Short-term Memory for Self-collecting Mutators [*]

Martin Aigner, Andreas Haas, Christoph M. Kirsch,
Michael Lippautz, Ana Sokolova, Stephanie Stroka, Andreas Unterweger

University of Salzburg

firstname.lastname@cs.uni-salzburg.at

Abstract

We propose a new memory model called short-term memory for managing objects on the heap. In contrast to the traditional persistent memory model for heap management, objects in short-term memory expire after a finite amount of time, which makes deallocation unnecessary. Instead, expiration of objects may be extended, if necessary, by refreshing. We have developed a concurrent, incremental, and non-moving implementation of short-term memory for explicit refreshing called self-collecting mutators that is based on programmer-controlled time and integrated into state-of-the-art runtimes of three programming languages: C, Java, and Go. All memory management operations run in constant time without acquiring any locks modulo the underlying allocators. Our implementation does not require any additional heap management threads, hence the name. Expired objects may be collected anywhere between one at a time for maximal incrementality and all at once for maximal throughput and minimal memory consumption. The integrated systems are heap management hybrids with persistent memory as default and short-term memory as option. Our approach is fully backwards compatible. Legacy code runs without any modifications with negligible runtime overhead and constant per-object space overhead. Legacy code can be modified to take advantage of short-term memory by having some but not all objects allocated in short-term memory and managed by explicit refreshing. We study single- and multi-threaded use cases in all three languages macro-benchmarking C and Java and micro-benchmarking Go. Our results show that using short-term memory (1) simplifies heap management in a state-of-the-art H.264 encoder written in C without additional time and minor space overhead, and (2) improves, at the expense of safety, memory management throughput, latency, and space consumption by reducing the number of garbage collection runs, often even to zero, for a number of Java and Go programs.

Categories and Subject Descriptors D3.4 [*Processors*]: Memory management (garbage collection)

General Terms Algorithms, Languages, Performance

Keywords Explicit Heap Management

[*] Supported by the EU ArtistDesign Network of Excellence on Embedded Systems Design, the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23), and an Elise Richter Fellowship (Austrian Science Fund V00125).

1. Introduction

At any time instant during mutator execution, an ideal dynamic heap management distinguishes the memory objects on the heap that are still needed by the mutator in the future (dynamically live) from the memory objects that are not needed anymore (dead). Heap management is correct if the memory allocated for the objects that are in what we call the needed set of objects is always guaranteed to be maintained. Heap management is bounded if the memory allocated for the objects in the (complementary) not-needed set of objects is always eventually reclaimed by deallocation or reuse.

Traditional heap management based on explicit deallocation or garbage collection implements different approximations of the needed and not-needed sets. Explicit deallocation, if used correctly, under-approximates the not-needed set. Tracing garbage collectors over-approximate the needed set by computing the set of reachable objects, which contains the needed set if used correctly, i.e., in the absence of reachable memory leaks. Reference-counting garbage collectors under-approximate the not-needed set by computing the set of unreachable objects, which is contained in the not-needed set. The needed and not-needed sets can also be approximated at the same time by tracing and reference-counting hybrids [5].

Despite the differences in approximation techniques, heap management based on explicit deallocation or garbage collection implements the same memory model for programming mutators. Allocated memory is guaranteed to be maintained until deallocation, either explicitly, or implicitly through unreachability. We refer to this model as persistent memory model throughout the paper. In the persistent memory model, memory is persistent until further notice. Thus objects in the needed set are safe without attention whereas objects in the not-needed set require action, either by explicit deallocation or garbage collection, hence the name. The advantages and disadvantages of explicit deallocation and garbage collection are direct consequences of the memory model. Explicit deallocation is fast but creates dangling pointers through premature deallocation and memory leaks through missing deallocation. Garbage collection removes the danger of dangling pointers but introduces cost and complexity for computing unreachability, directly or indirectly, and may therefore still allow for reachable memory leaks.

We propose short-term memory as an alternative model to the persistent memory model for studying an area of dynamic heap management that is in our opinion largely unexplored, at least by using a general model explicitly. In the short-term memory model, memory allocated for an object is only guaranteed to be maintained for a finite amount of time. Here, each object has, in addition to the memory that has been allocated for it, a so-called expiration date. When the object expires, its memory may be reclaimed by deallocation or reuse. If the object is needed beyond its expiration date, it may be refreshed before it expires, extending its expiration date but only by a finite amount of time. Refreshing may be repeated arbitrarily often but does not accumulate time.

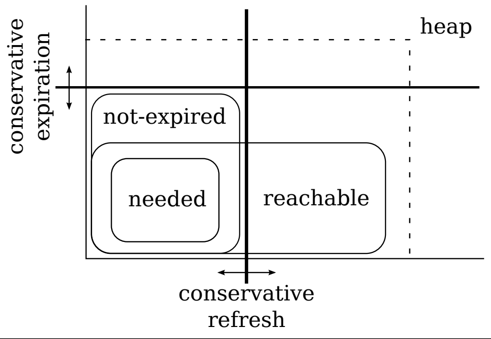

Figure 1. Approximation of the needed set by the not-expired set in the short-term memory model.

Thus, in the short-term memory model, memory is short-term until further notice. Now, objects in the not-needed set will be reclaimed without attention whereas objects in the needed set require action by refreshing.

Similar to the persistent memory model, short-term memory may be implemented by providing, in this case, refreshing information explicitly or implicitly. Note that explicitly refreshing needed objects can always be done since needed objects are always reachable, as opposed to explicitly deallocating not-needed objects, which may or may not be reachable. Moreover, unlike the persistent memory model, short-term memory induces the notion of two sets that provide structure that does not exist with persistent memory: the not-expired set of objects which have not yet expired, and the (complementary) expired set of objects. It is important to note that the two sets only exist if time is guaranteed to advance. Otherwise, all memory is permanent. As shown in Figure 1, the not-expired set is controlled by two concepts: conservative refresh of objects potentially preventing reachable but not-needed objects from expiring, and conservative expiration potentially delaying expiration of unreachable and thus not-needed objects.

Heap management in the short-term memory model is correct if the not-expired set always contains the needed set, and is bounded if the expired set always eventually contains the objects of the not-needed set, and time advances. Note that the mark phase of a mark-sweep garbage collector may readily be used to provide refreshing information that guarantees correctness by conservatively refreshing all reachable objects before time advances. However, this approach may again suffer from reachable memory leaks.

In this paper we focus on explicit refreshing. Unlike explicit deallocation, explicit refreshing only requires to know an upper bound on the lifetime of objects that may be arbitrarily large as long as it is finite. Explicit deallocation requires to know an upper bound that must be less than the time when the objects becomes unreachable. Like explicit deallocation, explicit refreshing may be done incorrectly. For example, incorrect use of explicit refreshing is missing refreshing information, resulting in memory being reclaimed too early creating dangling pointers. However, unreachable objects can never be explicitly deallocated in the persistent memory model (source of memory leaks) whereas refreshing needed and thus reachable objects is always possible. Other errors and their consequences are discussed at the beginning of Section 2.

We have developed a concurrent, incremental, and non-moving implementation of short-term memory for explicit refreshing called self-collecting mutators integrated into C as dynamic library using the ptmalloc2 allocator of glibc-2.10.1[1], the Jikes Research Virtual Machine [3] for Java, and the 6g runtime for Go[2]. The code is open source and available online [2]. In Jikes and 6g we use their mark-sweep garbage collectors because they are non-moving (and there is anyway no other choice for 6g) and do not incur runtime overhead when not running (unlike, e.g. the Jikes reference-counting collector). Note that, for brevity, we generally use the term "thread" to refer to a thread (as in C and Java) and a goroutine (as in Go, developers forgive us) alike. We also use the term "object" to refer to a memory block (as in C) and an object (as in Java or Go) alike.

