *new* bytecode is translated into a machine code which allocates an object with the selected allocator. In this way, we make a decision once and an allocation is done without additional overhead other than that the allocator allocates an object in a different way.

4.3.2 How to choose an allocator

Even though that a specific *new* bytecode accepts an isomorphic type, it is not easy to exploit the information to select an allocator wisely. We need whole type analysis on a Java program to make a correct decision and it is not eligible for a JITC or AOTC, because whole type analysis including class hierarchy analysis [33, 34] is not a simple problem and it takes much time. Therefore we consider three information as well as simple type information, i.e. class information which is known directly from the *new* bytecode itself.

First we identify a location where local-scoped objects are allocated. Also an allocation site within a loop is identified and being chosen to use a biased allocator. Finally we analyze the use of an allocated object which is assigned to static fields and identify locations where the object is allocated. Since an object can be allocated from multiple locations depending on control flow, we exploit traditional iterative data follow analysis. Of course, type information is always considered together with three properties.

Local-scoped objects

An object is known to be locally scoped if an object is live only within a specific scope. A scope can be anything such as a basic block, a super block, a trace, a method or even a program. There have been many researches to identify locally scoped objects and escape analysis is one of famous technique to identify locally scoped objects. Escape analysis has been used in Java to make use of stack allocation [35, 36] to relieve memory pressure on the heap and adopted in various JVM such as Java Standard Edition 6 [37]. We use an escape analysis to identify an allocation site where objects being allocated are locally scoped. We expect that such an allocation site can make use of traditional allocator or even stack allocator which uses a stack instead of the heap, because locally

scope objects are only live within a specific scope and liveness is limited to the scope which can be considered being relatively shorter than other objects which escape the scope. As a result, we don't have to consider such allocation sites for being a candidate for biased allocation to improve the performance of heap management with garbage collection.

Objects allocated inside loops

Loops have been a famous target for an optimization, because many programs spends most of the time in loops and small improvement in a loop can be result in large runtime improvement of the performance due to its repetition. We also look into loops, because an allocation in loops will continue allocate same type of objects until loop stops and quite large amount of objects are allocated inside of the loops.

We expect that objects allocate inside loops are relatively short lived compared to objects allocated outside of loops, because loops usually perform same computation repetitively and many objects allocated within loops are for temporary use. We decide objects allocated inside loop to be possibly short-lived at first. However we find that some objects, which are allocated in a loop but have relatively small size, are long-lived. Therefore allocation sites within loops are chosen when smaller objects are allocated. We can easily compute the size of objects, because the type of object being allocated is identified directly from *new* bytecode as we described before. We don't have to worry about leaving large objects behind in young area, because promotion overhead is more dominated by number of objects being promoted than size of objects as we discussed in previous sections.

Objects assigned to static fields

There are two types of objects in Java, i.e. an instance object and a class object. An instance object is an instance of a specific class which are usually allocated with *new* keyword of Java language and object we talked before in this paper are all instance objects. A class object is a unique object of a specific class and they are usually created implicitly by Java virtual machine when the class is being resolved. A static field is a field not related to an instance object but class object itself. Since a static field looks like a global variable, researches have shown that an object assigned to a static field tend to be immortal, i.e. never dead till the program ends [38].

We decide to make use of this property and use biased allocators for such allocation sites where any object allocated can be assigned to static fields. We make use of traditional analysis of reaching definition to identify allocation sites on the compilation unit. Candidate allocations sites can be one or more and even we can't find a site, because we perform analyses only within the compilation unit.

Of course, some candidate allocation sites can be duplicated with the previous analysis, i.e. allocation sites within loop. We will discuss how we arrange three analyses we discussed here to make a decision for biased allocation in the following section.

4.4 Analyses and implementation

Each allocation site can have three properties, i.e. local, loop and static. Local means this allocation site allocates objects which are live only within the scope. Loop means this allocation site is located within loops and size of allocation is larger than threshold. Static means this allocation site allocate objects which

can be possibly assigned to static fields. Only allocation sites which is neither local nor loop are selected for biased allocation. Then we find allocation sites with static property and add them to candidates and we are going to describe how we make use of three analyses.

At first, we assume that all allocation sites are candidate for biased allocation. We find locally scoped object with escape analysis. After we identify allocation sites which only allocate locally scoped objects, we remove those sites from candidates. We do not discard the list of allocation sites that are local and keep the list for later use.

We continue to identify allocation sites within loop and this analysis can be done with other traditional loop optimizations as well. However this analysis should be done after any control flow changes or code motions are made to loop, because the location of an allocation site can be changed with those optimizations and even allocation sites can be eliminated after optimizations. Furthermore we do not analyze and skip allocation sites which are already identified to allocate only locally scoped objects from previous escape analysis. Then we reduce candidate allocation sites with results of loop analysis. We find out that some allocation sites within loop allocate only locally scoped objects and it is obvious that these objects have relatively shorter lifetime than other objects which escape the same scope.

Finally we look into every assignment of an object to static fields and try to identify one or more allocation sites where the object was allocated. This analysis should be done just before the code generation, because any control and data flow changes can affect the result of this analysis. After we identify allocation sites, we add those allocation sites to candidates for biased allocation. In short, we can formulate above sequences as in Figure 4.2. We should keep the order of local, loop and static analysis, because there can be an allocation site

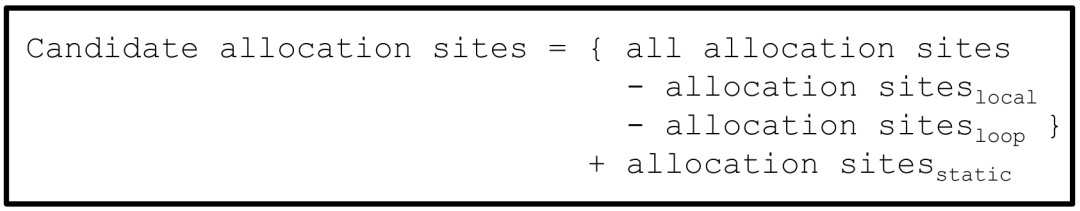

Figure 4.2 Candidate selection with three analyses

which reside in loop and allocate large objects, but allocates objects which can be assigned to static fields. Of course there is no allocation site which allocates locally scoped object and allocates objects assigned to static fields, because an object is not locally scoped if there is any assignments of an object to static fields.

We implemented these analyses on Oracle's phoneME Advanced MR2 version. This phoneME advanced MR2 is Java virtual machine for embedded devices and can run Java applications via interpreter and just-in-time compiler (JITC). Ahead-of-time compiler (AOTC) [29, 30, 31] is also available for translating Java bytecode to native machine code with optimizations where proposed approaches had been inserted.[1] Our analyses were also done within a method scope, since a translation unit of the AOTC is a method. Figure 4.3 depicts a implementation of biased allocator in virtual machine with AOTC. Analyses are implemented in AOTC and we generate hints at static time as shown in the figure. A biased allocator itself is available in virtual macine and allocates objects regarding hints at runtime.

[1] As we mentioned before, analyses can be implemented in any translator which translates code, such as JITC, AOTC and ITC.

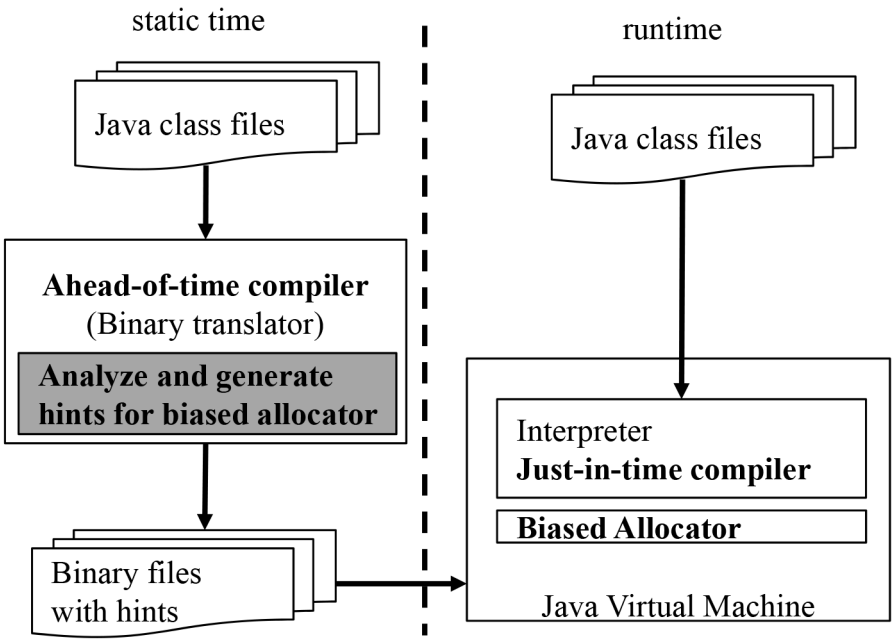

Figure 4.3 Implementation of biased allocator

4.5 Evaluation

We evaluate our proposed analyses on phoneME Advanced MR2 [39] with digital TV (DTV)[5] set-top box which includes MIPS based core with 128MB main memory. This software platform in digital TV supports advanced common application platform (ACAP) middleware and is running on the Linux with kernel 2.6.12.

We make use of AOTC to perform proposed optimization and observed the effect of biased allocation without runtime overhead of analyses. Java applications have been translated by AOTC before running and stored in set-top box for evaluation. We use six micro benchmarks from specjvm98 [40] to evaluate our approaches. We choose a generic generational garbage collector with two generations in phoneME Advanced MR2 to reclaim objects while running

specjvm98. Since total pause time due to garbage collections is relatively small compared to total running time, we compared total pause time separately instead of total running time and measured the amount of promotions occurred in generation garbage collections.

4.5.1 Total pause time of garbage collections

We measured total pause time of garbage collections before and after applying proposed approaches and compared them in Figure 4.4. About up to 12.2% of total pause time caused by garbage collection has been reduced and about 4.1% of pause time is removed in average. Figure 4.5 depicts the size of biased objects compared to total size of objects allocated. We identify lots of objects from _209_db where pause time has been reduced most. However even we biased more than 10% of objects from _228_jack, total pause time is not reduced much as we expected compared to other programs and we can't find direct correlations between the size of biased objects and total pause time. After careful examination, we find out that total pause time of generational garbage collector is affected by various factors and it is very hard to predict. For example, size and number of objects allocated in nursery area affect pause time. When promotion occurs, more factors affect pause time of generational garbage collection, because a promotion includes copying an object and updating pointers to copied object. Even worse promotions may incur a full major garbage collection when there is no sufficient space in a mature area.

On the other hand, our approaches may consume a mature area more aggressively due to false detection. Three analyses we proposed are all based on static analysis without runtime information. Therefore we can't predict exact life time of objects and availability of the heap is not concerned at all. As a result proposed approach may induce side effects in unexpected ways due to

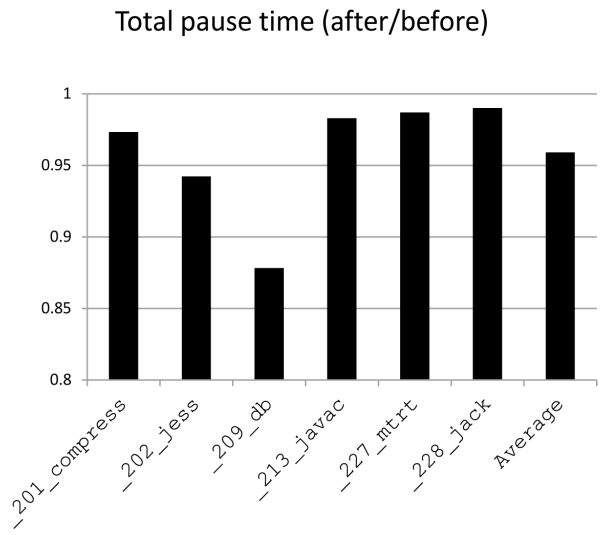

Figure 4.4 Ratio of total pause time after applying biased allocation compared to non-biased allocation

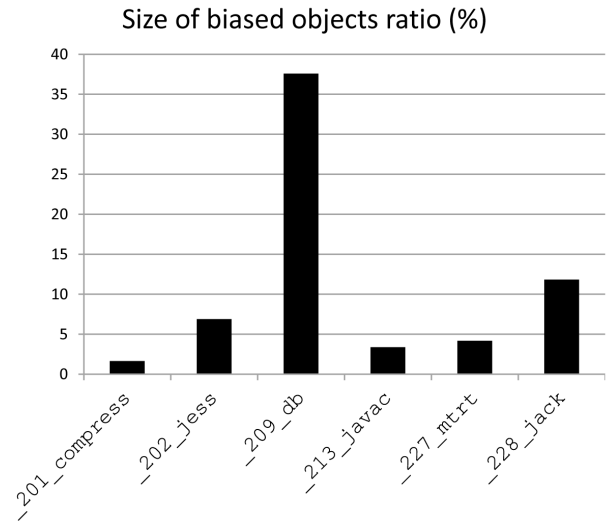

Figure 4.5 Ratio of biased objects size compared to total objects

exploiting a mature area much more than a nursery area. However it is not easy to calculate lifetime of objects exactly and our research is a start point to exploit different allocation based on analyses. We will discuss these matters in the last section again with future works.

