Runtime Escape Analysis in a Java Virtual Machine

by

Manfred Jendrosch

Bachelor of Science in Computer Science, Bonn-Rhein-Sieg University of Applied Sciences, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Gerhard W. Dueck, Prof., Computer Science
	André Hinkenjann, Prof. Dr., Computer Science
Examining Board:	Kenneth B. Kent, Prof., Computer Science, Chair
	David Bremner, Prof., Computer Science
	Richard J. Tervo, Prof., Electrical and Computer Engineering

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

May, 2013

Abstract

The Java Virtual Machine (JVM) executes the compiled bytecode version of a Java program and acts as a layer between the program and the operating system. The JVM provides additional features such as *Process*, *Thread*, and *Memory Management* to manage the execution of these programs. The *Garbage Collection (GC)* is part of the memory management and has an impact on the overall runtime performance because it is responsible for removing dead objects from the heap. Currently, the execution of a program needs to be halted during every GC run. The problem of this stop-the-world approach is that all threads in the JVM need to be suspended. It would be desirable to have a thread-local GC that only blocks the current thread and does not affect any other threads. In particular, this would improve the execution of multi-threaded Java programs.

An object that is accessible by more than one thread is called *escaped*. It is not possible to thread-locally determine if escaped objects are still alive so that they cannot be handled in a thread-local GC. To gain significant

performance improvements with a thread-local GC, it is therefore necessary to determine if it is possible to reliably predict if a given object will escape.

Experimental results show that the escaping of objects can be predicted with high accuracy based on the line of code the object was allocated from. A thread-local GC was developed to minimize the number of stop-the-world GCs. The prototype implementation delivers a proof-of-concept that shows that this goal can be achieved in certain scenarios.

Acknowledgements

I would like to thank my supervisors Gerhard W. Dueck and André Hinkenjann for their ongoing support and collaboration during my entire Dual Degree Master study. A special thanks goes to Charlie Gracie and Karl Taylor from the IBM Ottawa Lab for their assistance during all phases of the project.

Furthermore I would like to thank the internal readers Kenneth B. Kent and David Bremner as well as the external reader Richard J. Tervo for their useful comments and input to my thesis.

My parents — Hans-Werner and Sabine Jendrosch — merit a special thanks for their moral and financial support during my whole Bachelor and Master studies. Without their assistance I would not have been able to enrol in the Dual Degree Master Program and going abroad to Canada. Thank you very much!

The funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program and IBM is gratefully acknowledged.

I would like to thank Rainer Herpers and Nadine Fröbel from the *Bonn-Rhein-Sieg University of Applied Sciences* and Jodi O'Neill from the *University of New Brunswick* for their organizational support during the Dual Degree Program. I additionally acknowledge the funding support received by the *Bonn-Rhein-Sieg University of Applied Sciences* through the Transatlantic Exchange Partnership Program.

Last but not least, I would like to thank all my friends and family that supported my intention to study in a Dual Degree Master Program both at the *Bonn-Rhein-Sieg University of Applied Sciences* in Sankt Augustin, Germany and the *University of New Brunswick* in Fredericton, Canada.

Table of Contents

List of Tables

List of Figures

List of Symbols, Nomenclature or Abbreviations

AES	Advanced Encryption Standard
BOPS	Business Operations Per Second
DSA	Digital Signature Algorithm
DES	Data Encryption Standard
EM	Escape Marking
GC	Garbage Collection
GGC	Global Garbage Collection
GMP	Global Mark Phase
IDE	Integrated Development Environment
JavaEE	Java Platform, Enterprise Edition
JAR	Java Archive File
JDK	Java Development Kit
JIT	Just-In-Time Compiler
JNI	Java Native Interface

JRE	Java Runtime Environment
JVM	Java Virtual Machine
KB	KiloByte
LIFO	Last In First Out
MB	MegaByte
MD5	Message-Digest 5 Algorithm
MPEG	Moving Picture Experts Group
MP3	Encoding format for digital audio
NIST	National Institute of Standards and Technology
PC	Program Counter
PGC	Partial Garbage Collection
RSA	Encryption algorithm invented by Rivest, Shamir & Adleman
R&D	Research & Development Project
SBGC	Stack Based Garbage Collection
SHA1	Secure Hash Algorithm 1
STW	Stop-The-World
TLH	Thread Local Heap
TLR	Thread Local Region
TLRGC	Thread Local Region Garbage Collection
VM	Virtual Machine
XML	Extensible Markup Language

Chapter 1

Introduction

Java is a platform independent object-oriented programming language and has become an integral part of every-day computing. It is the most popular programming language according to the *TIOBE Programming Community Index for February 2013* [1]. The long term trends of the Top 10 programming languages in the last 10 years are depicted in Figure 1.1.

The source code of Java programs is compiled into bytecode that is then executed by a Java Virtual Machine (JVM), which is built as a layer between the running program and the operating system. The JVM provides additional features such as *Process*, *Thread*, and *Memory Management* to manage the execution of Java programs. The *Garbage Collection (GC)* is part of the memory management and has an impact on the overall runtime performance because it is responsible for removing dead objects from the heap. The exe-

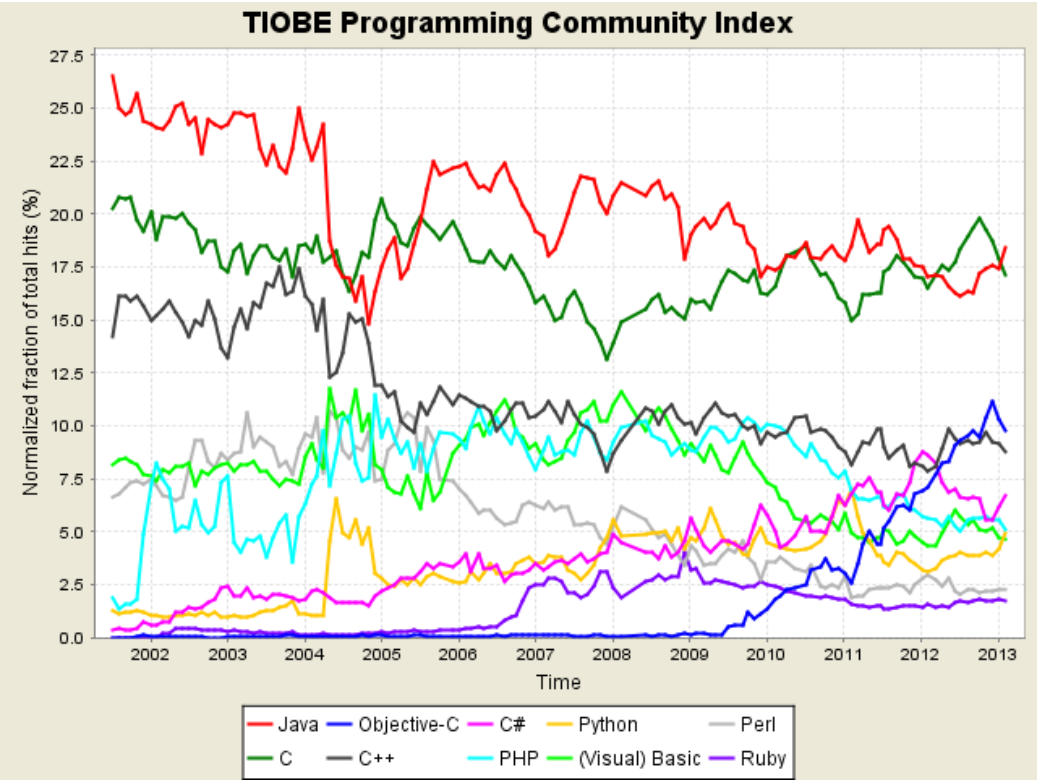


Figure 1.1: TIOBE Programming Community Index - Top 10 long term trends [1].

cution of a program needs to be interrupted during a GC, so it is important to keep the GC times as short as possible. Garbage collection policies use different approaches to achieve this goal and some of them will be discussed in the background chapter.

The work performed for this thesis is part of the Garbage Collection research project at the *IBM Centre of Advanced Studies Atlantic* located at the *University of New Brunswick* in Fredericton. The main goal of this project is

to reduce the amount and time spent in stop-the-world garbage collections, which means that the execution of all threads is suspended during a GC. All currently available JVM GC policies are stop-the-world, so that reducing the GC times would improve Java computing in highly multi-threaded environments. But even a Java end user would benefit from these improvements, because almost every computer nowadays contains a CPU with at least 4 cores enabling the multi-threaded execution of programs.

The thesis addresses the stop-the-world issue with the idea of a Thread Local Region (TLR) allocation and garbage collection. TLRs are special regions on the heap exclusively assigned to one particular thread. If these regions only contain thread-local objects, a thread-local garbage collection can be performed without influencing any other threads. Additionally, TLRs can be moved to the *freeRegions* list at thread death without performing GC.

Unfortunately not all objects in a TLR are thread-local objects. An object that is accessible by at least one other thread is called ***escaped***. A TLR object needs to be copied out to the global heap as soon as it escapes. This copy operation is expensive because all TLR objects need to be traversed to update the references to the object. It is therefore essential to analyze how many objects escape and determine if it is possible to reliably predict the escaping of objects.

The escape analysis part of this thesis is based on the JVM instrumentation and class based escape counting realized in the previous Research & Development project (R&D). The R&D approach stored the escaping of objects based on the class of the allocated Java object. The newly introduced approach in this thesis is based on the *Program Counter (PC)* as the distinguishing feature, which is a unique identifier for the line of code an object was allocated from. This new PC based approach led to a better differentiation between the escaped and non-escaped objects than with the old class based approach. Depending on the benchmark, the results show that the PC based approach can identify 50-80% of the allocated objects as non-escaping. Additionally, it is possible to identify about twice as many objects that never escape based on the program counter than based on the class.

The second part of this thesis is the implementation of a prototype for the TLR allocation and garbage collection to show the feasibility of a thread-local allocation. It relies on the assumption that a significant amount of non-escaping objects allocated by a thread are short-living and can immediately be reclaimed by the TLR garbage collection. The results show that 97-100% of the non-escaped objects and about 60% of all objects in the TLRs of a thread are already dead when a TLR garbage collection is triggered. Additionally, the prototype achieves the goal of reducing the number and runtime of the stop-the-world GCs in certain scenarios.

Chapter 2

Background

To understand the concept of the TLR allocation and garbage collection, it is important to know the basics of the Java Virtual Machine and its Memory Management. After an introduction into these basics, different garbage collection approaches, including the TLR garbage collection, will be presented in this chapter. Last but not least, an introduction into the idea of escape analysis is given.

2.1 Java Virtual Machine

The Java Virtual Machine (JVM) is defined in the Java Virtual Machine Specification [6] and refers to an abstract computing machine. Several vendors like Sun Microsystems (now aquired by Oracle) [7], Oracle [8] and IBM [9] have their own JVM implementation according to the specification. The

specification defines the following three basic parts [2] [6]:

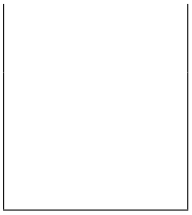
- A set of instructions and the definition of their meaning. The combination of these instructions is called *bytecode*.
- The *class file format*: Definition of a binary format to combine *bytecodes* and other class related information in a platform independent way that can be provided as a stream or file on disk.
- A *verification* algorithm to identify programs that can compromise the integrity of the JVM and avoid running them.

2.1.1 Instruction Set

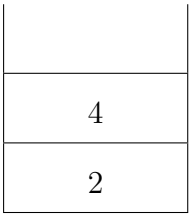
All executable code for the JVM is expressed with *bytecode* that contains multiple instructions. Here is a simple example of *bytecode* operations:

- **PUSH 2:** Push number 2 on the stack.
- **PUSH 4:** Push number 4 on the stack.
- **ADD:** Add the top two elements from the stack together and place the result back on the stack.

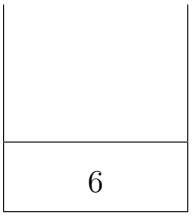
Initially, at the start of the program, the operand stack is empty:



Lets assume that the three example instructions are executed after each other. In the first instruction the number 2 is added on the stack, followed by number 4.



The instruction ADD pops the last two entries from the stack, adds them together and pushes the result back on the stack. For the example the stack will look like this after the end of the execution:



The *bytecode* instructions make programs written in Java platform independent, because every Java compiler just translates the Java program into the standardized *bytecode* and not directly into native machine code. The Java Virtual Machine interprets these *bytecode* programs at runtime, so that only the JVM needs to be compiled corresponding to the underlying architecture [2].

2.1.2 Class file format

Java programs are defined in a class file format, which is a binary format that represents the Java class. This format is just a stream of bytes that need not necessarily be stored in a file. The JVM provides mechanisms — the so called ClassLoaders — to convert these class files into classes. The ClassLoaders can load classes from the file system, from a database, from a JAR file or over the internet [2] [6].

2.1.3 Verification

The Java Virtual Machine has a verification algorithm to avoid tampering. Every loaded class will be checked, if it is working according to the specification to protect the security of the JVM. There are several ways a program can try to impact the stability and security of the JVM such as overflowing the stack to access or corrupt memory not related to them or casting an object inappropriately to access forbidden memory spots. Not every security breach can be avoided through this, but it adds an additional security level and can avoid some exploitations [2].

2.2 Memory Management in the JVM

The memory in a JVM is divided into four conceptual data spaces [2]:

- *Class Area*, where the source code and the constants are kept.

- *Java Stack*, which keeps track of the called methods and their invocation status.
- *Heap*, where objects are stored.
- *Native Method Stacks*, to support native method calls.

2.2.1 Class Area

The Class Area is responsible for storing the classes loaded into the system. Method implementations are kept in the *method area* while constants are stored in a *constant pool*. The class definitions loaded into the system are immutable, so that a certain class will stay unchanged during the whole runtime. The class definitions held in memory are also used as templates for new objects that will be stored on the heap.

Java Classes have several properties that are loaded into the system:

- Superclass
- List of interfaces (if available)
- List of fields
- List of methods and their implementation (stored in the method area)
- List of constants (stored in the constant pool)

All these properties are part of the fixed class definition and therefore are also immutable. This contributes to the stability of the JVM because every object of a particular class is always referring to the same definition [2].

2.2.2 Java Stack

The Java Virtual Machine creates a *stack frame* for each method invocation. All stack frames together build the *stack* of a single thread that works according to the last-in-first-out (LIFO) principle. The stack frame on top of the stack is called the *active stack frame*.

Every stack frame has an operand stack, an array of local variables and a pointer to the currently executed instruction. This pointer is called *program counter* (PC) and points into the method area. In most cases the program counter is moving continuously during the execution but due to some special instructions (e.g. jumps via goto) this is not always the case. The active stack frame is always reflecting the current execution and therefore only the operand stack and the local variable array of the active stack frame are accessible by the thread.

As soon as a method invocation occurs, a new stack frame with a separate program counter variable is created and pushed on the stack. After the method has been successfully executed, the stack frame is popped from the

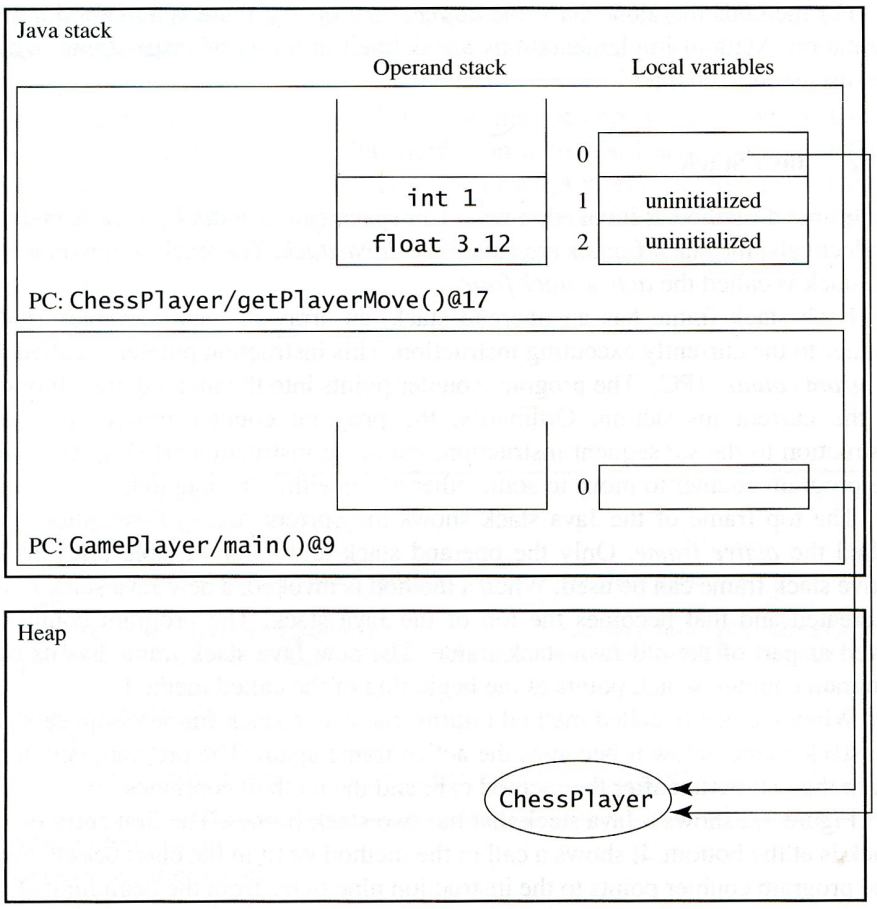


Figure 2.1: Example heap and stack during execution [2].

stack and the execution continues at the program counter of the underlying method whose stack frame now became the active one again. As soon as a thread dies, its whole stack can be deleted because it is not needed anymore.

Figure 2.1 shows an example of the stack and heap of a thread during execution. The stack currently contains two stack frames: The *GamePlayer.main()* method at PC 9 and above the *ChessPlayer.getPlayerMove()* method at PC

17. This means that the *getPlayerMove()* method was called from the *main()* method, which is suspended until the corresponding stack frame will become the active stack frame again. Both stack frames additionally hold a reference to the heap object *ChessPlayer* in their local variable array, but only the *getPlayerMove()* operand stack and variable array are currently accessible by the thread [2].

2.2.3 Heap

The *heap* is a memory storage space where objects can be allocated wherever free space is available. Objects on the heap may be referenced by pointers either from a stack frame and/or from one or multiple objects on the heap. Every object on the heap is associated with a class from the class area and holds memory space according to the class definition.

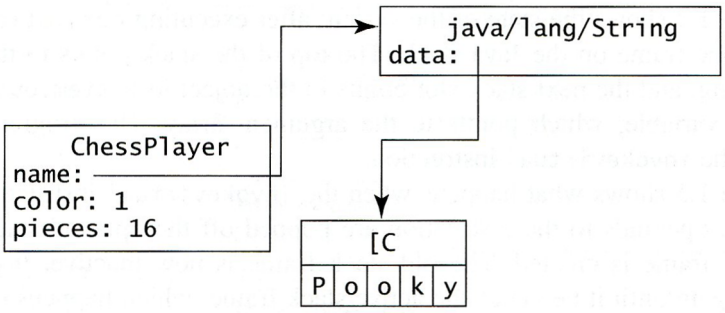


Figure 2.2: Example heap content [2].

Figure 2.2 shows an example of a simple object structure stored on the heap.

The class *ChessPlayer* contains three variables: *name*, *color* and *pieces*. While *color* and *pieces* are native *Integer* objects and directly stored into the object, the variable *name* is a reference to a *java/lang/String* object. All *String* objects in Java store their data in an internal char data array, which results to an additional reference to the char array with the characters *"Pooky"* (denoted with the descriptor [C in Figure 2.2) [2].

2.2.4 Native Method Stacks

Beside normal Java methods, the JVM provides the possibility to execute native methods. These methods are similar to JVM methods, but implemented in another programming language. This allows the developer to handle certain situations that either cannot be handled in Java at all or are difficult to realize. The JVM provides conventional stacks ("C stacks") to allow these native methods to keep track of their own state. All current JVM implementations provide the standardized Java Native Interface (JNI) to handle the integration of non Java code into the JVM [2].

2.3 Garbage Collection

In general, programming languages assign physical memory for each data structure and/or program code. The allocation of the necessary memory can be done in an implicit way, for example with a variable declaration or explicitly with a manual assignment of memory (via malloc() in C or new() in Java). The interesting part of memory management is how to handle the deallocation of memory so that it can be reused by other program parts. There are two paradigms for this:

- The programmer explicitly deallocates memory, for example via the free() command in C.
- The programmer does not care about deallocation, instead there is a central procedure that identifies unused objects and reclaims the corresponding memory.

The second approach is called *"Garbage Collection"* and was first described by John McCarthy at the Massachusetts Institute of Technology in Cambridge while inventing the programming language LISP between 1958 and 1960 [10].

The Java programming language uses this second approach and reclaims memory by running garbage collections in specified time slots or when an allocation failure occurs. A single garbage collection run consists of the following three steps:

1. **Garbage detection:** Identify all live and dead objects and/or distinguish between them.
2. **Reclamation of memory:** After all dead objects are clearly identified, the memory of these objects can be reclaimed.
3. **Defragmentation:** Compact all live objects to a contiguous space.

The rules for determining if an object is alive are as follows (taken from [2]):

- If there is a reference to the object on the stack, then it is alive.
- If there is a reference to the object in a local variable or a static field, then it is alive.
- If a field of a live object contains a reference to the object, then it is alive.
- The JVM may internally keep references to certain objects, for example, to support native methods. These objects are alive.

All objects that are not alive are dead and can be reclaimed by the garbage collector.

Figure 2.3 shows an example of two live objects on the heap. Object A is alive, because it is referenced by the stack and Object B is alive, because it is referenced by the live Object A. Figure 2.4 shows the same situation, but without the reference from Object A to Object B. This means that Object A

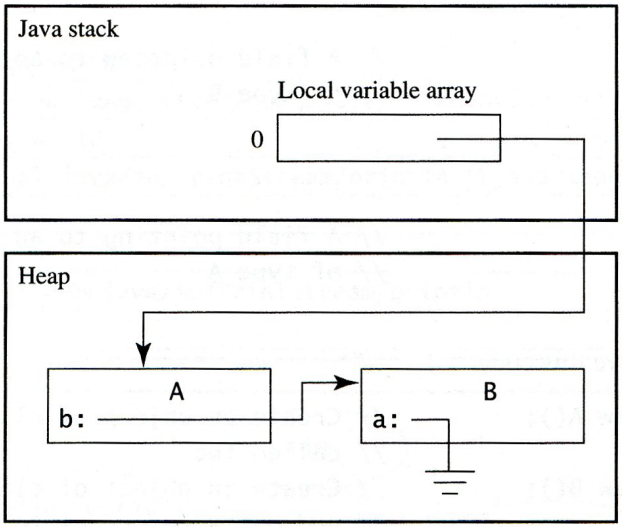


Figure 2.3: Two live objects [2].

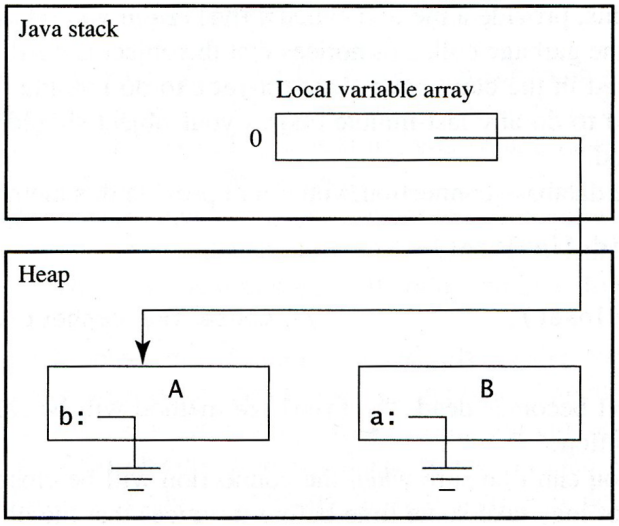


Figure 2.4: The right object is dead [2].

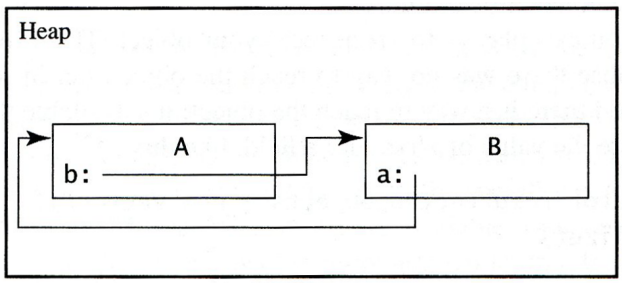


Figure 2.5: Both objects are dead [2].

is still alive, because it is referenced by the stack, and Object B is dead, because it is not referenced from any live object. Figure 2.5 shows another example where Object A and Object B reference each other. Due to the fact that both objects are not referenced by any live objects, both objects are dead and can be reclaimed by the GC.

With the garbage collection approach the corresponding memory of a dead object will not be released immediately after it dies. Instead, the garbage collector takes care of reclaiming all dead objects on the heap during a GC run, so that the developer does not need to explicitly free any memory.

The majority of executed Java code allocates many objects with a short life cycle, which may result in many dead objects on the heap after a short period. These objects will remain until the next garbage collection and therefore might trigger a significant fragmentation of the heap. This can decrease the performance of the application, since the JVM may need to trigger garbage collections more often due to the lack of enough contiguous space for the

allocation of new objects. It would therefore be advantageous to have the possibility to allocate objects with a short lifecycle in a local area of the heap, so that they can be implicitly reclaimed when the allocation thread dies. This might result in less objects on the global heap and therefore less global stop-the-world GCs that suspend the execution of all threads during the GC.

The garbage detection step is usually similar in different garbage collection approaches, because a traversal over all objects is always needed to identify the dead objects. There exist multiple approaches on how to reclaim the memory in the second step. A garbage collector can explicitly free every dead object or copy all the live objects to another memory block and afterwards free the entire unused block. Additionally, some garbage collectors combine both steps into one to avoid the traversal of all objects twice per collection. Some of them will be presented in the next sections.

2.3.1 Traditional Garbage Collection

Some garbage collection techniques like "Reference-Counting", "Mark-Sweep", "Mark-Compact" and "Copying" were the first GC approaches and still build the base for newer garbage collection techniques. These traditional methods are not the focus of this thesis and will only be introduced briefly, while the enhanced approaches "Generational" and "Balanced" will instead be presented more in detail.