All memory management operations are lock-free, i.e., do not acquire any locks, and run in constant time modulo the underlying allocators. The progress of time for expiring objects is programmer-controlled by explicit "tick" calls. Each memory management operation may collect, in addition to performing its actual function, any number of expired objects. There are no additional heap management threads in the system for this purpose, hence the name self-collecting mutators. The default collection strategy is lazy for maximal incrementality where each operation collects at most one object. Currently, the only implemented alternative is eager collection for maximal throughput and minimal memory consumption where each operation collects all objects that have expired. In this case, however, operations may not run in constant time. Alternative, more dynamic designs remain future work.

The integrated systems are heap management hybrids with persistent memory as default and short-term memory as option. We show in a number of use cases and benchmarks that using persistent memory for permanent and long-living objects while using short-term memory for short-living objects simplifies explicit heap management at the expense of slightly increased memory consumption and improves temporal and spatial performance of implicit heap management at the expense of safety. Re-establishing safety, which may require the development of adequate program analysis tools, remains future work.

The paper makes the following contributions: (1) the short-term memory model, (2) the self-collecting mutators implementation in C, Java, and Go, and (3) an experimental analysis of several macro- and micro-benchmarks.

2. Model and Implementation

For programming with short-term memory we propose to use a fully backwards-compatible approach. The default is that objects are allocated as persistent and managed by the existing heap management systems (malloc/free, GC). Then, any time after its allocation (and before its deallocation),

- an object o may be flagged as short-term and, as a consequence, managed by our heap management system, by refreshing o with a so-called expiration extension of $e \geq 0$ through a constant-time, lock-free refresh$(o,\ e)$-call.

The effect is that the object receives an expiration date $(l + e)$ where l is the current value of a software clock, which is simply an integer counter that is local to the refreshing thread. From then on the object will not be managed by the existing heap management system anymore. Instead,

- the object is now guaranteed to exist until the thread that refreshed the object advances thread-local time to $(l + e + 1)$ by incrementing the value of its thread-local clock through $(e + 1)$ constant-time, lock-free tick-calls.

After that the object is said to have expired and will be collected by our system. Objects may be flagged as short-term any time after their allocation, e.g. when an exact or at least reasonable expiration date is known. If it later turns out that the object will expire too early, the object may be refreshed again with a later expiration date. However, an object once flagged as short-term may not be returned to persistent memory anymore although an appropriate memory

[1] http://www.gnu.org/software/libc

[2] http://golang.org

management call would be easy to implement. It is rather a design choice we made because we did not find use cases.

There are a number of implications related to multiple refreshing of an object and the definition of time. We discuss them briefly right here before getting back to our use cases.

An object may be refreshed by multiple threads multiple times even in between time advance. As a result, an object may have multiple expiration dates (for one but also for different threads) since each refresh creates a new expiration date for the object. The expiration semantics is nevertheless simple. In general,

- an object in short-term memory expires when all its expiration dates have expired, and
- an expiration date has expired if its value is less than the thread-local time of the thread that created the expiration date through refreshing.

Thus multiple refreshing of an object by the same thread with the same expiration extension in between time advance has no effect other than wasting CPU time and memory (for creating and storing expiration dates). In contrast, multiple deallocations of an object with traditional explicit heap management systems (malloc/free) is an error. However, multiple refreshing across expiration, i.e., refreshing already expired objects, is also an error, which may lead to multiple deallocations of an object. The error may be detected at runtime to prevent multiple deallocations. An implementation remains future work.

Multiple refreshing of an object by different threads indicates that the object is not only shared but also short-term with respect to multiple thread-local clocks. The issue here is that refreshing and ticking requires coordination among the involved threads to prevent a shared object from expiring before all involved threads had a chance to refresh it. Coordination may either be done explicitly by the application (fast and space-efficient but difficult) or implicitly by our heap management system (easy and fast but less space-efficient), which effectively computes the notion of a global time for handling expiration dates of shared objects, see Section 2.6. For Go, there is a third option, namely for objects communicated through channels, to perform the necessary coordination implicitly and fast, and even without any space overhead.

Multiple refreshing may be avoided altogether by providing more than one clock per thread. The extreme case is that each object gets its own clock making short-term memory programming equivalent to persistent memory programming. Multiple clocks per thread are anyway interesting since, in terms of expressiveness, they are equivalent to non-zero expiration extensions if the clocks have a common base clock. Otherwise, multiple clocks are even more general, see Section 2.5.

Multiple refreshing does not make an object permanent but not advancing time does. Time advance may only be guaranteed by using real-time clocks rather than software clocks. This is an interesting topic for future work. However, even with software clocks, short-term memory may be used in real-time applications, see Section 2.4.

2.1 Single-Threaded Use Cases

We have manually ported a number of existing programs written in C, Java, and Go to short-term memory. We describe each port and argue informally about its correctness to provide intuition on the effort of using short-term memory explicitly in terms of lines of code and in terms of the difficulty of placing the needed code correctly. Table 1 shows the porting effort for each use case.

We first present single-threaded use cases: the mpg123[3] MP3 encoder and the x264 video encoder written in C, the Monte Carlo (MC) benchmark of the Grande Java Benchmark Suite [18] written in Java, and the Tree benchmark of the Computer Language Benchmarks Game[4] written in Go. In the next section we deal with multi-threaded use cases: a multi-threaded version of the x264 encoder and a simple web server (WS) written in Go.

	mpg123	x264	MC	Tree	WS
Original LoC	16043	61722	1450	104	29
Removed LoC	43	102	0	0	0
tick	1	1	1	3	1
refresh(0)	48	2	36	1	1
refresh((>0)	0	4	0	0	0
Aux LoC	0	63	0	11	0

Table 1. Original number of lines of code, number of removed lines of code, number of `tick`-calls, number of `refresh(0)`-calls, number of `refresh(>0)`-calls, and number of lines of auxiliary code, for each use case.

The following improvements are achieved by short-term memory. The C use cases logically need fewer lines of code and establishing correctness is easy. In Java and Go, unlike with garbage collection, correctness with explicit short-term memory is not guaranteed. Nevertheless, in the presented use cases establishing correctness is easy. Moreover, by using short-term memory, the Java and Go use cases improve in terms of number of garbage collection runs, total execution time, and memory consumption. The performance improvements are shown in Section 4.

We apply an informal translation scheme that helps establishing correctness. We first place a `tick`-call at the code location that marks the end of the period of the most frequent periodic behavior of the benchmark where most of the memory expires, which was easy to find in all benchmarks. Code locations where less memory expires but more frequently are also an option, which we may consider in future work. In this case more refreshing work will be required but memory consumption may be reduced. Next, we flag all objects as short-term that can expire safely at the `tick`-calls by placing `refresh`-calls right after their allocation with expiration extensions that depend on the use case and range from zero, in most cases, to some positive value, in more complex cases such as the x264 benchmark. Multiple refreshing is only used for optimizations in the x264 benchmark.

mpg123 in C. The mpg123 benchmark decodes a set of mp3 files into a set of corresponding wav files. All memory is needed just for the conversion of a single file, which means that all memory is flagged as short-term with `refresh(0)`-calls, and one `tick`-call, conveniently placed in the code where processing a file finishes, suffices to let all memory expire. This removes the need for all 43 free-calls in the original code. Note that we placed a `refresh(0)`-call after each of the 48 allocation sites (of which some may never be executed). The result is obviously correct without introducing any memory leaks.

