Self-Collecting Mutators are Self-Compacting

Stephanie Stroka

February 11, 2011

Chapter 1

Introduction

An average program often allocates temporary memory, so called short-term memory, which is discarded as the program proceeds. If we plot the amount of memory allocated on the heap against the execution time of the program, we will observe that it is growing and shrinking continuously. The time axis must therefore not represent the processor or environmental time, but logical points in the program where memory is deallocated.
Figure 1.1 shows the memory consumption of a simple program, which allocated temporary memory during loop-runs. Other more complex programs will show periodic memory consumption if the periods are abstracted by multiple graphs as sketched in Figure 1.2. These examples demonstrate correct programs without memory leaks.
[5] analyzed the net memory consumption of the GNU C compiler (version 2.5.1) and the results showed a similar periodic behavior in memory consumption (bottom line of Figure 1.3). Some allocators lose the periodic behavior when we have a look at the gross memory consumption. The top line of Figure 1.3, for example, represents the gross memory consumption of GCC 2.5.1, which used obstack[1] allocations. We see that there is an enormous difference between the

[1]The obstack allocator creates a stack for each allocation with an undefined size

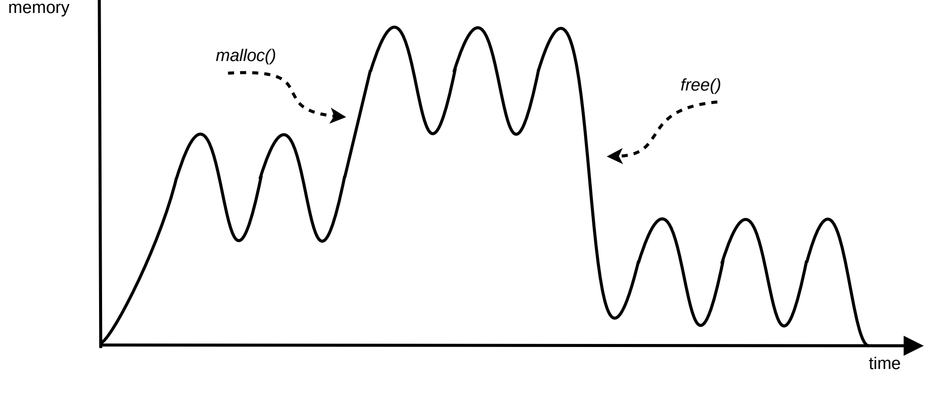

Figure 1.1: Periodic memory consumption

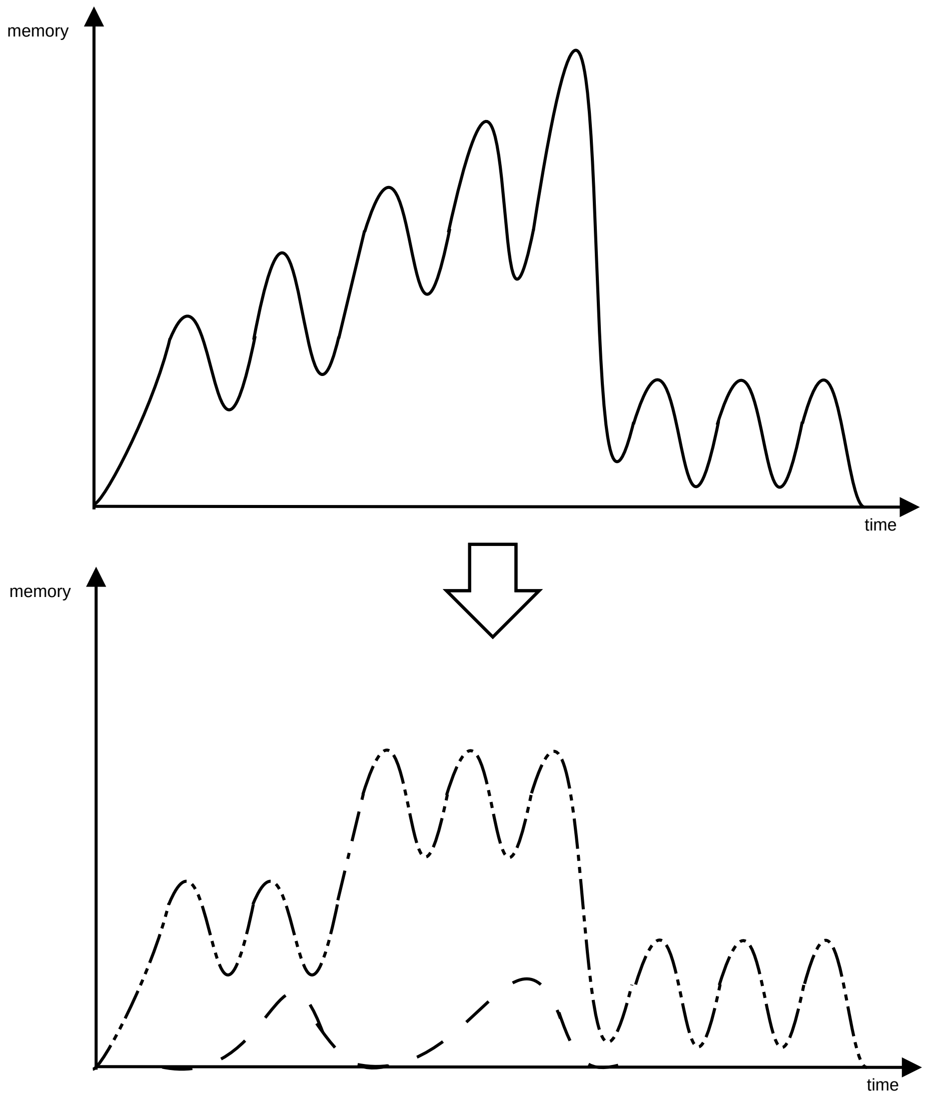

Figure 1.2: Periodic memory consumption abstracted with multiple graphs

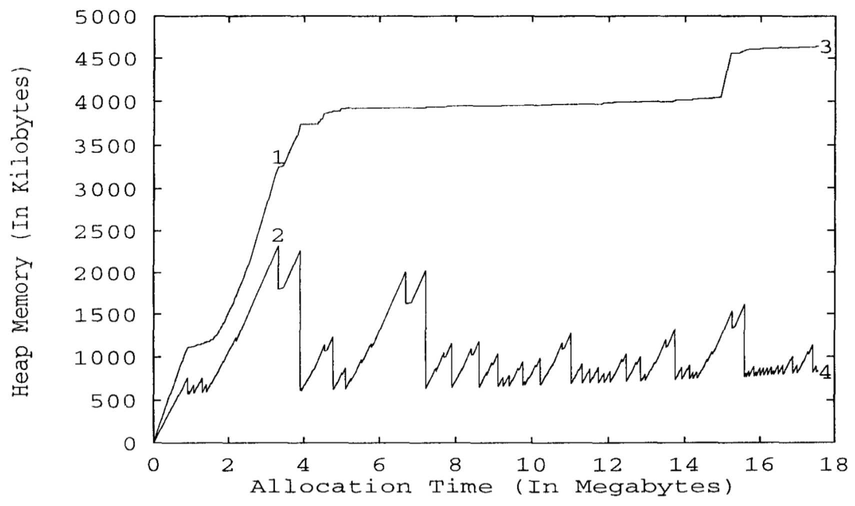

Figure 1.3: Periodic memory consumption in GCC 2.5.1

memory needed and the memory allocated for that program.