4.5.2 Effect of each analysis

We also evaluated the effect of each proposed analysis in Figure 4.6. When we choose objects with an escape analysis, we can't reduce total pause time of garbage collections effectively. We found that total pause time has been reduced much after analyzing loops. Even though we decide to bias objects which are allocated to static fields towards old generation, Figure 4.6 shows that there is only a little improvement with this optimization. However it is expected, because objects assigned to static fields are rarely overwritten and few allocations are related to static fields. Of course, there are some allocation sites where few objects are assigned to static fields and other objects are discarded soon. A proposed analysis may decide those allocation sites to be candidate for biased allocation but those are not desirable choices, because we want to allocate objects that live long. Therefore those candidates can be false-positive. Nevertheless it reduces pause time slightly in average.

4.5.3 Pause time of each garbage collection

We also examine each garbage collection to evaluate biased allocation. Since behavior of garbage collections is changed after applying proposed optimizations, it is not reasonable to compare each garbage collections one-to-one. For example, garbage collections are invoked at different phase of a program and each garbage collection may reclaim different objects after applying optimizations. Therefore we choose the first five garbage collections of _209_db where

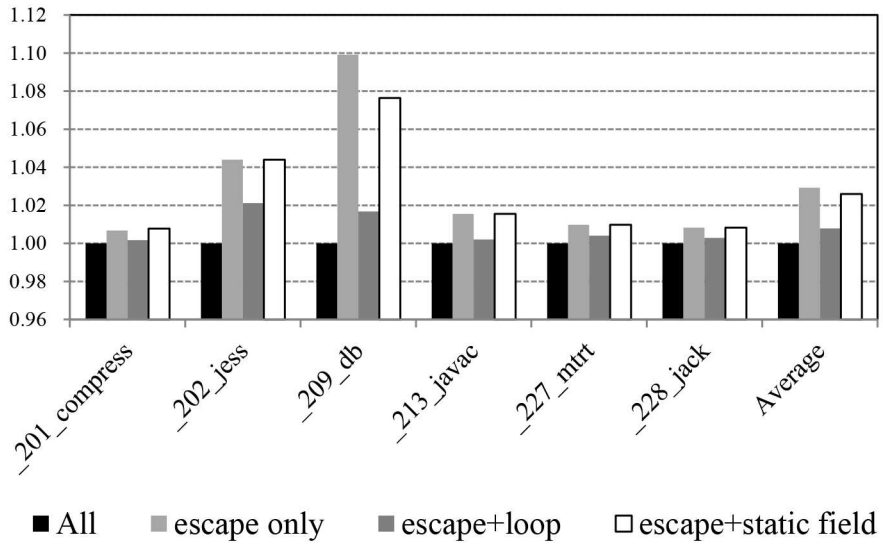

Figure 4.6 Ratio of total pause time of garbage collections compared to all analyses enabled. Therefore All is always one.

promotion occurs and compared number of promotions to original garbage collections as in Figure 4.7. We choose these five garbage collections, because they behaves different but not totally different. Even though it is not fair to compare them one-to-one, it is easily noticed that total number of promotion occurred in the first five garbage collections have been reduced about 25%. All five garbage collections have less number of promotions than original garbage collections. This is expected results, since biased allocator try to allocate objects in a mature area other than in a nursery where some objects should be promoted later. The first garbage collection has been invoked more lately than before, because a nursery is less populated after applying biased allocation.

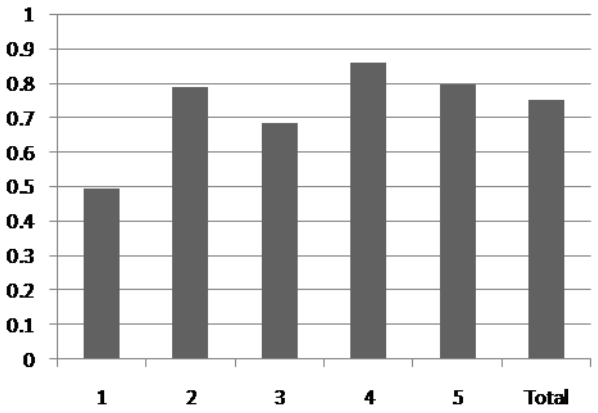

Figure 4.7 Ratio of promotions occurred for the first five garbage collections with biased allocator compared to original in _209_db.

4.6 Summary

We proposed a way that different allocators can cooperate with garbage collectors which have a critical role in memory management of virtual machine. For a generational garbage collector, we proposed approaches which make use of existing analysis techniques to relieve the side effect of generational garbage collector. Allocation sites have been chosen and biased with three analyses and each biased allocation site uses new biased allocators instead of original allocator. We implement a proposed approach in real embedded Java device and evaluate the effectiveness. Total pause time of garbage collections has been reduced and promotion overhead of generational garbage collection has been also reduced in overall.

However we can't guarantee correctness of biased allocation with analyses discussed in this paper. Furthermore analyses discussed in this paper are done at static-time and does not make use of any runtime information. We expect that analyses can be more accurate if runtime information is provided. Each

allocation site use same allocator after decision had made. We expect allocators can be chosen adaptively or allocator itself can evolve for further improvement. Also we use only single biased allocator to bias objects but more allocators can be used for various garbage collectors. We are also expecting that there are opportunities for biased allocation to improve other garbage collectors as well as generational garbage collector.

Chapter 5

Ahead-of-time Heap Management

5.1 Introduction

Automatic memory management improves productivity of programming and secures the stability of a program, since it frees the programmer from various memory management concerns including memory leakage problem. A variety of virtual machines adopts automatic memory management techniques such as garbage collection. For example Java virtual machine [1], JavaScriptCore in webkit [41, 42] and Dalvik virtual machine in Android [2] make use of garbage collector to reclaim dead objects automatically.

Garbage collection (GC) which automatically finds and reclaims dead objects, i.e. objects which are not used anymore, is a famous automatic memory management technique. [8, 23, 21, 22, 20] With garbage collection, programmers don't have to concern tedious implementation of memory management when writing programs. Numerous techniques about garbage collection have been proposed regarding diverse software environments and purposes. Reclaim-

ing dead objects at runtime incurs inevitable runtime overhead and many approaches have been proposed to reduce the overhead, because finding dead object requires a certain amount of computation to make a decision.

We can totally avoid those runtime overhead if infinite memory resources are available and no object is needed to be reclaimed. However in real world, memory resource is limited by hardware and multiple programs share the memory. Even worse, programs are competing for the memory in multitasking environments. A program allocates objects on a heap which is also allocated on total memory for private use of the program. When certain conditions are met, garbage collection starts to reclaim dead objects and secures free space in the heap for future object allocations. Usually garbage collection reclaims dead object when there is no sufficient space in the heap to satisfy a new object allocation request. However, garbage collection is not always successful to secure free space due to various reasons. In such cases, virtual machine tries to complete an object allocation by expanding the heap itself to make a room for new objects, i.e. allocating more heap space on the memory.

As we discussed before, it is obvious that we can avoid garbage collection overhead, if virtual machine chooses to expand the heap instead of performing garbage collection. But it may result in the very large heap and it is only feasible with infinite memory as discussed before. Therefore most virtual machine tries to secure free space in the heap by reclaiming dead object with garbage collection before expanding the heap when there is no sufficient free space in the heap for a new object allocation request.

While it is reasonable to choose garbage collection before expanding the heap to avoid excessive memory use, it is also true that expanding the heap can hide garbage collection overhead. Consequently virtual machine should make a choice carefully between garbage collection and heap expansion considering

overall performance and heap use. A choice of garbage collection and heap expansion does not guarantee the same results and the result is greatly affected by memory behavior of an application. For example, if an application allocate objects which are always live, garbage collection is almost useless because it cannot reclaim objects at all. In such case, expanding heap is better choice than garbage collection considering the performance and heap use, because the heap use is always same regardless of the choice but the performance differs with the choice. A variety of approaches has been introduced to compromise the performance and heap use by speculating the memory behavior of applications. [43, 44, 45, 46, 47] Previous researches shows that it is very hard to predict behavior of applications exactly and there are some ways to speculate the behavior indirectly and we are also inspired by those approaches.

We propose a heuristic for choosing heap expansion carefully to improve overall performance and to provide better user experience with runtime information observed from real applications. Runtime temporal information is taken into consideration as well as runtime spatial information when making decision between garbage collection and heap expansion. Then we try to expand heap ahead-of-time to fully avoid garbage collection overhead with temporal and spatial information.

In the following Section 5.2, we describe our motivation based on observations of real applications. We explain a existing heuristic for garbage collection and heap expansion in Android system in Section 5.3. Then we propose our heuristics based on spatial and temporal information in Section 5.4. We evaluate proposed heuristic in real device in Section 5.5 and summarize this chapter in Section 5.6.

5.2 Motivation

Android employs mark-and-sweep based garbage collector with a concurrent GC approach which is invoked periodically when certain conditions are met to secure sufficient free memory space in time. However it seems that many GC invocations failed to secure sufficient memory and even worse too many GC invocations are requested in a short time interval. Such behaviors of GC result in bad user experiences.

Figure 5.1 depicts GC distribution based on the secured free memory amount in Android. We look into six applications running on Galaxy Nexus to observe garbage collection behavior. Black indicates allocation GC which is invoked due to allocation failure and grey indicates concurrent GC which is invoked periodically.

We observed that more than 50% of GC invocations secure only small amount of free memory, i.e. less than 10 kilobytes, in a `Gallery` application and most of those GC are requested due to allocation failure, i.e. black. For `Camera` and `Maps`, more than 20% of GC invocations reclaim less than 10KB dead objects. Allocation GC tends to secure less amount of free memory than concurrent GC in many applications. Therefore we can infer that allocation GC is not successful to collect lots of dead objects and secure large free memory. In the such situation, Android is forced to expand the heap after garbage collection to secure additional free space when garbage collection secure relatively small amount of free space by reclaiming dead objects.

We also observed that in some applications many GC are requested in a short time. Figure 5.2 shows distribution of time interval depending on the number of GC invocation in the interval where each time interval is one second. Among 30 time intervals, 14 time intervals do not suffer GC overhead at all,

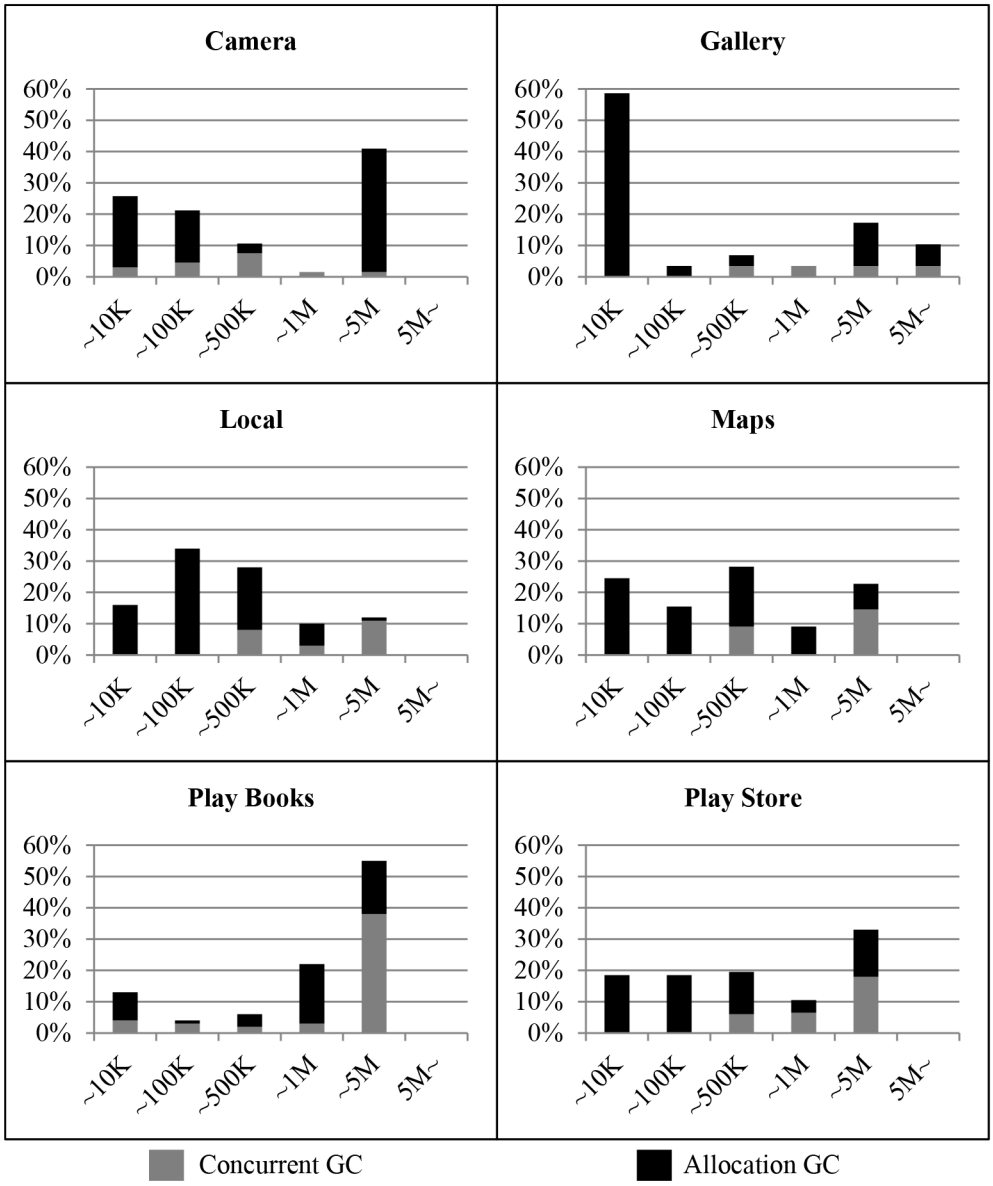

Figure 5.1 GC distribution by secured free memory amount

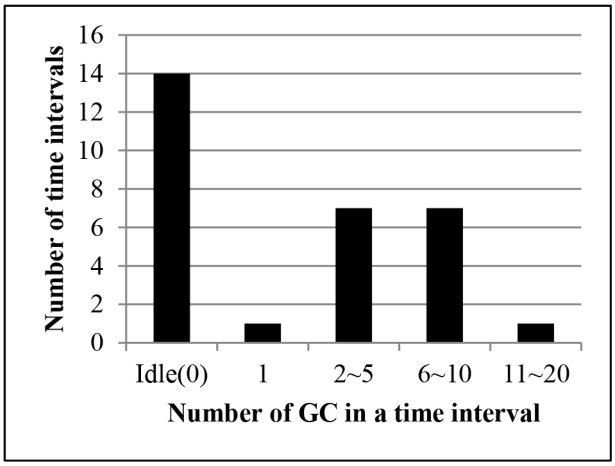

Figure 5.2 Number of time intervals depending on the number of GC in `Maps` application

whereas there is a interval where more than 10 garbage collection are requested in a second. Even with concurrent GC which is invoked periodically to secure free space before allocation GC is invoked, we can conclude from the observation that garbage collection is invoked excessively in a relatively short time interval.

