Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Master's thesis

Garbage Collector for Multi-threaded Scheme using Native Threads

Bc. Oleg Gul

Supervisor: Ing. Jan Vraný, Ph.D.

9th May 2015

Acknowledgements

I would like to thank my family and friends for support during writing this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 9th May 2015 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Gul, Oleg. *Garbage Collector for Multi-threaded Scheme using Native Threads.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Tento práce popisuje návrh, implementaci a testování paralelní stlačování garbage collector s implementace bariéry zápisu založené na hardwarové ochraně paměti. Realizace část této práce obsahuje integrace vytvořených Garbage Collector (GC) do existující implementace programovacího jazyka Scheme – TinyScheme

Klíčová slova Parallel Garbage Collector, Parallel Garbage Collection, Automatic Memory Management, Write Barrier, Scheme, TinyScheme.

Abstract

This thesis describes design, implementation and testing of parallel compacting garbage collector with the implementation of write barrier based on hardware memory protection. Implementation part of this thesis includes integration of created GC into existing implementation of the Scheme programming language – TinyScheme.

Keywords Parallel Garbage Collector, Parallel Garbage Collection, Automatic Memory Management, Write Barrier, Scheme, TinyScheme.

Contents

List of Figures

List of Tables

Introduction

Garbage collection is the automatic reclamation of computer storage. While in many systems programmers must explicitly reclaim heap memory at some point in the program, by using a "free" or "dispose" statement; garbage collected systems free the programmer from this burden. The GC function is to find data objects that are no longer in use and make their space available for reuse by the the running program. An object is considered garbage (and subject to reclamation) if it is not reachable by the running program via any path of pointer traversals. Live (potentially reachable) objects are preserved by the collector, ensuring that the program can never traverse a "dangling" pointer into a deallocated object.

Garbage collection is necessary for fully modular programming, to avoid introducing unnecessary intermodule dependencies. A software routine operating on a data structure should not have to depend what other routines may be operating on the same structure, unless there is some good reason to coordinate their activities. If objects must be deallocated explicitly, some module must be responsible for knowing when other modules are not interested in a particular object.

Since liveness is a global property, this introduces nonlocal bookkeeping into routines that might otherwise be locally understandable and flexibly composable. This bookkeeping inhibits abstraction and reduces extensibility, because when new functionality is implemented, the bookkeeping code must be updated. The runtime cost of the bookkeeping itself may be significant, and in some cases it may introduce the need for additional synchronization in concurrent applications.

The unnecessary complications and subtle interactions created by explicit storage allocation are especially troublesome because programming mistakes often break the basic abstractions of the programming language, making errors hard to diagnose. Failing to reclaim memory at the proper point may lead to slow memory leaks, with unreclaimed memory gradually accumulating until the process terminates or swap space is exhausted. Reclaiming memory too

soon can lead to very strange behavior, because an object's space maybe reused to store a completely different object while an old pointer still exists. The same memory may therefore be interpreted as two different objects simultaneously with updates to one causing unpredictable mutations of the other.

These programming errors are particularly dangerous because they often fail to show up repeatably, making debugging very difficult – they may never show up at all until the program is stressed in an unusual way. If the allocator happens not to reuse a particular object's space, a dangling pointer may not cause a problem. Later, after delivery, the application may crash when it makes a different set of memory demands, or is linked with a different allocation routine. A slow leak may not be noticeable while a program is being used in normal ways – perhaps for many years – because the program terminates before too much extra space is used. But if the code is incorporated into a long-running server program, the server will eventually exhaust the available memory and crash.

These problems lead many applications programmers to implement some form of application-specific garbage collection within a large software system, to avoid most of the headaches of explicit storage management. Many large programs have their own data types that implement reference counting, for example. Because they are coded up for a one-shot application, these collectors are often both incomplete and buggy. The garbage collectors themselves are therefore often unreliable, as well as being hard to use because they are not integrated into the programming language. The fact that such kludges exist despite these problems is a testimony to the value of garbage collection, and it suggests that garbage collection should be part of programming language implementations [1].

Taking into account all benefits connected with usage of GC instead of manual memory management, the task of design and implementation of efficient GC becomes very actual and important. In some cases using GC can speedup program execution because the whole application architecture becomes more elegant and less copy operations of same objects between different modules are performed. Also, some implementations allow faster allocations of new objects. But still, GC introduces generally unpredictable stalls to the execution of main program. During these pauses GC does those parts of its job which can't be done simultaneously with the application.

The goal of this thesis is to design and implement GC which would minimize these stalls allowing Virtual Machine (VM) to run as smoothly as possible. Also, it should be possible to use designed GC multithreaded environment, which seems to be quite natural and obvious nowadays.

The structure of thesis is following:

Chapter 1 gives a quick overview of some existing GC implementations which work in multithreaded environments.

Chapter 2 describes the process of garbage collection in more detail. Also, the design of implemented GC is given there.

Chapter 3 provides detailed technical description of implemented GC.

Chapter 4 contains results of testing of implemented GC integrated into TinyScheme.

Chapter 5 lists some possible enhancements for the GC which don't belong to the core functionality but are very convenient for VM implementers and end users.

Chapter 1

Existing Implementations

This chapter gives a quick overview of some existing GC implementations with focus on parallel collection.

1.1 Java

The Java HotSpot virtual machine includes four garbage collectors as of J2SE 5.0 update 6. All the collectors are generational. This section describes the generations and the types of collections, and discusses why object allocations are often fast and efficient. It then provides detailed information about each collector [2].

1.1.1 HotSpot Generations

Memory in the Java HotSpot virtual machine is organized into three generations: a young generation, an old generation, and a permanent generation. Most objects are initially allocated in the young generation. The old generation contains objects that have survived some number of young generation collections, as well as some large objects that may be allocated directly in the old generation. The permanent generation holds objects that the Java Virtual Machine (JVM) finds convenient to have the garbage collector manage, such as objects describing classes and methods, as well as the classes and methods themselves.

The young generation consists of an area called Eden plus two smaller survivor spaces. Most objects are initially allocated in Eden. (As mentioned, a few large objects may be allocated directly in the old generation.) The survivor spaces hold objects that have survived at least one young generation collection and have thus been given additional chances to die before being considered "old enough" to be promoted to the old generation. At any given time, one of the survivor spaces holds such objects, while the other is empty and remains unused until the next collection [2].

1.1.2 Fast Allocation

As you will see from the garbage collector descriptions below, in many cases there are large contiguous blocks of memory available from which to allocate objects. Allocations from such blocks are efficient, using a simple bump-the-pointer technique. That is, the end of the previously allocated object is always kept track of. When a new allocation request needs to be satisfied, all that needs to be done is to check whether the object will fit in the remaining part of the generation and, if so, to update the pointer and initialize the object.

For multithreaded applications, allocation operations need to be multithread-safe. If global locks were used to ensure this, then allocation into a generation would become a bottleneck and degrade performance. Instead, the HotSpot JVM has adopted a technique called Thread-Local Allocation Buffers (TLABs). This improves multithreaded allocation throughput by giving each thread its own buffer (i.e., a small portion of the generation) from which to allocate. Since only one thread can be allocating into each TLAB, allocation can take place quickly by utilizing the bump-the-pointer technique, without requiring any locking. Only infrequently, when a thread fills up its TLAB and needs to get a new one, must synchronization be utilized. Several techniques to minimize space wastage due to the use of TLABs are employed. For example, TLABs are sized by the allocator to waste less than 1% of Eden, on average. The combination of the use of TLABs and linear allocations using the bump-the-pointer technique enables each allocation to be efficient, only requiring around 10 native instructions [2].

1.1.3 Serial Collector

With the serial collector, both young and old collections are done serially (using a single CPU), in a stop-the-world fashion. That is, application execution is halted while collection is taking place.

The serial collector is the collector of choice for most applications that are run on client-style machines and that do not have a requirement for low pause times. On today's hardware, the serial collector can efficiently manage a lot of nontrivial applications with 64MB heaps and relatively short worst-case pauses of less than half a second for full collections [2].

1.1.4 Parallel Collector

These days, many Java applications run on machines with a lot of physical memory and multiple CPUs. The parallel collector, also known as the throughput collector, was developed in order to take advantage of available CPUs rather than leaving most of them idle while only one does garbage collection work.

The parallel collector uses a parallel version of the young generation collection algorithm utilized by the serial collector. It is still a stop-the-world and

copying collector, but performing the young generation collection in parallel, using many CPUs, decreases garbage collection overhead and hence increases application throughput. Figure 1.1: illustrates the differences between the serial collector and the parallel collector for the young generation.

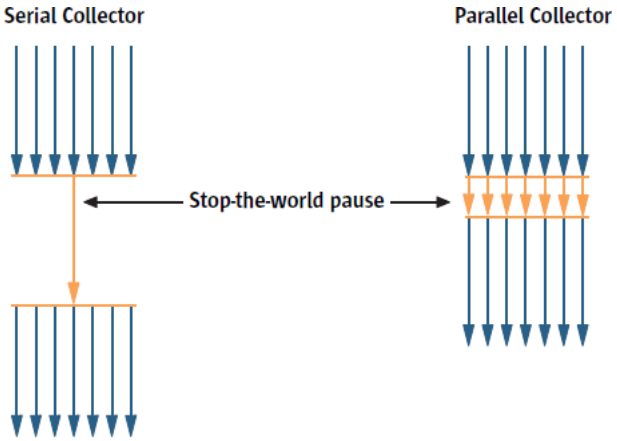

Figure 1.1: Comparison between serial and parallel young generation collection

Old generation garbage collection for the parallel collector is done using the same serial mark-sweep-compact collection algorithm as the serial collector.

Applications that can benefit from the parallel collector are those that run on machines with more than one CPU and do not have pause time constraints, since infrequent, but potentially long, old generation collections will still occur. Examples of applications for which the parallel collector is often appropriate include those that do batch processing, billing, payroll, scientific computing, and so on.

Choosing the parallel compacting collector (described next) over the parallel collector can be considered, since the former performs parallel collections of all generations, not just the young generation [2].

1.1.5 Parallel Compacting Collector

The parallel compacting collector was introduced in J2SE 5.0 update 6. The difference between it and the parallel collector is that it uses a new algorithm for old generation garbage collection.

Young generation garbage collection for the parallel compacting collector is done using the same algorithm as that for young generation collection using the parallel collector.

With the parallel compacting collector, the old and permanent generations are collected in a stop-the-world, mostly parallel fashion with sliding compaction. The collector utilizes three phases. First, each generation is logically

divided into fixed-sized regions. In the marking phase, the initial set of live objects directly reachable from the application code is divided among garbage collection threads, and then all live objects are marked in parallel. As an object is identified as live, the data for the region it is in is updated with information about the size and location of the object.

The summary phase operates on regions, not objects. Due to compactions from previous collections, it is typical that some portion of the left side of each generation will be dense, containing mostly live objects. The amount of space that could be recovered from such dense regions is not worth the cost of compacting them. So the first thing the summary phase does is examine the density of the regions, starting with the leftmost one, until it reaches a point where the space that could be recovered from a region and those to the right of it is worth the cost of compacting those regions. The regions to the left of that point are referred to as the dense prefix, and no objects are moved in those regions. The regions to the right of that point will be compacted, eliminating all dead space. The summary phase calculates and stores the new location of the first byte of live data for each compacted region.

In the compaction phase, the garbage collection threads use the summary data to identify regions that need to be filled, and the threads can independently copy data into the regions. This produces a heap that is densely packed on one end, with a single large empty block at the other end.

As with the parallel collector, the parallel compacting collector is beneficial for applications that are run on machines with more than one CPU. In addition, the parallel operation of old generation collections reduces pause times and makes the parallel compacting collector more suitable than the parallel collector for applications that have pause time constraints. The parallel compacting collector might not be suitable for applications run on large shared machines (such as SunRays), where no single application should monopolize several CPUs for extended periods of time. On such machines, number of threads used for garbage collection should be decreased or different collector should be selected [2].

1.1.6 Concurrent Mark-Sweep (CMS) Collector

For many applications, end-to-end throughput is not as important as fast response time. Young generation collections do not typically cause long pauses. However, old generation collections, though infrequent, can impose long pauses, especially when large heaps are involved. To address this issue, the HotSpot JVM includes a collector called the concurrent mark-sweep (CMS) collector, also known as the low-latency collector.

The CMS collector collects the young generation in the same manner as the parallel collector.

Most of the collection of the old generation using the CMS collector is done concurrently with the execution of the application.

