Short-term Memory for Self-collecting Mutators Revised Version

Martin Aigner Andreas Haas Christoph M. Kirsch
Ana Sokolova

Technical Report 2010-06 October 2010

Department of Computer Sciences

Jakob-Haringer-Straße 2
5020 Salzburg
Austria
`www.cosy.sbg.ac.at`

Technical Report Series

Short-term Memory for Self-collecting Mutators[⋆]
Revised Version

Martin Aigner, Andreas Haas, Christoph M. Kirsch, Ana Sokolova

Department of Computer Sciences
University of Salzburg, Austria
`firstname.lastname@cs.uni-salzburg.at`

Abstract. We propose a new memory model for heap management, called short-term memory, and concurrent implementations of short-term memory for Java and C, called self-collecting mutators. In short-term memory objects on the heap expire after a finite amount of time, which makes deallocation unnecessary. Self-collecting mutators requires programmer support to control the progress of time and thereby enable reclaiming the memory of expired objects. We informally describe a simple translation scheme for porting existing programs to self-collecting mutators. As shown by our experimental results on several benchmarks, self-collecting mutators performs competitively with garbage-collected and explicitly managed systems. Unlike garbage-collected systems, self-collecting mutators does not introduce pause times and provides constant execution time of all operations, independent of the number of reachable objects, and constant short-term memory consumption after a steady state has been reached. Self-collecting mutators can be linked against any unmodified C code introducing a per-object space overhead of one word and negligible runtime overhead. Short-term memory may then be used to remove explicit deallocation of some but not necessarily all objects.

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.

⋆ Supported by the EU ArtistDesign Network of Excellence on Embedded Systems Design and the Austrian Science Funds P18913-N15 and V00125.

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 report. 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 create reachable memory leaks. Note that the issue of memory fragmentation is orthogonal to the discussion here and not considered in this report.

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. 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 guaran-

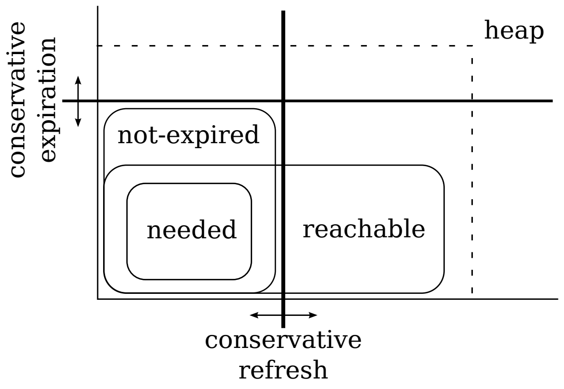

Fig. 1. Approximation of the needed set by the not-expired set in the short-term memory model.

teed 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. It is interesting to 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 report we focus on explicit refreshing. Like explicit deallocation, explicit refreshing may be done incorrectly but with different consequences. The source of incorrect use of explicit refreshing is missing refreshing information, resulting in memory being reclaimed too early creating dangling pointers. Other sources of errors in the explicit use of the persistent memory model are avoided with short-term memory. Multiple explicit deallocation of the same object is an error in the persistent memory model whereas multiple refreshing in the short-term memory model has no consequence other than creating runtime overhead. 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.

To summarize, correct explicit deallocation information must be “just right”. Too much information and too few information is a source of errors. In contrast, too much explicit refreshing information is still correct. As a consequence, any over-approximation of the minimal correct refreshing information is also correct. We believe that even static analysis has the potential to provide such an over-approximation eventually.

After presenting the short-term memory model in more detail, we discuss several use cases of short-term memory based on real code that was originally written in the persistent memory model. We then introduce concurrent implementations of short-term memory, called self-collecting mutators, written in C and in Java. Self-collecting mutators supports multi-threaded programs, in particular correct refreshing without concurrent reasoning similar to using garbage collectors.

The report presents the following contributions: (1) the short-term memory model and an analysis of its use by real code; (2) the concurrent self-collecting mutators implementations in Java and in C, and an implementation analysis; and (3) confirmation of the analysis with experimental results on several benchmarks.

The structure of the rest of the report is as follows. In Section 2 we introduce the concepts of short-term memory. The self-collecting mutators implementations are presented in Section 3. In Section 4 we present experimental results of a number of benchmarks. Section 5 concludes the report and presents future work.

2 Short-term Memory Model

	Persistent MM	Short-term MM
lifetime of an object	from allocation until deallocation	from allocation until expiration
lifetime management	deallocation	refreshing
errors	dangling pointers, memory leaks	dangling pointers, memory leaks
sources of errors	premature or no deallocation	incorrect refresh, no time progress
problems with concurrency	deallocation synchronization	time synchronization
problems with real time	implicit deallocation	(redundant) refresh

Table 1. Comparison of the persistent memory model with the short-term memory model.

In this section we present the short-term memory model and compare it with the persistent memory model. See Table 1 for a summary. We then introduce an explicit programming model for short-term memory and discuss four use cases to show how much effort it is to use short-term memory. The section concludes with a discussion of previous work related to short-term memory.

2.1 Model

With short-term memory, each object is only allocated for a finite amount of time after which the object expires, which means that its existence is not guaranteed anymore. So to say, every object has an expiration date. As long as an object has not expired, its expiration date can be extended by a finite amount of time through what we call refreshing.

The notion of time is important for the short-term memory model. It defines the lifetime of every object, which is the time from the allocation of an object until it expires. If time advances fast, objects will expire faster, and the system will require less memory. If time stands still, no object will ever expire. This would be equivalent to a system without deallocation. The definition of time determines some core properties of the memory management system and requires even more care in the presence of concurrency.

Use of the Model With absolute knowledge, an object can be allocated with its exact expiration date. Using exact expiration dates resembles explicit deallocation but may be even more difficult than knowing the position of explicit deallocation. On the other hand, for explicit deallocation a pointer to the object is required at deallocation time, which is not the case with expiration dates.

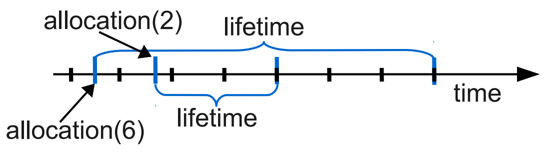

Fig. 2. Allocation with known expiration date.

Figure 2 presents an example of short-term memory with absolute knowledge about the expiration of objects. The lifetime of both allocated objects is known at allocation time. The expiration date can already be set then. For example, the command allocation(6) allocates an object for six time units.

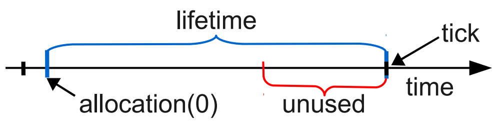

Fig. 3. All objects are allocated for one time unit.

In contrast to using exact expiration dates, every object can be allocated for zero time units, i.e., it will expire at the next time advance. Time advances when all existing objects are not needed anymore. An example is shown in Figure 3. All objects have the same expiration date. Even if an object is only used for a short amount of time, it will not expire before the next time advance.

Another choice between these two extremes is to allocate objects using estimated expiration dates, which can later be extended by refreshing. However, refreshing creates additional runtime overhead. It can be done explicitly by the programmer or implicitly by an underlying memory management system.

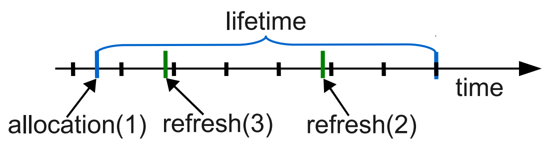

Fig. 4. Allocation with estimated expiration date. If the object is needed longer, it is refreshed.

Figure 4 illustrates refreshing. An object is allocated with an estimated expiration date. If the object is needed beyond its expiration date, it is refreshed. In Figure 4 the object exists for six time units in total. Since it was originally allocated for one time unit only, it had to be refreshed for another five, which is done by two consecutive refresh statements.

The notion of expiration date in the short-term memory model enables trading-off compile-time analysis effort, runtime overhead, and memory consumption. Allocation with known expiration date (cf. Figure 2) requires full compile-time analysis, but least runtime overhead and memory consumption. Allocation for one time unit (cf. Figure 3) requires only light-weight compile-time analysis needed for time control, but introduces additional memory consumption. With refreshing (cf. Figure 4), compile-time analysis effort remains light-weight and memory consumption improves at the expense of increased runtime overhead.

Sources of Errors A memory management system based on the short-term memory model could be used incorrectly creating dangling pointers and memory leaks.