From the first observation, we found that many garbage collections failed to secure sufficient free space in some applications where the heap is forced to be expanded as a consequence. From the latter observation, we observed that distribution of garbage collection is biased and there is a situation where excessively many garbage collections are invoked in a short time interval, resulting in bad user experiences. We are going to propose heuristics to make a choice between heap expansion and garbage collection to avoid such undesirable situation.

5.3 Android

Android adopts Dalvik virtual machine as core execution engine and Dalvik allocates memory from operating system as a heap and manages this heap. As we describe in previous sections, objects are allocated on the heap when an application requests new objects to be allocated. Dalvik employs garbage collection to reclaim dead objects automatically at runtime and secures free spaces for future object allocations. Consequently application programmers can rely on garbage collection and don't have to worry about memory management. Dalvik may expand the heap by allocating a new memory space from operating system when there is no sufficient free space after reclaiming dead objects. In the following subsections, we are going to describe heuristics used in Dalvik to reclaim dead object, i.e. garbage collection heuristic, and to expand the heap after the garbage collection, i.e. heap expansion heuristic.

5.3.1 Garbage Collection

Garbage collection in Dalvik adopts a mark-and-sweep strategy to find and reclaim dead objects. Mark-and-sweep garbage collection has two phases including a mark phase and a sweep phase. The first mark phase traverses all reachable objects recursively from objects in root set which is a predefined by the virtual machine, Dalvik itself in this case. All reachable objects are marked in the mark phase and we can consider all unmarked objects dead because those objects cannot be used from anywhere. We reclaim all unmarked objects and secure new free space by sweeping all unmarked objects in the sweep phase. [8]

Dalvik invokes the garbage collection in two ways. First, there is a dedicated thread for the garbage collection and this thread wakes periodically to reclaim dead objects when certain conditions are met, i.e. concurrent garbage collection. Secondly, a garbage collection starts when there is no sufficient free space

to satisfy the new object allocation request, i.e. allocation garbage collection. In the concurrent GC, a mark-phase of GC and application threads runs concurrently for a time being. Then a GC thread waits all application threads to be stopped and continues to complete remaining mark phases and the whole sweep phase. In the allocation GC, the garbage collection waits all other threads to be stopped and then continues to mark-phase and sweep-phase, often known as a stop-the-world approach.

5.3.2 Heap expansion heuristic

Dalvik decides to expand the heap in two conditions after the garbage collection. If android fails to secure free space which is less than preferred ratio of the total heap size, android chooses to expand the heap. Dalvik also expands the heap when an object allocation request failed to find room for allocation after the garbage collection which is invoked by the allocation request, because the garbage collection already reclaimed all known dead objects but still there is no free space suitable for a new object.

Even when garbage collection is successful to reclaim sufficient dead objects and secures free space larger than the size of allocation request, allocation request may not be satisfied due to fragmentation problem. Fragmentation problem occurs when there is sufficient free space in total but no continuous free space is available to satisfies the allocation request, because free space is fragmented in small pieces. [18] The fragmentation problem can be avoided with more complicated garbage collection, such as mark-and-compact GC and generational GC [8], but it is unavoidable with mark-and-sweep garbage collector used in Android. Those complicated garbage collection requires more computation than mark-and-sweep and may incur other performance problems. There have been approaches that various garbage collection is adaptively chosen [44]

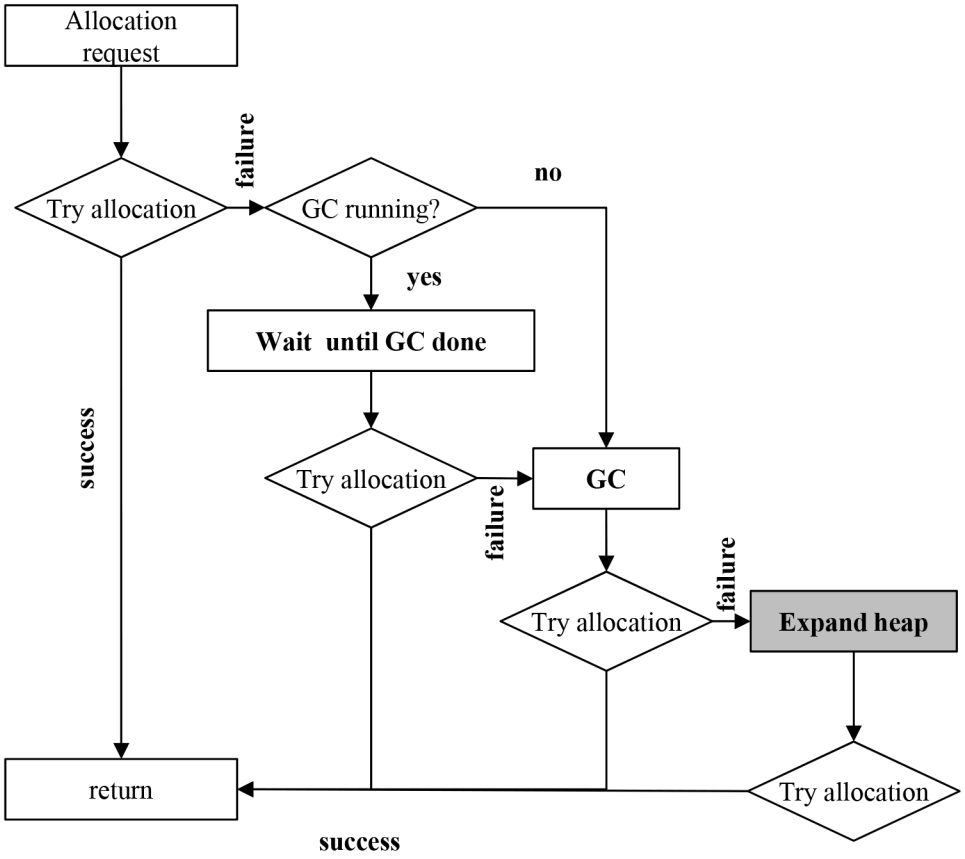

Figure 5.3 Flow of heap management in Android 4.1.2

but the topic is beyond the scope here. In this chapter, we will discuss how to make a choice between garbage collection and heap expansion when garbage collection technique is fixed.

Figure 5.3 depicts the flow of heap management in Android 4.1.2. Allocation trial, garbage collection and heap expansion caused by an allocation request is shown in the figure. We found that Android chooses to expand heap after three allocation trials and two garbage collections in the worst cases. We expect that by expanding heap wisely beforehand we can satisfy the allocation request with

less allocation trials and less garbage collection, i.e. avoiding the worst case scenario. We can also avoid future garbage collections, if we expand the heap more aggressively when expanding the heap in advance. In following sections, we are going to propose an ahead-of-time heap expansion heuristic to achieve less runtime overhead with less garbage collections by exploiting heap expansion aggressively in advance.

5.4 Ahead-of-time heap expansion

We have to consider several issues when expanding the heap. We can avoid every garbage collections except concurrent garbage collection, if we always expand the heap without limitation to satisfy object allocation requests. However size of the heap will grow too large for memory resource available in a device where multiple applications and services run altogether. As a result memory utilization will not be effective in such multi-programming environment, if one application solely consumes large amount of memory. Furthermore we can't avoid concurrent garbage collection in Android and it may incur unaffordable runtime overhead, because garbage collection, especially mark-and-sweep based one, has to traverse all objects to sweep unmarked objects in the whole heap which might be very large. As a result runtime overhead of each garbage collection will be increased as the heap grows, although total number of garbage collection is reduced by always expanding the heap. In other words, user experiences will be getting worse with such heavy runtime overhead of each concurrent garbage collection.

On the contrary we can also suppress the size of heap being increased, if we choose to expand the heap only when garbage collection cannot secure sufficient free space to satisfy an allocation request. Each garbage collection can be completed in a shorter time, because the size of heap is maintained as small as it

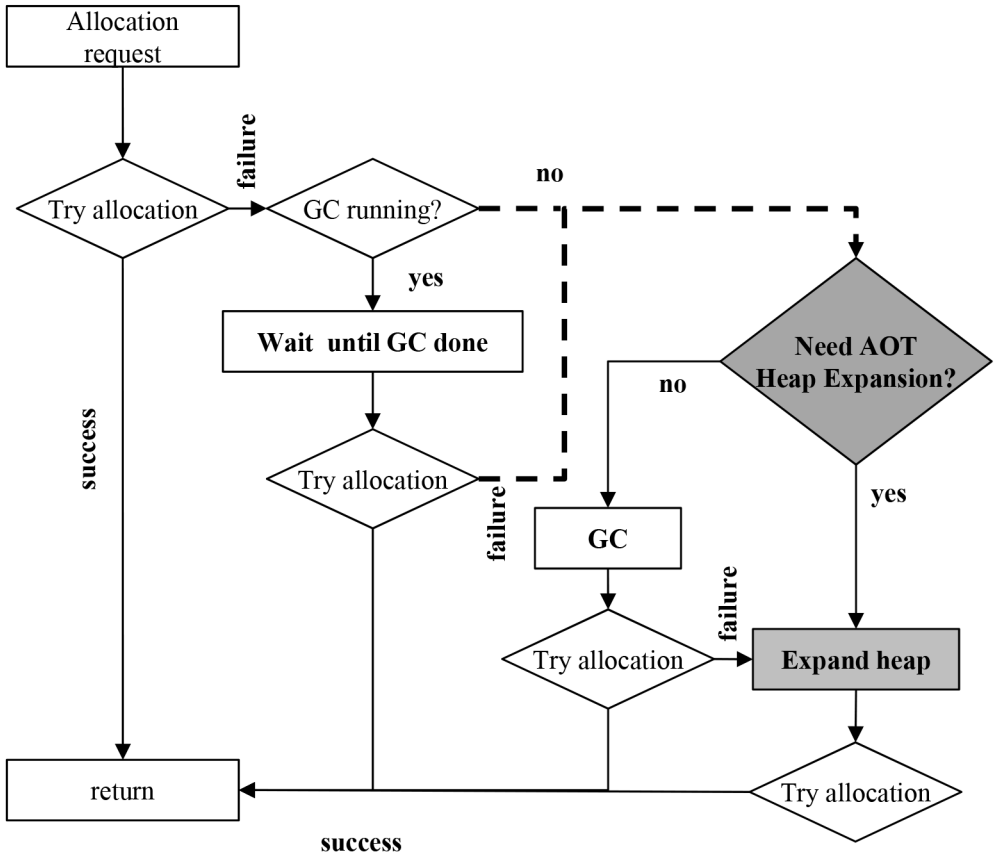

Figure 5.4 Flow of heap management with ahead-of-time heap expansion

can be, while garbage collection is invoked more frequently. Consequently total number of garbage collection will be increased and overall runtime overhead of garbage collections will be also increased, resulting in bad performance of whole Android system.

As we discussed, we have to choose heap expansion heuristic carefully, because heap expansion affects not only total heap size but also performance of whole system. In this chapter, we take into account the runtime spatial information which has been also exploited in other previous researches [44, 45, 47]

and propose a new heuristic to improve the existing heuristic in Android. Then we are going to exploit runtime temporal information to propose heuristics for better user experiences. With these spatial and temporal information, we try to expand heap in advance when there is no need to expand heap right away, i.e. ahead-of-time heap expansion.

Figure 5.4 shows how ahead-of-time heap expansion works in Android when an allocation request made. Unlike original flow in Figure 5.3, there is an additional computation to make a decision between garbage collection and heap expansion. Compared to original flow of Figure 5.3, we can avoid a garbage collection with heap expansion if certain conditions are met. In the following subsections, we will discuss what kind of information is used to make a decision.

