

# 國立交通大學

資訊工程系

碩士論文

在爪哇虛擬機器中於方法回傳時釋放堆積中區域性  
物件



**Freeing Local Objects in Heap upon Method Returning in JVM**

研究生：劉彥志

指導教授：鍾崇斌 教授

中華民國九十四年八月

在爪哇虛擬機器中於方法回傳時釋放堆積中區域性  
物件

**Freeing Local Objects in Heap upon Method Returning in JVM**

研究生：劉彥志

Student : Yen-Chih Liu

指導教授：鍾崇斌

Advisor : Chung-Ping Chung

國立交通大學

資訊工程系

碩士論文

A Thesis

Submitted to Department of Computer Science and Information Engineering  
College of Electrical Engineering and Computer Science  
National Chiao Tung University  
in partial Fulfillment of the Requirements  
for the Degree of  
Master  
in

Computer Science and Information Engineering

August 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年八月


# 在爪哇虛擬機器中於方法回傳時釋放堆積中區域性物件

學生：劉彥志

指導教授：鍾崇斌 博士

國立交通大學資訊工程學系碩士班

## 摘 要



爪哇虛擬機器(Java Virtual Machine)近來被利用在記憶體受限的嵌入式系統中。然而，在記憶體受限的環境下執行爪哇程式會導致呼叫垃圾收集的次數變多。生命週期不會超出配置他之方法的物件被稱為區域物件。如何在方法回傳後掃除生命週期已經結束之區域物件來減少垃圾收集的頻率就變成記憶體受限系統下的重要課題。現存將區域物件放入方法框(Method Frame)中的方案並無法掃除所有的被辨識出來的區域物件。所以我們將提出一種在堆積(Heap)而非方法框中管理區域物件的機制，讓堆積中被辨識出來的區域物件可以在方法回傳時被釋放。這個研究將分別針對如何在堆積中配置區域物件，如何在方法回傳時掃除堆積中該方法的區域物件，以及原機器上加入這個設計會碰到的問題做討論，並分析帶來的負擔。最後的結果，我們可以看到把物件配置在堆積中並在方法回傳時釋放他可以讓我們在記憶體極受限的情形下減少 60%的垃圾收集呼叫並得到比原始爪哇機器好 11%的整體執行時間速度效能。而只要堆積的大小不要大於程式最小執行大小的 9 倍，用我們的方法對於速度效能都有幫助。

# Freeing Local Objects in Heap upon Method Returning in JVM

Student: Yen-Shih Liu

Advisors: Dr. Chung-Ping Chung

Department of Computer Science and Information Engineering  
National Chiao Tung University

## ABSTRACT

Java Virtual Machine is adopted in embedded memory constrained system recently. However, executing a Java program in memory constrained system will result in more frequent invocation of garbage collection. Objects whose lifetime will not escape scope of method which allocates it are called local objects. Freeing local objects upon method return to reduce frequency of garbage collection is important in memory constrained system. Current approach to allocate local objects in method frame can not free all identified local objects. So, we propose a mechanism to manage local objects in heap but not method frame, to free all identified local objects upon method return. In this research issues about how to allocate local objects in heap, how to free local objects in heap upon method return, and problems to add this design to original JVM are discussed. Overhead of this design will be analyzed, too. As a result, it reduces 60% of GC invocation counts, and brings 11% speedup over original JVM on total execution time in extremely memory constrained environment. When heap size is less than 9 times of minimal execution heap size, our design helps on speed performance ◦

# Acknowledgment

能夠完成這篇論文，首先我要感謝的是我的父母，在我的求學過程中，不斷的給我勉勵，並在任何時候都給予我支持，讓我可以無後顧之憂的從事我的研究，一如過去二十幾年來，你們無怨無悔為我所做的一切。

再來，要感謝我的指導教授鍾崇斌教授，能夠在我於學習路途上怠惰時鞭策我，在我迷惑時指導我，指引我走向正確的追求學問之道。以及另一位實驗室的大家長，單智君教授，也在我研究的過程中，給了我許多的建議。如果我能在研究上有一點小小的成就，這兩位老師，絕對引領我前進最重要的指南。

另外，也要在此感謝實驗室的鄭哲聖學長以及喬偉豪學長，以及曾指導過我的碩士班學長們。當我在致學的過程中遇上了疑慮之時，你們總是不厭其煩的教導我，讓我避免了許多的錯誤。



還有，高中和大學時期所認識的好朋友們以及實驗室的同學們，你們也是我最需要感謝的人，因為有你們一起努力，一起鑽研，一起生活，在我消沉時激勵我奮起，在我自滿時提醒我繼續努力，我的研究所生涯才因此而美好，或者說，我的生命才因此而美好，你們所帶給我的感動，已經非這篇致謝所能表達，而是深深刻嵌進了我的生命之中。你們都是我最好的朋友，我愛你們。

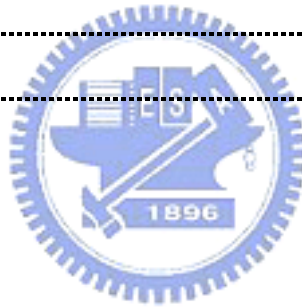
最後，還有一個重要的人，小葉小姐，你在我研究所生涯最開始的時候闖入我的生活中。然後，一直陪我走到最後，不論未來會往哪個方向走下去，你確實是我生命中最重要的人之一，謝謝你，永遠。

謝謝大家。

# Table of Contents

摘要.....	i
ABSTRACT.....	ii
ACKNOWLEDGMENT.....	iii
Table of Contents.....	iv
List of Figures.....	vi
List of Tables.....	vii
Chapter 1 Introduction.....	1
1.1 Reducing Frequency of Garbage Collection.....	1
1.2 Existing Methods.....	2
1.3 Motivation and Objective.....	2
1.4 Thesis Organization.....	3
Chapter 2 Background.....	4
2.1 Java Virtual Machine.....	4
2.1.1 Class Loader and Method Area.....	5
2.1.2 Interpreter.....	5
2.1.3 Java Stack.....	6
2.1.4 Heap.....	7
2.1.5 Garbage Collector.....	8
2.2 Escape Analysis.....	10
2.3 Stack Allocation.....	12
2.4 Summary.....	14
Chapter 3 Design.....	15
3.1 Design Overview.....	15
3.2 Allocating Local Objects.....	16
3.2.1 Data structure to store Local Objects in heap.....	16
3.2.2 Policy to Allocate Local Objects in heap.....	19
3.3 Freeing Local Objects upon method return.....	21

3.4 Cooperation of garbage collector and my design.....	23
3.5 Summary.....	25
Chapter 4 Simulation.....	28
4.1 Evaluation Equation.....	28
4.2 Simulation Environment.....	29
4.3 Benchmark.....	30
4.4 Discussion about minimal size of Local Chunk.....	31
4.5 Simulation Result.....	32
4.5.1 Ratio of GC time to total execution time.....	32
4.5.2 Times of GC in different design.....	33
4.5.3 Total execution time in different design.....	34
4.5.4 Overheads.....	36
Chapter 5 Conclusions.....	38
References.....	41



# List of Figures

Figure 2.1: Abstract inner structure of JVM.....	5
Figure 2.2: Method invocation and return.....	7
Figure 2.3: Data structure of Free Chunk.....	8
Figure 2.4: Data structure in heap.....	8
Figure 2.5: Garbage Collection.....	9
Figure 2.6: Escape Analysis.....	12
Figure 2.7: Stack Allocation.....	13
Figure 3.1: Data structure in heap in my design.....	17
Figure 3.2: Data structure of Local Chunk.....	18
Figure 3.3: Allocating Local Objects.....	20
Figure 3.4: Freeing Local Objects upon method return.....	23
Figure 3.5: Compaction in modified garbage collector.....	25
Figure 4.1: Times of garbage collection with different N.....	32
Figure 4.2: Ratio of garbage collection time to total execution time.....	33
Figure 4.3: Times of garbage collection in different design.....	34
Figure 4.4: Total execution time in different design.....	35
Figure 4.5: (GC time + Overhead in speed) in different design.....	35
Figure 4.6: Ratio of overhead to total execution time in my design.....	36



# List of Tables

Table 3.1: Comparing Stack Allocation with my design.....26



# Chapter 1

## Introduction

In this chapter, we will simply show that why reducing overhead of Garbage Collection is important and how can we reduce overhead of Garbage Collection. And then we will illustrate our motivation and objective to do this research. Finally, we will describe the organization of this thesis.

### 1.1 Reducing Frequency of Garbage Collection

Application of Java Virtual Machine in embedded system is getting popular recently. In an embedded system, memory constraint is usually a common issue that we have to face. However, because Java is an object oriented programming language which does not provide mechanism for programmer to free dead objects themselves, it uses Garbage Collector to collect dead objects when heap is full, which is called Garbage Collection. So, Java Virtual Machine in a memory constrained system will lead to frequently invocation of Garbage Collection, that is because heap tends to be full frequently. In extremely memory constrained environment, the overhead from Garbage Collection occupies significant portion of total execution time. Ratio of Garbage Collection time to total execution time can be more than 50%

Conventionally, Garbage Collection is invoked by Java Virtual Machine whenever more free space is required when allocating object. So, to reduce frequency of Garbage Collection, the trivial idea is to freeing dead objects in heap

to bring more free space in heap [1].

However, in Java Virtual Machine, we do not provide mechanism for programmers to free dead objects, because of concerning about safety. So, generally Java Virtual Machine does not know whether an object is dead except invoking Garbage Collection. Nevertheless, there are some objects whose lifetime will not escape scope of method which allocates them, called **Local Objects**. Because knowing Local Objects are surely dead after method return, we can free them upon method return to bring more free space in heap. As a result, freeing Local Objects upon method return helps to reduce frequency of Garbage Collection.

## 1.2 The Existing Method



To identify Local Objects in Java program, current conventional approach is Escape Analysis. And to free Local Objects upon method return, Java Virtual Machine will put local objects in method frame, which is called Stack Allocation. Then, when method returns, the frame will be popped and Local Objects will be freed, too.

However, objects in stack are not collectable. It leads to that there are some constraints in stack allocation that we can not free all identified Local Objects upon method return. The reason will be explained in detail in section 2.3.

## 1.3 Motivation and Objective

Because Stack Allocation can not free all identified Local Objects because

objects in stack is not collectable. And we know that objects in heap are collectable. So I am motivated to propose and evaluate a mechanism to allocate Local Objects in heap and free them upon method returning. Expecting to free all identified Local Objects upon method return.