2.3.1.1 Reference counting

The reference counting approach is — as its name says — counting the references pointing to an object in an additional field, so that every object knows how many objects are pointing at it. This enables an object to determine by itself, if it is dead (reference counter equals zero) or alive (reference counter greater than 0).

The reference counting approach reduces the time spent for distinguishing between dead and live objects at a GC, but increases the time used in normal runtime, because every time a pointer is created, the reference counter of the object pointed at needs to be updated. Furthermore, all reference counters of the objects a reclaimed objects points at, need to be decremented. This does not make the reference counting very efficient for real-time applications. Additionally, cyclic structures cannot be correctly detected and recycled with the reference counting approach, because their reference counter never reaches zero [3] [11].

2.3.1.2 Mark-Sweep

The mark-sweep approach consists of two phases. In the first phase the heap is traced in a depth-first or breadth-first search and all live objects are marked. In the second step the memory is traversed again and all unmarked objects — which are the dead objects — are reclaimed (swept).

The problem of this approach is the resulting fragmentation of the heap, because only the space of the dead objects is freed and can be reallocated by new objects. Additionally the costs of the garbage collection are proportional to the heap size [3] [10].

2.3.1.3 Mark-Compact

The mark-compact approach is similar to the mark-sweep algorithm, but tries to avoid its problems. When reclaiming the unmarked objects in the second phase, the algorithm moves the live objects to one block or contiguous space, so that there will be no fragmentation on the heap at all after a GC [3] [12].

2.3.1.4 Copying

The mark-sweep and mark-compact approaches both have the disadvantage that they are traversing the whole heap twice. The copying approach merges the marking and compacting traversals into one single run. Instead of marking the live objects with a flag, they are directly moved to the block next to the last block of contiguous live objects. This means that the approach only needs one traversal of the heap to block all live objects and therefore implicitly delete the dead objects [3] [13].

The problem of the copying collector is, that it needs additional memory to copy live objects. Minsky's garbage collector for LISP [14] was the first

copying collector and uses a file on disk as secondary space. This is very slow on modern computers, so that newer copying approaches were created [3]. One of them is the commonly used semispace collector [15] in combination with the Cheney compacting algorithm [16].

2.3.2 Generational Garbage Collection

The copying garbage collector presented in the previous section always has to copy all live objects. For large memory sizes this results in copying many data at each collection. In most programming languages and especially in Java many objects have a short lifetime while only a small percentage lives long term. As mentioned in [3] usually between 80 and 98 percent of the newly allocated objects die within a few million instructions, or before another megabyte has been allocated. A high percentage of these short-living objects die even faster and only survive some kilobytes of allocation.

Even if the garbage collection runs in short cycles of one kilobyte allocation, there is still a significant number of objects that die between two cycles. Otherwise the probability of surviving multiple other collections, when an object has survived once, is high. These objects will be copied over and over again during a garbage collection and therefore consume unnecessary computation time.

The solution for avoiding this overhead is dividing the memory space into different regions by the age of the objects. Each age region is called generation, which explains the name generational garbage collection. The idea was first described by Lieberman and Hewitt in 1983 [17]. The efficiency of the generational GC will grow as more generations exist. In the following example a generational space with two generations is considered. The area for short-living objects is called *"younger generation"* and the area for long-living objects *"older generation"*. Every generational space is divided into two subspaces to allow copying the objects and free a whole memory block at once.

At the start of the allocation every object is allocated in the first subspace of the younger generation. If an object survives a certain amount of collections, it will be copied to the older generation. The garbage collection in the younger generation runs more frequently than in the older generation, because its objects die fast which results in a high fragmentation of the subspace.

The term *root set* describes a set of references to all objects that are reachable from thread stacks and VM fields at the time the GC is called. Figure 2.6 shows a sample situation of a generational garbage collector before a collection. Some short-living objects are stored in the younger and some long-living objects are stored in the older generation. Objects in both generations are

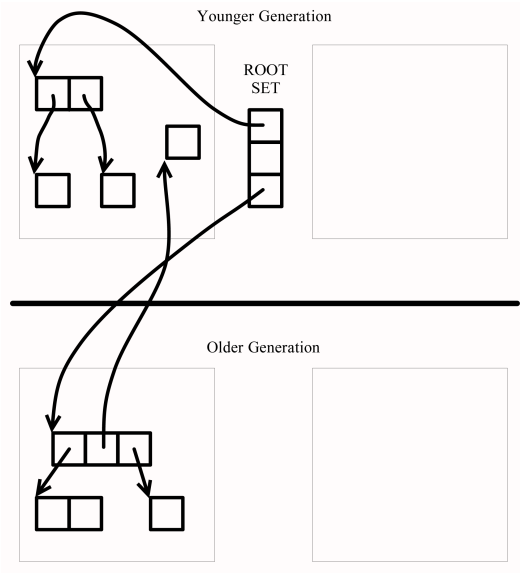


Figure 2.6: Generational garbage collector before a collection [3].

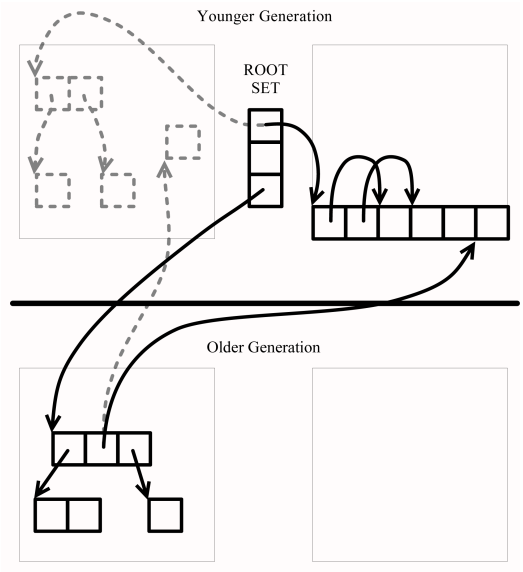


Figure 2.7: Generational garbage collector after a collection [3].

reachable via the root set. Figure 2.7 depicts the same situation after the collection. The live objects in the younger generation were either copied to the older generation (if they were old enough) or to the other subspace in the younger generation. After the garbage collection ran, the memory in the first subspace of the younger generation — which now only contains dead objects — can be considered as free and reused.

The generational garbage collection has many advantages compared to the simple garbage collection approaches. For example, the time spent in unnecessary copy operations is reduced and the revocation of the memory of dead short-living objects is performed in an efficient way. For this reason, the generational garbage collection is the basis for many modern garbage collection techniques [3] [17].

2.3.3 Balanced Garbage Collection

The balanced garbage collector was developed by IBM [18] and has been introduced as a new garbage collection policy with the IBM WebSphere Application Server V8 in 2011. It is optimized for large heaps and tries to equalize the stop-the-world pause times that are usually associated with garbage collections [4].

The costs for garbage collections in general depend on different factors. The

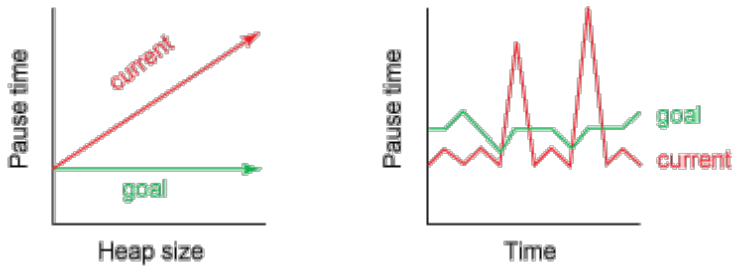


Figure 2.8: Pause time goals of the balanced garbage collector [4].

most important are:

- **Total size of the live data:** With the growing amount and sizes of live objects, the costs for tracing and discovering these live objects increases.
- **Fragmentation of the heap:** If the heap is highly fragmented, it is very likely that the GC needs to be triggered more often because sufficient contiguous space may not be found.
- **Rate of object allocation:** The rate at which objects are allocated influences the consumption of free memory. In the case of a high consumption rate, it is very likely that the GC will be triggered more often.
- **Parallelism of the GC:** When the garbage collector uses multiple threads during a GC, the workload can be processed faster which leads to shorter pause times.

The goal of the balanced garbage collector is to reduce the pause times as shown in Figure 2.8. Current garbage collectors increase their pause time with growing heap sizes, while the balanced garbage collector aims at keeping the pause times at the same level — even with large heaps. The global garbage collections in normal GC policies need a lot of time and their occurrence is not predictable. The balanced garbage collector tries to completely avoid global collections by distributing the work over partial GCs. This increases the time spent in local GCs, but reduces the pause time peaks occuring in classic garbage collection approaches [4].

2.3.3.1 Heap Organization

To achieve the ambitious pause time goals, the balanced garbage collector divides the heap into regions. This enables the garbage collector to only select parts of the heap for a collection in a so-called "collection set" as depicted in Figure 2.9.

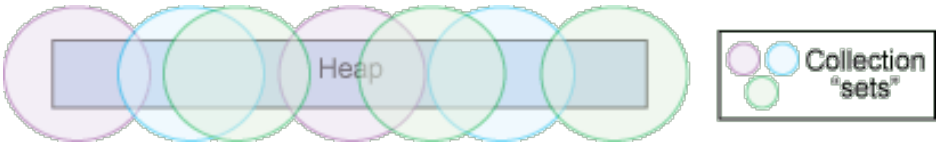


Figure 2.9: Dividing the heap into collection sets [4].

The garbage collector divides the heap into equally sized regions during the startup of the JVM. The region size is always a power of two (at least 512KB)

and is selected at startup based on the configured maximum heap size. The collector chooses the next power of two that results in less than 2048 regions, but creates a region size of at least 512KB. Except for small heaps (less than 512MB) the division will result in 1024 to 2047 regions on the heap. This division is static for the whole lifetime of the JVM.

Figure 2.10 shows an example part of the heap region structure with different region types.

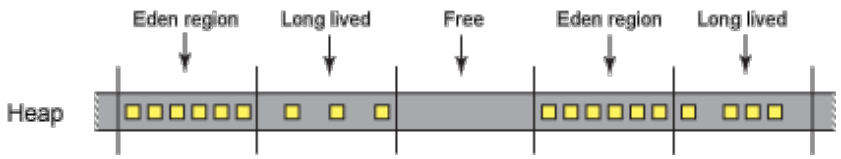


Figure 2.10: Region structure of the heap when using Balanced GC [4].

The regions serve as the basic units of the garbage collection, so that the freed memory will always correspond to one or multiple regions. Additionally, GC operations like tracing live objects or compacting are executed on a set of regions instead of a single region. The maximum object size is limited by the chosen region size, so that an object can only be located in a single region and not distributed over multiple regions. If an array cannot fit into a single region, it will be transformed in a special format called *"arraylet"* that takes care of splitting the array over multiple regions and accessing it later on. All newly allocated objects are kept together in so-called "eden regions" until the next garbage collection. This assures that all newly allocated objects

are seen at least once by the garbage collector, because the eden regions are automatically added to every collection set [4].

2.3.3.2 Collecting the heap

Like the generational garbage collector, the balanced garbage collector is based on the observation that recently allocated objects are more likely to become garbage than long-living objects. These objects are allocated in eden regions. While the generational garbage collector only collects the older generations when they are full, the balanced garbage collector also continuously performs GCs outside of the eden regions.

The balanced garbage collection can be split into the following three different cycles:

- **Partial Garbage Collection (PGC)** that collects a subset of the regions on the heap. It determines which regions will be included in the collection set for the next GC. In addition to the eden regions that have a high mortality rate, regions with a high fragmentation rate will also be included into the collection. The PGCs are the most common GC cycles in the balanced garbage collector.
- **Global Mark Phase (GMP)** that incrementally traces all live objects on the heap. The GMP runs less frequently than the PGCs, but

is still a common cycle in the balanced garbage collector. The GMP cycle only marks the live objects and does not free any memory.

- **Global Garbage Collection (GGC)** mark-compacts the whole heap in a stop-the-world manner. This cycle is normally only used when the garbage collection is explicitly triggered by the application via *System.gc()* and should be avoided as long as possible. It is additionally used as a last try to free memory before an *OutOfMemoryError* is thrown by the JVM.

While the PGCs are stop-the-world operations, the GMP can be split into multiple increments, which then are concurrently executed while the Java application runs. Figure 2.11 depicts a typical runtime behaviour of the balanced garbage collector [4].

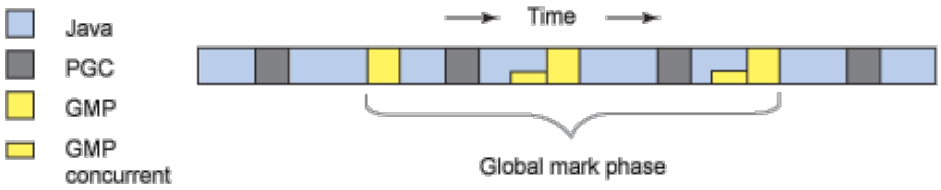


Figure 2.11: Timeline of a typical Balanced GC behaviour [4].

2.3.3.3 Understanding PGC

Each PGC has to ensure that enough free memory is available to continue the application after a GC. It therefore has to select suitable regions to add

to the collection set for the current collection. There are three criteria for regions to be eligible for addition to a collection set:

- Eden regions, which contain the newly allocated objects since the last PGC, are automatically included.
- Regions that are identified by the GMP to be highly fragmented and would free a certain amount of memory, if they were collected, are added too.
- The garbage collector gathers statistics about the mortality rate of objects. Regions outside the eden regions that contain a lot of objects with a high mortality rate are included as well.

Figure 2.12 shows an example collection set selection for a PGC.

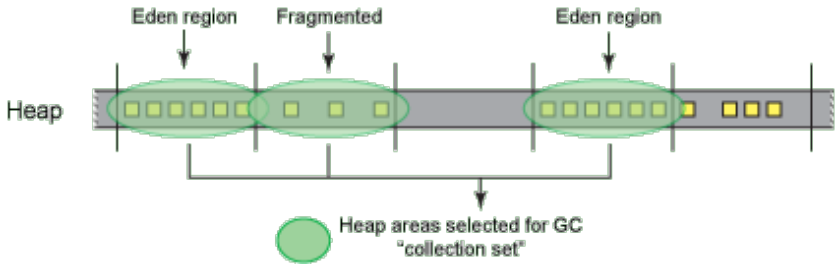


Figure 2.12: Collection set selection for a PGC [4].

Normally the PGC uses a copying approach similar to the generational garbage collector, but with one difference: There is no fixed reservation of memory for copying the surviving objects. Instead, the collector consumes

a set of free regions that is large enough to include all objects that are still alive and should be copied. In case there are not enough free regions for the copying operation, the garbage collector switches to an in-place trace-and-compact approach that does not require any free regions to perform its collection [4].

2.3.3.4 Understanding GMP

The GMP is triggered when the PGC is unable to keep up with the newly allocated objects inserted into the system. The garbage that is not collected by the PGC would slowly accumulate and fill up the heap. To avoid this, the GMP is triggered to create a new mark map as soon as the efficiency of the PGC deteriorates. The GMP then marks all live objects on the heap in a combination of stop-the-world increments and concurrent processing. The GMP increments are usually scheduled half-way between two PGCs. If there is additional CPU time left in a PGC execution, some threads will be dispatched to perform GMP work, so that the overall global marking phase can be finished in a shorter time. The PGC still works with the old mark map while the GMP is running. The newly created mark map gives an updated view on live and dead objects, so that the PGC should again be able to perform well afterwards [4].

2.3.4 Stack Based Garbage Collection

The Stack Based Garbage Collection [5] tries to allocate as many objects on the stack as possible. As stated in Section 2.2 the memory used by the JVM is divided into two heavily used large regions — the heap and the stack — and the two smaller regions — the class area and the native stack. The heap can be used without any location limitations and dead objects are reclaimed during a garbage collection run. The stack already has a thread-local organization with frames and can therefore be used to store additional thread-local data.

```
r1 = new Obj1; //global object ①
{
    Obj2 r2 = new Obj2; ②
    {
        Obj3 r3 = new Obj3; ③
    }
    ... ④
}
... ⑤
```

Figure 2.13: Example Java program [5].

Figure 2.13 shows a simple example of a Java program that creates objects in different scopes. In the traditional way of allocation — as you can see in Figure 2.14 — all reference pointers are stored on the stack and all corresponding data objects on the heap. The disadvantage of such a fixed distribution can be seen in step ⑤ of the example code when the program has been executed. The global object *Obj1* is still alive and referenced by

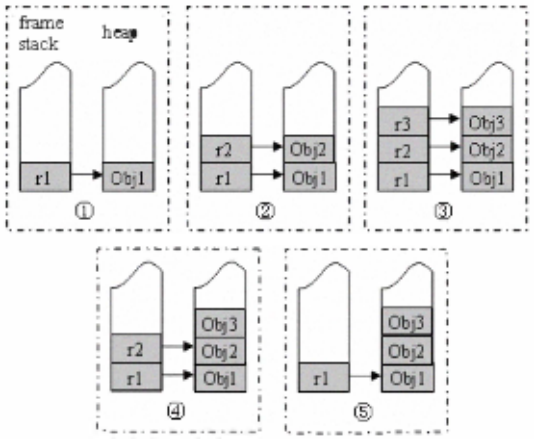


Figure 2.14: The heap and stack model [5].

the stack pointer *r1*. The pointers to the objects *Obj2* and *Obj3* were both deleted after their scope ended. This means that, although the lifecycle of these objects is very short, the corresponding memory on the heap is occupied until the next garbage collection. Under certain circumstances this might be a long time compared to the lifetime of the objects [5].

The idea of the stack based garbage collection by Xu and Shen [5] tries to improve this issue. As shown in Figure 2.15, the locally used objects *Obj2* and *Obj3* are stored on the stack during their creation in step ② and ③. Both objects have a short life cycle and can be deleted at step ④ and ⑤. Figure 2.15 shows that the reference pointers and the data objects themselves are directly deleted at this point and the corresponding memory is reclaimed. The result shown in step ⑤ equals to the memory allocation after step ①, so that only the global object *Obj1* with the corresponding reference pointer

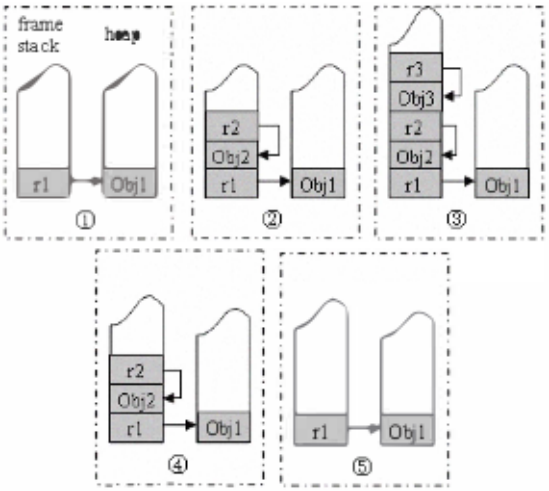


Figure 2.15: Heap and stack when using stack based allocation [5].

r1 still exists. This means that instead of blocking the memory until the next garbage collection, the memory used by these objects is immediately available for new allocations [5].

Komuro and Abe [19] developed a similar approach of a stack based garbage collection for the Lua processor and scripting language. Their approach is limited to loops, which means that all objects inside a for loop are allocated in a special dedicated region on the object stack. All objects in this region can then be deallocated as soon as the loop ends without explicitly triggering a garbage collection.

2.4 Escape Analysis

An essential precondition for a thread-local allocation of objects is an escape analysis, which is briefly introduced in this section. Escape analysis can be performed in two different ways:

- **Static analysis:** Identify objects eligible for thread-local allocation during compile time and handle them differently.
- **Runtime analysis:** Identify eligible objects during runtime and maybe revoke this decision, if the locally allocated objects escape too often.

The approach of the static analysis can only consider objects for a thread-local allocation that are definitely not accessed by any other threads. The compiler can then modify the bytecode for these objects to differentiate their allocation from normal heap allocation. An object eligible for thread-local allocation will then be created with a newly introduced new operation, such as **snew**, while all normal heap allocations will still use the traditional **new** operation. The Java Virtual Machine can easily switch between both allocation methods by checking, if the bytecode operation equals **snew** or **new**. An example for a possible switch is depicted in Figure 2.16. The disadvantage of the static analysis approach is that only objects, where the compiler is sure that they will not escape, can be locally allocated. Additionally, the with **snew** instructions modified bytecode is not portable to VMs of other vendors, which is not allowed by the JVM specification [6].

```
if (op = snew) {
    if (not enough free memory in the thread local region) {
        run thread local garbage collection
    }
    allocate memory from the thread local region
}
else if (op = new) {
    if (not enough free heap memory) {
        run garbage collection
    }
    allocate memory from the heap
}
```

Figure 2.16: Allocation with statically modified bytecode (inspired by [5]).

The second approach is to perform escape analysis during runtime. Due to the fact that the JVM can collect statistics during its execution, this approach can optimistically allocate objects in the thread-local region, even if they might be accessed by other threads in the future. The implementation must then ensure that a locally allocated object, that will be referenced by any other object, is copied back to the heap and all existing references are updated. This might result in high overhead.

As mentioned, the JVM is able to collect some data during runtime that can be used to improve the decision making process. For example, the escape percentage could be stored by the JVM for each line of code. If the escape percentage exceeds a certain threshold, the allocation in this particular line of code can be changed to global heap allocation to avoid costly copying operations in the future. The longer a program runs, the more accurate the predictions will become. Nevertheless the advantage of possibly more locally allocated objects might be better than the effort that must be

invested to create and update the statistic and to check and potentially copy the objects back to the heap during each creation of a reference [5].

2.5 Thread Local Garbage Collection

This section introduces into the thread-local allocation and garbage collection that is the basic idea of this thesis. The classic garbage collection approaches rely on a global GC, which causes delays in the program execution due to the stop-the-world phase. The idea of a thread-local allocation and garbage collection of objects is to improve this behaviour.

The allocation of as many objects as possible in a thread-local memory area would result in the following improvements:

- **Implicit GC:** The thread-local memory area can be reused without any GC directly after the associated thread dies.
- **Local GC:** When the local memory area is full, a local GC can be performed that will only impact the current thread.
- **Less effort for global GC:** Due to the fact that more objects are allocated in the thread-local memory area, fewer objects need to be allocated on the global heap. This will result in shorter processing time for each garbage collection run and fewer GC invocations overall.

- **Less fragmentation on the global heap:** Objects with a short lifecycle — that often cause fragmentation — might be eligible for a thread-local allocation, so that the global heap may become less fragmented.

The realization of a thread-local allocation and garbage collection for all objects would create a potential performance boost, but unfortunately not all objects are eligible for a thread-local allocation. Due to the fact that a thread-local area can only be accessed by the associated thread, only non-escaping objects qualify for this kind of allocation. A thread-local allocation and garbage collection can therefore only work well, if the prediction of the escaping of an object has a high degree of accuracy. If the prediction is wrong, the object needs to be copied back to the global heap and all reference pointers must be updated. This procedure will probably need more time than the time saved with a thread-local allocation and garbage collection. The foundation for a reliable and effective thread-local garbage collection is therefore a good escape analysis, whose concept has been introduced in the previous section.

The next section will introduce the Thread Local Region Garbage Collection approach as a variant of the thread-local garbage collection, followed by the discussion of related work in this field.

2.5.1 Thread Local Region Garbage Collection

Unfortunately, the access to a stack frame is limited by the LIFO principle, so that a realization of the Stack Based Garbage Collection (SBGC) with arbitrary object access would be difficult. For this reason, the idea of a Thread Local Region Garbage Collection (TLRGC) was born, which combines the idea of the SBGC with the balanced garbage collection on the heap.

The TLRGC aims at the same goal as the SBGC, which is to allocate objects thread-locally and reclaim them without the need of a global stop-the-world GC. The SBGC uses stack frames to allocate objects and automatically reclaim them when the stack frame is popped. This approach has several flaws:

- **Limited size:** The stack frames only have a certain amount of memory, so that maybe not all eligible objects will fit into the stack frame.
- **Only method-local:** Every stack frame represents a method invocation, so that only method-local objects could be stored in it. All other objects, such as class variables, still need to be stored on the heap.
- **No garbage collection:** There is no possibility to run a thread- or in this case method-local garbage collection on the stack frame. This means that as soon as the stack frame object storage space is exceeded, all additional objects need to be stored on the heap, although a certain percentage of the objects on the storage might already be dead.

- **Escaping:** Objects may escape from a method and the costs for copying them back to the heap are probably higher than the improvements gained by the method-local allocation.

The TLRGC is using the same idea of separating local objects from the global heap by allocating them in a special memory area, but is solving the flaws that occured with the SBGC. The idea of the TLRGC is to allocate objects in Thread Local Regions (TLR) that are exclusively assigned to threads. Each thread has at least one assigned TLR on which it can allocate local objects. The balanced garbage collector developed by IBM [4] already divides the heap into different regions, so that it can be used to build the TLR allocation and garbage collection on top of it. The advantages of Thread Local Regions are:

- **Thread-Local:** All objects allocated by a thread are stored in the same TLR.

- **Escaping:** It is more likely that an object escapes from a method than from a thread, so that a thread-local allocation will — compared to the method-local approach — reduce the costs for copying back escaping objects.

- **"Unlimited" size:** The allocation in regions is not limited to a certain object size, while stack frames are limited and need modifications to provide a similar functionality. Even if a region's max size is reached, it is possible to assign additional regions as TLR to one thread.

- **Local GC:** The TLRs can be locally garbage collected, so that only a single thread execution need to be stopped during the GC. Other threads will not be affected.

The TLRGC additionally has the property that only objects that are not referenced by any other thread can be allocated in a TLR. This means, that it is still necessary to copy escaping objects back to the global heap, but it should occur less often than with the SBGC approach. Additionally, the TLRGC provides extended possibilities to handle local garbage collections, which are described in detail in the following sections.

2.5.1.1 Heap Organization

The idea of the TLRGC — as already mentioned — is based on the balanced garbage collector and makes use of its region based heap division. Each region on the heap has an associated region type, such as *FREE* or *CONSUMED*. To implement the TLRGC it is necessary to distinguish between TLR and "normal" heap regions, so that the new region type *TLR* is introduced.