5.4.1 Spatial heap expansion

We are going to exploit spatial information to expand the heap in ahead-of-time. There is a lot of spatial information available at runtime regarding object allocation, garbage collection and the heap. For example, size of allocated objects after the last garbage collection, size of reclaimed objects from current garbage collection and size of used heap have been exploited in other researches. [47, 17, 45, 44] Furthermore crafted information with such spatial information, such as ratio, has been also used in various ways.

Among spatial information, we choose information directly related to garbage collection to determine whether the GC is successful or not. The total size of reclaimed objects can be calculated right after the garbage collection. In Section 5.2, we measured the size of reclaimed objects and found out that garbage collection often secures relatively small free space.

We suspected that those garbage collections try to reclaim dead object repeatedly even when there are only few dead objects available. The problem is

that amount of dead objects cannot be determined before the garbage collection. We decide to expand heap when current garbage collection secures relatively small free space. By expanding the heap now, we can reduce the chance of invoking future garbage collections with few dead objects due to allocation failure. If it works, we can reduce the number of garbage collection which secures small free space and total number of garbage collection will be reduced as well. With mark-and-sweep garbage collection, we can reduce overall overhead of garbage collection by reducing the number of garbage collection. As a result, heap management with less GC overhead provide better user experiences and better overall performance.

Spatial information other than size of reclaimed objects can be used as well. We also make use of other information in ahead-of-time heap expansion framework. First, size of total free space available after the current GC is used to make a decision regarding heap expansion, because size of available free space reflects how much amount of new objects can be allocated on the heap before next allocation failure. However the size of free space is not reliable information, since it is useless if the ratio of fragmentation is getting high. Furthermore the size of free space is not flexible and sufficient information, because the size of required memory and the size of working set differ from an application to an application.

To consider different memory requirement of applications, we tried to exploit ratio of free space compared to the heap. We can adaptively consider working set of applications with the ratio instead of the size of free space. However this information is turned out to be unreliable in heap management with mark-and-sweep garbage collection. We will discuss these other spatial information in Section 5.5.

5.4.2 Temporal heap expansion

Although spatial information is very useful and provides valuable insights, it is very hard to figure out correlation between spatial information and performance, especially user experiences. [20, 48] Therefore we try to exploit temporal information in addition to spatial information, because we think that temporal information reflects performance and user experiences directly.

Like spatial information, there are a variety of temporal information with memory management such as object allocation, garbage collection, page fault and etc. Among them, we try to exploit temporal information regarding garbage collection to reduce garbage collection overhead, because garbage collection in Android adopts a stop-the-world approach which stops the whole program execution when garbage collection is running. This strategy affects user experience directly in a bad way when the pause time is getting longer.

The simple and intuitive temporal information related to the garbage collection is garbage collection pause time. However the pause time alone is not enough to determine heap expansion, because pause time of mark-and-sweep collection depends on the number of objects and the number of objects in the heap is totally determined by applications. Therefore using pause time to determine heap expansion can mislead us and cannot be applied to various applications with different set of working objects in general. If we want to reduce the pause time for an application with large working set of objects, we have to change garbage collection itself and this problem is beyond the scope of this paper as we mentioned before.

In fact, as well as garbage collection with long pause time, garbage collection with short pause time can also cause a bad user experience if such short garbage collection is invoked frequently in a short period of time. In Section 5.2, we

observed that many garbage collections were requested in a short time. Based on the observation, we are going to propose a way to expand the heap in advance when garbage collection is called multiple times in a short time interval. The simplest temporal information is an interval between garbage collections and it can be measured directly. However this information only reflects the last two garbage collection and it is not enough to determine whether many garbage collections have been invoked frequently in a short time interval.

Instead we count up the number of consecutive garbage collections only when an interval between last two garbage collections is shorter than threshold and reset the counter if the interval is longer than threshold. When the counter meets predefined number of garbage collections, we ascertain that the last consecutive garbage collections have been invoked in a limited of time. Based on the information, we predict that there will be a upcoming garbage collection in a short time again. So we decide to expand heap in advance and we anticipate no more invocation of garbage collection in a short time.

5.4.3 Launch-time heap expansion

A user does not care about user responsiveness when an application is just being launched and there is no way to interact with the application. Instead what a user expected is a fast launching of the application. Therefore we can apply a completely different heuristic for garbage collection and heap expansion when an application is being started.

First, we don't have to rely on concurrent garbage collection, because few user input is required and responsiveness doesn't matter. Unlike the previous approaches, we exploit temporal information to suppress concurrent GC instead of allocation GC. When a signal wakes up a thread for a concurrent garbage collection, we compute the time since the last garbage collection including both

allocation and concurrent garbage collection. If the time interval is shorter than a threshold, we skip a concurrent garbage collection and the thread is being slept again.

We can also expand the heap more aggressively without concerning over expanding the heap, since the heap should grow to a certain size to satisfy a minimum memory requirement of the application when the application is being started. We exploit a spatial information and temporal information to make a decision on an aggressive heap expansion. We calculate the size of free space secured by the collection after an allocation garbage collection. If the size is less than a threshold and the time since the last collection, including both concurrent and allocation, is short, we decide to grow the heap to meet a certain utilization ratio before an allocation trial.

Unlike the ahead-of-time heap expansion in previous sections, we suppress a concurrent garbage collection and we do not avoid an allocation garbage collection but expand the heap more aggressively after the garbage collection. Without considering the responsiveness, we expect less concurrent garbage collections as well as less allocation GC in overall. Since we don't skip an allocation garbage collection, we assert that we can reclaim dead objects in time when relcaiming is really necessary, and therefore we can ease the side effect of an aggressive heap expansion.

5.5 Evaluation

We evaluated proposed heuristics on Galaxy Nexus with Android 4.1.2 Jelly Bean. Galaxy Nexus is a Android smartphone with touchscreen co-developed by Google and Samsung Electronics. It contains 1GB RAM and TI OMAP 4460 which have dual-core 1.2GHz Cortex-A9 supporting ARMv7 instruction set. Android 4.1.2 supports trace-based just-in-time compiler (JITC) to acceler-

ate application execution and manages the heap with mark-and-sweep garbage collector.

Default applications of Android have been used to observe the effect of heuristics. We choose three applications to evaluate proposed approaches while running with user inputs from those default applications, e.g. `Camera`, `Gallery` and `Maps`. `Camera` and `Gallery` invoke many garbage collections but reclaims few dead objects as shown in Section 5.2. On the other hand, `Maps` provided bad user experiences, because garbage collections are invoked a lot in a short time when a user interacts with the maps application. We are going to evaluate the effect of spatial heap expansion and temporal heap expansion with these applications.

To evaluate heuristic for application launching, we use 11 applications including above three applications. These applications include very simple applications as well as complex ones, i.e. `Gallery`, `Calculator`, `MMS`, `Settings`, `Deskclock`, `email`, `Browser`, `Maps`, `Calendar`, `Contacts` and `youtube`.

5.5.1 Spatial heap expansion

We choose threshold to be 10 kilobytes for spatial heap expansion heuristic considering size of reclaimed objects to compare behavior of garbage collection with original one shown in Section 5.2.

Figure 5.5 and Figure 5.6 depict garbage collections distribution depending on the size of reclaimed objects in `Camera` and `Gallery`. About one fourth of garbage collection in camera and about half of collection in gallery secured free space less than 10 kilobytes with original Android heuristic. After applying ahead-of-time heap expansion with spatial information, garbage collection distribution is changed. Ratio of garbage collections which reclaimed less than 10KB of objects has been reduced in both applications. Most of the reduction

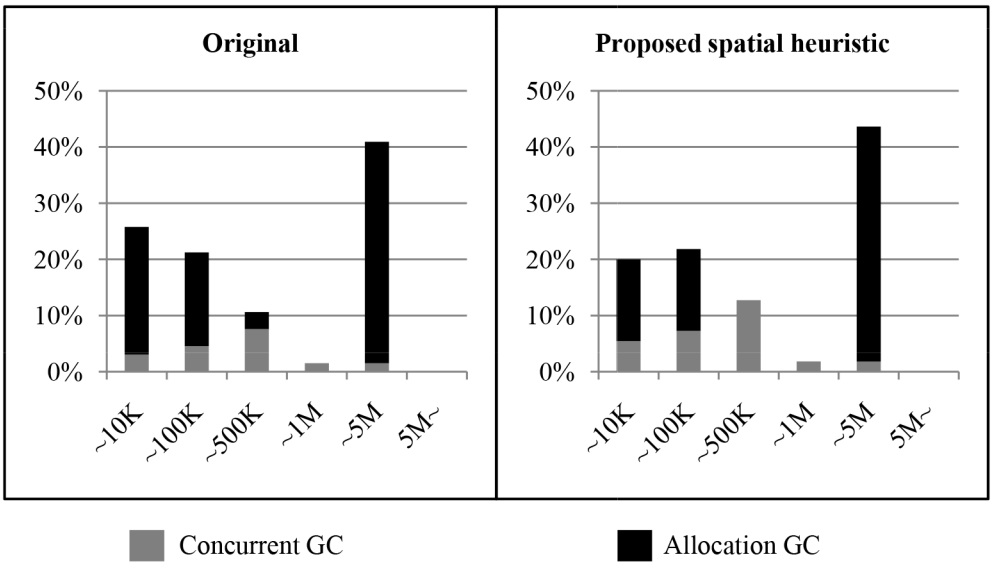

Figure 5.5 GC distribution by the size of reclaimed objects in `Camera`

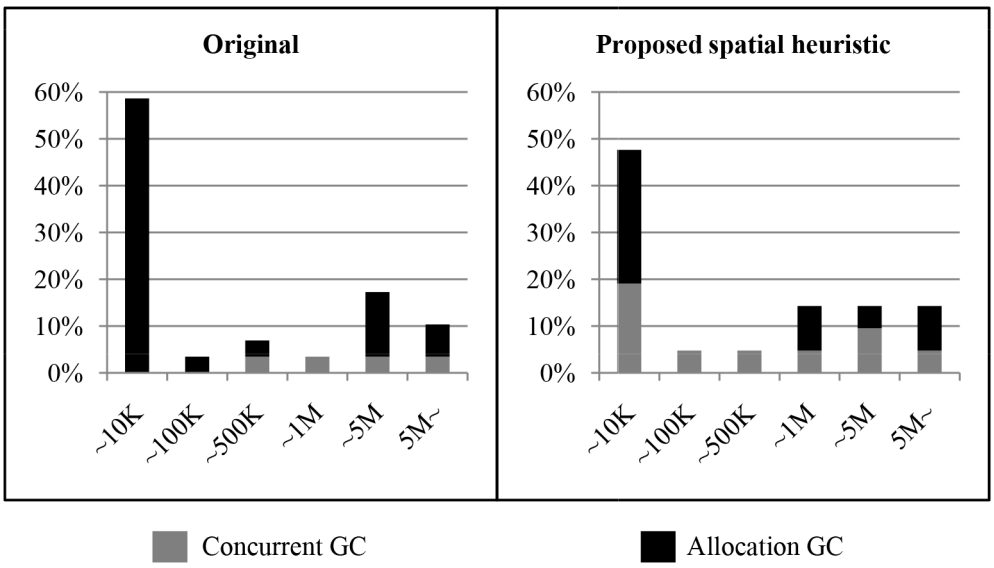

Figure 5.6 GC distribution by the size of reclaimed objects in `Gallery`

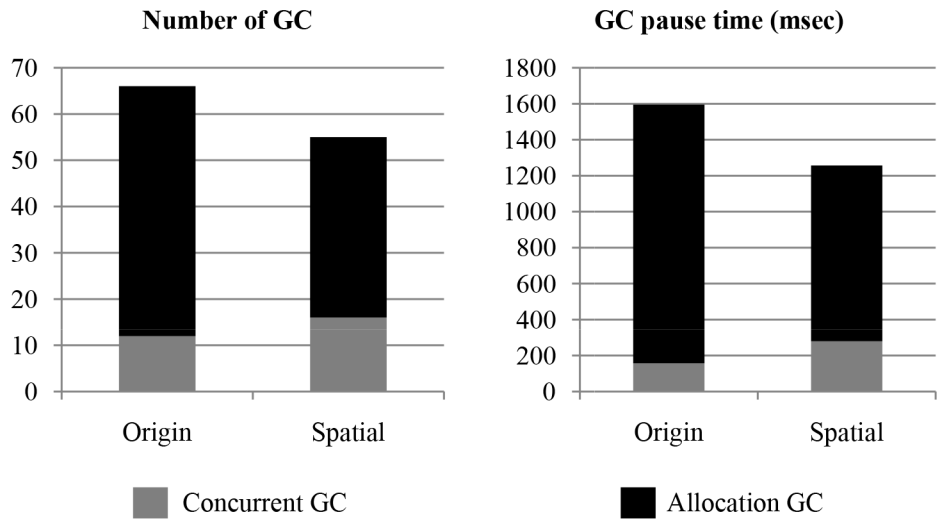

Figure 5.7 Changes of GC behavior in `Camera` after applying spatial heuristic

is due to reduction of allocation GC, while ratio of concurrent GC has been increased. This is expected consequences, because proposed heuristic avoids allocation GC and concurrent GC has more opportunities to be invoke due to less invocation of allocation GC.