To find an allocator that has good behavior in memory consumption (which would be if the gross memory slightly differs from the net memory), I propose to have a closer look on the memory periods that exist in most programs. With the knowledge of knowing which objects are short-living and which are long-living, even a simple sequential fit allocator might give optimal results. Figure 1.4 shows how a sequentially ordered memory block can provide optimal memory consumption if the life-time is known.
Automatically knowing the life-time of objects, however, is not a trivial task. I therefore concentrated on the short-term memory model [1] developed by the Computational Systems Group of Prof. Christoph Kirsch, which enables programmatically defined life-time information and logical clocks for memory objects on the heap. The intention is to build an allocator that is able to re-introduce the periodic behavior in gross memory consumption, if necessary by abstracting the periods with multiple clocks.
The goal of the *Self-Collecting Mutators are Self-Compacting* project is to provide a memory management system that allows to analyze the periodic memory consumption, and as a consequence also the fragmentation, of programs using a time-aware memory model: The short-term memory model. The two implementations - short-term memory with compact-fit and short-term memory with regions - use alternative allocators in despite of the original implementation with `ptmalloc2`. I decided to use compact-fit as an allocator which is able to bind memory fragmentation by performing compaction if the fragmentation exceeds the limit. The implementation attempts to provide information about how many compactions are necessary for different periods. For a multi-clock implementation, I used a region-based memory allocator. A clock is then bound to a region. The region-based implementation aims to prevent massive fragmentation, as memory with similar life-time is allocated into the same region.