All newly created objects of a thread will be allocated on its active TLR region. Once the active TLR is full, it is possible to create and hold a chain of TLRs for a single thread. New objects can then only be allocated in the TLR marked as active, while the other regions in the chain hold the older objects. As soon as objects escape, which means that they are accessible

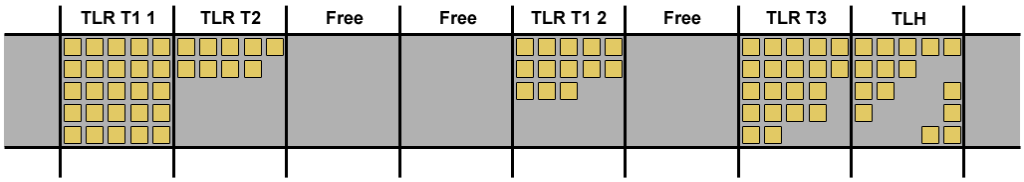


Figure 2.17: Region structure when using TLR allocation.

by other threads, they need to be copied to a global heap region. Statistics about the escaping of objects can be collected, so that they might be directly allocated on a global heap region in the future. Object classes escaping too often might cause a significant increase of the runtime, because of the time needed for copying them.

It is necessary to define how many regions a thread can consume as TLRs before a TLRGC is triggered. The parameter *MAX_TLR_REGIONS_PER_THREAD* defines this threshold and needs to be adjusted based on the experiences/observations in the prototype implementation. For the basic idea presented in this section, it is assumed that all heap regions can be consumed as TLRs until the heap is full, which leads to a global GC. Depending on the application different values for the *MAX_TLR_REGIONS_PER_THREAD* parameter might produce the best results. Figure 2.17 shows an example heap division when using TLR allocation.

2.5.1.2 Escape Handling

The escape handling is an important precondition for a correct working TLRGC, because it is essential that objects in a TLR are really thread-local and only referenced by objects allocated by the same thread. To ensure this, all objects that are newly referenced by another threads object or by an already escaped object from the same thread need to be copied out to the normal heap region called *Thread Local Heap (TLH)*. The difference between TLH and TLR is that the objects of a thread in TLH may be arbitrarily distributed over multiple regions and referenced by objects of other threads, while all objects in the TLRs are stored together in special regions exclusively assigned to the current thread and only referenced by each other. A good way to handle the escaping of objects in a TLR is to trigger a write barrier at every creation of a reference during the execution of the program. This barrier is then responsible for the following two steps:

1. **Copying** objects from TLR to TLH.
2. **Updating** all references from TLR objects to the new memory location.

Figure 2.18 shows an example of the barrier escape handling. On the left side a new reference will be created from a TLH object to a TLR object that is already referenced by another TLR object. The right side shows the same situation after the execution of the write barrier. The escaped TLR object

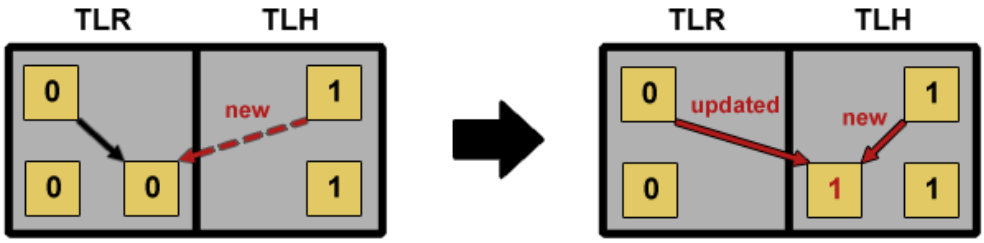


Figure 2.18: Barrier Escape Handling - Copying escaping objects

was copied to TLH, its escape bit set and the reference from the other TLR object updated to the new TLH location.

2.5.1.3 Thread Death Handling

The first advantage of the TLR allocation and garbage collection comes in when a thread dies. With the barrier escape handling it is ensured that the TLR regions associated with a thread only contain objects accessed by this particular thread and from nowhere else. This makes it easy to reclaim memory on the death of a thread: all TLR regions associated with the dying thread can entirely be added to the list of free regions, because they only contain dead objects.

Figure 2.19 shows an example for a thread death handling. The four TLR regions of Thread 1 (TLR T1) are automatically transformed to free regions with almost no cost.

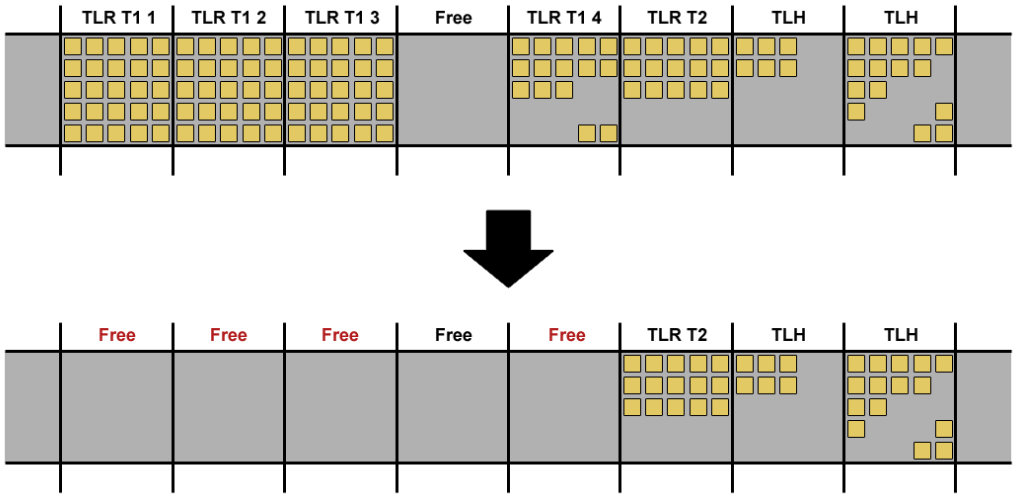


Figure 2.19: Thread Death Handling

2.5.1.4 TLR Garbage Collection

The second advantage of the allocation in TLR regions is the ability to perform a Thread Local Region Garbage Collection (TLRGC). Due to the fact that the TLR regions only contain thread-local objects, only the thread itself needs to be interrupted and not all threads as in the stop-the-world global garbage collection.

The TLRGC will be triggered in the following cases:

- **Not enough space:** The free space of the active TLR region is too small to fit the requested object size and *MAX_TLR_REGIONS_PER_THREAD* is already reached.

- **Allocation Threshold:** The defined capacity threshold of the active region until performing the next GC is reached.

The TLRGC then performs — depending on the free regions left to work with — either a copying or a mark-sweep-compact approach to free the thread-local memory occupied by the dead objects.

2.5.2 Related Work

This section introduces four existing approaches to realize a thread-local allocation and garbage collection.

2.5.2.1 Jones and King

Jones and King [20] partition the heap into a shared heaplet and multiple thread-local heaplets and use a snapshot analysis to determine the eligibility of objects for a local allocation. The snapshot of the system is taken at a point during the program's execution when enough classes are loaded by the JVM. All objects are classified into the following three categories during the snapshot analysis:

- **Strictly local (L):** These objects will never escape for all possible execution paths.
- **Optimistically local (OL):** These objects are local at the time of the snapshot, but might escape when they get a dynamically loaded class

passed in as method parameter in the future.

- **Global (G):** These objects are already accessible by multiple threads in the current snapshot.

After categorizing the objects, the bytecode is modified to allocate objects in heaplets according to the three categories. There will be one global shared heaplet and each a local and optimistically local heaplet per thread. To ensure the conformity of the approach, references from objects in optimistically heaplets to objects in local heaplets are forbidden.

The L and OL heaplets can be locally garbage collected without any further checks as long as no new classes are loaded into the system. Whenever a class is dynamically loaded by the JVM, it is necessary to check, if this new class is passed to any objects in OL heaplets. Every OL heaplet that is detected during that check must be treated as global from now on, because at least one object in it can now be reached from another thread.

2.5.2.2 Steensgaard

Steensgaard [21] also divides the heap into shared and thread-specific areas. His idea is based on the generational garbage collection, which manages the allocated objects in young and old generations (see section 2.3.2). Steensgaard's approach extends the basic generational GC by introducing young

and old generations per thread. Additionally a single young and old generation is held for the allocation of shared objects.

The eligibility of objects for a thread-local allocation is determined by a static bytecode analysis at compile time. The *new()* allocation instruction is replaced by a *threadNew()* instruction for every object eligible for a thread-local allocation. The modified JVM can then allocate all *threadNew()* objects in the thread-local regions while the objects allocated with the classic *new()* instruction are still allocated in the shared heap as usual.

The approach cannot perform local garbage collections independently per thread, because all GCs need to cover the whole heap (local and shared regions). Nevertheless, Steensgaard's approach distinguishes two kinds of collections:

1. **Minor:** All young generations (local and shared) are collected and the live objects are moved to the corresponding old generations.

2. **Major:** All generations in all regions are collected. The live objects from young and old generations are combined in a "new" old region.

The minor collections run more often than the major collections and should save time compared to the classic generational garbage collection approaches.

2.5.2.3 Domani et. al.

Domani et. al. [22] introduce a different approach of a thread-local allocation and garbage collection. Escaping objects are stored in a shared heap, while local objects are kept in thread-local heaps. Instead of using a static escape analysis, this new approach detects the escaping of objects during runtime with the help of a write barrier. Rather than introducing a bit/flag per each object, a separate bitmap is held to store the escaping status of all objects. Whenever a reference to a non-escaped object is created from another thread, the object itself and all related objects need to be marked as escaped and copied out to the shared heap. Objects that are guaranteed to escape, such as the thread object itself or JNI references, are directly allocated in the shared heap.

The local areas are split into the following two types:

1. **Local-free areas:** These areas contain global objects, but are free of local objects.

2. **Free areas:** These area are completely free of any objects and are available for future allocations.

The Domani et. al. approach distinguishes between local and global collections that are both mark-sweep collections. The local collection is independent from other threads and performed per thread whenever an allocation

failure occurs. The global collection is a stop-the-world collection and collects both the local and global areas of the heap.

The TLRGC approach introduced for this thesis in section 2.5.1 is similar to Domani's approach, but there are a few differences that distinguish both approaches:

- The escape analysis in the Domani et. al. approach is only performed globally, while this thesis analyses the escaping of objects based on the PC and can therefore provide more details about the escaping of objects.
- Domani's global collection collects all areas including the thread-local areas. The idea of the TLRGC in this thesis is to completely exclude the local areas from global garbage collections.
- The TLRGC approach is based on the balanced garbage collector instead of the simple mark-sweep collection used by Domani. The utilization of the balanced garbage collector provides multiple advantages, such as the possibility to run a partial garbage collection with the local-free areas without collecting the whole heap in a stop-the-world phase.

2.5.2.4 Marlow and Jones

All so far introduced approaches are based on the idea of avoiding references from global to local objects, so that they can be thread-locally garbage collected without influencing other threads. The approach introduced by Marlow and Jones [23] chooses a different way that still allows pointers from global to local objects. Rather than checking the escaping of objects at a write barrier and copy escaping objects out accordingly, the local objects are protected by a read barrier. This barrier ensures that pointers from global to local objects may only be followed by the CPU that allocated the local object. When a foreign object is accessed, the current CPU requests the copy out operation from the owning CPU before continuing its execution. The protection of local objects via the read barrier enables the execution of local and global GCs whenever suitable.

The implementation of Marlow and Jones is based on the Haskell compiler that implements a read barrier for each pointer dereference, so that it makes sense to extend the existing read barrier instead of implementing a write barrier from scratch. The overhead for introducing checks at a read or write barrier might be similar for other languages and/or approaches.

Chapter 3

Previous Work

This thesis is based on the previously completed R&D project, which will be introduced in this chapter. The first part will be an introduction into the instrumentation of the JVM, followed by a summary of the class based escape results.

3.1 Instrumentation of the JVM

The instrumentation of source code can be performed in different ways that can be summarized according to the following three paradigms:

- **Manual instrumentation:** The programmer adds extra instructions to the source code to measure the execution time of certain methods.

- **Compiler instrumentation:** The compiler adds additional instrumentation code to the bytecode during the compilation, often indicated by a special compile flag.
- **Runtime instrumentation:** The compiled bytecode is modified directly before its execution, e.g. by adding additional jumps to the instrumentation code.

Manual instrumentation is the easiest way to add instrumentation code and collect data, because all internal data structures and lines of code can directly be accessed and modified. The compiler instrumentation still has a white box view of the source code and can analyse it to find the best spots for an instrumentation. With the runtime instrumentation, the program can only be evaluated in a black box view, because just the compiled bytecode of the program is available. So it is only possible to collect general statistics and information about the behaviour of the program during runtime. This information can be used to optimize the execution, e.g. by the Just-In-Time Compiler (JIT). Special runtime instrumentation programs — so called profilers — are therefore often used to get an insight and debug the usage of $3rd$ party libraries and components.

The IBM research project has full access to the JVM source code, so that the runtime instrumentation was chosen to gain the most flexibility in collecting data about the escaping of objects.

3.1.1 Extension of the VMObject

Each object in the Java Virtual Machine is internally represented by a *VMObject* that holds general information about the allocated object, such as its class and certain kinds of flags. For the escape analysis, an object needs to keep track of its allocation thread. Unfortunately the *VMObject* does not provide the native possibility to store this data, so that it needs to be extended as shown in Figure 3.1. The new field *allocationThreadAndFlags* is initialized with the corresponding data at the allocation of an object.

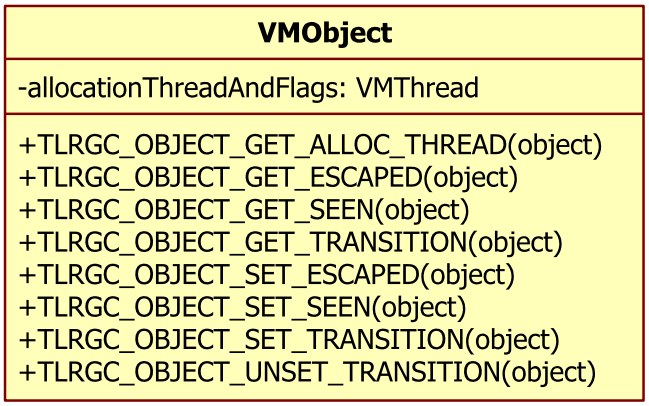


Figure 3.1: Extension of the VMObject.

To correctly identify and count the escaping of objects, the following boolean flags are stored per object:

- **Escaped:** Indicates, if the object is escaped or not.
- **Seen:** Indicates, if the object was seen during garbage collection.

- **Transition:** Special bit needed to correctly handle the transition from non-escaping objects into the escaping state.

Due to the fact that the *VMObject* is the central object class in the JVM, it is important to keep its memory footprint small. Fortunately the pointer to a thread has an empty byte that can be used to store the three flags mentioned above without using additional memory. The footprint of the *VMObject* is still increased by the thread pointer, but was the smallest possible addition necessary to perform the escape analysis in the R&D.

To simplify the access to the flags, the necessary offset and bit shifting operations were encapsulated in the following macro definitions:

- **TLRGC_OBJECT_GET_ALLOC_THREAD:**

  Returns the pointer to the thread the object was allocated by.

- **TLRGC_OBJECT_GET_ESCAPED:**

  Returns 1, if the object is escaped, otherwise returns 0.

- **TLRGC_OBJECT_GET_SEEN:**

  Returns 1, if the object has been seen in the GC, otherwise returns 0.

- **TLRGC_OBJECT_GET_TRANSITION:**

  Returns 1, if the transition bit is set, otherwise returns 0.

- **TLRGC_OBJECT_SET_ESCAPED:**

  Sets the escaped bit to 1.

- **TLRGC_OBJECT_SET_SEEN:**

  Sets the seen bit to 1.

- **TLRGC_OBJECT_SET_TRANSITION:**

  Sets the transition bit to 1.

- **TLRGC_OBJECT_UNSET_TRANSITION:**

  Resets the transition bit back to 0.

3.1.2 Escape Marking

The extensions made to the *VMObject* enable the marking of objects as escaped during the runtime of a program. To be sure that the escaping of every object is recognized, the creation and update of each reference in the JVM needs to be intercepted. Every garbage collector in the JVM has to implement read and write barriers to intercept the execution of programs and to have the ability to perform special checks [24]. The escape marking instrumentation code was added to the write barrier to perform the necessary operations each time a reference to an object or array is created or updated. Figure 3.2 shows the control flow. Both the normal object path (*objectStore()*) and the array path (*arrayObjectStore()*) call the corresponding method in *TlrgcUtil* that in turn calls the *tlrgcMarkAsEscaped()* method to actually perform the escape marking. To ensure that the escape checks are always performed, the JVM was modified so that the write barrier is called at every object allocation. Additionally the inline allocation of the JIT was

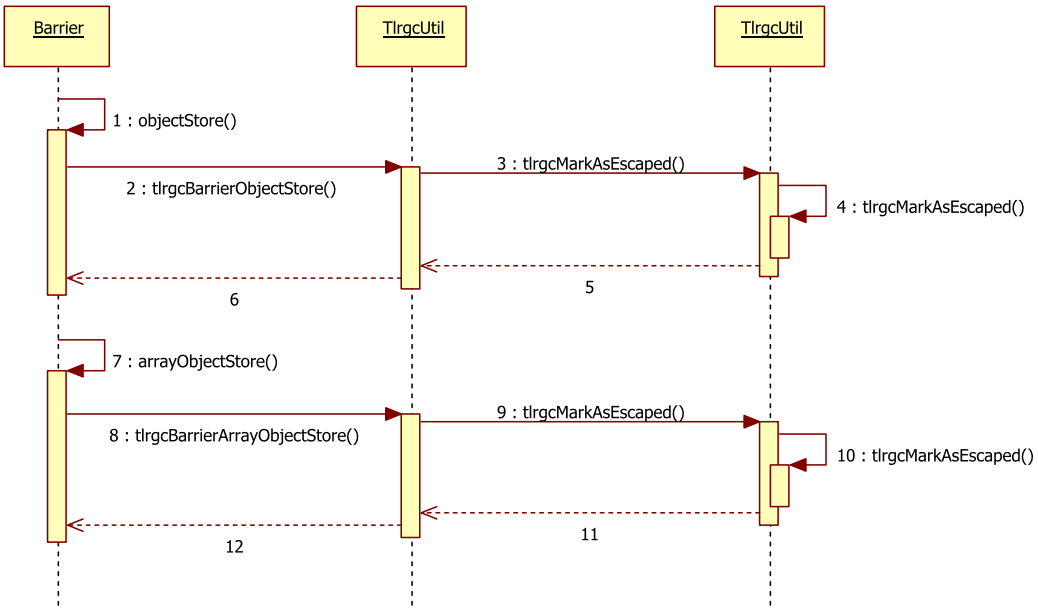


Figure 3.2: Barrier Escape Marking

disabled for the whole escape analysis, because its optimizations might distort the results [25].

A short definition of the term "escaping" was already given in the Introduction as being accessible by another thread than the own allocation thread.

More precisely the escaping of an object can be split into the following cases:

1. The object is referenced by an object that was allocated by another thread.

2. The object is referenced by an object that has already been marked as escaped.

3. A reference to the object is stored into a VM structure that is globally accessible.

Beside these categories special cases exist, where the objects are guaranteed to escape, but they were not specially handled in the R&D.

3.1.2.1 Write Barrier Check

Figure 3.3 shows the write barrier check in the normal object path (*tlrgcBarrierObjectStore()*). The array path (*tlrgcBarrierArrayObjectStore()*) is similar, so that only the normal object path is presented here. The destination object (*dstObject*) is the object into which the new reference will be stored and the source object (*srcObject*) is the object the new reference will point at. Nothing needs to be done if the *srcObject* has already escaped. Otherwise the three previously mentioned categories of escaping are reduced to two checks, because every VM structure is inherited from *VMObject* and knows its own allocation thread. The storing into a VM structure is therefore also covered by the else-if case. In summary, the method *tlrgcMarkAsEscaped()*, which marks the *srcObject* as escaped, is either called when the *dstObject* is marked as escaped or when it has a different allocation thread than the *srcObject*.

3.1.2.2 Recursive Escape Marking

The reason for a separate *tlrgcMarkAsEscaped()* method instead of directly marking the *srcObject* as escaped with the previously defined macro *TLR-*

```
void tlrgcBarrierObjectStore(VMObject dstObject, VMObject srcObject) {
    // only perform check, if srcObject has not already been escaped
    if (!TLRGC_OBJECT_GET_ESCAPED(srcObject)) {

        if (TLRGC_OBJECT_GET_ESCAPED(dstObject)) {
            // dstObject is escaped -> now the srcObject is also escaped
            tlrgcMarkAsEscaped(srcObject);
        }
        else if (TLRGC_OBJECT_GET_ALLOC_THREAD(srcObject)
                != TLRGC_OBJECT_GET_ALLOC_THREAD(dstObject)) {
            // allocThreads are not the same -> mark srcObject as escaped
            tlrgcMarkAsEscaped(srcObject);
        }
    }
}
```

Figure 3.3: Write Barrier Check - Normal Object Path

GC_OBJECT_SET_ESCAPED is that the escape marking must be performed recursively. As soon as an object escapes, all the objects that are referenced by this object — the object slots — need to be marked as escaped as well. Figure 3.4 shows an example object structure at the write barrier triggered by the creation of a new reference from *obj1* to *obj2*. Instead of only marking *obj2* as escaped, the whole object tree below *obj2* needs to be recursively marked as escaped as depicted in Figure 3.5. The source code shown in Figure 3.6 performs the recursive escape marking and will in the example mark all objects of *VM Thread 2* — with exception of *obj3* — as escaped, because they are all reachable from *obj2*. In addition to setting the escape bit and iterating the object slots, *tlrgcMarkAsEscaped()* also sets the transition bit when the object has already been seen in a GC. This ensures that the transition of non-escaping objects to the escaping state is correctly recognized at the next garbage collection.

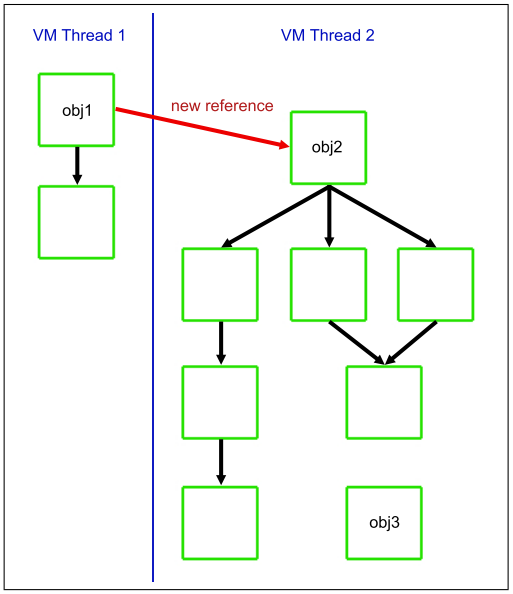


Figure 3.4: Example object structure **before** the recursive escape marking.

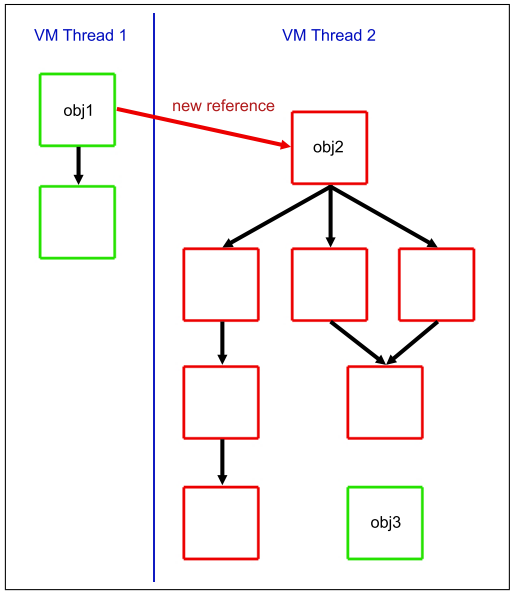


Figure 3.5: Example object structure **after** the recursive escape marking.

```
void tlrgcMarkAsEscaped(VMObject object) {
    // only continue recursion, if object is not marked as escaped yet
    if (!TLRGC_OBJECT_GET_ESCAPED(object)) {

        // if object has been seen before, set transition flag
        if (TLRGC_OBJECT_GET_SEEN(object)) {
            TLRGC_OBJECT_SET_TRANSITION(object);
        }

        // mark object as escaped
        TLRGC_OBJECT_SET_ESCAPED(object);

        // iterate all object slots and also mark them as escaped
        iterate_all_object_slots(object, tlrgcMarkAsEscapedObjSlotsCallback);
    }
}

void tlrgcMarkAsEscapedObjSlotsCallback(VMObject slotObject) {
    // also mark slotObject as escaped
    tlrgcMarkAsEscaped(slotObject);
}
```

Figure 3.6: Recursive Escape Marking - Source Code

3.1.3 Collecting Data

Two important decisions need to be made before collecting data:

1. What is the best spot to collect this data?

2. How should the data be stored?

All relevant information for the data collection is stored in the *VMObject* and can easily be accessed, because all objects on the heap are globally available and can be iterated everywhere in the JVM. It is useful to collect the data during global garbage collections, because the execution of all threads is stopped at that point, which enables the creation of a consistent snapshot for the escaping of objects.

The garbage collector in the JVM contains the two methods *preCollect()* and *postCollect()*, which are directly called before and after a single garbage collection. These methods can be used to hook in code for collecting the necessary escape analysis data. Figure 3.7 shows the hook in process for both methods in a sequence diagram.

