Short-term Memory for the C Programming Language

MASTER'S THESIS

to obtain the Master's degree
at the Faculty of Natural Sciences of the
University of Salzburg

submitted by
Martin Aigner, B.Eng.

Academic supervisor
Univ.-Prof. Dr.-Ing. Dipl.-Inform. Christoph M. Kirsch

Department of
Computer Sciences

Salzburg, August 2012

Acknowledgments

Thanks to Professor Christoph Kirsch and his group for guidance and motivation. I had a great time working on my masters thesis and you inspired me to give my best.

My special thanks goes to:

Ingrid and Manfred for roots and for wings.

To Christine, after all, for so many things.

To my family and friends, who helped me getting through,

I wouldn't have made it, without anyone of you!

Abstract

Short-term memory is a memory model for dynamic heap management [1]. Unlike the traditional persistent memory model implemented in explicit or implicit memory management systems, objects allocated in short-term memory expire after a finite amount of time or may be refreshed to extend their lifetime. We propose a concurrent, incremental, and non-moving implementation of short-term memory for the C programming language called self-collecting mutators which relies on explicit refreshing information and utilizes an explicit dynamic memory allocator. We present an extensive performance evaluation of our implementation using a new benchmark tool called ACDC. Based on size and lifetime characteristics of dynamic objects in allocation-intensive C programs, ACDC models a representative workload for dynamic memory allocators. We empirically verify a small and constant per-object time and space overhead of our implementation and show competitive performance to the persistent heap management model.

Keywords: Short-term Memory, Dynamic Heap Management, Multi-threaded Allocator Benchmark

Contents

1. Introduction

Dynamic heap management systems provide an interface to receive (allocate) and return (free) memory objects at runtime. There are two types of interfaces, namely explicit interfaces and implicit interfaces. Both types receive objects through an allocation routine. They differ in the way of returning memory objects. Explicit memory management requires the programmer to explicitly return an object to the management system. This is fast but error prone because it relies on information provided by the programmer which may be incorrect. Especially in multi-threaded applications concurrent reasoning about the last use of an object is a non-trivial problem. Implicit memory management systems, also known as garbage collectors, solve this problem at the expense of runtime overhead and increased complexity. Also, languages that lack type safety, e.g. the C programming language, increase the difficulty to implement implicit memory management. Both approaches guarantee dynamic objects to be persistent for the time between receiving and returning the object. Therefore, we call the model where objects are explicitly or implicitly returned the persistent memory model.

Short-term memory [1] is a memory model where objects expire after a finite amount of time which makes it unnecessary to return dynamic objects either explicitly or implicitly. In this work we present a concurrent, incremental, and non-moving implementation of short-term memory called self-collecting mutators followed by an extensive performance analysis.

> "There is as yet no standard suite of benchmarks for evaluating multi-threaded allocators." [2]

This statement from Berger et al. has its origin in the year 2000 and is still valid today. Most recent performance evaluations of dynamic memory allocators are based on simple simulations of allocations patters, e.g., the Larson benchmark [3], or using allocation traces. We present a new benchmark tool designed to create a representative workload

based on lifetime and size characterisitics of dynamic objects observed in allocation intensive applications in [4] and [5] and extend these characteristics to multi-threaded allocation patterns including shared objects.

1.1. Contributions

The main contribution of this work is an efficient and scalable, concurrent implementation of self-collecting mutators for the C programming language. Furthermore we contribute a multi-threaded memory allocator benchmark tool called ACDC which models a typical allocator workload. We use ACDC to perform a detailed performance evaluation of our implementation of short-term memory.

1.2. Outline of the Thesis

The thesis starts with an overview of process memory concepts, i.e., the way a process organizes its address space. After that we describe the short-term memory model and present the implementation self-collecting mutators. Then we introduce ACDC, a multi-threaded allocator benchmark tool that we use for the final part of the thesis; an extensive performance evaluation of our implementation.

Chapter 1, Introduction: We first present te problem and give an outline of the thesis.

Chapter 2, Process Memory Concepts: In this chapter we given an overview of existing techniques to manage dynamic memory objects and discuss the notion of liveness of such objects.

Chapter 3, Short-term Memory Model: Here we present the short-term memory model for managing dynamic objects. We explain the meaning of expiration dates in the context of thread-local and multi-threaded time management.

Chapter 4, Self-collecting Mutators in C: In this chapter we describe our concurrent implementation of short-term memory in C. We define the memory management routines and the data structures that enable full incrementality and constant-time operations.

Chapter 5, ACDC Benchmark: The ACDC benchmark tool models a representative allocator workload. In this chapter we describe the characteristics of dynamic memory objects and the implementation of ACDC.

Chapter 6, Experimental Evaluation: We perform an extensive performance evaluation of our implementation of short-term memory. We use ACDC to generate configurable workloads for self-collecting mutators to identify the important factors and quantify their impact on performance.

Chapter 7, Conclusion: We conclude the thesis in the last chapter and discuss future work.

2. Process Memory Concepts

On systems that use the GNU C library (GLIBC) [6], processes allocate memory in two major ways: by exec and programmatically [7]. Exec is the operation of creating a virtual address space, loading the program into it and executing the program. The virtual address space is logically divided into segments, i.e., independent and contiguous subsets of virtual addresses [8]. Three important segments are the text segment, the data segment and the stack segment. When the program file (executable) is loaded, exec allocates the necessary space inside the newly created address space to store the data from the executable including the program instructions and static or global variables. This data is stored in the text segment. Declaring a static or global variable in a C program is called static memory allocation. The necessary space is allocated once as part of the exec operation and is never freed until the termination of the process. After the process starts to execute, it uses programmatic allocation to get additional memory. A C program that uses GLIBC can perform programmatic allocation in two ways: automatically and dynamically. [7]

Automatic memory allocation happens when the programmer declares automatic variables, e.g. a function argument or a local variable. The space for an automatic variable is allocated, when the compound statement that contains the variable declaration is entered, and is freed, when that compound statement is exited. Automatic variables are usually allocated on the process' stack which resides in the stack segment. The stack segment grows as the stack grows, but it does not shrink as the stack shrinks. [7]

In case the amount of memory that is needed by a program depends on factors that are not known before the program runs, static or automatic memory allocation cannot be used since they can only rely on information known before execution. The technique that allows a process to gain and manage additional memory at runtime is called dynamic memory allocation. Note that the C language [9] by itself only supports static and automatic allocation. Dynamic allocation requires support by the operating system and

a runtime system like GLIBC. The dynamically allocated memory usually resides in the data segment. The data segment can be preallocated by exec and the process can extend and shrink it using system calls like `brk` [10]. However, a more common way to dynamically allocate memory at runtime is through library functions that we describe next.

In this work we focus on the GNU C library (GLIBC), which is the de-facto standard C library for most systems based on the Linux kernel [6]. Furthermore we limit the discussion on C to the ISO/IEC 9899:1999 standard [11] informally known as C99.

2.1. Explicit Dynamic Memory Management

Dynamic memory management is performed in C using library functions usually called `malloc` and `free` [12]. This functions are implemented in the C standard library. Listing 2.1 gives an example of dynamic memory allocation in a C program. Any Unix-like operating system distribution provides a C library that implements an interface to system calls to simplify and standardize basic operations like input/output and also memory management. The implementation of the memory management functionality is often called the allocator. It consists of algorithms and data structures to manage the address space of the data segment. In memory management terminology, this address space including the management metadata is called the heap.

```
1  #include <stdlib.h>
2  int main(int argc, char **argv) {
3          struct a {
4                  int val;
5                  char c;
6          };
7          struct a* a_ptr = (struct a*)malloc(sizeof(struct a));
8          a_ptr->val = 5;  /* access through pointer only */
9          a_ptr->c = 'a';  /* access through pointer only */
10         free(a_ptr);
11         return 0;
12 }
```

Listing 2.1: Example of a C program that uses dynamic memory through the standard C library.

Whenever a process demands additional memory it can call malloc to retrieve a pointer to a contiguous block of memory on the heap. This block is said to be dynamically allocated by the process and it can be accessed through the pointer that malloc returned. A pointer is the only way to refer to dynamically allocated memory. When the process calls malloc again to acquire another block, it is guaranteed that those two blocks do not overlap, i.e., those two sets of contiguous addresses are disjoint.

When a block of dynamically allocated memory is no longer needed by the process, it can be deallocated by calling free. After that, an access to this memory is considered to be a runtime error and the pointer to this memory region is called a dangling pointer. Memory, that is no longer needed, but not freed, is called a memory leak. This causes the overall memory consumption to be higher than necessary. The programmer must take care that every object (dynamically allocated piece of memory) is deallocated when it is no longer used and that no dangling pointer is dereferenced anymore. This must be done explicitly by the programmer and is therefore called explicit dynamic memory management.

The GLIBC implements additional functions like realloc to resize objects and calloc to allocate zero-initialized arrays. However, since both realloc and calloc can be expressed using malloc and free, we focus only on this functions for explicit dynamic memory management.

2.1.1. Sources of Errors

The fact that explicit memory management relies on the correct use by the programmer, makes it prone to errors. We now take a closer look at how memory management errors can negatively affect the execution of a program.

Premature free or dangling pointer is a programming error where a pointer to an object is referenced after it was freed. Depending on the implementation details of the memory management system this error can result in different scenarios. Either, it is possible that the program crashes if the object's address is no longer part of the data segment or that dereferencing the dangling pointer causes unpredictable behavior, if a new object took the place in the mean time. [13]

Missing and deferred free calls are a problem that commonly occurs in systems that use explicit dynamic allocation. For the programmer it is easy to know where to place a malloc call. This can be done right before the memory is used. In contrast, it is often hard to know when an object is not needed anymore. The correct placement of a free call is right after the last use of an object. However, the last use of an object may depend on conditions that are dynamic and therefore the correct place of deallocation is on different code paths. When programs grow complex, the task of finding the last use of an object is a challenging task and therefore error prone. Free calls, that are not placed right after the last use, temporarily increase the memory consumption of the process. While deferred deallocation can still be tolerated to simplify the code, a missing free call causes the corresponding memory to be lost until the program terminates. Such a memory leak can be a serious problem for long-running processes such as tasks running on servers or control systems. [13]

Invalid frees occur when an illegal address is passed to the free call. Free expects a pointer that was previously returned by malloc. C allows a technique called pointer arithmetic, where the arbitrary manipulation of a pointer is possible. Although programmers may find this a powerful tool, its use is highly error prone. For example, it allows a very simple way to iterate over the addresses in a block of memory that was allocated via malloc. However, doing so yields the risk of programming errors, for example by passing a wrong pointer to the corresponding free call. A special case of an invalid free is called double free. Here, a pointer to a memory object is passed to the free call more than once. The consequences are fatal again. Double free can cause

the process to crash or it unpredictably changes the behavior of the allocator because allocators have to keep track about every object that was freed by the process. This can be implemented by a data structure called free list. For efficiency reasons, the free list can be stored in the very same location where the freed object was stored before. When a free call is performed on the same location again (or an invalid location), it is possible to overwrite parts of the free list and therefore corrupt the heap management metadata. [13]

Violating block boundaries is also caused by the fact that C allows writing to arbitrarily forged addresses. A block of memory obtained by malloc has a start address (the pointer returned by malloc) and a fixed size. Similar to the free list, the allocator also keeps meta data for allocated objects like the size of the object. A straight forward way of implementation is reserving some space for this information in front of the object. This object header contains information that is important for the allocator, e.g., for later deallocation. The pointer that is returned to the process is the address right after the object header. This defines a lower and an upper bound for addresses that the process may access. It is easy to see that writing to an address right after that boundary may corrupt the object header. Similarly, writing to an address before the boundary may destroy the object header or contents of a subsequent object. A memory access outside the bounds of objects may even direct to an address outside of the data segment. This so-called segmentation fault is well known to every C programmer and causes the operating system to terminate the process. [13]

To sum up there are two problems with explicit dynamic memory management in C. Firstly, pointers can be altered in an unsafe way, which is very error prone. Secondly, the correct placement of free calls is a challenging task. Especially in multi-threaded and complex applications the use of dynamically allocated data structures becomes very tricky. Both types of bugs are hard to track down because they can vary depending on runtime parameters e.g. input data or execution time. Tools like Valgrind [14] help to find memory errors during development, but their introduced runtime overhead makes them unattractive for production systems.

In this work, we do not intend to question certain features of the C programming language. Altering pointers can be a powerful tool, if used correctly and it can be dangerous if done carelessly. Since we are primarily interested in the use of memory management functionality, we only discuss the problem of misplaced free calls.

2.2. Garbage Collection

In the previous section we have discussed the problems that occur when the responsibility for handling dynamic memory is burdened to the programmer. The biggest problems for programmers when developing software systems is complexity and one of the best ways to deal with complexity is abstraction. Abstraction is already in place for the management of static variables and automatic variables in most programming languages including C. The programmer does not have to take care where to allocate global variables and how to set up stack frames. High-level programming languages tend to make also use of an abstraction for heap objects. The technique that enables this abstraction is called garbage collection (GC) [15]. Garbage collectors keep track of all objects allocated on the heap and regularly determine which objects are no longer needed and can therefore be freed.

Runtime systems for the C programming language usually do not implement garbage collectors because of the unsafe nature of the language. Nevertheless, such systems exist [13] and therefore we introduce the basic concepts of garbage collection after introducing some important general terms of memory management.

Roots of computation: The values that can be directly manipulated by a program are the content of the registers, the automatic variables on the stack and the global variables. These values form the roots of a computation. Dynamically allocated memory objects are only accessible through pointers that are stored in the roots or by following a chain of pointers that start at the roots.

Pointer reachability: An object that can be referenced by either a pointer in the roots or by a chain of pointers is said to be reachable. If it is not, then we call such an object unreachable. Note that in languages like C every address inside the heap segment is always reachable by the manipulation of pointers. However, we restrict the discussion on reachability to a safe use of a language, i.e., no direct manipulation of pointers.

Object liveness: An object on the heap is live, if it's address is held in a root, or there is a pointer to it in another live object. Another way to informally define object liveness is pointer reachability starting at the roots.

We now describe a simple formalism to combine these terms and properties. Let *Roots* be the set of references in the roots of computation and *Objects* the set of heap objects. Let $\hookrightarrow$ be the "points-to" relation defined as follows: for every $M \in Roots \cup Objects$ and every heap object $N \in Objects$, $M \hookrightarrow N \iff M$ holds a reference to N. [15]

Intuitively, the set of live objects consists of all heap objects that maintain pointer reachability starting at the roots, i.e., an object $O \in Objects$ is life if and only if $\exists n \in \mathbb{N}$, $\exists R \in Roots$ and $\exists M_i \in Objects, 0 < i \leq n$ such that

$$R \hookrightarrow M_1 \hookrightarrow M_2 \hookrightarrow \ldots \hookrightarrow M_n \hookrightarrow O$$

The set of all live objects is therefore described by the transitive closure of the points-to relation $\hookrightarrow$ starting at the roots, i.e., the least set *Live* where

$$Live = \{O \in Objects : (\exists R \in Roots : R \hookrightarrow O) \vee (\exists M \in Live : M \hookrightarrow O)\}$$

Note that this definition of the set of live objects is a conservative over-approximation of all objects that are potentially accessible. It contains objects that have references to them but might never be used by the program anymore. [15]

Garbage collection includes the process of determining an object's liveness. This can be done either directly or indirectly. Direct methods require the objects to keep track of all the references that point to it. Indirect methods make use of the references to other objects that are stored in the roots or live objects. The most common direct method is reference counting and the most important indirect method is called tracing. These two methods are fundamental algorithms for storage reclamation and are actually algorithmic duals of each other. [16]

2.2.1. Reference Counting Collectors

We have already stated that most allocators maintain additional metadata for each object on the heap. The reference counting algorithm requires an extra field per object called reference count. This is a positive integer starting from zero that gives the number of references that point to this object. Such references can be stored either in the roots of a program or in other heap objects.

Let *refs* be a mapping $refs : Objects \times Objects \to \mathbb{N}_0$ and $refs(O_i, O_j) \mapsto$ the number of references from O_i to O_j. The reference count of an object is given by the mapping $rc : Objects \to \mathbb{N}_0$ where

$$rc(O) \mapsto |\{R : R \in Roots \wedge R \hookrightarrow O\}| + \sum_{O_i \in Objects} refs(O_i, O)$$

gives the number of references to O from the roots and other objects including itself. [15]

Initially the reference count of an unused chunk of memory is zero. When an object is allocated from e.g. a free list, the reference count is set to one because the first reference to the object is created. When such a reference is copied to another reference or the reference is passed as a function parameter and is thereby copied on the stack, the reference count of the object needs to be incremented. When a function returns, the stack frame becomes invalid and so do all references to heap objects stored in this stack frame. For each of this objects the reference count must be decremented. Similarly, when a pointer to an object is overwritten the reference count of the previous pointer target must be decremented. When the reference count of an object becomes zero this means that no pointers to this object remain. There is no legal way for the program to re-establish a reference to this object and so the object is no longer accessible. The object can be deallocated or returned to the free list. In the same step, all objects that are referenced from within this object need to have their reference count decremented accordingly. Note that this can cause a chain reaction if for example the last reference to a singly linked list is deleted. Deallocating the head node decrements the reference count of the next node and so on.

Incrementing and decrementing the reference count of an object can be done explicitly by the programmer or implicitly by code that was generated by the compiler. The latter approach is usually taken by languages that directly support garbage collection through reference counting.

Reference counting indirectly computes the set *Live* that we have defined above. It assumes that every object is live until a reference count of zero proves the opposite. In this way it is guaranteed that $Live \subseteq \{O \in Objects : rc(O) > 0\}$ because we defined *Live* to be the set of all reachable objects. A Object O where $rc(O) = 0$ is not reachable hence it is not live.

2.2.2. Tracing Collectors

Tracing garbage collection was the first algorithm for automatic storage reclamation [17]. Under this scheme dead objects are not reclaimed immediately but remain unreachable and are collected when a certain event takes place, e.g. a memory limit is exhausted or the collection is triggered explicitly. At this point the process is suspended and the garbage collection routine takes over.

The garbage collection routine consists of two phases. The mark phase and the sweep phase. Tracing GCs are therefore often called mark-sweep collectors. Starting from the roots, the mark phase performs a traversal of all objects that are reachable and hence are life by definition. All live objects are marked by setting a bit in the object header. The sweep phase performs the actual cleanup. The collector iterates over the whole set of objects and the non-marked objects are freed. The live objects survive and their mark bit is removed for the next collection cycle.

Tracing collectors directly compute the *Live* set defined above. They assume every object to be garbage until the mark phase proves the opposite. Starting from an empty set, $Live = \emptyset$, all Objects are marked as live that are reachable from *Roots*:

$$Live = Live \cup \{O \in Objects : \exists R \in Roots : R \hookrightarrow O\}$$

In a recursive step, all objects are marked that are reachable from objects that are already marked as live:

$$Live = Live \cup \{O \in Objects : \exists M \in Live : M \hookrightarrow O\}$$

The recursion terminates when the set *Live* reaches a fixed point. [15]

2.2.3. Comparison of Reference Counting and Tracing Collectors

A strength of reference counting GC is the fact that the memory management overhead is distributed throughout the computation [15]. The management of live and garbage objects is interleaved with the execution of the application logic. This contrasts with

(non-incremental) tracing algorithms in which the user thread is suspended while the garbage collector performs its work. This makes reference counting schemes useful for applications where a predictable response time is needed such as reactive and real-time systems.

A dynamic property of many typical programs is that most objects are short-lived [4]. Reference counting schemes can help to exploit that by reusing objects immediately after they have become unreachable. This can have a positive impact on data locality and cache performance. In tracing schemes dead objects remain unused until the next mark-sweep stage is finished. Note that this also increases the amount of memory that needs to be reserved for an application that uses a tracing GC.

Another disadvantage of reference counting is the high cost that is required to maintain the reference count invariant. Each time a pointer is overwritten, the reference count of both the old and the now target object needs to be updated. In tracing schemes, pointer updates have no additional costs associated with it.

Reference counting techniques also introduce a per-object space overhead for the reference count which can be as large as a pointer. In practice however, reference counts will not become this large and smaller fields can be used in combination with an overflow strategy. [15]

The major drawback of reference counting algorithms is their inability to collect cyclic structures. A doubly linked list with only two nodes is already such a cyclic structure and they are used very often. Consequently, reference counting collectors need some help to deal with cyclic structures. One way to deal with cycles is to combine a reference counter with a tracing scheme. This already is an example of a hybrid based on the two fundamental concepts tracing and reference counting [16].

The major drawback of basic tracing schemes is their stop-the-world nature. During a collection run the mutator is not allowed to change the state of the roots and the heap objects. Incremental tracing collectors can reduce the pause time of the mutator but they introduce a lot of complexity in the collection algorithm to deal with the changing mutator state, especially if predictable pause times must be guaranteed [18] [19].

2.2.4. Conservative Garbage Collection

Although not being the origin of garbage collection, object oriented languages often use built-in automatic storage reclamation. The type system of such languages support garbage collection because the layout of objects is known at runtime. The C programming language does not provide runtime information about the used data structures. Furthermore there are no guarantees that static types are used as intended at runtime. In C the programmer can alter pointers and write arbitrary data to arbitrary memory locations. A garbage collector will have little information about where roots are to be found, the stack frame layout, which memory locations store pointers and which to not. To be practically usable and compatible with existing C programs, a garbage collector for C will neither get support by the compiler nor the runtime system. Garbage collectors that have to operate under such restrictions are called conservative collectors.

The Boehm-Demers-Weiser collector is a fully conservative garbage collector that does not require any assistance from a compiler or runtime system [20]. A program using this garbage collector logically uses two distinct heaps. One heap for explicitly managed objects and another one for automatically managed objects. Under the restriction that objects from one heap do not point to objects on the other one, the collector can be used alongside standard libraries and traditional C code using malloc and free.

First of all, a conservative GC has to find the roots. They can be found in registers, static areas of the program and the stack. Although it is possible to access this locations, doing so heavily depends on the architecture of the target machine. This implies that a conservative GC must provide support for a given architecture. Finding the bottom of the stack requires knowledge of the runtime system or using heuristics like checking the address of a local variable in the main routine. [20]

Identifying a pointer is the second main problem that needs to be solved. Misidentification of a pointer might cause the referenced data to be recycled as garbage. This creates dangling pointers that can crash the program. On the other hand, if the collector is too conservative, it risks retaining too much garbage and crashing the program, because it runs out of space. Consequently, the collector needs to take some care to avoid misidentification. An object is only marked if the pointer passes each of three tests. Firstly, the potential pointer must refer to the heap. Secondly, the heap block that contains this object has been allocated. For this constraint the heap is organized into blocks, that

contain objects of the same size. The third constraint checks if the offset of the object in the block is a multiple of the object size for this block. Only if the pointer passes these tests, the corresponding object is treated as automatically managed memory using a tracing scheme. [20]

The problems with conservative garbage collection can be mitigated by relaxing the assumptions about the missing cooperation by the compiler and the runtime system. For example in [21], Boehm et al. suggest a modified C compiler that omits optimizations which could break the garbage collection and that also checks the source code for garbage-collector-safety.

2.2.5. Drawbacks of Garbage Collection

A garbage collector solves the problem of classical memory bugs: dangling pointers and memory leaks. However, GCs introduce their own problems and also does not solve the memory leak problem completely. If a data structure is no longer used by a mutator, but at some place a reference to this data structure remains, the garbage collector will never recycle this memory. This problem is called live memory leak because from the point of view of the collector this data structure is live whereas from the point of view of the mutator it is not. [15]

A second general problem that comes with garbage collection is the resource penalty. Automatic garbage collection takes both time and space. The computation of reachability always takes time and the necessary meta data always takes space. Mark-sweep collectors also suffer from an increased overall memory usage because of the deferred collection phase. This are the main costs that must be paid for simpler programs and to avoid memory bugs. [15]

Finally, a garbage collector adds additional complexity to the system. Modern hybrids of tracing and reference counting schemes are very complex and building them correctly is a difficult task. In addition to the complexity, a garbage collector increases the code size of an application. This problems often discourage the use of garbage collection in a restricted environment like embedded and real-time systems.

2.3. The Persistent Memory Model

So far we have discussed the advantages and disadvantages of explicit memory management and garbage collection. In this section we look at the similarities of this traditional approaches.

An ideal heap management system partitions the set of heap objects in two disjoint sets of objects. One set contains the objects that are needed by the mutator in the future and the other set contains the objects that are not needed anymore. We call them the needed set of objects and the not-needed set of objects, respectively [1]. Heap management is correct if the needed set of objects is always maintained. A trivial implementation of correct heap management is bump pointer allocation without freeing objects. However, such a behavior is not desirable because a general mutator will run out of space and crash. Therefore we require a heap management to be bounded, that is, the not-needed set of objects is always eventually reclaimed by deallocation or reuse.

Explicit heap management (if used correctly) and garbage collectors implement different approximations of the needed and not-needed set of objects. Explicit memory management using malloc and free under-approximates the not-needed set of objects by explicitly deallocating a subset of the not-needed set, i.e., by calling free on objects that are no longer needed by the mutator. Reference counting garbage collectors also under-approximate the not-needed set by reclaiming objects that are not referenced by the mutator anymore and can therefore no longer be needed. On the other hand, tracing collectors over-approximate the needed set of objects in the mark-phase where the set of reachable objects is calculated. Reachable objects are potentially needed by the mutator, i.e., the needed set of objects is a subset of the reachable objects. [1]

The common property of explicit heap management and garbage collection is that an allocated object is guaranteed to be persistent in memory until it is either freed explicitly or reclaimed implicitly by non-reachability. We call this the persistent memory model. Objects in the needed set are maintained without further attention whereas objects in the not-needed set require action, either through explicit freeing or through garbage collection. [1]

The problems with malloc/free and garbage collection can be described very well using this memory model. Missing free calls increase the gap between the not-needed set and its approximation. This is what we usually call a memory leak. A premature free call

operates on the needed set of objects and violates the correctness requirement for heap management. This results in a dangling pointer. Additionally, garbage collectors allow for a potentially unbounded gap between the needed or not-needed set and its approximation that is calculated by either tracing or reference counting. This phenomenon is known as reachable memory leak. [1]

2.4. Summary

In this chapter we explored the traditional ways of heap management using malloc/free and garbage collection. We discussed the advantages and disadvantages of this techniques and how the differences between them affect the way a process may use dynamic memory. From the similarities of explicit memory management and garbage collection we derived the persistent memory model and showed that the problems with both memory management systems directly draw the conclusion from the model.

3. Short-term Memory Model

The already discussed memory model deals with objects that are persistent from the moment they are allocated until the moment they are freed. We call this model the persistent memory model. In this section we describe a memory model where objects are not persistent until further notice, but expire after a finite amount of time. This memory model we call the short-term memory model [1].

In short-term memory, every allocated object has an attached expiration date and the mutator has a notion of time, a clock for example. When the clock reaches the expiration date of an object, the object is said to be expired. An expired object will eventually be reclaimed by a memory management that implements short-term memory. When a mutator needs an object beyond its expiration date, the object can be refreshed before it expires. In contrast to the persistent memory model, here the needed set of objects needs attention by refreshing objects whereas the not-needed set does not require any action taken by the mutator. [1]

The short-term memory model introduces two new disjoint sets of objects. The not-expired set of objects and the complementary expired set of objects. Note that this two sets only exist if time is guaranteed to advance. Otherwise all objects will never expire, i.e., they are permanent. Like the persistent memory model, short-term memory over-approximates the needed set of objects by the not-expired set of objects. However, instead of reachability or explicit freeing, short-term memory uses the expiration date of the objects and the speed of the time advances to control the approximation. Figure 3.1 illustrates the logical partitioning of the heap and the approximation of the needed set. [1]

Heap management is correct, based on the short-term memory model, if the needed set is a subset of the not-expired set of objects. It is bounded if the expired set always eventually contains the not-needed set and time advances. Again, if time does not

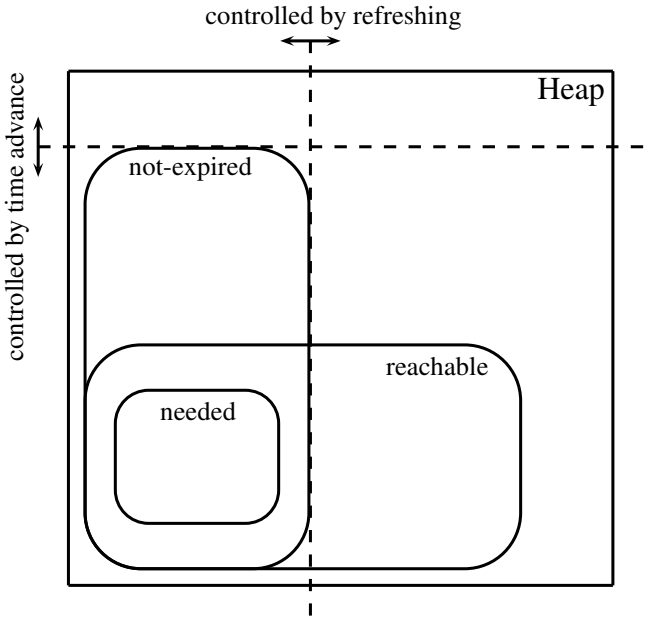

Figure 3.1.: Logical partitioning of the heap and the approximation of the needed set by the short-term memory model

advance the heap will grow without bounds like using bump pointer allocation without deallocation in the persistent memory model. [1]

Controlling the expiration dates of the objects and the advance of time can be done in different ways. The marking phase of a tracing collector can be used to refresh all reachable objects and the sweeping phase advances time. Another approach is to let the programmer explicitly provide refreshing information and explicitly advance time in the program. Unlike explicit memory management using the persistent memory model the programmer does not need to know exactly where the last use of an object is, but instead she gives an upper bound on the lifetime of an object. [1]

To use an analogy, short-term memory works like a library, where one can pick up books and bring them back after a while. Short-term memory provides a function to acquire an object, another function updates the expiration date. Above all, we have to introduce a notion of time to realize this concept.