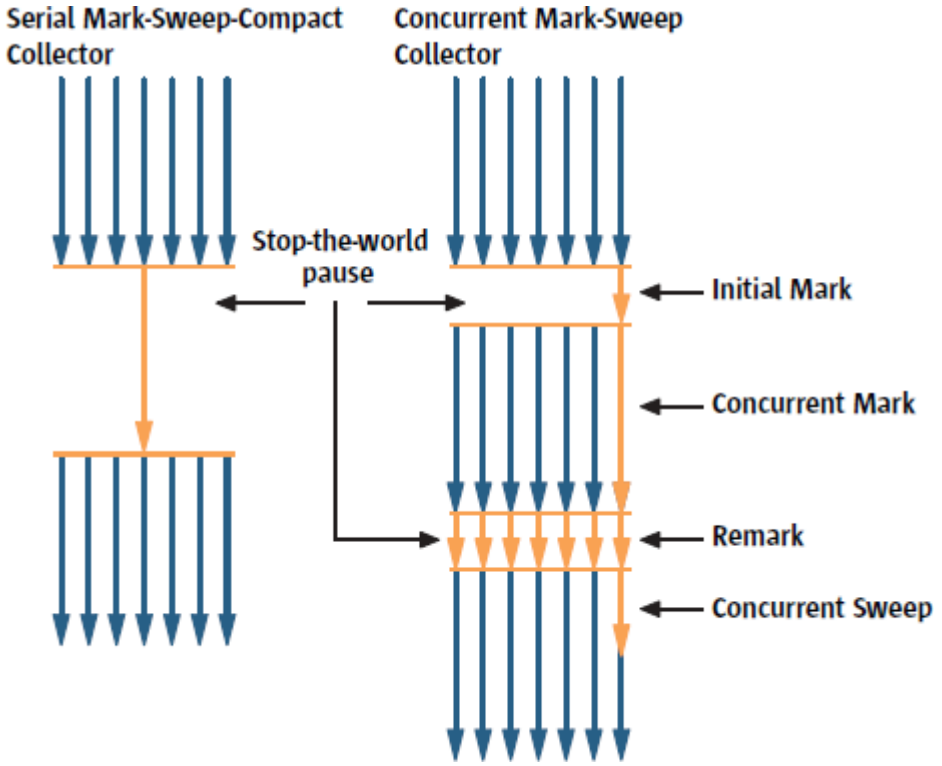

Figure 1.2: Comparison between serial and CMS old generation collection

A collection cycle for the CMS collector starts with a short pause, called the initial mark, that identifies the initial set of live objects directly reachable from the application code. Then, during the concurrent marking phase, the collector marks all live objects that are transitively reachable from this set. Because the application is running and updating reference fields while the marking phase is taking place, not all live objects are guaranteed to be marked at the end of the concurrent marking phase. To handle this, the application stops again for a second pause, called remark, which finalizes marking by revisiting any objects that were modified during the concurrent marking phase. Because the remark pause is more substantial than the initial mark, multiple threads are run in parallel to increase its efficiency.

At the end of the remark phase, all live objects in the heap are guaranteed to have been marked, so the subsequent concurrent sweep phase reclaims all the garbage that has been identified. Figure 1.2 illustrates the differences between old generation collection using the serial mark-sweep-compact collector and the CMS collector.

Since some tasks, such as revisiting objects during the remark phase, increase the amount of work the collector has to do, its overhead increases as well. This is a typical trade-off for most collectors that attempt to reduce pause times.

The CMS collector is the only collector that is non-compacting. That is, after it frees the space that was occupied by dead objects, it does not move the live objects to one end of the old generation.

This saves time, but since the free space is not contiguous, the collector can no longer use a simple pointer indicating the next free location into which the next object can be allocated. Instead, it now needs to employ free lists. That is, it creates some number of lists linking together unallocated regions of memory, and each time an object needs to be allocated, the appropriate list (based on the amount of memory needed) must be searched for a region large enough to hold the object As a result, allocations into the old generation are more expensive than they are with a simple bump-the-pointer technique. This also imposes extra overhead to young generation collections, as most allocations in the old generation occur when objects are promoted during young generation collections.

Another disadvantage the CMS collector has is a requirement for larger heap sizes than the other collectors. Given that the application is allowed to run during the marking phase, it can continue to allocate memory, thereby potentially continuing to grow the old generation. Additionally, although the collector guarantees to identify all live objects during a marking phase, some objects may become garbage during that phase and they will not be reclaimed until the next old generation collection. Such objects are referred to as floating garbage.

Finally, fragmentation may occur due to lack of compaction. To deal with fragmentation, the CMS collector tracks popular object sizes, estimates future demand, and may split or join free blocks to meet demand.

Unlike the other collectors, the CMS collector does not start an old generation collection when the old generation becomes full. Instead, it attempts to start a collection early enough so that it can complete before that happens. Otherwise, the CMS collector reverts to the more time-consuming stop-the-world mark-sweep-compact algorithm used by the parallel and serial collectors. To avoid this, the CMS collector starts at a time based on statistics regarding previous collection times and how quickly the old generation becomes occupied. The CMS collector will also start a collection if the occupancy of the old generation exceeds something called the initiating occupancy.

In summary, compared to the parallel collector, the CMS collector decreases old generation pauses – sometimes dramatically – at the expense of slightly longer young generation pauses, some reduction in throughput, and extra heap size requirements [2].

The CMS collector can be used in a mode in which the concurrent phases are done incrementally. This mode is meant to lessen the impact of long concurrent phases by periodically stopping the concurrent phase to yield back processing to the application. The work done by the collector is divided into small chunks of time that are scheduled between young generation collections. This feature is useful when applications that need the low pause times

provided by the concurrent collector are run on machines with small numbers of processors (e.g., 1 or 2).

CMS collector should be used if an application needs shorter garbage collection pauses and can afford to share processor resources with the garbage collector when the application is running. (Due to its concurrency, the CMS collector takes CPU cycles away from the application during a collection cycle.) Typically, applications that have a relatively large set of long-lived data (a large old generation), and that run on machines with two or more processors, tend to benefit from the use of this collector. An example would be web servers. The CMS collector should be considered for any application with a low pause time requirement. It may also give good results for interactive applications with old generations of a modest size on a single processor [2].

1.2 .NET

Garbage collector in .NET has 2 main modes of operation: **workstation** and **server**. They differ by the tradeoff between performance and heap efficiency. **Workstation** mode is tuned to give maximum UI responsiveness. **Server** mode is tuned to give maximum request throughput.

Since it is a generational collector, there are 3 generations of objects: Gen 0 (youngest), Gen 1 and Gen 2 (eldest).

1.2.1 Workstation GC Mode

This mode is designed to give maximum possible responsiveness to the user, and cut down on any pauses due to GC. Ideally, you want to avoid any perception of pauses or jerkiness in interactive applications so, to achieve this responsiveness, Workstation GC mode limits the number of thread suspensions.

Since .NET Common Language Runtime (CLR) version 1.0, Workstation GC could run as either concurrent or non-concurrent; this simply refers to which thread the GC runs on. In non-concurrent mode, thread execution of the application code is suspended, and the GC then runs on the application thread. It was designed for uniprocessor machines, where running threads concurrently wasn't an option.

As multicore/multiprocessor desktop machines are now very common, concurrent Workstation GC is now the norm and the default [3].

1.2.2 Concurrent Workstation GC

Concurrent GC has a separate thread for the GC to run on, meaning that the application can continue execution while the GC runs. Crucially, object allocation is also allowed as the GC is executing.

It's also worth remembering that concurrent GC only applies to full collections, so Gen 0 and Gen 1 collections still cause thread suspension. However, instead of just suspending all threads for the duration of a GC, the GC aims to only suspend threads for short periods, usually twice during execution. In contrast to non-concurrent GC, which suspends all threads for the duration of the entire GC process, concurrent GC is much less prone to latency issues.

Here's how it works. When a full concurrent GC takes places, the start and end positions of the allocated objects on the heap are determined, and garbage collection is limited to within this "GC domain". The nice thing about this is that the application can continue to allocate objects onto the heap outside of this domain.

The application can continue to allocate objects right up until the ephemeral segment limit is reached, which is the size of the segment minus a bit of space that we will call the "No Go Zone." Once this limit is reached, application execution is suspended until the full GC is finished [3].

1.2.3 Background Workstation GC Mode (.NET 4.0)

With background GC, a Gen 0 or Gen 1 GC can be triggered for the newly allocated objects while a full Gen 2 GC is in progress.

Gen 0 and Gen 1 now have tunable allocation thresholds which fire a background collection when exceeded, and allow rootless objects to be compacted and their space to be reclaimed. At the very least, this delays the inevitable reaching of the ephemeral segment boundary "No Go Zone."

It gets better: a background Gen 1 collection can now also create a new segment and copy Gen 0 objects into it just like in classic GC. That means there is no segment limit, which means no eventual thread suspensions due to exhaustion. There is, of course, a price to pay for all this, and that is the fact that application and foreground GC threads are suspended while background GC executes. However, due to the speed of Gen 0 and Gen 1 collections, this is usually only a small price [3].

1.3 Ruby

Both Rubinius and the JVM use concurrent garbage collection to reduce the amount of time your application spends waiting for collection to complete. When using concurrent garbage collection, the garbage collector runs at the same time as your application code. This eliminates, or at least reduces, pauses in your program due to garbage collection because your application doesn't have to stop and wait while the garbage collector runs.

Concurrent garbage collectors run in a separate thread from the primary application. Although in theory this could mean that your application will slow a bit because part of the CPU's time has to be spent running the GC thread, most computers today contain microprocessors with multiple cores,

which allow different threads to run in parallel. This means one of the cores can be dedicated to running the GC thread, leaving the other cores to run the primary application. (In practice, this still might slow down your application because fewer cores are available.)

MRI Ruby 2.1 also supports a form of concurrent garbage collection by performing the sweep portion of the mark-and-sweep algorithm in parallel while your Ruby code continues to run. This helps to reduce the amount of time your application is paused while garbage collection runs [4].

Chapter 2

High Level Design

2.1 GC Algorithm

Garbage collection automatically reclaims the space occupied by data objects that the running program can never access again. Such data objects are referred to as *garbage*. The basic functioning of a garbage collector consists, abstractly speaking, of two parts:

1. Distinguishing the live objects from the garbage in some way (garbage detection), and
2. Reclaiming the garbage objects' storage, so that the running program can use it (garbage reclamation) [1].

For detecting (marking) live object, implemented GC uses the concept of reachability. It means, that object is considered live (reachable) if it can be accessed by a sequence of traversing of pointers starting from some object.

There is a special set of objects which form Root Set. Objects in this set has special flag which marks them as root objects. These objects are considered to be always live. They are destroyed at the very end of program execution. Typically there are a few of them. For example, in TinyScheme there is only one such object – main interpreter object. Root objects can be created by passing special flag to the allocation routine.

Thus the set of live objects is simply the set of objects on any directed path of pointers from the root set. Any other object is considered to be garbage because there is no legal sequence of program actions that would allow the program to reach that object. Garbage objects therefore can't affect the future course of the computation, and their space may be safely reclaimed.

GCs that use concept of reachability is called "tracing" because it traces different paths of pointers starting from root set. Tracing GCs is a common choice for modern implementation opposite to reference counting, which seems to be better for integration into existing systems without garbage collection.