We are aware that this use case is somewhat trivial. Still, it shows the capabilities of our programming model and how easy it can be to use short-term memory.

x264 in C. As a second use case we have ported the open source video encoder x264 [20], which implements the H.264 standard [25]. We focus on the memory management of frames and frame buffers. All other memory is irrelevant for performance since it is only allocated once. Note that our port covers only the default configuration of x264, without additional features which may require additional memory.

Figure 2 shows the dataflow of frames in the x264 encoder. In the single-threaded use case, there is only a single (main) thread

[3] http://www.mpg123.de

[4] http://shootout.alioth.debian.org

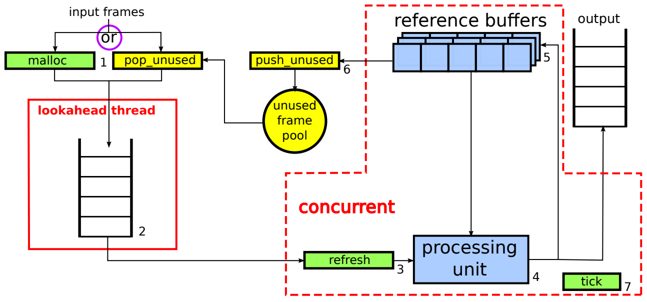

Figure 2. Dataflow of the x264 video encoder.

performing all work including frame lookahead and encoding. The numbers next to the boxes show the order in which the boxes are executed or activated. The malloc/refresh/tick boxes are introduced by the port to short-term memory. The original dataflow is as follows. Upon reading an input frame, memory is allocated for it. This frame is then encoded relative to previously encoded frames stored in so-called reference buffers. The result is output and written to the reference buffers for future reference.

One frame is an instance of a data structure consisting of multiple sub-objects, arrays, and management data. All frames require the same amount of memory. This allows for pool allocation and reuse of not-needed frames for new ones. Pool allocation may improve runtime performance and result in less memory fragmentation compared to general-purpose memory allocation. A frame can be reused (is returned to the pool) when it is removed from all frame buffers. Determining when this is the case is non-trivial since a frame may be stored in multiple reference buffers. The original implementation uses reference counting to determine when a frame can be reused. The reference counter of a frame increases when the frame is added to a frame buffer, and decreases when the frame is removed from a buffer. The frame is pushed into the unused frame pool when the reference counter becomes zero.

By porting to short-term memory we remove the need for reference counting. We have also removed pool allocation even though it can be used in combination with short-term memory. The relevant periodic behavior is given by encoding of a single frame. Therefore, we place a `tick`-call in the code where encoding a frame finishes. There are two approaches to refreshing: either refresh once with a long enough expiration extension, or else continuously refresh needed frames with an expiration extension of 1. Both approaches are implemented and tested, and described next.

The single-refresh approach leverages the fact that frames are removed from the reference buffers after a certain amount of time. The `refresh`-calls are placed such that a frame is refreshed when its encoding starts with an expiration extension that depends on the two x264 specific parameters `bframes` and `ref` which influence the size of the reference buffers. More details about the two parameters can be found in [20, 26]. We validated the calculated expiration extensions on several videos with different resolutions and length, and with different input parameters of the x264 encoder.

The expiration extensions in the single-refresh approach are conservative approximations of the lifetime of frames and thus introduce memory overhead when frames are actually needed for shorter amounts of time. In the continuous-refresh approach, we aim at removing that memory overhead by avoiding large extensions and instead refreshing all buffered frames right before the `tick`-call with an expiration extension of 1. The `refresh(1)`-calls guarantee that the frames will exist until the `tick`-call after the next frame encoding. Note that the runtime overhead of continuous refreshing is low due to the small number of frames in the frame buffers. Continuous refreshing is easy to do since it can be done right before a `tick`-call and is independent of the implementation of

Algorithm 1 Pseudo code of the Monte Carlo benchmark

```
monteCarlo(int repetitions)
{
  ResultSet results = createResultSet(repetitions);
  for(int i = 0; i < repetitions; i++)
  {
    RandomWalk walk = createRandomWalk();
    refresh(walk, 0);
    results.add(doCalculation(walk));
    tick();
  }
  evaluateResults(results);
}
```

the encoding unit, whereas the reference counting of the original implementation is done at every add- and remove-operation of the reference buffers.

Our experiments show that the effect of both refreshing approaches on throughput (total execution time) and memory management latency is negligible. Continuous refreshing is easier to use and consumes less memory than single refreshing in our benchmarks. In general, however, continuous refreshing may introduce more runtime overhead.

It is interesting to note that the memory usage pattern of the x264 encoder represents a bad case for many other memory management systems:

1. Explicit deallocation memory management is difficult to employ for x264 since it is unknown at compile time when a frame can be deallocated, which is the reason why the original implementation involves reference counting.
2. Generational garbage collectors are not suitable since at each collection of the nursery all live frames (which always exist and consist of multiple objects) need to be copied to the long-living part of the heap.
3. With region-based memory management for x264 it is difficult to form non-trivial regions (with more than one frame and not containing all frames).

Monte Carlo in Java. Algorithm 1 shows the Monte Carlo benchmark [18], which consists of a calculation loop to which we add a `tick`-call at the end. A result object is generated in every loop iteration. It is stored in a result set and exists until the end of the program. All other objects which are allocated in the calculation loop, e.g. the RandomWalk object and all objects allocated in the doCalculation method, only exist for one loop iteration and can be safely flagged as short-term.

With short-term memory all garbage collection runs are avoided by reusing the memory of expired objects. In Section 4 we show that with short-term memory every loop iteration takes nearly the same amount of time, in contrast to execution with a garbage collector where the loop iterations which are interrupted for garbage collection take significantly more time. Throughput and memory consumption also improve. Note that the same, although not with less effort, could be achieved with static preallocation, e.g. by reusing the same RandomWalk object for all loop iterations.

An interesting aspect of the Monte Carlo benchmark is that it contains a reachable memory leak. The result object contains a reference to the RandomWalk object which created it. Short-term memory fixes the memory leak by flagging the RandomWalk object as short-term, which lets the object expire at the end of the loop iteration. Clearly, the memory leak could also be fixed for garbage-collection use by deleting the reference from the result object to the RandomWalk object.

Tree Benchmark in Go. The Tree benchmark allocates one permanent tree and many short-living trees with different sizes. These trees are then dismissed after a validation step. For porting the benchmark to short-term memory we set a tick-call after every validation of a short-living tree. All but one short-living tree are allocated and validated in one loop, so two tick-calls are sufficient (one for the tree outside the loop, and one in the loop). Each loop iteration validates two trees. We therefore place a second tick-call in the loop for less memory consumption. The nodes of short-living trees are flagged as short-term right after their allocation by refresh(0)-calls. The permanent tree stays persistent and thus does not expire.

Using short-term memory in this benchmark results in higher throughput, lower near-constant latency, and less peak memory consumption, as shown in Section 4.