In the remainder of this chapter we focus on a short-term memory model based on explicit refreshing. We describe how single and multiple mutator threads deal with short-term memory objects and we discuss the implications of shared objects and how to handle multiple expiration dates.

3.1. Single-threaded Model

Short-time memory is designed to be used together with the persistent memory model. Programmers use the malloc call to allocate dynamic memory. An implementation of short-term memory intercepts malloc and adds the extra space for the expiration extension to the chunk of memory demanded by the program. This enables full backwards-compatibility with existing mutators.

At any time after the allocation of an object it can be transformed to a short-term memory object by calling a refresh(o, e) routine where o is a pointer to the object and e is the so-called expiration extension, $e \geq 0$. The refresh call adds a new expiration date $(l+e)$ to the object, where l is the current value of a clock controlled by the calling thread. The clock is an integer counter and its value is held in a variable local to the thread. From this moment on, the object is managed by the short-term memory system. The object is guaranteed to exist until the clock of the thread is increased to $(l+e+1)$. After this, the object is said to be expired and will eventually be reclaimed by the short-term memory management. The clock advances according to an explicit call of a tick function in the program which increases the clock from l to $l+1$. Consequently, the object expires after $(e+1)$ tick calls unless another refresh(o, e') call is performed on the object. [1]

Figure 3.2 illustrates the life-cycle of persistent and short-term memory objects on the heap. In the first step, objects a, b and c are allocated using malloc. Then, after some time, objects b and c are refreshed with an expiration extension of 1 and 0, respectively. Now, objects b and c are short-term memory objects while object a is still persistent. The clock of the mutator thread is 0 at this moment. The state of the clock is shown on the time axis. Object b receives an expiration date of (0+1)=1 and object c gets an expiration data of (0+0)=0. A tick call increases the clock to 1 and causes object c to expire. A second tick call increases the state of the clock to 2 which expires object b. Note that object c no longer exists in the rightmost picture because the memory management system can remove an expired object at any time.

A thread can also perform multiple refresh calls on the same object in between of two tick calls. Every refresh call creates a new expiration date for the object. However, the expiration extensions do not accumulate but the largest expiration extension determines the effective expiration date of this object. The only side-effect is wasting CPU power

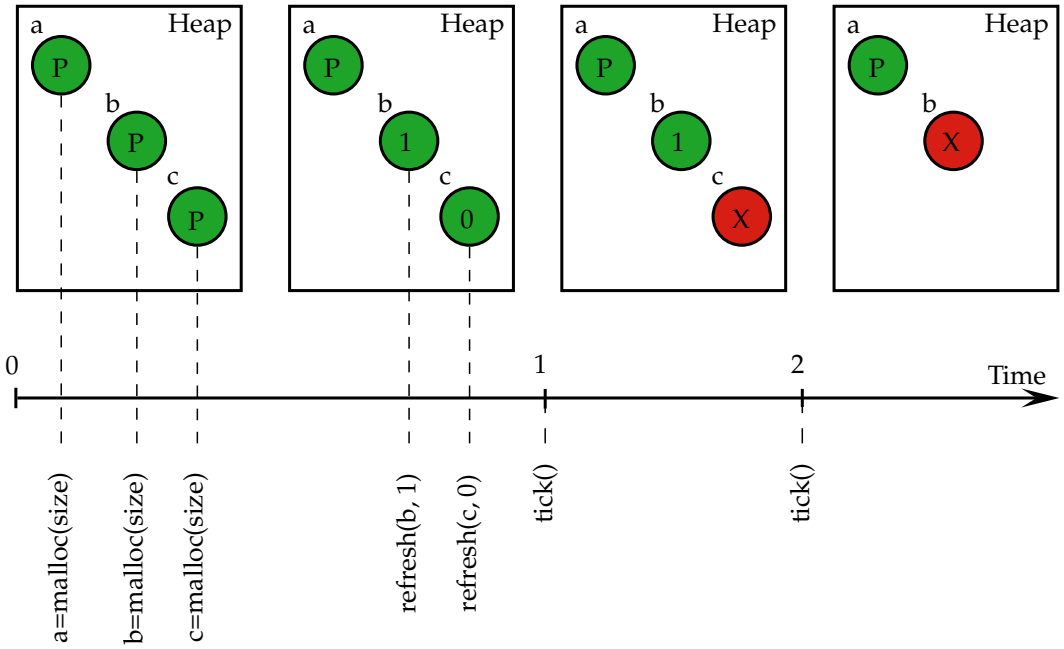

Figure 3.2.: Persistent and short-term objects of a single mutator thread

and memory for creating and storing the expiration dates that have no effect. [1]

3.2. Multi-threaded Model

In multi-threaded applications, it is possible that an object is shared by different threads and therefore it can be refreshed by more than one thread. An object can thereby have multiple expiration dates from multiple threads that are evaluated with respect to the different clocks of the threads. The expiration semantics of the single-threaded model therefore needs to be extended, however it remains still simple. According to our definition, an object in short-term memory expires, when all its expiration dates have expired. An expiration date has expired, if its value is less than the value of the clock of the thread that created the expiration date through refreshing. [1]

Refreshing an object by multiple threads has two consequences. First of all, the object is shared between the refreshing threads, i.e., every thread that wants to refresh an object needs a reference to it. Second, the refreshing of the object needs to be synchronized. Otherwise it is possible that a thread refreshes an object that has already expired. In other words, every thread needs enough time to refresh an object. This can be done

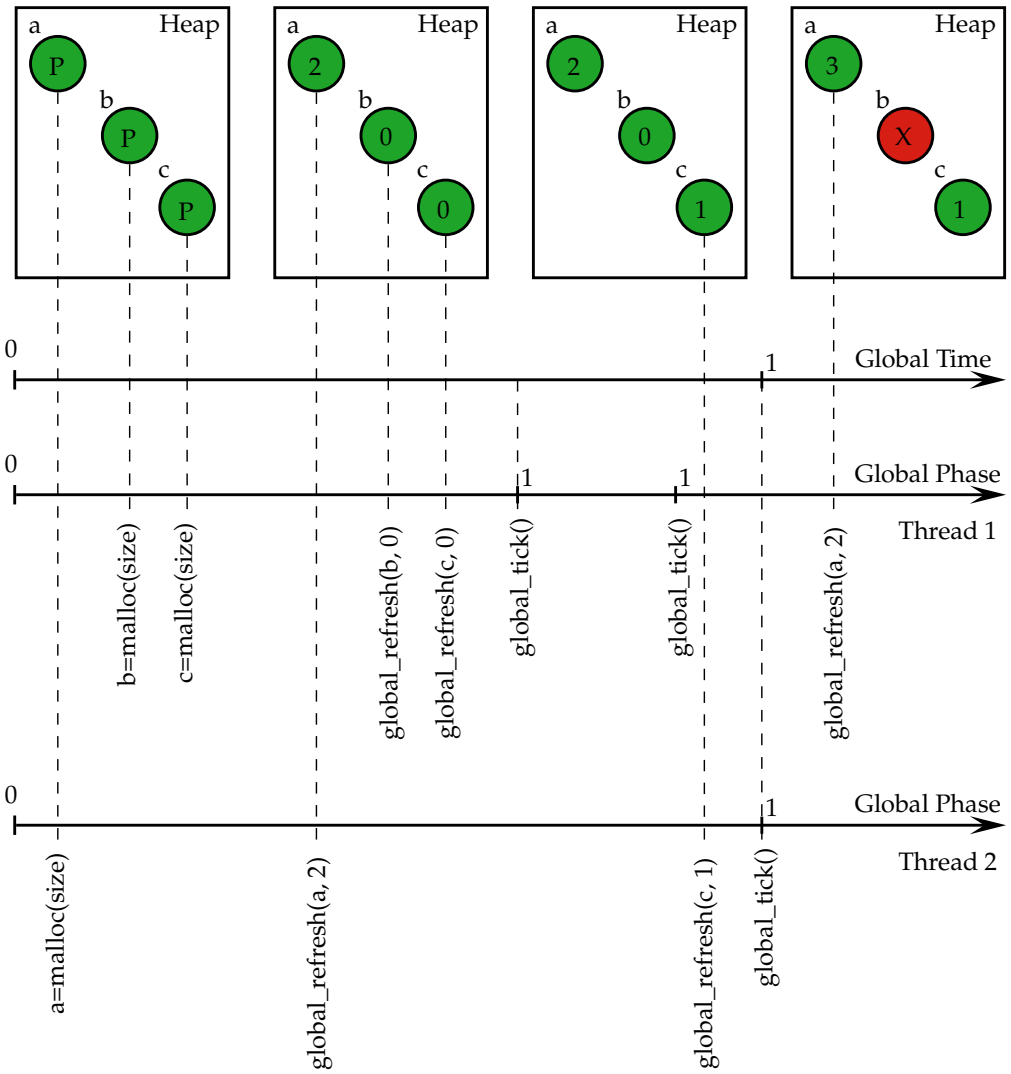

Figure 3.3.: Refreshing and expiration of short-term objects shared by two threads

either explicit by the application through synchronization mechanisms or implicit by the short-term memory management system. For short-term memory we bring up the notion of a global time to enable the expiration of shared objects. In this section we describe a mechanism for global-time management. [1]

3.2.1. Global-Time Management

In the single-threaded model of short-term memory, each thread stores its own expiration dates and evaluates them with respect to its own notion of time. Shared objects require

expiration dates to be created and evaluated with respect to a thread-global notion of time. We therefore introduce global_refresh to create expiration dates for shared objects and a global_tick routine that expires such objects according to a synchronized notion of global time. Like the refresh call for local objects, global_refresh(o, e) takes a reference to an object o and an expiration extension e as an argument. The expiration date for a shared object o is $(g + e)$, where g is the global time. The object is guaranteed to exist until the global time is advanced to $(g + e + 1)$. [1]

Global time is an integer counter visible to all threads. An expiration date created through global_refresh has expired if its value is less than global time. Global time advances in a synchronized fashion after all participating threads issued a global_tick call at least once. Intuitively, every thread needs enough time to create an expiration date for a shared object it wants to use in the future. For this, we define a thread-local variable called global phase which determines if the thread performed a global_tick since the last global-time advance. We also define a ticked threads counter which gives the number of threads that have ticked at least once since the last global time advance. The last thread, that has called global_tick, triggers the advance of global time and resets the ticked threads counter. This event also marks the end of the current global phase and the start of the next one. [1]

Figure 3.3 illustrates a heap of three objects that are shared among two threads. The global time counter starts at zero as well as the global phase counters in each thread. Thread 2 allocates object a and thread 1 allocates objects b and c. We assume that these three pointers are stored at a place that is accessible to both threads, e.g., global variables. After some time thread 2 sets an expiration extension of 2 on object a by calling global_refresh(a, 2). The expiration date is created with respect to the global time, i.e., $(g + e) = (0 + 2) = 2$. Thread 1 sets the expiration extension of 0 to objects b and c. Their expiration date results in $(g + e) = (0 + 0) = 0$. Thread 1 is the first thread to perform a global_tick. This increases the global phase of thread 1. Note, that the global time is not affected, since thread 2 has not ticked yet. A subsequent call to global_tick by thread 1 has no effect on the global time either.

As we stated above, global time is advanced only after all threads had a chance to create their own expiration dates for shared objects. Thread 2 uses this chance to set an expiration extension of 1 on object c. Since the global time is still 0, this results in an expiration date of $(g + e) = (0 + 1) = 1$ for object c. After thread 2 is done with

setting expiration extensions for all objects it is interested in, it calls global_tick. Note that thread 2 does not refresh object b, which means it does not want to use it. Thread 2 is the last thread that had to tick in global phase 0. Now, all participating threads did a global_tick and the global time is increased to 1. From this moment on object b is expired because its expiration date ($e = 0$) is less than the global time $g = 1$.

Based on this global time scheme the programmer does not need to provide any synchronization commands to ensure that objects do not expire too early. For example, if the programmer would use the local refresh and tick calls instead, thread 2 would refresh an expired object c. This is considered a programming error. Global-time management solves this problem for the programmer at the expense of higher memory consumption. In the example in Figure 3.3 object b is not used after thread 1 called global_tick. However, it remains on the heap until thread 2 also called global_tick.

3.3. Sources of Errors

Providing explicit expiration information runs the risk of being incomplete or even wrong. The same problem applies to explicit malloc and free in the persistent memory model. Missing free calls in the persistent memory model lead to memory leaks, i.e., unused objects not being freed. In short-term memory the memory usage can grow out of bounds, if time does not advance, i.e., tick calls are missing. On the other hand, missing refresh calls have the same effect as premature free calls in the persistent memory model and both result in dangling pointers. We believe that it is easier to reason about which objects are sill needed in the future than reasoning about which objects are not needed anymore. Furthermore, redundant refresh calls in short term memory have no effect (other than wasting CPU time and memory), while redundant free calls in the persistent memory model cause a runtime error. [1]

3.4. Summary

We have introduced the short-term memory model where the lifetime of an object is controlled by setting its expiration date which is evaluated with respect to a thread local clock if the object is local and with respect to a global clock if the object is shared.

When the expiration date is less than the value of the responsible clock, we say the object has expired and an implementation of short-term memory may reclaim it.

We have also presented a global-time management scheme that guarantees that multiple threads have the chance to refresh shared objects. The programmer does not need to synchronize the short-term memory management calls at the expense of additional memory consumption. However, if space efficiency is an issue, the programmer is free to take care of the synchronization by herself.

4. Self-collecting Mutators in C

In this chapter we present an efficient and concurrent implementation of short-term memory called self-collecting mutators (SCM) [1]. In self-collecting mutators the programmer provides explicit refresh and tick information. This information is passed to the short-term memory system through function calls that we define in the application programming interface (API). We call our implementation LIBSCM since it is implemented as dynamically loadable shared library for the C programming language [9].

In this chapter we describe the most important design decisions for the library and the technical details to achieve them. We define the API that is visible to the user and describe its functionality. The design of the data structures used in LIBSCM is of particular interest because they enable an efficient implementation of the self-collecting mutators algorithm. Finally, we expose some extra features of LIBSCM that can be used to debug or profile applications that use short-term memory.

4.1. Design Decisions

The primary goals in the development of LIBSCM was to achieve minimal and predictable overhead for all API calls. In this section, we describe the implementation details that enable for constant time and space overhead per operation even for multi-threaded use cases.

The main trade-off in short-term memory is to simplify the concurrent reasoning about the needed set of objects at the expense of memory consumption. Nevertheless, an important design constraint was space efficiency of the data structures that we use to maintain management information. We explain the most important data structures and the optimizations we performed on them. Furthermore we do not want to maintain extra threads for memory management. All actions are performed by mutators.

4.1.1. Backwards Compatibility

Another important goal to achieve with LIBSCM is backward compatibility, i.e., existing C programs using the persistent memory model run on top of LIBSCM without modifications. The reason is to allow iterative migration of an existing application to short-term memory. LIBSCM provides a thin layer between an application and the standard C library. A call to malloc or free is redirected through wrapper functions in LIBSCM. The malloc call is slightly changed to add one additional word of meta data to every object. This extra word has a meaning only if the object is later changed to short-term memory. We discuss this meaning in the next section. In the persistent case, this word has no functionality. The free call is also wrapped in LIBSCM. Here we check if an object is still in persistent mode and if so the deallocation function of the C standard library is called. The wrapping layer only imposes a very small and constant time overhead to the standard library calls. We use the GNU linker ld [22] and more specifically the `--wrap` option of ld to redirect calls to malloc and free from the standard library to the corresponding functions of LIBSCM.

4.1.2. Representation of Expiration Dates

The short-term memory model allows multiple expiration dates per object. At allocation time the number of expiration dates that this object will receive is unknown. Therefore, writing the expiration dates in an object header would require the header to grow. Increasing the size of a chunk of memory usually requires copying of its contents. This violates the constant runtime overhead requirement we have stated above.

The way we handle this problem is to inverse the information flow. Instead of giving the object the information about its expiration dates, we give every possible expiration date the information about which objects share this very expiration date. Figure 4.1 shows the difference of this two approaches. In (a), the different expiration dates are stored in header fields for each object. Note that, for example, adding a fourth expiration date to object A would require resizing of the whole object. In (b), we maintain a data structure for every possible expiration date (1 to 6). The example uses a list of pointers to the objects that have the corresponding expiration date. The list with the label 3 keeps pointers to the objects A and B. These are the objects that have the expiration date 3. The pointers in the lists that represent the expiration dates we call descriptors.

The header of each object now only consists of a descriptor count, i.e., the number of descriptors that point to this object. Note that the object has no information about its expiration dates.

In the example in Figure 4.1 we start at time 1. When the time advances to 2, all the descriptors in the list labeled with 1 are expired. When a descriptor expires, we simply delete it and decrement the descriptor count in the object the descriptor points to. As stated above, the object does not know which expiration dates are assigned to it. All it knows is that as long as the descriptor count is larger than zero, there still exist expiration dates that have not expired yet. When the descriptor count finally drops to zero, we know that all expiration dates represented by descriptors have expired and thereby the whole object has expired. In this way, we can keep a fixed size object header and maintain the expiration dates in data structures that have constant time access and can be optimized to efficiently store the descriptors. The implementation of this data structure is presented in section 4.3.3.

4.1.3. Threads

One of the main ideas on self-collecting mutators is that the threads that use short-term memory are responsible to perform the memory management through the LIBSCM API calls. Every action performed inside LIBSCM happens between the invocation and the return of an API function call.

The advantage of such an approach is reduced complexity. Garbage collectors usually run separate threads for tracing or cycle detection which requires synchronization between the workers and memory management threads. Another advantage in a time-sharing environment is that no context switches between the memory management routines and the actual work are necessary. This increases the predictability of the latency of memory management operations.

4.1.4. Allocator

Our implementation of a short-term memory system is closely coupled with a dynamic memory allocator. However, the main purpose of LIBSCM is not the allocation of objects but the management of expiration dates. Self-collecting mutators makes use

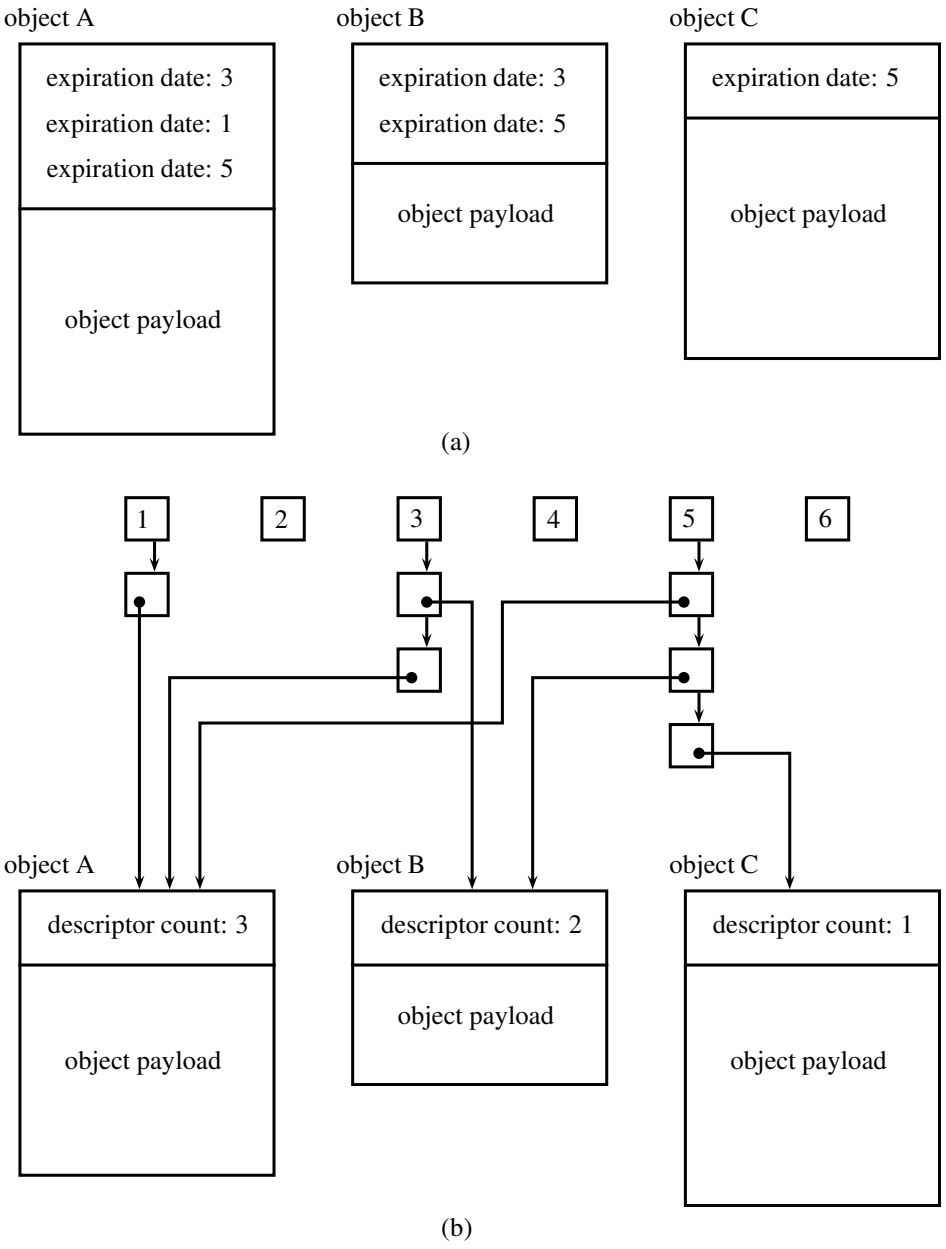

Figure 4.1.: Expiration dates stored in the object header (a) and using descriptors to represent multiple expiration dates of an object (b)

of the malloc and free call from the standard C library instead of implementing this functionality. We decided to use the GNU C library and build LIBSCM on top of the existing GLIBC allocator ptmalloc2 [7]. Still, it would be possible to change the default allocator to any other library that supports malloc and free in its API. An exploration on which existing allocator would best support the allocation pattern of typical short-term memory applications remains future work.

4.2. Operations of Self-collecting Mutators

An application can use short-term memory by calling the functions of LIBSCM that are defined in its public Application Programming Interface (API). The API of our implementation is defined closely to the abstraction that short-term memory suggests in Chapter 3. It defines calls for allocation, setting expiration extensions and time progression. In this section we describe the API of the implementation and give an overview of the semantics of the function calls.

In general, the design of the API also inteds to be used as easy and flexible as possible. The LIBSCM implementation can be compiled as a Dynamic Shared Object (DSO) and because of its backwards-compatibility it can be linked to existing code without re-compilation of this code (which might not be available). The implementation is based on a DSO best-practice guide described in [23].

4.2.1. Allocation

The LIBSCM allocation routine is called `scm_malloc`. The definition snipplet from the public header file stm.h is given in Listing 4.1. From the point of view of the programmer, the semantics of `scm_malloc` is very similar to `malloc` from the C standard library (defined in stdlib.h). The caller provides the size in bytes as unsigned integer and returns a pointer to the beginning of the usable address space. The content of this address space is undefined. The `size_t` type is the architecture independent representation of memory-related quantities between `0` and `SIZE_MAX` which is defined in stdint.h.

This function basically is all a programmer needs to allocate chunks of dynamic memory. However, the dynamic memory API from the C standard also defines the `calloc`,

`realloc`, and `free` call. They are described in the malloc man page [12]. Again, for backwards-compatibility we also implement wrappers for `calloc` and `realloc`. However, they are not part of the public API and only act as wrappers for `scm_malloc`. On the other hand, `scm_free` is part of the LIBSCM API. It is used to deallocate permanent objects, i.e., objects that never received an expiration date. Passing short-term memory objects to `scm_free` is a programming error. The only argument of `scm_free` is the pointer to the permanent object (obtained by `scm_malloc`). Listing 4.2 gives the definition of `scm_free`.

```
1 /*
2  * scm_malloc is used to allocate short term memory objects.
3  * This function can be used at compile time. Unmodified code
4  * which uses e.g. glibc's malloc can be used with linker
5  * option --wrap malloc
6  */
7 void *scm_malloc(size_t size);
```

Listing 4.1: Definition of scm_malloc from stm.h

```
1 /*
2  * scm_free is used to free short term memory objects with no
3  * descriptors on them e.g. permanent objects. This function can
4  * be used at compile time. Unmodified code which uses e.g. glibc's
5  * free can be used with linker option --wrap free
6  */
7 void scm_free(void *ptr);
```

Listing 4.2: Definition of scm_free from stm.h

4.2.2. Time Progress

The short-term memory model defines two kinds of clocks. Each thread that uses short-term memory objects needs to maintain a local notion of time to expire thread-local objects and a global notion of time to expire shared objects. In LIBSCM these clocks are implemented as integer variables (see Section 4.3.2) and the progress of one unit of time is mapped to incrementing this integer variables. The speed of time progression is controlled by the program through `tick` calls.

We distinguish two different tick calls depending on the clock we want to advance. The thread-local clock is controlled by the `scm_tick` call defined in stm.h. Listing 4.3 gives the prototype of this function which does not need any parameters.

```
1 /*
2  * scm_tick is used to advance the local time of the calling thread
3  */
4 void scm_tick(void);
```

Listing 4.3: Definition of scm_tick from stm.h

The semantics of `scm_tick` is very simple. It increases the thread-local time by one and, by this, expires all objects that have an expiration date which is smaller than the new local time.

Controlling the global time is a little bit trickier since it needs to reason about the state of all threads that participate in the short-term memory system. The API call however is still simple. The prototype of the function `scm_global_tick` is given in Listing 4.4.

```
1 /*
2  * scm_global_tick signals that the calling thread is ready to have
3  * the global time increased
4  */
5 void scm_global_tick(void);
```

Listing 4.4: Definition of scm_global_tick from stm.h

The semantics differs from the thread-local variant of time advance. Calling `scm_global_tick` does not directly increase the global time but raises a flag that the calling thread is willing to increase the global time. Only after all participating threads called `scm_global_tick` at least once the global time is actually advanced. As a consequence, two subsequent calls of `scm_global_tick` of a single thread executing without interference by other threads have the same effect as one single call. The last thread that calls `scm_global_tick` actually increments the global time.

The time interval between two global time increments we call a global phase. A thread calling `scm_global_tick` compares the global time with its thread-local global phase variable. If these two values differ it means that the thread called `scm_global_tick` for the first time in the current global phase. The thread then stores the global time in

its global phase variable and logically raises a flag. The flagging is implemented by decrementing a so-called ticked_threads_countdown. The last thread that calls `scm_global_tick` in the current global phase decrements the countdown to zero and triggers the global time advance. The countdown is reset to the number of participating threads and a new global phase begins. This expires all shared short-term memory objects that have an expiration date which is smaller than the new global time.

Global Time Advance for Blocking Threads

At some point in time a thread may block because of IO operations or synchronization requirements. The progress of the global time in self-collecting mutators depends on all threads. This means that a single blocking thread would stop any progress of global time.

We deal with this problem by changing the semantics of `scm_global_tick`. The computation of global time is restricted to not-blocking (or active) threads and the ticked_threads_countdown represents the number of active threads. For each thread we maintain a new clock called thread-global clock. This clock advances at the speed of the global time when a thread is active and stops when the thread is blocking. When a thread calls `scm_global_tick` for the first time in a global phase we increment the thread-global time. Note that the global time advances when the last thread globally ticks in this global phase. Therefore the global time and the thread-global time may be different but still advance with the same speed. [24]

When a thread blocks the number of active threads is decremented and the global time advances without involving the blocking thread. The thread-global time remains unchanged as long as the thread is blocked. When a thread has not yet globally ticked in the global phase it starts blocking we have to decrement the ticked_threads_countdown such that the other threads can proceed with their global time management. Therefore, when the thread resumes, it assumes that it already has globally ticked in the current global phase. This simplifies the management of the ticked_threads_countdown because it is not possible for the blocking thread to decrement this countdown twice in one global phase. Only the number of active threads needs to be incremented. The only exception is when a resuming thread is the only active thread in the system. Then it has to globally tick to make global time progress. [24]

Using the thread-global time instead of the global time to expire shared objects has one important consequence. Since the global time and the thread-global time advance with the same speed but may be out of sync, we need to take two thread-global time advances to guarantee that it contains one global time advance. Furthermore we need to add another time unit to the expiration extension of `scm_global_refresh` such that all threads have the chance to refresh a shared object withing one global phase. [24]

Runtime Complexity

The expiration dates of an object are represented by descriptors. As described in Figure 4.1, for every expiration date we maintain a set of descriptors. When time advances by one step this means that one such set of descriptors expires at the same time. For each descriptor, we have to decrement the descriptor count of the object it points to and if the descriptor count drops to zero, we have to free the object. It is easy to see that in this way the advance of time is not a constant time operation but linear to the size of the set of descriptors with the expired expiration date. The advantage, on the other hand, is that expired objects are reclaimed immediately after the time has advanced. We call this behavior of self-collecting mutators the *eager collection strategy* because expired objects are reclaimed as soon as possible.