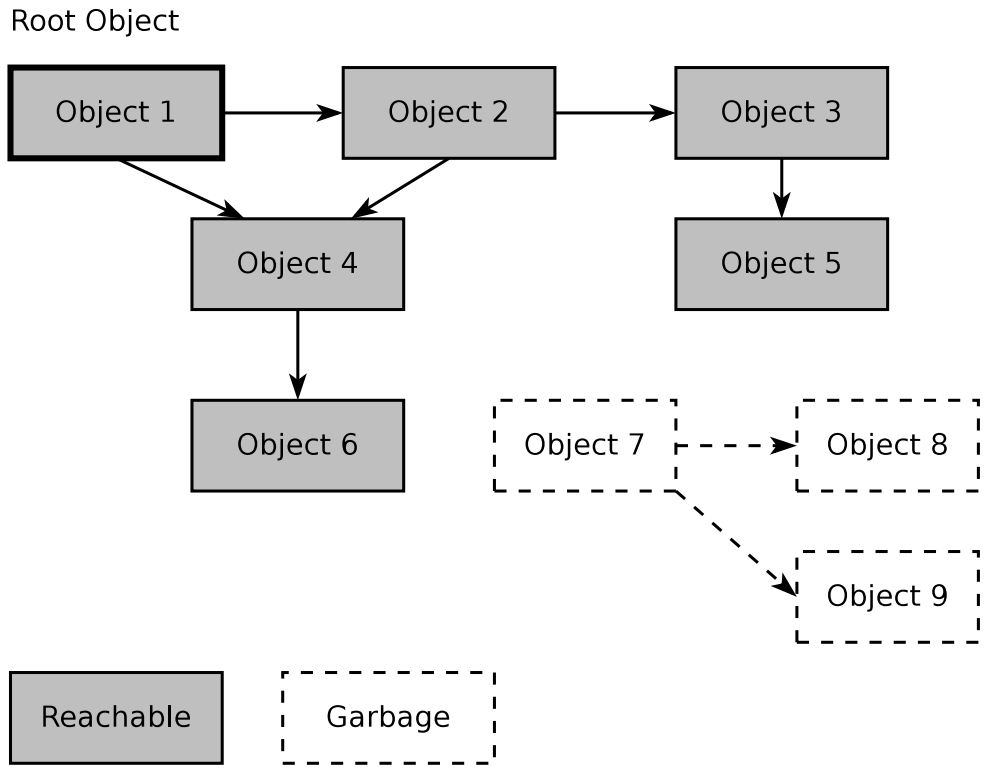

Figure 2.1: State of heap after tracing

In this figure, objects 1-6 are live, because they are reachable from object 1. Objects 7-9 are garbage, because they are not reachable from object 1.

This liveness criterion based on reachability is more relaxed than classical meaning. For example, temporary object may be used only once in some operation but still be accessible from local context. Such an object won't be considered as a garbage because it is still accessible. But eventually all these objects will be reclaimed by GC. This is what called conservative "liveness" of the object.

```
void SomeCode()
{
    // ...
    Object t = CreateTemporaryObject();
    ObjectUse1(t);
    ObjectUse2(t);
    // In fact, object pointed by t is garbage here.
    // But it isn't collected since t still
    // points to that object.
    SomeOtherCode(); // Not using t here.
    return;
```

```
}
```

Listing 2.1: Liveness conservatism example

For reclaiming of garbage implemented GC uses procedure called compaction. During compaction live objects are moved to the beginning of the pool, so they form one big sequence of reachable objects. During that move operation they overwrite existing garbage objects. At the end of compaction on the other end of the pool there is a big block of free memory.

Despite this approach requires moving of objects and updating references to moved objects, it is widely adopted because it offers major benefits:

- Improving locality of reference: live objects aren't scattered across whole heap but are gathered in one place; Objects which were created simultaneously continue to be neighbors; Whole heap is sorted from old to new objects with the increase of address
- Constant-time allocation of new objects: there is no need to keep track of list of free regions (freelist) in the heap. Instead of that only one pointer separating free and non-free regions is needed. Allocation is done simply by moving pointer by the distance equal to the size of new object
- No heap fragmentation means that all free memory is really accessible to the program

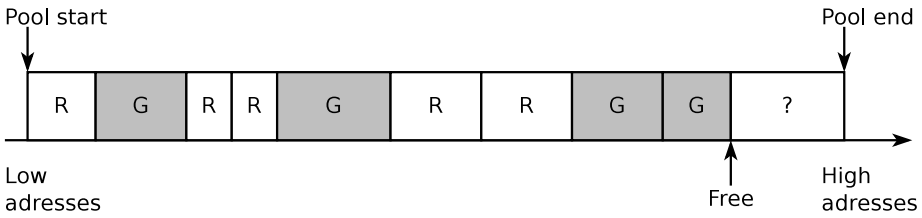

Figure 2.2: State of heap before compaction

On this figure, "R" mean reachable object, "G" – garbage object, "?" – uninitialized part of the heap.

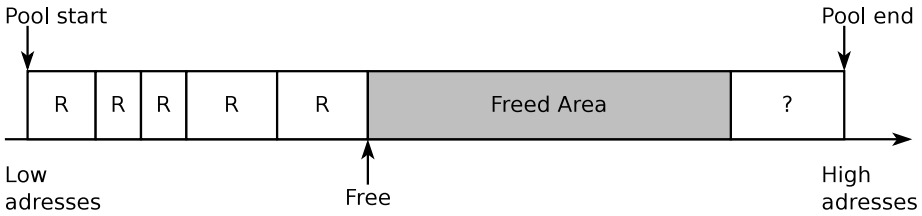

Figure 2.3: State of heap after compaction

These figures don't show references between objects. Whenever object is moved – all references pointing to it are updated to point to new object's location.

2.2 Object Representation

For GC to be able to traverse pointers in the objects and move them, it should be aware of the structure of the object: where pointers to other objects are located and what is size of the object. In dynamic languages the common solution for this is to have special hidden object header connected to each object and storing all needed information about the object. GC itself can use that header to store its object-specific information, which makes this approach even more attractive.

As this GC should be used together with TinyScheme, the approach with object header is used in this thesis. Keeping that in mind: object should contain following parts:

1. Header: all necessary information about object
2. Slots: pointers to other objects
3. Raw data: optional data which isn't traversed by GC but only holds some VM-specific data about the object

It should be possible to specify number of slots and size of raw data during allocation of the object. Also, GC should support object with no slots at all or only with raw data (for example, pure value objects).

Thus, object header should contain GC-specific data, number of slots, size of raw data and possibly some additional information.

2.3 Marking Process

As described before, marking is the process aimed to detect live objects by reference traversing.

For truly real-time applications, fine-grained incremental garbage collection appears to be necessary. Garbage collection cannot be carried out as one atomic action while the program is halted, so small units of garbage collection must be interleaved with small units of program execution [1].

Incremental tracing collector can pause in the middle of collection cycle while mutator continues, without ending up with inconsistent data.

Primitive garbage collectors, once they start a collection cycle, must either finish the task, or abandon all their work so far. This is often an appropriate restriction, but is unacceptable when the system must guarantee response

times; for example, in systems with a user interface and in real-time hardware control systems. Such systems might use incremental garbage collection so that the time-critical processing and the garbage collection can proceed effectively in parallel, without wasted effort [5].

In multithreaded environment, marking process could be performed in parallel with the main program (mutator), so the only time of "full stop" will be time of compaction, which can't be avoided because objects are moved during that phase of GC.

Due to using of compaction, this process becomes simpler: scanning starts from the beginning of the heap and new objects are allocated at the end of the heap. So scanning process "chases" free pointer which separates used and free parts of the heap. Process of marking stops when whole heap is scanned. In that case GC should pause VM, do compaction, resume VM and start new round of marking.

The only question which remains open in this approach is the modification of already scanned objects by mutator (that's how it got its name). If such modifications are ignored – it will lead to freeing live objects instead of garbage ones. That happens because of violation of Tri Color Invariant, which is described as follows:

The strong tri-color invariant is the property of a reference graph that there is no edge from a black node to a white node.

By preserving this property throughout tri-color marking, a tracing algorithm can ensure that the collector will not miss reachable objects, even if the mutator manipulates the graph during the collection. This invariant can also be used to ensure that a copying garbage collector doesn't confuse the mutator. Mutator actions might need to change the color of the nodes affected in order to preserve the invariant [5].

In the above definition term "black node" means scanned reachable object and "white node" means scanned garbage object. So, it is natural to prohibit black objects to point directly to white ones. Either black or white object should be gray – not yet scanned. But it is allowed for black object to point to gray and that gray to point to white. It just means that not all objects have been scanned yet.

An incremental scheme must have some way of keeping track of the changes to the graph of reachable objects, perhaps recomputing parts of its traversal in the face of those changes [1].

So, marking process should work correctly together with the mutator. Modifications caused by mutator should be correctly handled by the marking process. The solution for this is so-called write barrier: special operation which is performed during write to the object. If a pointer to a white object is stored into black object – write barrier marks white object as gray, so it will be rescanned again. Thus, Tri Color Invariant remains maintained.

Write barrier can be implemented in a different ways. The task of this thesis is to implement write barrier using hardware memory protection: pages

containing scanned object are marked as readonly. Whenever write to that page is performed segfault occurs passing control to custom handler which implements write barrier object making necessary fixes to object graph and notifying marking process. This is a way of synchronization between mutator and marker processes.

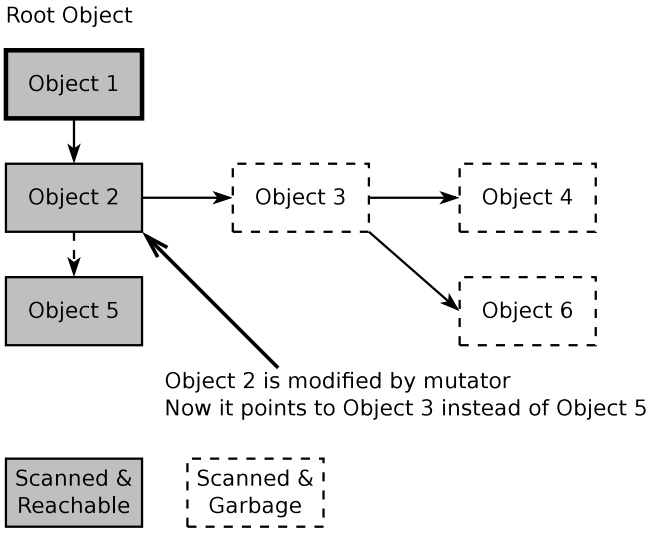

Figure 2.4: Violation of Tri Color Invariant

This figure shows the situation when objects 3, 4 and 6 will be collected but the shouldn't be.

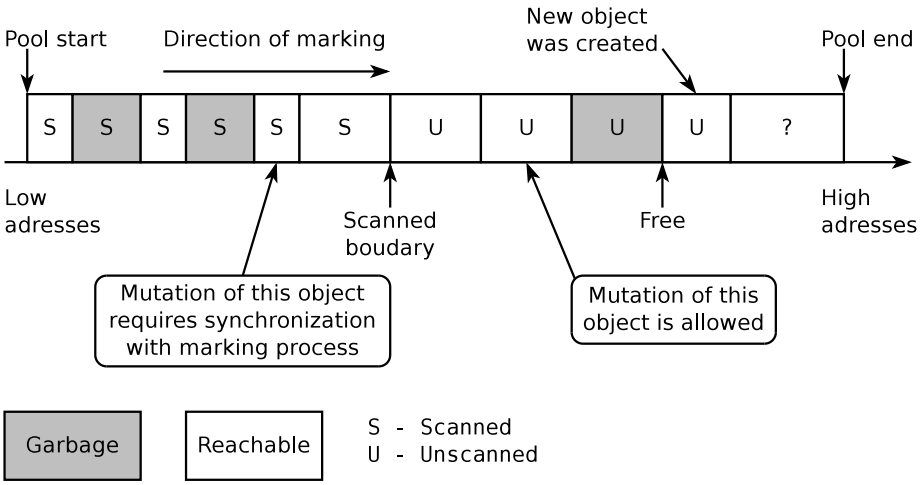

Figure 2.5: Process of parallel marking

When marking process scans object it ensures that memory page which

contains that object is marked as readonly. After marking is done and VM is paused all memory protection is removed and compaction is performed. When new cycle of garbage collection is started marking process starts to protect memory pages with scanned objects again.

An important characteristic of incremental techniques is their degree of conservatism with respect to changes made by the mutator during garbage collection. If the mutator changes the graph of reachable objects, freed objects may or may not be reclaimed by the garbage collector. Some floating garbage may go unreclaimed because the collector has already categorized the object as live before the mutator frees it. This garbage is guaranteed to be collected at the next cycle, however, because it will be garbage at the beginning of the next collection [1].

For a such kind of GC it is acceptable to be more relaxed (or conservative) about detection of garbage objects. But there are some conditions that have still be met by GC:

1. Eventually, all garbage objects must be collected: garbage mustn't leak
2. While garbage objects may be categorized as live for a while, no live object can be categorized as garbage

2.4 Compaction Process

Comparing to marking, compaction process is pretty straightforward. But there is one thing that has to be discussed more carefully: references to moved objects should be updated to point to new locations. To detect those invalid references whole heap should be scanned, because any object could reference moved ones. Taking into account that, process of compaction and updating references has to implemented efficiently enough. Otherwise it overcomes all benefits of compaction. The necessity of updating references is a price for using compaction.