We also observed changes of garbage collection behavior as shown in Figure 5.7 and 5.8. Total number of garbage collections is also reduced after applying spatial heap expansion in both applications. Especially allocation GC which is requested when an object allocation failure occurs has been invoked less than original. As discussed before this was expected, because spatial heap expansion has been proposed to avoid allocation garbage collection by expanding heap aggressively. Total pause time of garbage collection has been also reduced as the total number of garbage collection is reduced, although concurrent GC spends more time than before. We shorten the pause time 21.2% in camera and 31% in gallery.

While we reduced the pause time, max size of heap has been increased

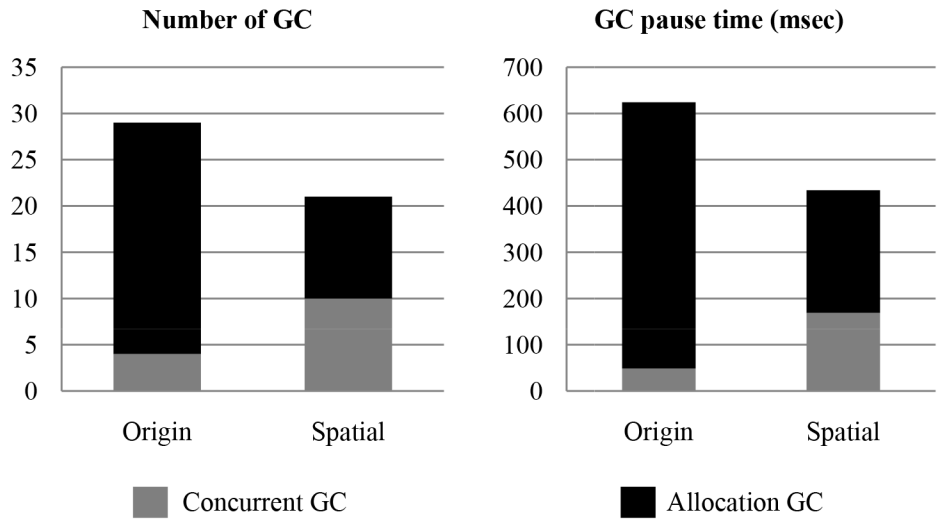

Figure 5.8 Changes of GC behavior in `Gallery` after applying spatial heuristic

somewhat as side effect due to aggressive heap expansion. With ahead-of-time heap expansion, camera requires 18.8% more heap, i.e. from 25.6MB to 30.4MB, and gallery allocates 3.5% more heap , i.e. from 37.6MB to 38.9MB.

5.5.2 Comparision of spatial heap expansion

We evaluate spatial heuristics with size of reclaimed objects in previous section. We also implemented and evaluated ahead-of-time heap expansion with other spatial information, such as size of free space and ratio of free space. Figure 5.9, 5.10 and 5.11 compares all four spatial heuristics, including original, size of reclaimed objects, size of free space and ratio of free space. `Cameara` application is used for the comparison.

Figure 5.9 describes the GC distribution after applying each heuristic. We found out that two spatial heuristic, i.e. size of reclaimed objects and size of free space, are effective in reducing the number of garbage collection with small size of reclaimed objects. Therefore it is reasonable to use those two spatial

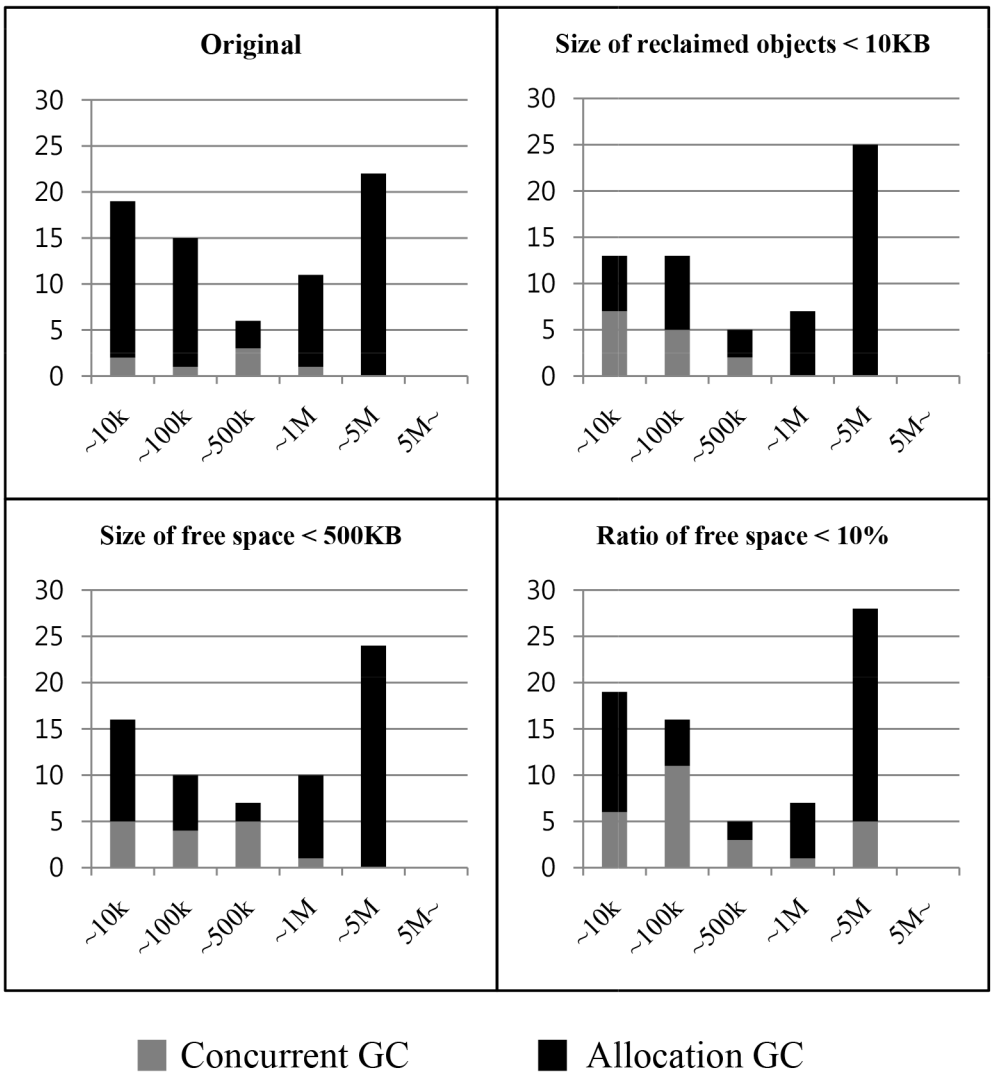

Figure 5.9 GC distribution depending on size of reclaimed objects in `Camera`

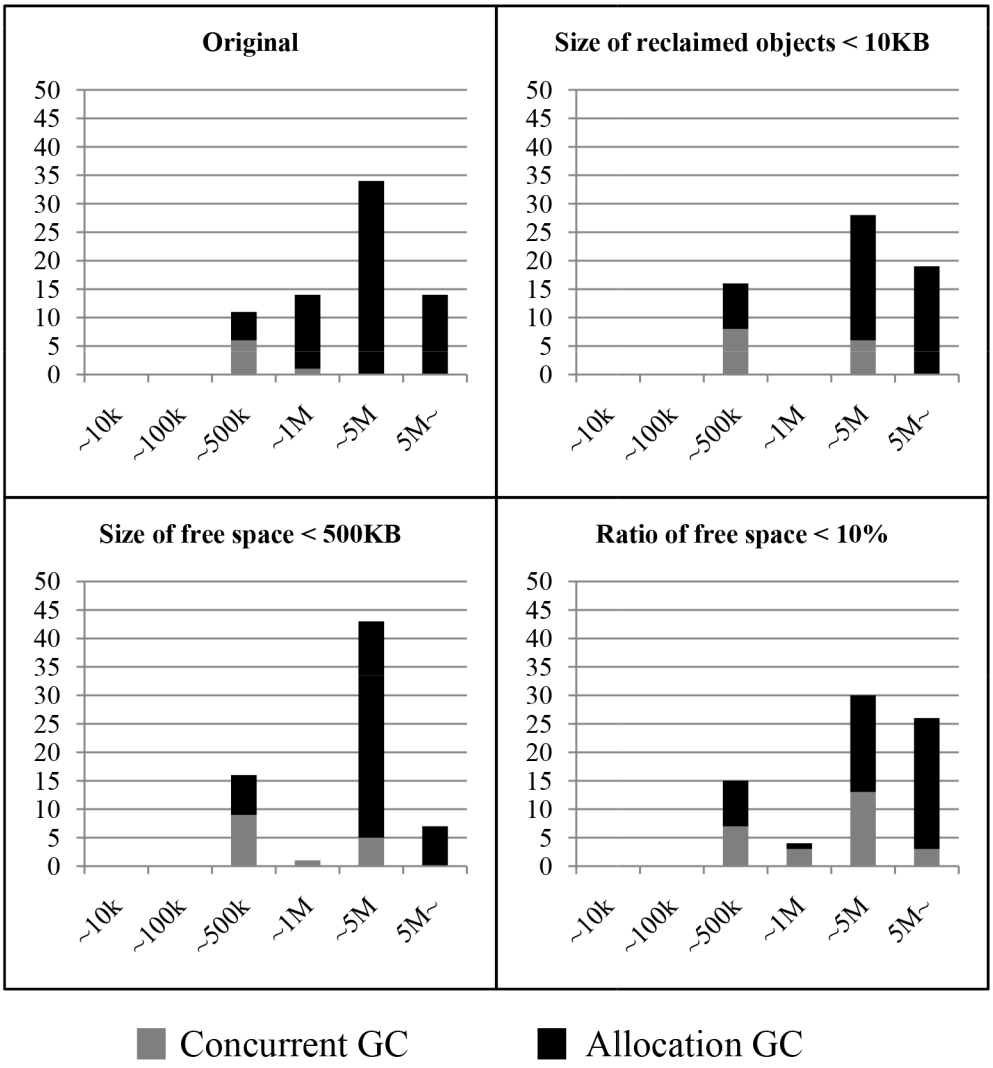

Figure 5.10 GC distribution depending on size of free space in `Camera`

information to predict future behavior of garbage collections.

When we examine the GC distribution by the size of free space as in Figure 5.10, we didn't find meaningful changes except slight changes in distribution. Even with the spatial heuristic with size of free space, there are still allocation

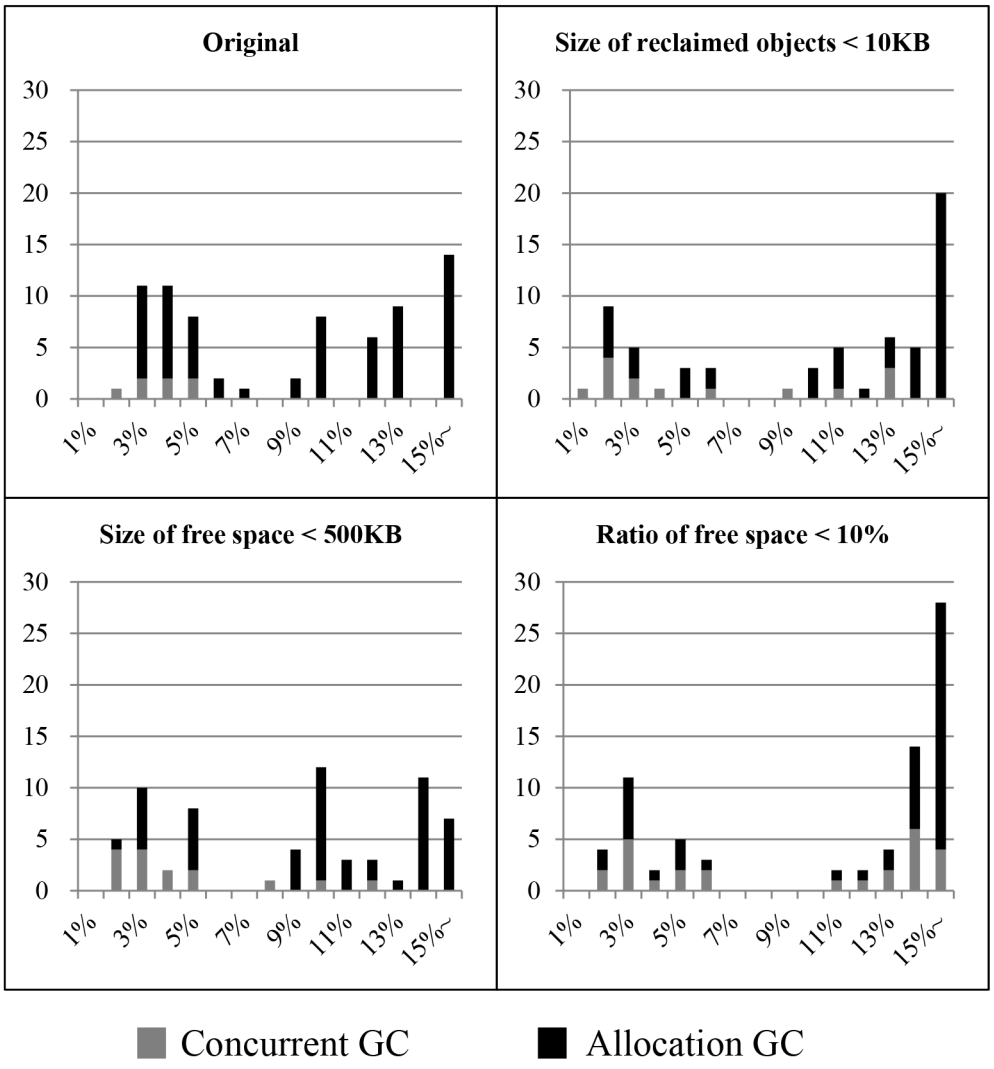

Figure 5.11 GC distribution depending on ratio of free space in `Camera`

GCs which produces less than 500KB free space. From the result, we suspect that size of total free space after the current garbage collection does not guarantee future behaviors of garbage collections.