To achieve a constant runtime behavior, the expiration process of the descriptors can be changed for incrementality. In this case, the set of descriptors that represents a just expired expiration date is moved to a set of descriptors that are already expired. This concatenation of lists can be done in constant time. Later, when the program calls a refresh or tick operation again, in addition to creating a new descriptor we also process one expired descriptor (or some other constant number of expired descriptors). This process we call *lazy collection strategy* because expired descriptors and objects are physically reclaimed some time after they have logically expired. This strategy enables a constant runtime bound for the tick operations at the expense of a higher memory consumption due to deferred reclamation. The refresh operations remains constant time since it only processes a constant number of descriptors.

4.2.3. Expiration Extensions

The LIBSCM API defines two different calls to set an expiration extension on an object. They are called `scm_refresh` and `scm_global_refresh`. The definitions from stm.h are shown in Listing 4.5. Both functions add the given expiration extension to the given object. The difference is that the resulting expiration date of the object will be expired based on the thread-local clock in case `scm_refresh` is called or based on the thread-global clock if `scm_global_refresh` is used. As a consequence, `scm_refresh` can be used to extend the expiration date of thread local objects and `scm_global_refresh` extends the expiration date of objects that are shared among other threads. The global time management described in Section 4.2.2 guarantees that other threads get the chance to globally refresh a shared object.

```
1  /*
2   * scm_global_refresh adds extension time units to the expiration
3   * date of ptr and takes care that all other threads have enough
4   * time to also call global_refresh(ptr, extension)
5   */
6  void scm_global_refresh(void *ptr, unsigned int extension);
7
8  /*
9   * scm_refresh is the same as scm_global_refresh but adds the
10  * expiration extension is evaluated using the thread local clock
11  */
12 void scm_refresh(void *ptr, unsigned int extension);
```

Listing 4.5: Definition of scm_refresh and scm_global_refresh from stm.h

The `scm_refresh` call with expiration extension e reads the current thread-local time l and adds a new expiration date $(l + e)$ to the object it is called on. This expiration date will be expired with respect to the thread-local clock. For shared objects, `scm_global_refresh` with expiration extension e takes the thread-global time g and adds $(g + e + 2)$ as a new expiration date for the corresponding object. The two extra time units are needed to support blocking threads and shared objects (see Section 4.2.2). This expiration date will be evaluated using the thread-global clock.

An object can have multiple expiration dates and each of them can be either locally and globally evaluated. The object is expired only if all expiration dates have expired.

Runtime complexity

When the program performs a `refresh` operation a new descriptor is created and stored in the corresponding data structure (see Figure 4.1). In addition to that, the descriptor count of the object is incremented. Both steps only require constant time and so the whole refresh operation runs in constant time. This holds for both the `scm_refresh` and `scm_global_refresh` calls.

4.3. Data Structures

This section gives an overview of the design of the data structures we used to implement self-collecting mutators in C. Some structures are direct consequences of the short-term memory model and some are designed to improve efficiency and performance rather than representing logic. In the following we start with a top-level view of the most important data structures and then focus on implementation details and how they effect the behavior of the system. Unless stated otherwise, the definitions of the data structures used in LIBSCM can be found in the header file scm-desc.h.

4.3.1. Descriptor Root

The descriptor root is the heart of the thread-local meta data. Figure 4.2 illustrates the high level contents of this data structure. In this section we start with the top view of this central data structure and then refine its contents step by step. The global_phase field is an integer counter that determines if the thread that owns this descriptor root already did a global_tick in the current global-time period. This directly reflects the short-term memory model for shared objects presented in Section 3.2.

The descriptor_root contains three different buffers to store descriptors. The list_of_expired_descriptors stores descriptors of objects that have expired at some time in the past. These objects will eventually be reclaimed. The locally_clocked_descriptor_buffer and the globally_clocked_descriptor_buffer store descriptors that represent objects with an expiration date in the future that will be evaluated with respect to the thread-local or the thread-global clock, respectively. The descriptor_page_pool structure and the

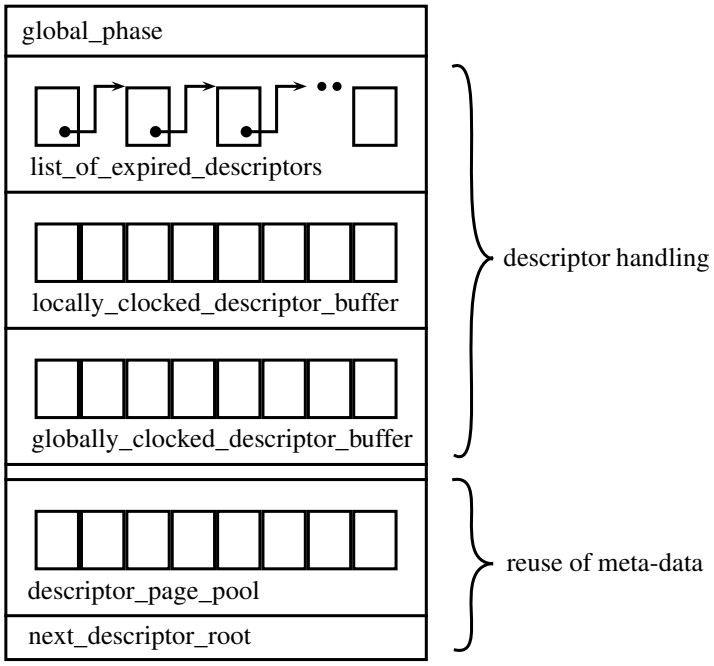

Figure 4.2.: Layout of the descriptor_root, the main thread-local data structure

next_descriptor_root field are designed to reuse meta-data in the LIBSCM implementation.

4.3.2. Descriptor Buffer

Figure 4.3 shows the structure of the descriptor buffer where not-expired descriptors are stored. It consists of an array of descriptor_page_lists. Each element of this array represents descriptors with the same expiration date. The length of the array is stored in the field not_expired_length and corresponds to the maximal expiration extension that can be handled by the refresh calls. The not_expired_descriptors array is organized as a circular buffer and current_index is the index of the descriptor_page_list that contains the descriptors that will expire after the next tick call. Consequently, an expiration date is not stored explicitly but implicitly through an offset to the thread-local time and current_index directly reflects the current thread-local time.

There are two operations that change the state of a descriptor_buffer. The first operation on a descriptor_buffer is to insert a new descriptor through a refresh operation with expiration extension e. The index of the descriptor_page_list that represents the

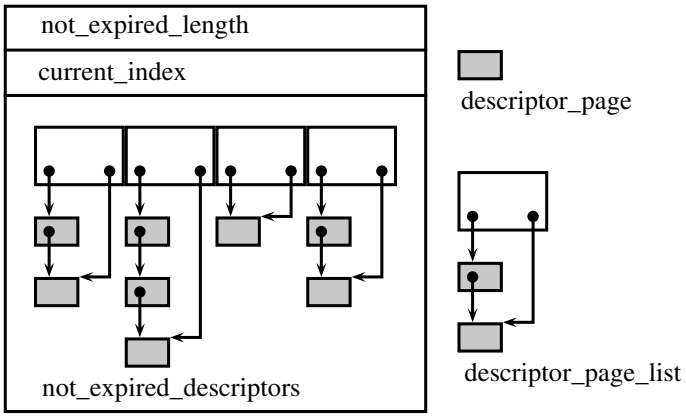

Figure 4.3.: Layout of the descriptor_buffer data structure

extension e is given through $(current\_index + e)\%not\_expired\_length$.

The second operation is to increment the current_index and expire descriptors. This implements the logic of time advance.
$Current\_index = (current\_index + 1)\%not\_expired\_length$ increments the index with respect to the circular buffer. Now, $current\_index - 1$ points to the descriptor_page_list that contains expired descriptors. This list is moved to list_of_expired_descriptors in the descriptor_root.

Support for Global-time Management

The descriptor_root contains two descriptor buffers. One locally clocked buffer and one globally clocked buffer. The locally clocked descriptor buffer contains not-expired descriptors that will be expired based on the thread-local clock and the globally clocked descriptor buffer stores not-expired descriptors that are evaluated based on the thread-global clock. When a descriptor is created through either a local or global refresh call it is stored in the locally or globally clocked descriptor buffer, respectively.

The expiration extension used in a refresh call can be between zero and a configured maximum expiration extension (max_exp_ext). Consequently, the not_expired_descriptors array in the descriptor buffer needs to be of size $max\_exp\_ext + 1$. For shared objects we need to set a larger expiration extension for the reasons we have already mentioned in Section 4.2.2. Firstly, when a threads calls a global tick operation for the first time in a global phase it has to increment the thread-global time which is implemented by

the current_index of the globally clocked buffer. Then it expires the descriptors stored in $current\_index - 1$. However, it is possible for other threads that have not ticked in this global phase yet to globally refresh a shared object that would have expired for this thread. So, we must make sure that shared objects do not expire too early. Therefore we add one additional expiration extension to each global refresh call. Secondly, it is possible that threads block for some time due to IO operations or some synchronization requirements. A blocking thread cannot perform any short-term memory operations including `global_tick`. This means that the global time cannot advance when a thread is blocking. Therefore we need to keep a shared object for another thread-global time advance to guarantee that one global phase has completed. When a thread calls `global_refresh` on an object with expiration extension e we create a descriptor that represents an expiration extension of $e+2$. As a consequence the not_expired_descriptors array in the globally clocked buffer is two elements larger than in the locally clocked buffer. This way, all operations of the descriptor_buffer data structure remain the same.

4.3.3. Descriptor Page

The descriptor_page is the data structure that actually contains the descriptors. It defines the descriptors array where descriptors are allocated in a stack-like fashion, i.e., starting at element 0 up to the maximum capacity of the descriptors array which is a compile time constant. The integer field number_of_descriptors is the index in the descriptors array where the next descriptor will be stored. The layout of the descriptor_page is shown in Figure 4.4.

In case the descriptors array runs out of capacity, descriptor pages can be extended by building a list of descriptor pages (see Section 4.3.4). The next_descriptor_page pointer is used for this purpose.

Descriptor Pages vs Descriptor List

An alternative approach to implement a collection of descriptors is a singly linked list of descriptors. The design decision to use descriptor pages instead is based on a time-space trade-off. We assume that the spacial locality of descriptor pages yields caching advantages when a page of descriptors expires because expired descriptors are likely to

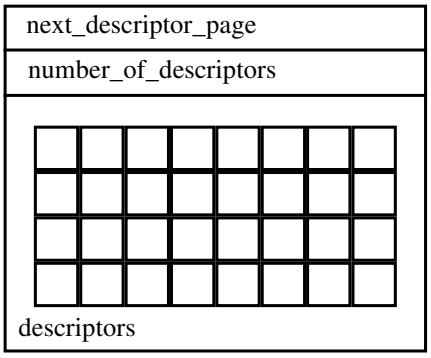

Figure 4.4.: Layout of the descriptor_page data structure

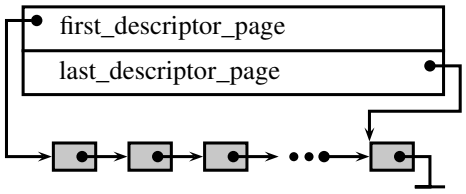

Figure 4.5.: Layout of the descriptor_page_list data structure

be processed one after another (especially in case of eager collection). In addition to that, the descriptors do not require a next pointer to build a linked list of descriptors. On the other hand a descriptor page might not be full, thus wasting space through internal fragmentation. However, we believe that the impact is relatively small and we conduct a performance evaluation on this factor in Chapter 6 that supports this claim.

4.3.4. Descriptor Page List

The descriptor_page_list data structure shown in Figure 4.5 consists of a first and a last pointer to represent the beginning and the end of a singly linked list of descriptor_pages. The first and last pointer enable for constant-time insertion and removal at either the head or the end of the list. A list of descriptor pages is used in the descriptor_buffer (see Section 4.3.2 for details) and stores descriptors that represent the same expiration date. Note however, that here descriptors are only inserted, never removed.

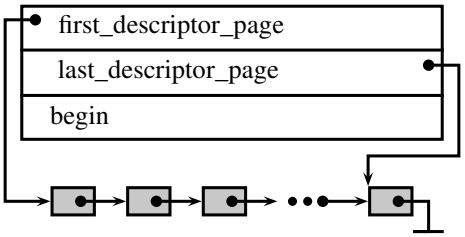

Figure 4.6.: Layout of the expired descriptor page list data structure

4.3.5. Expired Descriptor Page List

The expired_descriptor_page_list is basically the same as the descriptor_page_list. The difference is that it only stores expired descriptors. It is used in the descriptor_root data structure as a common collection for descriptors that have expired from either the locally_clocked_descriptor_buffer or the globally_clocked_descriptor_buffer. When a descriptor expires it does not matter anymore which clock expired it.

So, unlike the descriptor_page_lists used in the descriptor_buffers where descriptors are only added, here descriptors are only removed. Since we want to remove descriptors in the same order as they were added, we need to empty the descriptor_pages from the beginning. The descriptor_page has an index to the end of the descriptors that it contains (see Section 4.3.3). To not inverse the order when removing descriptors from a descriptor_page we need an index to the beginning of the page that is incremented whenever we remove a descriptor from it. Since we only remove descriptors from the first page in the expired_descriptor_page_list there is no point is storing such an index in every page. Instead we maintain the begin field in the expired_descriptor_page_list itself. It gives the position of the next-to-remove descriptor in the first descriptor_page.

4.3.6. Dynamically Allocated Management Data

The number of refresh and tick operations that a process will call are unknown when the application is started. Therefore it is necessary that most of the data structures that we have presented so far are dynamically allocated. The main portion of this data structures is used to manage the descriptors.

We aim to reuse dynamically allocated data structures instead of freeing and re-allocating

them on demand. This relaxes the utilization of the allocator. We have two main data structures that we reuse. The first one is the descriptor_page structure. This pages are used frequently when descriptors are created or expired. When a descriptor_page becomes empty we add it to the descriptor_page_pool in the descriptor_root (see Section 4.3.1) The size of this pool is bound by a compile time parameter to prevent unlimited growth. When an empty page is needed it is taken from the pool. In case the pool is empty we allocate a new descriptor_page from dynamic memory.

The second cached data structure in LIBSCM is the descriptor_root itself. When a thread terminates its descriptor_root is preserved in a lock-protected global list of descriptor_roots. When a new thread is created it reuses one of this descriptor_roots without resetting it. This way all descriptors that are still stored in the reused descriptor_root expire eventually when the new threads starts using tick and refresh. This prevents memory leaks because of threads that did not expire objects before termination.

4.4. Blocking Threads

Blocking threads are problematic because they prevent the system from making global time progress. We solved this issue by introducing a thread-global clock that is used to expire shared objects. We have discussed this in Section 4.2.2. When a thread is about to block or resume it needs to inform LIBSCM about that. We add two API calls to the public interface of LIBSCM called `scm_block_thread` and `scm_resume_thread`. Listing 4.6 gives the definition of this functions from stm.h.

```
1  /*
2   * scm_block_thread may be used to signal the short term memory
3   * system that the calling thread is about to leave the system
4   * for a while e.g. because of a blocking call. During this period
5   * the system does not wait for scm_tick calls of this thread.
6   * After the thread finished the blocking state it re-joins the
7   * short term memory system using the scm_resume_thread call
8   */
9  void scm_block_thread(void);
10 void scm_resume_thread(void);
```

Listing 4.6: Definition of scm_block_thread and scm_resume_thread from stm.h

The registration of a new thread is done automatically when it calls either refresh of tick for the first time. Upon termination a thread may call `scm_unregister_thread` to signal the system that it may reuse its descriptor_root. This way the not-expired objects kept by this thread will expire eventually. Listing 4.7 defines the interface.

```
1 /*
2  * scm_unregister_thread may be called just before a thread
3  * terminates. The thread's data structures are preserved for
4  * a new thread to join the short term memory system.
5  */
6 void scm_unregister_thread(void);
```

Listing 4.7: Definition of scm_unregister_thread from stm.h

4.5. Debug Extensions

When LIBSCM is compiled to use the lazy collection strategy it is unknown when an expired object is actually freed. This can happen at any time after the tick call that expired the object and it can be done by an other thread if the object was shared. When the programmer wants to perform a certain action when the object is freed, e.g., for heap profiling or to implement some application specific logic she can provide a finalizer function that is executed when a certain object is deallocated.

LIBSCM allows a total of 32 different finalizer functions. The programmer uses the API calls `scm_register_finalizer` to pass a function pointer of type int(*)(void*) to the short-term memory system. The return value is a finalizer index between 0 and 31. The programmer can bind a registered function to an arbitrary short-term memory object using this finalizer index and a pointer to the object through the `scm_set_finalizer` call. The finalizer index is stored in the object header of the short-term memory object. When the expired object is freed the finalizer function with the index bound to this object will be executed. A pointer to the short-term memory object is passed as parameter to the finalizer function. The programmer can perform any action in the finalizer code. It is possible to keep an expired object alive by returning a non-zero integer from the finalizer. In this case LIBSCM will not free the object. The programmer must take care of reusing the object by either freeing it using `scm_free` or by setting a new expiration extension using a refresh call.

```
1  /*
2   * scm_register_finalizer is used to register a finalizer function
3   * in libscm. A function id is returned for later use.
4   * It is up to the user to design the scm_finalizer function. If
5   * scm_finalizer returns non-zero, the object will not be
6   * deallocated. LIBSCM provides the pointer to the object as
7   * parameter of scm_finalizer.
8   */
9  int scm_register_finalizer(int(*scm_finalizer)(void*));
10
11 /*
12  * scm_set_finalizer can be used to bind a finalizer function id
13  * (returned by scm_register_finalizer) to an object (ptr).
14  * This function will be executed just before an expired object is
15  * deallocated.
16  */
17 void scm_set_finalizer(void *ptr, int scm_finalizer_id);
```

Listing 4.8: Definition of scm_register_finalizer and scm_set_finalizer from stm-debug.h

4.6. Summary

In this chapter we described LIBSCM, a concurrent, incremental, and non-moving implementation of a short-term memory algorithm called self-collecting mutators. Starting from our design goals like constant space and time overhead per object and the absence of additional memory management threads we developed the implementation details from a high-level view down to the actual operations performed on short-term memory objects.

LIBSCM supports explicit short-term memory functions including allocation, setting expiration extensions and thread-independent time progression. Furthermore LIBSCM implements the ability to manage shared objects including automatic lifetime reasoning for independently ticking threads which guarantees that shared objects expire only if they expired for all threads.

5. ACDC Benchmark

Our goal is to evaluate the performance of all features of our implementation of short-term memory which includes multi-threaded mutators, thread-local objects as well as shared objects and independent control of time for all threads. For this task, we need an application that exercises the features that LIBSCM provides. This chapter presents the way towards such an application.

In multi-threaded applications, the correct use of the persistent memory model can be a difficult job because the reasoning about the last use of an object is a non-trivial problem in general. This is why large-scale applications and libraries in C often come with some sort of domain-specific garbage collection - usually reference counting - to deal with the problem of lifetime reasoning [15]. One prominent example for such a strategy is the object memory management of the Gnome desktop environment [25].

The short-term memory model enables the programmer to use dynamic memory in a way that is very different to the traditional way of dynamic memory management in C using malloc and free. Given the fact that short-term memory is a novel and thereby unused model and that multi-threaded applications often use some higher-level abstraction for dynamic memory in preference to malloc and free, it turns out to be difficult to find a multi-threaded application that can easily be ported to short-term memory and being used as a benchmark application.

As a consequence of the non-existence of benchmarks that support short-term memory operations and the difficulties of porting complex multi-threaded applications we decided to create our own allocator benchmark tool that models the behavior of real-world mutators while remaining simple enough to fully understand how it exercises the system under test, our implementation of LIBSCM.

The amount of dynamic memory used by a typical mutator changes over time. When the mutator starts executing, the number of dynamically allocated objects is zero and then

starts growing until a free call is issued for the first time. This free call decreases the number of objects by one and from this time on the mutator can either allocate or free objects as the program demands. The time between the allocation and the deallocation is different for each object. They might be freed right after they were allocated or they can live for nearly the whole execution time of the program. The dynamic memory consumption of a mutator observed over time can be seen as an analogue signal that consists of different frequency components. The short-living objects correspond to high frequencies and the longer-living objects correspond to lower frequencies. They form the AC components of a signal while permanent memory or very long-living objects represent the DC component. A mixture of different object lifetimes that reflects a real-life mutator is the goal of our benchmark program and it is therefore called ACDC benchmark. In this chapter we describe the characteristics of mutators that we want to model and we take a look at the most important implementation details of ACDC.

5.1. Characteristics of Dynamic Objects in C Programs

The workload that the ACDC benchmark produces is a series of memory management API calls for either short-term memory implementations or persistent memory model implementations. The modeled workloads are inspired by the behavior of real-word applications to achieve meaningful results.

In [4] and [5] the behavior of a series of allocation-intensive C programs from a variety of application areas have been analyzed to predict the lifetime of objects allocated at a given allocation site. In [26] the authors analyze the allocation behavior of some additional applications to show that the right allocation policies can avoid fragmentation through the allocator. We now briefly discuss the most important observations made in this studies regarding the characteristics of dynamically allocated objects.

For the analyzed programs, the authors identified a connection between the time objects are allocated and the time objects are freed. They showed, that objects that are allocated together also die together. Intuitively, one can think of some objects that are allocated at the same time before a certain operation is performed on them and then are freed together after the operation is finished. [26]

A second observation made by the authors was that programs tend to allocate objects

of only a few different sizes. They showed that on average 90% of all objects are of just six to seven different sizes. The reason for this behavior is that a dynamically allocated object is usually associated with a type, i.e., a C-style structure. As a result, only a few structures that are used very often, e.g. a node of a list, dominate the number of dynamically allocated objects. Usually, structures in C are very small, e.g. a node object of a doubly linked list only requires a minimum of three words. From this we can conclude that small objects are more likely to occur compared to large ones. [26]

We synthetically model this observations of the allocation schemes of real-world applications in our benchmark tool.

5.2. A Notion of Time for a Mutator

We are interested in a meaningful model of a mutator that covers not only the size of the allocated objects but also the lifetime of an object. Defining the object size is straight forward since the malloc call takes the size in bytes as an argument. The lifetime of an object can be defined as whatever happens between allocation and deallocation of an object. It might seem that the most natural way to do so is counting CPU cycles or using some real-time clock to define a notion of object lifetime. However, allocations and deallocations are the only events of interest for an allocator [5]. The service demands placed upon an allocator are directly proportional to the rate of allocation and deallocation events rather than to the elapsed time. Two time metrics based on allocation events are directly available: the number of such allocation events and the number of bytes allocated. Like in [5], we use the number of bytes allocated to define a notion of logical time for each thread as follows:
For a thread T_i we have an integer counter $time_i$ that starts at zero. After the execution of N calls to malloc($size_j$), $j = 1 \ldots N$, a threshold level l_i for this thread will eventually be reached, i.e., $l_i \leq \sum_{j=1}^{N} size_j$ and this event will increment $time_i$ to $time_i + 1$.

5.3. Modeling the Workload

Based on the object size and lifetime characteristics described above, we model the mutator pattern that we use in the ACDC benchmark program. Instead of giving a static

model like a trace of malloc and free calls, we are interested in a tool that can be configured for different applications and different environmental conditions like the size of main memory or the number of threads that will concurrently allocate and deallocate memory. The ACDC benchmark therefore needs to support a number of runtime parameters that we will briefly describe before we proceed on how the workload is modeled based on given configuration options.

5.3.1. ACDC Runtime Options

The ACDC benchmark program can be configured using several options that change the behavior of the allocation pattern and therefore define the workload of an allocator. We now briefly describe the ACDC options.

Persistent vs Short-term memory

Using this flag the ACDC benchmark program selects the memory model to use. When persistent memory is chosen, ACDC issues a free call when the lifetime of an object has ended. The short-term memory option causes ACDC to invoke a refresh call with an expiration extension that corresponds to a given object. Then, this object is not touched anymore and it will expire eventually.

Benchmark runtime

This is an integer option that defines how often each thread increments its notion of time. When this limit is reached, the thread waits for the other threads to finish and then terminates.

Object size bounds

ACDC accepts two integer options that define a lower and upper bound for the size of each object that will be allocated. The values of this options are given in powers of two, e.g. given a lower bound of 3 and an upper bound of 20 will result in a minimum object size of $2^3 = 8$ bytes and in a maximum size of $2^{10} = 1$ kilobyte.

Time threshold

The time threshold is an integer that defines how fast time advances, i.e., the number of bytes to allocate until the clock is incremented. The value is given in powers of 2, e.g. a time threshold of 2^{20} will increment a thread's time after it allocated one megabyte of dynamic memory.

Maximum object lifetime

Every object that is allocated in ACDC is associated with a lifetime and depending on the memory model the object will either expire or will be freed after this time has elapsed. This option defines the maximum lifetime that will be associated with an object. The minimum lifetime is one, i.e., the object will only live until the time threshold is reached for the next time.

Number of threads

This integer option determines how many threads will concurrently invoke memory management calls. The current implementation of ACDC limits this number to 58. In Section 5.4.1 we discuss this limit.

Shared objects

Using this flag one can control if ACDC uses only thread-local objects or if a portion of the objects is shared among all threads. In the latter case one thread allocates an object but each thread has a reference to it and assigns its own lifetime to this shared object. It depends on the speed of time advance of each thread when a shared object is actually freed and which thread has to free it.

Share ratio

This option is given as a value between 0 and 100 that specifies the percentage of the allocated objects that are actually shared. This option only has an effect when used together with the shared objects flag.

5.3.2. Single Mutator Behavior

After this short introduction of the ACDC runtime options we give a high-level view of how ACDC works and how it can be reasonably configured to benchmark a given allocator. We start with the behavior of a single mutator thread and later describe the extensions to the algorithms to support multiple mutator threads including shared objects.

High-level View

The ACDC benchmark tool runs a given number of threads that execute the following main loop: A thread allocates a certain number of objects with a certain lifetime and records the number of allocated bytes. When the configured allocation threshold (or time threshold since we use allocated bytes as a time metric) is reached, the mutator proceeds in time. When the configured time limit is reached, the mutator terminates. Algorithm 1 gives the pseudo code for the main mutator loop.

Algorithm 1 Top-level view of an ACDC mutator

```
1: allocated_bytes ← 0
2: while time ≤ benchmark_runtime do
3:     x ← allocate_objects()
4:     allocated_bytes ← allocated_bytes + x
5:     if allocated_bytes ≥ time_threshold then
6:         proceed_in_time()
7:         allocated_bytes ← 0
8:     end if
9: end while
```

Allocation

We are now going to refine the mutator behavior of the ACDC benchmark tool and describe the functions `allocate_objects` and `proceed_in_time`.

The mutator models the characteristics of dynamic objects as described in Section 5.1. It creates more small objects than large ones and more short living objects than long living objects. We model the size and the lifetime of an object as random variables and we use probability distributions to determine their values.

We group the allocated objects in simple size classes that derive from the binary logarithm of the size of an object. We define a size class as follows:
Let $sz : Objects \to \mathbb{N}$ be a mapping where an object $o \in Objects$ is mapped to its size. The binary relation "same size-class", $R_{sz} \subseteq Objects \times Objects$, is defined as follows. Two objects are related, $(o_1, o_2) \in R_{sz}$ if and only if there exists an $sc \in \mathbb{N}$ such that $2^{sc} \leq sz(o_1) < 2^{sc+1}$ and $2^{sc} \leq sz(o_2) < 2^{sc+1}$. It is easy to see that R_{sz} is an equivalence relation and sc represents the equivalence class $[sc]$. We say o_1 and o_2 belong to the same size-class $[sc]$.

Allocators that use size-classes might define their size-classes in a more sophisticated way but for our model the powers of two are sufficient.

Let $lifetime : Objects \to \mathbb{N}$ be a mapping that determines the lifetime $lifetime(o)$ of an object $o \in Objects$. After the mutator reached the time threshold $lifetime(o)$ times the object o will be freed or expired. Objects with the same lifetime lt define the trivial equivalence class of the *equals to* relation $[lt]$.

The results from [26] and [4] suggest that the cardinality of a size-class is correlated to the size of the objects it contains. The same applies to the number of objects that share the same lifetime. Our goal is to derive the number of objects based on random variables for size-class sc and lifetime lt.

We start with discrete uniform distributions for the size-class and the object lifetime. They provide a random variable sc from the interval that satisfies the object size bounds and a random variable lt which is bound by the maximum object lifetime. Both constraints are given by the runtime options.

Both random variables have an indirectly proportional effect on the number of objects

that will be allocated with lifetime lt and a size between 2^{sc} and 2^{sc+1}.