Dangling pointers, which are pointers to objects that no longer exist, may be created by premature expiration. Dangling pointers can be avoided, for example, by continuously refreshing every reachable object at the expense of increased runtime overhead and memory consumption.

Memory leaks occur when not-needed objects are never deallocated or reused. With explicit deallocation, memory leaks occur due to missing free calls. Even with garbage-collected systems reachable memory leaks occur due to references to not-needed objects. Short-term memory avoids memory leaks which are present in explicit-deallocation systems, and may avoid reachable memory leaks provided that reachable but not-needed objects are not continuously refreshed, creating the potential for dangling pointers again. In Section 4.1 we present a benchmark where our explicit implementation of short-term memory repairs a reachable memory leak. Similar handling of memory leaks is described in [17].

However, with short-term memory, memory leaks do occur but under new circumstances, i.e., when not-needed objects never expire, caused by continuous refreshing or time standing still. In our explicit implementation of short-term memory, the programmer, possibly supported by static analysis, needs to make sure that time advances. It may be possible to implement short-term memory using real time instead of programmer-controlled time in which case time is guaranteed to advance eventually but refreshing may be more difficult to do correctly.

Concurrency In explicit-deallocation systems it can be difficult to place deallocation statements correctly, in particular in the presence of multiple threads. When several threads use the same object, only the last-accessing thread can deallocate the object correctly. The difficulty of deallocation comes from the need of synchronizing deallocation statements among threads. Garbage collectors solve the difficult problem of correct deallocation in particular for concurrent programs. The same can be achieved with short-term memory.

When using short-term memory, every thread refreshes the objects it uses, just as for single-threaded applications. Logically, each object has a separate expiration date per thread. An object expires when it expires for all threads. Depending on the notion of time, using short-term memory for concurrent programs is more or less convenient. Our implementation provides a synchronized global and an unsynchronized thread-local notion of time. With global time correct use of short-term memory does not require concurrent reasoning, similar to using garbage collectors.

We already stated before that memory leaks can be introduced in short-term memory if time stands still. For multi-threaded applications it is necessary that global time also advances if some threads are inactive or blocked. This is not a problem if real time is used but it has to be considered for systems in which time advance depends on the progress of the thread. Our implementation also solves the problem of blocking threads.

Real Time Many real-time programs use static memory management in which all memory is allocated when a program is started. Reasons for this are that it is difficult to guarantee the correct use of explicit dynamic memory management, and that garbage collection performance depends, in the worst case, on the total number of live objects, either in time or in space.

Short-term memory can be used for real-time programs if refreshing is done incrementally because refreshing incurs runtime overhead proportional to the number of needed objects. In a multi-threaded setting redundant refreshes of shared objects by multiple threads may need to be reduced for better performance.

2.2 Use Cases

What we propose in this report is that short-term memory can be used explicitly and thereby provides an interesting heap management interface not just to programmers but potentially also to static analysis tools. The main questions are then: how easy is it to use short-term memory explicitly, and how many of the required memory management calls can be added by a static analysis tool?

To answer the first question we define a programming model which explicitly uses short-term memory. Most important for such a programming model is the definition of time. We also define how object allocation and refreshing work. The answer to the second question remains future work.

Explicit Programming Model We use relative user-defined per-thread time represented by a thread-local clock which is a counter incremented by one whenever the thread to which the clock belongs invokes an explicit tick-call. In order to guarantee that time advances, the user is required to put tick-calls at locations in the program code that are always eventually executed. Logically, each object has a separate expiration date for each thread. An object expires when it has expired for all threads. An object expires for a thread when the thread-local time is greater than the expiration date of the object for this thread.

Upon allocation an object receives expiration dates for all threads that are initialized to the respective thread-local times. Refreshing is done by explicit refresh-calls, which take two parameters, an object which should be refreshed and an expiration extension. The new expiration date of an object (for the thread invoking the refresh-call) is the current (thread-local) time plus the given expiration extension. Moreover, the expiration dates for the threads for which the object has already expired are set to the respective thread-local times. This way other threads get a chance to refresh the object before it expires. For example, a producer of an object may stop refreshing the object and tick as soon as the object is consumed by a consumer, which then still has a chance to refresh the object and tick without further coordination with the producer. Note that it makes no difference if an object is refreshed once or multiple times (by the same thread) within one time unit. Moreover, in some cases it is useful to have a recursive refresh-call that refreshes all objects reachable from a given object. Performing a recursive refresh-call is similar to a mark-sweep garbage collector performing a (partial) mark phase.

The explicit programming model does not require concurrent reasoning for correct usage by the programmer similar to using garbage collectors. In other words, each thread may tick and refresh the objects it needs independently of any other threads. Note that our implementations do not actually maintain expiration dates for all threads and therefore only approximate this model in the sense that objects may expire later than they could, but never earlier. As a result, all memory management operations take constant time at the expense of potentially increased memory consumption.

Benchmarks We translated the following programs to use short-term memory:

benchmark	LoC	tick	refresh	free	aux	total
mpg123	16043	1	0	(-)43	0	44
JLayer	8247	1	6	0	2	9
Monte Carlo	1450	1	3	0	2	6
LuIndex	74584	2	15	0	3	20

Table 2. Use cases of short-term memory: lines of code of the benchmark, number of tick-calls, number of refresh-calls, number of free-calls, number of auxiliary lines of code, and total number of modified lines of code.

1. the mpg123[1] MP3 converter version 1.12 written in C,
2. the JLayer MP3 converter[2],
3. the Monte Carlo benchmark of the Grande Java Benchmark Suite [15],
4. the LuIndex benchmark of the Dacapo Benchmark Suite [6], version 9.12.

We informally applied a translation scheme that makes establishing correctness easy at the expense of potentially decreased runtime performance and increased memory consumption. We first identify the code location that marks the end of the period of the most frequent periodic behavior of the benchmark, and where most of the memory expires. We say that this memory is short-term with respect to that period. We then place a tick-call at this code location, which was easy for us to find in all four benchmarks. We finally add refresh-calls on objects that are still needed after executing the tick-call to maintain memory that is not short-term. All other memory is short-term and will then expire.

The mpg123 benchmark converts a set of mp3 files to a set of corresponding wav files. All memory needed for the conversion of a single file is short-term, which means that it expires once the file is converted. Therefore, one tick-call is sufficient and is conveniently placed in the code where processing a file is finished. This removes the need for all 43 free-calls in the original code. No refresh-calls are required.

For the remaining three programs written in Java we only use recursive refresh-calls. Similar to the mpg123 benchmark, the JLayer benchmark converts mp3 files to wav files. However, we have only benchmarked JLayer on a single file at a time, and therefore identified frame rather than file processing as the relevant periodic behavior for placing a tick-call in the code where processing a frame is finished. Four refresh-calls are required for input and output buffers. Another refresh-call is required for a progress-listener object. The application root object allocated in the main method of JLayer, which needs to exist during the whole program execution, also requires a refresh-call. This object is a local object, only reachable from within the main method. Making this object reachable from the code location where refreshing is done results in two auxiliary lines of code.

[1] http://www.mpg123.de
[2] http://www.javazoom.net/javalayer/javalayer.html

The Monte Carlo benchmark consists of a calculation loop to which we added a tick-call at the end. Hence, all memory allocated within one loop iteration is short-term, except for a result object that is generated in every loop iteration and stored in a result set which requires one recursive refresh-call. A second refresh-call is required to refresh the application root object, again with two auxiliary lines of code to make it accessible. A third refresh-call is required on an object used for time measurements.

The LuIndex benchmark consists of two threads. The first thread does not have a main loop but recursively iterates over files contained in a hierarchical file system. File processing is the relevant periodic behavior here according to our scheme but more difficult to identify because of the absence of a main loop. A tick-call is placed in the code where processing a file is finished. Two refresh-calls and three auxiliary lines of code are necessary to refresh the application root object and to prevent the current state of the recursion from expiring. Another refresh-call is required for a result data object. Finally, eleven refresh-calls are necessary to prevent static variables from expiring. The second thread processes, in a loop, the data generated by the first thread. We placed a tick-call at the end of this loop and added a refresh-call on its only root object.

2.3 Related Work

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 even a programmer that provides the information manually, as with self-collecting mutators. The representation may 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 [17]. 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 [21] can also be seen as implementing a special case of short-term memory where the representation are per-code-block regions, which allow to deallocate multiple objects in constant time. The algorithm always uses expiration dates equal to the current time plus one and maintains per-region time that advances upon events determined by an offline analysis tool. General refreshing is not possible but could be done by copying objects from one region to another.