Finally we look into the ratio of free space after applying four spatial heuristics as shown in Figure 5.11. Two spatial heuristics have changed the distribu-

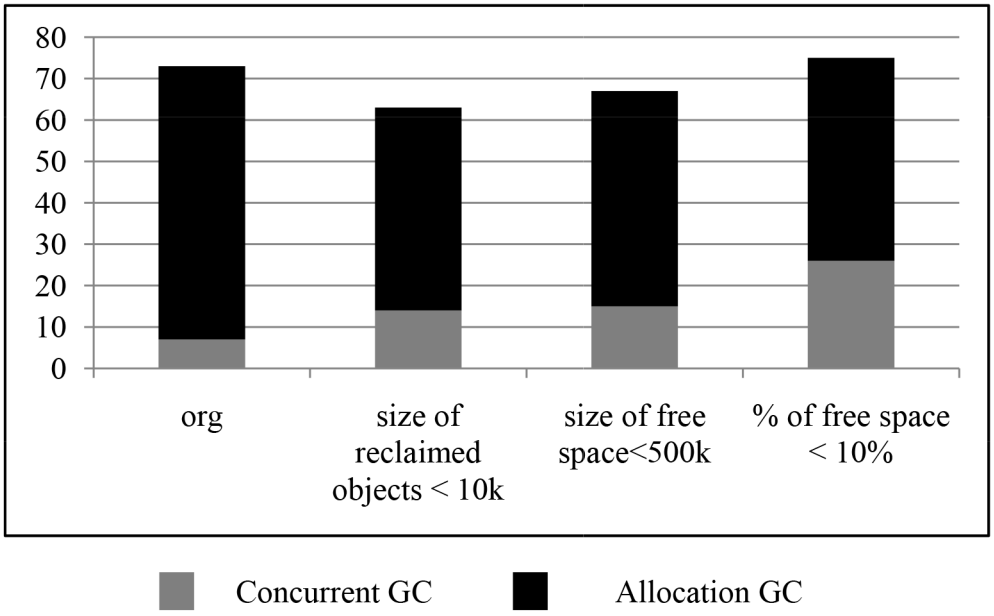

Figure 5.12 Total number of garbage collections of `Camera` with different heuristics

tion of GC. Heuristics with size of reclaimed objects and ratio of free space secures relatively more free space than before. It was expected that number of garbage collections which secures relatively less free space has been reduced with a heuristic with ratio of free space. However we are not convinced whether this changes is beneficial or not, because securing more free space does not promise better performance.

To evaluate the performance of each spatial heuristic, we measured the number and total pause time of GC in Figure 5.12 and 5.13. All three spatial heuristic reduce the number of allocation GC while number of concurrent GC increased. A heuristic with ratio of free space results in more number of GC when considering both allocation GC and concurrent GC. A proposed spatial heuristic with reclaimed object shows the least number of GC overall. Same

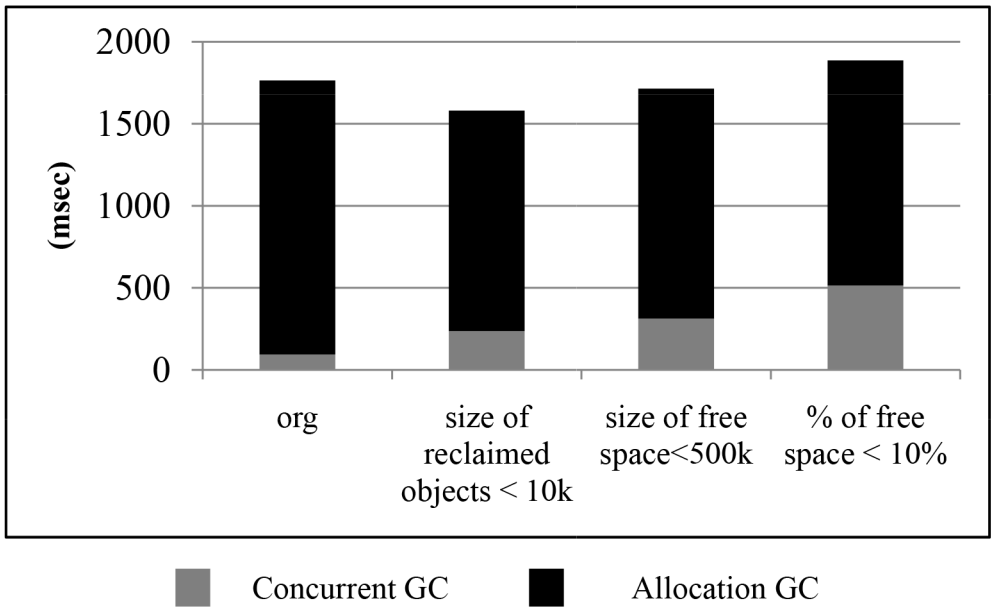

Figure 5.13 GC pause time of `Camera` with different heuristics

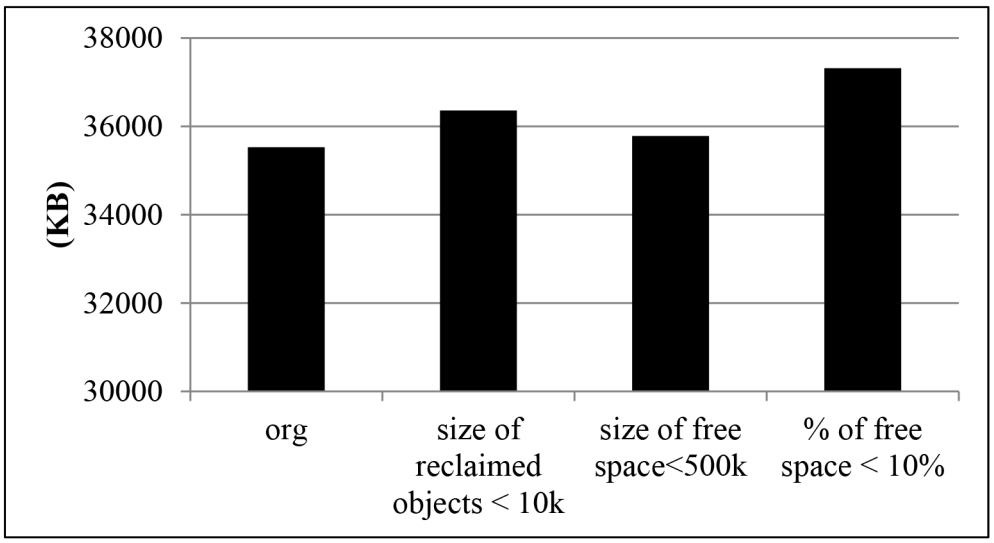

Figure 5.14 Size of max heap in `Camera` with different heuristics

result can be found with pause time of GC as in Figure 5.13, since number of GC and pause time of GC are strongly correlated when mark-and-sweep GC is used.

Size of max heap is also measured in Figure 5.14 to check the side effect of aggressive heap expansion. Original heuristic without ahead-of-time heap expansion shows the smallest size of max heap and it is expected as well, because it always invokes GC before expanding heap. The heuristic with size of reclaimed objects shows the best performance but requires more heap as discussed before.

We decide to track overall behavior of heap to analyze the effect of each heap expansion approach in more detail. During the execution of an application, we traced the size of heap and live objects when each garbage collection completed. The size of live objects is computed during mark phase of garbage collection and the size of heap is measured after heap expansion occurred. We also calculate the ratio of free space compared to total heap. Figure 5.15 and 5.16 show these values regarding each spatial heap expansion heuristic.

All four heuristics show that heap grows as time goes and the size of heap converges to the size of max heap. With ahead-of-time heap expansion, heap grows more rapidly than original in early time. Size of live objects also increases and converges at some point, and this should be same regardless of heuristics because size of live objects is solely depends on the behavior of the application. Therefore we can easily infer that size of free space may increase at first and converges to some point, since size of free space can be directly computed by subtracting size of live objects from size of total heap. Therefore a heuristic with size of free space may not work correctly after some point and threshold should be adaptively changed to cope with such application behavior. Finally ratio of free space is also increasing as time goes and we find out this was mainly due to fragmentation problem in mark-and-sweep garbage collector. Therefore we

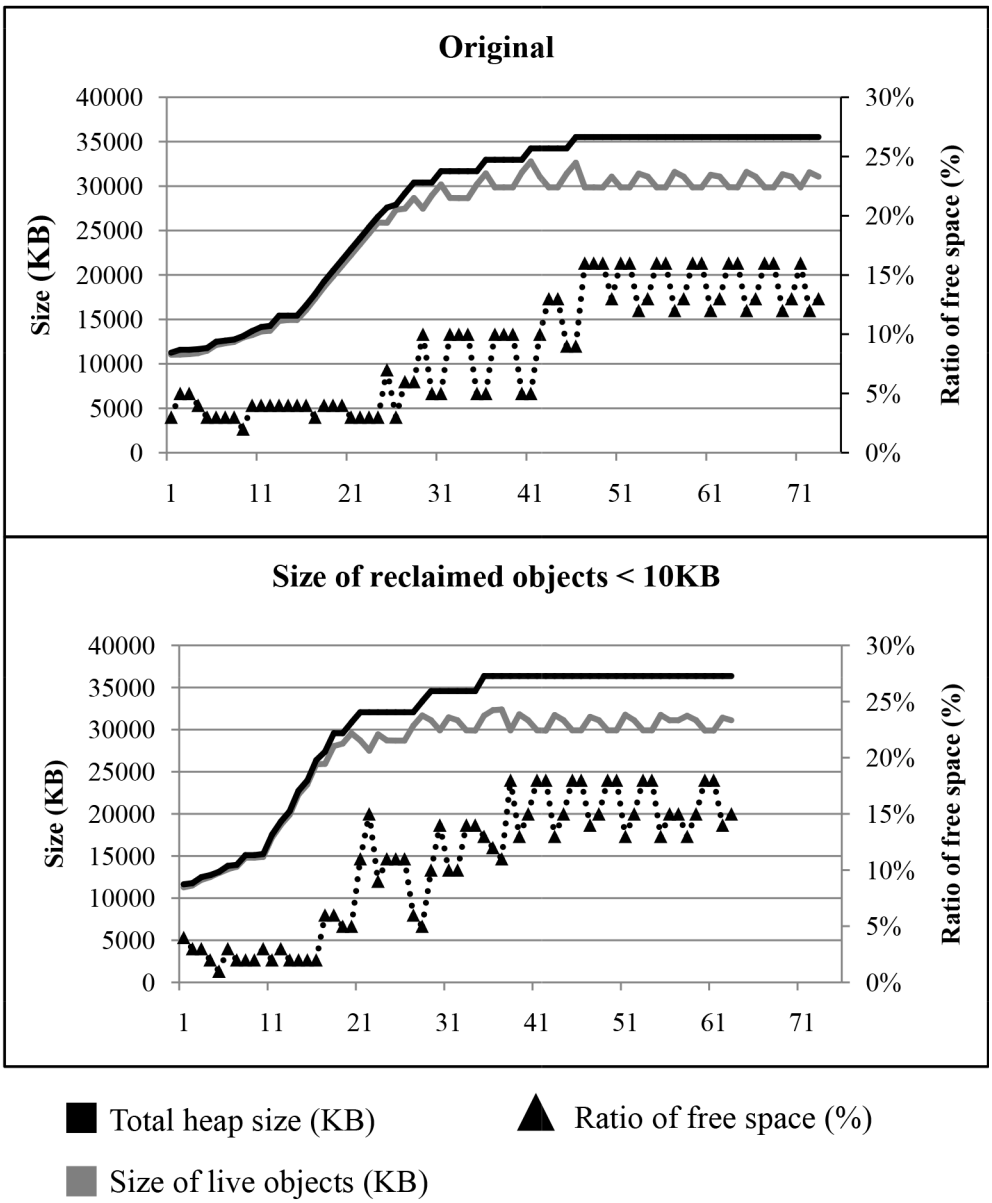

Figure 5.15 Heap behavior of `Camera` with original and proposed heuristics. X-axis denotes each garbage collection and left y-axis depicts the total heap size and the size of live objects in kilobytes, while right y-axis shows ratio of free space in percentage.