The impact on the number of objects that is caused by the random size-class and the random lifetime is defined as follows:

$$impact\_of\_sc = (sc_{max} - sc + 1)^2$$

and

$$impact\_of\_lt = (lt_{max} - lt + 1)^2$$

meaning that the impact is stronger if the value of the random variable has a larger difference to its configured maximum (sc_{max} and lt_{max} are runtime options). This gives a larger number of objects for smaller size-classes and shorter lifetimes and a smaller number of objects for larger size-classes and longer lifetimes. We even increase the effect by modeling a quadratic impact of both effects.

The actual number of objects depends on the impact of the size-class and the lifetime and additional runtime options to configure the mutator behavior of the ACDC benchmark.

$$\#objects = \frac{impact\_of\_sc \times impact\_of\_lt \times multiplier}{\#threads \times lt_{max}}$$

The constants $multiplier$, $\#threads$ and lt_{max} are the runtime parameters. *Multiplier* is used to directly affect the heap size while $\#threads$ and lt_{max} are used as correction parameters. The size of the heap grows proportionally with the number of threads and the maximum object lifetime. This fact is compensated by the denominator of the formula to achieve a constant per-thread memory consumption. If, on the other hand, the user wants to have a bigger heap, this can be achieved by setting the *multiplier* accordingly.

This model is used in the `allocate_objects` function called in Algorithm 1. The pseudo code of `allocate_objects` is shown in Algorithm 2. Again, the basic idea is to retrieve a random value for size and lifetime and then calculate how many objects with this very size and lifetime should be allocated. The mutator stores a reference to the allocated objects in a buffer. This object buffer groups the objects by their lifetime. When time advances, the corresponding group of objects in the buffer are either freed or expired, depending on the chosen memory model.

Algorithm 2 Allocate objects in ACDC

```
1: bytes ← 0
2: get sc from uniform distribution
3: get lt from uniform distribution
4: calculate #objects from impact_of_sc and impact_of_lt
5: for i=0; i< #objects; i++ do
6:    object o ← new object in size-class sc
7:    object_buffer[lt] ← object_buffer[lt] ⋃ o
8:    bytes ← bytes + sz(o)
9: end for
10: return bytes
```

Since the number of objects is derived from the random variables size-class sc and lifetime lt, the number of objects is also a random variable. The probability density function follows the product of the quadratic impacts for sc and lt. This models the described characteristics of dynamic objects resulting in a large number of small short-living objects and a small number of large long-living objects.

Figure 5.1 shows an example histogram of the lifetime distribution. The example is taken from the experiment in Section 6.4.3 using a time threshold of 2^{12} running 6 mutator threads for a benchmark runtime of 100 time advances. It shows a configured maximum lifetime of 10 and illustrates that a shorter lifetime is much more likely than a larger one. When the lifetime reaches the configured maximum object lifetime the number of objects approaches zero.

In Figure 5.2 we can see the characteristics of the randomly chosen size-class for the same experiment. In this example, the configured minimum size-class is three and the maximum size-class is 12. Again, smaller objects are much more likely to be allocated than larger ones. This scenario also reflects the characteristics of dynamically allocated objects in C programs.

Time Progression

The ACDC main loop in Algorithm 1 is repeated until the configured time threshold is reached. When this happens, the mutator discards the objects whose lifetime has ended.

Depending on the configured memory model, the routines `refresh_future` and `delete_past` either perform a refresh operation on objects that have some remain-

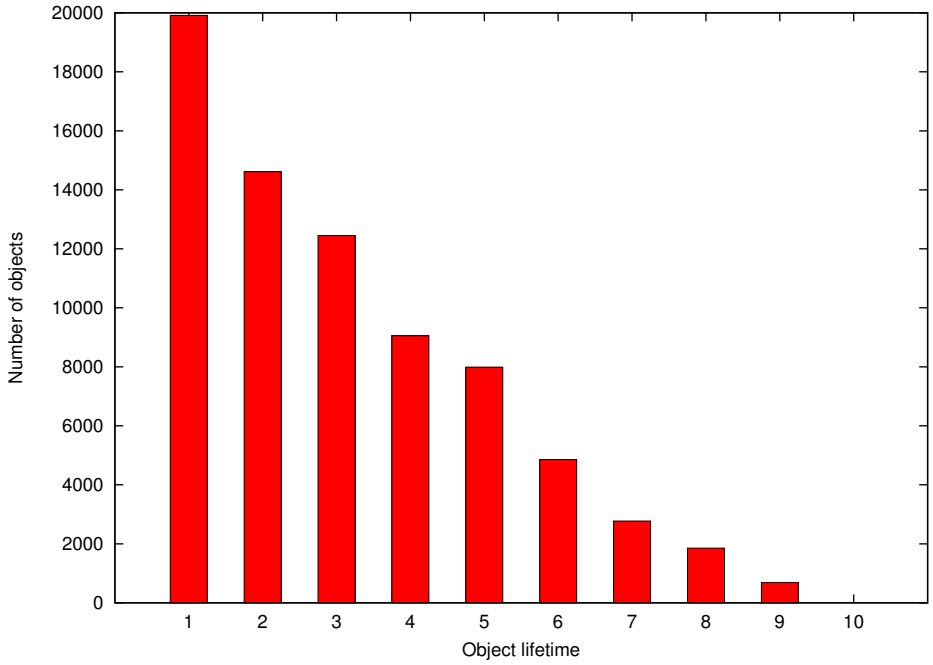

Figure 5.1.: Example lifetime histogram

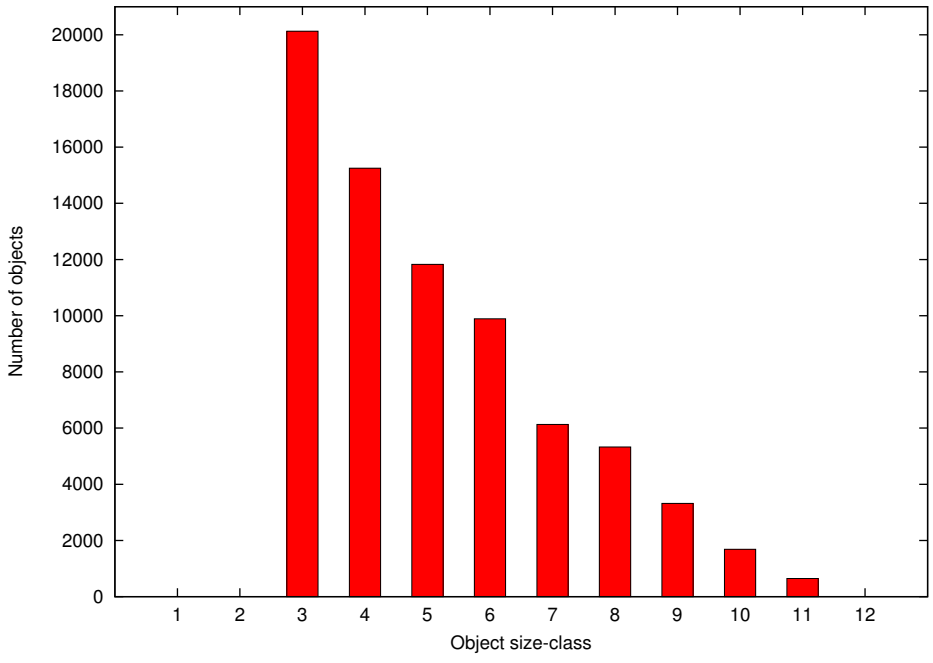

Figure 5.2.: Example size-class histogram

Algorithm 3 Advancing time in ACDC

```
1: if using short-term memory model then
2:    refresh_future()
3: else if using persistent memory model then
4:    delete_past()
5: end if
6: thread_time ← thread_time + 1
```

ing lifetime or perform a free operation on objects where the lifetime has ended. The lifetime is set for each object at allocation and determines for how many time advances of the mutator the object will survive.

In the short-term memory case, an object that must be kept alive is refreshed with its remaining lifetime as expiration extension. An objects o with a remaining lifetime of lt is treated with a `scm_refresh(o, lt)` call. This happens each time a mutator calls `refresh_future`. Note that this adds redundant expiration dates to short-term memory objects because the exact lifetime is known at the time of allocation and the first refresh operation already set the correct expiration date. However, we do not exploit this perfect knowledge of object lifetimes so we simulate a scenario where the mutator does not know the lifetime of an object at allocation time and therefore has to refresh an object more than once. After refreshing, `scm_tick` is called to advance time in the sense of short-term memory.

In the persistent memory model configuration, we determine the set of objects with no lifetime left. We deallocate them using the `free` call from the standard C library.

After processing the dynamic objects, the mutator increments its software clock and returns to the main loop until the software clock (thread_time) reaches the configured benchmark runtime.

5.3.3. Multi Mutator Behavior

The ACDC benchmark system supports multiple mutator threads and the user can choose between a shared and a unshared setup. In the unshared setup, all mutator threads behave the same, i.e., along the behavioral pattern described in the previous section. In a shared setup, a configurable amount of objects is shared with other threads.

Allocating Shared Objects

Algorithm 4 Allocate shared objects in ACDC

```
1: bytes ← 0
2: get sc from uniform distribution
3: get lt from uniform distribution
4: calculate #objects from impact_of_sc and impact_of_lt
5:
6: for object so in sharing_pool do
7:    if so not marked by this thread then
8:       object_buffer[lt] ← object_buffer[lt] ⋃ so
9:       set mark bit on so
10:      if all mark bits set on so then
11:         sharing_pool ← sharing_pool \{so}
12:      end if
13:   end if
14: end for
15:
16: for i=0; i< #objects/share_ratio; i++ do
17:   object o ← new object in size-class sc
18:   sharing_pool ← sharing_pool ⋃ o
19:   bytes ← bytes + sz(o)
20: end for
21:
22: for i=#objects/share_ratio; i< #objects; i++ do
23:   object o ← new object in size-class sc
24:   object_buffer[lt] ← object_buffer[lt] ⋃ o
25:   bytes ← bytes + sz(o)
26: end for
27:
28: return bytes
```

All threads allocate shared and unshared objects as described in the pattern in Algorithm 2. The shared objects do not end up directly in the object buffer but they are put in a separate data structure called sharing pool. The sharing pool is a concurrent lock-based linked list to ensure mutual exclusion among the running threads. From there, all other threads can get a reference to the shared objects and set a mark bit on the object. This bit indicates if a thread already got a reference to this object. Every thread that fetches a shared object from the sharing pool increments a reference counter on the object that is initially zero. We need this counter to later decide if it can be freed or not. This technique is related to domain-specific garbage collection presented in Chapter 2.

We look at the implementation details in Section 5.4. The last thread that reads the reference of a shared object sets its mark bit and removes it from the sharing pool. In this fashion, shared objects are distributed among the threads.

Algorithm 4 shows the extension to the function *allocate objects* where a percentage of *share_ratio* of the allocated objects are put in the sharing pool instead of being directly stored in the object buffer. In Lines 6 to 15, the mutator checks the sharing pool for objects that it has not fetched before, i.e., the corresponding mark bit is not set yet. Now the mutator copies the reference to *so* to its corresponding object buffer. If the mutator was the last thread that set its flag, i.e., all mark bits are set, the object is removed from the sharing pool. As a consequence, the size of the sharing pool is bounded if all threads make progress.

In Line 17 to 21 of Algorithm 4, the configured share ratio of objects is allocated and put in the sharing pool. Note that the allocating thread will receive these objects again in the next round when inspecting the sharing pool. Lines 23 to 27 allocate the remaining $(100 - \textit{share ratio})\%$ of objects and put them directly in the object buffer.

Multi-threaded Time Progression

When ACDC is configured to run multiple threads that share objects among each other the functions `refresh_future` and `delete_past` have to handle shared objects in a different way than unshared objects.

Using the short-term memory model, a shared object *so* with a remaining lifetime *lt* is refreshed with a `scm_global_refresh(so,lt)` call. Objects that have ended their lifetime are not refreshed and will be reclaimed by the short-term memory system. After refreshing the shared objects, we call `scm_global_tick` to notify our short-term memory implementation that the calling thread is ready to advance global time.

In the persistent memory model, objects with an ended lifetime are freed like in the single-thread case. The only difference is that we have to guarantee that only one thread actually frees the object. Determining the thread that keeps the last reference to the shared object is done by the reference count of the object. Every thread that wants to remove an unused object decrements the reference count and the one thread that decrements the counter to zero calls `free` to reclaim the memory.

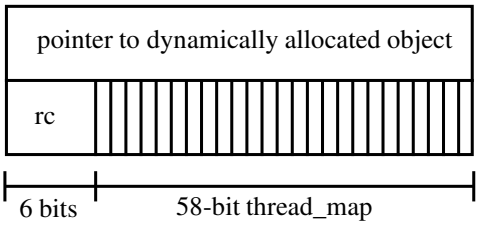

Figure 5.3.: Memory layout of the shared_pointer structure.

5.4. Implementation Details

In this section, we examine the most important parts of the ACDC implementation. Basically, the mutators only allocate objects, store references to them for a while in a buffer, and later free or refresh them. We describe the data structures and discuss the synchronization primitives we use to handle concurrent access to the shared data structures.

5.4.1. Data Structures

Shared Pointer

To make the sharing of objects easier, we need to add some meta data to every shared object. We do not modify the object header that is used by the allocator because we designed the benchmark tool to be usable with any allocator. We create a wrapper called shared_pointer around every object and add a 6-bit field as a reference count and a 58-bit field as thread mask to track how many and which threads share an object. These two fields can be combined in one 64-bit word, limiting the number of threads to 58. Supporting more threads would increase the overhead by at least another word. However, such a modification would be possible. Figure 5.3 illustrates the memory layout of the shared_pointer structure. We use this wrapper also for unshared objects to keep the code more general at the expense of two extra words per object.

Whenever a dynamic object is allocated, it is assigned to a shared_pointer wrapper. In case of a shared object, the reference count and the thread mask are important. The distribution of shared objects is done by a common data structure called sharing pool where the allocating thread puts in a new shared object and all threads (including the

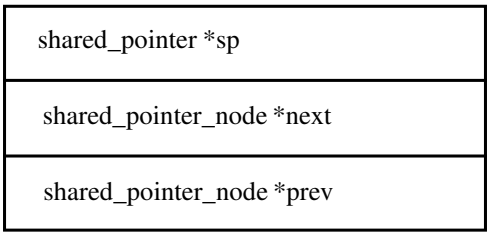

Figure 5.4.: Memory layout of the shared_pointer_node structure.

allocating thread) fetch the object from the pool. The bit mask is initially zero which means that no thread has a reference to this object. Each thread has a unique identifier starting from zero up to 57. When the i^{th} bit in the thread mask is set, this means that the thread with id i has copied a reference to the object. When the thread mask is full, i.e., all bits are set, this means that all threads have got a reference to this object and consequently it does not need to stay in the pool any longer. The last thread that marks its bit removes the object from the pool.

In addition to that, every thread that receives a reference to an object, increments the reference count of this object. This reference count is not important for the distribution of the shared objects but for being able to free the object later on. When the lifetime of an object ends for a thread, this thread decrements the reference count and deletes its reference to this object. Only the last thread that performs this step decrements the reference count to zero and is allowed to free the object.

List of Shared Pointers

Shared_pointers are organized in doubly-linked lists. A straight forward way would be to add a previous and next pointer to the shared_pointer structure. However, this is not possible in our case because a shared_pointer needs to be stored in multiple lists since multiple threads might have a reference to a shared object. We define a shared_pointer_node structure to add list functionality to a shared_pointer. Figure 5.4 depicts the layout of the shared_pointer_node structure.

A list of shared_pointer_nodes is called shared_pointer_list. It keeps a reference to the beginning and the end of the doubly-linked list and also stores a mutex variable to protect concurrent access if necessary. Figure 5.5 shows the layout of this data structure.

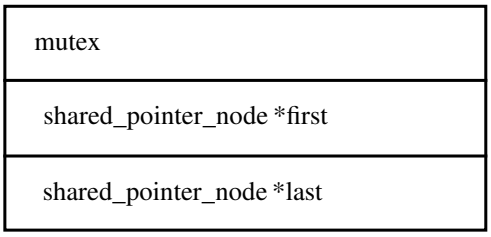

Figure 5.5.: Memory layout of the shared_pointer_list structure.

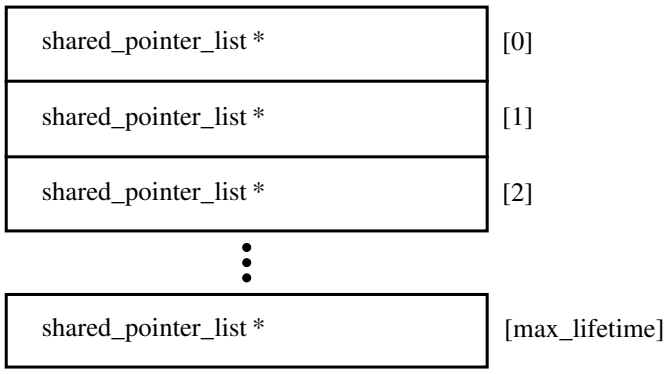

Figure 5.6.: Memory layout of the object buffer data structure. Each array element represents a common lifetime.

Object Buffers

Each thread stores its references to both shared and unshared objects in distinct object buffers. An object buffer is an array of shared_pointer_lists where each array element represents a common lifetime, i.e., all objects in one shared_pointer_list have the same lifetime. Consequently, the length of the array must be at least the maximum lifetime plus one, because we also need one slot for objects with lifetime zero.

The object buffer is organized as a cyclic buffer where the thread time modulo the maximum lifetime is used as an index into that array of size $max\_lifetime + 1$. When the thread time is t and the length of the buffer is l then $(t + i)\%l$ gives the index of the shared_pointer_list that contains objects with remaining lifetime i. Figure 5.6 shows the organization of an object buffer.

Since shared and unshared objects are treated differently, each thread has one object buffer for shared objects and one for unshared objects. Their functionality, however, is the same.

Sharing Pool

As mentioned before, we maintain a shared data structure to distribute the references to objects which are shared among threads. We call this the sharing pool although it is implemented as an ordinary shared_pointer_list. When a thread allocates shared objects, it puts them in the sharing pool, i.e., it enqueues them in a list. In order to not lock the list for every single object that a thread creates, this is done in a batch-like fashion. The mutator allocates shared objects in a loop and enqueues them in a temporary shared_pointer_list. When the loop is finished, the sharing pool is locked using a mutex variable and the temporary list is added to the sharing pool in one constant-time concatenation operation.

When the mutator threads inspect the sharing pool for new shared objects, they need to lock the sharing pool as well and iterate through it once. This operation runs in linear time with respect to the length of the sharing pool list. However, the size of this list is usually small since only a small fraction of all allocated objects is shared.

5.5. Summary

In this chapter we have presented the main aspects of ACDC, a memory management benchmarking tool that supports both the persistent and the short-term memory model. The key features of ACDC are the ability to run multiple mutators that share memory and have independent clocks. This allows us to utilize all features of our short-term memory implementation LIBSCM. Furthermore, ACDC implements the same mutator behavior using the persistent memory model in order to get a comparison of the temporal and spatial performance metrics of both models.

6. Experimental Evaluation

In the previous chapters we have discussed design decisions we have made to meet certain performance goals. In this chapter we run a number of experiments to empirically show if we met our goals. We start with a description of the experimental environment and then proceed to the definition of the workload, the system under test and the factors that we want to study. We perform various experiments to evaluate the impact of these factors on the performance of our implementation of short-term memory.

6.1. Prerequisites

6.1.1. Experimental Setup

All experiments that are presented in this chapter were carried out on the same hardware platform. We used an AMD based 24 core server machine with 48 GB of shared memory. The hardware specifications are given in Table 6.1.

Unless stated otherwise, we repeated all experiments until we reached 95% confidence for an interval within 10% of the arithmetic mean value.

6.1.2. Technical Measurement Details

Time

For the time measurements of the LIBSCM calls we use the Read Time-Stamp Counter (RDTSC) operation [27] of the x86 instruction set architecture. The time-stamp counter is a register that is incremented every clock cycle. We read the contents of this register

CPU type:	AMD Opteron 8425 HE
CPU Freq:	2.1 GHz
Number of CPUs:	4
Cores/CPU:	6
L1d cache:	64 kB per core, 2-way
L1i cache:	64 kB per core, 2-way
L2 cache:	512 kB per core, 16-way
L3 cache:	5118 kB per CPU, 48-way
Memory:	48 GB

Table 6.1.: Hardware Specifications for all experiments

at the beginning and the end of the operation we want to measure. This gives us a time metric in CPU cycles.

We are only interested in the temporal performance of the LIBSCM operations but not of the underlying allocator. Thus we only count the cycles spent inside LIBSCM by subtracting the time spent in the allocation routines of GLIBC.

Space

Space measurements need some more attention because we use different memory management systems. In the persistent memory model we use the malloc and free calls from the standard library where one can use the `mallinfo` call defined in malloc.h [7]. LIBSCM is based on malloc and free but also adds dynamic memory overhead by itself. Using `mallinfo` we have no chance to distinguish the space for managed objects from the space for bookkeeping.

We want to be able to compare the memory consumption for both the persistent and the short-term memory model. Specifically, we are interested in the additional memory consumption that is introduced by the over-approximation that short-term memory makes on the set of needed objects.

For this comparison we instrument the code of our benchmark tool ACDC. In the persistent-memory configuration we count the number of allocated and freed objects, and keep track of the allocated and freed bytes. Thereby we are able to calculate the number of used objects and bytes at any time instant during the execution of ACDC. For the short-term memory configuration this is not enough because we do not know when

LIBSCM actually frees an object. Therefore we also instrument the code of LIBSCM to keep track of every allocated and freed object together with its size. Furthermore we add an API call to the library so that an application using LIBSCM can fetch the memory information at any time.

Listing 6.1 gives the definition of the memory statistics calls described above. This functionality is only available if LIBSCM is compiled with the `-DSCM_PRINT_MEM` option. Note that the logging and calculation of memory consumption adds runtime overhead to LIBSCM. When running experiments that measure execution time, we compile LIBSCM without this functionality.

```
1  /*
2   * struct scm_mem_info is used to fetch information about memory
3   * consumption during runtime.
4   */
5  struct scm_mem_info {
6          unsigned long allocated;/* total allocated bytes */
7          unsigned long freed;    /* total freed bytes */
8          unsigned long overhead; /* overhead by LIBSCM */
9          unsigned long num_alloc;/* total of allocated obj. */
10         unsigned long num_freed;/* total of freed objects */
11 };
12
13 /*
14  * scm_get_mem_info is used to query the contents of a
15  * struct scm_mem_info from LIBSCM
16  */
17 void scm_get_mem_info(struct scm_mem_info *info);
```

Listing 6.1: Definition of scm_mem_info and scm_get_mem_info from stm-debug.h

6.2. Workload Selection

"*The workload is the most crucial part of any performance evaluation project.*" [28]

In this section we describe the way we exercise our short-term memory implementation LIBSCM using our synthetic ACDC benchmark tool. We briefly discuss the major

considerations in selecting the workload, i.e., the services exercised by the workload, the level of detail, representativeness, and timeliness. [28]

6.2.1. Services Exercised

We use the term System Under Test (SUT) to denote our short-term memory implementation LIBSCM. In order to understand if all services of the SUT are properly exercised we have to point out which services LIBSCM provides. The obvious choice of a service interface is the API of LIBSCM (see Section 4.2). Hence, the services of LIBSCM are the memory management calls to refresh an object and to tick. We define the following functions of the LIBSCM API as the services of our System Under Test.

- `scm_refresh` to refresh thread local objects
- `scm_global_refresh` to refresh shared objects
- `scm_tick` to increase the thread local notion of time
- `scm_global_tick` to increase the global notion of time

The goal is to select a workload that exercises all services in a way that we can reason about the performance of each of them as well as the combination of the services. ACDC can be configured to share objects between multiple threads and all threads have their own speed in time advance. In this way, all of the four services are exercised. Of course, ACDC was designed with this considerations in mind.

6.2.2. Level of Detail

After we have defined the services of our System Under Test it is clear that the workload will consist of a series of local or global refresh and tick calls. Now we have to define the level of detail that the workload should represent. We use the frequency of the service calls to model the workload for the SUT.

Let us review the characteristics of the way we use the short-term memory model. The model allows us to set the expiration date of an object and to advance time. We assume that multiple objects will be refreshed between two subsequent tick calls since

applications that make use of dynamic memory usually allocate more than just one object. On the other hand, subsequent tick calls without refreshes in between, we consider rather unlikely. Instead, one would set a smaller expiration extension on the objects and tick less often. As a consequence, we define the frequency of refresh calls to be much higher than the frequency of tick calls. Furthermore, we assume that local refresh calls have a higher frequency than global refresh calls because ususally more objects are thread local than shared among threads.

ACDC allows us to configure the frequencies of the API calls in two ways. Firstly, the time threshold determines the relative frequencies of refresh and tick calls. Setting a higher threshold increases the number of refreshs while a smaller threshold decreases the number of refresh calls between two subsequent tick calls. Secondly, the relative amount of shared objects is direcly proportional to the relative frequencies of local and global refresh calls. A higher share ratio yields a higher ratio of global refresh calls.

6.2.3. Representativeness

Another important goal is to create a workload that reflects typical application behavior. This goal is not easy to fulfill because applications can differ dramatically in their demand for dynamic memory. However, in [4] and [5] the authors give some characteristics on the allocation behavior of allocation intensive applications. We have already covered this characteristics in Section 5.1. ACDC also adds computations in between allocations to relax the utilization of the memory bus. These computations are simple calculations of logarithms of double values taken from the C standard math library. Of course, such a workload can never represent all applications but ACDC makes the best effort to cover a realistic snapshot of allocation intensive applications.

6.2.4. Timeliness

Timeliness is the workload property that tells us if the chosen workload represents the current usage pattern of systems, applications, users. When we design the workload we must make sure that we do not use outdated information. Fortunately, the use cases for the C programming language change slowly and the applications that were investigated in [4] and [5] in the early 1990s are still used today. We are confident

that this usage pattern still represents todays demands on a dynamic heap memory management system.

6.3. Experimental Design

"*The goal of a proper experimental design is to obtain the maximum information with the minimum number of experiments.*" [28]

In this section we describe an experimental design that allows us to separate the effects of the factors that affect the performance of the system. We also determine which factors have significant impact to further investigate this specific factors. We apply statistical tools to cope with measurement errors and argue about statistical significance. [28]

6.3.1. Terminology

The terms we use in this section are taken from [28]. We briefly describe them here.

The **response variable** is the outcome of an experiment. Usually, this is a performance metric of the system, e.g., the response time of LIBSCM operations.

The variables that effect the response of a system are called **factors**. Since we want to evaluate a multi-threaded memory management system, the number of threads is an obvious factor that we want to analyze. We identify all interesting factors in the next section.

For every factor we can identify the **levels** that it can take. For example, the number of threads can be any positive integer with an upper bound caused by physical limitations, e.g. the number of cores on the hardware platform.

The factors whose effects needs to be quantified are called **primary factors**. Those factors that affect the response but we do not want to quantify are called **secondary factors**. As an example, the CPU clock speed will affect the execution time of our operations but we are not interested in this effect. However, we want to take care that the secondary factors are under control and do not change by chance.

The repetition of an experiment is called **replication**. We perform a number of repli-

cations of our experiments to eliminate measurement errors. The specification of the number of experiments, the factor and level combinations and the number of replications is called a **design**. In the following sections we set all these terms to our demands and perform the resulting design.

6.3.2. Factors

We expect that the following factors have an performance impact on LIBSCM.

Number of threads: The LIBSCM API calls that operate on shared data are likely to experience contention on memory accesses. We are interested in how much this bounds the scalability of our system.

Descriptor_page size: The impact of this factor tells us if the use of descriptor pages instead of descriptor lists is a significant choice. A 24 byte descriptor page contains only a single descriptor which is nearly equivalent to a list of descriptors.

Maximum lifetime (max expiration extension): This defines the granularity of lifetimes in an application that uses short-term memory.

Collection strategy: The setting of the collection strategy moves the costs from the refresh to the tick operations, and vice versa.

Time Threshold (heap size): The size of the managed heap is of particular interest for the evaluation of every memory management system.

The compiler optimization level, architecture properties like cache size and CPU speed as well as the remaining ACDC settings, e.g., the ratio of shared objects, are the secondary factors that may have an performance impact. We control this factors in our experiments to ensure that their influence is not changing.

6.3.3. Evaluation

We perform a $2^k r$ factorial design [28] to determine the impact of the primary factors. This design investigates all combinations of k factors at two different levels. It is important that we can assume a monothonic response when we monotonically change the

Factor	Label	(-)	(+)
Number of threads	A	2	8
Descriptor page size	B	24	4096
Max. lifetime	C	10	30
Collection strategy	D	lazy	eager
Time threshold	E	2^{18}	2^{20}

Table 6.2.: Primary factor/Level settings for the $2^k r$ factorial design.

Max object lifetime:	10
Benchmark runtime:	2000 iterations
Min object size:	2^3 bytes
Max object size:	2^{16} bytes
Shared objects:	yes
Share ratio:	10%
GCC optimization:	O2

Table 6.3.: Secondary factor settings for the $2^k r$ factorial design.

level of a factor. The experiment for each factor combination is repeated r times to take care of measurement errors. This results in $2^k r$ experiments that we have to run. Given five primary factors and five replications we run 160 experiments to isolate the effect of each factor and the measurement errors.