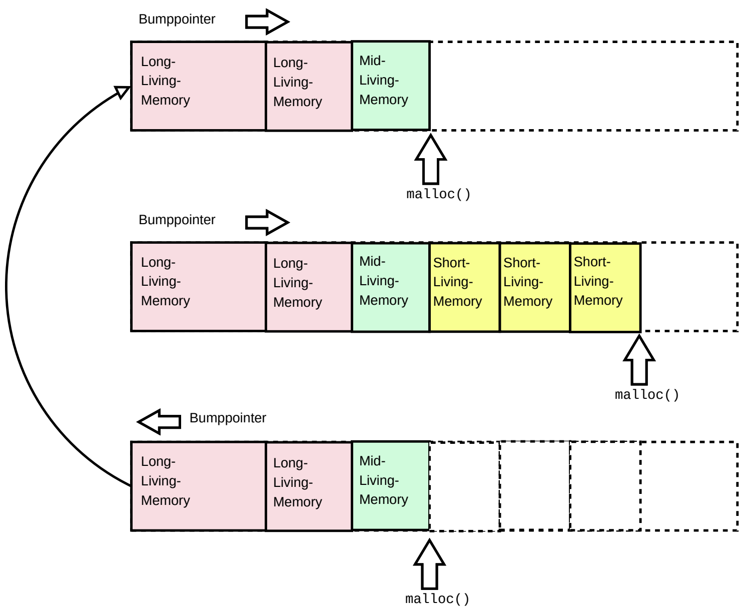

Figure 1.4: Optimal sequential fit allocation

Chapter 2

Implementation

In the short-term memory model, objects on the heap expire after a finite amount of time. Thus, like in a human brain, the oblivion is an automated process if the memory is not refreshed when time advances. Short-term memory that is continuously refreshed, on the other hand, evolves into long-term memory. Time advancing and refreshing of memory can be done explicitly by the programmer by applying so-called object life-time approximation.

During allocation, a one word header is reserved for a short-term memory object. This header includes the descriptor counter which counts the amount of refreshing calls that have been executed on the object. When an object is refreshed, a descriptor, which points to the refreshed object, is added to the short-term memory model and the specific descriptor counter is incremented. The descriptor is stored in a descriptor page, which is accessible via a thread-local descriptor buffer, as depicted in Figure 2.1.

A tick call removes a descriptor and decrements the descriptor counter. If the descriptor counter is then equal to zero, the object is removed.

In the following sections the implementations for short-term memory with compact-fit and short-term memory with regions are described in more detail. This includes the characteristics of both memory allocators, their design and their complexity constraints. Furthermore, the integration into the short-term memory library *libscm* is presented.

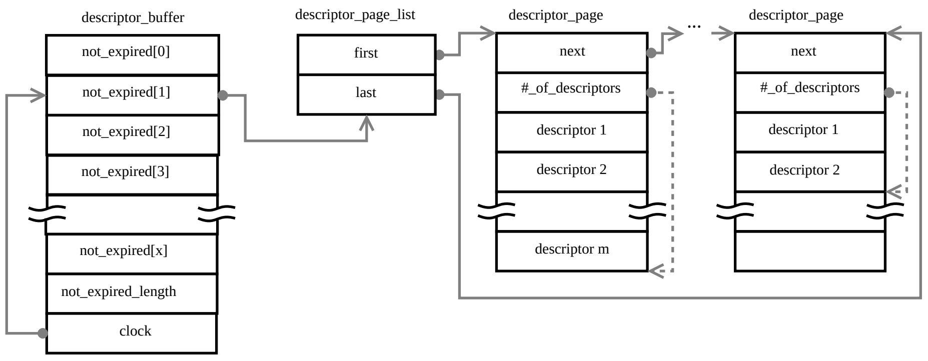

Figure 2.1: The descriptor buffer

2.1 Short-term memory with Compact-fit

Compact-fit is a compacting memory management system for allocating, deallocating, and accessing memory in real time which provides predictable memory fragmentation and response times [4] [3]. There exist two compact-fit implementations: Moving and non-moving, which have equal complexity constraints but differ in the way that non-moving is faster, because it does not move the actual objects on the physical memory. The non-moving variant adds another layer of abstraction by placing a block table between abstract addresses and physical addresses, which is then used for compaction. For simplicity I used the moving variant, but non-moving should work as well.
Compact-fit organizes its memory into pages and page-blocks as size-classes. The main idea is to allow partially fragmented pages and bind it to a constant `K`. If the amount of fragmented pages is greater than `K`, the memory is compacted by moving blocks from the end of the size-class into the fragmented pages, as illustrated in Figure 2.2.
To hide the physical memory address from the user, compact-fit manages objects through an abstract address. Thus, in despite of the typical `ptmalloc2`, which is the allocator used in the GNU C library, the compact-fit allocator returns an abstract address that the user must dereference first:

```
void** cf_malloc(size_t size);
```