2.2 Multi-Threaded Use Cases

We ported two multi-threaded use cases to self-collecting mutators, a multi-threaded version of the x264 encoder written in C, and a simple web server written in Go. In the latter use case, objects are not shared among threads, which enables straightforward porting. The x264 use case includes shared objects, i.e., shared frames.

We explored two approaches of using short-term memory for shared objects. With the first, global-time approach, every thread refreshes the shared objects it needs according to the notion of a global time rather than its own thread-local time. Global time advances when the thread-local times of all threads in the system have advanced by at least one time unit. Refreshing an object according to global time thus conservatively approximates the effect achieved by refreshing an object according to the thread-local times of the threads that really need the object. The definition and calculation of global time is described in Section 2.6 in more detail. With the second, local-time approach, we use thread-local times even for shared objects. Some concurrent reasoning may then be necessary to ensure that a shared object does not expire before all threads that need it have refreshed it for the first time.

x264 in C. With the x264 encoder, there is little difference in terms of porting effort between the single-threaded and the multi-threaded use case.

The multi-threaded version of the x264 video encoder processes multiple frames in parallel. It consists of a main thread, a lookahead thread for prefetching frames, and a number of encoding threads. The lookahead thread deals with frames before they are flagged as short-term. As indicated by the area with dashed borderline in Figure 2, the encoding unit and the reference buffers exist in multiple instances, one instance per encoding thread. The encoder uses one mutex per frame for thread synchronization. We use a finalizer function to destroy such mutexes properly when collecting expired frames. Finalizers are described in Section 2.3.

The single-threaded ports of the x264 use case work for multiple threads without further modifications except for the registration of the finalizer that destroys the mutex of an expired frame. In this case, all short-term memory operations are invoked from within the main thread only. However, we also demonstrate multi-threaded short-term memory management by moving the refresh- and tick-calls into the encoding threads and then using either the global-time approach or else the local-time approach.

For the single-refresh approach, the formula calculating the lifetime of a frame can be refined using the number n of threads, i.e., the expiration extension e of a frame in the single-threaded case reduces to $e' = \lceil e/n \rceil$. It turns out that using the thread-local time approach for shared objects also works for both refreshing approaches, including the refined extensions for the single-refresh approach, since all threads are anyway synchronized by mutexes in a way that thread-local time and global time only differ by at most one time unit.

The usage and performance results of the multi-threaded use case are similar to the single-threaded use case, see Section 4.

Web Server in Go. We have ported the godoc web server to short-term memory but eventually decided to use our own simple implementation of a static yet multi-threaded web server in Go for benchmarking. The godoc web server indexes all library packages, which dominates the performance of the parts relevant to short-term memory. Our server creates a goroutine for each request, which simply serves a static web page. In the port to short-term memory, the objects storing web page content are flagged as short-term. We show with this use case that self-collecting mutators in Go maintain goroutine scalability as well as memory consumption while reducing memory management latency.

2.3 Implementation

The C, Java, and Go implementations of self-collecting mutators are based on the same algorithm and data structures, and differ only slightly in some low-level details that we point out whenever they are relevant. The implementations are available in source code [2].

Descriptors. An expiration date of a given object is represented by a descriptor, which is a pointer to the object. Descriptors representing a given (not-expired) expiration date are gathered in a descriptor list. In other words, the expiration date value represented by a descriptor is implicitly encoded by storing the descriptor in a descriptor list for this value. Note that an object may even have multiple expiration dates with the same value, which means that there may be multiple descriptors in a descriptor list pointing to the same object.

Object Header. Every object (also if persistent) is extended by a 64-bit object header that stores a descriptor counter, which is a 32-bit integer that counts, similar to a reference counter, the number of descriptors that point to the object, i.e., the number of expiration dates the object has. Incrementing and decrementing the descriptor counter of an object are the only operations that must be done atomically by atomic increment and atomic decrement-and-test instructions, respectively. All other operations involved in refreshing and expiring objects as well as advancing time are thread-local.

The remaining 32 bits of the object header are used differently in the C and Go implementations and unused in the Java implementation. For C, five of the 32 bits identify an optional user-implemented finalizer that gets invoked right before deallocation. The other 27 bits are unused. Finalizers receive a permanent and unique 5-bit identifier upon user-controlled, constant-time registration in a simple 32-entry identifier-to-finalizer table. A more dynamic service remains future work. For Go, 16 bits store the offset from the object address to its garbage collector status flag. Another 8 bits store an identifier of an internal Go size-class, needed to free an object of size smaller than 32KB. The remaining 8 bits are unused. Objects larger than 32KB actually require an additional 64-bit word in the object header for storing a pointer.

Descriptor Management. A descriptor list is a singly-linked list of descriptor pages represented by a fixed-size record containing a head and a tail pointer to the first and the last page, respectively. A descriptor page is a fixed-size record that consists of a pointer to the next page, an integer word that counts the actual number of descriptors stored in the page, and a fixed number of pointers for storing descriptors. A descriptor page is therefore properly initialized if just the first two entries are zeroed. The size m of descriptor pages is fixed at compile time. Descriptor pages are allocated cache- and page-aligned for better runtime performance. We distinguish different size configurations of m in our benchmarks. Note that using descriptor pages provides only a constant-factor, yet potentially significant, optimization over a singly-linked list of descriptors.

Given a compile-time bound n on the expiration extensions for refreshing, we use a thread-local descriptor buffer to store $n + 1$

descriptor lists in an array of size $n + 1$, which supports expiration extensions between zero and n. The buffer also stores thread-local time denoted by l. The descriptors in the buffer are interpreted against l as follows. The descriptor list containing descriptors representing an expiration date l is located at position $l \mod (n+1)$ in the buffer. Given an expiration extension $0 \leq e \leq n$, a new descriptor representing an expiration date $l + e$ will therefore be appended to the descriptor list at position $(l + e) \mod (n + 1)$. Thus the descriptors in the descriptor list at position $l \mod (n+1)$ expire when thread-local time advances.

There are three descriptor management operations: create, move-expired, and collect, which all run in constant time.

Given an object and the index $i = (l + e) \mod (n + 1)$, the create-operation stores a new descriptor, i.e., a pointer to the object, in the last descriptor page of the descriptor list at position i in the buffer, if the page is not full. Otherwise, the descriptor is stored in a new page that is allocated, either from a thread-local descriptor-page pool or, if empty, from free memory, and appended to the list.

Given the index $i = l \mod (n + 1)$ upon thread-local time advance, the move-expired-operation removes the descriptor pages from the descriptor list at position i in the buffer, if it contains at least one descriptor, and appends the pages to a thread-local descriptor list called the expired-descriptor list. Unlike the descriptor lists in a descriptor buffer, the expired-descriptor list may contain descriptors that represent different expiration dates that have, however, all expired. Moreover, the expired-descriptor list stores, in addition to head and tail pointers, an integer counter that keeps track of how many descriptors in the first descriptor page of the list have already been collected.

The collect-operation only operates on the expired-descriptor list. If the list is empty, the operation immediately returns. Otherwise, the first descriptor in the list is removed from the list, i.e., from the first descriptor page in the list. If the page becomes empty, it is removed from the list. The empty page is then returned either to the descriptor-page pool, if the pool is not full, which is determined by a compile-time bound, or to free memory by calling the underlying free routine. Then, the descriptor counter of the object to which the removed descriptor points is decremented by an atomic decrement-and-test instruction. If the counter becomes zero, the object is deallocated, again by calling the underlying free routine. Note that, in the C implementation, we check if a finalizer has been set in the object header of the object. If yes, the finalizer is invoked right before deallocation.

