Short-term Memory for Self-collecting Mutators*

Marting Aigner, Andreas Haas, Christoph M. Kirsch,
Hannes Payer, Andreas Schönegger, Ana Sokolova

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

Abstract. We propose a new memory model, short-term memory, and an algorithm that employs it, called self-collecting mutators. In short-term memory objects expire after a certain amount of time, which makes deallocation unnecessary. Self-collecting mutators require programmer support to control the time and thereby enable reusing the memory of expired objects. We identify a class of programs for which programmer support is easy and correctness is guaranteed. We also provide support for multi-threaded applications. Self-collecting mutators perform competitively with garbage-collected systems, as shown by our experimental results on several benchmarks. Unlike garbage-collected systems, our system has no pause times, provides constant execution time of all memory operations, independent of number of live objects, and constant memory consumption after a steady state has been reached.

1 Introduction

Many memory management systems implement one memory model: allocated memory must be deallocated before it can be reused. For deallocation it is necessary to know which objects are not needed anymore. So to say, deallocation information is information about dead objects. We call this memory model the *deallocation memory model*.

When the programmer provides deallocation information, for example by explicit deallocation calls, the memory management system is called explicit. Explicit memory management is known to be error-prone. The two main sources of errors in explicit memory management, memory leaks and dangling pointers, both come from incorrect deallocation information. If an object is deallocated too early, a dangling pointer is created. Deallocating an object too late, i.e., when there is no reference to the object anymore, is not possible and results in a memory leak.

A memory management system which acquires deallocation information by itself is called an implicit memory management system. Garbage collected memory management systems are examples of implicit memory management systems.

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

They use reachability in the object graph as basis for the deallocation information. The complexity of such reachability analysis depends on the number of live objects, which is in the worst case proportional to the size of the heap.

In this report we propose the use of a complementary memory model which we call *short-term memory*. With short-term memory, memory is only allocated for a certain time span. Afterwards, the memory may be reused without additional information. If the memory is required longer, it has to be refreshed. This requires refreshing information, i.e., knowledge of which objects are required longer. Hence, in contrast to deallocation information, refreshing information is information about live objects.

Just like deallocation information, refreshing information can be provided explicitly or gathered implicitly. For example, the reachability analysis provided by a garbage collector can be used for implicit refreshing. It is interesting to note that some garbage collectors (tracing) directly provide refreshing information, whereas others (reference counting) provide deallocation information, c.f. [2].

In this report we focus on explicitly provided refreshing information. Like explicitly provided deallocation information, explicitly provided refreshing information can be incorrect. However, the consequences of an incorrect use are different than in the deallocation memory model. The source of incorrect use of explicit refreshing is missing refreshing information, resulting in memory being reused too early and creating a dangling pointer. Other sources of errors in the deallocation memory model, are avoided in short-term memory:

- Multiple deallocation of the same object is an error in the deallocation memory model, whereas multiple refreshing has no consequence other than wasting time.
- One can never deallocate unreachable objects (source of memory leaks) in the explicit deallocation memory model, whereas it is always possible to refresh reachable objects.

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 can provide such an over-approximation.

In this report we show a case study on the effort of using short-term memory explicitly, and we present an explicit implementation of short-term memory, called self-collecting mutators. With self-collecting mutators, the programmer has to care about the objects by herself. Therefore it is possible that an object expires too early. However, for a non-trivial class of programs self-collecting mutators provide correctness, constant memory consumption, time predictability, and high performance. These properties also hold for multi-threaded applications.

The report presents the following contributions:

- A new memory model, short-term memory.
- An analysis of the use of short-term memory.

- An algorithm that employs short-term memory, the self-collecting mutators algorithm.
- An implementation with support for multi-threaded applications and an implementation analysis.
- A non-trivial well-performing class of programs with correctness guarantees.
- 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 algorithm is presented in Section 3. Section 4 describes programs which are easy to use and perform well with self-collecting mutators. In Section 5 we present experimental results of benchmarks. Section 6 concludes the report and presents future work.

2 Short-term Memory Model

In this section we present the short-term memory model and compare it with the deallocation memory model. A case study shows how much effort it is to use short-term memory, and we give an overview of possible implementations.

2.1 Concepts

In short-term memory allocated objects do not live forever. Each object is only allocated for a certain amount of time. After this amount of time the object expires, which means its existence is not guaranteed anymore. So to say, every object has an expiration date. An object which has an earlier expiration date than another object is called *older*, the other object is called *younger*.