## 1.4 Thesis Organization

The rest of the thesis is organized as follows: Section 2 describes the background of freeing Local Objects upon method return, including architecture of Java Virtual Machine, Escape Analysis, and Stack Allocation. Section 3 presents the proposed design to allocate identified Local Objects in heap and free them upon method return. Section 4 simulate results of our mechanism and evaluate it. The last section summarizes this research and discuss about contribution of this research.




# Chapter 2

## Background

Before illustrating my design, some background technology should be known. So, they will be illustrated in this chapter. First, we will present the conceptual architecture and conventional implementation of Java Virtual machine, which executes Java class files. Then, current approach to identify and free local objects upon method returning will be presented. Finally, we will discuss about advantages and constraints of current approach.

### 2.1 Java Virtual Machine



To execute Java programs, we need Java Runtime Environment. The core of JRE is Java Virtual Machine. Java Virtual Machine is a virtual stack machine to execute Java class file. Specification of Java Virtual Machine is illustrated in The Java Virtual Machine Specification [2]. We will illustrate it simply in this section.

In Java Virtual Machine specification, it describes the function of each component and abstract inner architecture of Java Virtual Machine, but not the detail implementation of each component. Figure 2.1 shows the abstract inner architecture of Java Virtual Machine. Afterwards, we will illustrate the function of each component simply. [3][4][5]

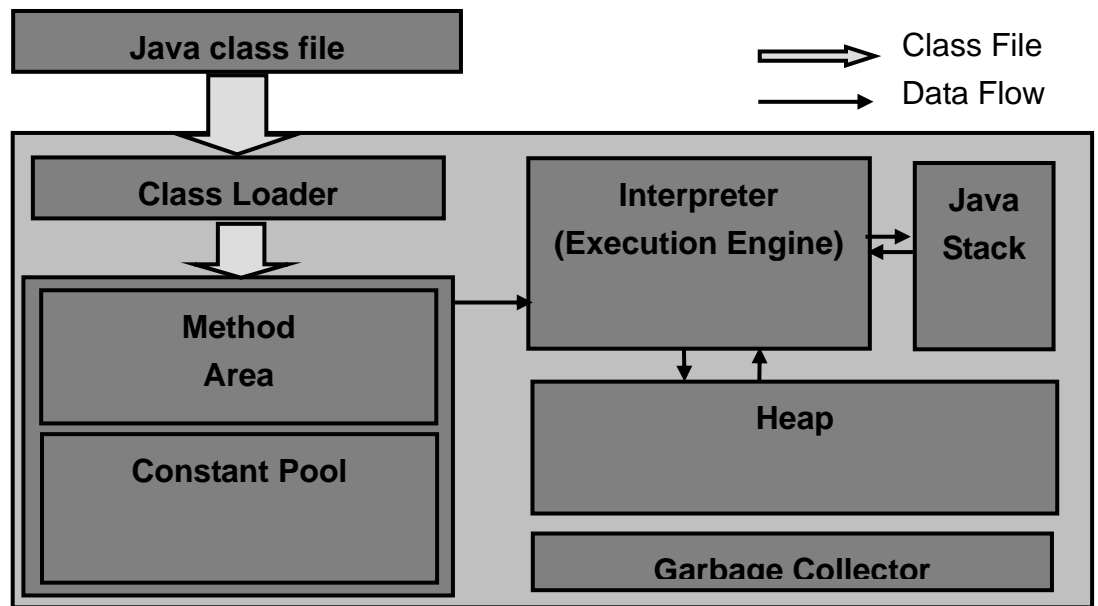


Figure 2.1: Abstract inner structure of Java Virtual Machine

## 2.1.1 Class Loader and Method Area

In Java Virtual Machine, the static information about each class is stored in Method Area, and loaded by Class Loader from Class file. The Class Loader reads Java Class file and converts information in Class file to corresponding data structure in Java Virtual Machine, and store it to Method Area. Class information in Method Area includes type information, constant pool, fields, method information, class variables, bytecodes, and method tables. All thread in a Java Virtual Machine instance share the same method area, so the method area should be designed to be thread safe.

## 2.1.2 Interpreter

Interpreter is the execution engine of Java Virtual Machine. It reads the runtime execution information in Stack to get the current PC and related information to know which bytecode should be interpreted now. Then, it reads the

corresponding bytecode and related information from method area, and executes it. In current research, there are some new technologies which replace some function from interpreter. For example, the Just-in-time Compiler helps interpreter to execute program with native code. But conventionally, interpreter is still the central controller in Java Virtual Machine.

### 2.1.3 Java Stack

Java Stack in Java Virtual Machine is responsible for maintaining method invocation and return. Each thread in Java Virtual Machine has a private Java Stack. The Java Stack is created when a thread is created. The stack is never manipulated directly except to push and pop frames. Each frame in Java Stack is a memory space corresponding to a invoked method. Runtime information like local variables and operand stack are stored in frame. Frame in Java Stack is created when the corresponding method is invoked, and then it will be pushed into Java Stack. Only method corresponding to the top frame in Java Stack is currently executed in the thread. So, when the currently executed method returns, the top frame will be popped, and then the currently executed method will become one which corresponding to new top frame in stack.

Figure 2.2 shows how Java Stack maintains method invocation and return.

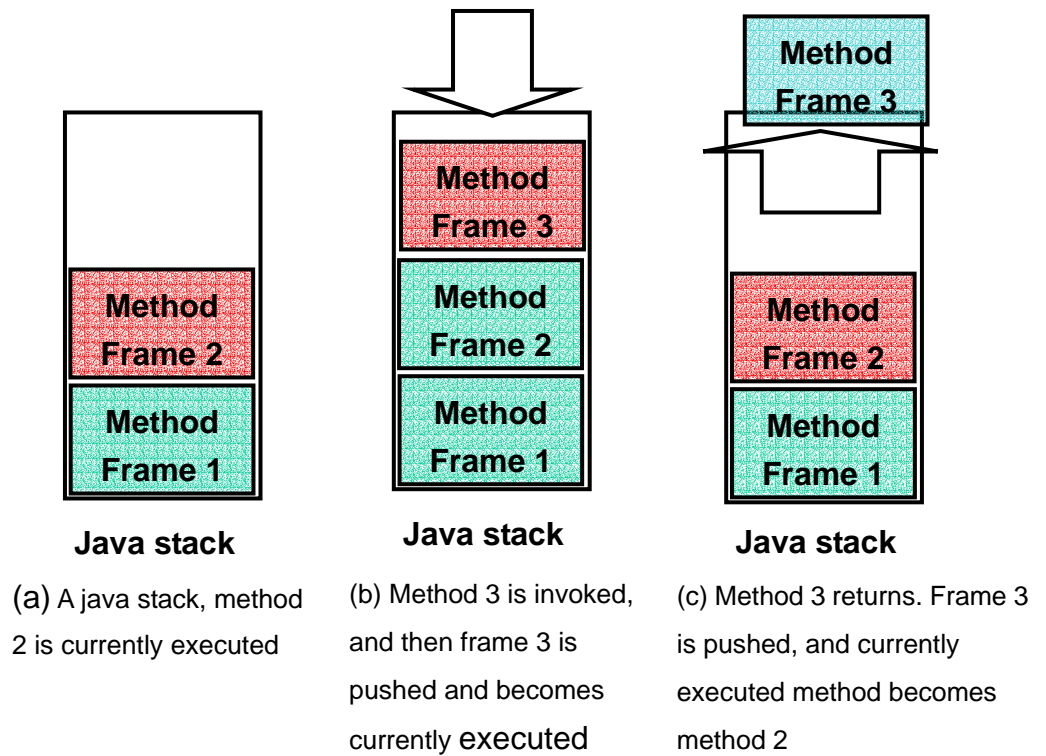
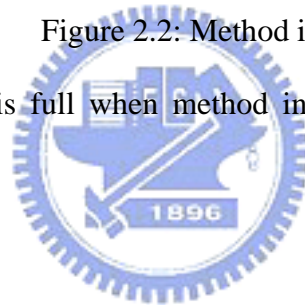


Figure 2.2: Method invocation and return

If Java Stack is full when method invocation, `StackOverflowError` will be thrown



## 2.1.4 Heap

During execution of Java program, if an object is created, we will allocate a free space for it in heap. Conventionally, heap is management with Free Chunk List. Free Chunk List is a linked list data structure which links Free Chunks in heap . Each Free Chunk is a contiguous free space and has a header which shows the size of Free Chunk, and the point to next Free Chunk in Free Chunk List. Data structure of Free Chunk is shown in Figure 2.3. Then free space in heap is represented with Free Chunk List, like Figure 2.4. We can see that free space in heap is linked by Free Chunk List.



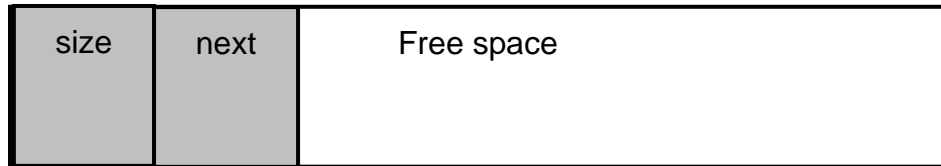


Figure 2.3: Data structure of Free Chunk

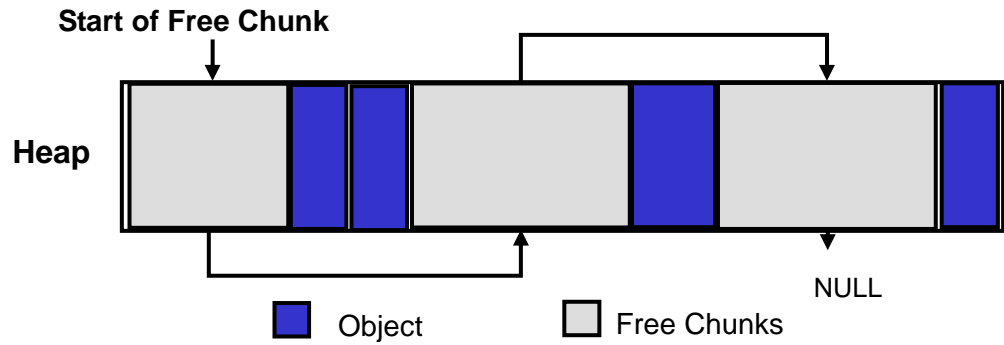


Figure 2.4: Data structure in heap

When Java Virtual Machine needs a free space to allocate new object, it will traverse the Free Chunk List. If there is suitable space for currently created object, Java Virtual machine allocates object in the found out free space. If there are no suitable space after traversing the whole Free Chunk list, Java Virtual Machine invokes Garbage Collection to sweep dead objects in heap. If there are still no suitable space after Garbage Collection, OutOfMemoryError will be thrown.