To facilitate the collection of escape analysis data, a suitable data structure was created. Figure 3.8 shows the chosen design for this data structure. *TlrgcClassEscapeData* stores the class based escape data, while *TlrgcEscapeData* holds the global escape counts and a HashTable. This HashTable

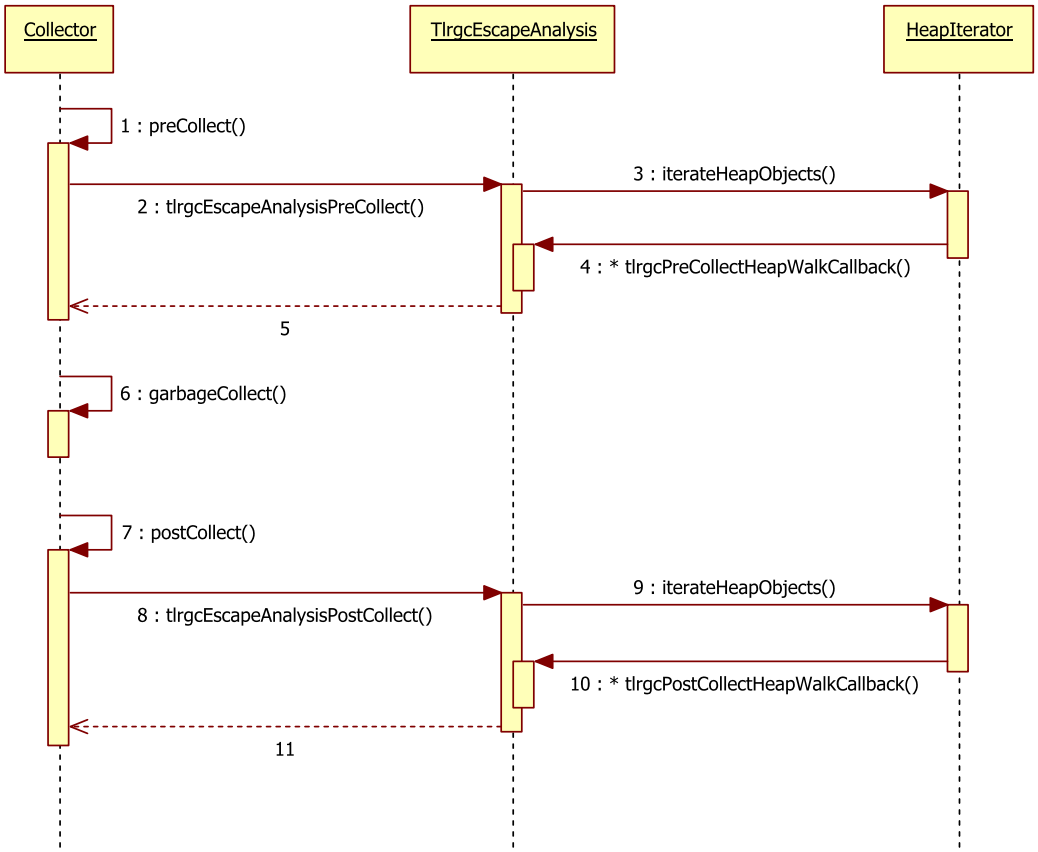


Figure 3.7: Sequence diagram of the data collection.

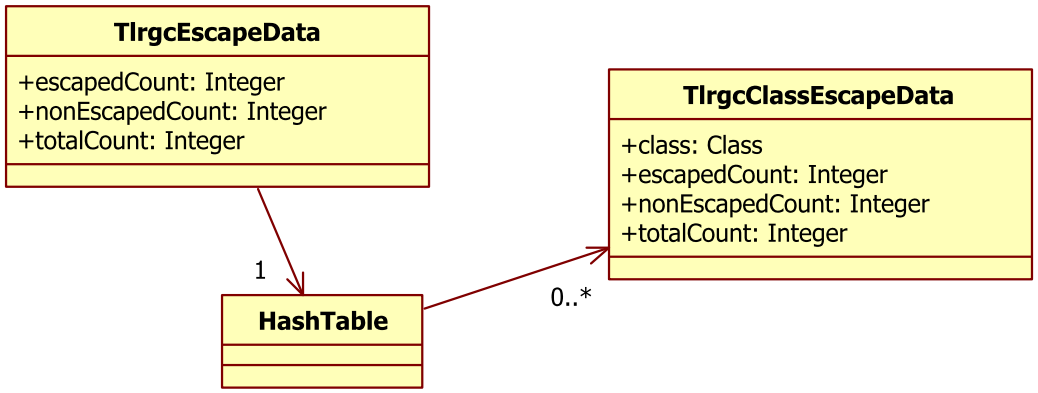


Figure 3.8: Class diagram of the escape data structures.

stores instances of *TlrgcClassEscapeData* and uses the class as key. Both the *TlrgcEscapeData* and the *TlrgcClassEscapeData* contain counters for escaped, non-escaped and total objects. The equation

$$escapedCount + nonEscapedCount = totalCount \tag{3.1}$$

must always be true. The top container *TlrgcEscapeData* is stored as local variable, so that the escape data is collected separately for each GC.

The method *tlrgcEscapeAnalysisPreCollect()* depicted in Figure 3.7 makes use of the data structures to perform the following tasks:

- Initialization of the data structures *TlrgcEscapeData* and *TlrgcClassEscapeData*.
- Iterate over all objects on the heap and respectively update the related counters.
- Print statistics to the console for debugging purposes.
- Save the collected data to a file (see Section 3.1.4).

The update of the escape counters is based on the *escaped*, *seen* and *transition* bit of each iterated object and performed according to the Transition Diagram in Figure 3.9. Every object starts in the 000 ($escaped = 0$, $seen = 0$, $transition = 0$) state after allocation and will either end up as a non-escaped

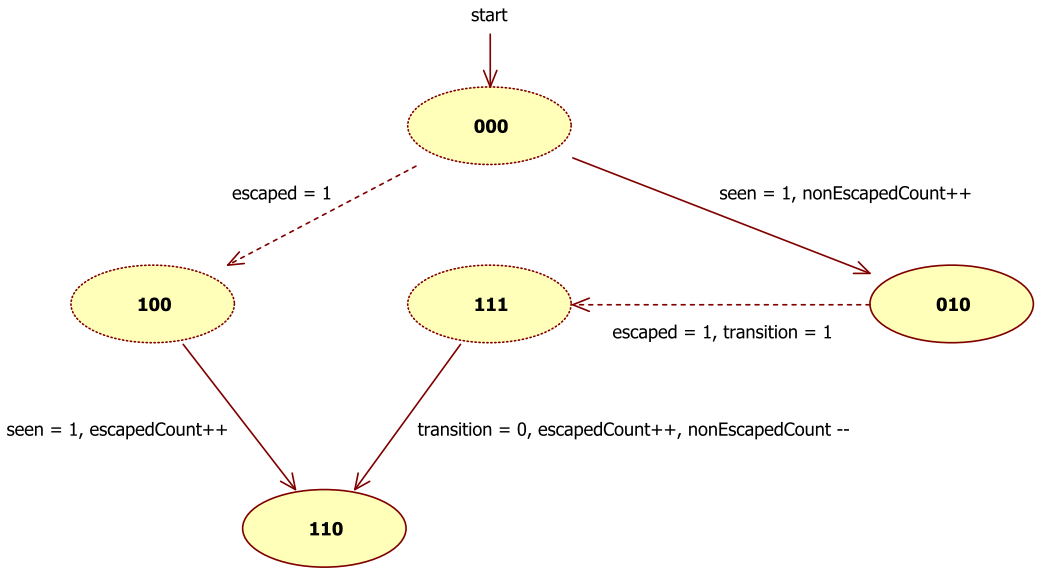


Figure 3.9: Transition Diagram

seen object (010 state) or an escaped seen object (110 state). These final states are depicted with solid borders in the diagram. The dotted transitions are performed in the write barrier, while the solid ones happen in the GC.

3.1.4 Saving the Data

After performing the heap traversal and updating the counters stored in *TlrgcEscapeData*, the collected data needs to be saved in a file to be analyzed later. To simplify parsing and aggregation of the collected data, the results are split into the following files:

- One file for the global escape data.

- One file for the class based escaped data per GC.

The data is written separated by commas to enable further handling with Excel as well as automated parsing and aggregation. Figure 3.10 shows an example of the created csv file structure.

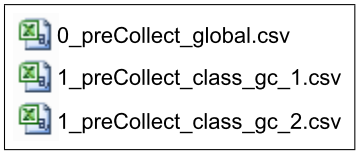


Figure 3.10: Structure of the created csv files.

3.1.5 Verification

For a reliable escape analysis result, it is important to verify the correctness of the instrumentation code with focus on the escape marking and counting. The performed verification checks are described in this section.

3.1.5.1 Internal Traversal

As stated in Section 3.1.2.2, the escape marking needs to be cascaded to all object slots, which is performed by the recursive call of *tlrgcMarkAsEscaped()*. If this code is working correctly, all object slots of an escaped object appearing at the write barrier should also be marked as escaped.

```
void tlrgcVerifyEscapingObjectSlotsCallback(VMObject object) {
    Class *objectClass = TLRGC_GET_OBJECT_CLASS(object);

    char objectClassString[200];
    tlrgcGetObjectClassAsString(objectClass, objectClassString);

    if (!TLRGC_OBJECT_GET_ESCAPED(object)) {
        printf("ALERT - object %p [%s] not marked as
        escaped - FAIL !!!\n", object, objectClassString);
    }

    Assert_true(TLRGC_OBJECT_GET_ESCAPED(object));
}
```

Figure 3.11: Verification callback for iterating all object slots.

To verify this, an additional verification traversal of the object slots was implemented. Figure 3.11 shows the verification method called by the object slot iterator to assert that the *slotObject* is also marked as escaped.

With n representing the number of objects on the heap, the verification traversal has a worst case complexity of $\Omega(n^2)$ when all heap objects are referenced by all other objects. Even with more likely scenarios, the overhead is large and would influence the benchmark performance. Due to the fact that all the benchmarks and tests are deterministic and perform the same workload at each run, the verification traversal was only performed once per benchmark/test and turned off, if the test was positive.

3.1.5.2 Escape Marking Verification Test

The Escape Marking Verification Test is a JUnit Test Case that verifies the correct implementation of the Transition Diagram depicted in Figure 3.9. The test case explicitly performs garbage collections by calling *System.gc()*. The JVM was instrumented to print the *escape*, *seen* and *transition* bits for each object passing the write barrier during the execution of the test. Additionally, the created csv result files were manually checked for the expected results.

```
public class MultiThread extends Thread {

    private MyObject objectSlot1;
    private MyObject objectSlot2;

    private List<MyObject> listSlot1;
    private List<MyObject> listSlot2;

    public void run() {
        System.out.println("Start MultiThread");

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            System.out.println(e);
        }

        System.out.println("End MultiThread");
    }
}
```

Figure 3.12: MultiThread class (without get and set methods).

All multi-threaded tests use the *MultiThread* class depicted in Figure 3.12 that runs concurrently for 5 seconds without performing any work. The test methods store objects into the object and list slots of *MultiThread*, so that

they will be covered by the write barrier and should be correctly processed by the *tlrgcMarkAsEscaped()* method.

The following test methods were implemented:

- **testSingleThreaded():** Allocate 10 normal objects and a linked list with 10 objects. None of these objects should escape.
- **testMultiThreadedSingleObject():** Test *nonEscaping*, *directEscaping* and *transitionEscaping* with normal objects.
- **testMultiThreadedInlineAllocation():** Test *nonEscaping*, *directEscaping* and *transitionEscaping* with objects that have an inline allocated object.
- **testMultiThreadedLinkedList():** Test *nonEscaping*, *directEscaping* and *transitionEscaping* with linked lists that contain 10 objects each.

The source code of these tests is available in Appendix B (Figure B.1 - B.4). The expected results for the *escape*, *seen* and *transition* bits at each GC are listed in a comment block before each call to *System.gc()* in the multi-threaded test cases.

The Escape Marking Verification Test covers every possible scenario for the escaping of objects and could therefore successfully verify that the escape marking and counting code is working as expected.

3.2 R&D Results

After instrumenting the JVM, an escape analysis was performed for the R&D project. This escape analysis used the class of an object as criteria for a fine-grained analysis. The measurements to gain the necessary data were performed with five different benchmarks. Due to the fact that *SimpleTest* and *Eclipse startup* with an empty workspace only ran two and one garbage collections respectively, their results are excluded from the tables in this section.

Benchmark	**esc #**[1]	**nEsc #**[2]	**total #**[3]	**esc**[4]	**nEsc**[5]	**obj**[6]
EclipseNormal	846,195	15,167	861,362	98%	2%	2.23%
SPECjbb2005	4,280,572	1,906	4,282,478	100%	0%	4.05%
SPECjvm2008	53,775,547	1,046,700	54,822,247	98%	2%	3.47%

[1] Number of Escaped Objects [2] Number of Non-Escaped Objects
[3] Total Number of Objects [4] Percentage of Escaped Objects [5] Percentage of Non-Escaped Objects [6] Percentage of Total Objects

Table 3.1: Top 10 escaping objects

The summarized result of the Top 10 escaping objects ($> 90\%$ escape percentage) is shown in Table 3.1. It shows that the class based approach can only identify a small amount of the total objects as definitely escaping. The results in Table 3.2 show that 25% - 75% of the objects fall into the Top 10 non-escaping objects ($\leq 10\%$ escape percentage) criteria. Unfortunately they still have an escape percentage of 1-5% that does not allow a clear identification of non-escaping objects based on the class. A further look into the gathered class based data supports this impression. It can be recognized that the two classes *java/lang/String* and *char* (used in arrays) were hotspots during each

Benchmark	esc #[1]	nEsc #[2]	total #[3]	esc[4]	nEsc[5]	obj[6]
EclipseNormal	1,490,683	27,096,652	28,587,335	5%	95%	74.17%
SPECjbb2005	805,188	26,911,166	27,716,354	3%	97%	26.21%
SPECjvm2008	7,855,178	873,246,798	881,101,976	1%	99%	55.79%

[1] Number of Escaped Objects [2] Number of Non-Escaped Objects [3] Total Number of Objects [4] Percentage of Escaped Objects [5] Percentage of Non-Escaped Objects [6] Percentage of Total Objects

Table 3.2: Top 10 non-escaping objects

benchmark. In *EclipseNormal* nearly 49%, in *SPECjbb2005* about 60% and in *SPECjvm2008* about 16% of the objects are an instance of one of these two classes. The relative low value of 16% at *SPECjvm2008* can be explained, because the benchmark itself covers a wide area, so that different sub-tests with different classes are used and not always the same classes during the whole runtime of the benchmark. The escape percentage of the two object classes varies from 8% and 15% up to 47% up and 65%. It is not useful to statically determine if these objects will escape. Even a runtime decision might be hard due to the wide variety of the results.

Based on the class, no reliable pattern could be identified when to allocate objects thread-locally without a possible need to copy them back to the global heap. This led to the assumption that it is necessary to consider other metrics/parameters to successfully identify a reproducible escape behaviour of objects.

Chapter 4

Approach

Based on the previous analysis, this thesis will use a different approach to distinguish between escaping and non-escaping objects. Instead of only using the class of the allocated objects as criteria, the Program Counter (PC) will be used. That means the collected escape data of an object will be saved more specifically, because the escaping of an object is broken down to the allocation line of code and not only to the object class. This should result in a more fine-grained analysis data compared to the R&D project and the possibility to identify a predictable pattern.

If the class based statistics for example show an escape percentage of 50% and the objects of this class are allocated at two spots, it might be possible that the objects of the first allocation spot always escape and the ones from the second never. If the analysis would yield this result, it would be possible

to always allocate the objects on the heap in the first case and always on a thread-local region in the second case.

The research in this thesis is divided into the following parts:

1. Instrumentation
2. Measurements
3. Analysis
4. Prototype implementation
5. Evaluation

The instrumentation part will add the necessary methods and extensions to the Java Virtual Machine source code to measure the escaping of objects and store this escape data based on the PC. The collected escape data will be stored in files on disk for further investigations.

After the code is instrumented, the following benchmarks are executed:

- Eclipse Startup
- SPECjbb2005 [26]
- SPECjvm2008 [27]

The resulting PC based escape data is analyzed and compared to the previous class based escape data. The gathered results provide a further insight into the escaping of objects and the possibility to identify patterns. Depending on the collected results, further test cases are taken into account, which can be other external benchmarks or self implemented tests to produce a certain behaviour.

If the analysis identifies a reproducible pattern and therefore a reliable amount of objects eligible for a thread-local allocation, the amount of heap allocated objects can be reduced. This would lead to fewer and shorter garbage collections and therefore increase the overall performance of the Java Virtual Machine. The following questions arise for the realization of the thread-local allocation:

- How to differentiate between thread-local and global heap allocation?
- Where to allocate the thread-local objects? Should we make use of the existing stack frames or reserve a special area on the heap?
- How to organize thread-local objects of different threads in a special allocation area?
- How to copy objects back to the heap and update all reference pointers, if the objects are accessed by other threads (escaping)?

- How to create and delete the objects in the thread-local regions? Should we make use of barriers?
- How to perform a thread-local garbage collection?

This thesis will address these questions by implementing a proof-of-concept prototype for the Thread Local Region allocation and garbage collection.

Finally, the work done in this thesis will be reviewed and an outlook to future work and improvements given.

Chapter 5

Instrumentation of the JVM

The work in this chapter is based on the previously realized R&D project presented in Chapter 3. The instrumentation code of the JVM has been extended to enable the gathering of PC based escape data. This chapter will only discuss the changes made compared to the R&D approach.

5.1 Extension of the VMObject

The R&D already added the *allocationThreadAndFlags* field to the *VMObject*, which holds a reference to the allocation thread and the three flags *escaped*, *seen* and *transition*. The PC based approach additionally needs to store the program counter in the *VMObject*.

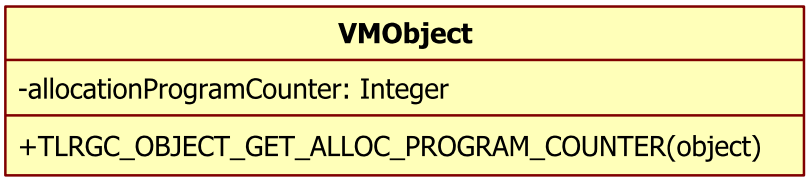


Figure 5.1: Extension of the VMObject

As depicted in Figure 5.1 the newly added field is called *allocationProgram-Counter* and also initialized at the allocation of an object.

To simplify the access to the allocation program counter, the following macro has been defined:

- **TLRGC_OBJECT_GET_ALLOC_PROGRAM_COUNTER:** Return the program counter associated with the object.

5.2 Escape Marking

The class based implementation did only mark objects as escaped at the write barrier. Some objects exist that are guaranteed to escape at allocation time, so that they can directly be allocated on the global heap. The JVM was modified to identify the following special cases and directly mark the corresponding objects as escaped:

- **Classes:** Each class object is only held once per JVM, so that they are accessible by every thread.

- **Static Variables:** Class static variables are accessible by every thread.
- **Interned Strings:** The JVM reuses interned Strings and char arrays, so that these objects need to be marked as escaped as well.
- **JNI Global References:** Objects that are made available via JNI are globally accessible.

All other write barrier checks and the recursive escape marking are still working the same way as in the R&D project.

5.3 Collecting Data

As already stated in Section 3.1.3, the JVM calls the methods *preCollect()* and *postCollect()* before and after each garbage collection. Both methods were used in the R&D to hook in code for collecting the escape analysis data. It is possible that *postCollect()* might see less objects on the heap due to the reclamation of dead objects during the GC. This especially affects short-living objects that already appear dead at their first garbage collection. It is necessary for the PC based escape analysis to cover these objects as well, so that the decision to only collect data in the *preCollect()* method was made. Figure 5.2 shows the updated hook in process in a sequence diagram.

To enable the collection of PC based escape analysis data, the R&D data

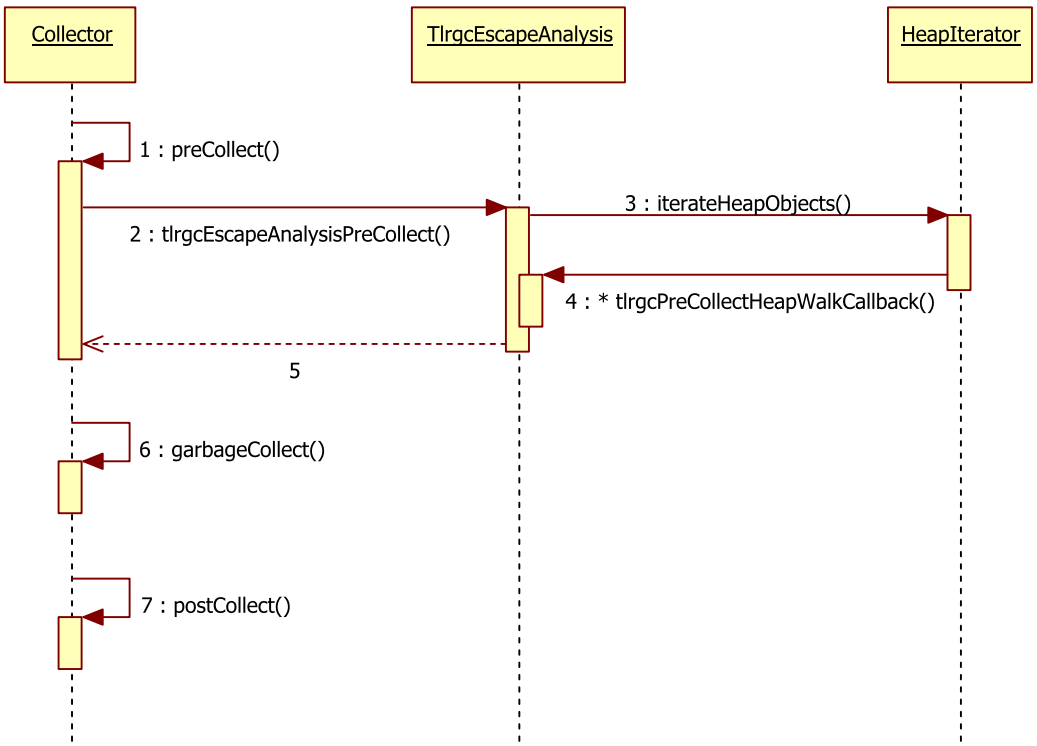


Figure 5.2: Sequence diagram of the data collection.

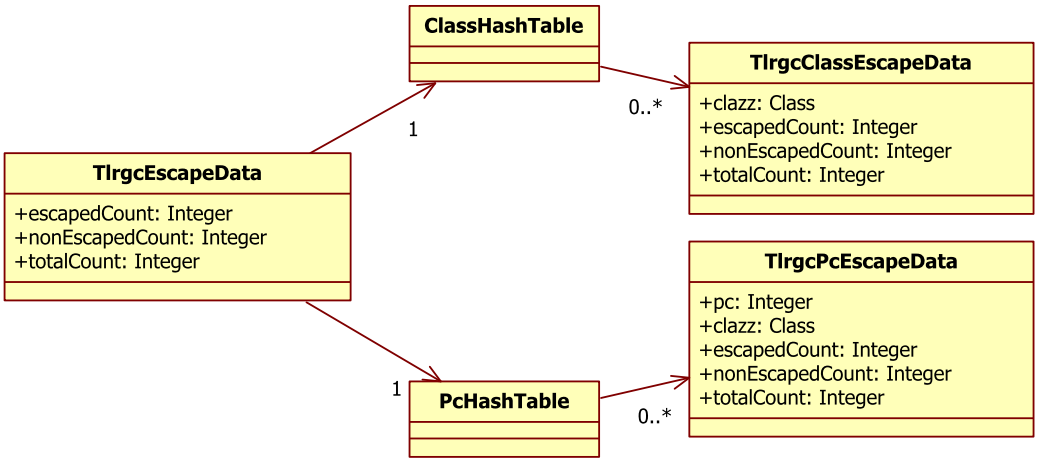


Figure 5.3: Class diagram of the escape data structures.

structure was extended as shown in Figure 5.3. *TlrgcClassEscapeData* still stores the class based escape data, while *TlrgcPcEscapeData* is newly created for the PC based escape data. *TlrgcEscapeData* acts as a top container and now holds the global escape counts as well as the two HashTables *ClassHashTable* and *PcHashTable*. These HashTables use the class and PC as keys and the respective instances of *TlrgcClassEscapeData* and *TlrgcPcEscapeData* as entries. All three data structures contain counters for escaped, non-escaped and total objects. The top container *TlrgcEscapeData* is now globally stored as a class static variable in the garbage collector, so that the escape data can be summarized over the whole runtime of the application. This facilitates the aggregation and evaluation of the escape data, because a detailed view per GC was not a requirement for the analysis performed for this thesis.

The method *tlrgcEscapeAnalysisPreCollect()* described in Section 3.1.3 was adapted to the newly introduced data structures and modified to handle the PC based escape counting.

5.4 Saving the Data

The code to save the escape data was extended to also cover and write the PC based data. This new instrumentation code now creates the following files:

- One file for the global escape data collected over the whole runtime of the program.
- One file for the class based escaped data per GC.
- One file for the PC based escaped data per GC.

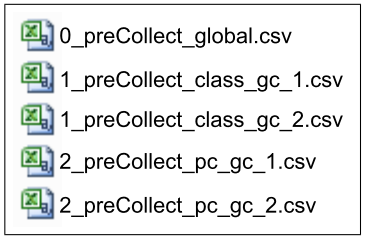


Figure 5.4: Structure of the created csv files.

Figure 5.4 shows an example of the extended csv file structure. The content of an example PC based csv file can be found in Appendix A. The data written per GC is now a snapshot of the whole execution up to the current GC and not only the difference compared to the last GC. This enables an easier analysis of the development of the escaping of objects on the heap.

Chapter 6

Measurements

After the JVM was extended with the necessary code, multiple tests and benchmarks were executed to gather data for the escape analysis. The benchmarks introduced in this chapter were chosen based on the ability to create a realistic representation for the usage of objects in the Java Virtual Machine.

6.1 SimpleTest

The *SimpleTest* is a simple Java test class that explicitly triggers two garbage collections via *System.gc()* as shown in Figure 6.1. This test was mainly used for debugging the escape analysis instrumentation code.

```
package ca.ibm.casa.tlrgc;

public class SimpleTest {

    public static void main(String[] args) {
        System.gc();
        System.gc();
    }

}
```

Figure 6.1: *SimpleTest* source code

6.2 TestDifferentObjects

TestDifferentObjects was created to verify the correct determination of the program counter and associated line of code in the PC based escape data.