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 is equivalent to a system without deallocation. The definition of time determines some core properties of the memory management system.

Object expiration. With absolute knowledge, an object can be allocated with its exact expiration date. After the expiration date, the object expires. If the expiration date was correctly determined, then such a strategy does not create memory errors. Using exact expiration dates resembles explicit memory management, but may be more difficult than knowing the position of explicit deallocation. On the other hand, for explicit deallocation one requires a pointer to the object, which is not the case here.

Figure 1 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(7) allocates an object for 7 time units.

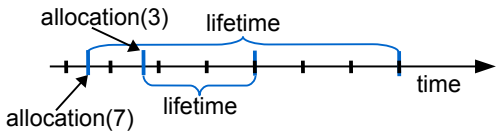

Fig. 1. Allocation with known expiration date.

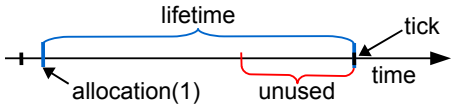

Fig. 2. All objects are allocated for one time unit.

In contrast to using exact expiration dates, without any knowledge, every object can be allocated for one time unit. Time advances when all existing objects are not needed anymore. An example for such an implementation can be seen in Figure 2. All objects have the same expiration date. Even if an object is only used for a short time, it will not expire until the next time advance.

Between these two extremes, if the expiration date of an object can only be estimated, then objects can be allocated for the estimated expiration date and their expiration can later be prolonged by refresh operations. If the program wants to use an object even after its expiration date, it has to refresh it. The refreshed object gets a new extended expiration date. Otherwise it expires. When refreshing is done implicitly, such a system is equivalent to implicit memory management like garbage collection and requires additional runtime.

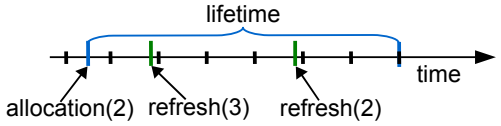

Fig. 3. Allocation with estimated expiration date. If the object is needed longer, it is refreshed.

Figure 3 illustrates refreshing. An object is allocated with an estimated expiration date. If the object is then still needed, it is refreshed. When it is not refreshed before its expiration date, the object expires. In Figure 3, the object exists for seven time units in total. Since it was originally allocated for two time units only, it had to be refreshed for another five, which happens with two refresh statements.

As illustrated in Figure 3, refreshing is used to extend the lifetime of an object after its allocation. In the deallocation memory model, object deallocation is used to get the opposite result: to shorten the lifetime of an object. However, when

all references to an object have been deleted, object deallocation is not possible anymore. Refreshing, on the other hand, is always possible. The program only refreshes objects which are intended to be used again. A program has to keep all objects which it wants to use again reachable anyway. Therefore, the objects which are to be refreshed are always reachable.

The notion of expiration date in the short-term memory model enables trading-off compile-time analysis effort, run-time overhead, and memory consumption. Allocation with known expiration date (cf. Figure 1) requires full compile-time analysis, but least run-time overhead and memory consumption. Allocation for one time unit (cf. Figure 2) requires only light-weight compile-time analysis needed for time control, but introduces additional memory consumption. With refreshing (cf. Figure 3), 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 which builds on the short-term memory model can be used in an incorrect way. Dangling pointers, which are pointers to objects that do not exist anymore, may be created by missing refreshing. The target object expires while the pointers to it still exist. As mentioned in the introduction, it is possible to over-approximate the set of objects which have to be refreshed by refreshing every reachable object. This would result in additional memory consumption, but dangling pointers would not exist then.

Memory leaks are objects which are not needed anymore but are never deallocated. With explicit memory management in the deallocation model, a source of memory leaks is unreachability. With short-term memory, unreachable memory is simply reused because it cannot be refreshed anymore. However, if, for example in a mark-sweep garbage collector, all reachable objects are always refreshed, reachable memory leaks may be created when a programmer does not delete references to objects in the leak. When short-term memory is used explicitly, both kinds of memory leaks do not exist, if only those objects are refreshed which are really needed for future use. We present a benchmark where our explicit implementation of short-term memory repairs a reachable memory leak in Section 5. Similar handling of memory leaks is described in [13].

If time never advances to the expiration date of an object, then a new type of memory leak is created. In our implementation the programmer, possibly supported by static analysis, needs to make sure that time advances. It may also be possible to use real time instead of programmer-controlled time in which case there are no memory leaks, at the expense of more difficult refreshing.

Concurrency. In deallocation memory management 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.