### 2.1.5 Garbage Collector

Garbage Collector is responsible to sweep dead objects in heap in Java Virtual Machine. Dead Objects are objects which are not referenced any more. Because Java programs access objects via reference, an object which is not referenced can not be used in future, so we say that it is dead. Conventionally, Garbage Collector is implemented with Mark-Sweep-Compact (MSC) algorithm. There are three phase in MSC algorithm: (1) Mark phase: traverse the reference tree and mark

reachable objects as live. (2) Sweep phase: traverse the heap to sweep unmarked objects in heap, because there are not referenced at all, which is called dead. Then , it links free space in heap to Free Chunk List. (3) Compact phase: if Java Virtual Machine need more space after Sweep phase, it compact objects in heap. It will traverse the heap and move live objects to become contiguous allocated from beginning of heap. Compact phase helps Java Virtual machine to eliminate fragmentation in heap. It should be noted that compact phase is optional in each invocation of Garbage Collection.

Figure 2.5 shows how Garbage Collector works in 3 phases. In Figure 2.5(a), the point means reference relation, we can see referenced objects are marked, In figure 2.5(b), we can see non-referenced objects are swept from heap. Finally, in figure 2.5(c), we can see objects are moved to contiguous address from beginning of heap.

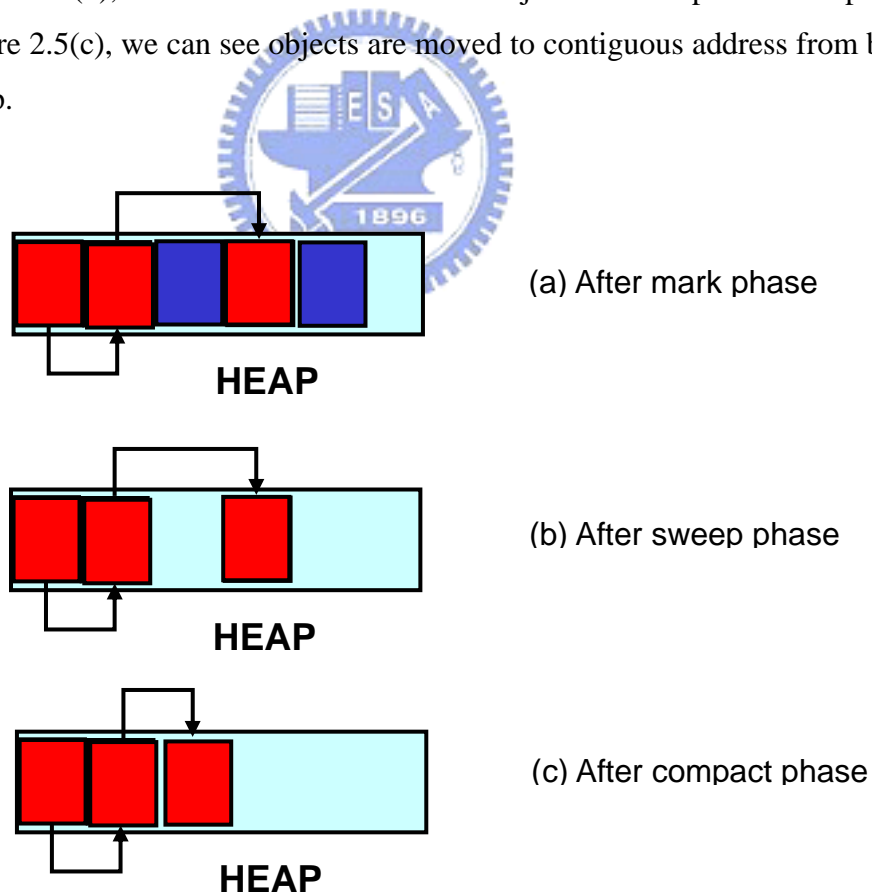
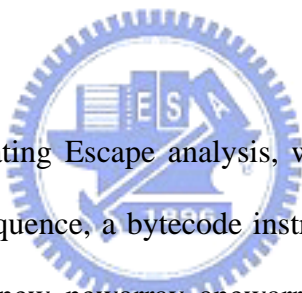


Figure 2.5: Garbage Collection

Garbage Collector helps Java Virtual Machine to sweep dead objects in heap, but also brings overheads in speed because of traversing reference tree and heap (traversing reference tree in mark phase, traversing heap in sweep and compact phase). So, if we can reduce frequency of Garbage Collection, we can reduce total execution time of Java program.

## 2.2 Escape Analysis

Before sweeping local objects upon method returning, we have to identify which objects are local ones. In current research, there is a kind of static analysis algorithm called Escape Analysis which can help us to find out Local Objects.[6][7]



Before illustrating Escape analysis, we have to define the term Allocation Site. In bytecode sequence, a bytecode instruction which will allocate new object (including bytecode new, newarray, anewarray, multianewarray in original JVM) is called Allocation Site (AS). And if an allocation site always allocates Local Object, then we call it a Local Allocation Site (LAS). It means that if an allocation site has any possibility to allocate non-Local Object, it is not a Local Allocation Site.

Escape Analysis is a static analysis algorithm. It statically analyzes the bytecode sequence in Class File to find out Local Allocation Sites in bytecode sequence.

To identify Local Allocation Site in byte code sequence, Escape Analysis will first do control flow analysis. Then it will traverse all control flow paths to see if an object from some allocation site will be assigned to global reference. If it will never

be returned or assigned to a global reference, it means that all references to it will be eliminated when method return. So, the object will become dead when method returns, too. Then we call the allocation site which allocate the object Local Allocation Site, because object allocated by it is always local. Global reference in Java Program including static reference, arguments, and reference field of global objects.

Escape Analysis can be done offline or after class loading, figure 2.6 shows the flow to do Escape Analysis. If we do Escape Analysis offline, we can store information of Local Allocation sites by (1) storing with annotation in class file or (2) replacing the Local Allocation site with other bytecode to point out Local Allocation Sites..



With Escape Analysis, we can find out Local Objects allocated by Local Allocation Sites. However, there are some objects which are not allocated by Local Allocation Site but still are Local Object. So, we have to note that Escape Analysis can not find out all Local Objects.

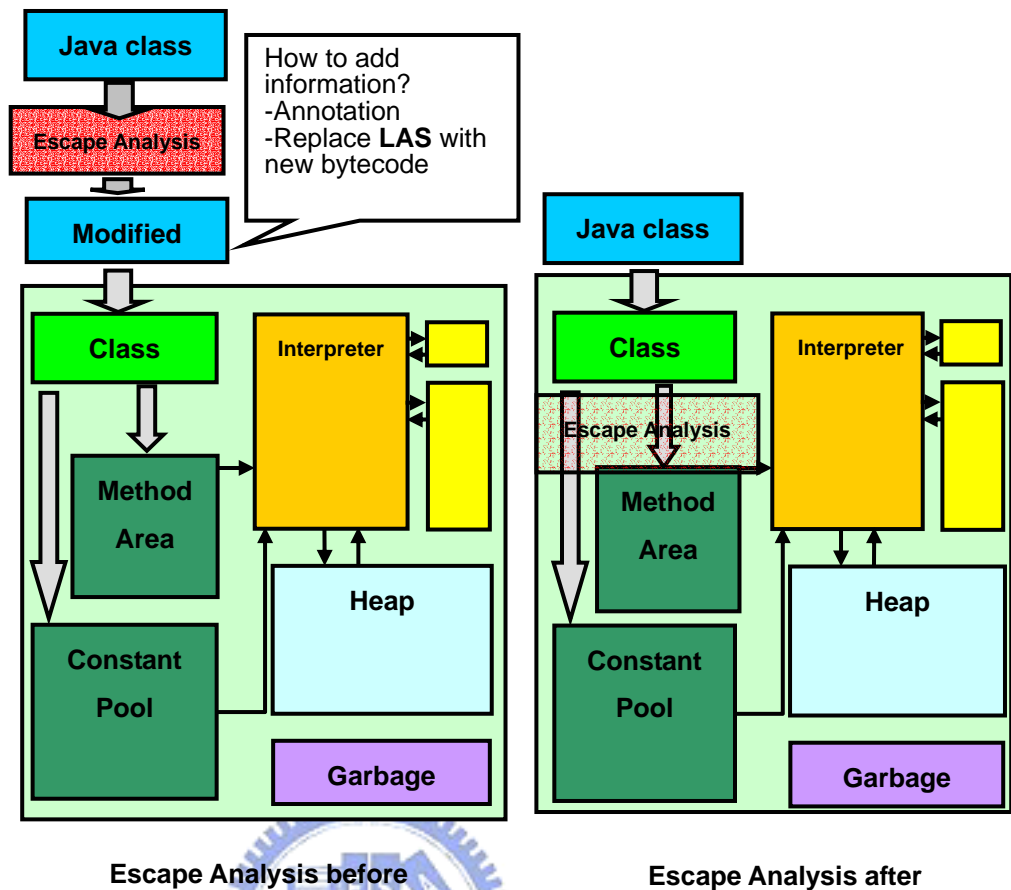
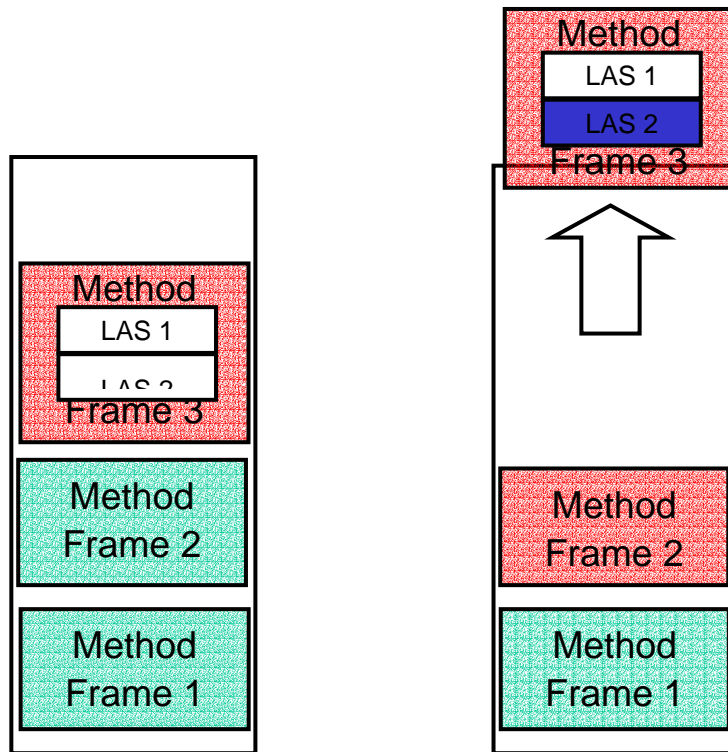


Figure 2.6: Escape analysis

## 2.3 Stack Allocation

After identifying Local Objects with Escape Analysis, then how can we free local objects upon method return? Current approach is Stack Allocation. [6][7] When method is invoked, JVM preserves contiguous space in the method frame for LASs in this method. Then, when LAS in method is executed, it allocates object in method frame but not heap. We can see figure 2.7. In figure 2.7 (a) spaces are preserved for Local Allocation site in frame. In Figure 2.7 (b) Local Objects are freed with popped frame.