In summary, each thread maintains a descriptor buffer containing $(n + 1)$ descriptor lists and an integer word for storing thread-local time, an expired-descriptor list, and a bounded descriptor-page pool.

Memory Management. We now describe the memory management calls, which do not require any locking and all run in constant time modulo the underlying malloc/new/free implementations.

A malloc-call or new-call simply calls the allocation routine of the underlying system to allocate a memory block that fits the requested size plus one 64-bit word for the object header with the descriptor counter initialized to zero.

In the C implementation, a free-call invokes the underlying free routine to deallocate the given memory block but only if its descriptor counter is zero. Otherwise, it returns without deallocation. The standard calloc and realloc routines have been wrapped in a similar way. If the realloc-call is invoked on a memory block that does not fit the requested adjustment in size, a new memory block that fits is allocated. The old memory block is then deallocated but again only if its descriptor counter is zero. This approach has an important consequence: our C library can readily be linked against any existing C code and used without introducing any new memory leaks and without any modifications to the code, unless the code makes assumptions on the layout of memory management data in memory blocks. We tested this claim by successfully linking the library against Apache HTTP server-2.2.15 and executing it. Without using short-term memory, our implementations only introduce a per-object space overhead of one 64-bit word and negligible runtime overhead as shown in Section 4.

For Java and Go, we have modified the underlying garbage collectors such that all objects including all short-term memory objects are considered in computing reachability but short-term memory objects that ever had a descriptor counter greater than zero are not deallocated even when determined unreachable. Instead, the memory of expired objects is deallocated by our system to be reused later upon allocation, again into persistent memory first. Our modifications only involve a few lines of code in both cases and do not incur any runtime overhead. Similar to the C implementation, legacy Java and Go applications run without any modifications.

The refresh-call first atomically increments the descriptor counter of the given object. Then, a new descriptor pointing to the object is inserted into the descriptor buffer by the previously described create-operation.

The tick-call first increments the thread-local clock and then invokes the previously described move-expired-operation to move the just expired descriptors to the expired-descriptor list.

In principle, any memory management operation may collect expired descriptors and deallocate expired objects. Each operation may collect anywhere between one at a time (lazy) for maximal incrementality and all at once (eager) for maximal throughput and minimal memory consumption.

In our implementations, the default is lazy. Moreover, only the refresh-call and the tick-call collect by invoking the previously described collect-operation once after performing their actual function. Lazy collection at a refresh-call makes sure that the memory allocated for descriptors is bounded in the number of refresh-calls between tick-calls, similar to the memory allocated for objects.

The alternative of eager collection is only implemented in tick-calls. Interesting future work may be to study collection strategies for trading-off memory management throughput, latency, and space consumption dynamically. For example, we may choose to collect varying numbers of expired descriptors per call, and to collect expired descriptors also in other calls such as the malloc-call, new-call, and free-call, and even in code unrelated to memory management. Another option is running auxiliary threads that collect expired descriptors concurrently to the mutator.

Managing memory objects in persistent or short-term memory can now be done as follows. A malloc-call or new-call allocates persistent memory for a given object. As long as the object is not refreshed, it remains in persistent memory and thus requires, in the C implementation, explicit deallocation by a free-call. However, the first refresh-call on the object, even with an expiration extension of zero, logically transfers the object to short-term memory. Then, again in the C implementation, explicit deallocation is unnecessary and should be replaced by invoking tick-calls instead. Existing free-calls on short-term memory objects may nevertheless remain in the code as long as they are invoked before any tick-call makes the objects expire. Note that the malloc-call or new-call could also do both allocation and refresh in one step, which we nevertheless chose not to do for backwards compatibility and since reasonable expiration dates may not be known at allocation time.

Thread Management. All thread management operations for short-term memory run in constant time. A thread maintains thread-local metadata called descriptor root, which is a fixed-sized record that contains a descriptor buffer, an expired-descriptor list, a descriptor-page pool, and a descriptor-root pointer for constructing a global, lock-protected, and unbounded pool of unused descriptor roots. Memory for descriptor roots is allocated either from that pool, if not empty, or else from free memory. Descriptor roots obtained from free memory can be efficiently initialized just by allocating zeroed memory. Roots obtained from the pool do not require initialization. Descriptor roots and pages are the only two metadata types for short-term memory that require heap allocation.

Similar to the allocation of metadata storage for the underlying allocators, which is on demand upon the first invocation of a malloc-call or new-call, a descriptor root is allocated upon the first invocation of a refresh-call by a thread, effectively registering the thread with the short-term memory system. This approach minimizes the impact on scalability of threads that do not use short-term memory. Note that, by integrating the descriptor-root pool deeper into the underlying allocators, which already use a global lock to protect their metadata pools, it may be possible, as part of future work, to avoid introducing an extra global lock for protecting the descriptor-root pool. Without negative impact on scalability, all metadata storage may then be allocated upon the first invocation of a malloc-call or new-call.

When a thread terminates, its descriptor root is inserted into the descriptor-root pool for later reuse by another thread. In Java and Go, this is done transparently by the runtime system whereas, in C, a manual unregister-call is required. Interestingly, reused descriptor roots are not initialized since they may still contain uncollected, expired and even not-expired descriptors. Instead, the new thread may safely reuse a descriptor root expiring the not-expired and collecting the expired descriptors of the previous thread exactly from where they were left off. In particular, the new thread advances the root's thread-local time from where it was left off.

2.4 Real Time and Fragmentation

As shown by our Java and Go benchmarks, self-collecting mutators may, at the expense of safety, significantly decrease memory management latency by reducing the number of garbage collection runs, sometimes to zero, while even improving throughput. Real-time garbage collectors such as Metronome [4] reduce latency as well and are safe but only at a significant loss in throughput and increase in code complexity, which makes it difficult to certify them for hard real-time applications. Self-collecting mutators perform all operations in constant time and may therefore even be suitable for managing hard real-time applications if combined with a real-time allocator such as Compact-fit [11] or TLSF [17]. Lastly, real-time instead of software clocks may be used to guarantee time advance. However, using short-term memory correctly may become more difficult in this case and even require execution time analysis.

Our implementations of short-term memory are based on existing allocators unaware of short-term memory and, in particular, its effect on fragmentation. In our benchmarks fragmentation has not been an issue but it may become one in others. Addressing fragmentation in short-term memory is part of our future work.

2.5 Multiple Clocks

Self-collecting mutators use thread-local clocks and, optionally, the notion of a global clock, to expire objects. Each clock is stored in a descriptor buffer along with the descriptors that the clock expires. In principle, self-collecting mutators can readily be generalized to use a dynamic set of multiple clocks, i.e., descriptor buffers to be precise. Multiple, independent clocks are more expressive than a single clock in the sense that they may facilitate expiration of objects with different, independent lifetimes more accurately and thus decrease memory consumption. Note that advancing multiple clocks according to a common base clock is equivalent to using a single clock with non-zero expiration extensions for refreshing. An implementation of multiple clocks may require synchronization if threads share clocks, which may harm performance and scalability. Alternatively, a programming convention, as in Go for sharing objects, could require each clock to be associated with a single thread at any time. Clock ownership along with object ownership could then be passed, as in Go through channels, from one thread to another. The issue of using short-term memory in library code may be addressed similarly, e.g. by maintaining dedicated library clocks. An implementation remains future work.