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 [16] 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. [8], and memory growth, e.g. [14], may be used to identify reachable memory leaks for expiring reachable but actually not-needed objects.

3 Self-collecting Mutators

In this report we present two new concurrent implementations of short-term memory, called self-collecting mutators, which conservatively approximate the explicit programming model previously introduced in Section 2.2. We produced one implementation written in C to enable programs written in C to use short-term memory, and another one written in Java for Java programs. We elaborate on the advantages and disadvantages of each implementation after presenting general design choices. Both implementations have the following properties and features:

– Constant time complexity for all operations.
– Constant memory consumption by all operations.
– No additional threads and no read/write barriers.

All memory management operations are constant-time, which enables full incrementality, and allocate at most constant memory. The system is self-collecting, i.e., there are no additional threads for memory management, and there are no read or write barriers. The combination of incrementality and self-collection is in general only possible at the expense of increased memory consumption since memory may be reclaimed with a mutator-dependent delay, which nevertheless bounds the increase in memory consumption.

3.1 Design

Recall the explicit programming model for short-term memory presented in Section 2.2. Logically, each object has a separate expiration date for each thread and expires when it has expired for all threads. Clearly, implementing this may cause too much space and time overhead since all expiration dates must be stored and refreshing requires updating all expiration dates of an object. However, for correct expiration, it is enough to ensure that no object expires earlier than prescribed by its logical expiration dates. Hence, we may approximate the set of all expiration dates of an object by a subset of it using a notion of global time that advances at the speed of the slowest-ticking thread.

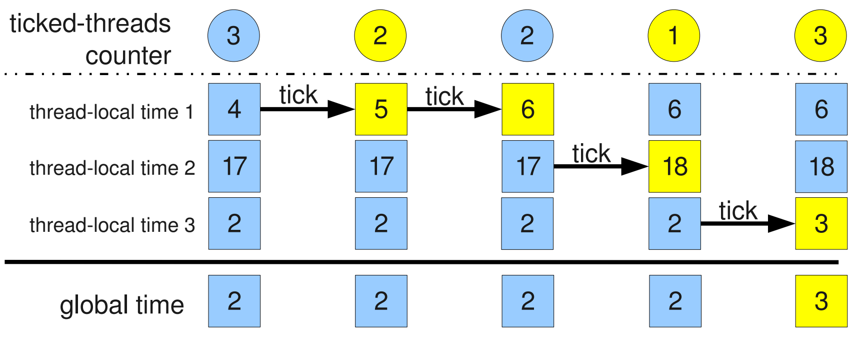

Fig. 5. Global time calculation.

Single-Expiration-Date Approximation Each thread has a thread-local clock which is incremented by one whenever the thread ticks, i.e., invokes a tick-call. The thread-local clock determines the thread-local time. In addition to the thread-local clocks, we keep track of global time represented by a global clock which is a counter incremented by one whenever all threads have ticked at least once. More precisely, we keep a ticked-threads-counter which is reset to the total number of threads at every increment of the global time. When a thread ticks for the first time after the ticked-threads-counter has been reset, we decrement the ticked-threads-counter by one (in an atomic decrement-and-test operation in C and using a lock in Java). Global time advances when the ticked-threads-counter reaches zero. For atomic global-time advance and reset of the ticked-threads-counter we use a lock (in both implementations). The time period between two advances of the global clock is called the global period. The calculation of global time is illustrated on an example in Figure 5.

The explicit programming model can be approximated by keeping a single expiration date evaluated against global time. This means that an object expires when the global time is greater than the expiration date. When an object is allocated its expiration date is set to the global time plus one. Adding one additional time unit to the global time is necessary since at the time of allocation some (but not all) threads may have already ticked in the current global period. Allocation and expiration of an object in the single-expiration-date approximation is shown in Figure 6(a). When a thread refreshes an object, the new expiration date is set to the global time plus one plus the expiration extension, unless the

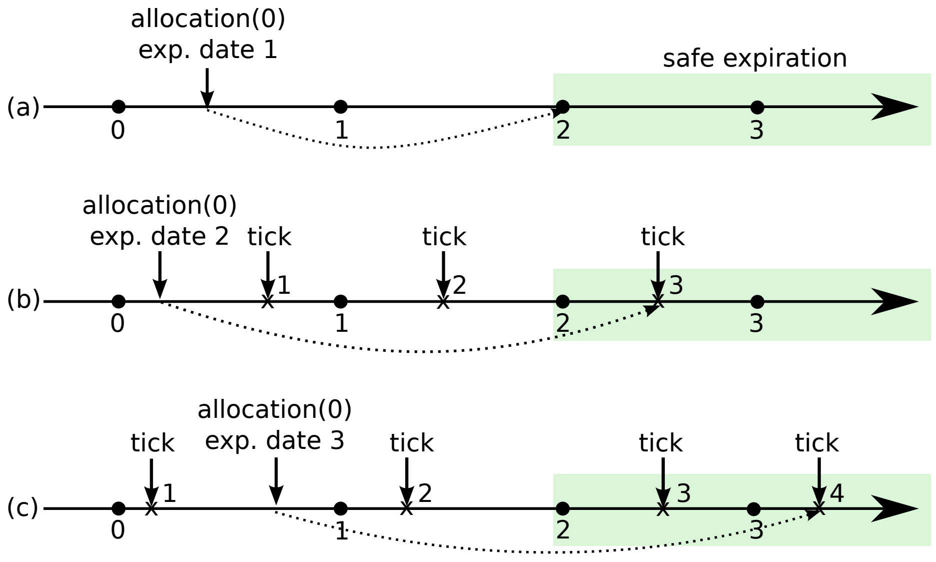

Fig. 6. Object expiration in the single-expiration-date approximation (a) and in the multiple-expiration-date approximation (b,c). Filled circles indicate global-time advance, and ×'s indicate thread-global-time advance.

result is lower than the old expiration date. This way no thread can shorten the lifetime of an object already prescribed by another thread. As a result, no object will expire too early. Moreover, a programmer need not know any of the implementation details of self-collecting mutators except for the explicit programming model.

A disadvantage of the single-expiration-date approximation is that global time does not advance if a thread stops ticking, due to blocking, faults, or programming errors. We provide a solution to the problem of blocking and faults (but not programming errors), using multiple expiration dates that approximate the programming model better than a single expiration date, at the expense of increased memory consumption.

Multiple-Expiration-Date Approximation Assume the set of threads is partitioned into active and passive (blocked/faulty) threads. The computation of global time remains the same but only on the active threads, i.e., the ticked-threads-counter counts only active threads. Each thread maintains an additional clock, called the thread-global clock, which advances at the speed of the global clock as long as the thread is active and stops when the thread is passive. The thread-global clock advances at the first tick of the thread in the current global period. Therefore, it is possible that the thread-global time is larger by one time unit than the global time but only in the remainder of the global period. When a thread blocks, it is moved from the set of active to the set of passive threads, so that the ticked-threads-counter does not consider it anymore, ensuring that the

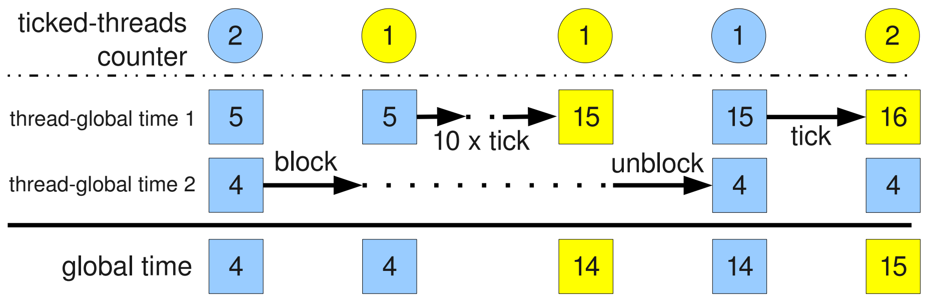

Fig. 7. Thread-global times.

global time advances without this thread. The thread-global time of the blocked thread remains unchanged as long as it stays blocked. When a thread resumes, it is moved from the set of passive to the set of active threads assuming that it has already ticked in the current global period, unless the set of active threads was empty. Hence, the ticked-threads-counter does not change. Figure 7 shows the advance of global time and thread-global times as well as the evolution of the ticked-threads-counter on an example.