The user must therefore be aware of any additional object header that has been added to the allocated memory. In my implementation, the short-term memory object header is added during allocation, which means that dereferencing is accomplished with the following macro which considers the object header:

```
#define DEREF(x) (PAYLOAD_OFFSET(*x))
```

In the following the library calls which are necessary to manage short-term memory with compact-fit are presented.

void scm_malloc(size_t size):** To integrate the compact-fit allocator into *libscm* a level of abstraction for `cf_malloc` has been added. This additional level wraps the initialization of the object header around the compact-fit malloc call.

void scm_refresh(void aaddr, unsigned int extension):** The `scm_refresh` call has also been adapted to allow abstract addresses as a parameter. Since the descriptor counter is located at the physical memory, the `refresh` function is responsible for dereferencing the abstract address to be able to increment the descriptor counter. The descriptor, however, points to the abstract address as illustrated in Figure 2.3. This is necessary to assure correctness after memory compaction.

void scm_tick(void);: The tick call slightly differs from the original implementation of *libscm* as the handling of abstract addresses during the collection of descriptors has to be considered. The expired descriptors are added to the beginning of the expired descriptor list which is then collected all at once or incrementally, depending on the compiler flags. Objects with a descriptor counter of 0 will then be deallocated.

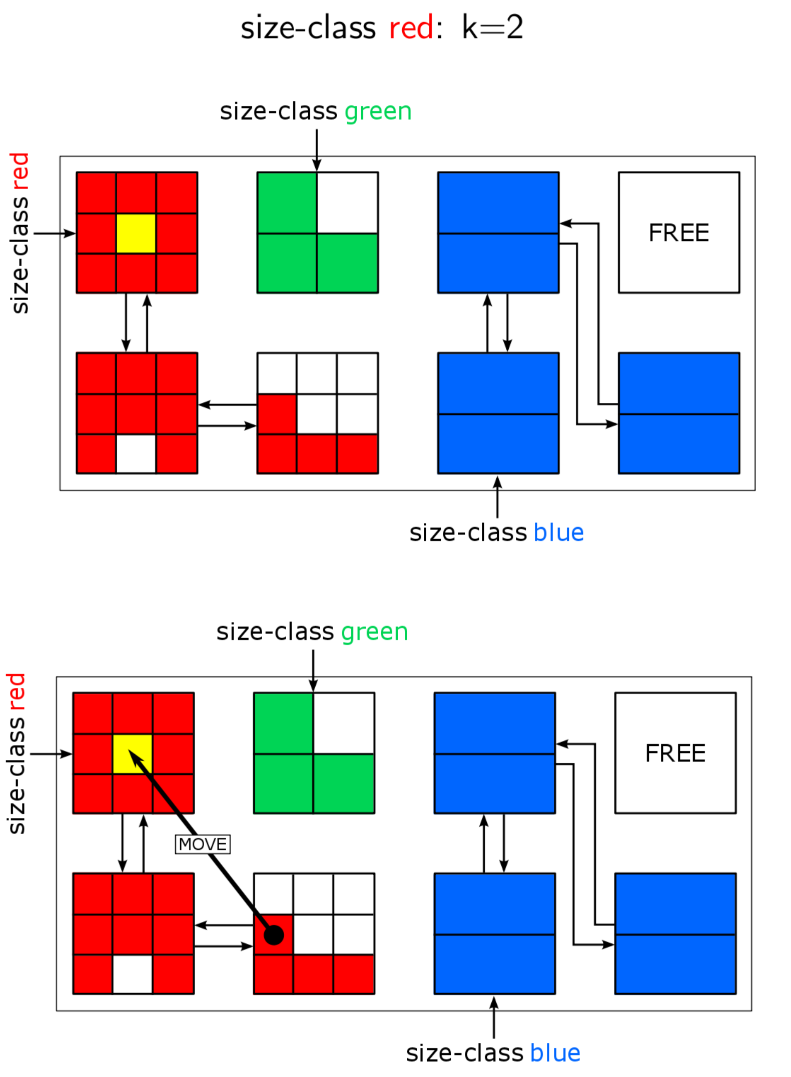

Figure 2.2: The process of compaction for a partial compaction bound of $\kappa = 2$