We assume that the most important response is the execution time for each API call. However, the primary factors also affect the memory consumption and the throughput of the system. We investigate this effects when we run detailed experiments for the factors that have the strongest impact on execution time.

Table 6.2 shows the levels that we set for the primary factors. We assign a label (A to E) to each factor to give a short-hand factor combination notation, e.g. ADE stands for the combined performance impact of factor A, D and E. The two different levels are labeled with a negaive (-) and a positive (+) sign. The secondary factors and their settings are listed in Table 6.3.

Allocation of Variation

We use the signtable method described in [28] to calculate the percentage of variation of each factor in the $2^k r$ experimental design. We performe the same setup for each of

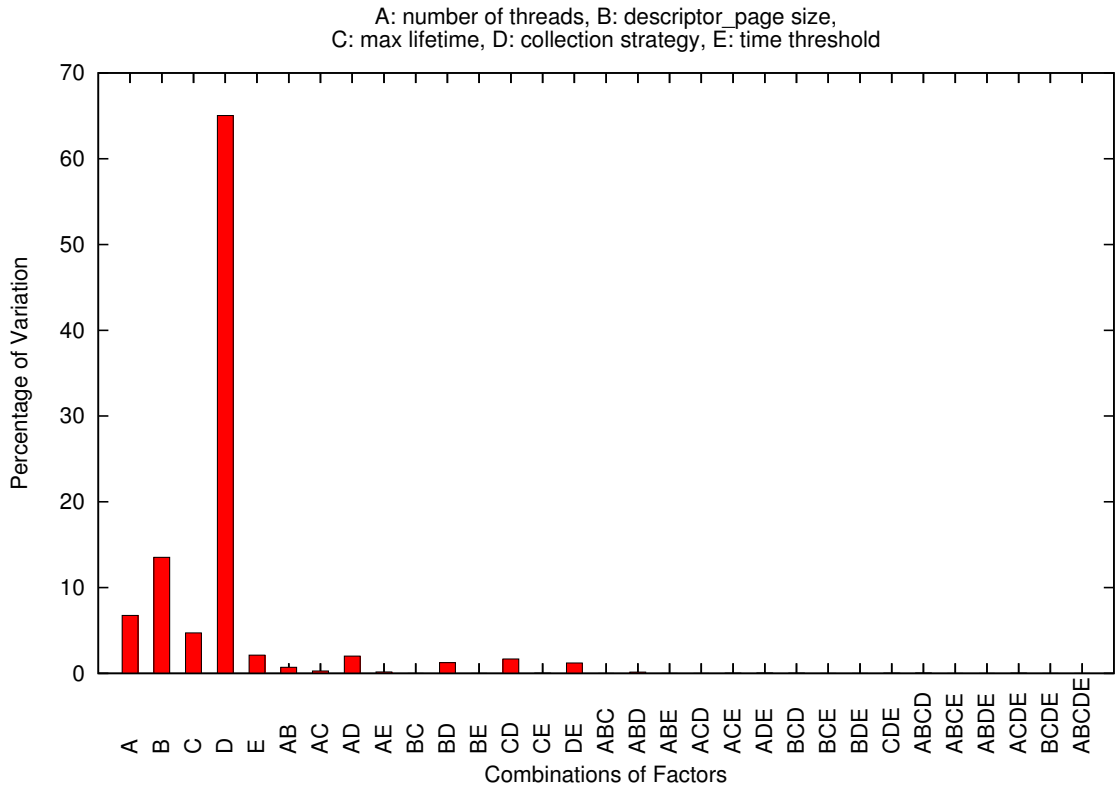

Figure 6.1.: Variation of the runtime of the scm_refresh call influenced by the combinations of factors A, B, C, D, E.

the LIBSCM API calls and derive the impact of each factor for the execution time of this function.

Figure 6.1 shows the percentage of variation of the `scm_refresh` operation. The combination of factors is given as concatenation of the factor label on the x-axis. We see that the collection strategy (D) has the strongest impact (65%) on the execution time of `scm_refresh`. In the eager collection scheme the refresh operations only create descriptors while in the lazy collection scheme they also process one expired descriptor. The descriptor_page_size (B) also shows significant influence because adding a descriptor to a non-full descriptor page is cheaper than adding a new node of a descriptor page list. The maximum object lifetime accounts for about 5% of the performance variation. This can be explained with the larger descriptor buffers used in LIBSCM and possible consequences on locality. However, we only concentrate on factor combinations with more than 10% variation. The number of threads (A) causes little variation and we can assume that `scm_refresh` scales well with the number of threads. Also, the size of the

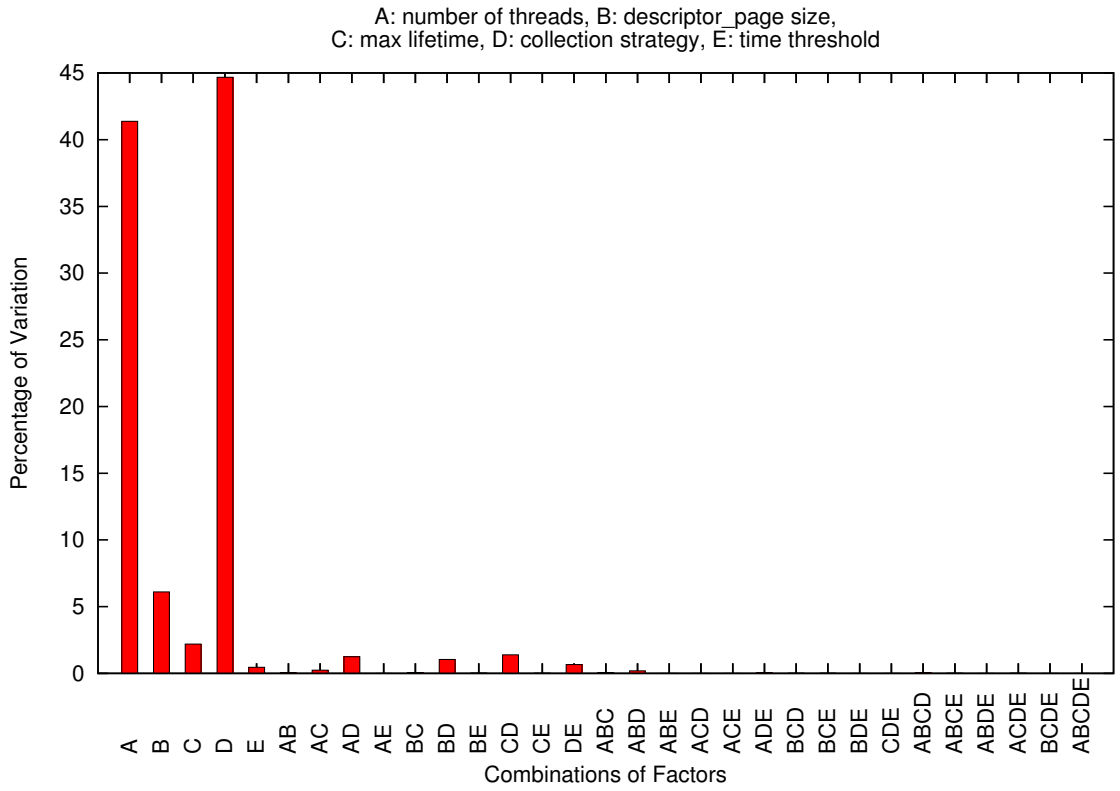

Figure 6.2.: Variation of the runtime of the scm_global_refresh call influenced by the combinations of factors A, B, C, D, E.

managed heap (time threshold, E) has no significant impact on the refreshing of local objects. None of the combinations of factors show any impact worth mentioning.

In Figure 6.2 we see the allocation of variation of the `scm_global_refresh` call. The results are similar to the `scm_refresh` call with the exception of factor A. The number of threads account for more than 40% of the performance impact. We can assume that refreshing shared objects results in contention on shared memory locations.

The percentage of variation for the `scm_tick` call is presented in Figure 6.4. Again, the collection strategy has the biggest influence on the execution time. The is explained with the extra collection work that the tick operations have to perform in the eager collection configuration. For `scm_tick`, the size of the heap (time threshold) shows significant impact on the performance. Furthermore the combination of the heap size and the collection strategy (factor DE) accounts for about the same percentage of variation. This means, that the size of the heap only affects the performance in combination with the collection strategy. Eager collection reclaims all expired objects as soon as possible,

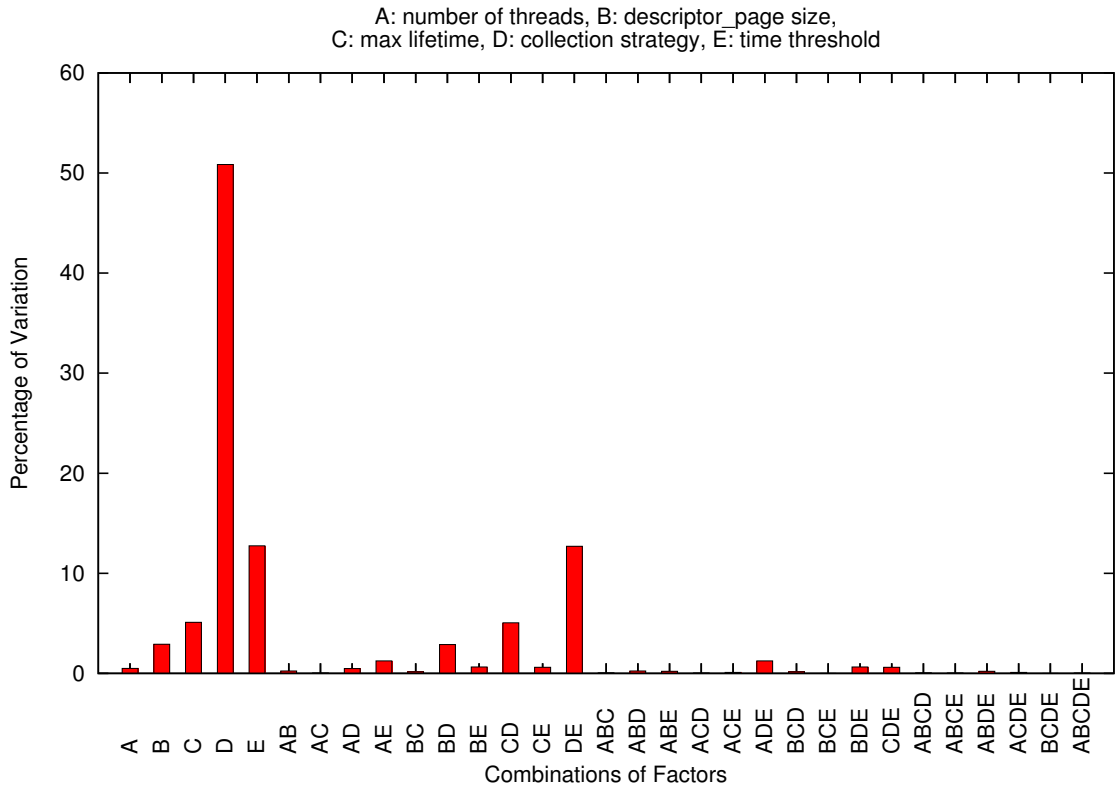

Figure 6.3.: Variation of the runtime of the scm_tick call influenced by the combinations of factors A, B, C, D, E.

i.e., as part of the tick call. In this case the execution time depends on the number of expired objects which is proportional to the total size of the heap. The maximum object lifetime has only little influence on the performance. Still, since ACDC keeps the number of objects constant per thread a larger lifetime yields more life objects than expired objects so the combination with the collection strategy (factor CD) is responsible for the performance variation.

Figure 6.4 gives the results for `scm_global_tick`. They are similar to the results of `scm_tick` because both functions perform the same operations only on different data structures. However, the number of threads accounts for more variation than in `scm_tick` because eager collection of shared objects may result in memory contention on the descriptor counts. Combination AD suggests that the number of threads only impacts performance together with the collection strategy.

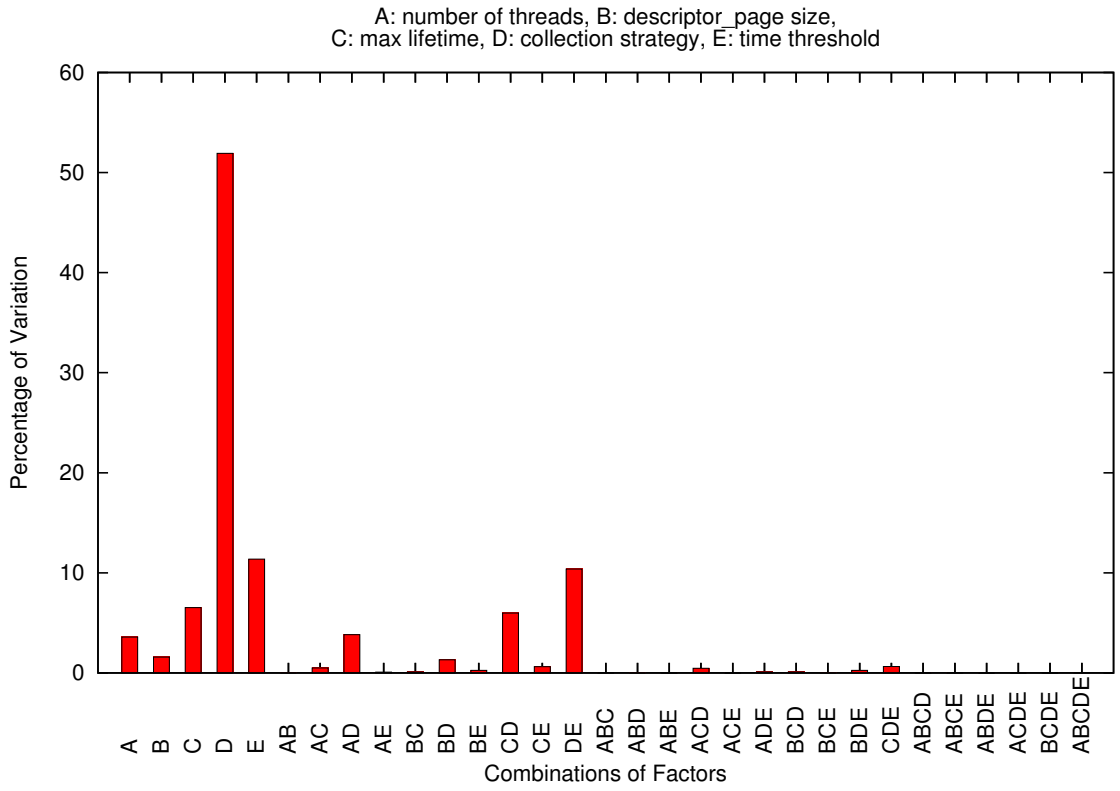

Figure 6.4.: Variation of the runtime of the scm_global_tick call influenced by the combinations of factors A, B, C, D, E.

6.4. Evaluation of the Important Factors of LIBSCM

The evaluation of our experimental design already gives us some idea of the factors that affect the performance of the LIBSCM API calls. In this section we explore these factors in more detail beginning with a scalability experiment. Starting with one mutator, we increase the number of threads and report the temporal performance of our short-term memory system. Another factor that deserves our attention is the collection strategy that LIBSCM uses. We perform a scalability experiment for both lazy and eager collection. Last but not least, the experimental design also gives a hint that the time threshold, which controls the size of the heap, has an impact on performance. In this section we take a detailed look at the performance of the API calls while we change the number of threads, the collection strategy, and the heap size of the ACDC benchmark tool.

6.4.1. Collection Strategy

LIBSCM implements two different collection strategies namely eager collection and lazy collection. The lazy collection strategy is the setting that allows a maximum of incrementality for object reclamation. At each refresh operation at most one expired object is freed. Eager collection reclaims expired objects as soon as possible, i.e., right after the tick call that expired these objects, so we expect maximum throughput using this strategy. We now describe the performance impact of this strategies.

Expectations

The evaluation of our experimental design in Section 6.3.3 shows that the collection strategy accounts for about 50% of the variation of the response time of the LIBSCM functions. However, the $2^k r$ setup does not tell us if the operations become faster or slower on either strategy setting. For the refresh calls, we expect a better performance using the eager collection strategy. The reason is that all expired descriptors are processed at the tick call which is the earliest possible moment to do so. Subsequent refresh calls do not need to process any descriptor because there are no more expired descriptors in the system. On the other hand we expect a longer execution time for the tick calls because in the eager collection strategy all expired descriptors are reclaimed here. This is the setting for maximum throughput but the time for the tick calls now depends on

Max object lifetime:	10
Benchmark runtime:	200 iterations
Min object size:	2^3 bytes
Max object size:	2^{12} bytes
Time threshold:	2^{15} bytes
Memory model:	short-term
Shared objects:	yes
Share ratio:	20%

Table 6.4.: ACDC settings for the scalability experiment

the number of objects that expire at this tick call.

For the lazy collection strategy we expect the opposite effect. The tick calls process only one descriptors after expiring a set of descriptors. This is a constant time operation with respect to the number of expired descriptors or objects. Every subsequent refresh call expires at most one descriptor. As a consequence, the refresh calls also run in constant time with respect to the number of expired descriptors or objects.

We perform detailed experiments for scalability in terms of the number of threads and the heap size for both, the lazy and eager collection strategy.

6.4.2. Number of Threads

We start with an experimental setup where we change the number of mutators that work with LIBSCM. We run the experiment on the hardware platform described in Table 6.1 which supports 24 concurrent threads. We do not want to fully utilize the system but make sure that every thread may run without preemption. Therefore we set the maximum number of concurrent threads to 12 and repeat the experiment starting from 1 up to 12 threads. The remaining factors we leave unchanged for every run. We repeat the whole experiment with different seed values for ACDC until we reach 95% confidence within 10% of the arithmetic mean value. A summary of the ACDC settings is given in Table 6.4. An explanation of the ACDC settings is given in Section 5.3.1. The LIBSCM settings are the default values defined in scm-desc.h and stm.h. We discuss the experiment using both lazy and eager collection.

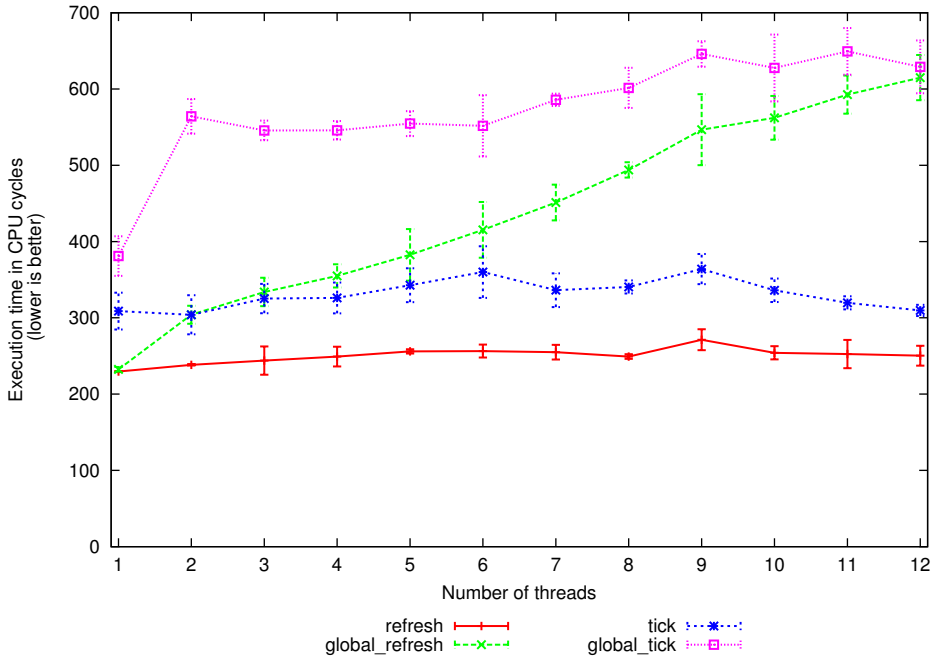

Figure 6.5.: Scalability of LIBSCM for an increasing number of threads using the lazy collection strategy. The graph shows the arithmetic mean execution time including a 95% confidence interval in CPU cycles.

Expectations

We measure the execution time of all LIBSCM API calls. Concerning the number of threads, the implementation of LIBSCM is designed to support constant time execution for all operations. Thus we expect the execution time to be the same for every number of threads that use LIBSCM. However, the experimental design encourages a detailed examination of the scalability of `global_refresh`.

Results

In Figure 6.5 we see the result of the scalability experiment using lazy collection. On the x-axis we change the number of threads from 1 up to 12. The response on the y-axis is the execution time of each API call measured in CPU cycles. The red line shows the execution time of the `refresh` operation. We see a constant behavior for all factor levels. The green dashed line is more interesting. It shows the average execution time of `global_refresh`. Here we see an increasing execution time with the number of

threads although the operations executed in this function do not change with the number of threads. The reason is contention on the descriptor count of shared objects. Two or more threads that increment or decrement the descriptor count increase the chance of cache misses because the cache coherence forces modified cache lines to be written back to memory before another thread running on another core can update it. We used OProfile [29] to measure the cache misses that occurred at the atomic increments and decrements of the descriptor count. When we increased the number of threads we observed an increasing number of cache misses at this locations.

A similar behavior can be observed for the tick calls. The blue dashed line depicts the average execution time of `tick` and the pink dotted graph shows the execution time of `global_tick`. Again, since `tick` is a thread local operation, it shows scalable performance with an increasing number of threads at about 300 CPU cycles on average whereas `global_tick` shows increasing execution time when adding more workers to the systems.

The most significant impact happens between one and two threads. Two sources of overhead are responsible for that. Firstly, the global time variable is not shared and furthermore a `global_tick` call is usually preceded by a `global_refresh` call. Thereby the global time variable is already in the cache and an increment on this memory location does not affect other caches. Secondly, `global_tick` contains two branches which always evaluate to true in case of only one thread. This supports branch prediction and prevents from pipeline stalls. Of course, the cache locality adds the greater portion of overhead to the `global_tick` than branch prediction.

For more than two threads the execution time of `global_tick` is slightly increasing and even constant for fewer threads. The reason for the better scalability than `global_refresh` is that only one thread actually advances global time. With more threads added to the system the likelihood for one thread to advance the global time decreases. This mitigates the penalty for the cache misses. Still, we consider `global_tick` to haven an execution time proportional to the number of threads although it scales much better than `global_refresh`.

Figure 6.6 shows us the same scalability experiment using the eager collection strategy. Again, the `refresh` operation stays constant but runs in shorter time than using the lazy strategy because it only creates a new descriptor now, but does not process expired descriptors. Also `global_refresh` runs faster in this configuration for a small number

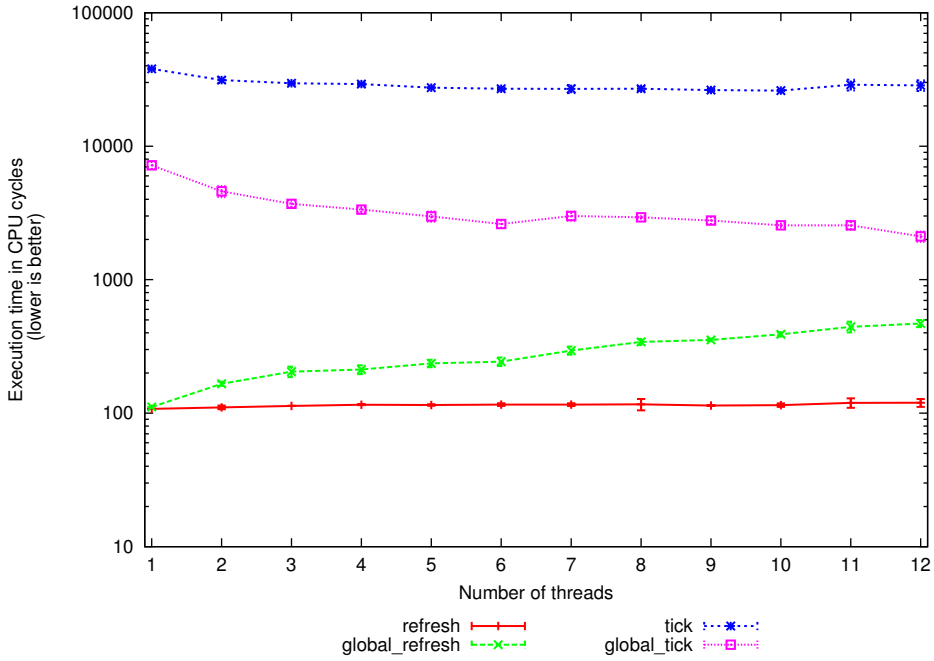

Figure 6.6.: Scalabilty of LIBSCM for an increasing number of threads using the eager collection strategy. The response is the arithmetic mean execution time of the LIBSCM API calls measured in CPU cycles.

of threads because it does not process expired descriptors either. However, for a large number of threads the execution time gets even bigger than in lazy collection mode. Running 12 threads `global_refresh` takes about 200 cycles longer than in the previous experiment running the lazy strategy. Again, this effect can be explained with additional cache misses because the descriptor count is shared among all threads. The effect is stronger than before because the total operations in `global_refresh` is smaller in eager collection mode and thereby the chance of contention is higher.

The `tick` call is now much slower because all the collection happens in this function now. Since the heap size per thread is constant in this experiment, also the response for the `tick` call is constant. In a later experiment, we will see how a changing heap size affects the `tick` operations. Another interesting effect can be observed for the `global_tick` operation. The average execution time decreases with an increasing number of threads. This phenomenon is caused by the slower global time advance if more threads enter the short-term memory system. Thereby shared objects expire later and thus the number of objects that expire at every `global_tick` call gets smaller.

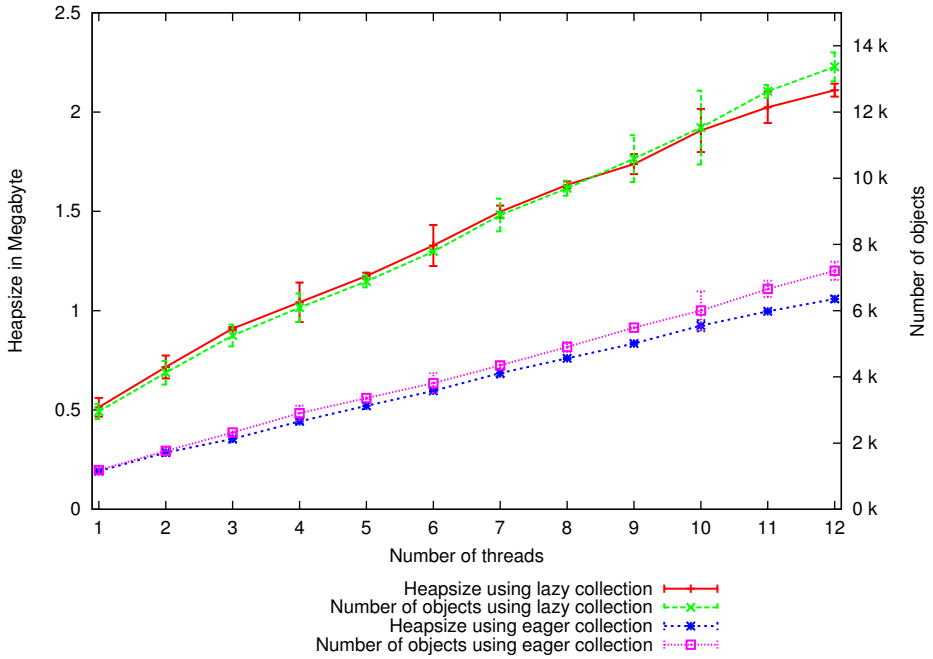

Figure 6.7.: Heap size and number of objects managed by LIBSCM for an increasing number of threads.

ACDC aims to maintain a constant per-thread memory usage. The overall memory consumption is proportional to the number of threads. In Figure 6.7 we measured the total heap size and the total number of objects managed by LIBSCM in the previous scalability experiment. On the x-axis we have the number of threads from 1 to 12. On the left y-axis we see the total size of the heap in megabytes. The right y-axis gives the total number of thousands of objects. The red line and the green line depict the lazy collection strategy for the heap size and the number of objects, respectively. The blue and pink graph give the heap size and the number of managed objects for the eager collection scheme. We can observe that eager collection yields lower memory consumption than lazy collection because of a lower number of objects since expired objects are reclaimed as soon as possible. However, both configurations show direct proportional behavior to the number of threads in the system.

Throughput

Figure 6.7 shows that the number of objects grows with the number of threads. This means that the number of LIBSCM operations grows with the number of threads.

Max object lifetime:	10
Benchmark runtime:	200 iterations
Min object size:	2^3 bytes
Max object size:	2^{12} bytes
Memory model:	short-term
Number of threads:	6
Shared objects:	yes
Share ratio:	20%

Table 6.5.: ACDC settings for the time threshold experiment

In Figures 6.5 and 6.6 we observed constant response time for all operations except `scm_global_refresh`. From this we conclude that the throughput of `scm_refresh`, `scm_tick` and `scm_global_tick` scales with the number of threads. We perform additional throughput experiments in Section 6.5.