An object will now have per-thread expiration dates but not necessarily an expiration date for each thread. An expiration date for a given thread is evaluated against the thread-global time of the thread. An object expires when it has expired for all its expiration dates. Upon allocation an object receives a single expiration date for the thread that allocated the object initialized to the thread-global time plus two. Adding two additional time units to the thread-global time is necessary since two time units of the thread-global time are guaranteed to include one global period. One could add a single additional time unit (instead of two) but only if the allocating thread has already ticked in the current global period. However, testing this condition and setting the expiration date must then be done atomically. We have not implemented this optimization since allocation (and refreshing) are frequent operations. Allocation and expiration of an object in the multiple-expiration-dates approximation is shown in Figure 6, distinguishing whether the allocating thread has not ticked, Figure 6(b), or has already ticked, Figure 6(c), in the current global period. When a thread refreshes an object a new expiration date is created and set to the thread-global time plus two plus the expiration extension. If the object already has an expiration date for the thread, we keep the larger of the two. It is also correct to keep multiple expiration dates per object and thread, as in the C implementation, as long as an object expires only when all expiration dates have expired.

The multiple-expiration-date approximation solves the problem of global time not advancing due to blocking and faulty threads, at the expense of additional memory consumption. Note that if an object is known to be local to the allocating thread (not shared by any other thread), then its expiration dates can be evaluated against thread-local time. In such a case, the initial expiration date can be set to the thread-local time (without additional time units), resulting in potentially earlier expiration. In the C implementation we provide an API for distinguishing shared and local objects.

3.2 Implementation

We implemented the single-expiration-date approximation for Java, for C we developed a multiple-expiration-date approximation. An advantage of the single-expiration-date approximation is that it allows for a simple implementation of recursive refresh. For Java, recursive refresh is particularly important because of the potentially large number of objects used in Java applications. The drawback of our current Java implementation is that it can not deal with blocking and faulty threads. Implementing recursive refresh in the multiple-expiration approximation is more difficult and left for future work. However, the absence of recursive refresh in C is not much of a disadvantage because the missing object model in C often results in programs with less hierarchical object graphs.

Single-Expiration-Date Implementation for Java The Java implementation of the single-expiration-date approximation [2] is based on the Jikes Research Virtual Machine [3], version 3.1.0, and the Gnu Classpath[3] class library, version 0.97.2.

We extended the Jikes object model by an object header that consists of three words storing a 16-bit integer representing the value of the (single) expiration date of an object, a 16-bit allocation-site identifier explained below, and two references to other objects for creating a doubly-linked list of objects sorted by increasing expiration dates. The list implements an object buffer that is FIFO for objects with the same expiration date. There are three buffer operations: insert, remove, and select-expired; and two buffer implementations: insert-pointer buffer and segregated buffer [2] for which the time complexity of all operations is independent of the size of the buffers. The time complexity of the segregated buffer operations are $O(1)$ for insert and remove, and $O(\log n)$ for select-expired where n is the maximal expiration extension, which is fixed at compile time in our implementation. The time complexity of the insert-pointer buffer operations are $O(1)$ for select-expired and remove, and $O(\log n)$ for insert. We only use the segregated buffer implementation in the Java benchmarks because select-expired operations may be done less often than insert operations.

Object buffers are allocated per allocation site [17] and lock-protected for multi-threading. Per-thread buffers are possible to reduce contention but remain future work. Upon allocation of a new object, the corresponding allocation-site buffer is checked for an expired object. If there is one, it is removed from the buffer and its memory is reused for the new object. Otherwise, memory for the new object is allocated from free memory. After setting the expiration date of the new object, it is inserted into the buffer. Refreshing an object removes the object from the buffer in which the object is stored. The correct buffer is identified by the previously mentioned 16-bit allocation-site identifier in the header of the

[3] http://www.gnu.org/software/classpath/

object. Then, the new expiration date of the object is set. Finally, the object is inserted back into the buffer. Recursive refresh works by traversing in a single round the objects that are reachable from a given object and have an expiration date less than the new expiration date, refreshing each object along the way. Object traversal stops whenever a leaf object or an object with an expiration date greater than or equal to the new expiration date is reached. Note that some objects may thus not be refreshed, i.e., if they are only reachable via objects that already had an expiration date greater than or equal to the new expiration date before the recursive refresh. However, this problem does not occur in our benchmarks. A general solution is related to parallel tracing [18] and remains future work.

In our current implementation, memory that was allocated once is never returned to free memory. In particular, the memory of expired objects of different size allocated at the same allocation site is only reused if it fits new allocation requests and is otherwise discarded and never reused again. This is an open issue for future work, which may be addressed by using per-object-size rather than per-allocation-site buffers. However, the use of per-allocation-site buffers has a convenient side effect. The memory of objects allocated at a given allocation site is only reused if the allocation site is executed again after time advanced. It turns out that in some benchmarks there are allocation sites that are only executed during program initialization for allocating permanent objects. We exploit the side effect in these benchmarks to handle permanent objects efficiently without refreshing. We may nevertheless handle permanent objects differently in the future through, e.g. infinite expiration dates.

Another interesting feature of using per-allocation-site buffers is that memory reuse is type-safe. However, reusing memory too early is still an error that results from missed refreshing or premature time advance. We implemented a debug mode in Jikes to detect this kind of error. In debug mode, instead of actually reusing the memory of an object, it is just marked. The error occurs if a marked object is accessed, which is detected by a read barrier. A marked object that triggers the barrier should have been refreshed. The debug mode helped us find all necessary refresh-calls in the LuIndex benchmark.

Jikes is a metacircular implementation, which uses the same heap management and garbage collector for the VM and the mutator. When using self-collecting mutators garbage collection is turned off. Since porting Jikes to short-term memory may be difficult and has not been done, we only redirect allocation requests of the mutator to short-term memory by annotating the classes that are exclusively used by the mutator. Allocation requests by the VM are therefore permanent. Allocation requests in classes used by both the VM and the mutator are also permanent, which is the case in the JLayer and LuIndex benchmarks. However, since the VM typically stops allocating at some point in time, an aggressive space optimization is possible by redirecting, after that point in time, all allocation requests, even in classes that are not annotated, to short-term memory. More details are discussed in Section 4.1.

Multiple-Expiration-Date Implementation for C The C implementation of the multiple-expiration-date approximation is a dynamic C library available under GPL [1] based on POSIX threads and the ptmalloc2 allocator of glibc-2.10.1[4].

Descriptors. An expiration date of a given memory object in the C implementation is represented by a descriptor, which is a single word that stores a pointer to the object. Moreover, each memory object is extended by an object header that consists of a descriptor counter, which is an integer word 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. 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.

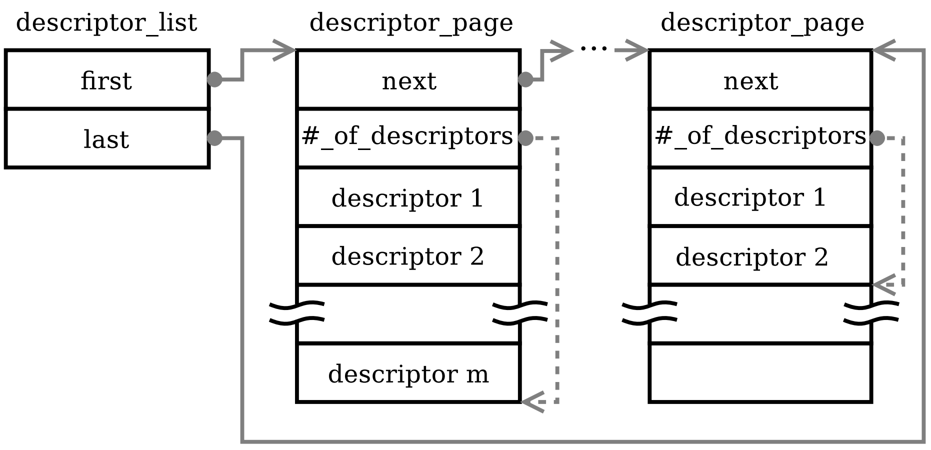

Fig. 8. The design of the descriptor list.

As shown in Figure 8, a descriptor list is a singly-linked list of descriptor pages with 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 words for storing descriptors. 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

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

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 descriptor buffer to store $n+1$ descriptor lists in an array of size $n+1$, which supports expiration extensions between zero and n. Note that $n+1$ lists are sufficient if the descriptors are evaluated against thread-local time. If they are evaluated against thread-global time, the buffer needs to store $n+3$ lists (recall the two additional time units necessary for correct expiration using thread-global time as previously discussed in Section 3.1). For better runtime performance descriptor pages are prefetched, i.e., the descriptor lists of a descriptor buffer always contain at least one descriptor page, which may nevertheless not contain any descriptors.