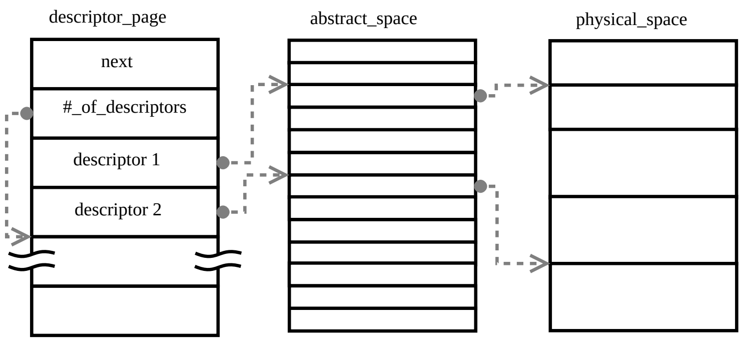

Figure 2.3: Descriptor page containing abstract addresses

2.2 Short-term memory with Regions

For the region-based approach, the short-term memory implementation has been adapted to support multiple clocks and descriptors for regions. Figure 2.4 presents the new configuration for the descriptor buffers. For each clock initialization, a new descriptor buffer is created and a default region is identified with the clock. Instead of an `scm_malloc()` and `scm_refresh()` call, region-based short-term memory is rather assigned to a clock during allocation. Refreshing is still possible, but optional, and allows the assignment of multiple clocks for one region. Thus, instead of managing descriptors and descriptor counters for each object, every allocation and refresh of an object results in a new descriptor for the region in which the object exists.
The regions design is illustrated in Figure 2.5. Regions are implemented as lists of pages, where each page contains a pointer to the next page, the overall memory and the size of used memory, which allows constant sequential fit allocation. If a memory request can not be satisfied by an existing page, a new page is created and the unused page memory of the last page is declared as fragmentation. An optimization would be to sort the pages for the biggest amount of fragmentation, so that a memory request is always applied on the page with the smallest size of used memory. Furthermore, the region desgin allows constant region-page-deallocation, or in other word linear region-deallocation in the amount of region-pages, if the descriptor counter is 0.
In the following I define a number of library calls which are necessary to manage the memory with the region-based short-term memory model.

void init_clock(pthread_key_t* clock_key): The programmer is responsible for initializing the needed clocks. The clock-key will be stored at the `pthread_key_t` address, given as a function parameter. The clocks should have different phases and frequencies. Although it would not be a program error, multiple clocks with the same phase and frequency would result in bad space and time consumption.

void* scm_malloc_clock(size_t size, pthread_key_t* clock_key): For region-based self-collecting mutators, memory must be assigned to a clock, which represents a region. This function takes the clock-key address initialized before and the size of the object in bytes as parameters and returns a pointer to the usable, allocated memory.