2.6 Global-Time Management

We describe a simple global-time management for brevity that suffices for the use cases considered here but does not support dynamically changing sets of threads. It also does not handle blocking and faulty threads properly since global time would not advance in their presence. However, we have developed a more general global-time management based on so-called thread-global time that does support dynamically changing sets of threads including blocking and faulty threads [1].

Shared objects in short-term memory may, in addition to regular refreshing, also be refreshed by a constant-time, lock-free global-refresh-call and expire according to a synchronized notion of global time, which is advanced by a constant-time, lock-free global-tick-call. The global calls operate on a new thread-local, global-time descriptor buffer, whereas the regular local calls still operate on the existing thread-local-time descriptor buffer. The clock in the global-time descriptor buffer is not used here but represents thread-global time in the more general global-time management [1].

Global time is represented by a global integer counter called the global clock. The expiration semantics remains simple. An expiration date created by a global-refresh-call has expired if its value is less than global time. The intention is that all threads sharing an object have a chance to refresh the object before it expires without coordinating refreshing and ticking explicitly.

In addition to the global clock, global-time management also requires a global ticked-threads counter and, for each thread, a thread-local integer counter representing the global phase of the thread. The global phase determines whether the thread performed a global-tick-call since global time has last advanced. Initially, the values of the global clock and phases are zero, and the ticked-threads counter is set to the (fixed) number of threads in the system. The global-tick-call increments the global phase of the invoking thread if the global phase of the thread is equal to global time, otherwise it immediately returns. In case the global phase was incremented, the ticked-threads counter is subsequently decremented by an atomic decrement-and-test instruction. If the counter becomes zero, global time is advanced and the ticked-threads counter is reset to the (fixed) number of threads in the system, marking the beginning of a new global period. Locking is not required since only one thread can decrement the counter to zero.

A global-refresh-call sets the expiration date of an object to the current global time plus the extension plus one time unit. The additional time unit is sufficient to guarantee that the object does not expire before one full global period has elapsed, i.e., all threads have invoked the global-tick-call at least once. Therefore, the global-time descriptor buffer needs to accommodate one more descriptor list. So far, we have implemented global-time management in C. Future work may focus on optimizing memory consumption, yet probably at the expense of runtime performance.

3. Related Work

We first discuss general memory management work related to the short-term memory model and then specific work related to the design and implementation of self-collecting mutators.

3.1 Short-term Memory

Implementing short-term memory essentially requires a representation of the not-expired and expired sets as well as an algorithm that determines expiration information and time advance. The algorithm may be an offline analysis tool or an online system, as with most related work, or a programmer who provides the information manually, as with self-collecting mutators. The representation may logically implement sets to support any algorithm, as in self-collecting mutators, or more specific data structures such as stacks and buffers that are more efficient but work only for specific algorithms, as in some related work.

Stack allocation can be seen as implementing a special case of short-term memory where the representation are per-thread stacks and the algorithm maintains per-frame expiration dates and per-stack time that advances upon returns from subroutines, which facilitates constant-time allocation and deallocation of multiple objects. General refreshing is not possible.

Short-term memory is originally inspired by cyclic allocation where the representation are cyclic fixed-size per-allocation-site buffers [21]. The algorithm maintains per-buffer expiration dates set to the size of the buffer and per-buffer time that advances upon each allocation in the buffer. For example, an allocation in a three-element buffer will always receive an expiration date equal to the current time plus three, i.e., memory allocated in the buffer will be reused after three subsequent allocations in the buffer, making deallocation unnecessary. Refreshing is again not possible. Note that cyclic allocation requires properly dimensioning the buffers, which is related to the more general problem of properly refreshing objects and advancing time with short-term memory.

Region-based memory management [13, 24] can also be seen as implementing a special case of short-term memory where the representation are regions, which allow deallocating multiple objects in constant time. The algorithm always uses expiration dates equal to the current time and maintains per-region time that advances upon events determined by either an offline analysis tool [24], or online reference counting [13], or explicit deallocate-region calls [13]. General refreshing is not possible but could be done by copying objects from one region to another. Similarly, choosing the appropriate region for an object must be done at its allocation and may only be avoided by copying the object. In short-term memory, each descriptor list forms a region and the associated clock is used to free (collect) the region. In contrast to the region-based approach, refreshing allows for choosing an appropriate region at any time after allocation and changing the region for an object without copying but at the expense of freeing a region in non-constant time. In addition, multiple clocks allow an object to be in multiple regions, i.e., to be associated with different clocks.

Objective-C [16] provides autorelease pools, which are a special case of short-term memory. The representation are stacked explicitly-allocated pools for delaying object deallocation. Objects can only be added to and removed from the top pool. The algorithm always uses expiration dates equal to the current time and maintains per-pool time that advances upon explicit deallocate-pool calls. An object may be in multiple pools and is deallocated when these pools have all been deallocated. General refreshing is again not possible.

Garbage collectors are implementations of the persistent memory model that compute unreachability, directly or indirectly, for reclaiming otherwise persistent memory. However, some portions of garbage collectors may be used to implement special cases of short-term memory. For example, as stated before, the mark phase of a mark-sweep garbage collector [19] may be used to implement an algorithm that prevents reachable objects from expiring. The transition from the mark to the sweep phase can then be seen as time advance for all objects. More recent work on object staleness, e.g. [7], and memory growth, e.g. [15], may be used to identify reachable memory leaks for expiring reachable but actually not-needed objects.

CPU	2x AMD Opteron DualCore, 2.0 GHz
RAM	4GB
OS	Linux 2.6.32-21-generic
C compiler	gcc version 4.4.3
C allocator	ptmalloc2-20011215 (glibc-2.10.1)
Java VM	Jikes RVM 3.1.0
Go compiler/runtime	6g, release 2010-11-02

Table 2. System configuration.

3.2 Self-Collecting Mutators

Reference-counting garbage collectors [9] determine reachability by counting references pointing to an object. In our implementations we determine expiration by counting descriptors pointing to an object. A drawback of reference counting are reference cycles which do not occur in descriptor counting. Moreover, the runtime overhead of descriptor counting is less than of reference counting since descriptor counters are only accessed at `tick`- and `refresh`-calls, which typically occur less frequently than reference changes.

Autorelease pools in Objective-C [16] approximate neededness by maintaining a so-called retain counter in each object that keeps track of the number of retain versus so-called release calls on the object. Thus the retain counters correspond to our descriptor counters. The pools contain references to objects and are thus similar to our descriptor lists.

The descriptor buffers in our implementations essentially implement priority queues [10] where expiration extensions correspond to priorities. Note that the time complexity of all our buffer operations is independent of the number of elements in the buffer, which may or may not be the case for general priority queues.

Global-time management in self-collecting mutators is related to epoch-based reclamation [12] and barrier synchronization [23]. A global period in global-time management corresponds to one epoch. A barrier forces a set of threads into a global state by blocking each thread when it has reached a particular point in its execution. Here, global time advance corresponds to the global state when all threads have ticked at least once in the current global period. However, threads that have ticked are not blocked until global time advance. We could also block threads, as in barrier synchronization, potentially reducing memory consumption at the expense of mutator execution speed. An implementation and adequate experiments are future work.

The idea of hybrid memory management systems is not new. One related example is the work presented in [8] and [14], which uses static analysis to insert `free`-calls in Java code, thus reducing the number of garbage collection runs. An interesting question is whether finding appropriate locations (and extensions) for `refresh`-calls as well as `tick`-calls is easier than for `free`-calls. In our Java implementation we collect expired objects in a way that is similar to the approach taken in [14].