The test case contains the following steps:

1. Allocate a String and an Integer object
2. Perform the $1st$ GC
3. String Concatenation of 100 Integer values
4. Allocation of an object with a self written object class
5. Perform the $2nd$ GC
6. Allocate two objects in one line of code (inline allocation)
7. Perform the $3rd$ GC

The explicit triggering of the garbage collections ensures the possibility to easily identify the corresponding PC in the escape data output. The source code of *TestDifferentObjects* is listed in Figure 6.2.

```
package ca.ibm.casa.tlrgc;

public class TestDifferentObjects {

    public static void main(String[] args) {

        // allocate two standard Java objects
        String string = new String();
        int number = 42;

        // perform 1st GC
        System.gc();


        // String concatenation
        String count = "";
        for (int i = 0; i < 100; i++) {
            count += ", " + String.valueOf(i);
        }

        // allocate object with self written class
        MyObject my = new MyObject();

        // perform 2nd GC
        System.gc();


        // two allocations at one line of code
        MyObject insideAlloc = new MyObject(new String("Inline allocation"));

        // perform 3rd GC
        System.gc();

    }

}
```

Figure 6.2: *TestDifferentObjects* source code

6.3 Eclipse Startup

The startup of the Eclipse IDE normally takes a long time that cannot be productively used by the developer. It would be good, if the thread-local allocation of objects could speed up the startup by reducing the amount and time spent in global GCs. To measure this possible enhancement, the following two test scenarios for the startup of the Eclipse IDE were taken into account:

- **EclipseEmpty:** Start of the Eclipse IDE with a newly created workspace without any projects.
- **EclipseNormal:** Start of the Eclipse IDE with the normal workspace of the IBM research project with over 80 opened projects.

Both test cases use the Eclipse IDE for C/C++ Developers in the Indigo release version [28].

6.4 SPECjbb2005

The *SPECjbb2005* benchmark evaluates the performance of the server side part of Java by emulating a three-tier client/server application. It is concentrated on the middle tier and exercises the implementation of the JVM itself, the JIT compiler, the GC, thread handling and some aspects of the operating system [26].

The highlights of the benchmark are (taken from [26]):

- Emulates a three-tier system, the most common type of server-side Java application today.
- Business logic and object manipulation, the work of the middle tier, predominate.
- Clients are replaced by driver threads, database storage by binary trees of objects.
- Increasing amounts of workload are applied, providing a view of scalability.

Due to the fact that JavaEE environments are nowadays considered to be three-tiered applications [29], the *SPECjbb2005* benchmark gives a good insight into the escaping of objects in Java business applications.

6.5 SPECjvm2008

The *SPECjvm2008* benchmark measures the performance of the Java Runtime Environment (JRE) by executing several real life applications and benchmarks. In combination of all those tests, the key duty of *SPECjvm2008* is to exercise the Java core functionality.

The highlights of the benchmark are (taken from [27]):

- Leverages real life applications (e.g. derby, sunflow, and javac) and area-focused benchmarks (e.g. xml, serialization, crypto, and scimark).
- Also measures the performance of the operating system and hardware in the context of executing the JRE.

6.5.1 Applications and Benchmarks

The following applications and benchmarks are included in the *SPECjvm2008* suite [30]:

- **Startup:** Starts each of the following benchmark for just one operation.
- **Compiler:** Uses the OpenJDK compiler to compile a set of *.java* files. The workload of the compilation is *javac* itself and the source code of the sunflow benchmark.
- **Compress:** This benchmark compresses data by using a modified Lempel-Ziv method.
- **Crypto:** This benchmark focuses on crypto methods and is split into the following three sub-benchmarks:
  - **aes** → encrypt and decrypt with AES and DES

  - **rsa** → encrypt and decrypt with RSA
  - **signverify** → sign and verify with MD5withRSA, SHA1withRSA, SHA1withDSA and SHA256withRSA

- **Derby:** This benchmark is based on the open-source relational database Derby [31], which is entirely implemented in Java. In combination with business logic the benchmark focuses on *BigDecimal* computations and database logic.

- **MPEGaudio:** Utilizes the MP3 decoding and is a floating point heavy benchmark.

- **SciMark:** The *SciMark* benchmark developed by the National Institute of Standards and Technology (NIST) is a benchmark for scientific and numerical computing [32] and widely used by the industry as a floating point benchmark. Each of the sub-tests (fft, lu, monte_carlo, sor and sparse) are executed with a "large" dataset of 32 MB and a "small" dataset of 512 KB, which stress different parts of the JVM.

- **Serial:** Performs serialization and deserialization of primitives and objects.

- **Sunflow:** Tests graphic visualization by using the open-source global illumination rendering system Sunflow [33].

- **XML:** Performs XML transformation and validation.

6.5.2 Limitations

The *SPECjvm2008* benchmark is comprehensive and a good test to gather escape data in different application fields related to the Java core functionality. Due to the fact that the Startup sub-benchmark justs starts each benchmark for one operation, the produced escape data is not really representative. The XML benchmark led to a VM crash that was probably caused by the added instrumentation code. Unfortunately this crash could not be fixed in the available time, so that the Startup and XML sub-benchmark were excluded from the measurement runs.

Chapter 7

Analysis Results

This chapter presents the escape analysis results obtained and aggregated from the measurements described in the previous chapter. The measurements were performed with both the Generational and the Balanced GC policies. The class based instrumentation of the JVM in the R&D project was only performed for the Generational GC policy, so that the presented results in this chapter will be based on this GC policy for an easier comparison of the PC and class based approaches. The gathered PC based data is almost identical for both GC policies, so that this selection is not limiting the results.

7.1 Global Escape Results

The results in this chapter are globally collected over the whole runtime of the program. Table 7.1 shows the global escape results for all six benchmarks:

SimpleTest, *TestDifferentObjects*, *EclipseEmpty*, *EclipseNormal*, *SPECjbb-2005* and *SPECjvm2008*.

Both *SimpleTest* and *TestDifferentObjects* allocated a total of 5-6K objects and almost all of these objects are created during the JVM startup. The garbage collections in both tests were manually triggered, because no GC would have been automatically performed with such a low object count. The garbage collections in the four other benchmarks were automatically triggered by the JVM. *EclipseEmpty* allocated about 1.3M, *EclipseNormal* about 38.5M, *SPECjbb2005* about 106M and *SPECjbb2008* about 1,610M objects. The escape percentages for the first three benchmarks that only perform a few GCs are relatively high. This means that many JVM internal objects and the basic objects of the Eclipse Startup process are shared between multiple threads. *SimpleTest*, *TestDifferentObjects* and *EclipseEmpty* only performed a small number of garbage collections, are therefore not representative and will be excluded from further evaluation.

The diagram in Figure 7.1 displays the escape percentage of the different benchmarks as well as the averaged total. It is interesting to see that the highest escape percentage is only 56% in *SimpleTest*, which is basically the JVM startup. 44% of the objects created during the startup are therefore non-escaping and could be allocated thread-locally. The longer a program runs, the fewer objects escape, which leads to the conclusion that in general

Benchmark	GC[1]	esc #[2]	non-esc #[3]	total #[4]	esc[5]	non-esc[6]
SimpleTest	2	3,011	2,400	5,411	56%	44%
TestDiffObj[7]	3	3,028	2,955	5,983	51%	49%
EclipseEmpty	1	519,963	776,304	1,296,267	40%	60%
EclipseNormal	19	4,667,568	33,874,955	38,542,523	12%	88%
SPECjbb2005	48	44,108,304	61,629,519	105,737,823	42%	58%
SPECjvm2008	743	271,049,699	1,340,605,024	1,611,654,723	17%	83%
Total	816	320,351,573	1,436,891,157	1,757,242,730	18%	82%

[1] Garbage Collection Count [2] Number of Escaped Objects [3] Number of Non-Escaped Objects
[4] Total Number of Objects [5] Percentage of Escaped Objects [6] Percentage of Non-Escaped Objects
[7] TestDifferentObjects

Table 7.1: Global Escape Results

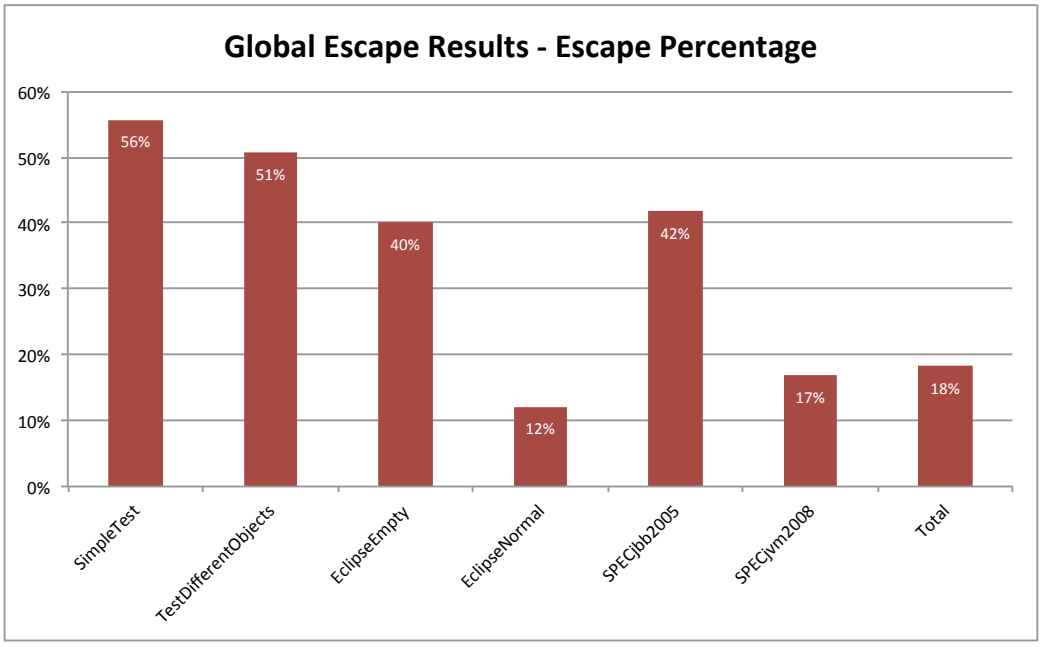


Figure 7.1: Global Escape Results - Escape Percentage

at least 50% of the objects on the heap are non-escaping and eligible for TLR allocation. Even in *SPECjbb2005* that emulates a three-tiered application and shares much data between multiple threads, the escape percentage is only 42%. Depending on the nature of the program or benchmark it is even possible to have non-escape percentages above 80%, like in *EclipseNormal* and *SPECjvm2008*.

This is a very encouraging result and the realization of the TLR allocation idea should bring a large performance boost.

7.2 Class Based vs. PC Based

After the global escape data was analysed, a closer look into the collected class and PC based data was taken. This section concentrates on the comparison of the results between the class and PC based approaches. The goal of both approaches is to break down the global non-escape results to classes and PCs to identify even more objects that almost never escape. These objects can be allocated in TLR with minimal penalty for copying escaping objects later on.

The following two diagram types will be used to present the comparison results in this section:

- **Escape Results:** This diagram shows the object percentage for escaping in 10% blocks, e.g. an object percentage of 50% in the 1-10% block means that 50% of the allocated objects are covered by classes or PCs that have a total escape percentage of 1-10%.

- **Cumulative Escape Results:** This diagram cumulates the different escape blocks, so that e.g. an object percentage of 80% at 20% escape percentage means that 80% of the total objects are covered by classes or PCs that have a total escape percentage of up to 20%.

7.2.1 EclipseNormal

The results for *EclipseNormal* are similar for the class and PC based approaches. 85% of the objects can be covered by classes and PCs that escape in less than 20% of the cases. The only real difference is that the PC based approach can identify 61% of the objects as completely non-escaping, while the class based approach could just identify 24%. This means that these 61% of the objects can be allocated on TLR without the necessity to copy them later on. When all objects with an escape percentage of up to 10% are allocated on TLR, the class based approach needs to copy out 6% escaped objects while the PC based only needs to copy 1% of the total objects back to the global heap. The normal and cumulative escape results for *EclipseNormal* are shown in Figure 7.2 and 7.3.

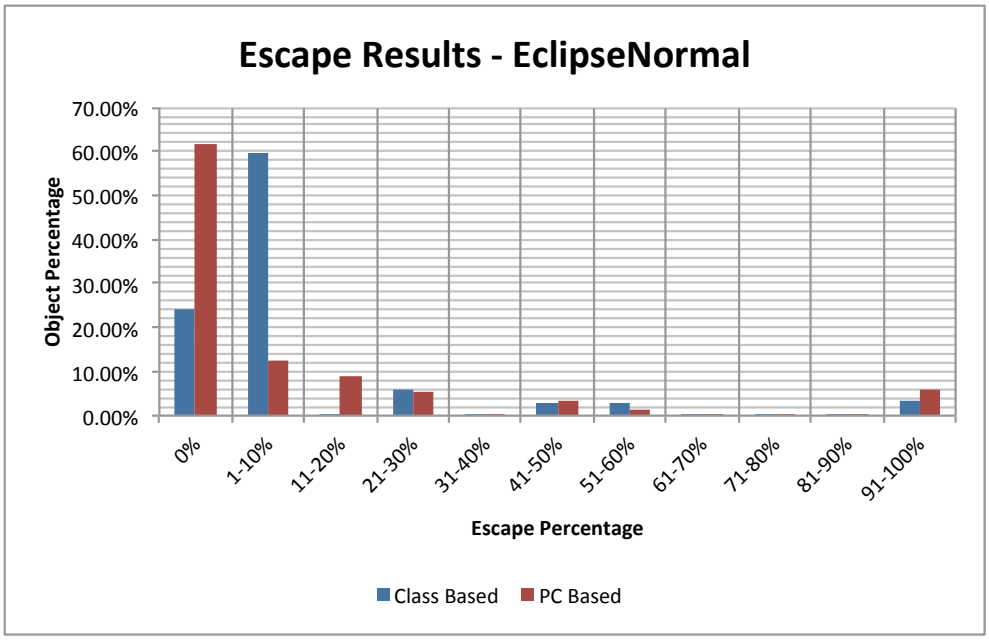


Figure 7.2: Escape Results - EclipseNormal

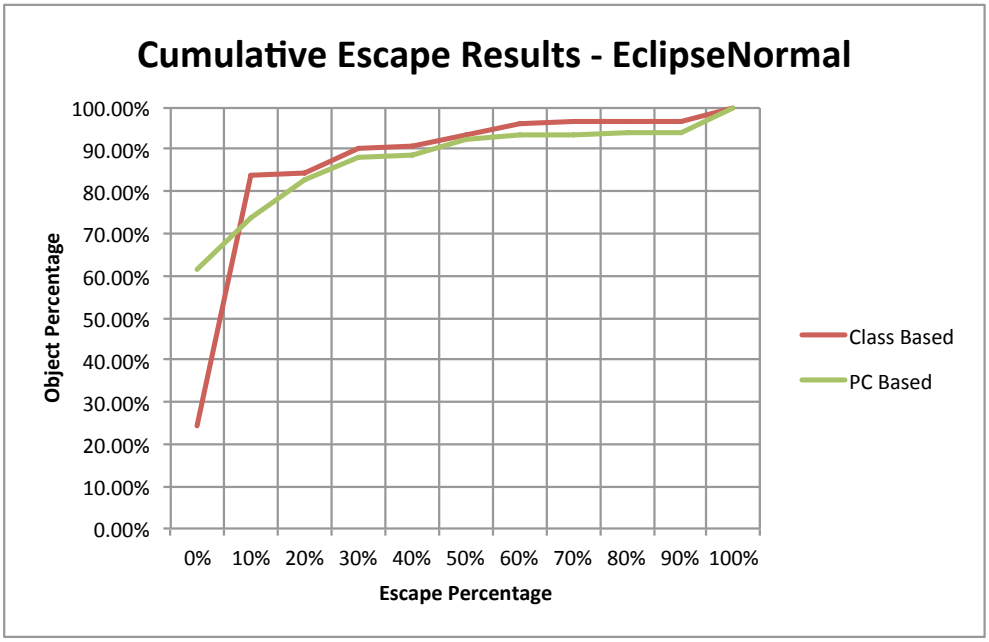


Figure 7.3: Cumulative Escape Results - EclipseNormal

7.2.2 SPECjbb2005

The *SPECjbb2005* benchmark is the only benchmark where the values of the class and PC based approaches differ significantly. This can be explained with the fact that this benchmark uses the same object classes locally and shared with other threads depending on the PC. The summary on class level therefore blurs the real fine-grained PC based values.

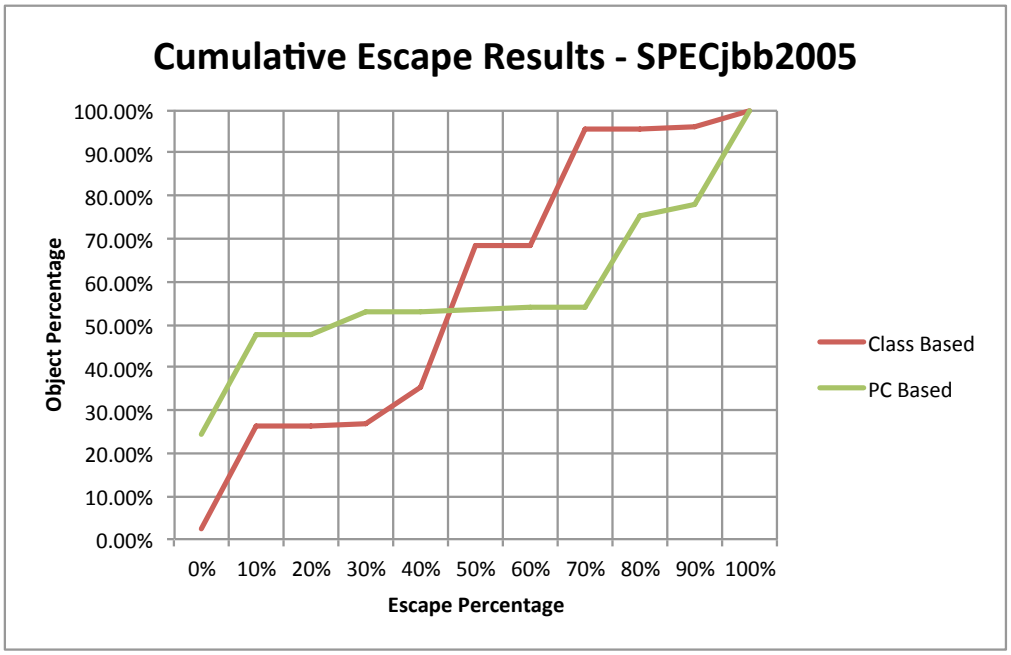


Figure 7.4: Cumulative Escape Results - SPECjbb2005

The PC based curve displayed in the cumulative escape results diagram in Figure 7.4 looks like stairs, while the class based curve is growing more steadily. The class based curve has two steep jumps at around 40% and 70% of escaping, which means that many objects fall into this category and would

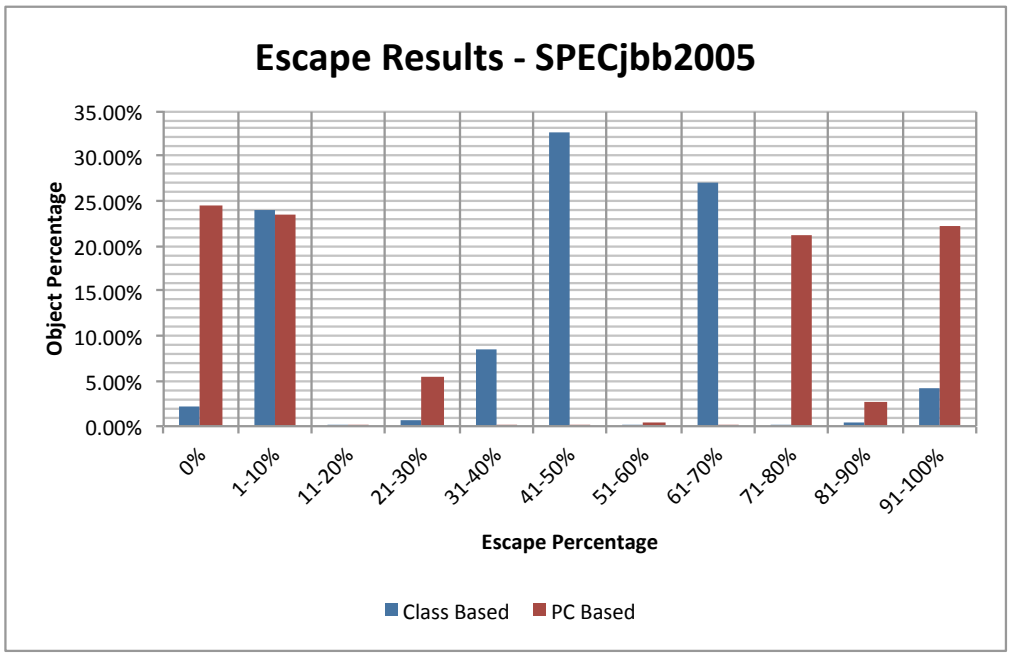


Figure 7.5: Escape Results - SPECjbb2005

lead to a poor escape prediction. The PC based approach instead has a flat behaviour with only jumps below 10% and above 70%. This is enables a good escape prediction, because the allocated objects can be clearly separated into escaping and non-escaping groups.

The PC based approach could identify 24% of the objects with 0% escaping and almost 50% with up to 10% escaping as depicted in Figure 7.5. This 50% level is constant until the 70% escaping level is reached. 43% of the objects could be identified with an escape percentage of more than 70%. The class based approach instead could only identify 2% non-escaping objects and 26% of the objects with an escape percentage of up to 10%. Additionally, 68%

of the objects in the class based approach fall into the wide spread area of 31-70% escaping. This leads to the assumption that the PC based approach is the best way to recognize objects that either escape in less than 10% or in more than 70% of the cases.

7.2.3 SPECjvm2008

The class and PC based results for *SPECjvm2008* are almost identical above the level of 20% escaping (see Figure 7.6 and 7.7). The interesting fact is that the PC based approach can identify 74% of the objects with 0% and about 80% of the objects with up to 10% escape percentage. The class based approach instead can only identify 49% of the objects with 0% escaping and around 15% of the objects for each of the 1-10% and 11-20% block. This means that both approaches can identify about 80% of the objects with up to 20% escape percentage, while the PC based approach is more concrete and can already identify this figure at the level of 10% escaping.

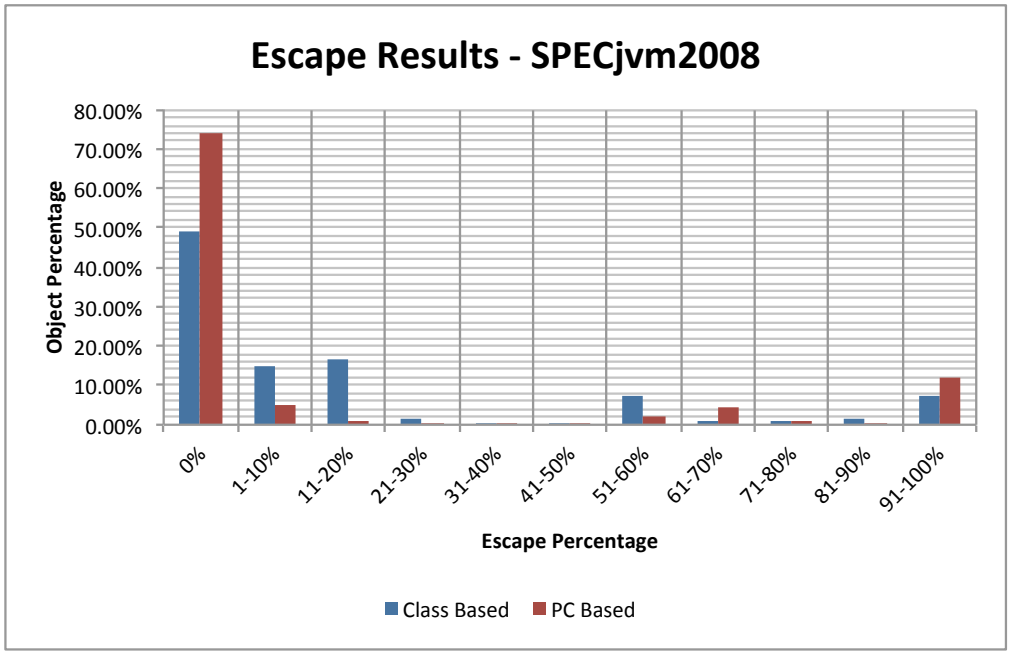


Figure 7.6: Escape Results - SPECjvm2008

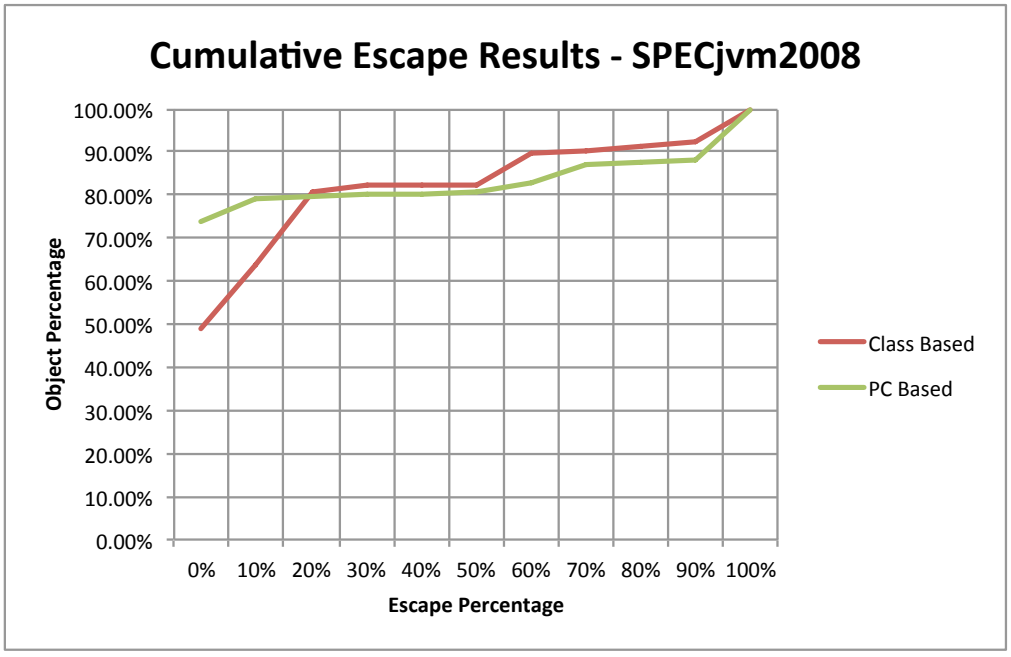


Figure 7.7: Cumulative Escape Results - SPECjvm2008

7.3 PC Based Results in Detail

The previous section showed that the PC based approach performs better than the class based approach in identifying eligible objects for TLR allocation. This section will provide some additional insights by analyzing the Top 10 of allocated, escaping and non-escaping objects. The selection criteria for the Top 10 of escaping and non-escaping objects is defined as follows:

- **Escaping:** PCs with an escape percentage $> 90\%$
- **Non-Escaping:** PCs with an escape percentage $\leq 10\%$

7.3.1 Top 10 allocated objects

Benchmark	**esc #**[1]	**nEsc #**[2]	**total #**[3]	**esc**[4]	**obj**[5]
EclipseNormal	873,431	13,769,667	14,643,098	6%	37.99%
SPECjbb2005	37,333,854	50,066,619	87,400,473	43%	82.66%
SPECjvm2008	748,364	587,482,066	588,230,430	0%	37.25%

[1] Number of Escaped Objects [2] Number of Non-Escaped Objects [3] Total Number of Objects [4] Percentage of Escaped Objects [5] Percentage of Total Objects

Table 7.2: Top 10 allocated objects

Table 7.2 shows the Top 10 of the allocated objects in the PC based escape data. It is really interesting that the Top 10 program counters can cover about 37% of the object allocations in *EclipseNormal* and *SPECjvm2008* and even 83% in *SPECjbb2005*. The detailed tables are available in the appendix as Tables C.1 - C.3. In *EclipseNormal* all Top 10 objects and

in *SPECjbb2005* 62% of the objects are allocated in *java/lang* or *java/util* classes. *SPECjvm2008* instead has many allocations in user-generated classes. Especially the Top 2 allocated objects in *SPECjbb2005* are interesting, because they cover 44.5% of the total objects.