Additionally to references within heap (from one object to another), there are implicit external references to objects. Consider following snippet of VM code, which creates new cons cell:

```
pointer cons(pointer car_value, pointer cdr_value)
{
    pointer result = alloc_object_with_2_slots();
    // WARNING: GC may occur here.
    car(result) = car_value;
    cdr(result) = cdr_value;
    return result;
}
```

Listing 2.2: External reference example

Pointers `car_value` and `cdr_value` are external references to some objects heap on the heap. They are external because GC isn't aware of them. They aren't stored is a slot of any object. When `cons` is called they are on the stack of VM. If collection occurs at point marked with corresponding comment, object pointed by `car_value` and `cdr_value` may be moved and those pointer will point at some random location in the heap. So, GC must have some mechanism for dealing with those external references.

Consider another example, which illustrates given problem from another point of view:

```
pointer create_new_binding(pointer env,
                           pointer name,
                           pointer value)
{
    pointer binding_cell = cons(name, value);
    // WARNING: GC may occur here.

    // List of pairs name-value.
    pointer binding_list = get_env_binding_list(env);

    // Push new pair to the head of the list.
    binding_list = cons(binding_cell, binding_list);

    // Save new binding list back to env.
    set_env_binding_list(env, binding_list);
    return binding_list;
}
```

Listing 2.3: External reference example with allocation

This piece of code creates new binding in the given environment. Again, if collection occurs in a commented place, bad things may happen. New object pointed by `binding_cell` isn't attached to the existing object graph. So, formally it is garbage an can be freed, so `binding_cell` will point to some random place. GC should have some mechanism to "catch" those kind of objects while they aren't attached to the object graph.

Support of external references and freshly created objects will be very convenient for VM implementers. Some implementations provide similar functionality. For example, .NET has `gcroot` class for that [6] and Java has `NewGlobalRef()` / `DeleteGlobalRef()` and `NewLocalRef()` / `DeleteLocalRef()` [7].

2.5 GC and VM Synchronization

Since marking is performed in parallel with VM, there should be some synchronization points between GC and VM.

Obviously, when there's no enough memory for allocation, collection may occur in the allocation call. Also, there should be a possibility for VM to trigger collection and to check with GC whether collection is needed.

From the point of view of GC, it should be able to pause all VM threads, verify that they are paused and then resume all VM threads again.

Chapter 3

Implementation Details

This chapter contains technical details about created GC and its integration into TinyScheme. For multithreading and synchronization linux implementation of Portable Operating System Interface (POSIX)[8] threads is used: Native POSIX Thread Library (NPTL).

Original TinyScheme was obtained from here: `https://bitbucket.org/janvrany/tinyscheme`.

Modified TinyScheme together with new GC is here: `https://bitbucket.org/aquirel/tinyscheme-newgc`.

Both versions also are stored on the enclosed CD.

3.1 Object Memory Layout

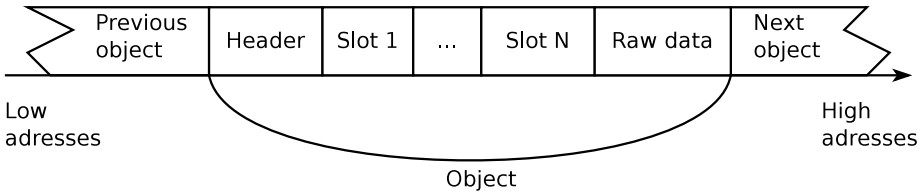

Figure 3.1: Object memory layout

Object header and slots have size of pointer on the target machine (i.e. `sizeof(void *)`). Size of raw data is rounded up to be a multiple of size of pointer. So, objects in the heap are always aligned.

Object header is the only mandatory part of the object. Slots and raw data are completely optional. Object may contain slots and raw data simultaneously, which is a convenient feature for VM implementers. Slots contain pointers to another objects on the heap (or `NULL`), so their content is used by GC during marking process and they are updated whenever corresponding objects are being moved. Contents of raw data is ignored by GC. Raw data can

be used to store values of primitive data types (numbers, booleans, strings, etc.) of the programming language working on top of VM or any other data which is private to VM or just has to be hidden from the programmer but still stay connected to the object.

As it can be seen from the figure, there are no gaps between objects. So, whole heap can be traversed from the beginning to the end (as singly linked list) without any additional data structures, which is quite natural, because all needed information is already in the heap. This feature appeared to be quite useful during work on this thesis.

Object header contains following fields:

Field	Bits	Description
`is_root`	1	Is this object a root
`is_scanned`	1	Object scanned state
`is_garbage`	1	Is this object garbage
`is_tracked`	1	Is it tracked by an external reference (see 3.5.14)
`flags`	8	VM-specific flags ignored by GC
`total_size`	10	Total size of the object in 4-byte units
`slot_count`	10	Number of slots object has

Table 3.1: Object header fields

Object's `is_scanned` bit doesn't have direct meaning: whether object is already scanned or not. This approach would require flipping of this bit after each marking phase for each object. Instead of that, there is an additional flag in the GC itself, and only this flag is flipped after marking. To determine whether object is scanned or not, object's `is_scanned` bit has to be compared with the corresponding flag of the GC.

Object's `flags` field can be used by VM to store any additional information about the object. For example, TinyScheme uses this field to store type of the object and its mutability status, since there is a few of builtin types in TinyScheme.

Total size of header fields is 32 bits. On x86_64 another 32 bits of padding are added to keep header aligned.

Traditionally, Lisp and Scheme implementations use cells of uniform format and size as objects. Actual content of cell is determined using type tag – small integer attached to each cell. This approach may be suitable for Lisp-like languages. This GC is more versatile: with its dynamic object format it can be used for any language. Moreover, support of raw data inside object is a great advantage for VM implementors.

3.2 Marking Process

Marking process takes advantage of the singly linked nature of the heap. It consists of the four following steps:

1. Take current object. If it is already scanned – move to next one
2. Mark objects referenced by current object as reachable and unscanned
3. Mark current object as scanned
4. Jump to the smallest (closest to beginning of the heap) unscanned object from step 2.

So heap may contain scanned ("S") object mixed with unscanned ("U") ones. Marking process stops when it reaches end of heap.

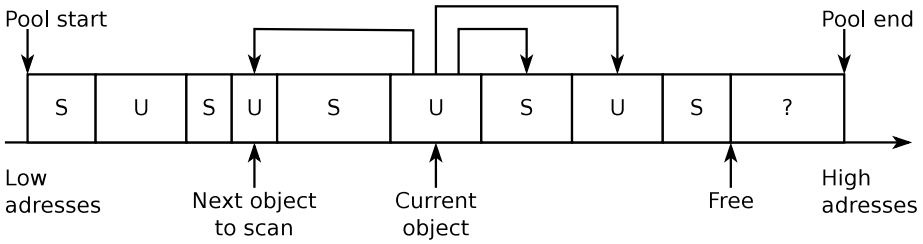

Figure 3.2: Example of marking process

Additionally, marking process protects pages with scanned objects. If page contains at least one scanned object – it will be protected for not to miss writes to scanned objects by Mutator.

When parallel marking needs to modify object header in protected part of the heap it unprotects its page temporarily.

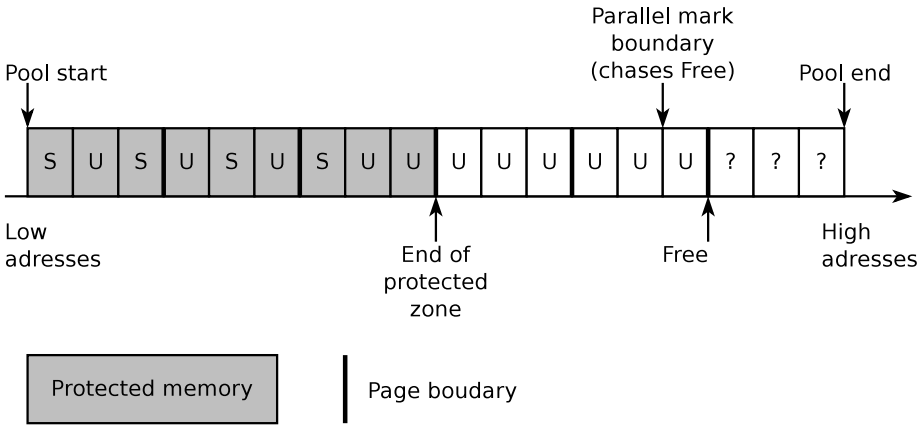

Figure 3.3: Memory protection of scanned objects during parallel marking

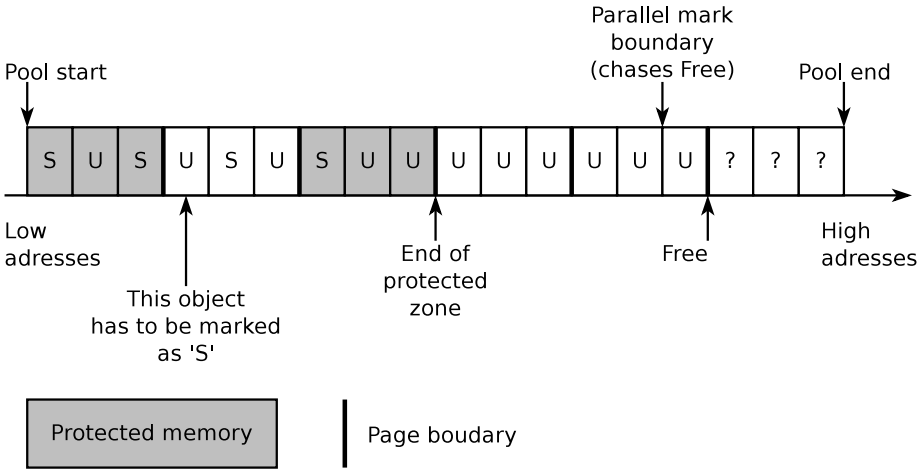

Figure 3.4: Temporary unprotection of the memory page

When parallel marking scans object beyond end of protected zone, it's page becomes protected.

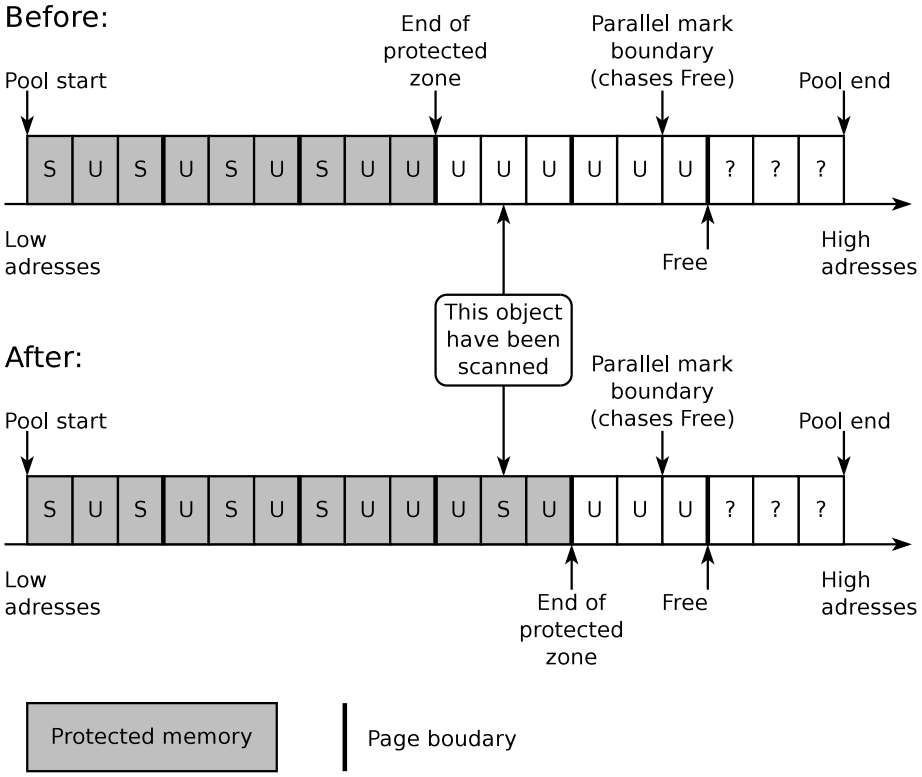

Figure 3.5: Protection of new memory page

After marking is finished and VM starts to wait for collection – finalization of garbage objects is performed.

3.3 Compaction Process

Compaction starts with defragmentation of garbage objects: they are glued together into special fill objects ("F"). These objects don't have slots and fill space between reachable objects. This approach allows to have consistent heap state during collection.