4. Time and Space

We discuss performance results obtained with the benchmarks described in Section 2.1 and 2.2. The setup of the benchmarking environment is shown in Table 2. Memory consumption is reported as gross consumption including fragmentation. Net memory consumption is not shown since fragmentation turned out to be bounded by small constants in all benchmarks.

For the mpg123 benchmark in C we compare self-collecting mutators and ptmalloc2. For the x264 benchmark we run the original video encoder with and without pool allocation, and compare it with both porting approaches described in Section 2.1, the single-refresh and the continuous-refresh approach. We measure throughput (total execution time) and memory consumption of both benchmarks in all configurations. The effect on latency is negligible.

For the Java benchmarks we compare self-collecting mutators and two garbage collectors available with Jikes, the mark-sweep garbage collector that we already use with self-collecting mutators, and, as baseline, the standard garbage collector of Jikes, a two-generation copying collector where the mature space is handled by an Immix collector [6]. We measure the replay phase of replay compilation [22] provided by the production configuration of Jikes, which runs a JIT compiler in the recording phase. For the Go benchmarks we compare self-collecting mutators with the mark-sweep garbage collector of the Go runtime. We measure throughput (total execution time) and memory consumption of both the Java and Go benchmarks. Moreover, we measure latency (loop execution time for Java, time between two allocation operations in Go) and show that self-collecting mutators has lower latency than the garbage-collected systems.

Note that our use cases may also perform competitively when using other, more specialized memory management systems, e.g. a region-based allocator. However, a meaningful comparison requires a proper integration with self-collecting mutators that we have begun but not yet completed.

4.1 C

In both the mpg123 and the x264 benchmark runtime overhead through self-collecting mutators (SCM) is negligible. The mpg123 benchmark (and the x264 benchmark in some cases) runs even slightly faster than the unmodified baseline.

Figure 3 shows the memory consumption of the mpg123 benchmark. Self-collecting mutators with eager collection tracks the original memory consumption except for a final portion of memory before time advance. Maximum memory consumption is similar for lazy and eager collection. With lazy collection memory consumption lags the original memory consumption. The execution time of the tick-calls is around 20 times longer with eager collection (not shown) since there are around 20 objects to be collected per time advance, whereas the execution time of the refresh-calls is slightly lower with eager collection because no objects are collected during refreshing (not shown).

Figure 4(a) shows the memory consumption of the single-threaded x264 benchmark on the 300-frame foreman video sequence[5] (appended two times for 900 frames total) commonly used for video coding benchmarks. The first 150 frames and the last 300 frames are not shown for better resolution. The omitted data shows repetitive results or less memory consumption at the beginning and end. Single-refresh, while in principle faster, introduces some memory overhead by conservatively chosen expiration extensions. Memory consumption with continuous-refresh is always below memory consumption with pool allocation.

Figure 4(b) shows the memory consumption of the multi-threaded x264 benchmark on the same video running a main thread, a lookahead thread, and four encoding threads. We distinguish single-threaded short-term memory management where refreshing

[5] ftp://ftp.ldv.e-technik.tu-muenchen.de/dist/cif/

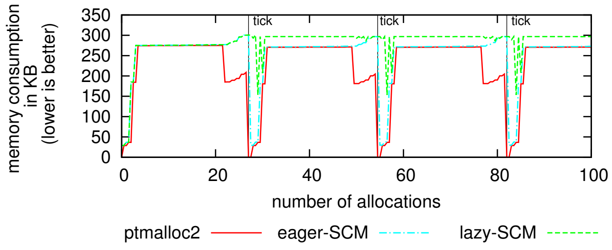

Figure 3. Memory consumption of the mpg123 benchmark.

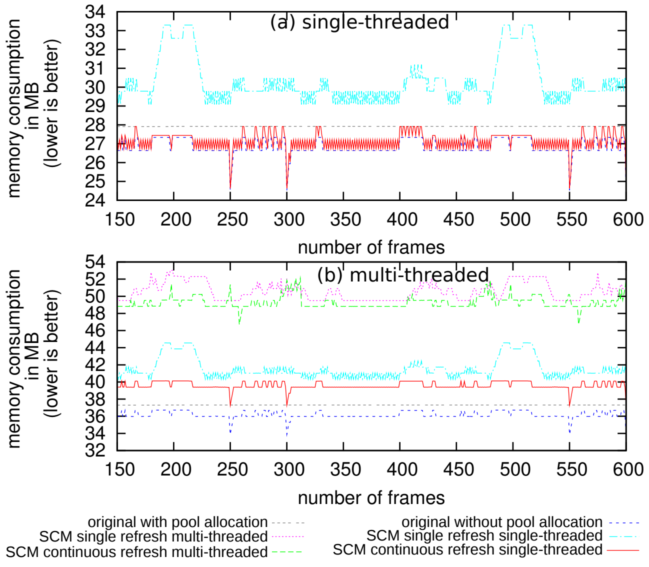

Figure 4. Memory consumption of the x264 benchmark (x-axis shortened for better resolution).

and ticking is done in the main thread from multi-threaded short-term memory management where refreshing and ticking is done in the encoding threads using either global time or else thread-local time. Both single-refresh and continuous-refresh perform well with single-threaded short-term memory management, introducing only low memory overhead over pool allocation (as expected the overhead of single refresh is slightly higher). With multi-threaded short-term memory management, memory consumption is higher except for continuous-refresh using thread-local time, which comes close to the maximum memory consumption of single-refresh in the main thread. Scalability is not affected, we obtain similar results on a 24-core machine.

4.2 Java

We execute the Monte Carlo benchmark 30 times and calculate the average of the total execution times.

The original Monte Carlo benchmark (MC leaky) produces a reachable memory leak which is not collected by a garbage collector. Self-collecting mutators (SCM) reuses the memory objects in the memory leak upon expiration. Therefore, the MC leaky benchmark can be executed in just 20MB with self-collecting mutators. The generational garbage collector (GEN) requires at least 95MB whereas the mark-sweep garbage collector (MS) requires 100MB. For this reason we benchmark MC leaky with heap sizes of 100MB as well as 1GB, which is enough memory to run the benchmark without garbage collection.

We then modified the Monte Carlo benchmark by removing the memory leak (MC fixed). The fixed benchmark runs successfully with a heap size of 20MB on all systems. For comparison we also benchmark MC fixed with a heap size of 50MB, which is the initial

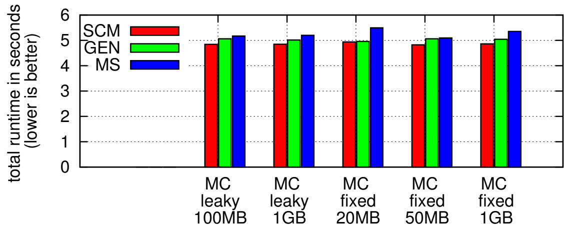

Figure 5. Total execution time of the Monte Carlo benchmarks.

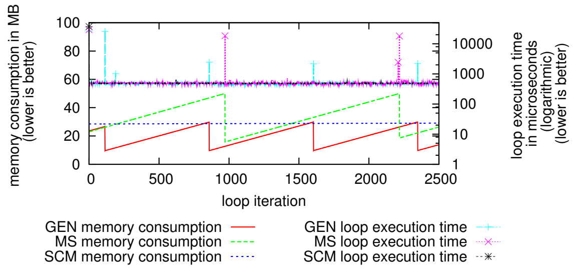

Figure 6. Memory consumption and loop execution time of the fixed Monte Carlo benchmark.

heap size of the production configuration of Jikes. The results are shown in Figure 5. SCM is slightly faster than the garbage-collected systems, even when more memory is available.