There are two descriptor buffer operations: insert and move-expired, which are both $O(1)$. Given a descriptor and an index $0 \leq i \leq n$, which is computed from the current time and the expiration extension as described below, the insert operation stores the descriptor 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 page pool or, if empty, from free memory, and appended to the list. Given an index $0 \leq i \leq n$, which is computed from the current time as described below, the move-expired operation removes from 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. The then empty descriptor list at i is refilled with an empty descriptor page prefetched either from the thread-local page pool or, if empty, from free memory.

There are two descriptor buffers per thread: a locally-clocked buffer and a globally-clocked buffer, containing $n+1$ and $n+3$ descriptor lists, respectively. The descriptors in the locally-clocked and the globally-clocked buffer are evaluated against thread-local and thread-global time, respectively. Let l be the current thread-local time. Then the descriptor list in the locally-clocked buffer containing descriptors representing an expiration date l is located at position $l \mod (n+1)$. Given an expiration extension $0 \leq e \leq n$, a new descriptor representing an expiration date $l+e$ is therefore inserted by an insert operation into the descriptor list at position $(l+e) \mod (n+1)$. Thus the descriptors in the descriptor list at position $(l+n) \mod (n+1)$ expire when the thread-local time advances and must be removed by a move-expired operation. Similarly, if g is the current thread-global time, then the descriptor list in the globally-clocked buffer containing descriptors representing an expiration date g is located at position $g \mod (n+3)$. A new descriptor representing an expiration date $g+e$ is therefore inserted into the descriptor list at position $(g+e+2) \mod (n+3)$. Here, the descriptors in the descriptor list at position $(g+n+2) \mod (n+3)$ expire when the thread-global time advances and must be removed.

Memory management operations. We now describe each memory management call. A malloc-call simply calls the underlying malloc routine of ptmalloc2 to allocate a memory block that fits the requested size plus one word for the object header containing the descriptor counter initialized to zero. Similarly, a free-call invokes the underlying free routine of ptmalloc2 to deallocate the given memory block but only if its descriptor counter is zero. Otherwise, it returns without deallocation. The calloc and realloc routines of ptmalloc2 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 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[5] and executing it. Without using short-term memory, our library only introduces a per-object space overhead of one word, a per-thread space overhead of $\Theta(n*m)$, where n is the compile-time bound on expiration extensions and m is the size of descriptor pages, and negligible runtime overhead as shown in Section 4.2.

There are two refresh calls: a local-refresh-call and a global-refresh-call, which we denote by refresh-call whenever the distinction is irrelevant. Both calls are $O(1)$ and do not require locking, just atomic increment and atomic decrement-and-test operations. The local-refresh-call first atomically increments the descriptor counter of the given memory object. Then, if the expired-descriptor list is not empty, the first expired descriptor in the list, which is most likely unrelated to the object under consideration, 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 thread-local 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 of ptmalloc2. Then, the descriptor counter of the object to which the expired descriptor points is atomically decremented. If the counter becomes zero, the object is deallocated, again by calling the underlying free routine of ptmalloc2. Finally, using the previously described index computation, a new descriptor pointing to the memory object that is to be refreshed is inserted into the locally-clocked buffer by an insert operation. Since refreshing always removes one expired descriptor, if there is one, the memory allocated for descriptors is bounded in the number of refresh calls between tick-calls, similar to the memory allocated for objects.

The global-refresh-call works similarly but on the globally-clocked buffer. It should be used on all objects that may be shared among threads. In contrast, the local-refresh-call results in less memory consumption but should only be used on objects that are known to be unshared. Note that replacing global-refresh-calls on shared objects by local-refresh-calls is possible but may involve concurrent reasoning. For example, in a producer-consumer scenario with a lock-protected

[5] http://httpd.apache.org/

communication buffer, the producer may invoke local-refresh-calls on the objects in the buffer but only advance its thread-local time when it holds the lock on the buffer. Conversely, the consumer must then invoke local-refresh-calls on the objects it removes from the buffer before releasing the lock in order to prevent any premature expiration.

The tick-call computes the thread-local, thread-global, and global times, as described in Section 3.1, in $O(1)$. The call also expires, similar to a refresh-call, one descriptor from the expired-descriptor list, if there is one. This is optional but may improve performance. Note that, in future work, we may choose to expire descriptors also in all other calls such as the malloc-call and free-call, and to expire more than one descriptor per call as long as it is a constant number of descriptors. Another option is running auxiliary threads that expire descriptors concurrently to the mutator.

The tick-call requires a lock but only when it actually advances the global time, which is anyway only done by a single thread. The lock is still required to protect the thread against interference from threads that have just been created and are now registering with the short-term memory management system. Thread creation and thus registration is a process that is likely to be much less frequent than the invocation of tick-calls. Contention on the lock may therefore be unlikely.

Managing memory objects in persistent or short-term memory can now be done as follows. A malloc-call allocates persistent memory for a given object. As long as the object is not refreshed, it remains in persistent memory and thus requires 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, explicit deallocation is unnecessary and should be replaced by invoking tick-calls instead. A free-call on the object is still possible but should be avoided since it may lead to double-freeing. Note that the malloc-call could also do both allocation and refresh in one step, which we nevertheless chose not to do for backwards compatibility.

Thread registration. Thread registration is the only $O(n)$ operation, where n is again the compile-time bound on expiration extensions. Thread registration needs to allocate memory for storing the locally-clocked and globally-clocked buffers, and the expired-descriptor list. However, there may have been threads in the past that already terminated. Before a thread is destroyed, thread termination blocks it and saves its buffers and list in a global, lock-protected, unbounded thread-data pool since the thread may still hold descriptors that need to be expired. When a new thread is later created and registered, thread registration reuses the buffers and lists, if available, instead of allocating new memory. The new thread is then resumed, as if it already executed and blocked before, and may thus expire the descriptors of a terminated thread. Other choices such as auxiliary threads expiring descriptors of terminating threads are possible but remain future work.

We have already implemented blocking, resuming, registering, and unregistering threads but have only integrated thread registration into the thread management system. The other operations still require manual invocation. Integrating them as well remains future work.

3.3 Related Work

As already stated in the related work of short-term memory, the work presented in [17] also describes the use of buffers per allocation site with the intention of eliminating memory leaks. There, cyclic buffers whose size is determined in experiments are used. Self-collecting mutators determines buffer sizes dynamically depending on tick-calls. Moreover, it allows trading-off space consumption caused by sparse tick-calls and time consumption caused by required refresh-calls.

The memory management system described in [17] maintains type safety as self-collecting mutators for Java does. Other work which provides memory management type safety to support the design of non-blocking thread synchronization algorithms is reported on in [13]. In [12] the authors propose the use of type-safe pool allocation to support program analysis.

Reference-counting garbage collectors [9] determine reachability by counting references pointing to an object. In our C implementation 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.

The buffers in our implementations essentially implement priority queues [10] where expiration extensions correspond to priorities. It is important to 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.

The calculation of global time in self-collecting mutators is related to barrier synchronization [20]. 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. Global time advance corresponds to the global state when all active threads have ticked at least once in the current global period. However, it does not impose blocking on the threads. Similar to a barrier, we could also block threads when they have ticked. In this case, refreshing objects would only require one instead of two additional time units, potentially reducing memory consumption at the expense of mutator execution speed. An implementation and adequate experiments are future work.

Finally, note that the memory behavior of self-collecting mutators can also be achieved with static preallocation. However, as visible from the benchmarks in Table 2, self-collecting mutators is convenient to use and does not require many code changes.

4 Experimental Setup and Evaluation

We discuss performance results obtained with the benchmarks described in Section 2.2. The benchmarking setup is shown in Table 3. For the Java benchmarks we compare self-collecting mutators and two garbage collectors available with

CPU	2x AMD Opteron DualCore, 2.0 GHz
RAM	4GB
OS	Linux 2.6.32-21-generic
Java VM	Jikes RVM 3.1.0
C compiler	gcc version 4.4.3
C allocator	ptmalloc2-20011215 (glibc-2.10.1)

Table 3. System configuration.