**Java stack**

(a) Preserve space in frame for LAS

**Java stack**

(b) Local Objects in frame are freed with popped frame

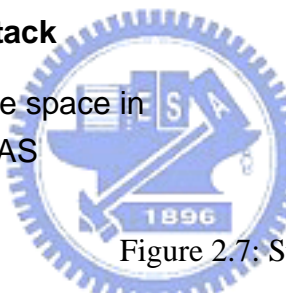


Figure 2.7: Stack Allocation

Then because Local Objects are allocated in method frame, when method returns all local Objects allocated in frame will be freed when the frame is popped. It should be noted that because reference recorded the real address in memory, and accessing to objects will be independent to storing in heap or stack. So, mechanism and time to access objects in heap and stack is the same.

However, because Frame is not collectable, if there are too many dead objects in it, the Java Virtual Machine will tend to throw StackOverflowError or OutOfMemoryError. So, if there is a Local Allocation Site which allocates lots of Local Objects in a method execution. We will only allocate the first one in frame, or it violates the space safety. For example, if there is a Local Allocation Site in loop, it will allocate a new Local Object in each iteration. However, the object may

be only live in scope of the iteration. Then, if we allocate each one in frame, because the object is not collectable, space complexity variants and space safety is violated. This constraint make Stack Allocation not be able to free all identified Local Objects upon method return.

## 2.4 Summary

In this Chapter, we introduce the basic concepts of Java Virtual Machine, Escape Analysis, and Stack Allocation. We also describe the constraint of Stack Allocation. Stack Allocation is constrained because objects in stack are not collectable. To overcome the constraint and free all identified Local Objects upon method return, I am motivated to design a mechanism to manage Local Objects in heap where objects in it is collectable. The details about my design will be presented in next chapter.

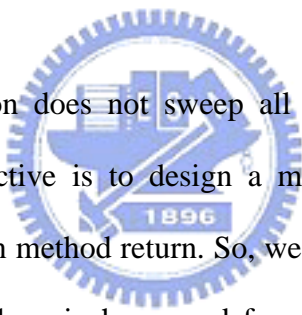


# Chapter 3

## Design

In this chapter, the mechanism to free Local Objects upon method returning is presented. Section 3.1 will introduce the overview of the whole design, section 3.2 to 3.3 shows two main issues in my design, section 3.4 will discuss about problems and solution about adding my design to original Java Virtual Machine, and section 3.5 will summarize whole my design.

### 3.1 Design Overview



Stack Allocation does not sweep all identified local objects upon method return. So my objective is to design a mechanism to free all identified Local Objects in heap upon method return. So, we will introduce a mechanism to specify local objects and others in heap, and free local objects upon method return. To manage Local Objects in heap, there are two main issues about my design (1) how to allocate and (2) how to free upon method return. And there is also a problem that how can garbage collector still work after adding our mechanism. So, we will discuss about these two issues and one problem in later section.

About issue 1, we will discuss about how to identify Local Objects and how to allocate Local Objects efficiently. In our design, Local Objects are identified by Escape Analysis, mentioned in chapter 2. So, we will not discuss about it in detail here. Discussion about allocating Local Objects efficiently including time to allocate and space to store local objects and management information.



About issue 2, we will discuss about how to know which objects are local in specific method upon return, and how to free Local Objects efficiently. Because method invocation and return is frequently in Java program execution, time to specify and free should be short to lower overhead of my design.

About problem, we will discuss about how can my design cooperate with the original Garbage Collector, we will show the modification of Garbage Collector. And we will discuss about additional overhead after adding modification, too.

## 3.2 Allocating Local Objects

To allocate Local Objects in heap, we have to know how Local Objects are stored in heap. So, we will introduce the data structure to store Local Objects in heap first. Then, we will illustrate the allocation policy when we allocating Local Objects or others. And we will discuss about overhead of each design after introducing them, too.

### 3.2.1 Data structure to store Local Objects in heap

To know how our mechanism allocates Local Objects, we have to know how Local Objects are stored in heap in my design. To store Local Objects in heap, we need a data structure to point out which objects in heap are local ones, and this structure must be variable size, because new Local Objects may be allocated anytime and we have to point out them, too. To satisfy the requirements, we add a linked list chunk in heap, which is called **Local Chunk List**.

Local Chunk List is a linked list data structure which links all Local Chunks

in heap. Each Local Chunk is a contiguous memory space that points out that the space is for Local Objects only. Figure 3.1 (a) is heap of original Java Virtual Machine, and Figure 3.1 (b) is heap after adopting my design.

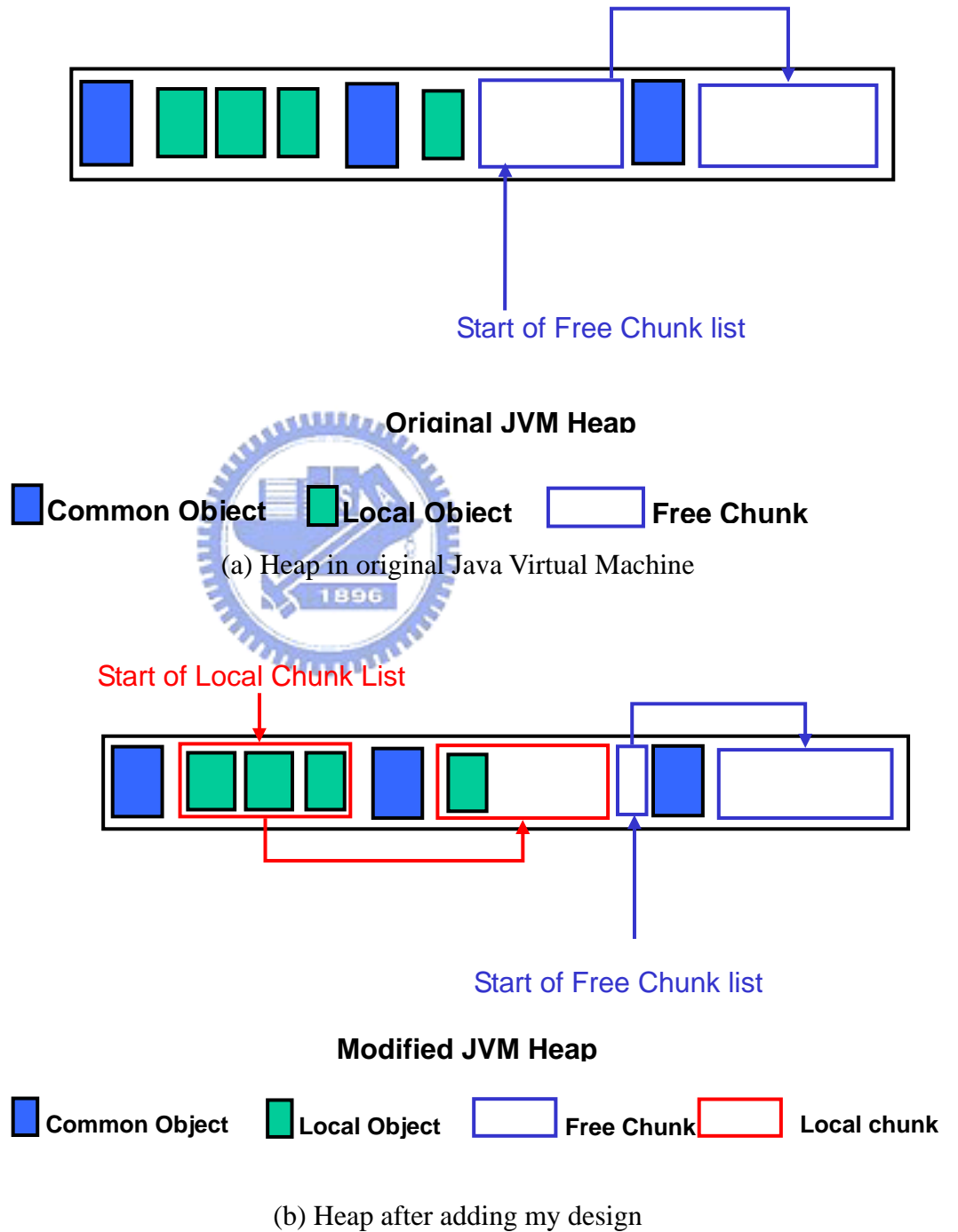


Figure 3.1: Data structure in heap in my design

The difference of Free Chunk and Local Chunk is that Free Chunk is a contiguous free space, and Local Chunk is a contiguous space that points out objects in this chunk is Local. But space in Local Chunk can be either allocated by Local Object or not. Data structure of Local Chunk is shown in Figure 3.2.



### Local Chunk

Figure 3.2: Data structure of Local Chunk

First three cells in local chunk are used to store information about Local Chunk List management. The field “Size” is size of this Local Chunk, the field “Size\_used” is how much space has been used in this chunk, and field “Next” maintains the address of next Local Chunk. The other space in the Local Chunk is used to store Local Objects.

Size of Local Chunk will not grow after Local Chunk allocation, because to increase size of Local Chunk is complex. To increase size of Local Chunk, Java Virtual Machine has to traverse the Free Chunk List to see if there is free space in end or begin of the Local Chunk. In some condition, there is no free space to extend size of Local Chunk, because the end of Local chunk has been allocated with some other object, and then we have to move the object to another address if we want to extend size of Local chunk. So, in our design, size of Local Chunk will not increase after allocation. However, compare with increasing size, decreasing size of Local chunk is much easier. Java Virtual Machine can just link the free space in end of Local Chunk to Free Chunk List, and reset the value of field “size”.