6.4.3. Time Threshold

The next factor that shows a significant performance impact in the $2^k r$ experimental design is the time threshold. This factor controls the size of the heap that is managed by LIBSCM. We are interested in how the execution time of the API calls scales with the size of the heap. Therefore we perform an experiment where we increase the time threshold factor between the level settings 2^{12} and 2^{24} bytes and measure the per-operation execution time in CPU cycles. The remaining settings of ACDC are given in Table 6.5. We run the experiment for both, the lazy and the eager collection strategy setting. Finally, we also measure the change of the total memory consumption as we increase the time threshold.

Expectations

In the lazy collection configuration we expect constant behavior of all four LIBSCM operations. None of them depends on the number of objects or the size of the heap. We keep the number of threads constant so we do not expect any performance variation due to concurrency issues like synchronization or contention.

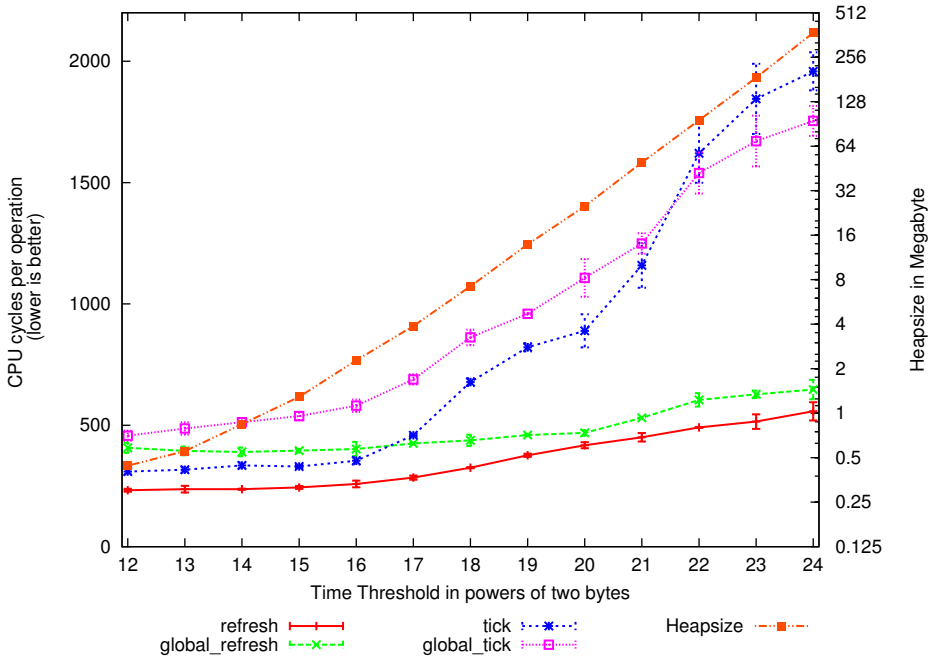

Figure 6.8.: Execution time and memory consumption for an increasing time threshold using the lazy collection strategy.

Eager collection, however, causes linear time execution of the tick operations with respect to the number of expired objects. This number is proportional to the total number of objects and therefore we expect an increasing execution time as the size of the heap grows. The levels of the time threshold are in powers of two. Consequently, we expect exponential growth of the heap size when we increase this factor.

Results

We start our evaluation using the lazy collection strategy. Figure 6.8 shows the results for this setup. The x-axis gives the time threshold in powers of two. On the left y-axis we can see the execution time of the LIBSCM operations and the right y-axis shows the size of the heap in megabytes. The heapsize is given on a logarithmic scale and the orange graph shows the growth of the heap with an increasing time threshold.

The red line shows the execution time of `scm_refresh`. Unlike our expectations it depends on the time threshold and grows from 250 cycles up to about 500 cycles. This can be explained with memory locality issues based on the size of the largest CPU cache (5 MB L3 per CPU). The `scm_refresh` operation shows constant execution time up

to a time threshold of 2^{16} and then starts to linearly increase. At a time threshold of 2^{17} bytes the total heapsize exceeds the cache size and `scm_refresh` is slowed down by capacity misses.

The green dashed line gives the execution time of `scm_global_refresh`. Again, we can observe a slight increase of the operations latency starting at about the same time threshold as `scm_refresh`. However, the execution time grows slower because ACDC creates four times more local objects than shared objects.

For time thresholds smaller than 2^{21} bytes, the `scm_tick` and `scm_global_tick` operations show a similar behavior. However, the growth is stronger than for the refresh operations because the tick calls perform more memory operations. At a time threshold of 2^{21}, the execution time of the tick calls increases even more because at this point, the total heap size exceeds 30 megabytes which is the combined cache size of all four CPUs. Note, that `scm_global_tick` becomes a little faster than `scm_tick` for large heaps. This might be explained by the order ACDC refreshes the heap objects. Shared objects are refreshed after local objects and therefore it is possible that the descriptor pages of the globally clocked buffer are more likely to be preserved in the cache than the pages of the locally clocked buffer. However, one can observe that the confidence intervals also grow with the time threshold which makes the execution time for larger heaps less predictable.

We repeated the experiment using the eager collection strategy. The results are presented in Figure 6.9. Note that the execution time on the left y-axis is now given in a logarithmic scale. The red and the green dashed line show the execution time of `scm_refresh` and `scm_global_refresh`, respectively. They behave the same as in the lazy collection configuration but about twice as fast because no expired descriptors are processed here. The tick calls however, grow proportional to the total heap size because they process all expired objects at once.

6.5. Throughput of LIBSCM Compared to the Persistent Memory Model

The previous sections showed that the overhead of LIBSCM is small in a typical configuration but so far we have only concentrated on the response time of the LIBSCM

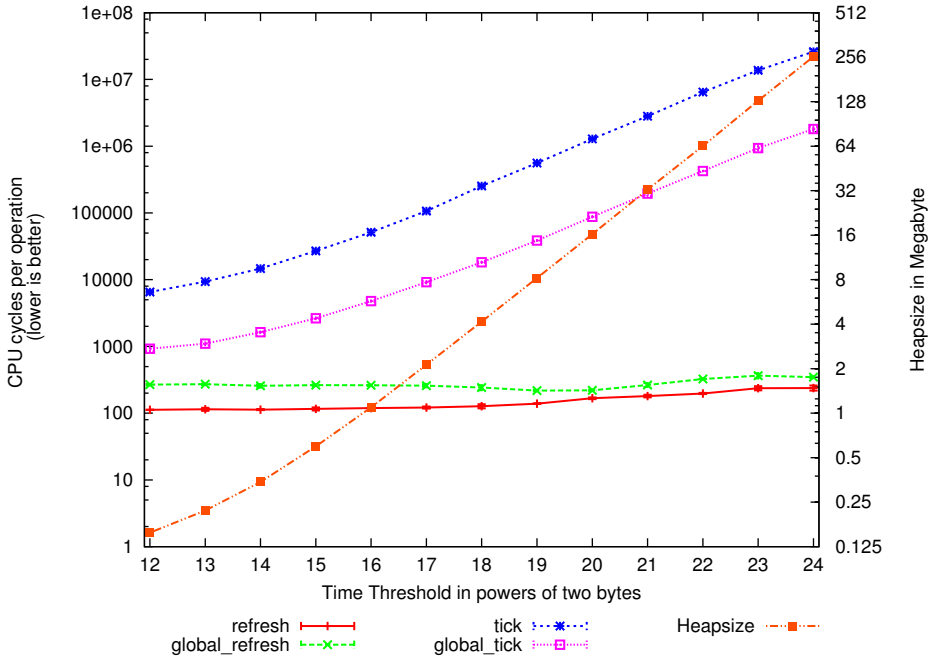

Figure 6.9.: Execution time and memory consumption for an increasing time threshold using the eager collection strategy.

operations. In this section we define a throughput metric to compare LIBSCM with the persistent memory configuration of ACDC.

The ACDC benchmark suite is designed to model a realistic allocation pattern rather than running only memory management calls. It also models a think time in between the LIBSCM and malloc/free calls which relaxes the allocator utilization. This think time is the same for all threads. The drawback is that we cannot use the number of completed LIBSCM operations of a complete run of ACDC as throughput metric. Instead we use the number of allocated bytes per second to compare the throughput of the short-term memory configuration with the persistent memory configuration of ACDC. This performance metric can be used because the allocation scheme of the persistent memory configuration of ACDC emulates the pattern of the short-term memory configuration. Furthermore, the think time is the same in both settings.

The experiments in this section also take into account the underlying allocator because we measure the total execution time of an ACDC run instead of cycles per operation. We now briefly review some characteristics of the allocator we use.

Max object lifetime:	10
Benchmark runtime:	200 iterations
Min object size:	2^3
Max object size:	2^{12}
Number of threads:	6
Shared objects:	yes
Share ratio:	10%

Table 6.6.: ACDC settings for the throughput evaluation changing the time threshold from 2^{12} to 2^{19} bytes.

In a multi-threaded application the standard C library's allocator is ptmalloc2, named after the POSIX threads library. Ptmalloc2 is based on the allocator by Doug Lea and was adapted to multiple threads by Wolfgang Gloger. The main properties of the ptmalloc2 algorithms are as follows. For large ($\geq$ 512 bytes) requests, it is a pure best-fit allocator, with ties normally decided via FIFO, i.e., least recently used. For small ($\leq$ 64 bytes by default) requests, it is a caching allocator, that maintains pools of quickly recycled chunks. In between, and for combinations of large and small requests, it does the best it can trying to meet both goals at once. For very large requests ($\geq$ 128KB by default), it relies on system memory mapping facilities, if supported. [30]

For the first throughput evaluation, we increase the time threshold and thereby the number of objects per thread. We measure the total execution time of ACDC and also the total number of allocated bytes. From this responses we calculate:

$$Throughput[MB/sec] = \frac{Memory\ allocated[MB]}{Execution\ time[s]}$$

Table 6.6 lists the settings of the experiment and Figure 6.10 presents the results. The x-axis gives the time threshold that we change from 2^{12} to 2^{19} bytes. The right y-axis shows the execution time of ACDC running for 200 iterations. The pink dotted line is the baseline, i.e., the execution time using the persistent memory model. The black and the orange graph give the execution time for the short-term memory setup using the lazy and eager collection strategy, respectively. Short-term memory introduces a slight runtime overhead for descriptor management compared to the persistent memory configuration. The collection strategy has no effect on the total execution time because it does not matter when objects are reclaimed when we measure a complete program execution.

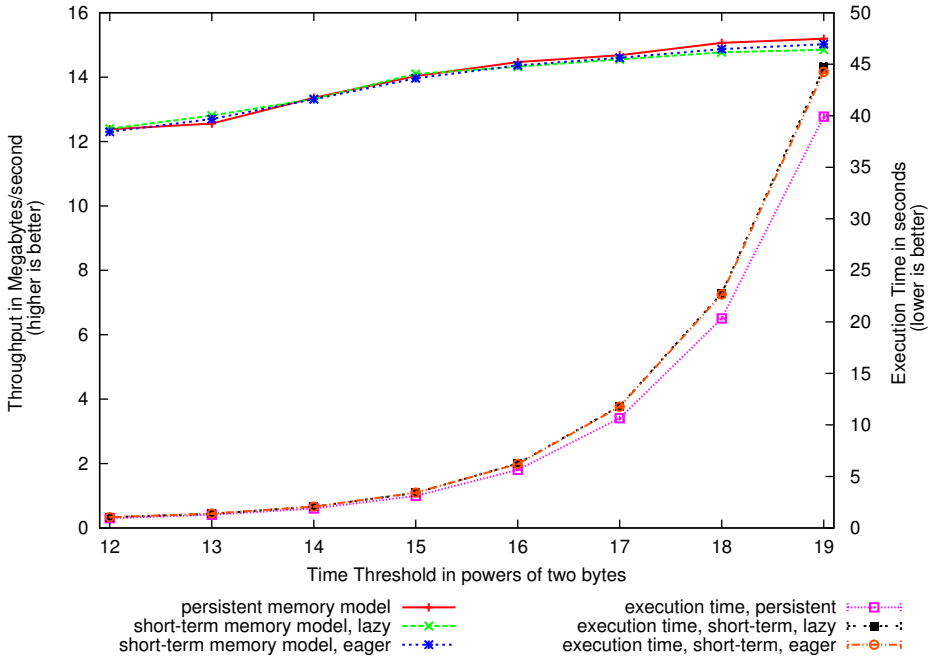

Figure 6.10.: ACDC execution time with 95% confidence intervals and allocation throughput. We compare the persistent memory model and the short-term memory model for an increasing size of the heap.

The left y-axis shows the allocation throughput for the same ACDC settings. At each time threshold setting the persistent, the lazy, and the eager short-term memory setup allocate the same amount of memory. This amount grows with the time threshold (remember Figures 6.8 and 6.9). We divide the allocated memory by the execution time to get the allocation throughput in megabytes per second. The red graph is the baseline showing the throughput for the persistent memory setup. The green and the blue lines show the throughput of ACDC running on short-term memory using lazy and eager collection, respectively. We can see that the allocation throughput is nearly the same for all setups so we can assume that the throughput of LIBSCM scales with the same rate as malloc and free.

The second throughput experiment changes the number of threads in ACDC. Again, we measure total execution time and allocated bytes to derive the allocation throughput. Table 6.7 lists the ACDC settings used for this experiment and Figure 6.11 depicts the results. On the x-axis we change the number of threads from one to eight. The right y-axis gives the execution time of ACDC. The persistent memory configuration is the baseline shown in the pink graph. The execution time grows linearly to the number of

Max object lifetime:	10
Benchmark runtime:	200 iterations
Min object size:	2^3
Max object size:	2^{12}
Time threshold:	2^{18}
Shared objects:	yes
Share ratio:	10%

Table 6.7.: ACDC settings for the throughput evaluation changing the number of threads from 1 to 8.

threads because the heap size is a linear function of the number of threads, too. Short-term memory suffers from the execution time overhead of descriptor management. The black and orange lines show the execution time of the short-term memory configuration for lazy and eager collection, respectively. Again, there is no notable difference between the two collection strategies.

The left y-axis presents the allocation throughput of the three configurations. The red line shows the throughput of the persistent memory configuration. The green and blue graphs give the throughput for the short-term memory settings in lazy and eager collection mode. Like in the previous experiment we observe only a small impact on throughput due to short-term memory. Both short-term memory setups perform competitive to malloc/free for a larger number of threads. Running less than four threads we can see that the persistent memory configuration yields better throughput results.

6.6. Approximation Overhead

The needed set of objects in ACDC depends on the random lifetime that is assigned to each of them. The persistent memory configuration is a perfect approximation because ACDC immediately frees objects after their last use, i.e., the lifetime of the objects has ended. Running the benchmark tool in short-term memory mode we also have perfect refreshing information since the lifetime of each object is exactly known. However, delayed reclamation of the lazy collection strategy will increase the memory consumption of an application using short-term memory. Also the implicit global time management will maintain objects longer than necessary to simplify the concurrent reasoning of the last use of an object.

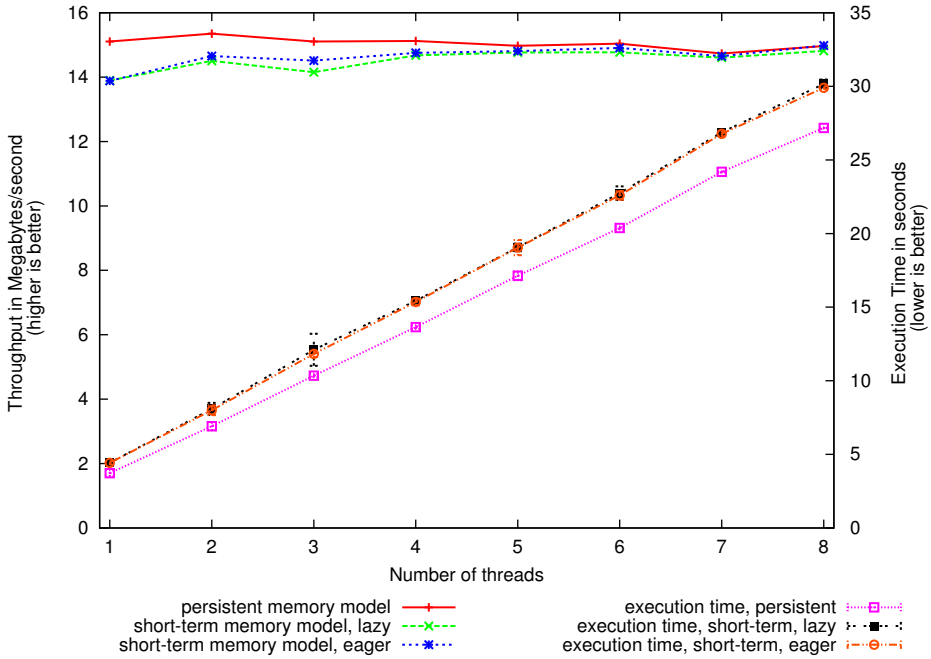

Figure 6.11.: ACDC execution time with 95% confidence intervals and allocation throughput. We compare the persistent memory model and the short-term memory model for an increasing number of threads.

In this section we describe an experiment that shows the total memory consumption of a complete execution of ACDC. The settings of the parameters are shown in Table 6.8. We used only one thread to produce the memory trace to guarantee deterministic allocation behavior, i.e., every configuration allocates the same objects. Adding more threads to the system causes a different interleaving of the allocation calls and thereby a different allocation scheme. ACDC allows the simulation of shared objects even for a single-threaded setup. We run each trace with the same initial random seed.

Max object lifetime:	10
Benchmark runtime:	100 iterations
Min object size:	2^3
Max object size:	2^{12}
Time threshold:	2^{20}
Number of threads:	1
Shared objects:	yes
Share ratio:	10%

Table 6.8.: ACDC settings for the comparison of the heap approximations.

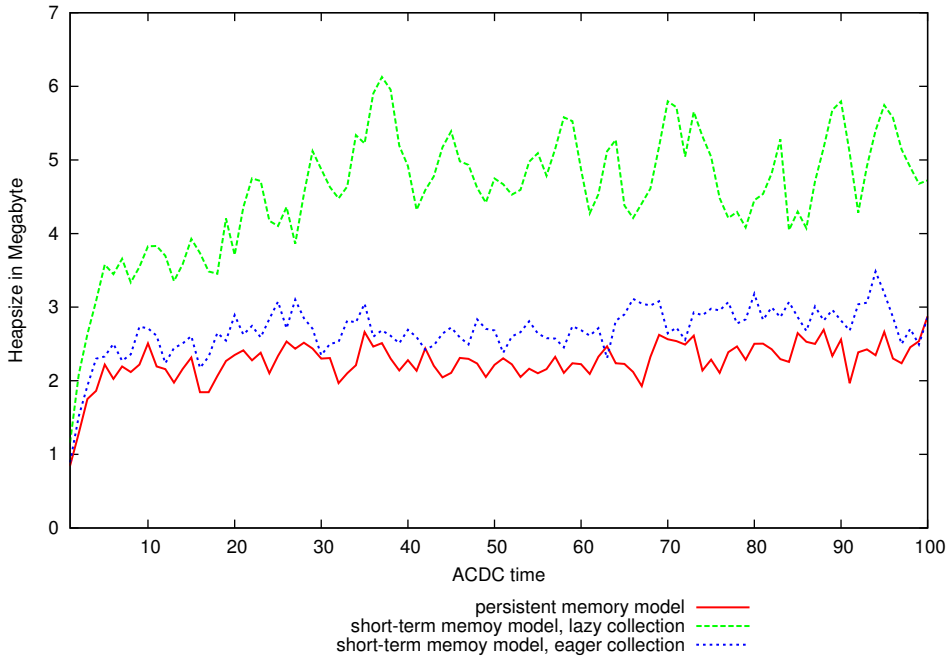

Figure 6.12.: Memory consumption.

Figure 6.12 shows a memory trace of the ACDC benchmark. On the x-axis we have the ACDC time, i.e., the number of time advances determined by the time threshold of ACDC. The y-axis gives the heap size in megabytes. Running ACDC in persistent memory mode gives the baseline of this experiment which is represented by the red line. This is a typical allocation pattern of ACDC. The blue dotted line shows the memory consumption of the short-term memory configuration using the eager collection strategy. We can observe a slightly higher demand for memory because 10% of the objects are shared so their expiration extension is set using `scm_global_refresh`. Remember that global refreshing adds two extra time units to the expiration date. This delays the reclamation of 10% of the objects. Note that using only `scm_refresh` would give the same pattern as the persistent memory setup.

The green dashed graph shows the memory consumption of the short-term memory setup using lazy collection. Here, one expired object is reclaimed at each LIBSCM operation. This results in a significant delay which causes a memory consumption of about twice the size of the eager collection strategy. However, since time advances in ACDC the size of the heap is bound.

6.7. Summary

In this chapter we performed an extensive experimental evaluation of our short-term memory implementation LIBSCM. We described a workload based on our ACDC benchmark suite and identified the factors that have the strongest impact on the execution time of the LIBSCM operations. For this factors we conducted detailed experiments to evaluate the performance metrics response time, memory consumption and throughput.

We saw that the performance of LIBSCM mostly depends on the size of the managed heap. For the eager collection strategy, this was an observation we have already expected because the tick calls perform linear to size of expired objects. For the lazy collection strategy however, we expected constant time behavior for all operations. The experiments showed the strong impact of spatial locality on all operations. Still, the runtime overhead of refresh and tick using lazy collection is small and the comparison to the persistent memory configuration showed competitive performance.

7. Conclusion

In this thesis we have presented the short-term memory model for heap management and introduced self-collecting mutators, an implementation of short-term memory for the C programming language. The fundamental difference to the persistent memory model which is implemented by malloc and free or garbage collection is the way the application returns not-needed objects to the memory management system. Without further notice, objects allocated in short-term memory expire after a finite amount of time and are eventually reclaimed by self-collecting mutators. Objects that belong to the so-called needed set of objects can be maintained by explicitly extending the expiration date of this objects. We believe that the reasoning about the needed set of objects is more intuitive in a concurrent environment than the reasoning about the not-needed set of objects. We have built a concurrent heap memory management system that is fully backwards compatible, i.e., it can be used to manage both short-term and persistent memory objects.

In order to perform an extensive performance evaluation of self-collecting mutators we have introduced a multi-threaded allocator benchmark tool called ACDC that, in our opinion, models a representative workload for multi-threaded dynamic memory allocators. The results show low, constant-time response time of the short-term memory management operations for setting expiration extensions and advancing time. The overall throughput is competitive to the persistent memory model implemented in the GLIBC allocator ptmalloc2.

7.1. Future Work

The implementation of short-term memory presented in this work uses programmer-controlled clocks to expire objects. C is an important language in the context of embed-

ded and real-time computing and an adoption of LIBSCM to support real-time clocks offers interesting application scenarios, e.g., using the worst-case execution time (WCET) of an operation as an upper lifetime bound for dynamic short-term memory objects used by this operation.

We used the default GLIBC allocator ptmalloc2 because we were primarily interested in the performance impact of the short-term memory layer on top of the allocator. However, it might be interesting to compare multiple state-of-the-art allocators to find out if this factor has an impact on the performance of self-collecting mutators.

The ACDC benchmark tool is a step towards a standard benchmark for multi-threaded explicit heap management systems. The trend for cloud computing suggests further research on the characteristics of dynamic heap objects in multi-threaded server systems, e.g., databases and application servers, in order to further increase the representativeness of ACDC.

Bibliography

[1] Martin Aigner, Andreas Haas, Christoph M. Kirsch, Michael Lippautz, Ana Sokolova, Stephanie Stroka, and Andreas Unterweger. Short-term memory for self-collecting mutators. In *Proceedings of the international symposium on Memory management*, ISMM '11, pages 99–108, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0263-0. doi: 10.1145/1993478.1993493. URL `http://doi.acm.org/10.1145/1993478.1993493`.

[2] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, November 2000. ISSN 0362-1340. doi: 10.1145/356989.357000. URL `http://doi.acm.org/10.1145/356989.357000`.

[3] Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In *Proceedings of the 1st international symposium on Memory management*, ISMM '98, pages 176–185, New York, NY, USA, 1998. ACM. ISBN 1-58113-114-3. doi: 10.1145/286860.286880. URL `http://doi.acm.org/10.1145/286860.286880`.

[4] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 187–196, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. doi: 10.1145/155090.155108. URL `http://doi.acm.org/10.1145/155090.155108`.

[5] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive c programs. *SIGPLAN Not.*, 27(12):71–80, December 1992. ISSN 0362-1340. doi: 10.1145/142181.142200. URL `http://doi.acm.org/10.1145/142181.142200`.

[6] GLIBC, the GNU C Library, 2012. URL `http://www.gnu.org/software/libc`.

[7] Free Software Foundation, Inc. The GNU C Reference Manual, 2012. URL `http://www.gnu.org/software/libc/manual`.

[8] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007. ISBN 9780136006633.

[9] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, 2nd edition*. Prentice-Hall, 1988.

[10] brk(2) - Linux man page, 2012. URL `http://linux.die.net/man/2/brk`.

[11] ISO/IEC 9899:1999, 2011. URL `http://www.iso.org/iso/catalogue_detail.htm?csnumber=29237`.

[12] malloc(3) - Linux man page, 2012. URL `http://linux.die.net/man/3/malloc`.

[13] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134000. URL `http://doi.acm.org/10.1145/1133981.1134000`.

[14] Valgrind, 2012. URL `http://valgrind.org`.

[15] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996. ISBN 0-471-94148-4.

[16] David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 50–68, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-8. doi: 10.1145/1028976.1028982. URL `http://doi.acm.org/10.1145/1028976.1028982`.

[17] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL `http://doi.acm.org/10.1145/367177.367199`.

[18] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 285–298, New York, NY, USA, 2003. ACM. ISBN 1-58113-628-5. doi: 10.1145/604131.604155. URL `http://doi.acm.org/10.1145/604131.604155`.

[19] Gabriel Kliot, Erez Petrank, and Bjarne Steensgaard. A lock-free, concurrent, and incremental stack scanning for garbage collectors. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE

'09, pages 11–20, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-375-4. doi: 10.1145/1508293.1508296. URL http://doi.acm.org/10.1145/1508293.1508296.

[20] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, September 1988. ISSN 0038-0644. doi: 10.1002/spe.4380180902. URL http://dx.doi.org/10.1002/spe.4380180902.

[21] Hans-J. Boehm and David Chase. A proposal for garbage-collector-safe c compilation, 1992.

[22] ld(1) - Linux man page, 2012. URL http://linux.die.net/man/1/ld.

[23] Ulrich Drepper. How To Write Shared Libraries, 2011. URL http://www.akkadia.org/drepper/dsohowto.pdf.

[24] 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.

[25] Gnome Object memory management, 2012. URL http://developer.gnome.org/gobject/stable/gobject-memory.html.

[26] 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. ISBN 1-58113-114-3. doi: 10.1145/286860.286864. URL http://doi.acm.org/10.1145/286860.286864.

[27] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z*, 2012.

[28] Raj Jain. *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling.* Wiley, 1991.

[29] OProfile - A System Profiler for Linux, 2012. URL http://oprofile.sourceforge.net.

[30] Ptmalloc2 Source Code, GNU C Library, malloc.c, 2012.

List of Figures

Listings

List of Algorithms

Appendix

A. Source Code of LIBSCM

The source code of LIBSCM is published under the GNU General Public license Version 2 and can be downloaded from `http://cs.uni-salzburg.at/~maigner/`.