Jikes, the mark-sweep garbage collector and the standard garbage collector of Jikes, a two-generation copying collector where the mature space is handled by an Immix collector [7]. We measured throughput (total execution time) and memory consumption of the three Java benchmarks and of four concurrent instances of the Monte Carlo benchmark. Moreover, for the Monte Carlo benchmark, we measure latency (loop execution time) for the comparison with the garbage-collected systems and show, for the single-instance Monte Carlo benchmark, the effect of using different frequencies of tick-call invocations on latency and memory consumption.

For the C benchmark we compare self-collecting mutators and ptmalloc2. We measure the average (min/max) execution time of all self-collecting mutators operations, throughput (total execution time), and memory overhead and consumption.

4.1 Java Benchmarks

For the Java benchmarks we use replay compilation [19] provided by the production configuration of Jikes, which runs a JIT compiler in the recording phase.

Total Execution Time and Memory Consumption We execute each measurement 30 times and calculate the average of the total execution times. We determine the minimal heap size necessary to execute each benchmark with self-collecting mutators. The same heap size is then used for the garbage-collected systems if it suffices for the execution which is true in all but one benchmark.

	MC leaky	MC fixed	4×MC fixed	JLayer	LuIndex
SCM(1,1)	40MB	40MB	60MB	95MB	370MB
SCM(50,20)	50MB	40MB	70MB	/	/
aggressive SCM(1,1)	/	/	/	90MB	250MB
GEN	95MB	40MB	70MB	95MB	370MB
MS	100MB	40MB	70MB	95MB	370MB

Table 4. Heap size for the different system configurations. SCM(n, k) stands for self-collecting mutators with a maximal expiration extension of n. A tick-call is executed every k-th round of the periodic behavior of the benchmark.

For comparison, we also measure the total execution times with double amount of memory. The minimal heap sizes for the benchmarks are shown in Table 4.

The original Monte Carlo benchmark (MC leaky) is the only benchmark where garbage-collected systems need significantly more memory to run. The reason is that 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, running the benchmark with self-collecting mutators requires much less memory than running it with garbage-collected systems. With self-collecting mutators the MC leaky benchmark can be executed in 40MB. We improve the total execution time of the benchmark by refreshing with an expiration extension of 50 and reducing the frequency of tick-calls to one every 20th loop iteration (SCM(50,20)) resulting in the need for 10MB more, i.e., 50MB heap size. The generational garbage collector (GEN) requires at least 95MB for a successful run whereas the mark-sweep garbage collector (MS) requires 100MB. We modified the Monte Carlo benchmark and removed the memory leak (MC fixed). The MC fixed benchmark needs only 40MB heap size on all systems. We executed four concurrent instances of the Monte Carlo benchmark without memory leak (4×MC fixed). The execution of 4×MC fixed requires 60MB heap size in the SCM(1,1) configuration of self-collecting mutators, additional 10MB are needed in the SCM(50,20) configuration.

The JLayer benchmark needs 95MB heap size to execute and the LuIndex benchmark needs 370MB. Both of them do not require refreshing. All objects which would require refreshing in the JLayer benchmark are allocated permanently because their allocation sites are never executed again. In the LuIndex benchmark tick-calls are only executed when all memory expires, as in Figure 3. Both benchmarks benefit from the aggressive space optimization of self-collecting mutators described in Section 3.2. At some point in time, determined by test runs, we turn on the aggressive optimization. For the JLayer benchmark this point in time is after the conversion of ten frames of the mp3 file, resulting in memory consumption decreased by 5MB. We execute the LuIndex benchmark ten times for one measurement, and turn on the optimization after the first round. The reason is that towards the end of the first round classes are loaded dynamically by the VM resulting in memory allocations. We start measuring the time after the first round. In the LuIndex benchmark the effect of the aggressive optimization is significant, it reduces the needed heap size by 120MB.

For the performance measurements of the Monte Carlo benchmarks we use the SCM(50,20) configuration. In our experience this configuration results in the best performance. We measure the performance of the systems with the heap sizes shown in Table 4 as well as with doubled amount of memory. The results are shown in Figure 9. Self-collecting mutators is slightly faster than the garbage-collected systems, even when more memory is available. The sharing of the buffers in self-collecting mutators between concurrent threads does not affect the performance much because the contention on each buffer is low. The JLayer and LuIndex benchmarks were not measured with the SCM(50,20) configura-

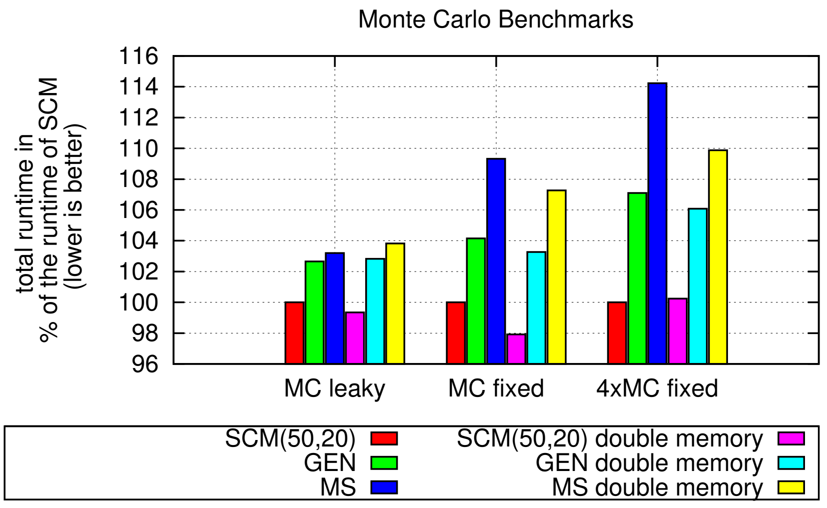

Fig. 9. Total execution time of the Monte Carlo benchmarks in percentage of the total execution time of the benchmark using self-collecting mutators.

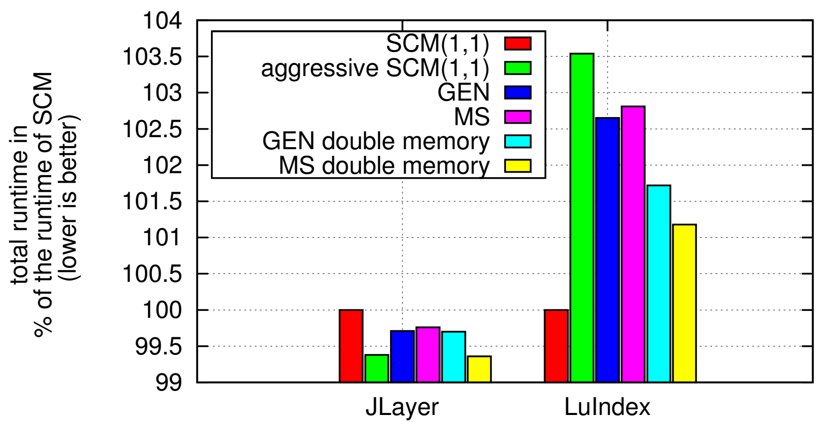

Fig. 10. Total execution time of the JLayer and the LuIndex benchmarks in percentage of the total execution time of the benchmark using self-collecting mutators.

tion since they do not require refreshing and SCM(50,20) induces additional overhead. These benchmarks were measured with the SCM(1,1) configuration as well as with the aggressive space optimization, the results are shown in Figure 10. Note that the aggressive optimization may result in decreased execution time as in the JLayer benchmark or in increased execution time as in the LuIndex benchmark. Self-collecting mutators is competitive to the garbage-collected systems in temporal performance of all benchmarks.

Loop Execution Time and Memory Consumption To measure the pause times of the memory management system and the memory consumption we recorded the loop execution time and the amount of free memory at the end of every loop iteration in the Monte Carlo benchmark.

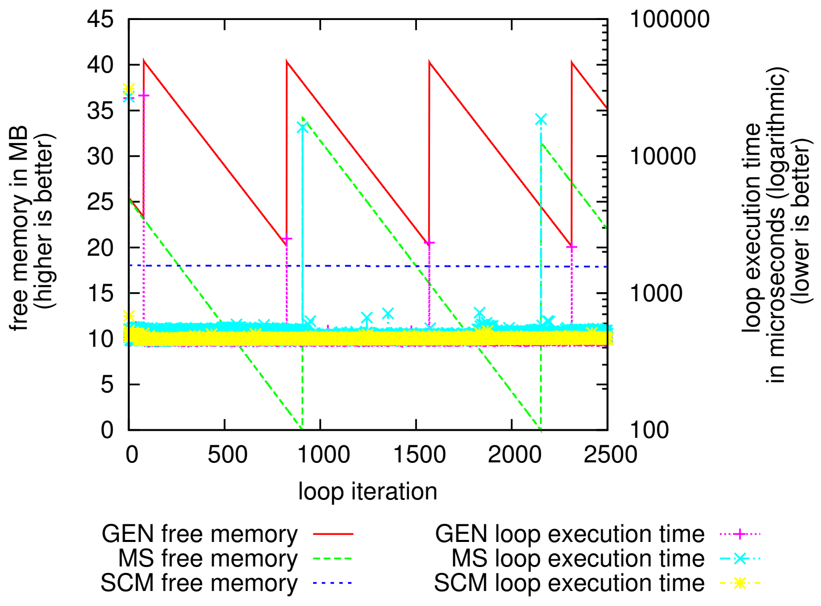

Fig. 11. Free memory and loop execution time of the fixed Monte Carlo benchmark.