So, decreasing size of Local Chunk is allowed. We will decrease the size of Local Chunk in compact phase of Garbage Collection, which will be mentioned later.

After adding Local Chunk List, the heap is separate to Local Area and Other Area. Local Area includes Local Chunk List, and Other Area includes Free Chunk List and objects which are not in Local Area. So, Local Area is space for Local Objects, Other Area is space for objects which is not sure if local.

## Overhead

Then, we can discuss about the space overhead to use Local Chunk. First Overhead is space to store management information. For each Local Chunk, we need three cells to store management information, which is shown in Figure 3.2. So, the more Local Objects allocated in the Local Chunk, the less space overhead per Local Object brings. Second overhead in space is the internal fragmentation in Local Chunk. When we allocating Local Objects in Local Chunk, there may be some free space which is too small to allocate another Local Object. Then, it will lead to internal fragmentation in Local Chunk.

### 3.2.2 Policy to allocate Local Objects in heap

After introducing how Local Objects are stored in heap. We will see that how Java Virtual Machine allocate objects in my design. Initially, the size of Local Area is zero, which means Local Chunk List is empty. The basic principle of allocation is that if allocated object is local, we put it into Local Chunk. And if free space in Local Chunk is not enough when allocating, we will allocate a new Local Chunk from Free Chunk List for it. However, as mentioned before, the more Local

Objects in a Local Chunk, the less space overhead per Local Object brings. So, constant N bytes will be set to be the minimal size of local chunk when allocating new local chunk. And appropriate N will be discussed in section 4.4. However, if size of allocated Local Object is bigger than minimal Local Chunk allocation size N, then we will allocate a Local Chunk which is fit to size of the big Local Object, and link the allocated Local chunk to Local Chunk List. So, the allocation policy is:

- **Initially, size of local area is zero**
- **When a object not sure if local is going to be allocated**
  - Allocating it by traversing free chunk list, as original JVM
- **When local object is going to be allocated**
  - **If object size  $S > N$  bytes**, allocate a big enough S bytes local chunk for it
  - **Else if space from end of Local Objects to end of Local Chunk is enough to allocate it**
    - Allocating object in end of local objects
  - **Else**
    - Allocate and link an N bytes local chunk from free chunk list
    - Allocate local object in start of new local

We can see example in figure 3.3. Figure 3.3 (a) shows the initial state in heap. In Figure 3.3 (b) When a Local Object is to be allocated, we allocate a new Local Chunk for it. In figure 3.3(c), a Local Object is going to be allocated, and space is enough, so we allocate it right after the end of Local Objects in Local Chunk List. In figure 3.3(d), another Local Object is to be allocated, but space is not enough. So, we allocate a new Local Chunk and allocate the Local Object in it.

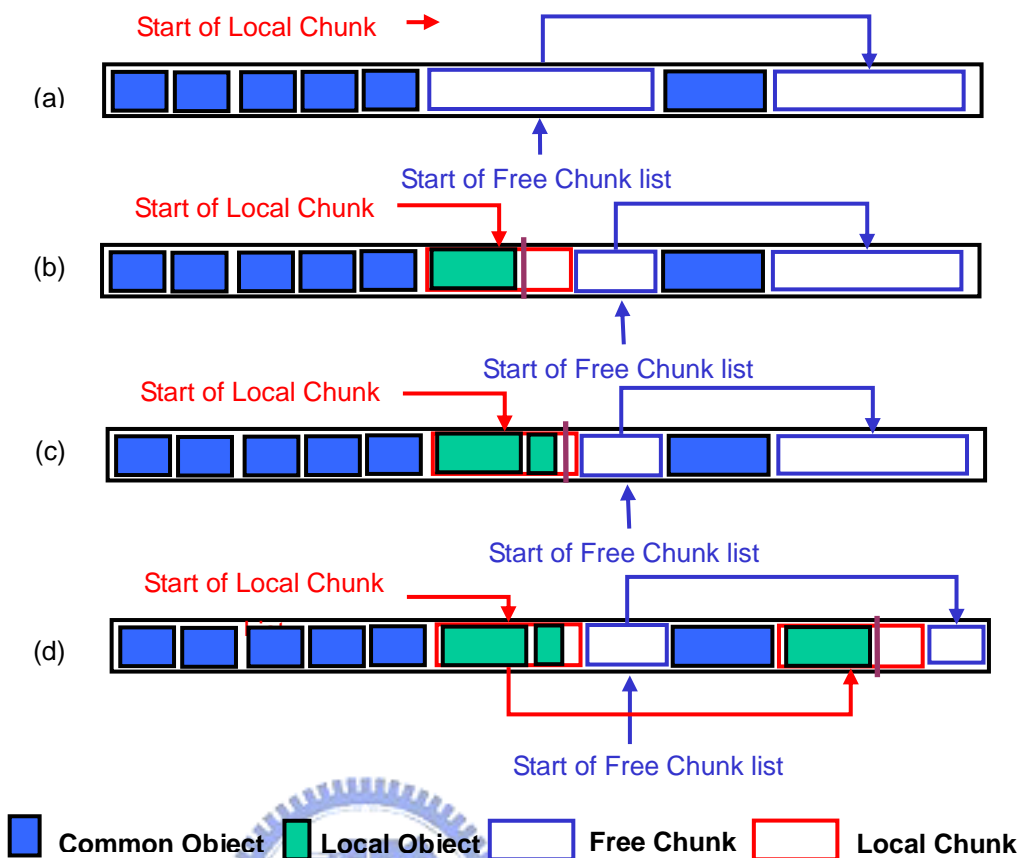


Figure 3.3: Allocating Local Objects

## Overhead

Speed overhead of allocation mechanism in my design is similar to allocation mechanism in original Java Virtual Machine. Because both of them use linked chunk list to allocate Local Objects in heap.

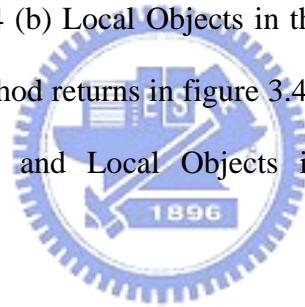
### 3.3 Freeing Local Objects upon method return

After discussing about Local Object allocation, let's focus on freeing Local Objects upon method returning. In this issue, we have to separate which Local Objects are in returning method and free them. And because method invocation and return in Java Program is frequent, speed overhead to free should be noted,

too.

In our design, we separate Local Objects from different method by recording the end of current end of Local Object in Local chunk List when method invocation. This recorded end of Local Objects is called Method Boundary, which will be stored in corresponding method frame. And when the method returns, the end of Local Objects in Local Chunk List will be set to value recorded when method invocation. Then, Local Objects in returning method are freed.

We can see example in figure 3.4. In Figure 3.4 (a), when method is invoked, we record the end of Local Objects in Local Chunk List to be Method Boundary in frame. In Figure 3.4 (b) Local Objects in the method is allocated in Local Chunk List. And when method returns in figure 3.4 (c), end of Local Objects is reset to be recorded boundary, and Local Objects in returning method become out of boundary and freed.



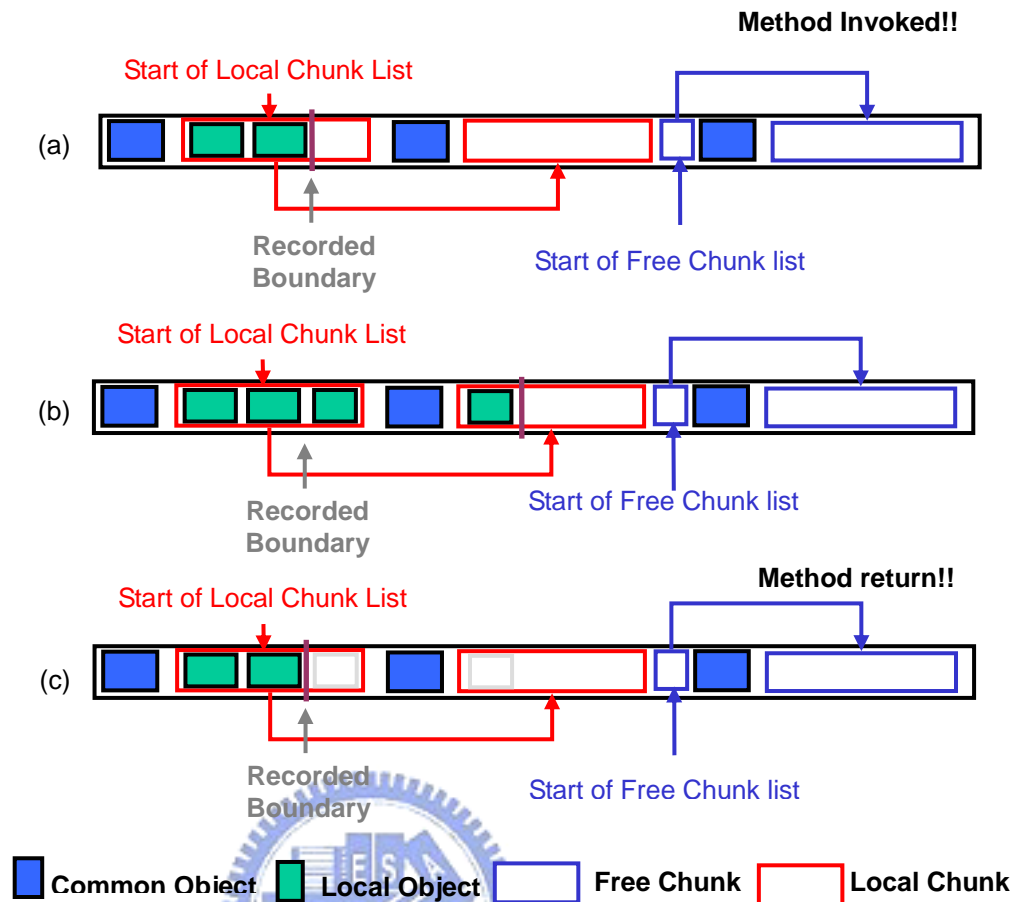


Figure 3.4: Freeing Local Objects upon method return

## Overhead

To discussing about the issue of speed overhead in freeing, we can see that we only record the address of end of Local Objects when method invocation, and reset the address when method return. Both of them are simple action and overhead will be light in each invocation and return. And total overhead of my mechanism in freeing will be direct proportion to numbers of method invocation and return in java Program.