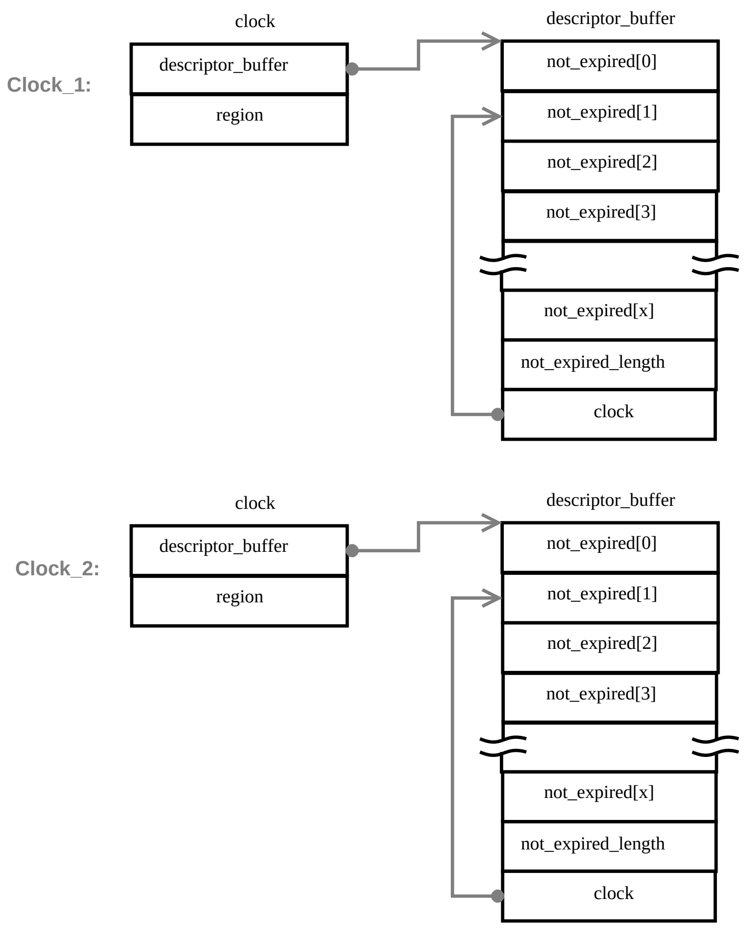

Figure 2.4: Multiple descriptor buffers for multiple clocks

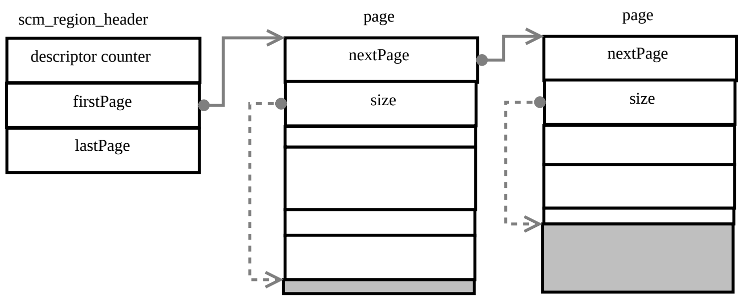

Figure 2.5: The region design: Header design and sequential fit allocations.

void scm_refresh_clock(void* ptr, unsigned int extension, pthread_key_t* clock_key): Refreshing with a certain expiration extension and/or another clock extends the life-time of a memory object. The expiration extension value defines the amount of tick calls of a certain clock by which the memory is still needed. An expiration extension of n, therefore, results in deallocation at the $(n+1)$-th tick call. Refreshing with an expiration extension of 0 is done implicitly when memory is allocated to a clock. Memory can also be refreshed with an additional clock, which results in a life-time extension equal to the maximum life-time of both clock-regions. An explicit refresh call results in the expiration extension of all objects in the same region.

void scm_tick_clock(pthread_key_t* clock_key): A tick call results in the deallocation of all regions assigned to a clock that have an expiration extension of 0 and in a decrement of all other expiration extensions $n > 0$. The collection of the regions is either done all at once (*eager*) or incremental (*lazy*), which means that the deallocation of a region is at least delayed by $(d-1)$ further `scm_refresh_clock` and `scm_tick_clock` calls, where d is the amount of memory objects allocated in one region.

Chapter 3

Tests

3.1 Short-term memory with Compact-fit

Both implementations have been tested on simple scenarios. A representative benchmark remains future work. The short-term memory with compact-fit implementation has been tested on a simple program that allocates long-term, mid-term and short-term memory in loops, as depicted in Figure 3.1. The tick calls have been incrementally added in three executions, starting with 1) one tick call at the end of the program, 2) an additional tick call at the end of the outter loop and 3) in the third run, four tick calls, additionally at the end of the inner loops. The result of the test showed that the runtime increases with O(1) which is caused by the overhead of additional refresh, tick and free calls and with O(n) if compaction is necessary. For the case that no compaction is necessary, that is for $\kappa = 3$, the maximum number of used pages for the amount of tick calls is listed in Table 3.1.
Another interesting result showed that, for a certain number of κ, the number of compaction increased (Table 3.1). This phenomenon happens, because compaction causes compaction, as Figure 3.2 and 3.3 illustrate. In Figure 3.2 the partial compaction bound κ has been set to 1 and the first page that is deallocated is the first that has been allocated. Note that the first page is not a

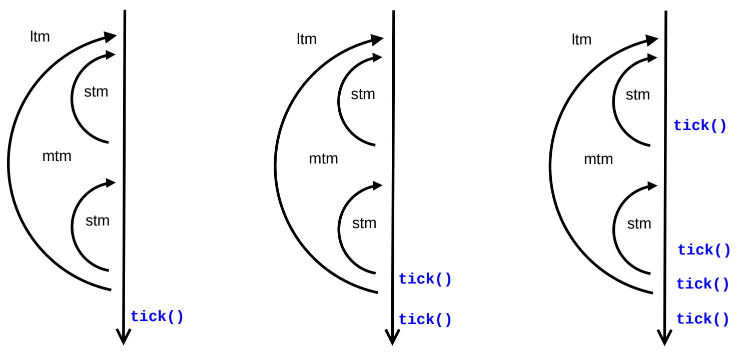

Figure 3.1: The test-scenario for short-term memory with compact-fit.
ltm: long-term memory, mtm: mid-term memory, stm: short-term memory