In the short-term memory model, every thread refreshes the objects it uses, just as for single-threaded applications. If more than one thread refreshes an object, the latest expiration date is used. By using this expiration date, all threads can use the object correctly. An object has to expire for all threads at the same time, otherwise an object could expire for one thread although another thread might still need it. Hence, the difficulty of concurrency support for short-term memory comes from the need of time synchronization.

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 the synchronized time also advances if one thread is 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.

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 implicit deallocation always depends on the number of live objects, either in time or in space.

Short-term memory can be used for real-time programs. The lifetime of objects is bounded by their expiration date and therefore the peak memory consumption is bounded too. However, in short-term memory one has to count with the time overhead of refreshing. In addition, in a multi-threaded setting, redundant refreshes of shared objects may happen since every thread refreshes its own objects.

	Deallocation 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
source of errors	invalid explicit deallocation	missing refresh, no time progress
problems with concurrency	deallocation synchronization	time synchronization
problems with real time	implicit deallocation	(redundant) refresh

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

Table 1 summarizes the comparison of the deallocation memory model with the short-term memory model presented in this section.

2.2 Experiments

What we propose in this report is that short-term memory, which has been used now implicitly for a long time by garbage collection, can also be used explicitly and thereby provide interesting properties. 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 have to define a programming model which explicitly uses short-term memory. Most important for such a programming model is the definition of time used for short-term memory. Then we have to define how object allocation and refreshing work. The answer to the second question remains future work.

Explicit Programming Model. We use relative time defined by the user. Time is represented by a counter which is incremented by tick-calls called by the program itself. This implies that the programmer has to put tick-calls at positions in the program code that are always eventually executed. For multi-threaded applications time synchronization is done transparently by the system.

Refreshing is done by explicit refresh-calls, which take two parameters, the object which should be refreshed and the expiration extension. The new expiration date of an object is the current time plus the given expiration extension. Therefore it makes no difference if an object is refreshed once or multiple times within one time unit. For more convenience we provide a recursive refresh-call which refreshes the object given as parameter and all objects reachable from it.

benchmark	LoC	# tick	# refresh	total # of new LoC
Monte Carlo	1450	1	3	6
JLayer MP3 converter	8247	1	6	9

Table 2. Lines of code of the benchmarks, number of tick-calls, number of refresh-calls, and total number of lines of code which had to be added to use short-term memory.

Benchmarks. We translated two Java programs to use our programming model:

1. The Monte Carlo benchmark of the Grande Java Benchmark Suite [11],
2. the JLayer MP3 converter[1].

For both benchmarks we added a tick-call at the end of the main loop. In the Monte Carlo benchmark a result object is generated in every loop iteration which is stored in a result set. The result set is then processed in the finalization phase. This result set requires one recursive refresh-call. A second refresh-call is

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

required to refresh the application root object which is allocated in the main method and exists during the whole program execution. This object is a local object which is only reachable from within the main method. We had to make this object reachable from the code location where refreshing is done, which resulted in two additional lines of code. A third refresh-call is required on an object used for time measurements of the benchmark.

For the JLayer MP3 converter we also have to refresh the application root object. Four other refresh-calls are required for input and output buffers, and another refresh-call is required for a progress listener object.

2.3 Related Work

The short-term memory model can be implemented in many ways, and several existing memory management systems can be seen as implementations of short-term memory. In this section we discuss some of these implementations.

In Section 1 we already stated that a garbage collector can refresh all reachable objects before they expire. Actually, any tracing garbage collector like a mark-sweep garbage collector [12] is an implementation of short-term memory. Time only advances at collection runs, so no objects can expire between two garbage collection runs. In the marking phase of garbage collection all reachable objects are refreshed. At the end of the marking phase time advances. The following sweep phase then deallocates all expired objects. A copying garbage collector [4] does not even require a sweep phase.

Another implementation of short-term memory is stack allocation. Memory which is allocated on the stack is only allocated for a certain amount of time. It expires when the corresponding function returns. However, stack allocation does not use a global time for all objects, but every stack frame has its own time which advances at the end of the function. Refreshing is not generally possible.

Cyclic allocation, as presented in [13], limits the number of objects that are allocated at the same allocation site and exist at the same time. An allocation site is a statement in the code which allocates a new object. For example, if the limit of an allocation site is five objects, then the sixth allocation will overwrite the object that was allocated first. Cyclic allocation is an implementation of short-term memory. Every allocation site has its own local notion of time which advances at every object allocation. The expiration date of a new object is set to k if k is the limit of the allocation site. This implies that the object expires after k further allocations and can then be reused.