Figure 6 shows the memory consumption and loop execution time of the MC fixed benchmark recorded at the end of every loop iteration, with a heap size of 50MB. The results show that the memory consumption and latency of SCM are nearly constant, in particular there are no latency peaks (the loop execution time jitter is less than 100 microseconds) after startup. Both garbage-collected systems have similar loop execution times as SCM except for the iterations in which garbage collection is performed. The memory-consumption function of the garbage-collected systems has the typical saw-tooth shape with peaks right before each garbage collection run. The chart depicts the first 2500 loop iterations, further iterations show the same pattern. The measurements are done with all short-living objects flagged as short-term by 36 `refresh(0)`-calls. For avoiding all garbage collection runs it is sufficient to flag as short-term the objects of just 10 allocation sites (which allocate large objects).

Note that the first generational garbage collection run collects much more memory than the memory allocated by the application until then. The same is true for MS but to a lesser extent. This memory (not allocated by the application) is not collected when using SCM (it would be collected if garbage collection would trigger with SCM but it does not here). The overhead of SCM can be seen at time 0 as the difference between the GEN/MS versus the SCM memory consumption ($\leq$18%).

4.3 Go

The results of the Tree benchmark are similar to the results of the Monte Carlo benchmarks. Out of a total of 23 garbage collection runs all but one are avoided with self-collecting mutators improving latency and memory consumption, as shown in Figure 7, and total execution time by up to 28%. The remaining garbage collection run does not collect any objects.

The self-collecting mutators version of our implementation of a static multi-threaded web server in Go improves client-side server latency (from 1.87 to 1.54 milliseconds on average, from 1.7 to 1.05 milliseconds in standard deviation, and from 20.65

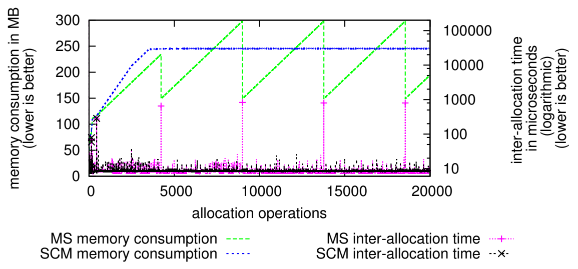

Figure 7. Memory consumption and inter-allocation time of the Tree benchmark.

to 10.85 milliseconds maximum) by decreasing the number of garbage collection runs while consuming about the same amount of memory as the unmodified server. The non-trivial aspect of this benchmark is that goroutine scalability is maintained since reuse of descriptor roots and pages across goroutines is effective.

5. Conclusion

We have proposed a memory model for heap management called short-term memory and developed an implementation of short-term memory for explicit refreshing called self-collecting mutators. Short-term memory may be particularly useful in applications with complex, data- rather than control-dependent object lifetime scenarios such as the x264 use case. Interesting, principled future work may be to develop an allocator aware of short-term memory using, e.g. regions, and to study alternative notions of time as well as program analysis for enabling safe use of short-term memory.

Acknowledgments

We are grateful to all reviewers of all previous versions of this paper, we have learned a lot from their comments and suggestions.

References

[1] Aigner, M., Haas, A., Kirsch, C. M., and Sokolova, A. Short-term memory for self-collecting mutators - revised version. Tech. Rep. 2010-06, Department of Computer Sciences, University of Salzburg, October 2010.
[2] Aigner, M., Haas, A., and Lippautz, M. Short-term memory implementation for C, Java, and Go, 2010. `http://tiptoe.cs.uni-salzburg.at/short-term-memory/`.
[3] Alpern, B., Attanasio, C. R., Barton, J. J., Burke, M. G., Cheng, P., Choi, J.-D., Cocchi, A., Fink, S. J., Grove, D., Hind, M., Hummel, S. F., Lieber, D., Litvinov, V., Mergen, M. F., Ngo, T., Russell, J. R., Sarkar, V., Serrano, M. J., Shepherd, J. C., Smith, S. E., Sreedhar, V. C., Srinivasan, H., and Whaley, J. The Jalapeño virtual machine. *IBM Syst. J. 39*, 1 (2000), 211–238.
[4] Bacon, D. F., Cheng, P., and Rajan, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proc. POPL* (2003), ACM.
[5] Bacon, D. F., Cheng, P., and Rajan, V. T. A unified theory of garbage collection. In *Proc. OOPSLA* (2004), ACM.
[6] Blackburn, S. M., and McKinley, K. S. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proc. PLDI* (2008), ACM.
[7] Bond, M. D., and McKinley, K. S. Leak pruning. In *Proc. ASPLOS* (2009), ACM.
[8] Cherem, S., and Rugina, R. Compile-time deallocation of individual objects. In *Proc. ISMM* (2006), ACM.
[9] Collins, G. E. A method for overlapping and erasure of lists. *Commun. ACM 3*, 12 (1960), 655–657.
[10] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001, ch. 6.5: Priority queues, pp. 138–142.
[11] Craciunas, S., Kirsch, C. M., Payer, H., Sokolova, A., Stadler, H., and Staudinger, R. A compacting real-time memory management system. In *Proc. USENIX ATC* (2008).
[12] Fraser, K. *Practical Lock-Freedom*. PhD thesis, Computer Laboratory, University of Cambridge, 2003.
[13] Gay, D., and Aiken, A. Memory management with explicit regions. In *Proc. PLDI* (1998), ACM.
[14] Guyer, S. Z., McKinley, K. S., and Frampton, D. Free-me: a static analysis for automatic individual object reclamation. In *Proc. PLDI* (2006), ACM.
[15] Jump, M., and McKinley, K. S. Cork: dynamic memory leak detection for garbage-collected languages. In *Proc. POPL* (2007), ACM.
[16] Kochan, S. *Programming in Objective-C 2.0*, 2nd ed. Addison-Wesley Professional, 2009.
[17] Masmano, M., Ripoll, I., Crespo, A., and Real, J. TLSF: A new dynamic memory allocator for real-time systems. In *Proc. ECRTS* (2004), IEEE Computer Society, pp. 79–86.
[18] Mathew, J. A., Coddington, P. D., and Hawick, K. A. Analysis and development of Java Grande benchmarks. In *Proc. JAVA* (1999), ACM.
[19] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM 3*, 4 (1960), 184–195.
[20] Merritt, L., and Vanam, R. X264: A high performance H.264/AVC encoder, 2006.
[21] Nguyen, H. H., and Rinard, M. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proc. ISMM* (2007), ACM.
[22] Ogata, K., Onodera, T., Kawachiya, K., Komatsu, H., and Nakatani, T. Replay compilation: improving debuggability of a just-in-time compiler. In *Proc. OOPSLA* (2006), ACM.
[23] Tanenbaum, A. S. *Modern Operating Systems*. Prentice Hall, 2001.
[24] Tofte, M., and Talpin, J.-P. Region-based memory management. *Inf. Comput. 132*, 2 (1997), 109–176.
[25] Wiegand, T., Sullivan, G. J., Bjøntegaard, G., and Luthra, A. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology 13*, 7 (2003).
[26] http://mewiki.project357.com/wiki/x264_settings.