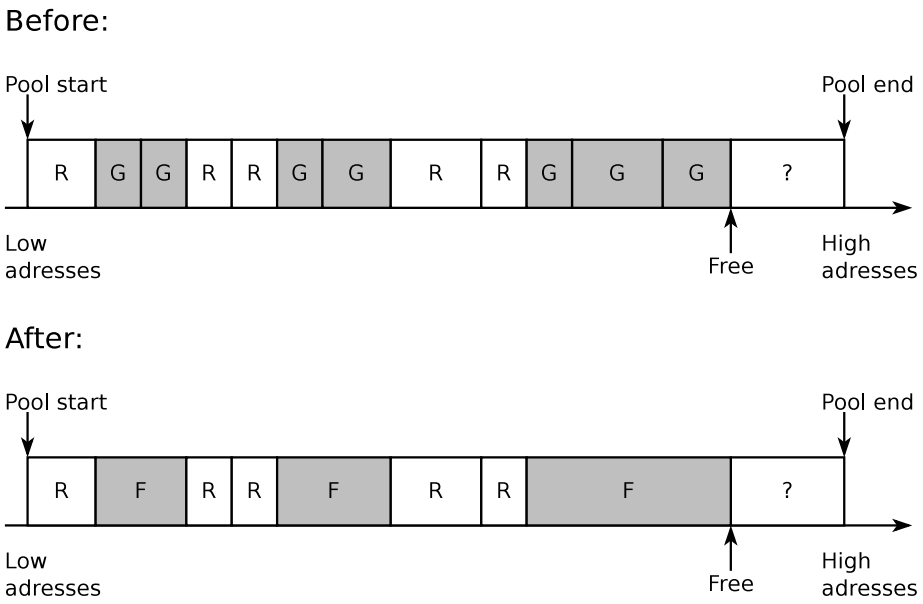

Figure 3.6: Defragmentation of garbage objects

After defragmentation is finished – blocks of reachable objects are moved to the start of the pool. Also, for each reachable block of reachable object information about its original position is remembered. This information is used later to update all references to moved objects in one pass over the heap.

So, compaction can be performed in 2 passes over the heap:

1. During first pass over the heap all reachable blocks are detected and moved. Additionally, for each moved block its original position and shift are remembered

2. During second pass, all references of reachable objects are tested against remembered ranges of reachable blocks. In case of match, matching pointer is updated with corresponding shift value.

There is a nice feature in the implementation of compaction: for a reachable block only its end and shift are remembered. Since this information is remembered even for not moved reachable blocks – beginning of the block isn't needed to update references correctly. Saved information about reachable blocks is automatically sorted because heap is traversed from beginning to the end.

It may seem to be an overhead to store information about not moved reachable block. But in fact, there can only be 1 such block: in the very beginning of the pool. Following reachable blocks of reachable objects are separated from previous ones by "F" objects.

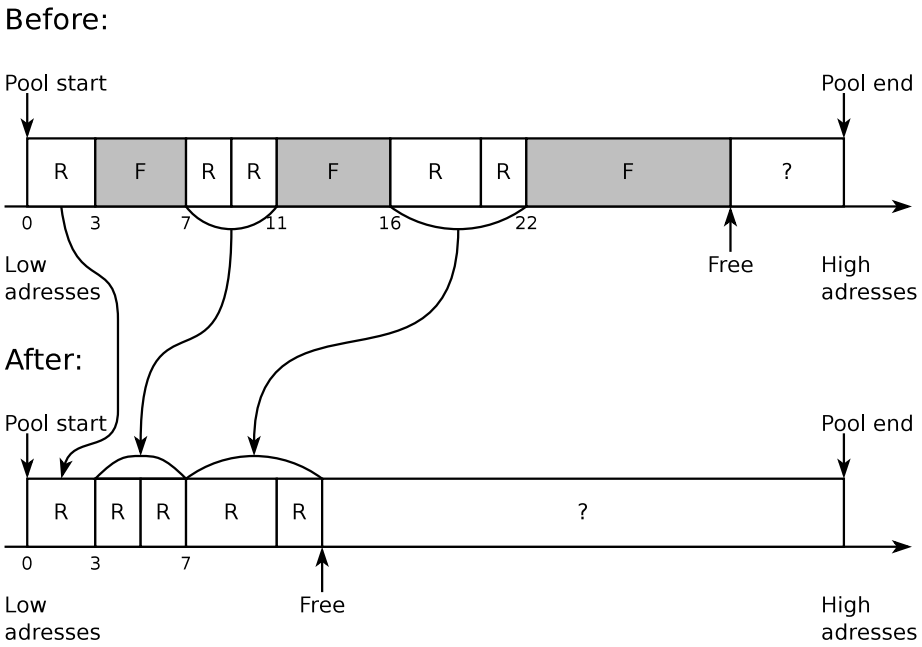

Figure 3.7: Example of compaction

Following information about moved reachable blocks is remembered:

Shift	**Block End**
0 - 0 = 0	3
7 - 3 = 4	11
16 - 7 = 9	22

Table 3.2: Information about moved blocks

Having this information, updating of references (and external references, see 3.5.14) can be done in one pass over the heap: if slot of non-garbage

objects is less then some block end boundary – its value is decreased by the corresponding shift.

Initial implementation of GC was moving reachable objects one by one, also updating all references separately after each move. This approach was very slow, so compaction was improved a lot comparing to the first version.

3.4 GC Thread

Described processes of marking and compaction are called from dedicated GC thread, which is the main part of GC. It can be described in pseudocode as follows:

```
void gc_thread()
{
    while (gc_is_working) // Not requested to stop by VM.
    {
        clear_statistics_counters();

        if (is_parallel_marking_enabled)
        {
            gc_mark();
            gc_pause_vm();
            gc_wait_for_vm_is_sleeping();
        }
        else
        {
            gc_wait_for_vm_is_sleeping(); // Just wait.
        }

        // Before collection.
        gc_fire_collection_callback();

        gc_unprotect_heap();

        // Mark objects referenced by external handles
        // (REFERENCE_TRACKING_MODE_HANDLE) as reachable.
        gc_mark_objects_as_reachable();

        // Do extra marking in case there are any
        // unscanned objects.
        gc_mark();

        // Set references to freed objects to NULL.
        gc_update_freed_tracked_references();

        // Do collection.
        gc_finalize_garbage();
        gc_defragment_garbage();
```

```
            gc_compact();
            gc_update_references();

            // After collection.
            gc_fire_collection_callback();

            gc_wake_up_vm();
        }
    }
```

Listing 3.1: `gc_thread()` pseudocode

3.5 GC Interface

This section describes interface of the GC. It also defined layout of object header, which was described before: see section 3.1.

3.5.1 Struct GCStatistics

Stores GC statistics. Used by `gc_get_statistics` function (3.5.33) and `gc-stat` scheme builtin (3.9.3.1).

3.5.2 Function gc_create(size_t pool_page_count, bool use_parallel_marking)

Creates new GC object with given number of pages in the pool. If `use_parallel_marking` is `false` – GC does marking process only when VM sleeps. This mode of operation is used in benchmarks to compare stop-the-world (marking and compaction on full stop) and parallel approaches.

3.5.3 Function gc_free(GC *gc)

Frees given GC object. Used by VM at the very end of execution.

3.5.4 Function gc_start(GC *gc)

Starts GC thread.

3.5.5 Function gc_stop(GC *gc)

Asks GC thread to stop and waits till it's stopped.

3.5.6 Function gc_set_notification_signal(GC *gc)

Sets SIGSEGV to be handled by GC. This is an implementation of write barrier based on memory protection faults.

3.5.7 Function gc_unset_notification_signal(GC *gc)

Restores original SIGSEGV handler.

3.5.8 Function gc_register_object_finalizer(GC *gc, gc_object_finalizer new_object_finalizer)

Registers user-supplied callback for object finalization.

3.5.9 Function gc_register_no_memory_callback(GC *gc, gc_no_memory_callback new_no_memory_callback)

Registers user-supplied callback to be called when there is no enough memory.

3.5.10 Function gc_register_collection_callback(GC *gc, gc_collection_callback new_collection_callback)

Registers user-supplied callback to be called before and after collection is performed.

3.5.11 Function gc_set_callback_data(GC *gc, void *new_callback_data)

Sets data pointer to pass to all callbacks. In TinyScheme it is used to pass main interpreter object to all callbacks.

3.5.12 Function gc_alloc(GC *gc, bool is_root, unsigned flags, size_t slot_count, size_t byte_count)

Allocates new object with given properties. `Flags` is user-defined object header field ignored by GC (see 3.1 section). In TinyScheme this field is used to store type of the object. This function may block till collection is completed if there is no enough memory.

3.5.13 Function gc_wait_for_collect_completed(GC *gc, bool check_vm_working)

Blocks calling thread till collection is completed. If `check_vm_working` is `true` then blocks only if GC has requested VM to pause. Otherwise, blocks unconditionally (i.e. when there is no memory).

3.5.14 Function gc_track_reference(GC *gc, ObjectHeader **reference, ReferenceTrackingMode tracking_mode)

Asks GC to update external `reference` to the object when it is moved. This Application Programming Interface (API) is heavily used by VM.

There are 3 kinds of external references:

Name	Description
REFERENCE_TRACKING_MODE_WEAK	During its lifetime reference can point only to single object. Object isn't marked as externally referenced. When its collected – external reference is set to NULL. This is some kind of weak references for VM.
REFERENCE_TRACKING_MODE_SINGLE_OBJECT	During its lifetime reference can point only to single object. This type of reference can't point to NULL. Object is marked as externally referenced. So, it won't be collected till reference exists.
REFERENCE_TRACKING_MODE_HANDLE	During its lifetime reference can point to different objects. This type of reference can point to NULL. Object pointed by this type of reference won't be collected till reference exists.

Table 3.3: Kinds of external references

Typical scenario for using of single object external reference in VM is following:

```
pointer cons(GC *gc, pointer car_value, pointer cdr_value)
{
    gc_track_reference(
        gc,
        &car_value,
        REFERENCE_TRACKING_MODE_SINGLE_OBJECT);

    gc_track_reference(
        gc,
        &cdr_value,
        REFERENCE_TRACKING_MODE_SINGLE_OBJECT);

    pointer result = alloc_object_with_2_slots();

    // car_value and cdr_value are protected
    // from being collected here.
    // If they are moved --
    // corresponding pointer are updated.

    car(result) = car_value;
    cdr(result) = cdr_value;
```

```
    gc_untrack_reference(gc, &car_value);
    gc_untrack_reference(gc, &cdr_value)
    return result;
}
```

Listing 3.2: Single object external reference example

Typical scenario for using of external reference of type "handle" in VM is following:

```
pointer cons(GC *gc, pointer some_object)
{
    // Temporary variable used in different operations.
    pointer t = NULL;

    gc_track_reference(
        gc,
        &t,
        REFERENCE_TRACKING_MODE_HANDLE);

    t = object_get_slot(gc, some_object, SOME_SLOT);

    // Some code trigerring collection.
    // t points to the same object here
    // and that object is still alive.

    t = object_get_slot(gc, t, SOME_OTHER_SLOT);

    // Some code trigerring collection.
    // Previous object pointed by t might be collected.
    // t points to new other object here, and that
    // object is protected from being collected.

    gc_untrack_reference(gc, &t);
    // t points to a valid object here.
    return t;
}
```

Listing 3.3: Handle external reference example

3.5.15 Function gc_untrack_reference(GC *gc, ObjectHeader **reference)

Stops tracking given `reference`. If `reference` has mode `REFERENCE_TRACKING_MODE_SINGLE_OBJECT` and it is last reference to the object – resets `is_tracked` bit of pointed object to `0` making object collectable again.

3.5.16 Function object_get_slot(GC *gc, ObjectHeader *o, size_t slot_index)

Returns value of given slot of the object `o`.

3.5.17 Function object_set_slot(GC *gc, ObjectHeader *o, size_t slot_index, ObjectHeader *value)

Sets value of given slot of the object `o` to the new `value`.

3.5.18 Function object_get_byte(ObjectHeader *o, size_t byte_index)

Returns pointer to given byte of raw data of the object `o`.

3.5.19 Function object_fill_slots(GC *gc, ObjectHeader *o, ObjectHeader *value)

Sets all slots of the object `o` to the given `value`.

3.5.20 Function gc_object_get_size_in_bytes(GC *gc, ObjectHeader *object)

Returns size of given object in bytes.