The three implementations, tracing garbage collection, stack allocation and cyclic allocation, are existing implementations of short-term memory. In the next section we present self-collecting mutators, which is an explicit implementation of short-term memory. However, one could also think of other implementations such as the following.

Region-based memory management [15] could be used to implement short-term memory. For example, one region could contain all objects which would expire at the same time. For refreshing one would then have to copy an object

from one region to another region. The region-based approach does not depend on a specific definition of time.

In a multi-threaded environment another option is an expiration thread which scans the heap concurrently to program execution and deallocates all expired objects that it finds. This implementation of short-term memory is also orthogonal to any definition of time. The only difficulty is that the expiration thread has to be able to recognize expired objects.

3 Self-collecting Mutators

In this report we present a new explicit implementation of short-term memory, called self-collecting mutators. The motivation for this memory management system was to develop a system with the following properties:

- Competitive performance to systems with garbage collection.
- Constant time complexity for all operations.
- Predictable execution times.
- No read/write barriers.
- No additional threads for memory management.

Note that all these properties focus on temporal performance. Self-collecting mutators is a memory management system which achieves these goals at the expense of increased memory consumption.

3.1 Concepts

Self-collecting mutators is an explicit memory management system. The programmer of an application using self-collecting mutators is responsible for the correctness of program execution.

Most important for the implementation of short-term memory is the definition of time. The expiration of every object depends on the time model. Self-collecting mutators use a logical system time. The programmer controls the logical system time by using a tick operation. Every tick-call increases the logical system time by one. This is done by a simple constant-time integer increment. Thereby the programmer already knows at compile time how fast time will advance according to the progress of the application. Other time models are also possible, see Section 6.

Our design choice for self-collecting mutators is a combination of the design choices shown in Figure 2 and Figure 3. Every object is allocated for exactly one time unit and refreshed by an *expiration extension* if it needs to exist longer. Lifetime of one time unit suffices for many allocations. Every object contains a timestamp in its header, which indicates its expiration date.

The expiration date of an object after refreshing is the current time plus the expiration extension. Therefore it makes no difference if an object is refreshed once or multiple times, the expiration extensions do not accumulate.

Memory Reuse. When an object expires, its memory may be reused. A new object may be allocated in the memory of the expired object. An expiring object has to be refreshed if it should be guaranteed to exist in the next time unit. In our system the memory of an object is only reused by objects allocated at the same allocation site.

Objects which are allocated at the same allocation site are stored in a common buffer. When an object is allocated, the memory management system searches through the buffer to find an expired object which can be overwritten. If none was found, the buffer gets extended and the new object is allocated in the extension. In Section 3.2 we present buffer implementations which support fast selection of expired objects and fast insert and delete operations.

Allocation site buffers allow constant-time selection because all objects allocated at the same allocation site have the same size. The same would be the case if size-class buffers would be used. However, allocation site buffers have an important advantage. The lifetimes of objects of the same allocation site are often similar. Allocation site buffers may therefore require less reordering than size-class buffers.

A drawback of the proposed implementation is that memory which is once allocated by a specific allocation site cannot be reused by any other allocation site. This drawback can be solved in several ways. One could stop the system and start a compaction run, or one could remove unused objects from a buffer and reuse them in a different buffer. However, we did not implement any of these improvements because they are orthogonal to our goals. This issue is part of our future work.

Another drawback of our implementation are allocation sites which allocate arrays of different size. The memory of an array cannot be easily reused because the new array may not fit into the memory of the expired array.

There are several possible solutions to this problem. For example, one can search in the buffer for an expired array which provides enough space, or, one creates a separate buffer for every array size. Resolving the array problem is another issue for future work. For now, we provide a fast implementation at the expense of increased memory consumption. If an expired array does not provide enough memory, we simply remove it from the buffer and extend the buffer to gain memory for the new array. The memory of the expired array could be reused by a different allocation site but is not in the current implementation since the underlying allocator does not allow deallocation. In our benchmarks there are no allocation sites which allocate arrays of different size.

Allocation. Memory allocation consists of three steps:

1. The allocator tries to fetch an expired object from the buffer of the allocation site.
2. If an expired object can be found, its memory is used for the new object. The expired object is then removed from the buffer. Otherwise new memory is allocated from free memory.

3. The new object is initialized. It gets its expiration date, the current logical system time, and is inserted into the buffer.