Figure 11 shows the free memory and the loop execution time of the fixed Monte Carlo benchmark. The amount of free memory is nearly constant when the benchmark is executed with self-collecting mutators. New result objects are allocated in every loop iteration, but they do not require much space. The loop execution time is nearly constant. It has a jitter of less than 100 microseconds. Both garbage-collected systems have similar loop execution times as self-collecting mutators except for the iterations in which garbage collection is triggered. The loop execution time is much larger then. The free-memory curve of the garbage-collected systems looks like a saw-tooth curve which has a peak after every garbage collection run. The chart depicts the first 2500 loop iterations, further iterations show the same pattern.

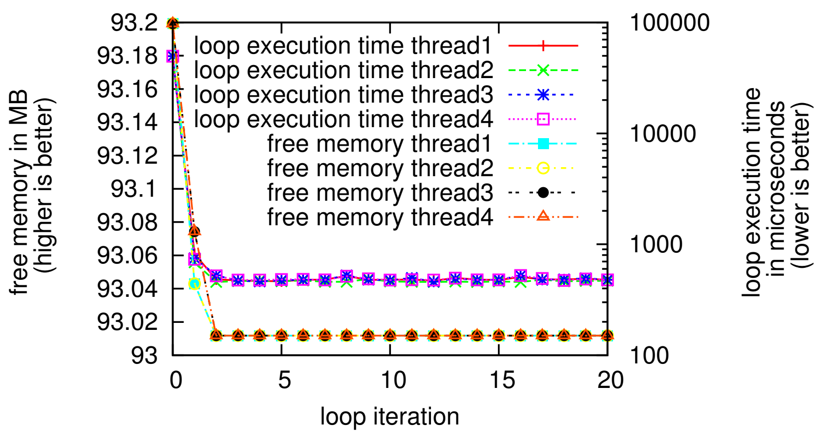

Fig. 12. Free memory and loop execution time of four concurrent instances of the Monte Carlo benchmark using self-collecting mutators.

Next we measure the memory consumption and the loop execution times of self-collecting mutators with four concurrent instances of the Monte Carlo benchmark. Figure 12 shows the first 20 loop iterations. The values representing free memory for a given thread correspond to the overall free memory measured at the end of a loop iteration of the thread. The memory consumption is constant (also for all further iterations), but the system initially requires some loop iterations to find its steady state. Thereafter the buffers of all allocation sites are large enough and no additional memory is needed. The loop execution time still does not vary much.

At last we analyze the time-space trade-off controlled by the number of loop iterations per tick-call. The loop execution times are shown in Figure 13, the free memory over time is visualized in Figure 14. For the measurements we considered three scenarios: tick at every loop iteration, tick at every 50th loop iteration and tick at every 200th iteration. We distributed the required refresh-calls uniformly over all time units to achieve full incrementality. As a result, the loop execution time has only small variance. The results show that the more ticks, and thus the more refreshing happens, the greater the loop execution time is. However, with less ticks memory consumption increases. When a tick-call is executed only every 200th loop iteration, memory consumption is maximal, but temporal performance is much better than in the scenario with one tick every

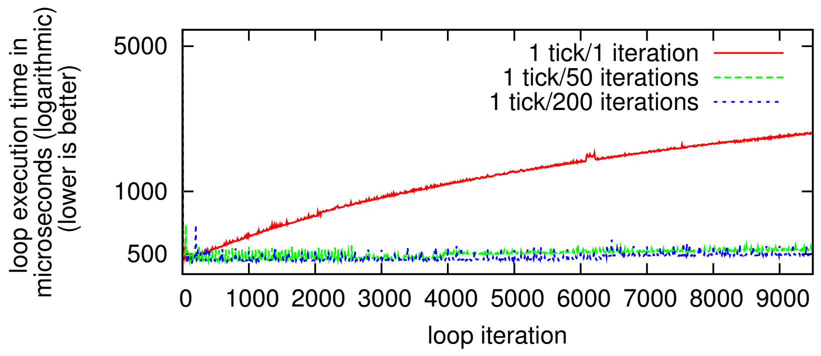

Fig. 13. Loop execution time of the Monte Carlo benchmark with different tick frequencies. Self-collecting mutators is used.

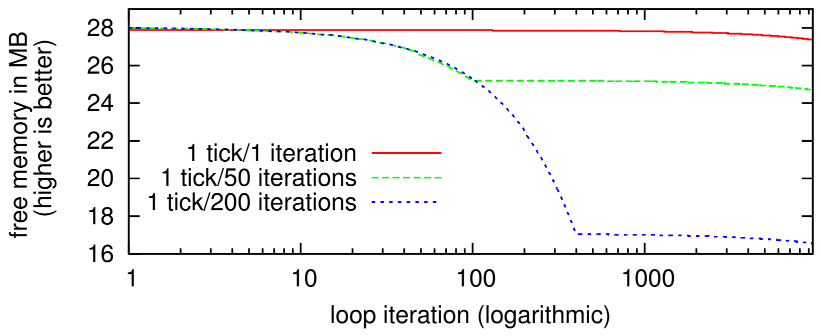

Fig. 14. Free memory of the Monte Carlo benchmark with different tick frequencies. Self-collecting mutators is used.

loop iteration and slightly better than in the scenario with one tick every 50th loop iteration.

Note that the total number of allocated objects increases over time requiring more and more refreshing. This explains not only the obvious increase in loop execution time when ticking every loop iteration but also the slight increase in loop execution time when ticking every 50th and every 200th loop iteration, nearly not visible in the figure. Memory consumption increases as time elapses since a new result object is allocated in every loop iteration. Temporal and spatial performance appear to be inversely proportional, for example, the scenario with the least frequent tick-calls is the fastest because of less refreshing but is the most memory-consuming.

4.2 C Benchmarks

We first discuss the average (min/max) execution time of the self-collecting mutators operations measured during the execution of the mpg123 benchmark. We then compare self-collecting mutators and ptmalloc2 with this benchmark, first in terms of total execution time and then in terms of memory overhead and consumption.

Table 5 shows the execution times in CPU clock cycles of the memory management operations in the mpg123 benchmark averaged over 100 repetitions. The middle and the right column show the results when persistent and short-term memory memory are used, respectively. The results confirm that self-collecting mutators introduce negligible runtime overhead when persistent memory is used. Interestingly, the average execution time of allocation is less (138 cycles versus 166 and 172 cycles) when short-term memory is used probably because here self-collecting mutators allocates and deallocates memory in a different order than when persistent memory is used. The other entries for short-term memory show that the bound on expiration extensions and the descriptor page size do not influence execution times.

The total execution times of the mpg123 benchmark averaged over 100 repetitions are shown in Table 6. We experiment with several configurations of self-collecting mutators using the expiration extension bounds 1 and 10, and the descriptor page sizes 256B and 4KB. Note that in the mpg123 benchmark we only use an expiration extension of zero. In each loop iteration of the benchmark 27 descriptors are created. The total execution time is nearly the same for all configurations, independently of using local-refresh-calls or global-refresh-calls. However, memory overhead and consumption does differ as discussed next.

The memory overhead for storing descriptors and descriptor counters as well as the total memory consumption of the mpg123 benchmark is shown in Figure15. Memory overhead and consumption are measured before and after every malloc-call. The expiration extension bound 10 obviously introduces more memory overhead than the bound 1. The descriptor buffers clearly introduce less overhead with 256B than with 4KB descriptor pages.