A.1. stm.h

```
1  /*
2   * Copyright (c) 2010 Martin Aigner, Andreas Haas
3   * http://cs.uni-salzburg.at/~maigner
4   * http://cs.uni-salzburg.at/~ahaas
5   *
6   * University Salzburg, www.uni-salzburg.at
7   * Department of Computer Science, cs.uni-salzburg.at
8   */
9
10 /*
11  *  This program is free software; you can redistribute it and/or modify
12  *  it under the terms of the GNU General Public License as published by
13  *  the Free Software Foundation; either version 2 of the License, or
14  *  (at your option) any later version.
15  *
16  *  This program is distributed in the hope that it will be useful,
17  *  but WITHOUT ANY WARRANTY; without even the implied warranty of
18  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
19  *  GNU General Public License for more details.
20  *
21  *  You should have received a copy of the GNU General Public License
22  *  along with this program; if not, write to the Free Software
23  *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
24  */
25
26 #ifndef _STM_H
27 #define _STM_H
28
29 #include <stddef.h>
30
31 /*
32  * one may use the following compile time configuration for libscm.
```

```
33   * See Makefile for different configurations
34   *
35   * turn on debug messages
36   * #define SCM_DEBUG
37   *
38   * add thread id to debug messages
39   * #define SCM_MT_DEBUG
40   *
41   * print memory consumption after memory operations
42   * #define SCM_PRINTMEM
43   *
44   * print bookkeeping memory oberhead.
45   * works only in addition with SCM_PRINTMEM
46   * #define SCM_PRINTOVERHEAD
47   *
48   * print information if contention on locks happened
49   * #define SCM_PRINTLOCK
50   *
51   * the maximal expiration extension allowed on the scm_refresh calls
52   * #define SCM_MAX_EXPIRATION_EXTENSION 5
53   *
54   * the size of the descriptor pages. this should be a power of two and a
55   * multiple of sizeof(void*)
56   * #define SCM_DESCRIPTOR_PAGE_SIZE 4096
57   * the SCM_DESCRIPTOR_PAGE_SIZE results in SCM_DESCRIPTORS_PER_PAGE equal to
58   *    ((SCM_DESCRIPTOR_PAGE_SIZE - 2 * sizeof(void*))/sizeof(void*))
59   *
60   * an upper bound on the number of descriptor pages that are cached
61   * #define SCM_DECRIPTOR_PAGE_FREELIST_SIZE 10
62   *
63   * print the number of cpu cycles for each public function. Make shure to NOT
64   * enable any other debug options together with SCM_MAKE_MICROBENCHMARKS
65   * #define SCM_MAKE_MICROBENCHMARKS
66   */
67
68  /*
69   * default configuration
70   */
71  #ifndef SCM_DESCRIPTOR_PAGE_SIZE
72  #define SCM_DESCRIPTOR_PAGE_SIZE 4096
73  #endif
74
75  #ifndef SCM_MAX_EXPIRATION_EXTENSION
76  #define SCM_MAX_EXPIRATION_EXTENSION 10
77  #endif
78
79  #ifndef SCM_DECRIPTOR_PAGE_FREELIST_SIZE
80  #define SCM_DECRIPTOR_PAGE_FREELIST_SIZE 10
81  #endif
82
83  /*
84   * scm_malloc is used to allocate short term memory objects. This function
85   * can be used at compile time. Unmodified code which uses e.g. glibc's
```

```
86   * malloc can be used with linker option --wrap malloc
87   */
88  void *scm_malloc(size_t size);
89
90  /*
91   * scm_free is used to free short term memory objects with no descriptors on
92   * them e.g. permanent objects. This function can be used at compile time.
93   * Unmodified code which uses e.g. glibc's free can be used with linker
94   * option --wrap free
95   */
96  void scm_free(void *ptr);
97
98  /*
99   * scm_global_tick signals that the calling thread is ready to have the global
100  * time increased
101  */
102 void scm_global_tick(void);
103
104 /*
105  * scm_tick is used to advance the local time of the calling thread
106  */
107 void scm_tick(void);
108
109 /*
110  * scm_global_refresh adds extension time units to the expiration time of
111  * ptr and takes care that all other threads have enough time to also call
112  * global_refresh(ptr, extension)
113  */
114 void scm_global_refresh(void *ptr, unsigned int extension);
115
116 /*
117  * scm_refresh is the same as scm_global_refresh but does not take
118  * care for other threads.
119  */
120 void scm_refresh(void *ptr, unsigned int extension);
121
122 /*
123  *
124  *
125  */
126 void scm_register_thread(void);
127
128 /*
129  * scm_unregister_thread may be called just before a thread terminates.
130  * The thread's data structures are preserved for a new thread to join
131  * the short term memory system. Registration of a thread is done
132  * automatically when a thread calls *_tick or *_refresh the first time.
133  */
134 void scm_unregister_thread(void);
135
136 /*
137  * scm_block_thread may be used to signal the short term memory system that
138  * the calling thread is about to leave the system for a while e.g. because of
```

```
139  * a blocking call. During this period the system does not wait for scm_tick
140  * calls of this thread.
141  * After the thread finished the blocking state it re-joins the short term
142  * memory system using the scm_resume_thread call
143  */
144 void scm_block_thread(void);
145 void scm_resume_thread(void);
146
147 /*
148  * scm_collect may be called at any appropriate time in the program. It
149  * processes all expired descriptors of the calling thread and frees objects
150  * if their descriptor counter becomes zero.
151  */
152 void scm_collect(void);
153
154 #endif  /* _STM_H */
```

A.2. scm-desc.h

```
1  /*
2   * Copyright (c) 2010 Martin Aigner, Andreas Haas
3   * http://cs.uni-salzburg.at/~maigner
4   * http://cs.uni-salzburg.at/~ahaas
5   *
6   * University Salzburg, www.uni-salzburg.at
7   * Department of Computer Science, cs.uni-salzburg.at
8   */
9
10 /*
11  *  This program is free software; you can redistribute it and/or modify
12  *  it under the terms of the GNU General Public License as published by
13  *  the Free Software Foundation; either version 2 of the License, or
14  *  (at your option) any later version.
15  *
16  *  This program is distributed in the hope that it will be useful,
17  *  but WITHOUT ANY WARRANTY; without even the implied warranty of
18  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
19  *  GNU General Public License for more details.
20  *
21  *  You should have received a copy of the GNU General Public License
22  *  along with this program; if not, write to the Free Software
23  *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
24  */
25
26 #ifndef _SCM_DESC_H
27 #define _SCM_DESC_H
28
29 #include <stdlib.h>
30 #include <pthread.h>
31 #include "stm.h"
32 #include "arch.h"
```

```
33
34 #ifdef SCM_MT_DEBUG
35 #define printf printf("%lu: ", pthread_self());printf
36 #endif
37
38 #ifdef SCM_MAKE_MICROBENCHMARKS
39 #define MICROBENCHMARK_START \
40         allocator_overhead = 0; \
41         unsigned long long _mb_start = rdtsc();
42 #define MICROBENCHMARK_STOP unsigned long long _mb_stop = rdtsc();
43 #define MICROBENCHMARK_DURATION(_location) \
44         printf("tid: %lu microbenchmark_at_%s:\t%llu\n", \
45         pthread_self(), \
46         _location, \
47         allocator_overhead == 0 ? \
48         (_mb_stop-_mb_start) : \
49         ((_mb_stop-_mb_start) - allocator_overhead)); \
50         allocator_overhead = 0;
51 #define OVERHEAD_START unsigned long long _overhead_start = rdtsc();
52 #define OVERHEAD_STOP \
53         unsigned long long _overhead_stop = rdtsc(); \
54         allocator_overhead += (_overhead_stop - _overhead_start);
55
56 #else
57 #define MICROBENCHMARK_START //NOOP
58 #define MICROBENCHMARK_STOP //NOOP
59 #define MICROBENCHMARK_DURATION(_location) //NOOP
60 #define OVERHEAD_START //NOOP
61 #define OVERHEAD_STOP //NOOP
62 #endif
63
64 #ifndef SCM_DESCRIPTORS_PER_PAGE
65 #define SCM_DESCRIPTORS_PER_PAGE \
66         ((SCM_DESCRIPTOR_PAGE_SIZE - 2 * sizeof(void*))/sizeof(void*))
67 #endif
68
69 extern void *__real_malloc(size_t size);
70 extern void *__real_calloc(size_t nelem, size_t elsize);
71 extern void *__real_realloc(void *ptr, size_t size);
72 extern void __real_free(void *ptr);
73 extern size_t __real_malloc_usable_size(void *ptr);
74
75 /*
76  * objects allocated using libscm haven an additional object header that
77  * is added before the chunk returned by the malloc implemented in the
78  * standard library.
79  *
80  * ------------------------------- <- pointer to object_header_t
81  * | 32 bit descriptor counter dc |
82  * | 32 bit finalizer index       |
83  * ------------------------------- <- pointer to the payload data that is
84  * | payload data                 |    returned to the user
85  * ~ returned to user             ~
```

```
86   * |                                 |
87   * ----------------------------------
88   *
89   */
90  #define OBJECT_HEADER(_ptr) \
91      (object_header_t*)(_ptr - sizeof (object_header_t))
92  #define PAYLOAD_OFFSET(_o) \
93      ((void*)(_o) + sizeof(object_header_t))
94
95  typedef struct descriptor_root descriptor_root_t;
96  typedef struct object_header object_header_t;
97  typedef struct descriptor_page_list descriptor_page_list_t;
98  typedef struct expired_descriptor_page_list expired_descriptor_page_list_t;
99  typedef struct descriptor_page descriptor_page_t;
100 typedef struct descriptor_buffer descriptor_buffer_t;
101
102 struct object_header {
103     unsigned int dc; /* number of descriptors pointing here */
104     int finalizer_index; /* identifier of a finalizer function */
105 };
106
107 /*
108  * singly linked list of descriptor pages
109  */
110 struct descriptor_page_list {
111     descriptor_page_t *first;
112     descriptor_page_t *last;
113 };
114
115 /*
116  * singly linked list of expired descriptor pages
117  */
118 struct expired_descriptor_page_list {
119     descriptor_page_t *first;
120     descriptor_page_t *last;
121
122     /* index of the first descriptor in first descriptor page. This is the
123      * descriptor that will we processed upon expiration */
124     long begin;
125 };
126
127 /*
128  * statically allocate memory for the locally clocked descriptor buffers
129  * size of the locally clocked buffer is SCM_MAX_EXPIRATION_EXTENSION + 1
130  * because of the additional slots for
131  * 1. slot for the current time
132  *
133  * statically allocate memory for the globally clocked descriptor buffers
134  * size of the globally clocked buffer is SCM_MAX_EXPIRATION_EXTENSION + 3
135  * because of the additional slots for
136  *  1. slot for the current time
137  *  2. adding descriptors at current + increment + 1
138  *  3. removing descriptors from current - 1
```

```
139  *
140  * Note: both buffers allocate SCM_MAX_EXPIRATION_EXTENSION + 3 slots for
141  * page_lists but the locally clocked buffer uses only
142  * SCM_MAX_EXPIRATION_EXTENSION + 1 slots
143  */
144 struct descriptor_buffer {
145     /* for every possible expiration extension, there is an array element
146      * in "not_expired" that contains a descriptor page list where the
147      * descriptor is stored in */
148     descriptor_page_list_t not_expired[SCM_MAX_EXPIRATION_EXTENSION + 3];
149
150     /* "not_expired_length" gives the length of the "not_expired" array */
151     unsigned int not_expired_length;
152
153     /* "current_index" is a index to the descriptor_page_list in
154      * "not_expired" that will expire after the next tick.*/
155     unsigned int current_index;
156 };
157
158 /*
159  * A collection of data structures. Each thread has a pointer to
160  * a "descriptor_root" in a thread-specific data slot
161  */
162 struct descriptor_root {
163     /* "global_phase" indicates if the thread has already ticked in the actual
164      * global phase. A global phase is the interval between two increments of
165      * the global clock
166      *
167      * global_phase == global time => thread has not ticked yet
168      * global_phase == global time +1 => thread has already ticked at least once
169      */
170     long global_phase;
171
172     expired_descriptor_page_list_t list_of_expired_descriptors;
173
174     /* globally_clocked_buffer->current_index is the thread-global time */
175     descriptor_buffer_t globally_clocked_buffer;
176
177     /* locally_clocked_buffer->current_index is the thread-local time */
178     descriptor_buffer_t locally_clocked_buffer;
179
180     /* a pool of descriptor pages for re-use */
181     descriptor_page_t * descriptor_page_pool[SCM_DECRIPTOR_PAGE_FREELIST_SIZE];
182     long number_of_pooled_descriptor_pages;
183
184     /* used to build a list of terminated descriptor_roots. This is only
185      * used after the thread terminated */
186     descriptor_root_t *next;
187 };
188
189 /*
190  * a chunk of contiguous memory that holds a set of descriptors with the same
191  * expiration date.
```

```
192  */
193 struct descriptor_page {
194     descriptor_page_t* next; /* used to build a linked list*/
195     unsigned long number_of_descriptors; /* utilization of descriptors[] */
196     object_header_t * descriptors[SCM_DESCRIPTORS_PER_PAGE]; /* memory area */
197 };
198
199 descriptor_root_t *get_descriptor_root(void)
200 __attribute__((visibility("hidden")));
201
202 int expire_descriptor_if_exists(expired_descriptor_page_list_t *list)
203 __attribute__((visibility("hidden")));
204
205 void insert_descriptor(object_header_t *o,
206         descriptor_buffer_t *buffer, unsigned int expiration)
207 __attribute__((visibility("hidden")));
208
209 void expire_buffer(descriptor_buffer_t *buffer,
210         expired_descriptor_page_list_t *exp_list)
211 __attribute__((visibility("hidden")));
212
213 inline void increment_current_index(descriptor_buffer_t *buffer)
214 __attribute__((visibility("hidden")));
215
216 int run_finalizer(object_header_t *o);
217
218 #endif  /* _SCM_DESC_H */
```

A.3. stm-debug.h

```
1  /*
2   * Copyright (c) 2010 Martin Aigner, Andreas Haas
3   * http://cs.uni-salzburg.at/~maigner
4   * http://cs.uni-salzburg.at/~ahaas
5   *
6   * University Salzburg, www.uni-salzburg.at
7   * Department of Computer Science, cs.uni-salzburg.at
8   */
9
10 /*
11  *  This program is free software; you can redistribute it and/or modify
12  *  it under the terms of the GNU General Public License as published by
13  *  the Free Software Foundation; either version 2 of the License, or
14  *  (at your option) any later version.
15  *
16  *  This program is distributed in the hope that it will be useful,
17  *  but WITHOUT ANY WARRANTY; without even the implied warranty of
18  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
19  *  GNU General Public License for more details.
20  *
21  *  You should have received a copy of the GNU General Public License
```

```
22  *  along with this program; if not, write to the Free Software
23  *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
24  */
25
26 #ifndef _STM_DEBUG_H
27 #define _STM_DEBUG_H
28
29 #ifndef SCM_FINALZIER_TABLE_SIZE
30 #define SCM_FINALZIER_TABLE_SIZE 32
31 #endif /*SCM_FINALZIER_TABLE_SIZE*/
32
33 #ifdef SCM_PRINTMEM
34 #define SCM_CALCMEM
35 #endif
36
37 #ifdef SCM_PRINTOVERHEAD
38 #define SCM_CALCOVERHEAD 1
39 #endif
40
41
42 /* scm_register_finalizer is used to register a finalizer function in
43  * libscm. A function id is returned for later use. (see scm_set_finalizer)
44  *
45  * It is up to the user to design the scm_finalizer function. If
46  * scm_finalizer returns non-zero, the object will not be deallocated.
47  * libscm provides the pointer to the object as parameter of scm_finalizer.
48  */
49 int scm_register_finalizer(int(*scm_finalizer)(void*));
50
51 /*
52  * scm_set_finalizer can be used to bind a finalizer function id
53  * (returned by scm_register_finalizer) to an object (ptr).
54  * This function will be executed just before an expired object is
55  * deallocated.
56  */
57 void scm_set_finalizer(void *ptr, int scm_finalizer_id);
58
59 /*
60  * struct scm_mem_info is used to fetch information about memory
61  * consumption during runtime.
62  */
63 struct scm_mem_info {
64         unsigned long allocated;/* total allocated bytes */
65         unsigned long freed;    /* total freed bytes */
66         unsigned long overhead; /* overhead by LIBSCM */
67         unsigned long num_alloc;/* total of allocated obj. */
68         unsigned long num_freed;/* total of freed objects */
69 };
70
71 /*
72  * scm_get_mem_info is used to query the contents of a
73  * struct scm_mem_info from LIBSCM
74  */
```

```
75 void scm_get_mem_info(struct scm_mem_info *info);
76
77 #endif  /* _STM_DEBUG_H */
```

A.4. meter.h

```
1  /*
2   * Copyright (c) 2010 Martin Aigner, Andreas Haas
3   * http://cs.uni-salzburg.at/~maigner
4   * http://cs.uni-salzburg.at/~ahaas
5   *
6   * University Salzburg, www.uni-salzburg.at
7   * Department of Computer Science, cs.uni-salzburg.at
8   */
9
10 /*
11  *  This program is free software; you can redistribute it and/or modify
12  *  it under the terms of the GNU General Public License as published by
13  *  the Free Software Foundation; either version 2 of the License, or
14  *  (at your option) any later version.
15  *
16  *  This program is distributed in the hope that it will be useful,
17  *  but WITHOUT ANY WARRANTY; without even the implied warranty of
18  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
19  *  GNU General Public License for more details.
20  *
21  *  You should have received a copy of the GNU General Public License
22  *  along with this program; if not, write to the Free Software
23  *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
24  */
25
26 #ifndef _METER_H
27 #define _METER_H
28
29 #ifdef SCM_CALCOVERHEAD
30     void inc_overhead(long inc) __attribute__ ((visibility("hidden")));
31     void dec_overhead(long inc) __attribute__ ((visibility("hidden")));
32 #endif
33
34 #ifdef SCM_CALCMEM
35     void inc_freed_mem(long inc) __attribute__ ((visibility("hidden")));
36     void inc_allocated_mem(long inc) __attribute__ ((visibility("hidden")));
37     void enable_mem_meter(void) __attribute__ ((visibility("hidden")));
38     void disable_mem_meter(void) __attribute__ ((visibility("hidden")));
39     void print_memory_consumption(void) __attribute__ ((visibility("hidden")));
40 #endif
41
42 #endif  /* _METER_H */
```

A.5. arch.h

```
1  /*
2   * Copyright (c) 2010 Martin Aigner, Andreas Haas
3   * http://cs.uni-salzburg.at/~maigner
4   * http://cs.uni-salzburg.at/~ahaas
5   *
6   * University Salzburg, www.uni-salzburg.at
7   * Department of Computer Science, cs.uni-salzburg.at
8   */
9
10 /*
11  *  This program is free software; you can redistribute it and/or modify
12  *  it under the terms of the GNU General Public License as published by
13  *  the Free Software Foundation; either version 2 of the License, or
14  *  (at your option) any later version.
15  *
16  *  This program is distributed in the hope that it will be useful,
17  *  but WITHOUT ANY WARRANTY; without even the implied warranty of
18  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
19  *  GNU General Public License for more details.
20  *
21  *  You should have received a copy of the GNU General Public License
22  *  along with this program; if not, write to the Free Software
23  *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
24  */
25
26 #ifndef _ARCH_H
27 #define _ARCH_H
28
29 #define atomic_int_inc(atomic) (atomic_int_add ((atomic), 1))
30 #define atomic_int_dec_and_test(atomic)                                  \
31   (atomic_int_exchange_and_add ((atomic), -1) == 1)
32
33 static inline void toggle_bit_at_pos(int *bitmap, int pos) {
34     *bitmap = *bitmap ^ (1 << pos);
35 }
36
37
38 #if defined __i386__ || defined __x86_64__
39
40 static inline unsigned long long rdtsc(void) {
41     unsigned hi, lo;
42     asm volatile ("rdtsc" : "=a"(lo), "=d"(hi));
43     return ( (unsigned long long) lo) | (((unsigned long long) hi) << 32);
44 }
45
46 /* 32bit bit-map operations */
47
48 /* bit scan forward returns the index of the LEAST significant bit
49  * or -1 if bitmap==0 */
50 static inline int bsfl(int bitmap) {
51
```

```
52      int result;
53
54      __asm__ ("bsfl %1, %0;"       /* Bit Scan Forward                          */
55               "jnz 1f;"            /* if(ZF==1) invalid input of 0; jump to 1: */
56               "movl $-1, %0;"      /* set output to error -1                    */
57               "1:"                 /* jump label for line 2                     */
58              : "=r" (result)
59              : "g" (bitmap)
60              );
61      return result;
62  }
63
64  /* bit scan reverse returns the index of the MOST significant bit
65   * or -1 if bitmap==0 */
66  static inline int bsrl(int bitmap) {
67
68      int result;
69
70      __asm__("bsrl %1, %0;"
71              "jnz 1f;"
72              "movl $-1, %0;"
73              "1:"
74              : "=r" (result)
75              : "g" (bitmap)
76              );
77      return result;
78  }
79
80  /*code adapted from glib http://ftp.gnome.org/pub/gnome/sources/glib/2.24/
81   * g_atomic_*: atomic operations.
82   * Copyright (C) 2003 Sebastian Wilhelmi
83   * Copyright (C) 2007 Nokia Corporation
84   */
85  static inline int atomic_int_exchange_and_add(volatile int *atomic,
86          int val) {
87
88      int result;
89
90      __asm__ __volatile__("lock; xaddl %0,%1"
91              : "=r" (result), "=m" (*atomic)
92              : "0" (val), "m" (*atomic));
93      return result;
94  }
95
96  static inline void atomic_int_add(volatile int *atomic, int val) {
97      __asm__ __volatile__("lock; addl %1,%0"
98              : "=m" (*atomic)
99              : "ir" (val), "m" (*atomic));
100 }
101
102 static inline int atomic_int_compare_and_exchange(volatile int *atomic,
103         int oldval, int newval) {
104
```

```
105     int result;
106
107     __asm__ __volatile__("lock; cmpxchgl %2, %1"
108             : "=a" (result), "=m" (*atomic)
109             : "r" (newval), "m" (*atomic), "0" (oldval)
110             );
111
112     return result;
113 }
114
115 #endif /* defined __i386__ || defined __x86_64__ */
116
117 #endif  /* _ARCH_H */
```