3.5.21 Function gc_object_get_header_field(ObjectHeader *object, int field)

Atomically returns given header `field` of the given `object`. Possible `field` values are following ones:
OBJECT_HEADER_FIELD_IS_ROOT
OBJECT_HEADER_FIELD_IS_SCANNED
OBJECT_HEADER_FIELD_IS_GARBAGE
OBJECT_HEADER_FIELD_IS_TRACKED
OBJECT_HEADER_FIELD_FLAGS
OBJECT_HEADER_FIELD_TOTAL_SIZE
OBJECT_HEADER_FIELD_SLOT_COUNT.

These values correspond to object header fields (see section 3.1).

3.5.22 Function gc_update_object_header(GC *gc, ObjectHeader *object, size_t number_of_flags, ...)

Atomically updates header of given object. Takes necessary locks and temporarily unprotects the object in case of parallel execution with VM. Takes `number_of_flags` of pais of arguments. Each pair correspond to header field and its new value. Example:

```
gc_update_object_header(
    gc,
    object,
    OBJECT_HEADER_FIELD_IS_SCANNED,
    (unsigned) true,
    OBJECT_HEADER_FIELD_IS_GARBAGE,
    (unsigned) false);
```

Listing 3.4: gc_update_object_header() example

3.5.23 Function gc_lock_object(GC *gc, ObjectHeader *object)

Locks object's memory and allows direct modifications of its raw data.

3.5.24 Function gc_unlock_object(GC *gc, ObjectHeader *object)

Unlocks object's memory.

3.5.25 Function gc_get_pool(GC *gc)

Returns pointer to start of the pool.

3.5.26 Function gc_get_pool_size(GC *gc)

Returns size of pool in bytes.

3.5.27 Function gc_get_used_size(GC *gc)

Returns used size of pool in bytes.

3.5.28 Function gc_vm_register_thread(GC *gc)

Notifies GC that new VM thread was born.

3.5.29 Function gc_vm_unregister_thread(GC *gc)

Notifies GC that VM thread died.

3.5.30 Function gc_vm_thread_awoke(GC *gc)

Notifies GC that existing VM thread awoke.

3.5.31 Function gc_vm_thread_fall_asleep(GC *gc)

Notifies GC that existing VM thread went to sleep. So, GC can safely do collection.

Typical pattern of usage of `gc_vm_*` thread-related functions is the following:

```
void vm_thread()
{
    gc_vm_register_thread();
    gc_vm_thread_awoke();

    // Some code.

    gc_vm_thread_fall_asleep();
    // Waiting for something.
    gc_vm_thread_awoke();

    // Some code.

    gc_vm_thread_fall_asleep();
    gc_vm_unregister_thread();
}
```

Listing 3.5: gc_update_object_header() example

3.5.32 Function gc_is_collection_needed(GC *gc)

Returns `true` if collection is requested by GC – when it has no objects to scan and is ready to start collection.

3.5.33 Function gc_get_statistics

Returns pointer to the `GCStatistics` struct (3.5.1) of given GC.

3.5.34 Function gc_set_parallel_marking(GC *gc, bool enable_parallel_marking)

Enables or disables parallel marking. See description of function `gc_create()` (3.5.2) for details.

3.6 TinyScheme Integration

Strictly speaking, `gc_track_reference()` 3.5.14 and `gc_untrack_reference()` 3.5.15 are enough to implement VM. But they have to be used carefully: all tracked references must be untracked, single object external references must not point to another object, etc.

The process of porting TinyScheme to new GC included a lot of tedious work regarding managing this references. Moreover, incorrect usage of this API could result in a lot of hard to find bugs: floating garbage, freeing used objects and so on.

To solve that, several C macros were introduced. They somehow mimic the behavior of C++ destructors and also provide automatic initialization. The idea is to have some set of local variables (pointers to heap objects), call `gc_track_reference()` for them at the beginning and `gc_untrack_reference()` at the end. And this should happen automatically. For all this local pointers `REFERENCE_TRACKING_MODE_HANDLE` is used, since it is the most reliable approach. Also, references managed by these macros usually point to several objects during reference's lifetime, since functions where this feature is used are quite big and complex (for example `opexe_*()` family).

The general implementation if this scope macro accepts type of managed object, its initial value, initializer and finalizer expressions. When applied to TinyScheme, this macro is used with `pointer` type, `NULL` as an initial value, `gc_track_reference()` as initializer and `gc_untrack_reference()` as finalizer.

In TinyScheme these macro is used as follows:

```
pointer some_function(pointer _scheme_object, other_args)
scope(scheme_object, sc, gc, x, y)
{
    local(x) = get(scheme_object, SCHEME_ARGS);
    local(y) = car(local(x));

    // Some code.

    scope_return local(x);
} endscope;
```

Listing 3.6: Scope example

In this example, `scheme_object`, `x` and `y` are automatically managed references. They can be accessed using `local(name)` macro. Also, TinyScheme customization of this macro automatically extracts pointers to `scheme` and `GC` structs from `scheme_object`. This operation is quite common in TinyScheme. It also defines inner `get`, `set` and `get_byte` functions, which don't need a `GC` pointer as parameter: it's automatically extracted from the scope. These functions are analogous to `object_get_slot()`, `object_set_slot()` and `object_get_byte()`.

Usage of this approach saved a lot of effort during porting TinyScheme to new GC.

This scope macro can be used everywhere, where original C scope is allowed:

```
int main()
```

```
{
    // Regular scope.
    scope(object1, object2)
    {
        local(object1) = 1;

        scope()
        {
            printf("Hello from empty scope!\n");
        } endscope;

        // Breakable scope.
        while (true)
        {
            scope(object1)
            {
                local(object1) = 42;
                scope_break;
            } endscope;
        }

        // Continuable scope.
        while (true)
        {
            scope(object2)
            {
                local(object2) = 42;
                scope_continue;
            } endscope;
        }

        scope(object1)
        {
            local(object1) = 42;
            scope_leave;
            printf("Not reached!\n");
        } endscope;

        local(object2) = 2;
    } endscope;

    scope(another_example);
    printf("another_example.\n");

    // Manual scope finalization.
    scope_finalize();
    endscope;

    return 0;
```

```
}
```

Listing 3.7: Scope example

The limitation of current implementation of scopes it that inner scopes can't access object in outer scopes. But it is not a problem for TinyScheme since it doesn't use nested scopes: 1 level is enough.

Scope macro uses some GNU Compiler Collection (GCC) C extensions: Locally Declared Labels and Nested Functions [9]. Also, C99 variadic macros are used.

But actually this scope macro solves only the half of the problem. When code only works with external references – everything is fine. But it's also needed to be able to work efficiently with slot content of usual objects. With the external references approach, slot value will be copied to local external reference and accessed through it. But during execution slot value of actual object may change (point to another object) while external reference still will be pointing to old object.

So the problem here is to access actual slot value of the object and still have benefits of scope macro. To solve that another C macro was introduced. It binds information about slot index and object to a locally defined name and provides convenient access to the slot. It doesn't introduce new external reference, which is good for GC for performance reasons.

This macro is used together with scope macro as follows:

```
pointer some_function(pointer _scheme_object, other_args)
scope(scheme_object, sc, gc, x)
{
    defslot(local(scheme_object), SCHEME_ARGS, sc_args);

    local(x) = car(slot(sc_args));
    setslot(sc_args, cdr(slot(sc_args));
    scope_return local(x);
} endscope;
```

Listing 3.8: Slot example

In this example, `sc_args` is a slot of `scheme_object` with index `SCHEME_ARGS`. This combination of slot and scope macros proved to be very handy in process of porting of TinyScheme to the new GC.

3.7 Implementation of Write Barrier

There are several special things about write barrier based on memory protection:

- SIGSEGV is delivered not to the particular thread, but to the whole process. It is not possible to handle several segfaults simultaneously

in each thread. So, heap memory writes from all threads should be serialized

- When SIGSEGV occurs – corresponding memory page should be unprotected to allow write. While page is unprotected, other threads may write to scanned the objects on the page. So, all writes to the page must be serialized

This special nature of SIGSEGV leads to a following pseudocode of the write barrier:

```
// Write given value to the slot of the object.
void write_barrier(object, slot, value)
{
    lock_mutation_mutex();
    lock_object_memory(object);

    object.slot = value;

    if (SIGSEGV_occured)
    {
        unprotect_page(object);
        object.slot = value;

        if (object_is_scanned(object))
        {
            // To be scanned again.
            mark_object_as_unscanned(object);
        }

        protect_page(object);
    }

    unlock_object_memory(object);
    unlock_mutation_mutex();
}
```

Listing 3.9: Write barrier pseudocode

Actually, there are 2 possible candidates to be scanned again: object written to or new slot value. This GC chooses first candidate to be rescanned since it showed slightly better performance. It's because in TinyScheme there are few object which are written to very frequently: main interpreter (scheme) objects. If new value will be rescanned, then in current round of collection each object that was written to the modified scheme object will be treated as live. But only last written value is actually live. So, this approach produces more floating garbage. It will be collected during next round, but still it hinders performance.

3.8 Description of Selected GC Internals

This section describes some tricks, which were used in GC implementation.

3.8.1 Flipping of Garbage and Scanned Bits of the Object

After collection is performed, all live objects must be marked as garbage and unscanned. This could be achieved by separate pass over the heap, but in this GC another approach is used. GC struct itself has garbage and scanned bits and only they are flipped. Do determine whether object is garbage, its garbage bit is compared with garbage bit of GC struct. If they match – object is garbage, otherwise – not. Same approach is used with scanned bit of the object.

3.8.2 Condition Variables Instead of Sleeps

Initial implementation of GC was using `nanosleep()` function to synchronize GC with VM: wait till VM is paused or resumed. But usage of POSIX condition variables showed better performance, despite additional mutex locks are needed to work with condition variables.

3.9 New TinyScheme Functions

This section lists new builtin functions of TinyScheme, which were added during implementation of this thesis.

These new functions don't expose whole POSIX possibilities. Just bare minimum for creating multithreaded applications is represented in current TinyScheme implementation.

3.9.1 Multithreading Support

This group of functions deal with thread handle stored in the interpreter object.

3.9.1.1 (current-thread)

Returns interpreter object corresponding to the calling thread.

3.9.1.2 (thread? object)

Returns #t if `object` is a thread. Otherwise returns #f.

3.9.1.3 (thread-finished? thread)

Returns #t if given `thread` is finished. Otherwise returns #f.

3.9.1.4 (thread-create callable)

Creates new thread which runs `callable`. Returns freshly-created thread object.

3.9.1.5 (thread-get-id thread)

Returns string containing id of given `thread` or id of current thread if function is called with no arguments.

3.9.1.6 (thread-join thread)

Joins given `thread` and returns its return value. If `thread` is already finished – returns immediately. Only 1 join to a working thread is allowed.

3.9.1.7 Example

```
TinyScheme 1.41
ts> (current-thread)
#<THREAD 7fdb78d73700>
ts> (thread-finished? (current-thread))
#f
ts> (thread? (current-thread))
#t
ts> (thread? 1)
#f
ts> (thread-get-id)
"7fdb78d73700"
ts> (define t (thread-create (lambda () (display "Hello
   from thread!\n") 42)))
t
ts> Hello from thread!
(thread? t)
#t
ts> (thread-finished? t)
#t
ts> (thread-join t)
42
ts> (thread-get-id t)
"7fdb784ee700"
ts> (quit)
```

Listing 3.10: Example TinyScheme session

3.9.2 Mutex Support

These functions provide mutex support for TinyScheme. Mutexes by default are recursive.

3.9.2.1 (make-mutex)

Creates new mutex.

3.9.2.2 (mutex? object)

Returns #t if `object` is a mutex. Otherwise returns #f.

3.9.2.3 (mutex-lock mutex)

Locks given `mutex`. Returns #t in case of success. Otherwise returns string describing error occured.

3.9.2.4 (mutex-unlock mutex)

Unlocks given `mutex`. Returns #t in case of success. Otherwise returns string describing error occured.

3.9.2.5 (mutex-trylock mutex)

If given `mutex` is free – locks it and returns #t. If mutex is locked – immediately returns #f. In case of error returns string with error description.