# TICKS	USED PAGES
1	52
2	5
4	3

Table 3.1: More ticks result in less used pages

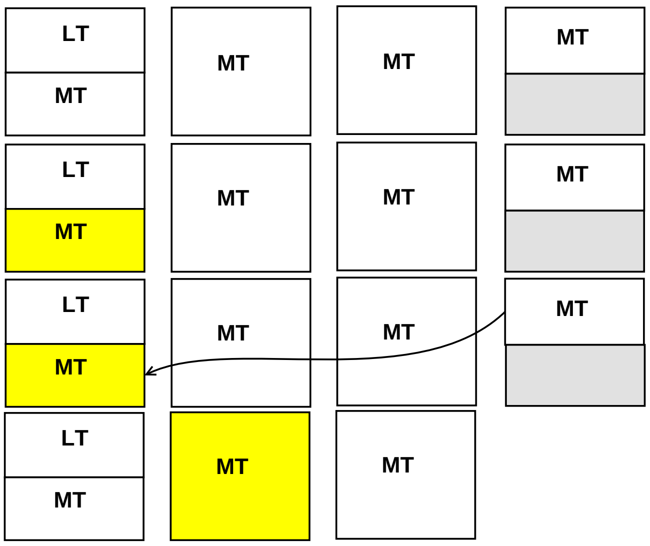

Figure 3.2: Compaction with K = 1: The second page

pure middle-living memory. The first deallocation series results in a half-full first page, where it is necessary to move memory from the last page. After the movement, the second page is completely deallocated, as is the third page. Therefore, no further compaction is necessary, because the memory in the first page will be deallocated at last.
In Figure 3.3, the partial compaction bound has been set to 2 and therefore compaction starts at the second page. This compaction causes a domino effect in other compactions, which would be unnecessary in that case. A possible solution, which intuitively comes to the mind when we reconsider Figure 1.4, is that we deallocate memory in a LIFO manner.

K	COMPACTION
1	160
2	520
3	0
4	...

Table 3.2: Compaction may cause more compaction

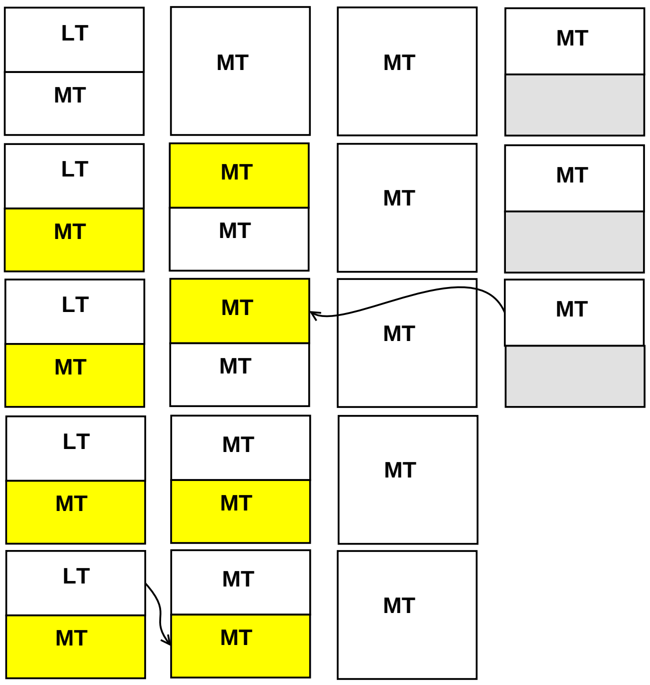

Figure 3.3: Compaction with $\kappa = 2$: The domino effect

3.2 Short-term memory with Regions

The short-term memory with regions implementation has been tested with two clocks which tick with different frequencies. The first clock-region allocates 30 x 512bytes in a loop that runs 10 times. It ticks every time at the end of the loop. The seconds clock allocates the same memory, but only every second time. It also ticks at the end of every second loop-iteration. Thus, the second clock runs with half of the frequency of the first clock. If *libscm* has been compiled to support eager collection, the difference between needed and allocated memory is low. Figure 3.4 shows the net memory consumption for this simple example. Figure 3.5 displays the gross memory consumption. The steps in the allocated memory results from the allocation of region pages. If lazy collection is on, the descriptors are incrementally removed. The deallocation of a region is then delayed by a constant *d*, where *d* is the amount of descriptors that point to a region. Since every allocated object creates a new descriptor for the region, *d* is at least as big as the amount of objects in a region.