A.6. scm-desc.c

```
1  /*
2   * Copyright (c) 2010 Martin Aigner, Andreas Haas
3   * http://cs.uni-salzburg.at/~maigner
4   * http://cs.uni-salzburg.at/~ahaas
5   *
6   * University Salzburg, www.uni-salzburg.at
7   * Department of Computer Science, cs.uni-salzburg.at
8   */
9
10  /*
11   *  This program is free software; you can redistribute it and/or modify
12   *  it under the terms of the GNU General Public License as published by
13   *  the Free Software Foundation; either version 2 of the License, or
14   *  (at your option) any later version.
15   *
16   *  This program is distributed in the hope that it will be useful,
17   *  but WITHOUT ANY WARRANTY; without even the implied warranty of
18   *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
19   *  GNU General Public License for more details.
20   *
21   *  You should have received a copy of the GNU General Public License
22   *  along with this program; if not, write to the Free Software
23   *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
24   */
25
26  #include <stdio.h>
27  #include <pthread.h>
28  #include <string.h>
29  #include <malloc.h>
30  #include "stm.h"
31  #include "scm-desc.h"
32  #include "stm-debug.h"
33  #include "meter.h"
34  #include "arch.h"
35
```

```
36 #ifdef SCM_MAKE_MICROBENCHMARKS
37 //declare a thread-local field to measure the overhead by malloc and free
38 __thread unsigned long long allocator_overhead __attribute__ ((aligned (64))) = 0;
39 #endif
40
41 static long global_time = 0;
42 static unsigned int number_of_threads = 0;
43
44 //the number of threads, that have not yet ticked in a global period
45 static unsigned int ticked_threads_countdown = 1;
46
47 //protects global_time, number_of_threads and ticked_threads_countdown
48 static pthread_spinlock_t global_time_lock;
49
50 static descriptor_root_t *terminated_descriptor_roots = NULL;
51
52 //protects the data structures of terminated threads
53 static pthread_mutex_t terminated_descriptor_roots_mutex =
54         PTHREAD_MUTEX_INITIALIZER;
55
56 //the first thread joining libscm has to initialize the thread local storage
57 static unsigned int init_threads = 0;
58 //static pthread_key_t descriptor_root_key;
59
60 //thread local reference to the descriptor root
61 __thread descriptor_root_t *descriptor_root;
62
63
64 static inline void lock_global_time();
65 static inline void unlock_global_time();
66 static inline void lock_descriptor_roots();
67 static inline void unlock_descriptor_roots();
68 static descriptor_root_t *new_descriptor_root();
69
70 void *__wrap_malloc(size_t size);
71 void *__wrap_calloc(size_t nelem, size_t elsize);
72 void *__wrap_realloc(void *ptr, size_t size);
73 void __wrap_free(void *ptr);
74 size_t __wrap_malloc_usable_size(void *ptr);
75
76 //avoid ELF interposition of exported but internally used symbols
77 //by creating static aliases //TODO: check
78 static void scm_resume_thread_internal(void)
79 __attribute__((alias("scm_resume_thread")));
80
81 static void scm_block_thread_internal(void)
82 __attribute__((alias("scm_block_thread")));
83
84 void scm_free(void *ptr)
85 __attribute__((alias("__wrap_free")));
86
87 void scm_collect_internal(void)
88 __attribute__((alias("scm_collect")));
```

```
89
90  void* scm_malloc(size_t size)
91  __attribute__((alias("__wrap_malloc")));
92
93  static void* __wrap_malloc_internal(size_t size)
94  __attribute__((alias("__wrap_malloc")));
95
96  void *__wrap_malloc(size_t size) {
97      MICROBENCHMARK_START
98
99      OVERHEAD_START
100     object_header_t *object =
101             (object_header_t*) (__real_malloc(size + sizeof (object_header_t)));
102     OVERHEAD_STOP
103
104     if (!object) {
105         perror("malloc");
106         MICROBENCHMARK_STOP
107         MICROBENCHMARK_DURATION("malloc")
108         return NULL;
109     }
110     object->dc = 0;
111     object->finalizer_index = -1;
112
113 #ifdef SCM_CALCOVERHEAD
114     inc_overhead(sizeof (object_header_t));
115 #endif
116
117 #ifdef SCM_CALCMEM
118     inc_allocated_mem(__real_malloc_usable_size(object));
119 #ifdef SCM_PRINTMEM
120     print_memory_consumption();
121 #endif
122 #endif
123
124     void *ptr = PAYLOAD_OFFSET(object);
125
126     MICROBENCHMARK_STOP
127     MICROBENCHMARK_DURATION("malloc")
128
129     return ptr;
130 }
131
132 void *__wrap_calloc(size_t nelem, size_t elsize) {
133
134     void *p = __wrap_malloc_internal(nelem * elsize);
135     //calloc returns zeroed memory by specification
136     memset(p, '\0', nelem * elsize);
137     return p;
138 }
139
140 void *__wrap_realloc(void *ptr, size_t size) {
141     MICROBENCHMARK_START
```

```
142
143     if (ptr == NULL) return __wrap_malloc_internal(size);
144     //else: create new object
145     OVERHEAD_START
146     object_header_t *new_object =
147             (object_header_t*) __real_malloc(size + sizeof (object_header_t));
148     OVERHEAD_STOP
149
150     if (!new_object) {
151         perror("realloc");
152         MICROBENCHMARK_STOP
153         MICROBENCHMARK_DURATION("realloc")
154         return NULL;
155     }
156     new_object->dc = 0;
157     new_object->finalizer_index = -1;
158
159 #ifdef SCM_CALCOVERHEAD
160     inc_overhead(sizeof (object_header_t));
161 #endif
162
163     //get the minimum of the old size and the new size
164     size_t old_object_size =
165             __real_malloc_usable_size(OBJECT_HEADER(ptr))
166             - sizeof (object_header_t);
167     size_t lesser_object_size;
168     if (old_object_size >= size) {
169         lesser_object_size = size;
170     } else {
171         lesser_object_size = old_object_size;
172     }
173
174     object_header_t *old_object = OBJECT_HEADER(ptr);
175     //copy payload bytes 0..(lesser_size-1) from the old object to the new one
176     memcpy(PAYLOAD_OFFSET(new_object),
177             PAYLOAD_OFFSET(old_object),
178             lesser_object_size);
179
180     if (old_object->dc == 0) {
181         //if the old object has no descriptors, we can free it
182 #ifdef SCM_CALCMEM
183         inc_freed_mem(__real_malloc_usable_size(old_object));
184 #endif
185
186 #ifdef SCM_CALCOVERHEAD
187         dec_overhead(sizeof (object_header_t));
188 #endif
189         OVERHEAD_START
190         __real_free(old_object);
191         OVERHEAD_STOP
192     } //else: the old object will be freed later due to expiration
193
194 #ifdef SCM_CALCMEM
```

```
195     inc_allocated_mem(__real_malloc_usable_size(new_object));
196 #ifdef SCM_PRINTMEM
197     print_memory_consumption();
198 #endif
199 #endif
200
201     void *new_ptr = PAYLOAD_OFFSET(new_object);
202
203     MICROBENCHMARK_STOP
204     MICROBENCHMARK_DURATION("realloc")
205
206     return new_ptr;
207 }
208
209 void __wrap_free(void *ptr) {
210     MICROBENCHMARK_START
211
212     if (ptr == NULL) {
213         MICROBENCHMARK_STOP
214         MICROBENCHMARK_DURATION("free")
215         return;
216     }
217
218     object_header_t *object = OBJECT_HEADER(ptr);
219
220     if (object->dc == 0) {
221 #ifdef SCM_CALCOVERHEAD
222         dec_overhead(sizeof (object_header_t));
223 #endif
224 #ifdef SCM_CALCMEM
225         inc_freed_mem(__real_malloc_usable_size(object));
226 #endif
227         OVERHEAD_START
228         __real_free(object);
229         OVERHEAD_STOP
230     }
231
232     MICROBENCHMARK_STOP
233     MICROBENCHMARK_DURATION("free")
234 }
235
236 size_t __wrap_malloc_usable_size(void *ptr) {
237     MICROBENCHMARK_START
238
239     object_header_t *object = OBJECT_HEADER(ptr);
240     size_t sz = __real_malloc_usable_size(object) - sizeof (object_header_t);
241
242     MICROBENCHMARK_STOP
243     MICROBENCHMARK_DURATION("malloc_usable_size")
244     return sz;
245 }
246
247 void scm_tick(void) {
```

```
248     MICROBENCHMARK_START
249
250     descriptor_root_t *dr = get_descriptor_root();
251
252     //make local time progress
253     //current_index is equal to the so-called thread-local time
254     increment_current_index(&dr->locally_clocked_buffer);
255
256     //expire_buffer operates on current_index - 1, so it is called after
257     //we incremented the current_index of the locally_clocked_buffer
258     expire_buffer(&dr->locally_clocked_buffer,
259             &dr->list_of_expired_descriptors);
260
261
262 #ifndef SCM_EAGER_COLLECTION
263     //we also process expired descriptors at tick
264     //to get a cyclic allocation/free scheme. this is optional
265     expire_descriptor_if_exists(&dr->list_of_expired_descriptors);
266 #else
267     scm_collect_internal();
268 #endif
269
270 #ifdef SCM_PRINTMEM
271     print_memory_consumption();
272 #endif
273
274     MICROBENCHMARK_STOP
275     MICROBENCHMARK_DURATION("scm_tick")
276 }
277
278 void scm_global_tick(void) {
279     MICROBENCHMARK_START
280     descriptor_root_t *dr = get_descriptor_root();
281
282 #ifdef SCM_DEBUG
283     printf("scm_global_tick GT: %lu GP: %lu #T:%d ttc:%d\n",
284             global_time, descriptor_root->global_phase,
285             number_of_threads, ticked_threads_countdown);
286 #endif
287
288     if (global_time == dr->global_phase) {
289         //each thread must expire its own globally clocked buffer,
290         //but can only do so on its first tick after the last global
291         //time advance
292
293         //my first tick in this global period
294         dr->global_phase++;
295
296         //current_index is equal to the so-called thread-global time
297         increment_current_index(&dr->globally_clocked_buffer);
298
299         //expire_buffer operates on current_index - 1, so it is called after
300         //we incremented the current_index of the globally_clocked_buffer
```

```
301         expire_buffer(&dr->globally_clocked_buffer,
302                 &dr->list_of_expired_descriptors);
303
304         if (atomic_int_dec_and_test((int*) & ticked_threads_countdown)) {
305             // we are the last thread to tick in this global phase
306
307             lock_global_time();
308
309             //reset the ticked_threads_countdown
310             ticked_threads_countdown = number_of_threads;
311
312             //assert: descriptor_root->global_phase == global_time + 1
313             global_time++;
314             //printf("GLOBAL TIME ADVANCE %lu\n", global_time);
315
316             unlock_global_time();
317
318         } //else global time does not advance, other threads have to do a
319         //global_tick
320
321     } //else: we already ticked in this global_phase
322     // each thread can only do a global_tick once per global phase
323
324
325 #ifndef SCM_EAGER_COLLECTION
326     //we also process expired descriptors at tick
327     //to get a cyclic allocation/free scheme. this is optional
328     expire_descriptor_if_exists(&dr->list_of_expired_descriptors);
329 #else
330     scm_collect_internal();
331 #endif
332
333 #ifdef SCM_PRINTMEM
334     print_memory_consumption();
335 #endif
336     MICROBENCHMARK_STOP
337     MICROBENCHMARK_DURATION("scm_global_tick")
338 }
339
340 void scm_collect(void) {
341     //MICROBENCHMARK_START
342
343     descriptor_root_t *dr = get_descriptor_root();
344
345     while (expire_descriptor_if_exists(
346             &dr->list_of_expired_descriptors));
347
348     //MICROBENCHMARK_STOP
349     //MICROBENCHMARK_DURATION("scm_collect")
350 }
351
352 static inline unsigned int check_extension(unsigned int given_extension) {
353     if (given_extension > SCM_MAX_EXPIRATION_EXTENSION) {
```

```
354 #ifdef SCM_DEBUG
355         printf("violation of SCM_MAX_EXPIRATION_EXTENT\n");
356 #endif
357         return SCM_MAX_EXPIRATION_EXTENSION;
358     } else {
359         return given_extension;
360     }
361 }
362
363 void scm_global_refresh(void *ptr, unsigned int extension) {
364     MICROBENCHMARK_START
365 #ifdef SCM_DEBUG
366             printf("scm_global_refresh(%lx, %d)\n", (unsigned long) ptr, extension);
367 #endif
368
369     descriptor_root_t *dr = get_descriptor_root();
370     object_header_t *object = OBJECT_HEADER(ptr);
371
372     extension = check_extension(extension);
373
374     atomic_int_inc((int*) & object->dc);
375     insert_descriptor(object,
376             &dr->globally_clocked_buffer, extension + 2);
377
378 #ifndef SCM_EAGER_COLLECTION
379     expire_descriptor_if_exists(&dr->list_of_expired_descriptors);
380 #else
381     //do nothing. expired descriptors were already collected at last tick
382 #endif
383
384 #ifdef SCM_PRINTMEM
385     print_memory_consumption();
386 #endif
387     MICROBENCHMARK_STOP
388     MICROBENCHMARK_DURATION("scm_global_refresh")
389 }
390
391 void scm_refresh(void *ptr, unsigned int extension) {
392     MICROBENCHMARK_START
393 #ifdef SCM_DEBUG
394             printf("scm_refresh(%lx, %d)\n", (unsigned long) ptr, extension);
395 #endif
396
397     descriptor_root_t *dr = get_descriptor_root();
398     object_header_t *object = OBJECT_HEADER(ptr);
399
400     extension = check_extension(extension);
401
402     atomic_int_inc((int*) & object->dc);
403     insert_descriptor(object,
404             &dr->locally_clocked_buffer, extension);
405
406 #ifndef SCM_EAGER_COLLECTION
```

```
407     expire_descriptor_if_exists(&dr->list_of_expired_descriptors);
408 #else
409     //do nothing. expired descriptors were already collected at last tick
410 #endif
411
412 #ifdef SCM_PRINTMEM
413     print_memory_consumption();
414 #endif
415     MICROBENCHMARK_STOP
416     MICROBENCHMARK_DURATION("scm_refresh")
417 }
418
419 /*
420  * get_descriptor_root is called by operations that need to access the thread
421  * local meta data structures.
422  */
423 inline descriptor_root_t *get_descriptor_root() {
424     //void *ptr = pthread_getspecific(descriptor_root_key);
425     //return (descriptor_root_t*) ptr;
426     return descriptor_root;
427 }
428
429 /*
430  * scm_register_thread is called on a thread when it operates the first time
431  * in libscm. The thread data structures are created or reused from previously
432  * terminated threads.
433  */
434 void scm_register_thread() {
435     //descriptor_root_t *descriptor_root;
436
437     lock_descriptor_roots();
438
439     if (init_threads == 0) {
440         //initialize thread local data key and the global_time_lock spinlock
441         //when the first thread is registered
442         //pthread_key_create(&descriptor_root_key, NULL);
443         pthread_spin_init(&global_time_lock, PTHREAD_PROCESS_PRIVATE);
444         init_threads = 1;
445     }
446
447     if (terminated_descriptor_roots != NULL) {
448         descriptor_root = terminated_descriptor_roots;
449         terminated_descriptor_roots = terminated_descriptor_roots->next;
450     } else {
451         descriptor_root = new_descriptor_root();
452     }
453
454     unlock_descriptor_roots();
455
456     //pthread_setspecific(descriptor_root_key, descriptor_root);
457
458     //assert: if descriptor_root belonged to a terminated thread,
459     //block_thread was invoked on this thread
```

```
460     scm_resume_thread_internal();
461
462 }
463
464 /*
465  * scm_unregister_thread is called upon termination of a thread. The thread
466  * leaves the system and passes its data structures in a pool to be reused
467  * by other threads upon creation.
468  */
469 void scm_unregister_thread() {
470     MICROBENCHMARK_START
471
472     scm_block_thread_internal();
473
474     descriptor_root_t *dr = get_descriptor_root();
475
476     lock_descriptor_roots();
477
478     dr->next = terminated_descriptor_roots;
479     terminated_descriptor_roots = dr;
480
481     unlock_descriptor_roots();
482
483     MICROBENCHMARK_STOP
484     MICROBENCHMARK_DURATION("scm_unregister_thread")
485 }
486
487 /*
488  * scm_block_thread is called when a thread blocks to notify the system about it
489  */
490 void scm_block_thread() {
491     MICROBENCHMARK_START
492
493     descriptor_root_t *dr = get_descriptor_root();
494
495     //assert: we do not have the descriptor_roots lock
496     lock_global_time();
497     number_of_threads--;
498
499     if (global_time == dr->global_phase) {
500         //we have not ticked in this global period
501
502         //decrement ticked_threads_countdown so other threads don't have to wait
503         if (atomic_int_dec_and_test((int*) & ticked_threads_countdown)) {
504             //we are the last thread to tick and therefore need to tick globally
505             if (number_of_threads == 0) {
506                 ticked_threads_countdown = 1;
507             } else {
508                 ticked_threads_countdown = number_of_threads;
509             }
510             global_time++;
511         } else {
512             //there are other threads to tick before global time advances
```

```
513             }
514         } else {
515             //we have already ticked globally in this global phase.
516         }
517         unlock_global_time();
518 
519         MICROBENCHMARK_STOP
520         MICROBENCHMARK_DURATION("scm_block_thread")
521     }
522 
523     /*
524      * scm_resume_thread is called when a thread returns from blocking state to
525      * notify the system about it.
526      */
527     void scm_resume_thread() {
528         MICROBENCHMARK_START
529 
530         descriptor_root_t *dr = get_descriptor_root();
531 
532         //assert: we do not have the descriptor_roots lock
533         lock_global_time();
534 
535         if (number_of_threads == 0) {
536             /* if this is the first thread to resume/register,
537              * then we have to tick to make
538              * global progress, unless another thread registers
539              * assert: ticked_threads_countdown == 1
540              */
541             dr->global_phase = global_time;
542         } else {
543             //else: we do not tick globally in the current global period
544             //to avoid decrement of the ticked_threads_countdown
545             dr->global_phase = global_time + 1;
546         }
547         number_of_threads++;
548 
549         unlock_global_time();
550 
551         MICROBENCHMARK_STOP
552         MICROBENCHMARK_DURATION("scm_resume_thread")
553     }
554 
555     static descriptor_root_t *new_descriptor_root() {
556 
557         //allocate descriptor_root 0 initialized
558         OVERHEAD_START
559         descriptor_root_t *descriptor_root =
560                 __real_calloc(1, sizeof (descriptor_root_t));
561         OVERHEAD_STOP
562 
563     #ifdef SCM_CALC_OVERHEAD
564         inc_overhead(__real_malloc_usable_size(descriptor_root));
565     #endif
```

```
566 #ifdef SCM_CALCMEM
567     //inc_allocated_mem(__real_malloc_usable_size(descriptor_root));
568 #endif
569 
570     descriptor_root->globally_clocked_buffer.not_expired_length =
571             SCM_MAX_EXPIRATION_EXTENSION + 3;
572     descriptor_root->locally_clocked_buffer.not_expired_length =
573             SCM_MAX_EXPIRATION_EXTENSION + 1;
574 
575     return descriptor_root;
576 }
577 
578 static inline void lock_global_time() {
579 #ifdef SCM_PRINTLOCK
580     if (pthread_spin_trylock(&global_time_lock)) {
581         printf("thread %ld BLOCKS on global_time_lock\n", pthread_self());
582         pthread_spin_lock(&global_time_lock);
583     }
584 #else
585     pthread_spin_lock(&global_time_lock);
586 #endif
587 }
588 
589 static inline void unlock_global_time() {
590     pthread_spin_unlock(&global_time_lock);
591 }
592 
593 static inline void lock_descriptor_roots() {
594 #ifdef SCM_PRINTLOCK
595     if (pthread_mutex_trylock(&terminated_descriptor_roots_mutex)) {
596         printf("thread %ld BLOCKS on lock_descriptor_roots\n", pthread_self());
597         pthread_mutex_lock(&terminated_descriptor_roots_mutex);
598     }
599 #else
600     pthread_mutex_lock(&terminated_descriptor_roots_mutex);
601 #endif
602 }
603 
604 static inline void unlock_descriptor_roots() {
605     pthread_mutex_unlock(&terminated_descriptor_roots_mutex);
606 }
```

A.7. descriptor_page_list.c

```
1 /*
2  * Copyright (c) 2010 Martin Aigner, Andreas Haas
3  * http://cs.uni-salzburg.at/~maigner
4  * http://cs.uni-salzburg.at/~ahaas
5  *
6  * University Salzburg, www.uni-salzburg.at
7  * Department of Computer Science, cs.uni-salzburg.at
```

```
8   */
9
10  /*
11   *  This program is free software; you can redistribute it and/or modify
12   *  it under the terms of the GNU General Public License as published by
13   *  the Free Software Foundation; either version 2 of the License, or
14   *  (at your option) any later version.
15   *
16   *  This program is distributed in the hope that it will be useful,
17   *  but WITHOUT ANY WARRANTY; without even the implied warranty of
18   *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
19   *  GNU General Public License for more details.
20   *
21   *  You should have received a copy of the GNU General Public License
22   *  along with this program; if not, write to the Free Software
23   *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
24   */
25
26  #include <stdio.h>
27  #include <stdlib.h>
28  #include <unistd.h>
29  #include "stm.h"
30  #include "scm-desc.h"
31  #include "meter.h"
32  #include "arch.h"
33
34  #ifdef SCM_CALCMEM
35  #include <malloc.h>
36  #endif //SCM_CALCMEM
37
38  #ifdef SCM_MAKE_MICROBENCHMARKS
39  extern __thread unsigned long long allocator_overhead;
40  #endif
41
42  static descriptor_page_t *new_descriptor_page();
43  static inline void recycle_descriptor_page(descriptor_page_t *page);
44  static object_header_t *get_expired_object(
45          expired_descriptor_page_list_t *list);
46
47  static descriptor_page_t *new_descriptor_page() {
48
49      descriptor_root_t *dr = get_descriptor_root();
50      descriptor_page_t *new_page = NULL;
51
52      if (dr->number_of_pooled_descriptor_pages > 0) {
53          dr->number_of_pooled_descriptor_pages--;
54          new_page = dr->descriptor_page_pool
55                  [dr->number_of_pooled_descriptor_pages];
56      } else {
57          OVERHEAD_START
58          new_page = __real_malloc(sizeof (descriptor_page_t));
59          OVERHEAD_STOP
60
```

```
61          if (!new_page) {
62              perror("new_descriptor_page");
63              return NULL;
64          }
65
66  #ifdef SCM_CALC_OVERHEAD
67          inc_overhead(__real_malloc_usable_size(new_page));
68  #endif
69      }
70      new_page->number_of_descriptors = 0;
71      new_page->next = NULL;
72      return new_page;
73  }
74
75  static inline void recycle_descriptor_page(descriptor_page_t *page) {
76
77      descriptor_root_t *dr = get_descriptor_root();
78
79      if (dr->number_of_pooled_descriptor_pages <
80              SCM_DECRIPTOR_PAGE_FREELIST_SIZE) {
81          dr->descriptor_page_pool
82                  [dr->number_of_pooled_descriptor_pages] = page;
83          dr->number_of_pooled_descriptor_pages++;
84      } else {
85  #ifdef SCM_CALC_OVERHEAD
86          dec_overhead(__real_malloc_usable_size(page));
87  #endif
88          OVERHEAD_START
89          __real_free(page);
90          OVERHEAD_STOP
91      }
92  }
93
94  static object_header_t *get_expired_object(
95          expired_descriptor_page_list_t *list) {
96
97      //remove from the first page
98      descriptor_page_t *page = list->first;
99
100     if (page == NULL) {
101         //list is empty
102         return NULL;
103     }
104
105 #ifdef SCM_DEBUG
106     printf("list->begin: %lu, p->num_of_desc: %lu\n", list->begin,
107             page->number_of_descriptors);
108 #endif
109
110     if (list->begin == page->number_of_descriptors) {
111         //page has already been emptied
112         //free this page and proceed with next one at index 0
113         list->begin = 0;
```

```
114
115         if (list->first == list->last) {
116             //this was the last page in list
117
118             recycle_descriptor_page(list->first);
119
120             list->first = NULL;
121             list->last = NULL;
122
123             return NULL;
124         } else {
125             //there are more pages left. remove empty list->first from list
126             page = list->first->next;
127
128             recycle_descriptor_page(list->first);
129
130             list->first = page;
131         }
132     }
133
134 #ifdef SCM_DEBUG
135     if (list->begin == page->number_of_descriptors) {
136         printf("more than one empty page in list\n");
137         return NULL;
138     }
139 #endif
140
141     //fetch first descriptor from the non-empty page
142     object_header_t *expired_object = page->descriptors[list->begin];
143     list->begin++;
144     return expired_object;
145 }
146
147 /*
148  * this function returns 0 iff no more expired descriptors exist.
149  */
150 int expire_descriptor_if_exists(expired_descriptor_page_list_t *list) {
151
152     object_header_t *expired_object = get_expired_object(list);
153     if (expired_object != NULL) {
154
155         //decrement the descriptor counter of the expired object
156         if (atomic_int_dec_and_test((int*) & expired_object->dc)) {
157
158             //if the descriptor counter is now zero, run finalizer and free it
159
160             int finalizer_result = run_finalizer(expired_object);
161
162             if (finalizer_result != 0) {
163 #ifdef SCM_DEBUG
164                 printf("WARNING: finalizer returned %d\n", finalizer_result);
165                 printf("WARNING: %lx is a leak\n",
166                         (unsigned long) PAYLOAD_OFFSET(expired_object));
```

```
167 #endif
168                 return 1; //do not free the object if finalizer fails
169             }
170
171 #ifdef SCM_DEBUG
172             printf("FREE(%lx)\n",
173                     (unsigned long) PAYLOAD_OFFSET(expired_object));
174 #endif
175
176 #ifdef SCM_CALCOVERHEAD
177             dec_overhead(sizeof (object_header_t));
178 #endif
179
180 #ifdef SCM_CALCMEM
181             inc_freed_mem(__real_malloc_usable_size(expired_object));
182 #endif
183             OVERHEAD_START
184             __real_free(expired_object);
185             OVERHEAD_STOP
186             return 1;
187
188         } else {
189 #ifdef SCM_DEBUG
190             printf("decrementing␣DC==%u\n", expired_object->dc);
191 #endif
192             return 1;
193         }
194     } else {
195 #ifdef SCM_DEBUG
196         printf("no␣expired␣object␣found\n");
197 #endif
198         return 0;
199     }
200 }
201
202 void insert_descriptor(object_header_t *object, descriptor_buffer_t *buffer,
203         unsigned int expiration) {
204
205     unsigned int insert_index = (buffer->current_index + expiration) %
206             buffer->not_expired_length;
207
208     descriptor_page_list_t *list = &buffer->not_expired[insert_index];
209
210     if (list->first == NULL) {
211         list->first = new_descriptor_page();
212         list->last = list->first;
213     }
214
215     //insert in the last page
216     descriptor_page_t *page = list->last;
217
218     if (page->number_of_descriptors == SCM_DESCRIPTORS_PER_PAGE) {
219         //page is full. create new page and append to end of list
```

```
220         page = new_descriptor_page();
221         list->last->next = page;
222         list->last = page;
223     }
224
225     page->descriptors[page->number_of_descriptors] = object;
226     page->number_of_descriptors++;
227 }
228
229 /*
230  * expire buffer operates always on the current_index-1 list of the buffer
231  */
232 void expire_buffer(descriptor_buffer_t *buffer,
233         expired_descriptor_page_list_t *exp_list) {
234
235     int to_be_expired_index = buffer->current_index - 1;
236     if (to_be_expired_index < 0)
237         to_be_expired_index += buffer->not_expired_length;
238
239     descriptor_page_list_t *just_expired_page_list =
240             &buffer->not_expired[to_be_expired_index];
241
242     if (just_expired_page_list->first != NULL) {
243
244         if (just_expired_page_list->first->number_of_descriptors != 0) {
245
246             //append page_list to expired_page_list
247             if (exp_list->first == NULL) {
248                 exp_list->first = just_expired_page_list->first;
249                 exp_list->last = just_expired_page_list->last;
250                 exp_list->begin = 0;
251             } else {
252                 exp_list->last->next = just_expired_page_list->first;
253                 exp_list->last = just_expired_page_list->last;
254             }
255
256             //reset just_expired_page_list
257             just_expired_page_list->first = NULL;
258             just_expired_page_list->last = just_expired_page_list->first;
259         } else {
260             //leave empty just_expired_page_list where it is
261         }
262     } else {
263         //buffer to expire is empty
264     }
265 }
266
267 inline void increment_current_index(descriptor_buffer_t *buffer) {
268     buffer->current_index = (buffer->current_index + 1) %
269             buffer->not_expired_length;
270 }
```

A.8. meter.c

```
1  /*
2   * Copyright (c) 2010 Martin Aigner, Andreas Haas
3   * http://cs.uni-salzburg.at/~maigner
4   * http://cs.uni-salzburg.at/~ahaas
5   *
6   * University Salzburg, www.uni-salzburg.at
7   * Department of Computer Science, cs.uni-salzburg.at
8   */
9
10 /*
11  *  This program is free software; you can redistribute it and/or modify
12  *  it under the terms of the GNU General Public License as published by
13  *  the Free Software Foundation; either version 2 of the License, or
14  *  (at your option) any later version.
15  *
16  *  This program is distributed in the hope that it will be useful,
17  *  but WITHOUT ANY WARRANTY; without even the implied warranty of
18  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
19  *  GNU General Public License for more details.
20  *
21  *  You should have received a copy of the GNU General Public License
22  *  along with this program; if not, write to the Free Software
23  *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
24  */
25
26 #include <stdio.h>
27 #include <malloc.h>
28 #include <sys/time.h>
29 #include "stm.h"
30 #include "stm-debug.h"
31 #include "meter.h"
32
33
34 #ifdef SCM_CALC_OVERHEAD
35 long mem_overhead __attribute__ ((visibility("hidden"))) = 0 ;
36
37 void inc_overhead(long inc) {
38     //mem_overhead += inc;
39     __sync_add_and_fetch(&mem_overhead, inc);
40
41 }
42 void dec_overhead(long inc) {
43     //mem_overhead -= inc;
44     __sync_sub_and_fetch(&mem_overhead, inc);
45 }
46 #endif
47
48 #ifdef SCM_CALC_MEM
49 static int mem_meter_enabled = 1;
50 static long start_time = 0;
51 void enable_mem_meter() {
```

```
52      mem_meter_enabled = 1;
53  }
54  void disable_mem_meter() {
55      mem_meter_enabled = 0;
56  }
57
58  static long freed_mem = 0;
59  static long alloc_mem = 0;
60  static long num_freed = 0;
61  static long num_alloc = 0;
62
63  void inc_freed_mem(long inc) {
64      //freed_mem += inc;
65      __sync_add_and_fetch(&freed_mem, inc);
66      __sync_add_and_fetch(&num_freed, 1);
67  }
68  void inc_allocated_mem(long inc) {
69      //used_mem += inc;
70      __sync_add_and_fetch(&alloc_mem, inc);
71      __sync_add_and_fetch(&num_alloc, 1);
72  }
73
74  void print_memory_consumption() {
75
76      struct timeval t;
77      gettimeofday(&t, NULL);
78
79      long usec = t.tv_sec * 1000000 + t.tv_usec;
80
81      if (start_time == 0) {
82          start_time = usec;
83      }
84
85      struct mallinfo info = mallinfo();
86
87      if (mem_meter_enabled != 0) {
88          printf("memusage:\t%lu\t%ld\n", usec - start_time, alloc_mem - freed_mem);
89          printf("memoverhead:\t%lu\t%lu\n", usec - start_time, mem_overhead);
90          printf("mallinfo:\t%lu\t%d\n", usec - start_time, info.uordblks);
91      }
92  }
93
94  void scm_get_mem_info(struct scm_mem_info *info) {
95          info->allocated = alloc_mem;
96          info->freed = freed_mem;
97          info->overhead = mem_overhead;
98          info->num_alloc = num_alloc;
99          info->num_freed = num_freed;
100 }
101 #endif //SCM_CALCMEM
```

A.9. finalizer.c

```
1  /*
2   * Copyright (c) 2010 Martin Aigner, Andreas Haas
3   * http://cs.uni-salzburg.at/~maigner
4   * http://cs.uni-salzburg.at/~ahaas
5   *
6   * University Salzburg, www.uni-salzburg.at
7   * Department of Computer Science, cs.uni-salzburg.at
8   */
9
10 /*
11  *  This program is free software; you can redistribute it and/or modify
12  *  it under the terms of the GNU General Public License as published by
13  *  the Free Software Foundation; either version 2 of the License, or
14  *  (at your option) any later version.
15  *
16  *  This program is distributed in the hope that it will be useful,
17  *  but WITHOUT ANY WARRANTY; without even the implied warranty of
18  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
19  *  GNU General Public License for more details.
20  *
21  *  You should have received a copy of the GNU General Public License
22  *  along with this program; if not, write to the Free Software
23  *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
24  */
25
26 #include "stm-debug.h"
27 #include "arch.h"
28 #include "scm-desc.h"
29
30
31 //finalizer table contains 100 function pointers;
32 static int (*finalizer_table[SCM_FINALZIER_TABLE_SIZE])(void*);
33
34 //bump pointer on the finalizer table
35 static int finalizer_index = 0;
36
37 int scm_register_finalizer(int(*scm_finalizer)(void*)) {
38
39     int index = atomic_int_exchange_and_add(&finalizer_index, 1);
40
41     if (index >= SCM_FINALZIER_TABLE_SIZE) return -1; //error, table full
42
43     finalizer_table[index] = scm_finalizer;
44     return index;
45 }
46
47 void scm_set_finalizer(void *ptr, int scm_finalizer_id) {
48     //set function index
49     object_header_t *o = OBJECT_HEADER(ptr);
50     o->finalizer_index = scm_finalizer_id;
51 }
```

```
52
53
54 int run_finalizer(object_header_t *o) {
55
56     //INVARIANT: object o is already expired
57
58     if (o->finalizer_index == -1) return 0; //object has no finalizer
59
60     void *ptr = PAYLOAD_OFFSET(o);
61     int (*finalizer)(void*);
62     //get function pointer to objects finalizer
63     finalizer = finalizer_table[o->finalizer_index];
64
65     //run finalizer and return the result of it
66     return (*finalizer)(ptr);
67 }
```

A.10. Makefile

```
1
2 CC=gcc
3 OBJECTDIR=build
4 DISTDIR=dist
5 WRAPPER=-Wl,--wrap=malloc -Wl,--wrap=free -Wl,--wrap=calloc -Wl,--wrap=realloc -Wl,--
      wrap=malloc_usable_size
6
7 # for compile time options uncomment the corresponding line
8 # see stm.h for a description of the options
9
10 # SCM_OPTION:=$(SCM_OPTION) -DSCM_DESCRIPTOR_PAGE_SIZE=4096
11 # SCM_OPTION:=$(SCM_OPTION) -DSCM_DECRIPTOR_PAGE_FREELIST_SIZE=10
12 # SCM_OPTION:=$(SCM_OPTION) -DSCM_MAX_EXPIRATION_EXTENSION=10
13 # SCM_OPTION:=$(SCM_OPTION) -DSCM_DEBUG
14 # SCM_OPTION:=$(SCM_OPTION) -DSCM_MT_DEBUG
15 # SCM_OPTION:=$(SCM_OPTION) -DSCM_PRINTMEM
16 # SCM_OPTION:=$(SCM_OPTION) -DSCM_PRINTOVERHEAD
17 # SCM_OPTION:=$(SCM_OPTION) -DSCM_PRINTLOCK
18 # SCM_OPTION:=$(SCM_OPTION) -DSCM_MAKE_MICROBENCHMARKS
19 # SCM_OPTION:=$(SCM_OPTION) -DSCM_EAGER_COLLECTION
20
21 SOURCEFILES= \
22         stm.h \
23         stm-debug.h \
24         descriptor_page_list.c \
25         meter.h \
26         meter.c \
27         scm-desc.h \
28         scm-desc.c \
29         arch.h \
30         finalizer.c
31
```

```
32 OBJECTFILES= \
33         $(OBJECTDIR)/descriptor_page_list.o \
34         $(OBJECTDIR)/meter.o \
35         $(OBJECTDIR)/finalizer.o \
36         $(OBJECTDIR)/scm-desc.o
37
38 LDLIBSOPTIONS=-lpthread
39
40 CFLAGS=$(SCM_OPTION) -O3 -Wall -fPIC
41
42 all: libscm
43
44 $(OBJECTDIR)/descriptor_page_list.o: $(SOURCEFILES)
45         mkdir -p build
46         $(CC) -c $(CFLAGS) -o $(OBJECTDIR)/descriptor_page_list.o descriptor_page_list.
               c
47
48 $(OBJECTDIR)/meter.o: $(SOURCEFILES)
49         mkdir -p build
50         $(CC) -c $(CFLAGS) -o $(OBJECTDIR)/meter.o meter.c
51
52 $(OBJECTDIR)/finalizer.o: $(SOURCEFILES)
53         mkdir -p build
54         $(CC) -c $(CFLAGS) -o $(OBJECTDIR)/finalizer.o finalizer.c
55
56
57 $(OBJECTDIR)/scm-desc.o: $(SOURCEFILES)
58         mkdir -p build
59         $(CC) -c $(CFLAGS) -o $(OBJECTDIR)/scm-desc.o scm-desc.c
60
61
62 libscm: $(OBJECTFILES)
63         mkdir -p dist
64         $(CC) $(CFLAGS) $(WRAPPER) -shared -o $(DISTDIR)/libscm.so -fPIC $(OBJECTFILES)
               $(LDLIBSOPTIONS)
65         cp stm.h dist
66         cp stm-debug.h dist
67
68 clean:
69         rm -rf dist
70         rm -rf build
```