## 3.4 Cooperation of Garbage Collector and my design



After adding my design, we have to consider about the cooperation of my design and original Garbage Collector. Because our design modifies the data structure in heap, we have to modify Garbage Collector to suit our data structure, and ensure the correctness of my design after Garbage Collection invoked. The main problem we will meet is that compact phase in Garbage Collection moves positions of objects in heap, then our data structure Local Chunk and method boundary will point to wrong address. So the boundary of method and Local Chunk must be adjusted, or the program sweep objects which are still alive when method returns, and result in error when executing program.

So, we will modify the Garbage Collector. In mark phase, it traverses the reference tree and marks the referenced objects, the same as original Garbage Collector. In sweep phase, the garbage collector will sweep dead objects, but not sweep Local Chunk containing live Local Objects. And in compact phase, because positions of objects are moved, we will adjust location and compact the size of Local Chunks. And Method Boundary in method frame must be adjusted, too. We can see figure 3.5 as example to compact. Figure 3.5 (a) is heap after sweep phase, and figure 3.5 (b) is heap after compact phase. We can see Local Chunk and method boundary are adjusted in figure 3.5 (b).

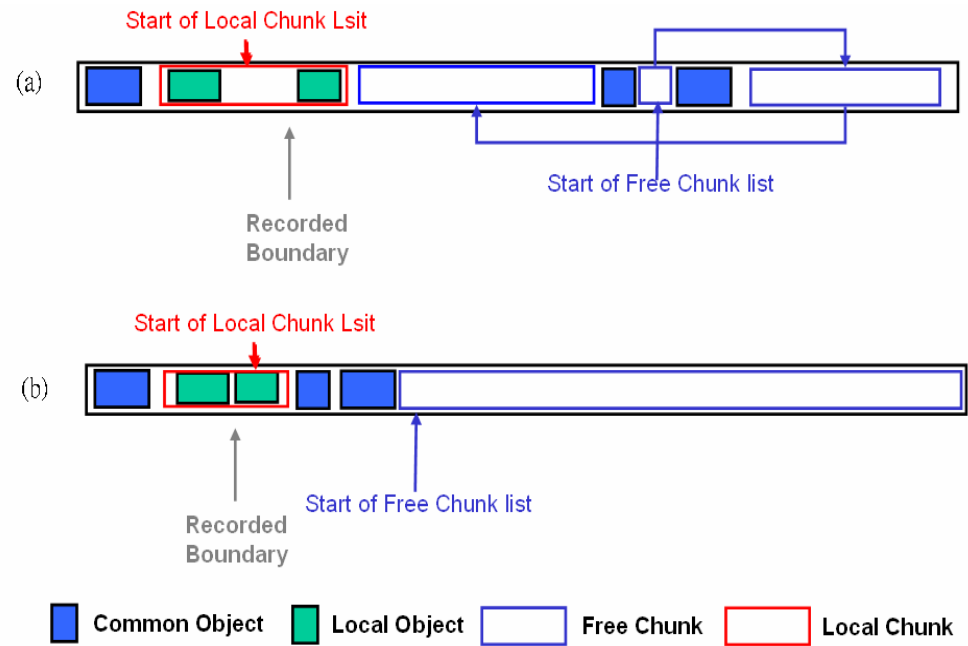


Figure 3.5: Compaction in modified Garbage Collector

## Overhead

In speed overhead issue, my modification on Garbage collector will result in more actions in compact phase, to adjust Local Chunk and method boundary. So, the overhead will be direct proportion to number of Local Chunks and depth of Java Stack, because number of method boundary is the same as number of method frames in stack..

## 3.5 Summary

In this chapter, we present the mechanism of our design. We add a Local Chunk List in heap to separate Local Objects from others. And when allocating local objects, we will contiguously allocate Local Objects in end of Local Objects in Local Chunk List if space is enough, or we will allocate a new Local Chunk for it. To free Local Objects in heap upon method return, we will record end of Local

Objects in Local Chunk List to be method boundary when method invoked, and reset it when method returns. The modified garbage collector will adjust Local Chunk and method boundary in compact phase.

Then, We will summarize my design by comparing my design with Stack Allocation, Table 3.1 shows the results.

Table 3.1: Comparing Stack Allocation with my design

Issues	Stack Allocation	My Design
<b>Multi local objects from single LAS</b>	Only one of them will be freed upon method return	All of them will be freed upon method return
<b>Garbage Collection</b>	Preserved space in frame can not be collected, even if objects in it is dead	Preserved Local Chunk can be collected, if all objects in the Local Chunk are dead
<b>Reducing fragmentation</b>	Local objects in the same method will be allocated continuously	Local objects allocated in the same method will not allocated contiguously in boundary of local chunks
<b>Overhead</b>	<ol style="list-style-type: none"> <li>1.Escape analysis (Optional)</li> <li>2.Allocating space for local objects when method frame is pushed</li> </ol>	<ol style="list-style-type: none"> <li>1.Escape analysis (Optional)</li> <li>2.Management of local area</li> <li>3.Garbage collection s</li> </ol>

Compare to stack allocation, there are four issues to be discussed. We show them in Table 3.1. Because of constraints of Stack Allocation, My design can allocate more identified Local Objects than Stack Allocation. An in Garbage Collection issue, Garbage Collector can collect Local Chunk which all Local

Object in it is dead in my design, but it can not collect preserved space in method frame even if there are no living Local Objects in it in Stack Allocation. And because Stack Allocation allocates Local Objects which will be freed in the same time in contiguous address, it reduces fragmentation. In my design, Local Objects are allocated contiguous in Local Chunk, too. But Local Objects in different Local Chunks are scatter. So, Stack Allocation can reduce more fragmentation than my design. Finally, the overhead in stack allocation is slight, because almost all computation can be processed offline. Overhead in my design is discussed in early section, and we will show how much it occupies in total execution time in next chapter.



# Chapter 4

## Simulation

In this chapter, we will evaluate my design with simulation. In section 4.1 we will introduce the evaluation equation. In section 4.2, Simulation Environment will be described. In section 4.3, the benchmark will be introduced. In section 4.4, the appropriate minimal size of local chunk when allocating new local chunk N will be discussed. In Section 4.5, simulation results of my design will be presented, and compared with original Java Virtual Machine and Stack Allocation.

### 4.1 Evaluation Equation

The evaluation overhead is show as bellow:

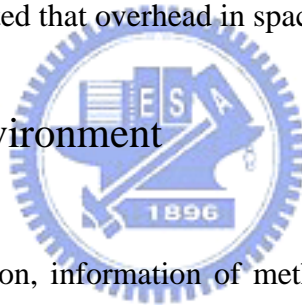


- Total Execution Time =  $N_{GC} * T_{GC} + (O_{Invoke\_total} + O_{Return\_total} + O_{compact\_total}) + T_{Unchanged}$ 
  - $N_{GC}$  : Number of GC
  - $T_{GC}$  : Average time of each Garbage Collection
  - $O_{Invoke\_total}$  : Extra overhead when method invoked
  - $O_{Return\_total}$  : Extra overhead when method return
  - $O_{compact\_total}$  : Extra overhead when method return
  - $T_{Unchanged}$  : execution time of unchanged components
  
- $O_{Invoke\_total} = N_{Invoke} * O_{Invoke}$ 
  - $N_{Invoke}$  : Number of method invocation
  - $O_{Invoke}$  : Extra overhead in each method invocation (constant)

- $O_{\text{Return\_total}} = N_{\text{Return}} * O_{\text{Return}}$ 
  - $N_{\text{Return}}$  : Number of method return
  - $O_{\text{Return}}$  : Extra overhead in each method Return (constant)
  
- $O_{\text{compact\_total}} = N_{\text{Adj\_boundary}} * O_{\text{Adj\_boundary}} + N_{\text{Adj\_LocalChunk}} * O_{\text{Adj\_LocalChunk}}$ 
  - $N_{\text{Adj\_boundary}}$  : Number of adjusted boundary
  - $O_{\text{Adj\_boundary}}$  : Overhead to adjust each boundary
  - $N_{\text{Adj\_LocalChunk}}$  : Number of adjusted Local Chunk
  - $O_{\text{Adj\_LocalChunk}}$  : Overhead to adjust each Local Chunk

It should be noted that overhead in space influences  $N_{GC}$  in simulation.

## 4.2 Simulation Environment



In our simulation, information of method invocation and return and objects life time is recorded by executing benchmark with modified KVM CLDC 1.1. The KVM is an embedded Java Virtual Machine produced by SUN. We modify it to do excessively Garbage Collection whenever (1) method returns and invocation (2) object allocation to get the more accurate life time of objects.

Then, we run Arm Develop Suite 1.2 to simulate total execution time and each overhead parameter in my design in equation in section 4.1. Values of parameters in my simulation are:

- $T_{GC}$  : 100467 cycles
- $O_{\text{Invoke}}$  : 4 cycles
- $O_{\text{Return}}$  : 4 cycles

- $O_{Adj\_boundary}$  : 35 cycles
- $O_{Adj\_LocalChunk}$  : 83 cycles
- $T_{Unchanged}$  : 244053153 cycles

Finally, We use information get by Modified KVM and ADS 1.2 to simulate the management behavior in heap and total execution time in my design. Management behavior includes allocation, freeing local objects upon method return, and garbage collection. The results will be presented in section 4.5.

### 4.3 Benchmark

Embedded CaffeineMark < <http://www.benchmarkhq.ru/cm30/info.html> > is a typical bench mark to test performance of embedded Java Virtual Machine, for example KVM CLDC 1.1. Because my design is focus on memory constrained system, we adopt it to be my benchmark. The following is a briefly description of what embedded CaffeineMark do:

- **Sieve**

The classic sieve of Eratosthenes finds prime numbers.

- **Loop**

The loop test uses sorting and sequence generation as to measure compiler optimization of loops.

- **Logic**

Tests the speed with which the virtual machine executes decision-making instructions.

- **Method**

The method test executes recursive function calls to see how well the VM handles method calls.

– **String**

String comparison and concatenation

#### 4.4 Discussion about minimal size of Local Chunk when allocating new Local chunk

The minimal size  $N$  of Local Chunk when allocating new Local Chunk influences our results in simulation. If  $N$  is too small, it will lead to (1) more overhead in management space and (2) fragmentation in heap. When number of Local Objects is the same, if Local Chunks are small ones, it needs more Local Chunks to maintain them and bring more overhead in management space. And because Local Objects in the same method will be freed at the same time, if chunk is small, it means objects which will be freed at the same time will scattered around the heap. Then fragmentation problem in heap will be more serious than sequential allocation.