The ptmalloc2 system already deallocates and reuses the memory of some objects within one loop iteration. With self-collecting mutators memory consump-

	persistent MM	short-term MM
malloc of ptmalloc2	166 (78 / 199k)	/
free of ptmalloc2	86 (14 / 169k)	/
malloc of SCM	172 (82 / 267k)	138 (75 / 271k)
free of SCM	91 (10 / 157k)	/
local-refresh(1, 256B)	/	227 (131 / 548k)
local-refresh(10, 256B)	/	225 (131 / 548k)
local-refresh(1, 4KB)	/	228 (131 / 548k)
local-refresh(10, 4KB)	/	230 (131 / 548k)
global-refresh(1, 256B)	/	226 (116 / 551k)
global-refresh(10, 256B)	/	224 (116 / 551k)
global-refresh(1, 4KB)	/	227 (116 / 551k)
global-refresh(10, 4KB)	/	228 (116 / 551k)
local-tick(1, 256B)	/	378 (277 / 164k)
local-tick(10, 256B)	/	359 (277 / 71k)
local-tick(1, 4KB)	/	375 (277 / 164k)
local-tick(10, 4KB)	/	366 (277 / 164k)
global-tick(1, 256B)	/	367 (229 / 169k)
global-tick(10, 256B)	/	352 (229 / 151k)
global-tick(1, 4KB)	/	365 (229 / 169k)
global-tick(10, 4KB)	/	361 (229 / 169k)

Table 5. Average (min/max) execution time in CPU clock cycles of the memory management operations in the mpg123 benchmark. Here, e.g. local-refresh(n, m) stands for the local-refresh-call with a maximal expiration extension of n and descriptor page size m. When local/global-refresh is used then the tick-call is denoted by local/global-tick.

ptmalloc2	895.25ms	100.00%
ptmalloc2 through SCM	899.43ms	100.47%
local-SCM(1, 256B)	890.18ms	99.43%
local-SCM(10, 256B)	898.28ms	100.34%
local-SCM(1, 4KB)	892.18ms	99.66%
local-SCM(10, 4KB)	892.28ms	99.67%
global-SCM(1, 256B)	893.76ms	99.83%

Table 6. Total execution times of the mpg123 benchmark averaged over 100 repetitions. Here, local/global-SCM(n, m) stands for self-collecting mutators with a maximal expiration extension of n and descriptor page size m, using local/global-refresh.

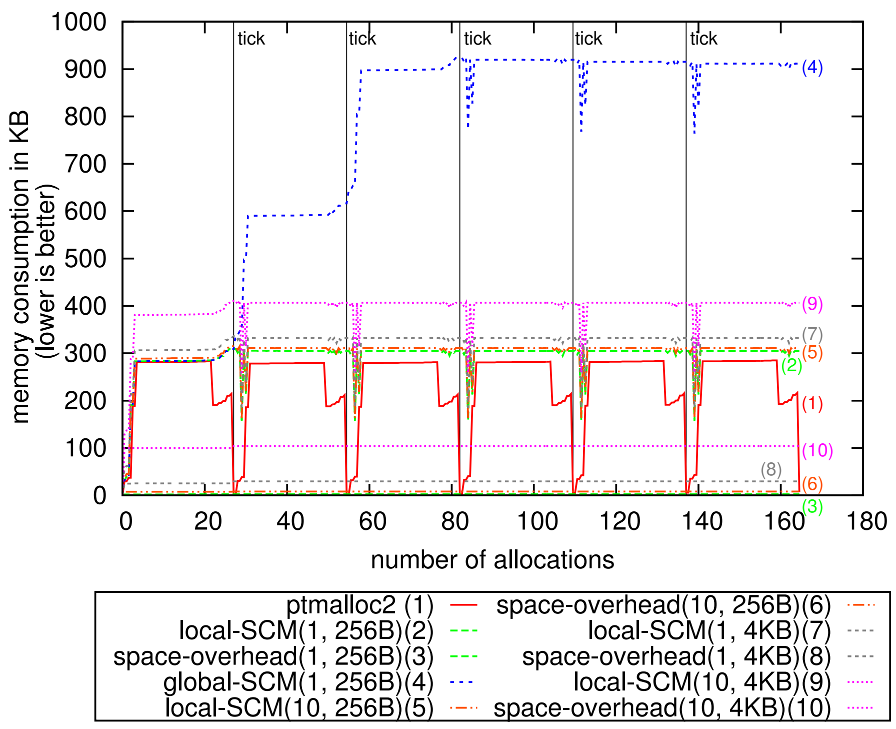

Fig. 15. Memory overhead and consumption of the mpg123 benchmark. Again, local/global-SCM(n, m) stands for self-collecting mutators with a maximal expiration extension of n and descriptor page size m, using local/global-refresh. We write space-overhead(n, m) to denote the memory overhead of the local-SCM(n, m) configurations for storing descriptors and descriptor counters.

tion is higher because objects allocated in one loop iteration are not deallocated before the next loop iteration. The use of the global-refresh-call introduces three times more memory consumption than the use of the local-refresh-call because the descriptors and thereby the corresponding objects expire later.

Self-collecting mutators is also competitive to explicit deallocation in temporal performance of the mpg123 benchmark, at the expense of moderately increased memory consumption. However, self-collecting mutators significantly simplifies memory management usage over explicit deallocation as shown in Table 2.

5 Conclusion and Future Work

We proposed the short-term memory model and presented Java and C implementations of a memory management system called self-collecting mutators that uses the model. In short-term memory objects are allocated with an expiration date, which makes deallocation unnecessary. Self-collecting mutators provides constant-time memory operations, supports concurrency, and performs competitively with garbage-collected and explicitly managed systems. Moreover, short-term memory consumption typically becomes constant after an initial period of time. We presented experiments that confirm our claims in a number of benchmarks.

We informally described a simple translation scheme for porting existing programs to self-collecting mutators. In most of the benchmarks we only had to insert a negligible number of lines of code compared to the total number of lines of code. Using self-collecting mutators was here almost as easy as programming in a garbage-collected system, yet with decreased runtime overhead and improved predictability.

As near-term future work, we plan to implement the multiple-expiration-dates approximation in Java in order to deal with blocking and faulty threads. The challenge will be to support recursive refresh. In the C implementation we plan to finish integrating blocking, resuming, and unregistering threads into the thread management system, and perform further experiments on non-trivial, concurrent benchmarks. Additionally, we started working on implementing the multiple-expiration-dates approximation for Go. The challenge will probably be to maintain the scalability of goroutines.

Medium-term future work may be on fully time- and space-predictable memory management by combining self-collecting mutators with real-time allocators such as Compact-fit [11] and comparing the result with real-time garbage collectors such as Metronome [4]. More long-term research may focus on exploring different time definitions, e.g. based on real time, but also establishing correctness, by providing a program analysis tool for automatic translation of programs to short-term memory or more advanced runtime support.

References

1. Aigner, M., and Haas, A. Short-term memory implementation for C, 2010. http://tiptoe.cs.uni-salzburg.at/short-term-memory/.
2. Aigner, M., Haas, A., Kirsch, C. M., Payer, H., and Sokolova, A. Short-term memory for self-collecting mutators. Tech. Rep. TR 2010–03, University of Salzburg, 2010.
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., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. OOPSLA* (2006), ACM.
7. 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.
8. Bond, M. D., and McKinley, K. S. Leak pruning. In *Proc. ASPLOS* (2009), 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. S., Kirsch, C. M., Payer, H., Sokolova, A., Stadler, H., and Staudinger, R. A compacting real-time memory management system. In *Proc. ATC* (2008), USENIX.
12. Dhurjati, D., Kowshik, S., Adve, V., and Lattner, C. Memory safety without runtime checks or garbage collection. In *Proc. LCTES* (2003), ACM.
13. Greenwald, M. Non-blocking synchronization and system design. Tech. rep., Stanford University, 1999.
14. Jump, M., and McKinley, K. S. Cork: dynamic memory leak detection for garbage-collected languages. In *Proc. POPL* (2007), ACM.
15. Mathew, J. A., Coddington, P. D., and Hawick, K. A. Analysis and development of Java Grande benchmarks. In *Proc. JAVA* (1999), ACM.
16. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM 3*, 4 (1960), 184–195.
17. Nguyen, H. H., and Rinard, M. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proc. ISMM* (2007), ACM.
18. Oancea, C. E., Mycroft, A., and Watt, S. M. A new approach to parallelising tracing algorithms. In *Proc. ISMM* (2009), ACM.

19. 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.
20. Tanenbaum, A. S. *Modern Operating Systems.* Prentice Hall, 2001.
21. Tofte, M., and Talpin, J.-P. Region-based memory management. *Inf. Comput. 132*, 2 (1997), 109–176.