3.9.2.6 Example

```
TinyScheme 1.41
ts> (define m (make-mutex))
m
ts> (mutex? m)
#t
ts> (mutex? 1)
#f
ts> (mutex-lock m)
#t
ts> (mutex-trylock m)
#t
ts> (mutex-unlock m)
#t
ts> (mutex-unlock m)
#t
ts> (mutex-unlock m)
"Operation not permitted"
ts> (quit)
```

Listing 3.11: Example TinyScheme session

3.9.3 Other Functions

3.9.3.1 (gc-stat)

Prints various GC statistics to default output. Returns #t.

```
TinyScheme 1.41
ts> (gc-stat)
GC Statistics:
Live: 5202, 129712b, 0.2474.
Garbage: 5892, 147392b, 0.2811.
Last Collection: 711, 18160b, 0.0346, 66 moves.
Duration: 0.0029s.
SIGSEGVs: 0 (total 36175).
Peak Usage: 524280b, 1.0000.
Current Usage: 130016b, 0.2480.
#t
ts> (quit)
```

Listing 3.12: Example TinyScheme session

First group of 3 numbers represent number of live objects, their total size in bytes and ratio of size of live object to total heap size.

Second group of 3 numbers represent same kind of information regarding garbage objects.

Then information about last collection is given: number of collected objects, their size in bytes, number of moves of blocks of reachable objects, duration of last collection (compaction phase).

Next 2 numbers mean number of memory protection faults (writes to scanned objects) during last marking and in total.

Then general memory usage is provided: peak and current usage in bytes and as a ratio.

3.9.3.2 (rand)

Returns pseudo-random number.

```
TinyScheme 1.41
ts> (rand)
58595509
ts> (rand)
1136457986
ts> (rand)
1524161434
ts> (quit)
```

Listing 3.13: Example TinyScheme session

Chapter 4

Testing

This chapter describes tests which were performed on TinyScheme with new GC.

4.1 Functional Testing

GC and TinyScheme can be built in 3 build configurations:

1. Debug
2. Profile
3. Release

by passing appropriate argument to `make`.

In debug configuration different assertions and checks are added to the GC code. They ensure that heap state is consistent, objects have correct value of header fields and many other checks. All these checks cause significant slowdown of the GC (especially `gc_assert_heap()`, which is also most important one), but this mode is used only for testing. Also, this configuration includes debugging information.

Profile configuration still includes debug information but assertions and checks are disable, so they do not blur the performance picture. This configuration is used for profiling, which can report more detailed results using debug information from the executable.

Release configuration is built without debug information, assertions and checks and with full optimization.

Also, for testing of GC itself dummy VM was created (file `vm.c`). It creates several threads which allocate objects of random size and modify existing objects. So, behavior of real VM is simulated in this way. This dummy VM was used during initial testing (before integration with TinyScheme) and also in cases of major changes in the implementation of GC.

4.2 Performance Testing

After GC implementation has been finished and all bugs (hopefully) were defeated, main testing which was performed was performance one. The methodology of performance testing in this thesis is the comparison of run times of original TinyScheme and TinyScheme with new GC on same programs.

For measuring of running time standard `time` command was used like this: `/usr/bin/time --format "%e" /path/to/scheme /path/to/benchmark.scm`.

Tests were performed on the machine with Intel Core i5 2410M CPU (max frequency 2.3 GHz), 8 GB of RAM under control of Debian GNU/Linux OS.

4.2.1 Benchmarks

Following benchmarks were used in performance testing.

Simple loop:

```
; Singlethreaded.
; Simple benchmark.

(define (f i) (when (< i 256) (f (+ 1 i))))

(f 0)
```

Listing 4.1: benchmark_1_st.scm

Comparison of running simple loop 4 times in a sequence or in parallel:

```
; Multithreaded.
; Illustration of parallelizing of same amount of work.

(define thread-count 4)

(unless (defined? 'thread-create)
  (define (thread-create f) (f)))

(define (f i) (when (< i 64) (f (+ 1 i))))

(do ((i 0 (+ 1 i)))
    ((= i thread-count))
    (thread-create (lambda () (f 0))))
```

Listing 4.2: benchmark_2_mt.scm

Parallel quicksort (runs as singlethreaded on original TinyScheme):

```
; Multithreaded.
; Parallel quicksort example.

(unless (defined? 'thread-create)
  (define (thread-create f) (f)))
```

```
(unless (defined? 'thread-join)
  (define (thread-join t) '()))

(unless (defined? 'rand)
  (define (rand) (random-next)))

(define (qsort-parallel-worker array l r)
  (let ((pivot (vector-ref array (quotient (+ l r) 2)))
        (left l)
        (right r))
    (do () ((> l r))
      (do ()
          ((>= (vector-ref array l) pivot))
          (set! l (+ l 1)))

      (do ()
          ((<= (vector-ref array r) pivot))
          (set! r (- r 1)))

      (when (<= l r)
        (let ((t (vector-ref array l)))
          (vector-set! array l (vector-ref array r))
          (vector-set! array r t))
        (set! l (+ l 1))
        (set! r (- r 1))))

    (let
      ((left-sorter-thread
        (thread-create
          (lambda ()
            (when (< left r)
              (qsort-parallel-worker array left r)))))

       (right-sorter-thread
        (thread-create
          (lambda ()
            (when (< l right)
              (qsort-parallel-worker array l right))))))

      (thread-join left-sorter-thread)
      (thread-join right-sorter-thread))))

(define (qsort-parallel array)
  (qsort-parallel-worker array
                         0
                         (- (vector-length array) 1)))

(define array-size 32)
```

```
(define array (make-vector array-size))

(do ((i 0 (+ 1 i)))
    ((= i (vector-length array)))
    (vector-set! array i (remainder (rand) array-size)))

(qsort-parallel array)
```

Listing 4.3: benchmark_3_mt.scm

Sequential quicksort:

```
; Singlethreaded.
; Quicksort example.

(define (thread-create f) (f))

(define (thread-join t) '())

(unless (defined? 'rand)
  (define (rand) (random-next)))

(define (qsort-parallel-worker array l r)
  (let ((pivot (vector-ref array (quotient (+ l r) 2)))
        (left l)
        (right r))
    (do () ((> l r))
      (do ()
          ((>= (vector-ref array l) pivot))
          (set! l (+ l 1)))

      (do ()
          ((<= (vector-ref array r) pivot))
          (set! r (- r 1)))

      (when (<= l r)
        (let ((t (vector-ref array l)))
          (vector-set! array l (vector-ref array r))
          (vector-set! array r t))
          (set! l (+ l 1))
          (set! r (- r 1))))

    (let
      ((left-sorter-thread
        (thread-create
          (lambda ()
            (when (< left r)
              (qsort-parallel-worker array left r)))))

       (right-sorter-thread
        (thread-create
```

```
          (lambda ()
            (when (< l right)
              (qsort-parallel-worker array l right))))))

      (thread-join left-sorter-thread)
      (thread-join right-sorter-thread))))

(define (qsort-parallel array)
(qsort-parallel-worker array
                       0
                       (- (vector-length array) 1)))

(define array-size 32)
(define array (make-vector array-size))

(do ((i 0 (+ 1 i)))
    ((= i (vector-length array)))
    (vector-set! array i (remainder (rand) array-size)))

(qsort-parallel array)
```

Listing 4.4: benchmark_4_st.scm

4.2.2 Results

When GC was finished integrated into TinyScheme, initial testing was performed. Test results are given in the first section of table 4.2.2.

Since they were not very pleasant, some profiling using valgrind (callgrind tool) [10] with kcachegrind [11] and gprof [12] was done. Based on profiling data: following optimizations were performed:

- In hot functions `gc_object_get_size()`, `object_get_slot()`, `object_set_slot()` and `object_get_byte()` safety checks were moved to debug configuration
- Events instead of sleeps were used for synchronization of VM and GC
- In `gc_alloc()` function, VM thread goes to sleep only if `alloc_mutex` is locked
- Flipping of global `garbage_bit` in one place instead of iterating over all objects
- Functions `gc_object_get_header_field()`, `gc_object_get_size()` and `object_get_slot()` were translated to macros using excellent feature of GCC: Statements and Declarations in Expressions [9]

- For compaction phase, new function was introduced: `gc_object_get_size_exclusive()`, which uses direct access to the object header instead of the atomic load.

After these optimizations results were much better than before – see second section of table 4.2.2. But still much worse than original TinyScheme. Another interesting point is that test cases without parallel marking started to overcome those with parallel marking enabled.

So, another round of optimizations was performed:

- Usage of pointer arithmetics instead of indexed access to internal dynamic arrays, which store external references
- Updating all references to moved objects in one pass using remembered information about reachable blocks instead of separate pass for each move
- `get_cell()` function was optimized and functions using it (`cons()`, `immutable_cons()`, `mk_closure()`, `mk_continuation()`, `_cons()`) were turned to macros.

Results of these optimizations are shown in the third section of table 4.2.2 and they are better than previous ones.

According to profiling data, total time spent in GC thread is 30-35% instead of 70-80% as before optimizations. But speedup is not so dramatic as after previous optimizations. And still, cases without parallel marking continue to overcome.

Last round of optimization was about following improvements:

- Unprotecting of a page after first SIGSEGV on it and marking all objects there as reachable and unscanned – should reduce number of SIGSEGVs
- Inline singlethreaded version of `object_set_slot()` used during reference update
- Delay time of marking thread adjusted

Results of these optimizations are in the fourth section of table 4.2.2. As it will be shown later in this section, number of SIGSEGVs is quite important for overall performance. It was decreased during these optimizations but number of floating garbage increased instead, so total improvement wasn't very big.

Another important point is the duration of delays during marking process. If it's too small then parallel marking process almost instantly traverses whole object graph and marks it as reachable. On subsequent changes. another live objects are being added, so total amount of floating garbage becomes enormous. On the other hand, if parallel marking is too slow, it becomes not

effective and its cost overcomes its profit. So, this duration of delay has to be carefully adjusted, so that GC thread runs in the "background" of the VM.

Benchmark	**Parallel Mark Enabled**	**Parallel Mark Disabled**	**Original TinyScheme**
Initial benchmark results			
Simple Loop	9.34s	9.94s	0.14s
Parallel Loop	23.20s	28.35s	0.11s
Parallel QuickSort	56.97s	100.16s	0.37s
Singlethreaded QuickSort	55.72s	72.37s	0.33s
Benchmark results after first optimization round: events for synchronization, flipping garbage bit in one place, turning hot functions to macros			
Simple Loop	5.83s	1.74s	0.10s
Parallel Loop	5.97s	4.33s	0.10s
Parallel QuickSort	15.36s	11.63s	0.28s
Singlethreaded QuickSort	17.35s	4.84s	0.30s
Benchmark results after second optimization round: one pass to update all references, optimizing some internal TinyScheme functions, usage of pointer arithmetics for arrays			
Simple Loop	4.99s	1.16s	0.09s
Parallel Loop	3.47s	3.24s	0.10s
Parallel QuickSort	12.91s	8.52s	0.33s
Singlethreaded QuickSort	15.57s	3.75s	0.29s
Benchmark results after third optimization round: unprotecting pages after first SIGSEGV, adjusted mark delay			
Simple Loop	4.89s	1.16s	0.11s
Parallel Loop	4.75s	3.74s	0.10s
Parallel QuickSort	13.35s	10.56s	0.28s
Singlethreaded QuickSort	14.11s	4.19s	0.27s

Table 4.1: Benchmark results for different configurations

Benchmark	**Peak**		**Final**	
	Size	Ratio	Size	Ratio
Simple Loop	524288b	1.0000	369000b	0.7038
Parallel Loop	524280b	1.0000	338896b	0.6464
Parallel QuickSort	524288b	1.0000	436040b	0.8317
Singlethreaded QuickSort	524288b	1.0000	256592b	0.4894

Table 4.2: Benchmark memory usage

Peak and final memory usages for each benchmark are given in table 4.2.2.

Another round of testing with different problem size and pool size was performed:

- benchmark_1_st.scm (Simple Loop): number of iterations is 512 instead of 256
- benchmark_2_mt.scm (Parallel Loop): number of iterations for single thread is 128 instead of 64
- benchmark_3_mt.scm (Parallel QuickSort): array size is 64 instead of 32
- benchmark_4_st.scm (Singlethreaded QuickSort): array size is 64 instead of 32

As it can be seen in table 4.2.2, running times are also doubled and the gap between new and old schemes has grown even more, as also a gap between cases with and without parallel marking.