However, if  $N$  is too big, problems will be (1) Allocating new chunk tends to invoke Garbage Collection and (2) Occupying lots of free space results in that common object can not be allocated.

So an appropriate  $N$  should be discussed, we can see figure 4.1. Because overhead in speed of each Garbage Collection is much bigger than each other overhead, so we will consider times of Garbage Collection first.



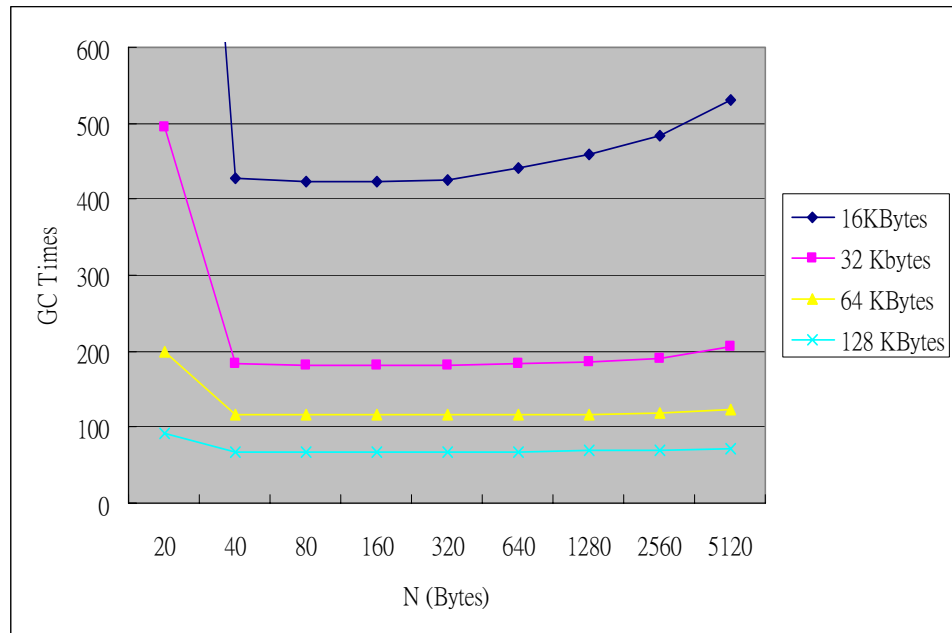


Figure 4.1: Times of Garbage Collection with different N

We can find that when N is between 80 bytes to 320 bytes, number of Garbage collection is similar. However, the bigger N, the less times to allocate Local Chunks for the same number of Local Objects. So we adopt N to be 320 bytes in our simulation.

## 4.5 Simulation results

### 4.5.1 Ratio of GC time to total execution time

First we will show the ratio of Garbage Collection time to total execution in different heap size. It shows how much opportunity we can improve. Figure 4.2 is the ratio is different heap size in original Java Virtual Machine. The heap size is 16 to 128 Kbytes, because the minimal heap size of KVM is 16 Kbytes, and the default heap size in CLDL 1.1 is 128 Kbytes.

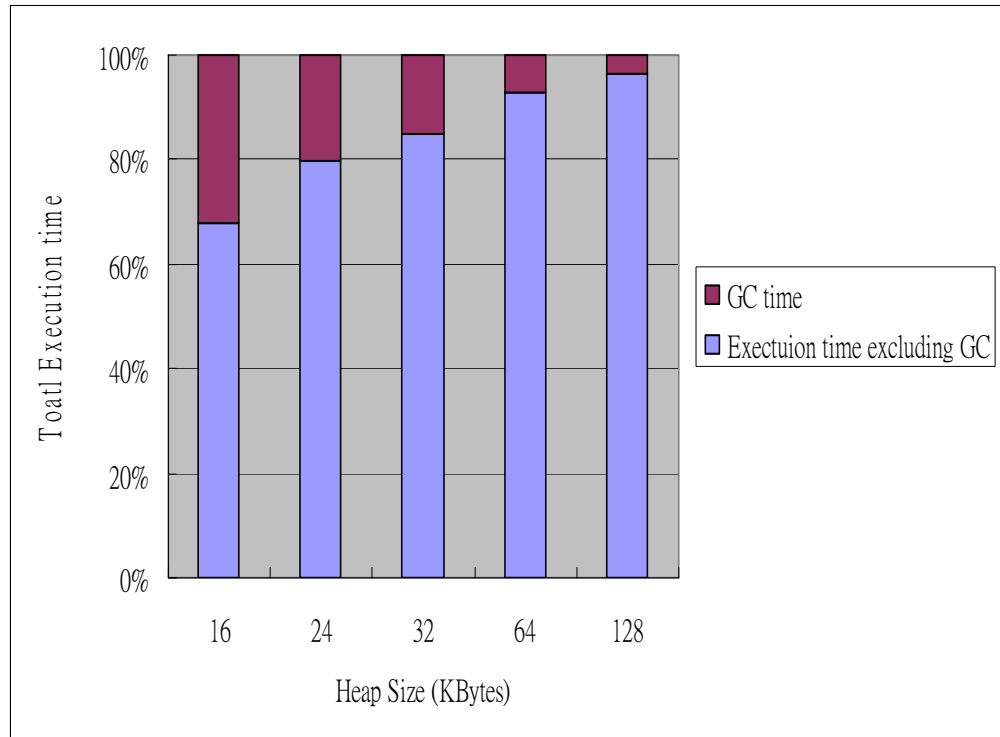


Figure 4.2: Ratio of Garbage Collection time to total execution time

We can see that the Garbage Collection occupies about 32% of total execution time when heap size is 16Kbytes. However, when heap size is 128 Kbytes, the ratio of Garbage Collection decreases to about 4%. It means that our design have more opportunities to improve speed performance in memory constrained system.

#### 4.5.2 Times of Garbage Collection in different design

Number of garbage collection invoked in different design is presented here. In this chart, our design adopt 320 Kbytes to be the minimal size N of Local Chunk when allocating new Local Chunk, as discussed in section 4.4.

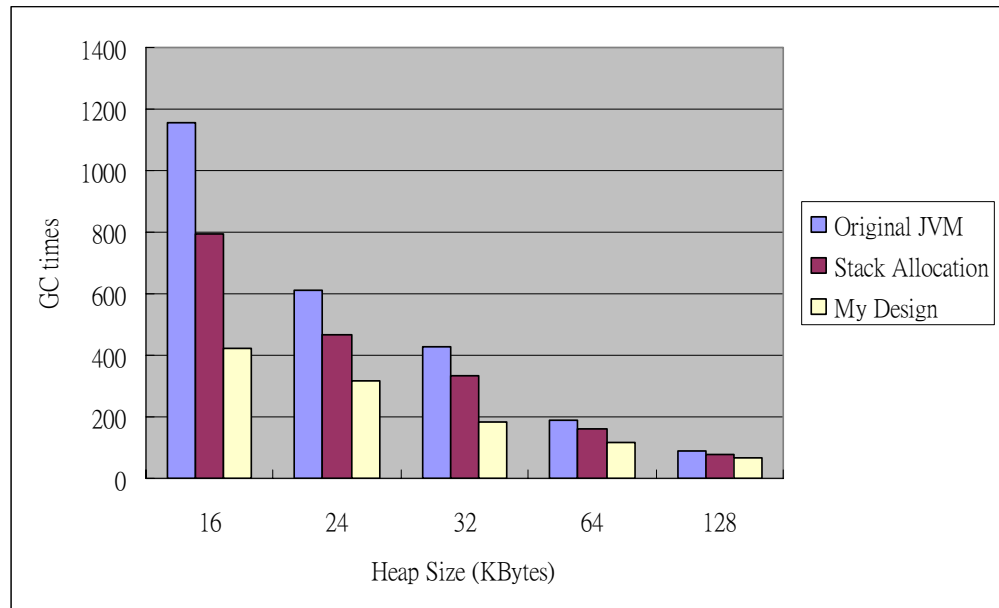
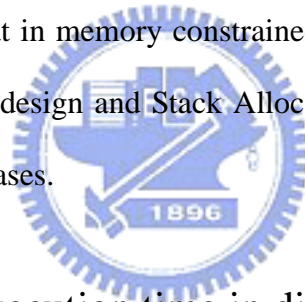


Figure 4.3: Times of garbage collection in different design

We can see that in memory constrained environment, my design surely works better than original design and Stack Allocation. However, with heap size grows, the difference decreases.



### 4.5.3 Total execution time in different design

Finally, total execution time (including GC, pure execution, and overhead) is presented. N is 320 Kbytes, too.

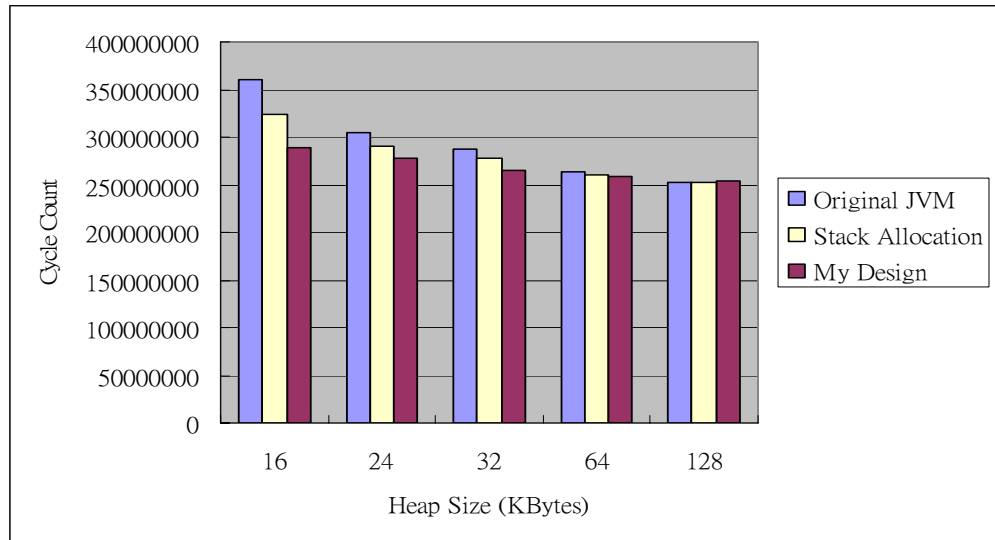


Figure 4.4: Total execution time in different design

We can see that in memory constrained environment, we can improve speed about 11%, even including overhead in speed. However, when the heap size is 128 Kbytes, the overhead will become bigger than the improvement in my design. The total execution time in my design is even worse than original Java Virtual Machine and Stack Allocation when heap size is 128 Kbytes