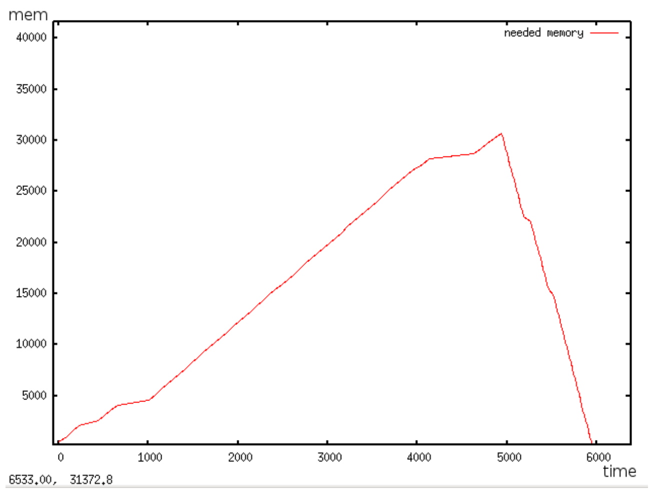

Figure 3.4: Needed Memory for the short-term memory with regions test example

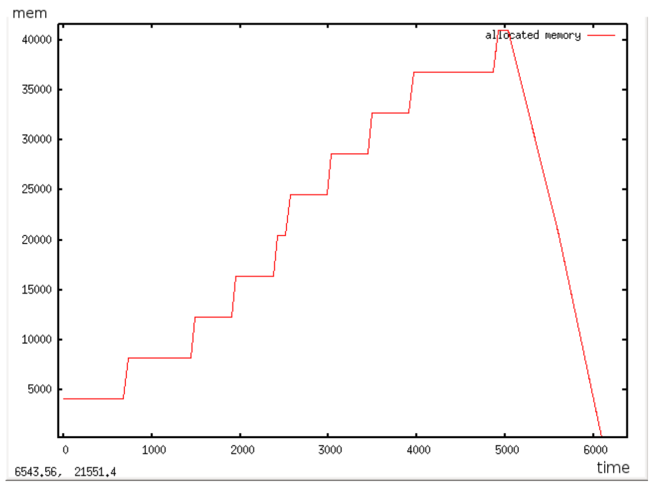

Figure 3.5: Allocated Memory for the short-term memory with regions test example. Eager collection is on

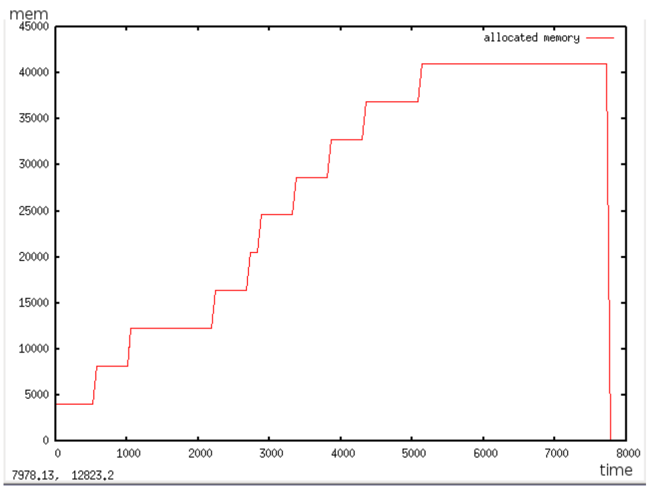

Figure 3.6: Allocated Memory for the short-term memory with regions test example. Eager collection is off

Chapter 4

Conclusion

The *self-collecting mutators are self-compacting* project provides a basis for further memory consumption analysis. Two implementations, short-term memory with compact-fit and short-term memory with regions, have been presented as variants of the existing self-collecting mutator library *libscm*.
Short-term memory with compact-fit can be used as a time- and space-predicting explicit dynamic memory management, although it is often more efficient to disclaim the compaction process for a better run-time behavior, especially to prevent the compaction domino-effect.
Short-term memory with regions use region-based memory management by binding clocks to regions. Thereby we have seen that it is possible to preserve the periodic memory behavior while still achieving fast deallocation, in our case O(n), n = number of region-pages.
Future work includes the implementation and interpretation of representative benchmarks and the formalization of the coherence between program periods and fragmentation for both implementations. Furthermore, it would be interesting to analyze memory consumption with a short-term memory implementation for the famous REAPS allocator [2].

Bibliography

[1] M. Aigner, A. Haas, C.M. Kirsch, and A. Sokolova. Short-term memory for self-collecting mutators - revised version. Technical Report 2010-06, Department of Computer Sciences, University of Salzburg, October 2010.

[2] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 1–12, New York, NY, USA, 2002. ACM.

[3] S.S. Craciunas, C.M. Kirsch, H. Payer, H. Röck, and A. Sokolova. Concurrency and scalability versus fragmentation and compaction with compact-fit. Technical Report 2009-02, Department of Computer Sciences, University of Salzburg, April 2009.

[4] S.S. Craciunas, C.M. Kirsch, H. Payer, A. Sokolova, H. Stadler, and R. Staudinger. A compacting real-time memory management system. In *Proc. USENIX Annual Technical Conference*, 2008.

[5] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: solved? In *Proceedings of the 1st international symposium on Memory management*, ISMM '98, pages 26–36, New York, NY, USA, 1998. ACM.