7.3.2 Top 10 escaping objects

Benchmark	**esc #**[1]	**nEsc #**[2]	**total #**[3]	**esc**[4]	**obj**[5]
EclipseNormal	1,055,775	17,576	1,073,351	98%	2.78%
SPECjbb2005	21,787,674	1,018	21,788,692	100%	20.61%
SPECjvm2008	77,628,274	1,488,539	79,116,813	98%	5.01%

[1] Number of Escaped Objects [2] Number of Non-Escaped Objects [3] Total Number of Objects [4] Percentage of Escaped Objects [5] Percentage of Total Objects

Table 7.3: Top 10 escaping objects

The Top 10 of escaping objects are depicted in Table 7.3. In *SPECjbb2005* 50% of the escaping objects are covered by the Top 10 program counters, *EclipseNormal* and *SPECjvm2008* instead can only catch around 25%. This observation is supported by the detailed data in Tables C.4 - C.6, because even the Top PC in *EclipseNormal* and *SPECjvm2008* only covers 0.42% and 0.95% of the objects. This leads to the assumption that the escaping spots are either widely spread or most of the escaping objects are not covered by the criteria of $> 90\%$ escaping. In *SPECjbb2005* all 10 program counters have a 100% escape percentage and the Top PC covers 11.30% of the total objects. The object tree structure $Order \rightarrow Orderline \rightarrow String$ is always

escaping and *Order* can therefore be considered as the main shared object structure between the different worker threads in *SPECjbb2005*.

7.3.3 Top 10 non-escaping objects

Benchmark	**esc #**[1]	**nEsc #**[2]	**total #**[3]	**esc**[4]	**obj**[5]
EclipseNormal	139,963	12,222,228	12,362,191	1%	32.07%
SPECjbb2005	793,932	45,818,874	46,612,806	2%	44.08%
SPECjvm2008	748,364	587,482,066	588,230,430	0%	37.25%

[1] Number of Escaped Objects [2] Number of Non-Escaped Objects [3] Total Number of Objects [4] Percentage of Escaped Objects [5] Percentage of Total Objects

Table 7.4: Top 10 non-escaping objects

The Top 10 results for non-escaping objects in Table 7.4 are very promising. 32%, 44% and 37% of the total objects can be covered by the Top 10 non-escaping program counters. If these objects can all be allocated on TLR and almost never need to be copied back to the heap, a large performance improvement should be possible. The detailed tables for this section are available in the appendix as Table C.7 - C.9. In *EclipseNormal* all Top 10 objects are allocated in *java/lang* and *java/util* classes, which enables the instrumentation of these classes to directly allocate the objects of the corresponding PCs on TLR memory during *Eclipse* startup. The run with *SPECjbb2005* could identify a spot in *Integer.java* on line 642 that is only escaping in 3% of the cases, but covers 23.25% of the total objects. A TLR allocation of this single PC should gain significant improvements in the *SPECjbb2005* bench-

mark performance. In *SPECjvm2008* the Top 10 objects are distributed over internal Java classes and user-generated classes. Nevertheless 9 out of 10 Top 10 program counters have a non-escape percentage of 100% while the $10th$ PC reaches 99%. Additionally the Top 10 of non-escaping objects are exactly matching with the Top 10 of allocated objects. This means that *SPECjvm2008* allocates many objects at program counters that almost never escape, which is good for allocating them on TLR.

7.4 Conclusion

The previous sections showed that the PC based approach is more suitable than the class based approach to identify non-escaping objects and determine escape patterns. Figures 7.8 and 7.9 show the PC based normal and cumulative escape results for *EclipseNormal*, *SPECjbb2005* and *SPECjvm2008*. The diagrams indicate that the escaping per program counter can be split into the following three categories:

1. **Almost never escape:** $\leq 10\%$ escape percentage
2. **Unpredictable escaping:** $11 - 90\%$ escape percentage
3. **Almost always escape:** $> 90\%$ escape percentage

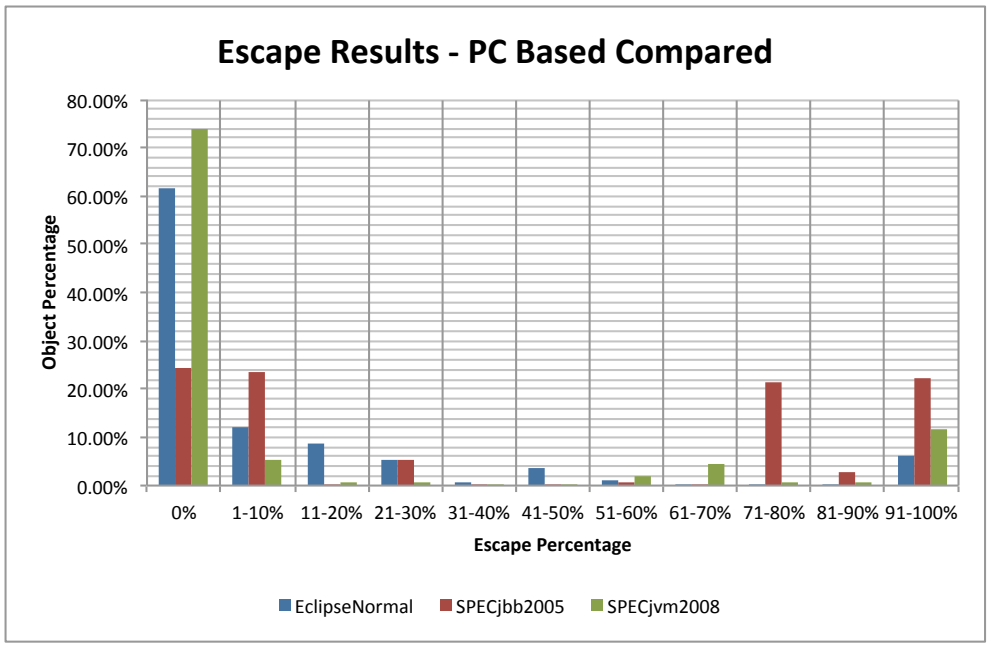


Figure 7.8: Escape Results - PC Based Compared

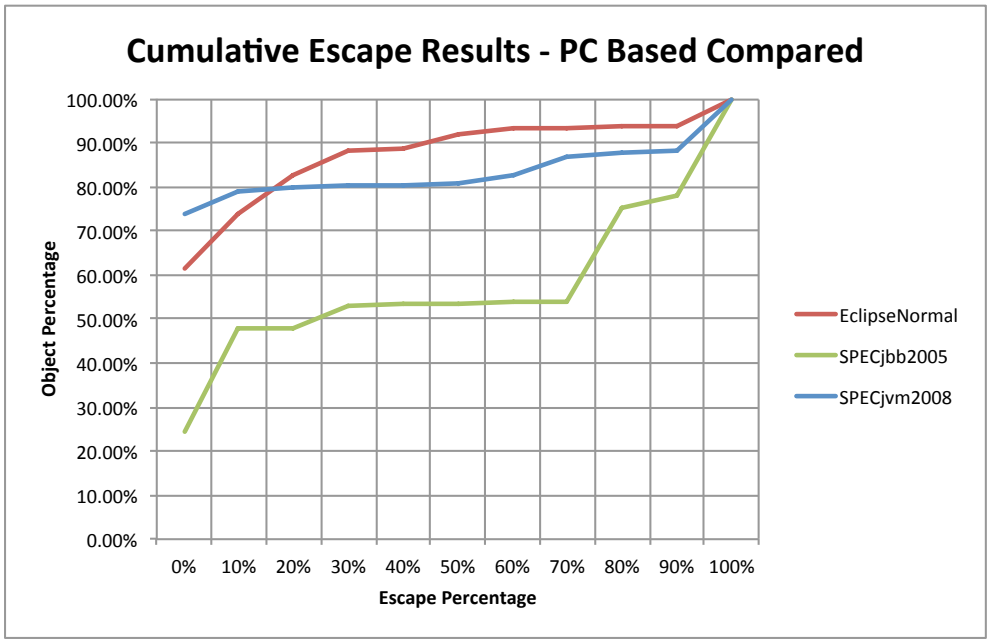


Figure 7.9: Cumulative Escape Results - PC Based Compared

The program counters falling into the first category can always be allocated on TLR with a small penalty for the objects that need to be copied back to the global heap. The escape percentage in the second category varies from 11% to 90%, so that the escaping of objects with PCs falling into this category is unpredictable. The program counters falling into the third category always need to be allocated on the global heap, because they almost always escape and would produce a large overhead for copying them back to the heap soon after their allocation in a TLR.

In summary, the escape analysis results show that the PC based approach can reliably identify program counters with a high escape and non-escape percentage. This data can be used to consider, if an object should be allocated in TLR or on the global heap. The results are very positive and led to the decision to implement a prototype for the TLR allocation and garbage collection that is described in the next chapter.

Chapter 8

Prototype

This chapter describes the implementation of a prototype for the Thread Local Region Allocation and Garbage Collection introduced in the Background chapter (in Section 2.5.1). A detailed description of the implementation is given in Section 8.1, followed by an evaluation of the performance results in Section 8.2.

8.1 Realization

The first approach chosen for the realization of the prototype was the solution described in Section 2.5.1. Unfortunately, it was not possible to update the references of the escaping objects in the write barrier without rewriting major parts of the JVM, so that a slightly different approach was taken. This

second approach marks the objects and corresponding regions as escaped in the barrier without copying them to the global heap. The handling of the escaped objects is performed during the TLRGC, where all objects in the TLRs are traversed anyway. This new approach is described in detail in the next sections.

8.1.1 TLR Allocation

The TLR allocation is enabled by default and can be disabled with the newly introduced VM argument *-XXgc:disableTlrAllocation*. The util class *TlrgcUtil* provides a method named *isTLRAllocationEnabled()* to easily determine, if objects shall be allocated in TLR. The interface *ObjectAllocationModel* defines the abstract method *allocateObject()* that is called for every object allocation. This call is either forwarded to the new *TLRAllocation* class or to the standard *TLHAllocation* depending on the value of the *tlrAllocationEnabled* flag. The new *TLRAllocation* class acts as an intermediate layer that falls back to normal *TLHAllocation* as soon as an object could not be successfully allocated.

Figure 8.1 shows the new processing of *allocateObject()* in a sequence diagram. It is known that some objects always escape, so it does not make sense to allocate them on TLR. These objects can be flagged with *TLRGC_ALLOCATE_OBJECT_NON_TLR* in their *allocationDescription* so that

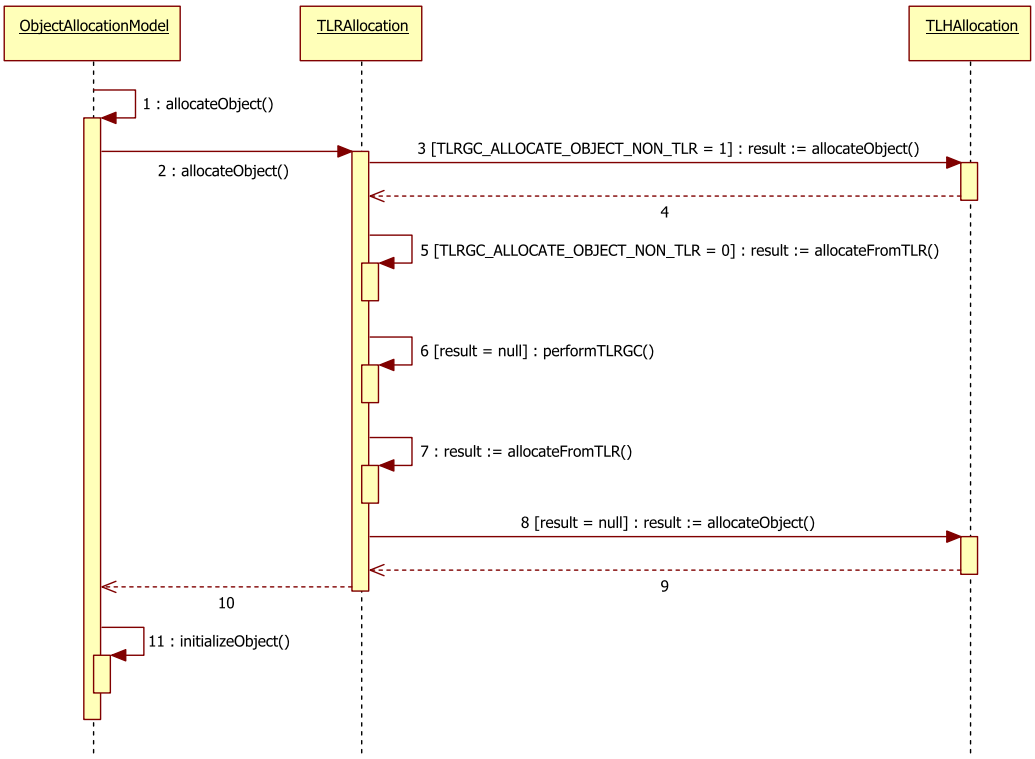


Figure 8.1: Sequence Diagram - TLR Allocation

the TLR allocation is bypassed and the object is directly allocated on TLH. The VM tries to allocate all other objects on its TLRs and triggers a TLRGC, if an allocation failure occurs. The allocation will finally fall back to normal TLH allocation, if even the TLRGC could not free any memory.

8.1.2 TLR Full Handling

The method *allocateFromTLR()* in the *TLRAllocation* class checks if the requested object fits into the currently active TLR and if not, calls the *refresh()* method. This method gets a new free region and assigns it as the new active TLR to the thread. Each thread holds a list of its *tlrRegions* that is limited by the *MAX_TLR_REGIONS_PER_THREAD* parameter. A global GC is either triggered when the refresh method cannot receive any free region to use as TLR or when the limit of *tlrRegions* is reached and the TLRGC cannot free any memory.

8.1.3 Barrier Escape Handling

Figure 8.2 shows the barrier escape handling. Each region holds a variable called *isEscapedRegion* that indicates, if the region contains any escaped objects. Instead of directly copying the escaped objects at the write barrier as originally planned, the barrier now recursively marks the designated objects and the entire TLR as escaped. Further handling of escaped objects is performed by the TLRGC and the Thread Death Handling procedure.

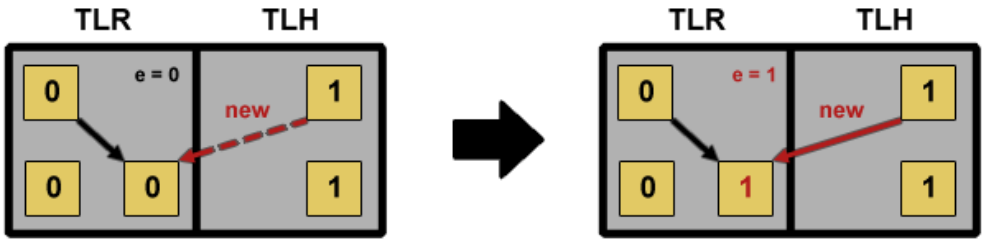


Figure 8.2: Prototype - Barrier Escape Handling

8.1.4 Thread Death Handling

A TLR can either contain some escaped objects *(isEscapedRegion = 1)* or only non-escaped thread-local objects *(isEscapedRegion = 0)*. These two types of regions are handled differently when a thread dies:

- **Escaped Region:** Objects in this region might still be referenced from live TLH objects, so that the whole region needs to be added to the *flushedRegions* list, which contains all TLH regions considered in a global GC.
- **Non-Escaped Region:** All objects in this region are only thread-local, so that the whole region can be added to the *freeRegions* list for reuse. This implies the reclamation of all objects in this region without performing a GC.

Figure 8.3 illustrates the advantages gained by the thread death handling.

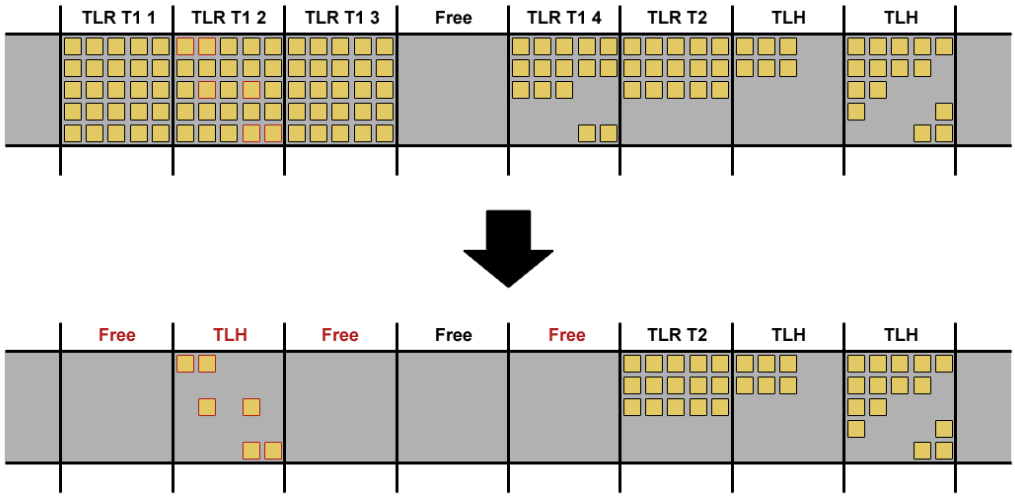


Figure 8.3: Prototype - Thread Death Handling

8.1.5 Thread Local Region Garbage Collection

The Thread Local Region Garbage Collection (TLRGC) is triggered as soon as the *MAX_TLR_REGIONS_PER_THREAD* parameter is reached for the current thread and its active TLR region has not enough space left for the object size requested in *allocateObject()*. In the worst case (the *tlrRegions* of the current thread are completely filled with live objects), the TLRGC needs at least as many free regions as present in the *tlrRegions* list to execute. If this is not the case, the allocation immediately falls back to TLH allocation that first fills up the eden regions and then triggers a global GC as worst case.

The TLRGC is only performed when the precondition of enough free regions available is met and performs the following steps:

1. **Copy all non-escaped live objects:** Starting at the root of the thread, which means the *Thread Stack* and the *Thread fields*, the TLR-GC traverses all live objects and allocates them in new regions that are stored in a *newRegions* list. Additionally a *HashTable* named *updatedReferencesHashTable* is created with the *oldObject* as key and the *newObject* as value to update all references to live objects later on. All other non-escaped objects in the *oldRegions* are considered to be dead.

2. **Update all references of live objects:** The non-escaped live objects can reference each other and already escaped objects. Due to the fact that the address of an escaped object stays untouched with the new approach, only the references of non-escaped live objects need to be updated according to the *updatedReferencesHashTable*.

3. **Recycle the old regions:** After updating the references of the live objects in *newRegions*, the *oldRegions* list can only contain two types of regions:

    - **Non-Escaped Region:** This region only contains dead non-escaped objects, because all references to objects in this region that were previously alive are already updated to their new location in *newRegions*.
    - **Escaped Region:** This region can contain dead non-escaped objects and escaped objects that may still be referenced by TLH objects.

The TLRGC cannot detect, if an escaped object is dead or alive without traversing the whole heap. To avoid such a full heap traversal, the escaped regions are moved to the *flushedRegions* list, so that they are included in the next stop-the-world GC. The non-escaped regions instead can be moved to the *freeRegions* list without any further action. After recycling all regions in the *oldRegions* list, the thread-local environment variable *tlrRegions* will finally be updated to the pointer to *newRegions*.

Figure 8.4 displays the process of the *performTLRGC()* method in a sequence diagram.

8.2 Performance Results

After the prototype was implemented, different tests were performed to measure the impact and improvements of the Escape Marking (EM) and TLR allocation and garbage collection code. The *SPECjbb2005* benchmark is used for all comparisons on the lab test machine equipped with an Intel Core i7-2600 CPU @ 3.40 Ghz and 8 GB of RAM. Due to the fact that the TLRGC is aimed for multi-threaded environments, the *SPECjbb2005* is the best benchmark of the available choices (*EclipseNormal*, *SPECjbb2005*, and *SPECjvm2008*) to measure the performance of the prototype implementation, because it scales up to *16 threads* (twice the logical CPU cores) in the used default configuration [34].

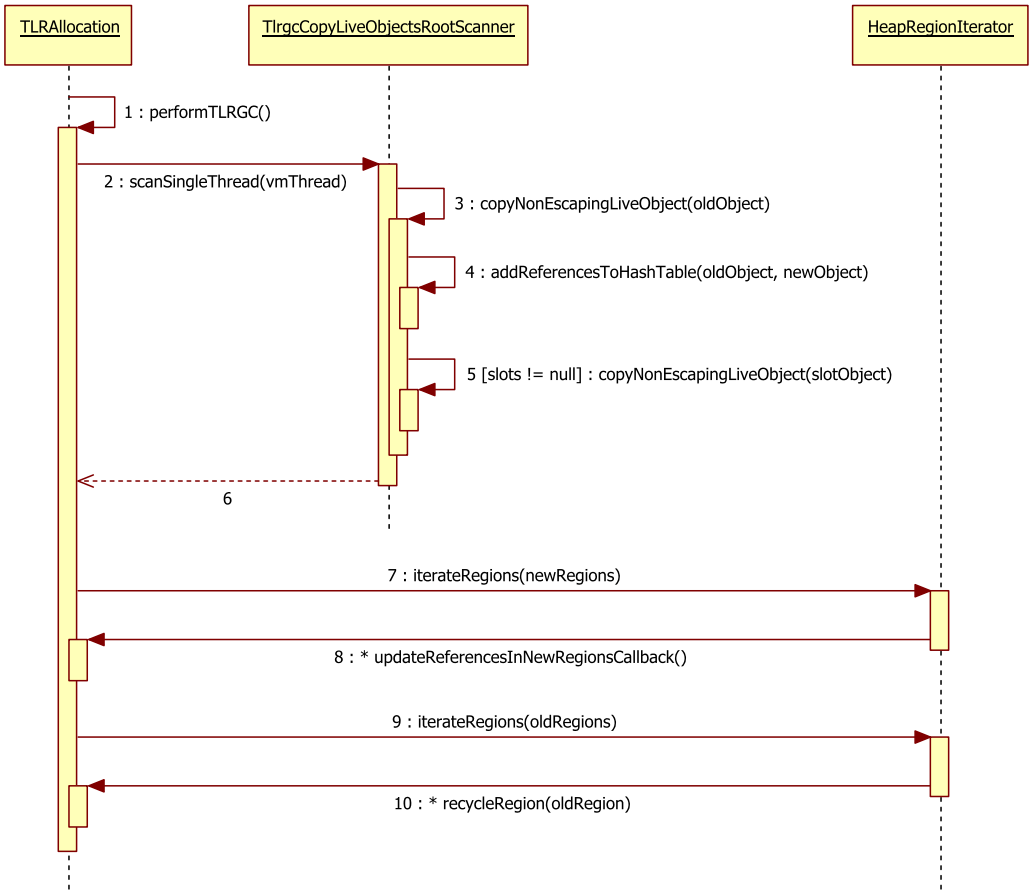


Figure 8.4: Sequence Diagram - TLRGC

The following test scenarios are performed to gather data for the performance evaluation:

1. **Original JVM:** The unmodified JVM without any additional code.
2. **JVM with EM:** The modified JVM with a disabled TLR allocation and garbage collection.
3. **JVM with EM and TLRGC:** The modified JVM with an enabled TLR allocation and garbage collection.

The TLR allocation and garbage collection code is based on the *Balanced GC* policy. All three test cases use this GC policy to produce comparable results. The tests are executed with the following command line that sets the heap size to 1 GB, disables the JIT and enables the *Balanced GC*:

```
D:\dev\jvm\jre\bin\java -Xjvm:cas -Xms1G -Xmx1G -Xint -Xgcpolicy:enable
Unsupported -Xgcpolicy:balanced -cp jbb.jar;check.jar spec.jbb.JBBmain
```

The following sections discuss the prototype performance results based on different metrics. The TLRGC test cases were executed with 1, 2, 5 and 10 regions as *MAX_TLR_REGIONS_PER_THREAD* parameter. The JVM divides the available heap into 2047 regions with a size of 512 KB each.

8.2.1 Runtime / Throughput

Figure 8.5 shows the overall runtime of *SPECjbb2005* with the different test cases. Due to the fact that the prototype is a proof-of-concept and no performance optimizations have been attempted, it is not surprising that the overall runtime increases with the additional code. The original VM was not compiled to always call the write barrier, which explains the shorter runtime compared to all other test cases. The overhead added by the TLRGC in addition to the Escape Marking does not impact the runtimes for 1 and 2 regions. With 5 and 10 regions used as TLR the event of running out of free regions and triggering a stop-the-world GC occurs more often, so that the overall runtime increases significantly.