Benchmark	**Parallel Mark Enabled**	**Parallel Mark Disabled**	**Original TinyScheme**
Original configuration, pool 128 pages			
Simple Loop (256 iterations)	4.89s	1.16s	0.11s
Parallel Loop (64 iterations)	4.75s	3.74s	0.10s
Parallel QS (32 elements)	13.35s	10.56s	0.28s
Singlethreaded QS (32 elements)	14.11s	4.19s	0.27s
Doubled problem size, pool 128 pages			
Simple Loop (512 iterations)	9.40s	2.33s	0.19s
Parallel Loop (128 iterations)	12.77s	7.57s	0.19s
Parallel QS (64 elements)	45.57s	23.17s	0.67s
Singlethreaded QS (64 elements)	32.26s	8.57s	0.65s
Original configuration, pool 256 pages			
Simple Loop (256 iterations)	4.52s	1.50s	0.11s
Parallel Loop (64 iterations)	4.86s	3.66s	0.10s
Parallel QS (32 elements)	14.12s	8.67s	0.28s
Singlethreaded QS (32 elements)	16.32s	4.79s	0.33s
Doubled problem size, pool 256 pages			
Simple Loop (512 iterations)	9.11s	2.72s	0.22s
Parallel Loop (128 iterations)	10.41s	7.25s	0.19s
Parallel QS (64 elements)	33.30s	28.06s	0.62s
Singlethreaded QS (64 elements)	30.41s	9.91s	0.68s

Table 4.3: Benchmark results for different pool and problem sizes

Another testing was performed on original problems, but with doubled size of pool: 256 pages instead of 128. These results are comparable to those with original pool size. But numbers should be less since collections should happen rarely.

Taking into account this strange behavior and overcoming of cases with disabled parallel marking, the testing of write barrier performance was performed.

For that, separate small application was created. This application performs given number of segmentation violations and handles them the same way as GC does. The pseudocode of main loop of this testing application is the following:

```
while (not_all_iterations_finished())
{
    protect_page();
    do_write();
    // SIGSEGV occurs.
    return_from_sigsegv_handler();
    unprotect_page();
    do_write();
}
```

Listing 4.5: SIGSEGV testing application pseudocode

At first, number of segmentation violations of new TinyScheme is measured, till it's ready to run user code (after processing `init.scm`):

```
$ /usr/bin/time -f "%e" ./scheme -c "(gc-stat)"
Live: 5101, 127280b, 0.2428.
Garbage: 20766, 524280b, 1.0000.
Last collection: 15665, 397000b, 278 moves,
13707 sigsegvs (total 42648),
0.0066s.
0.36
```

Listing 4.6: Measuring of number of segmentation violations in new TinyScheme

So, 42648 SIGSEGVs and 0.36 seconds.

Then, time for same number of SIGSEGVs was measured in a separate test application:

```
$ ./sigsegv 42648
Timespan: 0.1607s.
```

Listing 4.7: Simulating given number of SIGSEGVs

Surprisingly, half of time is spent only in SIGSEGV handling, which turns out to be quite expensive operation. Taking into account this fact, other portion of time should be spent in locks, which were added to use memory

protection as a basis for write barrier implementation: `mutation_lock` and `page_locks` for particular pages.

So, now it is clear why disabled parallel marking showed better performance: after optimizing biggest bottlenecks during optimization sessions, cost of handling protection faults came into the first place among all other performance issues.

Same results were obtained by Zorn[13]:

> The severe disadvantage of the simplistic approach (write barrier using write protection faults) is the high cost of the operating system handling a protection violation (typically several thousands instructions). Because, in the past, protection faults have been associated with program errors or security violations, operating system designers have not attempted to implement these faults with great efficiency. Faulting on **every** store is clearly too expensive.

The proposed approach to fix this performance issue is that after first failing write page is unprotected and all subsequent writes do not fail. To detect modified pages, modification of OS virtual memory interface is suggested: to give access to dirty bits, so all modified pages will be rescanned.

In this thesis, another option was used: unprotect memory page after first SIGSEGV on it and mark all objects on this page as unscanned and reachable. The original goal was achieved. This approach gives 35017 SIGSEGVS on startup (comparing to 42648 before). This decrease is not so big, as expected. Also it doesn't improve performance significantly. The reason for that is big increase of amount of floating garbage, which appears after unprotecting a page. This increase of garbage leads to more frequent collections, which take time.

Another question that arises is whether it will be effective to avoid compaction – do it only for some collections or after hitting some threshold. According to profiling data, 6.57% of time was spent in `gc_compact()` (24.29% in `gc_mark()`, 34.30% in `gc_thread()`). Taking into account this and the fact that avoiding of compaction will introduce free lists (explicit or implicit – walking over heap) with additional overhead, it may be concluded that avoiding of compaction will not increase overall performance, since TinyScheme allocates object quite often (cons cells).

Moreover, collection round without compaction means that the only useful job done is finalization of garbage.

Chapter 5

Possible Enhancements

5.1 Reimplementation of Write Barrier

According to performed performance tests, write barrier should be reimplemented. One possibility for it is to use software write barrier and ensure that its implementation is as small as possible, so it would be inlined effectively.

Another possibility is to unprotect memory page after first memory protection fault and record that page is dirty in some external place. After that, dirty pages should be rescanned either in parallel or just before compaction.

Also, another implementation of write barrier possibly will allow to get rid of some global locks, which definitely shall improve overall performance.

5.2 Dynamic Heap Size

One of the first steps towards enhancing this GC is an ability to grow heap size dynamically upon request when there is no enough memory in the existing heap. Currently, heap size is set on startup and can't be changed during VM execution. This enhancement may be implemented using multiple independent pools. So, GC heap will consist of several independent memory regions.

Another approach is to reallocate existing pool.

First way seems to be better because it allows to give back unused memory to OS and doesn't depend on reallocation of a huge memory block (which can be impossible). But it will require major modifications to the existing GC implementation:

- marking procedure will have to traverse object graph correctly across multiple regions of memory
- compaction procedure will have to move objects across different memory regions or within same region.

5.3 Immediate Values

Another valuable enhancement is to add support for Immediate Values. Since they are quite small to allocate whole object to store them. For example, integers, floats, characters – these values could be stored inside a pointer instead of separate object. Pointer which holds Immediate Value can be recognized by its nonzero 2 or 3 lower bits, which are all zeros for a usual pointer. This feature should be particularly useful on x86_64 with its huge pointer size.

5.4 Delayed Finalization

GC has to run custom finalizer for objects which are freed. Currently it is done during VM sleep before compaction. But it should be possible to do finalization for some objects in parallel with VM, for example during GC waits for VM to sleep before compaction.

5.5 Reducing Number of Passes over Heap

Current implementation of marking sometimes goes back to previous heap objects during process of marking. Albeit those jumps back don't mean new pass over whole heap, they still introduce some amount of extra work. This extra work can be avoided using the queue of yet unscanned objects. This approach requires some extra memory but saves from needles jumping over objects. The question here is whether this trade-off reasonable or not.

5.6 Better Vectors Implementation

Currently vectors are implemented as objects with number of slots corresponding to the size of vector. But vector may contain thousands of elements and these huge vectors aren't supported by current implementation. Having 10 bits for storing object's size means that maximum possible size of the object is 4096 bytes which corresponds to 511 slots on x86_64.

It will be useful to have support of objects with large number of slots and their size limited only by available pool size.

Conclusion

As the result of this thesis, parallel compacting GC was implemented. Implemented GC provides well-defined interface which makes it easy to use both for implementation of new VMs and integration into existing ones.

New GC was integrated into existing TinyScheme implementation, which was also extended to support native multithreading. After some helper structures were created, which were described in the implementation details (chapter 4), the process of integration was almost automatic.

Performed performance testing revealed some major performance issues. All of them were fixed except one: implementation of write barrier, which was added as the most important item for the possible enhancements.

It was showed that SIGSEGV handling is quite expensive operation and can't be used for write barrier implementation, at least in the trivial way.

During performance optimizations, profiling tools (callgrind together with kcachegrind and gprof) helped a lot to discover bottlenecks. Otherwise, it would have taken a lot of time to find that places manually, if even possible to do that. Usage of these tools allowed to significantly improve performance of GC comparing to the original implementation.

Also, valgrind's memcheck tool helped to debug TinyScheme integration issues: memory leaks and incorrect accesses to memory.

Bibliography

[1] Wilson, P. R. Uniprocessor Garbage Collection Techniques. In *ACM Computing Surveys*, 1992.

[2] Microsystems, S. *Memory Management in the Java HotSpot^{TM} Virtual Machine*. 2006.

[3] Farrell, C.; Harrison, N. *Under the Hood of .NET Memory Management*. Simple Talk Publishing, 2011, ISBN 978-1-906434-74-8.

[4] Shaughnessy, P. *Ruby Under a Microscope: An Illustrated Guide to Ruby Internals*. No Starch Press, 2014, ISBN 978-1-59327-527-3.

[5] Memory Management Glossary [online]. Available from: `http://www.memorymanagement.org/glossary`

[6] MSDN Library [online]. Available from: `https://msdn.microsoft.com/library`

[7] Java Documentation [online]. Available from: `http://docs.oracle.com/javase/8`

[8] Open Group Base Specification Issue 7, 2013 Edition [online]. 2013. Available from: `http://pubs.opengroup.org/onlinepubs/9699919799`

[9] GCC Online Documentation [online]. Available from: `https://gcc.gnu.org/onlinedocs`

[10] Valgrind Documentation [online]. Available from: `http://valgrind.org/docs`

[11] KCachegrind Documentation [online]. Available from: `http://kcachegrind.sourceforge.net/html/Documentation.html`

[12] Gprof Documentation [online]. Available from: https://sourceware.org/binutils/docs/gprof

[13] Zorn, B. Barrier Methods for Garbage Collection. 1990.

[14] Wikipedia The Free Encyclopedia [online]. Available from: https://en.wikipedia.org/wiki

Appendix A

Glossary

Collector In a garbage-collected system, the part that executes the garbage collection code, which discovers unused memory and reclaims it [5].

Garbage Collector Garbage collector is an implementation of a garbage collection algorithm [5].

Heap The heap or free store is the memory area managed by dynamic allocation [5].

Immediate Value Immediate value is the representation of a value object as one or more machine words, as a register, or as a field in an instruction. Immediate data takes its name from the value of the object being immediately available, rather than requiring a load or indirection through a reference [5].

Mutator (AKA client program) In a garbage-collected system, the part that executes the user code, which allocates objects and modifies, or mutates, them. For purposes of describing incremental garbage collection, the system is divided into the mutator and the collector. These can be separate threads of computation, or interleaved within the same thread. The user code issues allocation requests, but the allocator code is usually considered part of the collector. Indeed, one of the major ways of scheduling the other work of the collector is to perform a little of it at every allocation. While the mutator mutates, it implicitly frees memory by overwriting references [5].

Pool In this thesis, same as heap.

Reachability An object is reachable if it is referred to by a root, or is referred to by a reachable object; that is, if it can be reached from the roots by

following references. Reachability is used as an approximation to liveness in tracing garbage collection [5].

Root Set The root set is the collection of roots that the mutator declares to the collector [5].

TinyScheme Minimalistic Scheme implementation in C.

Tri Color Invariant The strong tri-color invariant is the property of a reference graph that there is no edge from a black node to a white node. By preserving this property throughout tri-color marking, a tracing algorithm can ensure that the collector will not miss reachable objects, even if the mutator manipulates the graph during the collection. This invariant can also be used to ensure that a copying garbage collector doesn't confuse the mutator. Mutator actions might need to change the color of the nodes affected in order to preserve the invariant. Algorithms using this invariant are incremental update algorithms [5].

Virtual Machine A process VM, sometimes called an application virtual machine, or Managed Runtime Environment (MRE), runs as a normal application inside a host OS and supports a single process. It is created when that process is started and destroyed when it exits. Its purpose is to provide a platform-independent programming environment that abstracts away details of the underlying hardware or operating system, and allows a program to execute in the same way on any platform [14].

Appendix B

Acronyms

API Application Programming Interface.

GC Garbage Collector.

GCC GNU Compiler Collection.

JVM Java Virtual Machine.

NPTL Native POSIX Thread Library.

POSIX Portable Operating System Interface.

VM Virtual Machine.

Appendix C

Contents of Enclosed CD

- `readme.txt` ..... this description
- `bin` ..... binaries of new and original schemes and GC dummy VM
- `src` ..... sources
  - `tinyscheme` ..... sources of new TinyScheme and GC
  - `tinyscheme-orig` ..... sources of original TinyScheme
  - `thesis` ..... LaTeX sources of the thesis
    - `benchmarks` ..... benchmark sources and raw logs
    - `figures` ..... figures used in thesis
- `text` ..... thesis text
  - `task.pdf` ..... thesis topic assigment in PDF
  - `thesis.pdf` ..... thesis text in PDF