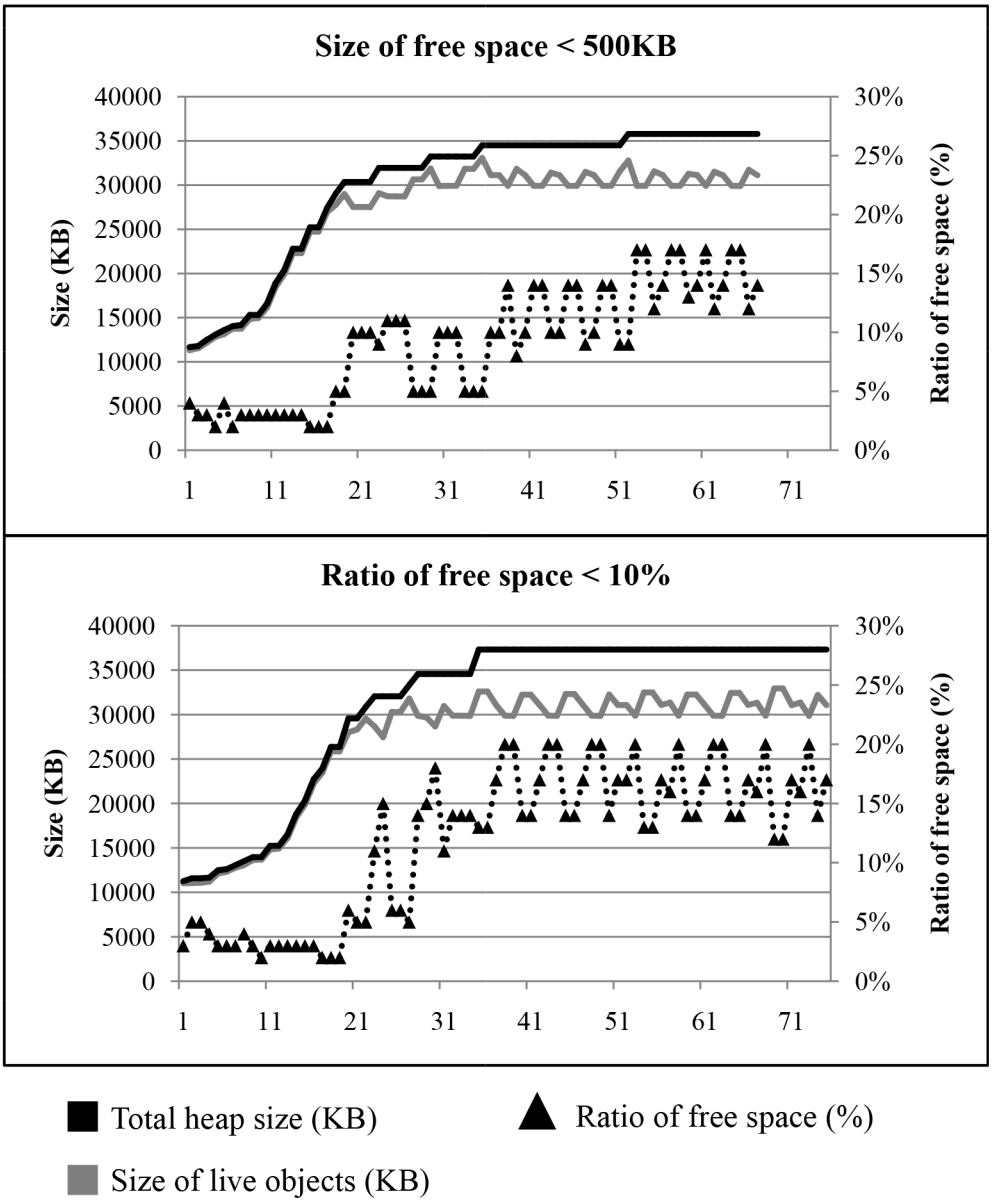

Figure 5.16 Heap behavior of `Camera` with other spatial heuristics. X-axis denotes each garbage collection and left y-axis depicts the total heap size and the size of live objects in kilobytes, while right y-axis shows ratio of free space in percentage.

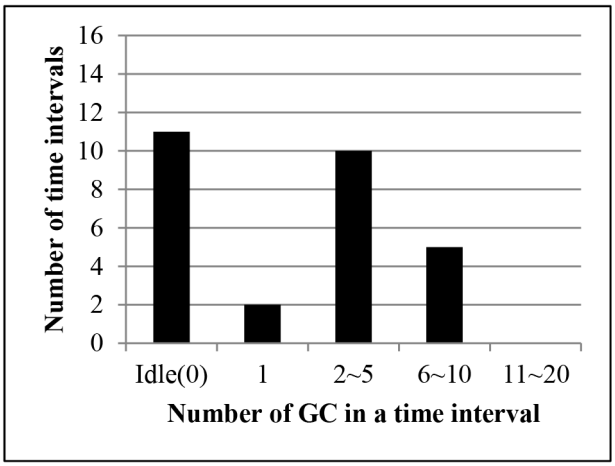

Figure 5.17 Number of time intervals depending on the number of GC in a time interval after applying temporal heap expansion in `Maps`

can conclude that fixed size of free space or ratio of free space are not reliable information to determine ahead-of-time heap expansion with mark-and-sweep garbage collection here, while size of reclaimed object is reliable information to predict future behavior of garbage collection.

5.5.3 Temporal heap expansion

We also evaluate ahead-of-time heap expansion with temporal information. Figure 5.2 in the section 5.2 shows time interval distribution depending on the number of garbage collections invoked within a time interval, where each time interval is one second. We count up the number of garbage collection if time interval between two garbage collection is less than 300ms. Then we expand heap ahead-of-time when counter exceed the threshold. A histogram of time interval after applying ahead-of-time temporal heap expansion is shown in Figure 5.17. Compared to Figure 5.2, we can easily observe that we completely removed time intervals where garbage collection is invoked more than 10 times

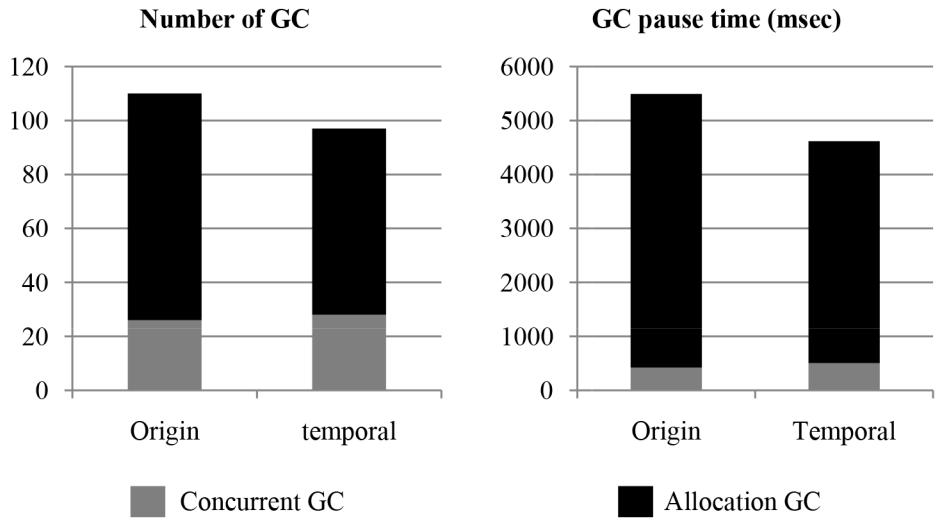

Figure 5.18 Changes of GC behavior in `Maps` after applying temporal heuristic

in a second. We also observed that much less lags were observed when a user interacts with the `Maps` application but it cannot be measured quantitatively. We figure out the improvement qualitatively by recording the behavior of maps application in video and comparing them.

Temporal heap expansion also reduces total number of garbage collections by avoiding garbage collection with timely heap expansion, especially allocation garbage collection. In consequences, number of GC and pause time of GC are reduced in meaningful amount as shown in Figure 5.18. Like spatial heap expansion, allocation GC is avoided with temporal heap expansion, because we expands heap when allocation failure occurred and garbage collection has been invoked too much in a short time.

Although we expand heap based on temporal information other than spatial information, max size of heap has been increased with temporal heuristic. Because we expand the heap even when garbage collection can secure sufficient free space, heap expansion occurred more frequently than before. In `Maps` appli-

Table 5.1 Number of garbage collection and heap expansion. Only heap expansion due to allocation failure after the GC has been counted

Benchmarks	Before	After
Allocation GC	109	111
Concurrent GC	135	112
Heap Expansion	26	16
Total	270	239

Table 5.2 Pause time of garbage collections

Pause time(msec)	Before	After
Allocation GC	5144	5065
Concurrent GC	1353	1077
Total	6497	6142

cation, we require 10.9% more heap than before, e.g. from 27.4MB to 30.4MB.

5.5.4 Launch-time heap expansion

We evaluated a launch-time heuristic with spatial and temporal information when applications start to run. Total 11 applications are launched and applications have been launched explicitly in serial manner five times.

We measured number of garbage collections and number of heap expansion due to allocation failure as in Table 5.1. Concurrent GC has been invoked much less than before, because we avoid the concurrent GC with the heuristic as well as allocation GC. When the last garbage collection, regardless of concurrent or allocation, has already reclaimed objects shortly before, we skip concurrent GC. The number of heap expansion due to allocation failure has been reduced, since we expand heap aggressively to secure sufficient free space after allocation garbage collection when temporal and spatial thresholds are met.

We also measured pause time caused by garbage collections in Table 5.2. Overall pause time has been reduced 5.5% and most of the improvement has been from the concurrent garbage collection as we already expected, because

the number of concurrent garbage collection have been reduced.

5.6 Summary

In this chapter, we propose ahead-of-time heap expansion heuristics to avoid bad garbage collection behavior in Android with temporal and spatial heuristic.

We proposed an ahead-of-time heap expansion framework to enhance existing Android heap management heuristic. Then size of reclaimed objects is considered to determine ahead-of-time heap expansion in addition to existing utilization information. Two more kinds of spatial information are exploited and evaluated with size of reclaimed object. We also exploited temporal information to detect bad garbage collection behavior when many GCs are invoked in a short time and to apply ahead-of-time heap expansion. In such case, we skip next GC invocation by expanding heap ahead-of-time instead of GC. Finally we also propose a heuristic when an application is being launched where the responsiveness doesn't matter. We evaluated proposed heuristics with default key applications in Android. Results show that we can relieve the situation where GCs are invoked many times but reclaim relatively few objects and too many GCs are invoked in a short time. Also we reduce total pause time caused by garbage collections when an application is launched by a user.

We exploit three spatial information and one temporal information in this paper. We can refine these information more carefully and there can be more kinds of information which might be useful for ahead-of-time heap expansion. We use a totally different heuristic when an application starts, but we expect that more improvement can be achieved if we can apply different heuristics of ahead-of-time heap expansion adaptively as an application behavior changes.[47]

Chapter 6

Conculsion

In this paper, I propose three optimizing approaches for memory management in virtual machine. Proposed approaches address memory management issues including object allocation, garbage collection and heap management. Memory management issues of a variety of virtual machine including Dalvik virtual machine in Android platform which is widely spread recently as well as famous Java virtual machine are considered. Also wide range of virtual machine environment is considered including embedded, mobile and server environment.

First, I've proposed a lazy worst fit allocator which is a fast object allocator with low fragmentation. Proposed allocation has been implemented in Java virtual machine and has been evaluated on desktop and server environment. A lazy worst fit allocator outperforms other allocators including segregated first fit and lazy first fit and shows good fragmentation as low as first fit allocator which is known to have the lowest fragmentation.

Secondly, a biased allocator is suggested to address extra overhead of generational garbage collector. A proposed approach has been implemented in embed-

ded Java virtual machine and evaluated on embedded device including digital TV. With three analyses, a biased allocator reduces 4.1% of pause time caused by generational garbage collections in average.

Finally, ahead-of-time heap expansion framework is introduced to avoid worst-case behavior of garbage collection. The proposed approach has been implemented in Dalvik virtual machine of Android platform and evaluated on mobile device, i.e. smartphone, with real applications. Ahead-of-time heap expansion reduces both number of garbage collections and total pause time of garbage collections. Pause time of GC reduced up to 31% in default applications of Android platform.

Memory management deals with a variety of issues and new problems are raised as new devices and software environment are being introduced. These problems are complicated, because several issues are interconnected each other, including object allocation, garbage collection and heap management. I've addressed problems of object allocation, garbage collection and heap management separately, but also tried to address garbage collection overhead by introducing new allocator and new heap management technique. I hope such approaches is useful to deal with future problems in memory management.

Bibliography

[1] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] D. Ehringer, "The dalvik virtual machine architecture," *Techn. report (March 2010)*, 2010, http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf.

[3] "Android official website." [Online]. Available: http://www.android.com

[4] "900 million Android activations!" May 2013, Google I/O 2013. [Online]. Available: https://developers.google.com/events/io/2013/, http://www.youtube.com/watch?v=1CVbQttKUIk

[5] "Interactive tv web," http://www.interactivetvweb.org.

[6] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[7] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Addison-Wesley, 1996.

[8] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, 1st ed. New York, NY, USA: John Wiley and Sons, Inc., 1996.