The throughput comparison of *SPECjbb2005* is displayed in Figure 8.6. The original VM was excluded from the diagram, because it starts with 2,300 bops (business operations per second) at one warehouse and scales up to around 12,000 bops with 8 warehouses and above. The Escape Marking decrease from 2,300 to 895 bops with one warehouse results from the already mentioned activated write barrier check that is always performed. As already seen in the runtime comparison, the current TLRGC implementation additionally decreases the throughput. An interesting fact is that the different region settings of the TLRGC almost result in an identical throughput.

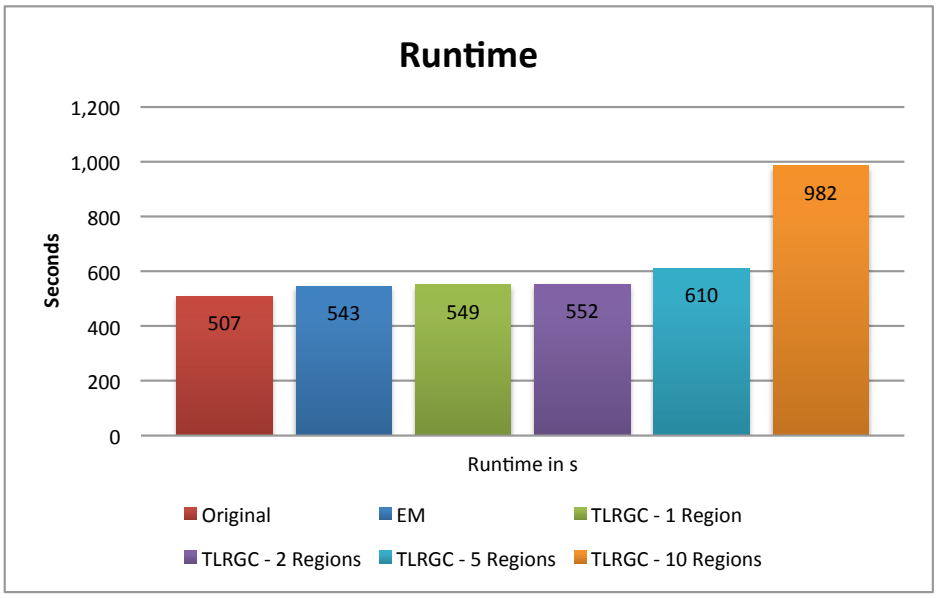


Figure 8.5: Prototype Results - Runtime

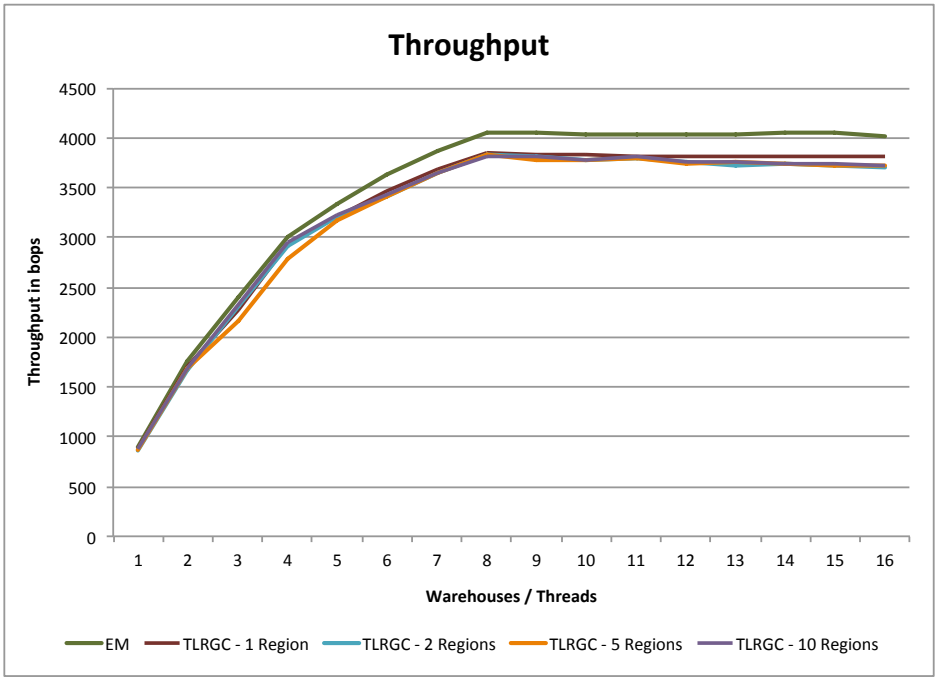


Figure 8.6: Prototype Results - Throughput

8.2.2 Stop-the-world GCs

The main goal of the prototype was to show that the TLRGC approach can reduce the amount and time spent in stop-the-world (STW) GCs. Figures 8.7 and 8.8 show the number and runtime of the STW GCs for all test cases. The results of the original VM are included in these diagrams, but will be excluded from the further discussion, because the throughput of the original VM is at least twice as high as all other test cases. A higher throughput automatically leads to more object allocations and GCs, which makes a reasonable comparison impossible.

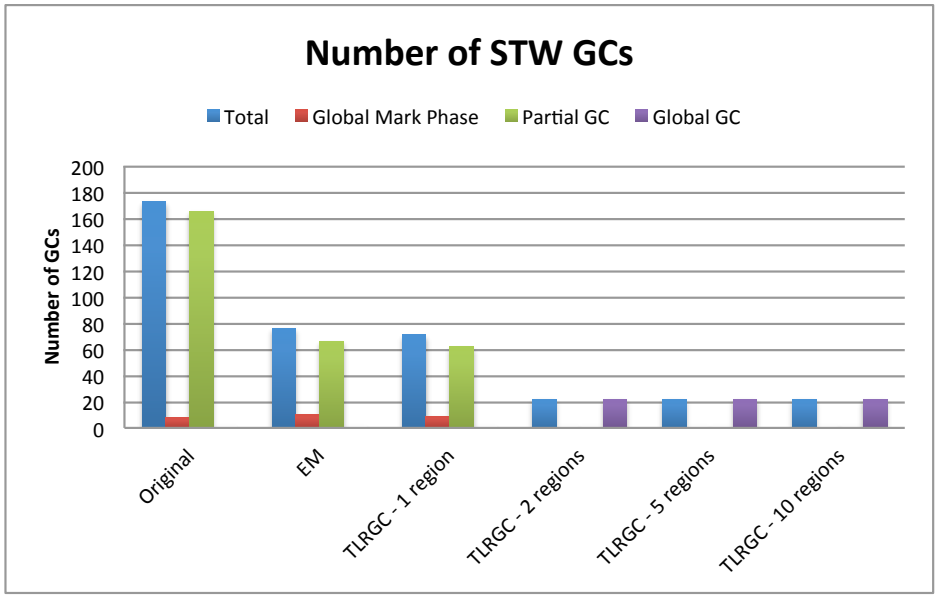


Figure 8.7: Prototype Results - Number of STW GCs

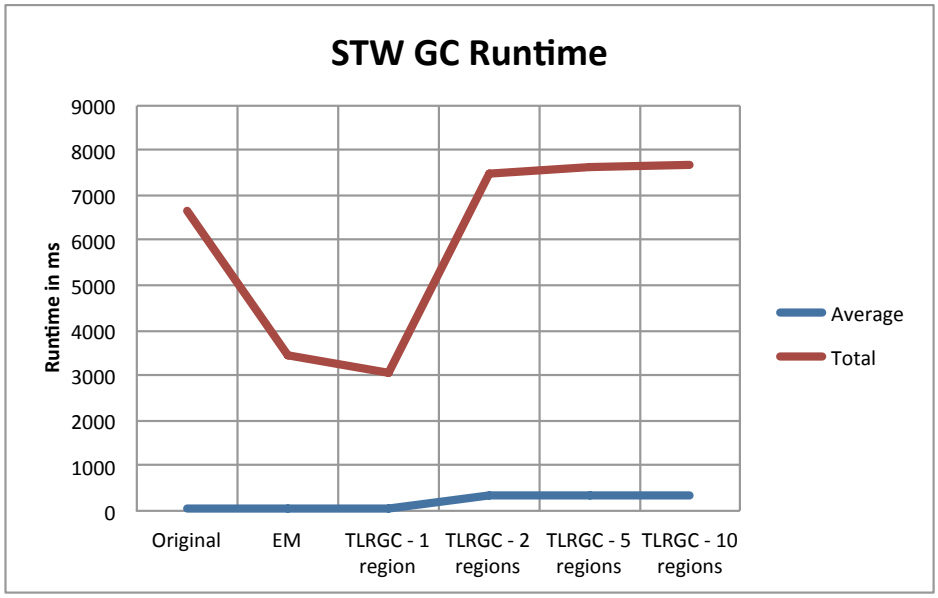


Figure 8.8: Prototype Results - STW GC Runtime

The results show that the TLRGC with 1 region can reduce the number of STW GCs from 76 to 71 and the overall runtime of the STW GCs from 3,450 ms to 3,060 ms. All the STW GCs happening at the EM and 1 region test cases are Partial Garbage Collections. The runtime of these PGCs could be reduced from 45 ms to 43 ms with the 1 region TLRGC.

With more regions used as TLRs before a TLRGC, the problem of the unavailability of free regions on the heap occurs. The current prototype handles this situation with moving all TLRs to the *flushedRegions* list and performing a global GC over the whole heap. This has the effect that only global and no partial garbage collections are performed for the 2, 5 and 10 regions TLRGC test cases. The overall runtime of the GC is therefore increased from about

3,000 ms to 7,500 ms, which slows down the whole application. For the further development of the prototype, a new handling for the unavailability of free regions needs to be implemented. It might be possible to remember all *escapedRegions* that were moved to TLH in a collection set and run a PGC with that collection set first before falling back to the global GC. Additionally, the TLRs should be excluded from the STW GCs in a future version so that they do not need to be moved to the *flushedRegions* list anymore.

8.2.3 TLRGC Objects

The tests show that the TLRs on average contain 37% escaped objects and 63% non-escaped objects as depicted in Figure 8.9. A positive result is that only a small number of these non-escaped objects are alive. This means that the TLRGC can on average free 60% of the objects in the TLRs, because they are already dead.

The hope for gaining a significant improvement from the TLRGC was based on the assumption that a high percentage of the non-escaping objects allocated by a thread are short-living. The results show that 97-100% of the non-escaped objects and about 60% of all objects in the TLRs are already dead when a TLRGC is triggered.

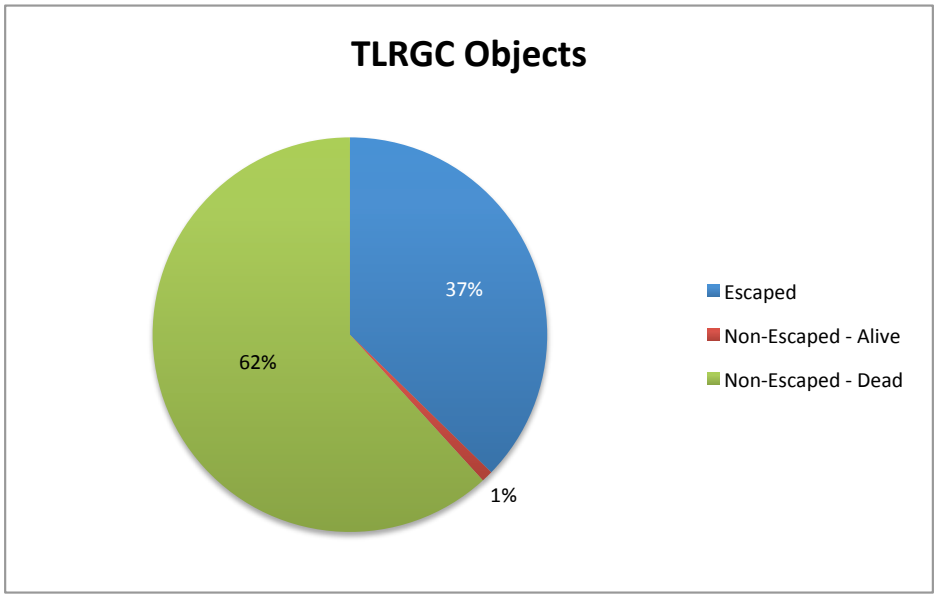


Figure 8.9: Prototype Results - TLRGC Objects

8.2.4 TLRGC Region Usage

The TLRGC region usage is displayed in Figure 8.10. The 1 region TLRGC always has to allocate one new region, because the old region contains at least one live object. This leads to the result that the number of *oldRegions* and *newRegions* used as TLRs is identical for the 1 region TLRGC. An interesting fact is that the 2, 5 and 10 region TLRGCs almost use the same amount of about 29,000 regions in total. It is natural that the 2 region TLRGC has to allocate more *newRegions* than the 5 or 10 region TLRGCs to reach this total amount.

The average region usage depicted in Figure 8.11 shows that all four approaches only require one new region at each TLRGC. Only the 10 region TLRGC consumes on average 1.1 new regions. This can be explained by the fact that 10 TLRs can store more total objects and therefore contain more live objects than 1, 2 or 5 regions. The average region usage corresponds with the results of the previous section that many non-escaping objects in the TLRs are dead and can be reclaimed by the TLRGC.

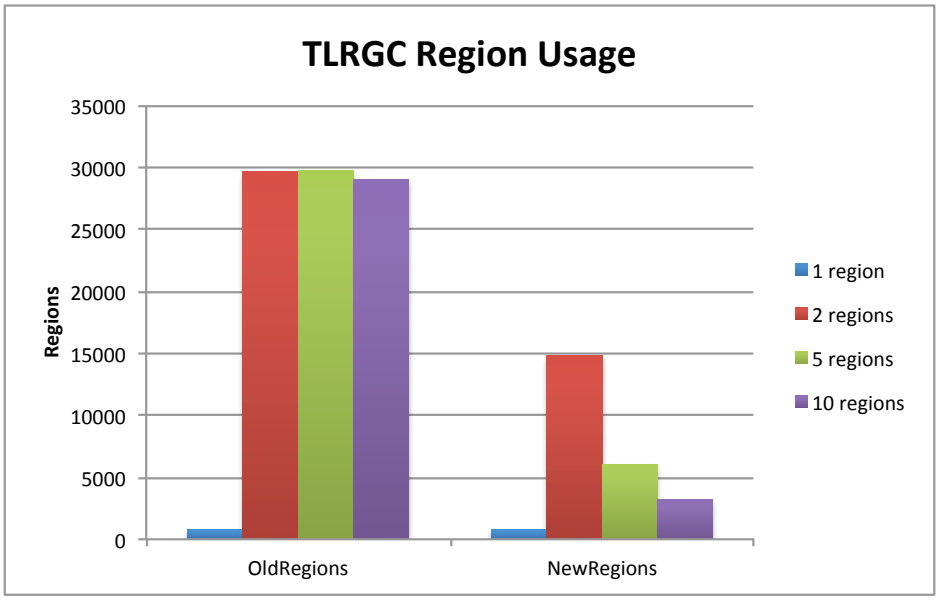


Figure 8.10: Prototype Results - TLRGC Region Usage

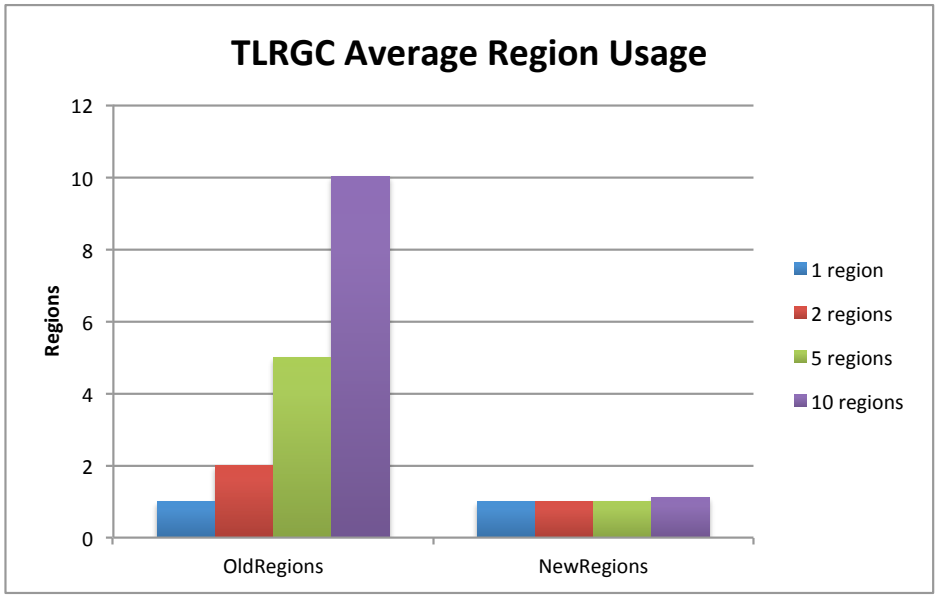


Figure 8.11: Prototype Results - TLRGC Average Region Usage

Chapter 9

Conclusions

The goal of this thesis was to perform a Runtime Escape Analysis for the Java Virtual Machine to reduce the time and amount spent in stop-the-world garbage collections. The realization was split into the following three parts:

1. Instrumentation of the JVM based on the program counter.
2. Measurement and analysis of the escaping of objects with different benchmarks.
3. Implementation of a proof-of-concept prototype.

All three parts could successfully be realized and are evaluated in this chapter.

The instrumentation of the JVM written in the R&D project was evaluated and redesigned to suit the need for a further insight into the escaping of objects. The new instrumentation adds the ability to count escaped objects

based on the program counter. The PC based approach led to a better differentiation between the escaped and non-escaped objects than the class based approach. Depending on the performed benchmark, the results show that the PC based approach can identify 50-80% of the total objects as non-escaping. Additionally, many program counters with 0% escaping objects could be identified. The objects allocated at these PCs can be allocated thread-locally without any need to copy them to TLH later on. In summary, the PC based approach gives a better and usable insight into the escaping of objects and is a suitable approach for further investigations in this area.

The second part of this thesis is the implementation of a prototype for the TLR allocation and garbage collection. The prototype was designed to work without using the escape analysis results or any kind of dynamic escape counting during runtime. It allocates all objects on TLR at their creation, except for special objects like *java/lang/Class* objects or *JNI global references* that always escape. The first approach of copying escaping objects at the write barrier could not be implemented without major VM changes and was therefore dropped.

A new approach to avoid copying the escaping objects at the barrier was developed. Instead of only marking the objects as escaped at the barrier, the whole region is marked as escaped. These escaped regions can contain objects that are still referenced from TLH objects, so that they need to be handled

differently in the TLRGC. Regions marked as non-escaped can be moved to the *freeRegions* list after the TLRGC similar to the idea of the first approach.

The realized prototype implementation for the TLR allocation and garbage collection is focussed on the proof-of-concept and not on performance optimization. This led to the result that the overall runtime of *SPECjbb2005* is increased and the corresponding throughput decreased when using TLRGC. Nevertheless, the TLRGC could prove the concept and fulfil the goal of reducing the number and runtime of the stop-the-world GCs in certain scenarios.

This thesis is only the initiation of the TLR allocation and garbage collection sub-project in the GC research project at the *IBM Centre of Advanced Studies Atlantic* and will be continued by successive researchers. The TLRGC should gain significant speed-ups once the missing parts are implemented and the workarounds removed. The next section discusses how this project can be improved.

9.1 Future work

The realized prototype implementation is not optimized for performance and does not perfectly handle all occurring situations such as the unavailability of free regions on the heap. This leads to poor performance compared to the original JVM. This section introduces the future work and ideas for the TLR allocation and garbage collection project.

The following points should produce significant performance improvements for the TLRGC once they are implemented:

- **Improved swapping:** The TLR allocation currently falls back to the TLH allocation as soon as the TLRGC cannot free the requested object size, which means that all TLRs are filled with live objects. A better solution might be to mark these long-living non-escaped objects as escaped, move them to TLH and start from scratch with empty TLRs or as an alternative just dynamically grow the limit of usable TLRs per thread.

- **Improved handling for the unavailability of free regions:** The TLRGC currently triggers a global GC, if it cannot acquire a free region from the heap. To avoid these global GCs, it might be better to remember all *escapedRegions* moved to TLH in a collection set and trigger a PGC with them before falling back to a global GC as worst case scenario.

- **Exclude TLRs from stop-the-world GCs:** The regions used as TLRs currently do not have their own region type. This means that they are still covered by the STW GCs and need to be added to the *flushedRegions* list before a STW GC can be performed. Using a unique TLR region type and excluding the TLRs from STW GCs might result in less unnecessary escape marking and copying operations of normally non-escaped objects.

- **Keep track of the GCs per TLR:** It could be good to keep track of the number of GCs per TLR. If a region keeps showing up at the TLRGC, it contains many long-living objects that will be copied at each TLRGC. Instead of copying them every time, the objects might be marked as escaped and copied out to the TLH as soon as a certain threshold is exceeded to gain additional space for short-living objects.

- **Introduce efficient escape marking:** Currently the recursive escape marking at the write barrier and always calling the barrier consumes too much time. It needs to be investigated, if there is a better solution that can produce the same input values for the TLRGC with less impact on the runtime performance.

- **Introduce dynamic escape prediction:** Currently all objects are directly allocated in a TLR and specially handled when they escape. Combining the escape analysis results and the prototype implementation might result in a significant performance boost. A further develop-

ment might even be an escape prediction during the runtime of the JVM to dynamically create and update the decision for TLR allocation of a certain program counter.

- **Completely avoid the write barrier:** Investigate the possibility to completely avoid the escape marking at the write barrier. Currently the TLRGC depends on the correct escape marking when an allocation request comes in. Future solutions might remove always calling the write barrier and instead implement a fast detection of the escaped objects at the start of a TLRGC.

When some of these ideas are implemented and show performance improvements, the TLR allocation and garbage collection should also be tested on machines with 128 cores or more, to see if it efficiently scales.

In summary, the prototype implementation created for this thesis has shown that the idea of the TLR allocation and garbage collection is feasible. It also laid a good base for successive researchers to continue the sub-project at the *IBM Centre of Advanced Studies Atlantic*.

Bibliography

[1] TIOBE Software BV, "TIOBE Programming Community Index." `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, February 2013. Accessed: 19/02/2013.

[2] J. Engel, *Programming for the Java Virtual Machine*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1999.

[3] P. R. Wilson, "Uniprocessor garbage collection techniques," in *Proceedings of the International Workshop on Memory Management*, IWMM '92, (London, UK, UK), pp. 1–42, Springer-Verlag, 1992.

[4] R. Sciampacone, P. Burka, and A. Micic, "Garbage collection in WebSphere Application Server V8, Part 2: Balanced garbage collection as a new option." `http://www.ibm.com/developerworks/websphere/techjournal/1108_sciampacone/1108_sciampacone.html`, 2011.

[5] X. Xu and J. Shen, "Research on Stack-allocation based JVM Garbage Collection," in *Advanced Computer Theory and Engineering (ICACTE),*

2010 3rd International Conference on, vol. 2, pp. V2–346 –V2–349, August 2010.

[6] Oracle, "The Java Virtual Machine Specification, Java SE 7 Edition." `http://docs.oracle.com/javase/specs/`, 2011.

[7] Oracle, "Java.com." `http://www.java.com`, 2012. Accessed: 10/12/2012.

[8] Oracle, "Java Technology." `http://www.oracle.com/us/technologies/java/overview/index.html`, 2012. Accessed: 10/12/2012.

[9] IBM, "Java Technology." `http://www.ibm.com/developerworks/java/`, 2012. Accessed: 10/12/2012.

[10] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I," *Commun. ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[11] G. E. Collins, "A method for overlapping and erasure of lists," *Commun. ACM*, vol. 3, pp. 655–657, Dec. 1960.

[12] D. G. Bobrow and D. W. Clark, "Compact encodings of list structure," *ACM Trans. Program. Lang. Syst.*, vol. 1, pp. 266–286, Oct. 1979.

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

[14] M. L. Minsky, "A LISP Garbage Collector Algorithm Using Serial Secondary Storage," in *Memorandum MAC-M ; no. 129.; AI Memo 58*, Massachusetts Institute of Technology, Project MAC, Dec. 1963.

[15] R. R. Fenichel and J. C. Yochelson, "A lisp garbage-collector for virtual-memory computer systems," *Commun. ACM*, vol. 12, pp. 611–612, Nov. 1969.

[16] C. J. Cheney, "A nonrecursive list compacting algorithm," *Commun. ACM*, vol. 13, pp. 677–678, Nov. 1970.

[17] H. Lieberman and C. Hewitt, "A real-time garbage collector based on the lifetimes of objects," *Commun. ACM*, vol. 26, pp. 419–429, June 1983.

[18] IBM, "IBM Homepage." `http://www.ibm.com/`, 2012. Accessed: 28/12/2012.

[19] S. Komuro and K. Abe, "A proposal of stack-based garbage collection and its evaluation in scripting language lua," in *Proceedings of the International Workshop on Modern Science and Technology*, IWMST '10, pp. 94–99, 2010.

[20] R. Jones and A. King, "A fast analysis for thread-local garbage collection with dynamic class loading," in *Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 129–138, 2005.

[21] B. Steensgaard, "Thread-specific heaps for multi-threaded programs," in *Proceedings of the 2nd international symposium on Memory management*, ISMM '00, (New York, NY, USA), pp. 18–24, ACM, 2000.

[22] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald, "Thread-local heaps for Java," in *Proceedings of the 3rd international symposium on Memory management*, ISMM '02, (New York, NY, USA), pp. 76–87, ACM, 2002.

[23] S. Marlow and S. Peyton Jones, "Multicore garbage collection with local heaps," in *Proceedings of the international symposium on Memory management*, ISMM '11, (New York, NY, USA), pp. 21–32, ACM, 2011.

[24] S. M. Blackburn and K. S. McKinley, "In or out?: putting write barriers in their place," *SIGPLAN Not.*, vol. 38, no. 2 supplement, pp. 175–184, 2002.

[25] J. Dolby, "Automatic inline allocation of objects," *SIGPLAN Not.*, vol. 32, pp. 7–17, May 1997.

[26] Standard Performance Evaluation Corporation, "SPECjbb2005." `http://www.spec.org/jbb2005/`, August 2006. Accessed: 26/06/2012.

[27] Standard Performance Evaluation Corporation, "SPECjvm2008." http://www.spec.org/jvm2008/, May 2008. Accessed: 26/06/2012.