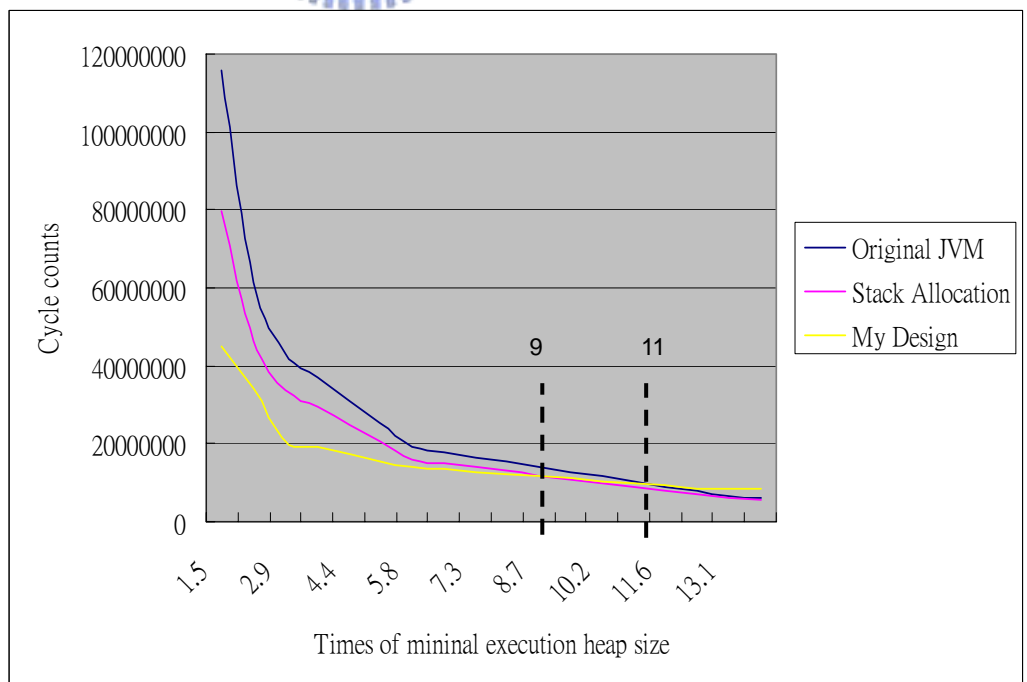


Figure 4.5: (GC time + overhead in speed) in different design

After discussion about total execution time in different design, we know that when size of memory is large, my design will become useless, or even harmful to speed performance. So, let us see the range of heap size which we should adopt our design. Figure 4.5 is (GC time + overhead in speed) in different design. (GC time + overhead in speed) is (total execution time – unchanged execution time), which unchanged execution time is the pure execution time (excluding GC) in original Java virtual machine. And the minimal execution heap size without error of our benchmark is 11 Kbytes. It helps us to see the curve of execution time with different heap size in different mechanism clearly. We can see that we become worse than original Java Virtual Machine when heap size is bigger than 11 times of minimal execution heap size. And it becomes worse than Stack Allocation when bigger than 9 times of minimal execution heap size.

#### 4.5.4 Overheads

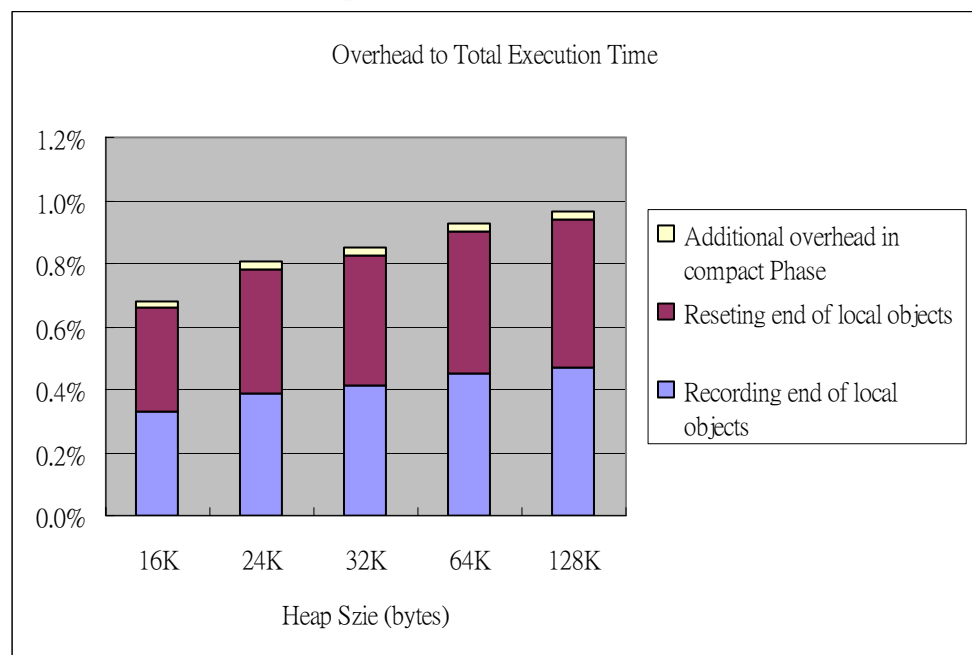
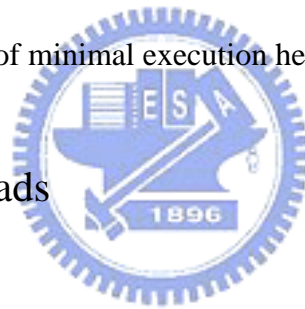


Figure 4.6: Ratio of overhead to total execution time in my design

Then let us discuss about additional overhead when adopting my design. We can see figure 4.6, in my simulation, the overhead is 0.6% ~ 0.9%. And about 95% of overhead is to record and reset end of local objects in local chunk list when method invocation and return. Only 5% of additional overhead is because the modification of garbage collector in compact phase. It occupies more little portion in memory constrained environment because the total execution time in memory constrained environment is larger. But amount of overhead in different heap size is similar because most portion of it is to record and reset end of local objects in local chunk list when method invocation and return, and number of method invocation and return is invariant to heap size. It means that overhead is invariant to heap size. So, the more we can improve in reducing frequency of Garbage Collection, the more we can improve in total execution time.

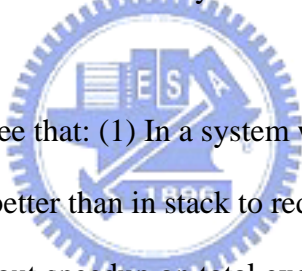


# Chapter 5

## Conclusions

In this thesis, a mechanism to free all identified local objects in heap is purposed and evaluated.

First, we propose the mechanism to allocating Local Objects in heap but not stack to avoid the constraint in current approach and then discuss about meted problems when we add it to original Java Virtual machine. Finally, we evaluate my design with simulation and make some conclusions by observing results in my simulation..



As a result, we can see that: (1) In a system with memory constraints, allocating Local Objects in heap is better than in stack to reduce frequency of garbage collection. It brings improvement about speedup on total execution time. (2) If heap is large enough, my design have less chance to improve the speed performance. Moreover, overhead of my design will make modified JVM even slower than the original JVM and Stack Allocation mechanism.

So, my design is suitable for memory constrained system. In our simulation, in a memory constrained environment, my design leads to 11% speedup over original Java Virtual Machine and reduce 60% of Garbage Collection in vocation counts, and 7% speedup over Stack Allocation on total execution time.

However, if heap is large, my design becomes less useful than memory constrained environment, or even harmful to speed performance. But what is the range of size of heap which is appropriate to adopt my design? In our simulation, we can see that if heap size is less than 11 times of minimal execution heap size, my design is better than original Java Virtual Machine in speed issue. If heap size is less than 9 times of minimal execution heap size, my design is better than Stack Allocation in speed issue. So, we can know when we should adopt our design.

And in overhead issue, it shows that overhead of my design is about 0.6% ~ 0.9% of total execution time, which varies because the variation of total execution time. And most portion of my overhead is to record and reset end of local objects in local chunk list, when method invocation and return. So, we can focus on it if we want to reduce the overhead.



In future work, to make our design suitable for all system but not only memory constrained system, we can design a mechanism to dynamically profile the ratio of garbage collection time to execution time to figure out whether JVM will adopt my design in runtime. It can help to turn off my design to reduce overhead when there are not too opportunities to improve. However, the overhead of this mechanism should be considered, too. And the threshold to turn on or turn off my design is also an issue should be discussed.

Besides freeing Local Objects in heap upon method return to reduce overhead of Garbage Collection., allocating Local Objects in the same method contiguously may also be helpful in locality in environment which has cache system because objects in the same method tend to be used together. So, it can bring some benefits in locality to



access objects in heap because data may had been cached. This issue can be a further research to be discussed, too.



# References

- [1] C. E. McDowell\* , “Reducing garbage in Java” Volume 33 , Issue 9 (September 1998) , Pages: 84 - 86 ,Year of Publication: 1998
- [2] Tim Lindholm, Frank Yellin “The Java™ Virtual Machine Specification”. [http://  
http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html](http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html)
- [3] Sun Microsystems, Inc , “J2Me Building Blocks for Mobile Devices – White Paper on KVM and the Connected, Limited Device Configuration (CLDC)” 901 San Antonio Road, Palo Alto, CA 94303 USA, 650 960-1300 fax 650 969-9131, May 19, 2000
- [4] 探矽工作室著, 深入嵌入式 Java 虛擬機器 Inside KVM 學貫行銷股份有限公司出版, 2002[民 91]
- [5] Joshua Engel, Tim Lindholm, Java Virtual Machine, 1 edition, O'Reilly; (April 1, 1997)
- [6] Erik Corry, “Stack Allocation for Object-Oriented Languages” Daimi University of Aarhus, Denmark [corry@daimi.au.dk](mailto:corry@daimi.au.dk) ,May 24, 2004
- [7] David Gay, Bjarne Steensgaard, ”Stack Allocating Objects in Java”, Microsoft Technical Report, November 1998