[9] P. Wilson, M. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Memory Management*, ser. Lecture Notes in Computer Science, H. Baler, Ed. Springer Berlin Heidelberg, 1995, vol. 986, pp. 1–116. [Online]. Available: http://dx.doi.org/10.1007/3-540-60368-9_19

[10] C. J. Cheney, "A nonrecursive list compacting algorithm," *Commun. ACM*, vol. 13, no. 11, pp. 677–678, Nov. 1970. [Online]. Available: http://doi.acm.org/10.1145/362790.362798

[11] P. Wilson, "Uniprocessor garbage collection techniques," in *Memory Management*, ser. Lecture Notes in Computer Science, Y. Bekkers and J. Cohen, Eds. Springer Berlin Heidelberg, 1992, vol. 637, pp. 1–42. [Online]. Available: http://dx.doi.org/10.1007/BFb0017182

[12] Y. Chung and S.-M. Moon, "Memory allocation with lazy fits," in *Proceedings of the 2Nd International Symposium on Memory Management*, ser. ISMM '00. New York, NY, USA: ACM, 2000, pp. 65–70. [Online]. Available: http://doi.acm.org/10.1145/362422.362457

[13] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Memory allocation policies reconsidered," Technical report, University of Texas at Austin Department of Computer Sciences, Tech. Rep., 1995.

[14] W. T. Comfort, "Multiword list items," *Commun. ACM*, vol. 7, no. 6, pp. 357–362, Jun. 1964. [Online]. Available: http://doi.acm.org/10.1145/512274.512288

[15] D. E. Knuth, *The art of computer programming, Volumn 1: Fundamental algorithms*, 3rd ed. Boston, MA, USA: Addison-Wesley Professional, 1997.

[16] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman, "Latte: A Java VM just-in-time compiler with fast and efficient register allocation," in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 128–. [Online]. Available: http://dl.acm.org/citation.cfm?id=520793.825720

[17] Y. C. Chung, S.-M. Moon, K. Ebcioğlu, and D. Sahlin, "Reducing sweep time for a nearly empty heap," in *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '00. New York, NY, USA: ACM, 2000, pp. 378–389. [Online]. Available: http://doi.acm.org/10.1145/325694.325744

[18] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: Solved?" in *Proceedings of the 1st International Symposium on Memory Management*, ser. ISMM '98. New York, NY, USA: ACM, 1998, pp. 26–36. [Online]. Available: http://doi.acm.org/10.1145/286860.286864

[19] A. W. Appel, "Simple generational garbage collection and fast allocation," *Software: Practice and Experience*, vol. 19, no. 2, pp. 171–183, 1989. [Online]. Available: http://dx.doi.org/10.1002/spe.4380190206

[20] M. Hertz, Y. Feng, and E. D. Berger, "Garbage collection without paging," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05.

New York, NY, USA: ACM, 2005, pp. 143–153. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065028

[21] F. Xian, W. Srisa-an, C. Jia, and H. Jiang, "AS-GC: An efficient generational garbage collector for Java application servers," in *Proceedings of the 21st European Conference on Object-Oriented Programming*, ser. ECOOP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 126–150. [Online]. Available: http://dl.acm.org/citation.cfm?id=2394758.2394768

[22] P. Reames and G. Necula, "Towards hinted collection: Annotations for decreasing garbage collector pause times," in *Proceedings of the 2013 International Symposium on Memory Management*, ser. ISMM '13. New York, NY, USA: ACM, 2013, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2464157.2464158

[23] Y. Levanoni and E. Petrank, "An on-the-fly reference counting garbage collector for Java," in *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '01. New York, NY, USA: ACM, 2001, pp. 367–380. [Online]. Available: http://doi.acm.org/10.1145/504282.504309

[24] "Memory management in the Java HotSpot virtual machine," Apr. 2006, http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf.

[25] D. Doligez and X. Leroy, "A concurrent, generational garbage collector for a multithreaded implementation of ml," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '93. New York, NY, USA: ACM, 1993, pp. 113–123. [Online]. Available: http://doi.acm.org/10.1145/158511.158611

[26] A. Krall, "Efficient JavaVM just-in-time compilation," in *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 205–. [Online]. Available: http://dl.acm.org/citation.cfm?id=522344.825703

[27] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "Adaptive optimization in the Jalapeno JVM," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '00. New York, NY, USA: ACM, 2000, pp. 47–65. [Online]. Available: http://doi.acm.org/10.1145/353171.353175

[28] J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 97–113, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/857076.857077

[29] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson, "Toba: Java for applications a way ahead of time (wat) compiler," in *Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3*, ser. COOTS'97. Berkeley, CA, USA: USENIX Association, 1997, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268028.1268031

[30] A. Varma and S. S. Bhattacharyya, "Java-through-C compilation: An enabling technology for Java in embedded systems," in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 3*, ser. DATE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 30 161–. [Online]. Available: http://dl.acm.org/citation.cfm?id=968880.969233

[31] A. Nilsson and S. Robertz, "On real-time performance of ahead-of-time compiled Java," in *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 372–381.

[32] H.-K. Choi, D.-H. Jung, and S.-M. Moon, "Install-time compiler for embedded mobile devices," in *Proceedings of Workshop on Interaction between Compilers and Computer Architectures*, ser. INTERACT-12, 2008.

[33] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proceedings of the 9th European Conference on Object-Oriented Programming*, ser. ECOOP '95. London, UK, UK: Springer-Verlag, 1995, pp. 77–101. [Online]. Available: http://dl.acm.org/citation.cfm?id=646153.679523

[34] G. Snelting and F. Tip, "Understanding class hierarchies using concept analysis," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 3, pp. 540–582, May 2000. [Online]. Available: http://doi.acm.org/10.1145/353926.353940

[35] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for Java," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '99. New York, NY, USA: ACM, 1999, pp. 1–19. [Online]. Available: http://doi.acm.org/10.1145/320384.320386

[36] D. Gay and B. Steensgaard, "Stack allocating objects in Java," Microsoft Research, Tech. Rep., 1999.

[37] "Java HotSpot^{TM} virtual machine performance enhancements," 2013, http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html.

[38] M. Hirzel, J. Henkel, A. Diwan, and M. Hind, "Understanding the connectivity of heap objects," in *Proceedings of the 3rd International Symposium on Memory Management*, ser. ISMM '02. New York, NY, USA: ACM, 2002, pp. 36–49. [Online]. Available: http://doi.acm.org/10.1145/512429.512435

[39] "Phoneme project," https://java.net/projects/phoneme.

[40] "SPECjvm98 documentation," 1999, http://www.spec.org/osg/jvm98/jvm98/doc/index.html.

[41] "JavaScriptCore," http://trac.webkit.org/wiki/JavaScriptCore.

[42] "JS Core Garbage Collector," http://trac.webkit.org/wiki/JS%20Core%20Garbage%20Collector.

[43] E. Andreasson, F. Hoffmann, and O. Lindholm, "To collect or not to collect? machine learning for memory management." in *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, S. P. Midkiff, Ed. Berkeley, CA, USA: USENIX Association, 2002, pp. 27–39.

[44] S. Soman, C. Krintz, and D. F. Bacon, "Dynamic selection of application-specific garbage collectors," in *Proceedings of the 4th International Symposium on Memory Management*, ser. ISMM '04. New York, NY, USA: ACM, 2004, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/1029873.1029880

[45] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Automatic heap sizing: Taking real memory into account," in *Proceedings of the 4th International Symposium on Memory Management*, ser. ISMM

'04. New York, NY, USA: ACM, 2004, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/1029873.1029881

[46] D. Buytaert, K. Venstermans, L. Eeckhout, and K. De Bosschere, "Garbage collection hints," in *Proceedings of the First International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 233–248. [Online]. Available: http://dx.doi.org/10.1007/11587514_16

[47] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara, "Program-level adaptive memory management," in *Proceedings of the 5th International Symposium on Memory Management*, ser. ISMM '06. New York, NY, USA: ACM, 2006, pp. 174–183. [Online]. Available: http://doi.acm.org/10.1145/1133956.1133979

[48] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI. New York, NY, USA: ACM, 2004, pp. 177–188. [Online]. Available: http://doi.acm.org/10.1145/1024393.1024415

요약

메모리 관리는 가상머신의 핵심 기능 중 하나이며 가상머신의 성능에 큰 영향을 준다. 자바와 같은 가상머신을 위한 최신의 프로그래밍 언어들은 동적 메모리 할당 기법을 사용하며 객체를 heap에서 자주 할당한다. 이렇게 할당된 객체들은 추후 더 이상 사용되지 않게 되면 추후 할당할 객체들을 위한 빈 공간을 확보하기 위해 회수된다. 많은 가상 머신들이 쓰레기 수집기라 불리는 기법을 채택하여 heap에서 사용하지 않는 죽은 객체들을 회수한다. 반면에 heap 자체의 크기를 늘려서 더 많은 객체를 할당하도록 할 수도 있다. 이처럼 메모리 관리의 성능은 객체 할당기법, 쓰레기 수집기 그리고 heap 관리 기법에 의해서 결정된다.

본 논문에서는 가상머신에서 메모리 관리 성능을 향상시키기 위한 세가지 기법을 제안하려고 한다. 우선 lazy worst fit이라는 객체 할당기법을 제안하여 쓰레기 수집기가 있는 가상머신에서 작은 객체들을 빠르게 할당할 수 있도록 하였다. 다음으로 biased allocator를 제안하여 쓰레기 수집기의 추가적인 시간 소모를 줄여 쓰레기 수집기의 수행 시간을 줄일 수 있도록 하였다. 마지막으로 ahead-of-time heap expansion기법을 제안하여 쓰레기 수집기의 호출을 억제하여 사용자 반응성과 메모리 관리 성능을 개선시키도록 하였다.

이렇게 제안된 기법들은 데스크톱, 내장형 그리고 모바일 기기 등과 같은 다양한 환경에서 구현되어 평가되었으며, Java 수행환경을 위한 자바 가상 머신과 Android 환경을 위한 Dalvik 가상머신에 적용되었다. Lazy worst fit 객체 할당기는 다른 할당 기법들과 비교해서 압도적인 성능을 보였으며, 가장 좋은 단편화 현상을 보이는 first fit과 비슷한 수준의 단편화 현상을 보여주었다. Biased allocator는 쓰레기 수집기의 수행시간을 평균적으로 4.1%의 개선하였다. Ahead-of-time heap expansion 기법은 쓰레기 수집기의 수행 횟수와 시간을

모두 줄일 수 있었다. Android 환경의 기본 응용 프로그램들을 이용하여 평가하였을 때, 쓰레기 수집기의 수행 시간은 최대 31% 줄일 수 있었다.

주요어: 최적화, 가상머신, 메모리 관리, 객체 할당, 쓰레기 수집기, 힙 관리
학번: 2002-30447

Acknowledgements

대학원을 시작하면서 알게 된 가상 머신이 최근에는 일반인들에게도 널리 쓰이고 있어 시간이 많이 흘렀음을 느끼게 되며 지금까지 옆에서 기다리면서 언제나 응원을 해 준 가족들 특히 아내 윤경이에게 고마운 마음을 전합니다. 또한 언제나 밝은 모습으로 삶의 활력을 불어 넣어준 종원이와 지민에게도 고맙다는 말을 하고 싶습니다. 또한 마음 고생 많이 시켜드렸는데도 묵묵히 응원해 주신 부모님에게도 감사 드리고 동생에게도 고맙다는 말을 전하고 싶습니다.

다양한 연구 경험을 제공해 주시고 필요한 조언을 해주시며 지도해 주신 지도교수님께 감사 드립니다. 또한 바쁘신 중에도 박사 논문 지도를 위해 시간을 내어주신 백윤흥 교수님, 이재진 교수님, 이혁재 교수님에게 감사 드립니다. 그리고 마지막으로 박사 심사에 위원으로 참여하여 시간을 쪼개어 여러 조언을 아끼지 않은 김수현 선배님에게 감사하다는 말을 전하고 싶습니다.

언제 봐도 반가운 친구들, 성엽, 동희, 준석, 성수, 철오 등에게도 덕분에 어려운 일이나 좋은 일이 있을 때 힘을 얻을 수 있었다고 말을 전하고 싶고 앞으로도 계속 변치 않기를 바라며 대학에서 엔지니어로서의 고민 그리고 이제는 삶에 대한 고민까지 나눌 수 있는 친구들인 영균, 용하, 재목, 정환, 성국, 용식, 기린, 영규, 재영, 효진 등에게도 같은 말을 전하고 싶다.

그리고 연구실에서 매일 얼굴을 보면서 시간을 보냈던 여러 분들에게도 인사의 말을 전하고 싶습니다. 우선 연구실에서 오랜 시간을 같이 보내며 연구실

생활에 활력을 준 이제형 선배님, 홍성현, 정동헌, 오형석에게도 고마웠다고 말을 전하고 싶습니다. 또한 연구실에 처음 들어와서 많은 것을 가르쳐 주셨던 박진표 선배님을 비롯하여 여러 선배님들에게 많은 도움을 받았던 기억이 납니다. 또한 벤처창업이라는 경험과 추억을 같이 쌓았던 양병선, 이준표, 이승일, 이흥복 선배님들 그리고 동기 하영에게도 덕분에 좋은 경험을 할 수 있었다는 말을 하고 싶습니다. 그 외에도 연구실에서 수학하며 서로를 알게 된 정홍집, 이상규, 문민수, 김정래, 유준민, 최선일, 배성환, 박종국, 김진철, 김성무 등도 기억에 남습니다. 마지막으로 최근 알게 된 성원, 원기, 진석, 혁우, 지환, 진우 등 후배들에게도 덕분에 연구실 생활이 즐거웠다고 전하고 싶습니다. 모두 하나하나 언급하지 못하지만 덕분에 좋은 추억을 가지고 졸업한다고 전하고 싶습니다.