[28] The Eclipse Foundation, "Eclipse IDE for C/C++ Developers (includes Incubating components)." http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers-includes-incubating-components/indigosr2, June 2012. Accessed: 26/06/2012.

[29] Oracle, "The Java EE 6 Tutorial - Distributed Multitiered Applications." http://docs.oracle.com/javaee/6/tutorial/doc/bnaay.html, 2013. Accessed: 10/02/2013.

[30] Standard Performance Evaluation Corporation, "SPECjvm2008 Benchmarks." http://www.spec.org/jvm2008/docs/benchmarks/index.html, May 2008. Accessed: 10/02/2013.

[31] Apache Software Foundation, "Apache Derby." http://db.apache.org/derby/, 2013. Accessed: 10/02/2013.

[32] National Institute of Standards and Technology (NIST), "SciMark 2.0." http://math.nist.gov/scimark2/, 1999. Accessed: 10/02/2013.

[33] C. Kulla, "Sunflow - Global Illumination Rendering System." http://sunflow.sourceforge.net, 2007. Accessed: 10/02/2013.

[34] Standard Performance Evaluation Corporation, "SPECjbb2005 User's Guide." http://www.spec.org/jbb2005/docs/UserGuide.html, April 2006. Accessed: 28/02/2013.

Appendix A

Example of a PC based csv file

Attached is an excerpt of a PC based csv file created by an *EclipseNormal* run. Each line contains the following values: PC, Line of Code, Class, EscapedCount, NonEscapedCount, TotalCount, EscapePercentage and Non-EscapePercentage.

```
1148989132,StringBuilder.java(812),String,6525,2292033,2298558,0,1
1148782080,StringCoding.java(490),char[],8462,2111750,2120212,0,1
1148984696,StringBuilder.java(78),char[],24536,2093206,2117742,0.01,0.99
1234857375,WrathReadStream.java(52),StringBuilder,0,1936540,1936540,0,1
1234857386,WrathReadStream.java(52),String,0,1936540,1936540,0,1
1136543854,String.java(1451),String,63,1247308,1247371,0,1
1148986200,StringBuilder.java(342),char[],100377,1139458,1239835,0.08,0.92
1136535792,String.java(305),char[],222554,811786,1034340,0.22,0.78
1136540883,String.java(1108),String,127512,552658,680170,0.19,0.81
1150014120,Pattern.java(1088),Matcher,0,668311,668311,0,1
1150262799,Matcher.java(224),int[],0,668311,668311,0,1
1150262809,Matcher.java(225),int[],0,668311,668311,0,1
```

```
1150017580,Pattern.java(1636),int[],0,666770,666770,0,1
1150017664,Pattern.java(1655),int[],0,666770,666770,0,1
1150017672,Pattern.java(1656),Pattern$GroupHead[],0,666770,666770,0,1
1150013904,Pattern.java(1022),Pattern,58,666703,666761,0,1
1151415937,Pattern.java(3397),Pattern$TreeInfo,0,666583,666583,0,1
1150017894,Pattern.java(1685),Pattern$Start,22,666369,666391,0,1
1150029497,Pattern.java(3305),Pattern$Single,30,666326,666356,0,1
1148984627,StringBuilder.java(66),char[],2183,650417,652600,0,1
1148666512,HashMap.java(646),HashMap$Entry,258218,254096,512314,0.5,0.5
1149010907,StringBuffer.java(62),char[],100530,351125,451655,0.22,0.78
1149499517,ArrayList.java(132),Object[],196926,240403,437329,0.45,0.55
1149015364,StringBuffer.java(802),String,107446,325674,433120,0.25,0.75
1148989110,StringBuilder.java(809),String,849,425896,426745,0,1
1150280317,DataInputStream.java(661),String,173703,153487,327190,0.53,0.47
1149012480,StringBuffer.java(338),char[],12780,296010,308790,0.04,0.96
1150264743,Matcher.java(756),StringBuilder,0,222606,222606,0,1
1190632690,MessageFormat.java(450),StringBuilder,0,218426,218426,0,1
1190636546,MessageFormat.java(1409),String[],0,218426,218426,0,1
1162899135,Path.java(470),String[],11431,194310,205741,0.06,0.94
1162897778,Path.java(248),Path,85759,99228,184987,0.46,0.54
1234857286,WrathReadStream.java(45),StringBuilder,0,166706,166706,0,1
1234857297,WrathReadStream.java(45),String,0,166706,166706,0,1
1163122856,TreeMap.java(4816),TreeMap$Node,157881,3918,161799,0.98,0.02
1163137102,TreeMap.java(77),Object[],157881,3918,161799,0.98,0.02
1163137114,TreeMap.java(78),Object[],157881,3918,161799,0.98,0.02
1163144438,Format.java(157),StringBuffer,0,109328,109328,0,1
1163144445,Format.java(157),FieldPosition,0,109328,109328,0,1
1190632265,MessageFormat.java(1188),Format[],0,109270,109270,0,1
1190632274,MessageFormat.java(1195),int[],0,109270,109270,0,1
1190632282,MessageFormat.java(1203),int[],0,109270,109270,0,1
1190632553,MessageFormat.java(426),StringBuilder[],0,109270,109270,0,1
1190632559,MessageFormat.java(429),StringBuilder,0,109270,109270,0,1
1190634304,MessageFormat.java(835),MessageFormat,0,109270,109270,0,1
(...)
```

Appendix B

EscapeMarkingVerificationTest

```
1  public void testSingleThreaded() throws InterruptedException {
2      System.gc(); // Collect 1
3
4      MyObject myObject1 = new MyObject();
5      MyObject myObject2 = new MyObject();
6      MyObject myObject3 = new MyObject();
7      MyObject myObject4 = new MyObject();
8      MyObject myObject5 = new MyObject();
9      MyObject myObject6 = new MyObject();
10     MyObject myObject7 = new MyObject();
11     MyObject myObject8 = new MyObject();
12     MyObject myObject9 = new MyObject();
13     MyObject myObject10 = new MyObject();
14
15     System.gc(); // Collect 2
16
17     List<MyObject> list = new LinkedList<MyObject>();
18     for (int i = 1; i <= 10; i++) {
19         list.add(new MyObject());
20     }
21     assertTrue(list.size() == 10);
22
23     System.gc(); // Collect 3
24 }
```

Figure B.1: EscapeMarkingVerificationTest - testSingleThreaded()

```
1  public void testMultiThreadedSingleObject() throws InterruptedException {
2      System.gc(); // Collect 4
3
4      // create second thread
5      MultiThread multiThread = new MultiThread();
6      multiThread.start();
7
8      System.gc(); // Collect 5
9
10     MyObject nonEscaping = new MyObject();
11     MyObject directEscaping = new MyObject();
12     MyObject transitionEscaping = new MyObject();
13
14     // nonEscaping: escaped = 0, seen = 0, transition = 0
15     // directEscaping: escaped = 0, seen = 0, transition = 0
16     // transitionEscaping: escaped = 0, seen = 0, transition = 0
17
18     assertNull(multiThread.getObjectSlot1());
19     multiThread.setObjectSlot1(directEscaping);
20     assertNotNull(multiThread.getObjectSlot1());
21
22     // nonEscaping: escaped = 0, seen = 0, transition = 0
23     // directEscaping: escaped = 1, seen = 0, transition = 0
24     // transitionEscaping: escaped = 0, seen = 0, transition = 0
25
26     System.gc(); // Collect 6
27
28     // nonEscaping: escaped = 0, seen = 1, transition = 0
29     // directEscaping: escaped = 1, seen = 1, transition = 0
30     // transitionEscaping: escaped = 0, seen = 1, transition = 0
31
32     assertNull(multiThread.getObjectSlot2());
33     multiThread.setObjectSlot2(transitionEscaping);
34     assertNotNull(multiThread.getObjectSlot2());
35
36     // nonEscaping: escaped = 0, seen = 1, transition = 0
37     // directEscaping: escaped = 1, seen = 1, transition = 0
38     // transitionEscaping: escaped = 1, seen = 1, transition = 1
39
40     System.gc(); // Collect 7
41
42     // nonEscaping: escaped = 0, seen = 1, transition = 0
43     // directEscaping: escaped = 1, seen = 1, transition = 0
44     // transitionEscaping: escaped = 1, seen = 1, transition = 0
45
46     System.gc(); // Collect 8
47 }
```

Figure B.2: EscapeMarkingVerificationTest - testMultiThreadedSingleObject()

```
1  public void testMultiThreadedInlineAllocation() throws InterruptedException {
2      System.gc(); // Collect 9
3
4      // create second thread
5      MultiThread multiThread = new MultiThread();
6      multiThread.start();
7
8      String inlineAllocationString = new String("Inline alloc");
9
10     System.gc(); // Collect 10
11
12     MyObject nonEscaping = new MyObject(new String("Inline alloc"));
13     MyObject directEscaping = new MyObject(inlineAllocationString);
14     MyObject transitionEscaping = new MyObject(new String("Inline alloc"));
15
16     // nonEscaping: escaped = 0, seen = 0, transition = 0
17     // directEscaping: escaped = 0, seen = 0, transition = 0
18     // transitionEscaping: escaped = 0, seen = 0, transition = 0
19
20     assertNull(multiThread.getObjectSlot1());
21     multiThread.setObjectSlot1(directEscaping);
22     assertNotNull(multiThread.getObjectSlot1());
23
24     // nonEscaping: escaped = 0, seen = 0, transition = 0
25     // directEscaping: escaped = 1, seen = 0, transition = 0
26     // transitionEscaping: escaped = 0, seen = 0, transition = 0
27
28     System.gc(); // Collect 11
29
30     // nonEscaping: escaped = 0, seen = 1, transition = 0
31     // directEscaping: escaped = 1, seen = 1, transition = 0
32     // transitionEscaping: escaped = 0, seen = 1, transition = 0
33
34     assertNull(multiThread.getObjectSlot2());
35     multiThread.setObjectSlot2(transitionEscaping);
36     assertNotNull(multiThread.getObjectSlot2());
37
38     // nonEscaping: escaped = 0, seen = 1, transition = 0
39     // directEscaping: escaped = 1, seen = 1, transition = 0
40     // transitionEscaping: escaped = 1, seen = 1, transition = 1
41
42     System.gc(); // Collect 12
43
44     // nonEscaping: escaped = 0, seen = 1, transition = 0
45     // directEscaping: escaped = 1, seen = 1, transition = 0
46     // transitionEscaping: escaped = 1, seen = 1, transition = 0
47
48     System.gc(); // Collect 13
49 }
```

Figure B.3: EscapeMarkingVerificationTest - testMultiThreadedInlineAllocation()

```
1  public void testMultiThreadedLinkedList() throws InterruptedException {
2      System.gc(); // Collect 14
3
4      // create second thread
5      MultiThread multiThread = new MultiThread();
6      multiThread.start();
7
8      String inlineAllocationString = new String("Inline alloc");
9
10     System.gc(); // Collect 15
11
12     List<MyObject> nonEscapingList = new LinkedList<MyObject>();
13     List<MyObject> directEscapingList = new LinkedList<MyObject>();
14     List<MyObject> transitionEscapingList = new LinkedList<MyObject>();
15
16     for (int i = 1; i <= 10; i++) {
17         nonEscapingList.add(new MyObject());
18         directEscapingList.add(new MyObject(inlineAllocationString));
19         transitionEscapingList.add(new MyObject(new String("Inline alloc")));
20     }
21
22     // nonEscaping: escaped = 0, seen = 0, transition = 0
23     // directEscaping: escaped = 0, seen = 0, transition = 0
24     // transitionEscaping: escaped = 0, seen = 0, transition = 0
25
26     assertNull(multiThread.getListSlot1());
27     multiThread.setListSlot1(directEscapingList);
28     assertNotNull(multiThread.getListSlot1());
29
30     // nonEscaping: escaped = 0, seen = 0, transition = 0
31     // directEscaping: escaped = 1, seen = 0, transition = 0
32     // transitionEscaping: escaped = 0, seen = 0, transition = 0
33
34     System.gc(); // Collect 16
35
36     // nonEscaping: escaped = 0, seen = 1, transition = 0
37     // directEscaping: escaped = 1, seen = 1, transition = 0
38     // transitionEscaping: escaped = 0, seen = 1, transition = 0
39
40     assertNull(multiThread.getListSlot2());
41     multiThread.setListSlot2(transitionEscapingList);
42     assertNotNull(multiThread.getListSlot2());
43
44     // nonEscaping: escaped = 0, seen = 1, transition = 0
45     // directEscaping: escaped = 1, seen = 1, transition = 0
46     // transitionEscaping: escaped = 1, seen = 1, transition = 1
47
48     System.gc(); // Collect 17
49
50     // nonEscaping: escaped = 0, seen = 1, transition = 0
51     // directEscaping: escaped = 1, seen = 1, transition = 0
52     // transitionEscaping: escaped = 1, seen = 1, transition = 0
53
54     System.gc(); // Collect 18
55 }
```

Figure B.4: EscapeMarkingVerificationTest - testMultiThreadedLinkedList()

Appendix C

PC Based Results in Detail

This appendix contains the full Top 10 data tables of allocated, escaping and non-escaping objects for *EclipseNormal*, *SPECjbb2005* and *SPECjvm2008*.

The class names in the column ***Class*** and the file names in the column ***Line of Code*** are shortened due to space constraints. The file names are displayed without *".java"* extension and the class names without their package details, e.g *String* instead of *java/lang/String*.

Line of Code	Class	esc #[1]	nEsc #[2]	total #[3]	esc[4]	nEsc[5]	obj[6]
String(1129)	String	383,402	2,184,846	2,568,248	15.00%	85.00%	6.66%
StringBuilder(812)	String	6,525	2,292,033	2,298,558	0.00%	100.00%	5.96%
StringCoding(490)	char[]	8,462	2,111,750	2,120,212	0.00%	100.00%	5.50%
StringBuilder(78)	char[]	24,536	2,093,206	2,117,742	1.00%	99.00%	5.49%
String(1451)	String	63	1,247,308	1,247,371	0.00%	100.00%	3.24%
StringBuilder(342)	char[]	100,377	1,139,458	1,239,835	8.00%	92.00%	3.22%
String(305)	char[]	222,554	811,786	1,034,340	22.00%	78.00%	2.68%
String(1108)	String	127,512	552,658	680,170	19.00%	81.00%	1.76%
Pattern(1088)	Matcher	0	668,311	668,311	0.00%	100.00%	1.73%
Matcher(224)	int[]	0	668,311	668,311	0.00%	100.00%	1.73%

[1] Number of Escaped Objects [2] Number of Non-Escaped Objects [3] Total Number of Objects
[4] Percentage of Escaped Objects [5] Percentage of Non-Escaped Objects
[6] Percentage of Total Objects

Table C.1: Top 10 allocated objects - EclipseNormal

Line of Code	Class	esc #[1]	nEsc #[2]	total #[3]	esc[4]	nEsc[5]	obj[6]
Integer(642)	Integer	793,137	23,789,119	24,582,256	3.00%	97.00%	23.25%
String(305)	char[]	15,868,715	6,600,843	22,469,558	71.00%	29.00%	21.25%
TransLogBuf[7]	String	11,944,639	150	11,944,789	100.00%	0.00%	11.30%
BigDec[8](5734)	char[]	0	6,498,015	6,498,015	0.00%	100.00%	6.15%
BigDec[8](5736)	String	0	6,498,014	6,498,014	0.00%	100.00%	6.15%
BigDec[8](1184)	BigDec[8]	754,015	2,743,466	3,497,481	22.00%	78.00%	3.31%
JBButil	char[]	0	3,480,220	3,480,220	0.00%	100.00%	3.29%
Stock	String	3,400,000	0	3,400,000	100.00%	0.00%	3.22%
BigDec[8](2039)	BigDec[8]	2,286,674	456,792	2,743,466	83.00%	17.00%	2.59%
Orderline	String	2,286,674	0	2,286,674	100.00%	0.00%	2.16%

[1] Number of Escaped Objects [2] Number of Non-Escaped Objects
[3] Total Number of Objects [4] Percentage of Escaped Objects [5] Percentage of Non-Escaped Objects
[6] Percentage of Total Objects [7] TransactionLogBuffer [8] BigDecimal

Table C.2: Top 10 allocated objects - SPECjbb2005

Line of Code	Class	esc #[1]	nEsc #[2]	total #[3]	esc[4]	nEsc[5]	obj[6]
Ray	Ray	0	130,702,936	130,702,936	0.00%	100.00%	8.28%
BigDec[7](1184)	BigDec[7]	22,501	75,000,000	75,022,501	0.00%	100.00%	4.75%
InstantGI	Ray	0	65,258,424	65,258,424	0.00%	100.00%	4.13%
OSC[8](317)	OSC$WCK[9]	42	57,869,974	57,870,016	0.00%	100.00%	3.66%
StrBlder[10](66)	char[]	254,444	52,864,280	53,118,724	0.00%	100.00%	3.36%
StrBlder[10](812)	String	358,587	51,050,260	51,408,847	1.00%	99.00%	3.26%
OIS[11](3019)	StrBlder[10]	0	47,250,008	47,250,008	0.00%	100.00%	2.99%
BigDec[7](2385)	BigDec[7]	112,500	37,500,232	37,612,732	0.00%	100.00%	2.38%
Color	Color	0	37,535,778	37,535,778	0.00%	100.00%	2.38%
String(1140)	char[]	290	32,450,174	32,450,464	0.00%	100.00%	2.05%

[1] Number of Escaped Objects [2] Number of Non-Escaped Objects [3] Total Number of Objects
[4] Percentage of Escaped Objects [5] Percentage of Non-Escaped Objects [6] Percentage of Total Objects
[7] BigDecimal [8] ObjectStreamClass [9] ObjectStreamClass$WeakClassKey [10] StringBuilder
[11] ObjectInputStream

Table C.3: Top 10 allocated objects - SPECjvm2008

Line of Code	Class	esc #[1]	nEsc #[2]	total #[3]	esc[4]	nEsc[5]	obj[6]
TreeMap(4816)	TreeMap$Node	157,881	3,918	161,799	98%	2%	0.42%
TreeMap(77)	Object[]	157,881	3,918	161,799	98%	2%	0.42%
TreeMap(78)	Object[]	157,881	3,918	161,799	98%	2%	0.42%
TreeSet(54)	TreeMap	132,089	12	132,101	100%	0%	0.34%
TreeMap(4429)	TreeMap$1	98,622	3,651	102,273	96%	4%	0.27%
Path(834)	String[]	88,152	1,079	89,231	99%	1%	0.23%
Path(836)	Path	88,152	1,079	89,231	99%	1%	0.23%
MarkerReader_3(119)	Integer	66,332	0	66,332	100%	0%	0.17%
DataTreeReader(110)	DataTreeNode	54,437	1	54,438	100%	0%	0.14%
Workspace(2126)	ResourceInfo	54,348	0	54,348	100%	0%	0.14%

[1] Number of Escaped Objects [2] Number of Non-Escaped Objects [3] Total Number of Objects
[4] Percentage of Escaped Objects [5] Percentage of Non-Escaped Objects [6] Percentage of Total Objects

Table C.4: Top 10 escaping objects - EclipseNormal

Line of Code	Class	esc #[1]	nEsc #[2]	total #[3]	esc[4]	nEsc[5]	obj[6]
TransLogBuf[7]	String	11,944,639	150	11,944,789	100%	0%	11.30%
Stock	String	3,400,000	0	3,400,000	100%	0%	3.22%
Orderline	String	2,286,674	0	2,286,674	100%	0%	2.16%
Order	Orderline	2,286,674	0	2,286,674	100%	0%	2.16%
HashMap(646)	HashMap$E[8]	392,895	868	393,763	100%	0%	0.37%
Stock	String	340,000	0	340,000	100%	0%	0.32%
Stock	String[]	340,000	0	340,000	100%	0%	0.32%
Warehouse	Stock	340,000	0	340,000	100%	0%	0.32%
Order	Orderline[]	228,396	0	228,396	100%	0%	0.22%
Order	Date	228,396	0	228,396	100%	0%	0.22%

[1] Number of Escaped Objects [2] Number of Non-Escaped Objects [3] Total Number of Objects
[4] Percentage of Escaped Objects [5] Percentage of Non-Escaped Objects
[6] Percentage of Total Objects [7] TransactionLogBuffer [8] HashMap$Entry

Table C.5: Top 10 escaping objects - SPECjbb2005

Line of Code	Class	esc #[1]	nEsc #[2]	total #[3]	esc[4]	nEsc[5]	obj[6]
SQLChar	String	15,000,307	640	15,000,947	100%	0%	0.95%
HashMap(646)	HashMap$E[7]	11,455,613	439,549	11,895,162	96%	4%	0.75%
TreeMaker(404)	JCTree$JCI[8]	8,756,608	44,769	8,801,377	99%	1%	0.56%
Scope(400)	ScopeISIE[9]	8,136,558	0	8,136,558	100%	0%	0.52%
Env(97)	Env	6,536,826	333,337	6,870,163	95%	5%	0.44%
RecordId	PageKey	6,818,248	25,857	6,844,105	100%	0%	0.43%
BaseConHdl[10]	RecordId	6,818,248	25,857	6,844,105	100%	0%	0.43%
Scope(218)	Scope$Entry	5,465,234	0	5,465,234	100%	0%	0.35%
AttrContext(73)	AttrContext	4,784,027	266,828	5,050,855	95%	5%	0.32%
Items(151)	Items$MI[11]	3,856,605	351,702	4,208,307	92%	8%	0.27%

[1] Number of Escaped Objects [2] Number of Non-Escaped Objects
[3] Total Number of Objects [4] Percentage of Escaped Objects [5] Percentage of Non-Escaped Objects
[6] Percentage of Total Objects [7] HashMap$Entry [8] JCTree$JCIdent
[9] Scope$ImportScope$ImportEntry [10] BaseContainerHandle [11] Items$MemberItem

Table C.6: Top 10 escaping objects - SPECjvm2008

Line of Code	Class	esc #[1]	nEsc #[2]	total #[3]	esc[4]	nEsc[5]	obj[6]
StringBuilder(812)	String	6,525	2,292,033	2,298,558	0%	100%	5.96%
StringCoding(490)	char[]	8,462	2,111,750	2,120,212	0%	100%	5.50%
StringBuilder(78)	char[]	24,536	2,093,206	2,117,742	1%	99%	5.49%
String(1451)	String	63	1,247,308	1,247,371	0%	100%	3.24%
StringBuilder(342)	char[]	100,377	1,139,458	1,239,835	8%	92%	3.22%
Pattern(1088)	Matcher	0	668,311	668,311	0%	100%	1.73%
Matcher(224)	int[]	0	668,311	668,311	0%	100%	1.73%
Matcher(225)	int[]	0	668,311	668,311	0%	100%	1.73%
Pattern(1636)	int[]	0	666,770	666,770	0%	100%	1.73%
Pattern(1655)	int[]	0	666,770	666,770	0%	100%	1.73%

[1] Number of Escaped Objects [2] Number of Non-Escaped Objects
[3] Total Number of Objects [4] Percentage of Escaped Objects [5] Percentage of Non-Escaped Objects [6] Percentage of Total Objects

Table C.7: Top 10 non-escaping objects - EclipseNormal

Line of Code	Class	esc #[1]	nEsc #[2]	total #[3]	esc[4]	nEsc[5]	obj[6]
Integer(642)	Integer	793,137	23,789,119	24,582,256	3%	97%	23.25%
BigDecimal(5734)	char[]	0	6,498,015	6,498,015	0%	100%	6.15%
BigDecimal(5736)	String	0	6,498,014	6,498,014	0%	100%	6.15%
JBButil	char[]	0	3,480,220	3,480,220	0%	100%	3.29%
Stock	String	0	2,286,674	2,286,674	0%	100%	2.16%
BigDecimal(7257)	BigDecimal	0	1,336,981	1,336,981	0%	100%	1.26%
TreeMap(5358)	TreeMap$UVI[7]	0	621,197	621,197	0%	100%	0.59%
BigDecimal(5317)	BigDecimal	791	505,288	506,079	0%	100%	0.48%
Integer(331)	char[]	2	401,683	401,685	0%	100%	0.38%
Integer(333)	String	2	401,683	401,685	0%	100%	0.38%

[1] Number of Escaped Objects [2] Number of Non-Escaped Objects
[3] Total Number of Objects [4] Percentage of Escaped Objects [5] Percentage of Non-Escaped Objects
[6] Percentage of Total Objects [7] TreeMap$UnboundedValueIterator

Table C.8: Top 10 non-escaping objects - SPECjbb2005

Line of Code	Class	esc #[1]	nEsc #[2]	total #[3]	esc[4]	nEsc[5]	obj[6]
Ray	Ray	0	130,702,936	130,702,936	0%	100%	8.28%
BigDecimal(1184)	BigDecimal	22,501	75,000,000	75,022,501	0%	100%	4.75%
InstantGI	Ray	0	65,258,424	65,258,424	0%	100%	4.13%
OSC[7](317)	OSC$WCK[9]	42	57,869,974	57,870,016	0%	100%	3.66%
StringBuilder(66)	char[]	254,444	52,864,280	53,118,724	0%	100%	3.36%
StringBuilder(812)	String	358,587	51,050,260	51,408,847	1%	99%	3.26%
OIS[8](3019)	StringBuilder	0	47,250,008	47,250,008	0%	100%	2.99%
BigDecimal(2385)	BigDecimal	112,500	37,500,232	37,612,732	0%	100%	2.38%
Color	Color	0	37,535,778	37,535,778	0%	100%	2.38%
String(1140)	char[]	290	32,450,174	32,450,464	0%	100%	2.05%

[1] Number of Escaped Objects [2] Number of Non-Escaped Objects [3] Total Number of Objects
[4] Percentage of Escaped Objects [5] Percentage of Non-Escaped Objects [6] Percentage of Total Objects
[7] ObjectStreamClass [8] ObjectInputStream [9] ObjectStreamClass$WeakClassKey

Table C.9: Top 10 non-escaping objects - SPECjvm2008

Vita

Candidate's full name: Manfred Jendrosch

Universities attended:

Bonn-Rhein-Sieg University of Applied Sciences (2005 - 2008)
Bachelor of Science in Computer Science
Specialization: Telecommunication

Bonn-Rhein-Sieg University of Applied Sciences (2010 - 2013)
Master of Science in Computer Science
Specialization: Complex Software Systems

University of New Brunswick (2010 - 2013)
Master of Computer Science
Specialization: Software Development