Free memory is handled by a bump pointer. A single pointer indicates the position where the next object will be allocated. The amount of free memory only decreases because no memory will ever be returned from a buffer to free memory.

Self-collecting mutators can also be based on other allocators, see [9,10,6], which may be necessary to allow some kind of buffer shrinking in future work.

The complexity of allocation depends on the buffer implementation. Our buffer implementation is discussed in Section 3.2. All other, buffer-independent operations take constant time. In particular, bump pointer allocation requires only a pointer increment and an expiration date is assigned to an object by a single statement.

Refresh. Refreshing is the third important operation in our system, after tick and allocation. It is a three-step operation:

1. The object is removed from its buffer;
2. It gets a new expiration date; and
3. It is inserted into the buffer again with its new expiration date.

Refreshing extends the expiration date of a given object. Conceptually, unbounded extensions of the expiration date are possible. However, we decided to limit the maximal expiration extension because this allows better buffer implementations. The design choices for buffer implementations are explained in Section 3.2 in more detail.

Objects can only be overwritten by objects of the same allocation site. Therefore, only those objects have to be refreshed which are stored in the buffer of an allocation site which will be called in the near future. Many programs allocate permanent data at the beginning of a program. Such data need not be refreshed if its allocation sites are never called again. Here we exploit a side-effect of our implementation which makes the system more convenient. Permanent objects could also be handled by repeated refreshing or by introducing an infinite expiration date. Handling this issue is part of our future work.

Most of the time there is not a single object which has to be refreshed but a whole data structure. Refreshing can be inconvenient for the programmer if she has to walk through the object graph recursively by herself. Moreover, the system can provide a more efficient version of such a walk-through because it knows the structure of all objects of the system.

The termination condition has a big influence on the performance of the recursion. For example, if the recursive traversion ends at an already refreshed object, it can happen that not all reachable objects are refreshed. Objects which have not been refreshed can hide behind objects which have already been refreshed by a different refresh call.

A write barrier could prevent such situations. However, this would result in a significant runtime overhead.

The object graph can also be traversed twice, the first traversion marks all reachable objects, and the second iteration refreshes all marked objects.

In our implementation we provide both the one-traversion recursion and the double-traversion recursion. In the benchmarks we only use the one-traversion recursion because it is not difficult for a programmer to avoid situations in which the recursion is incomplete.

Concurrency support. In the following we present the key aspects of our concurrency support:

Buffer sharing. All threads allocate in the same buffers if they execute the same allocation site. Threads can refresh and reuse objects allocated by other threads. A thread cannot reuse the memory of an object as long as another thread keeps refreshing it. If an object is shared between multiple threads the object will exist until the last thread stops refreshing it. Afterwards, it may be reused.

We use locks to control buffer access. Every buffer has its own lock. A thread only gets blocked if another thread executes the same allocation site at the same time.

Thread-local buffers could also be used. Then an object may only be reused by an object allocated at the same allocation site and by the same thread. However, the access to buffers still had to be synchronized because an object can still be refreshed by different threads.

In our experiments we discovered that allocation site buffers are fine-grained enough to allow good scalability with locks. However, the memory of a thread can be reused by a different thread, which results in bad caching performance. Thread-local buffers show better caching performance but require more memory.

Time synchronization. In the concurrency support of self-collecting mutators, every thread has its own thread-local time. The global time is the lowest thread-local time, and determines the expiration of objects. A fast thread is forced to wait for slow threads in order to reuse memory.

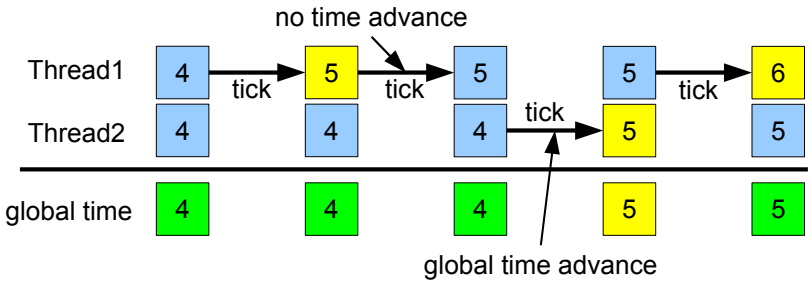

Fig. 4. Thread-local and global time advance in multi-threaded applications.

When the thread-local time of a thread would advance fast, the time difference to the thread-local time of the slowest thread would increase. However, in our implementation we bound the time difference of different threads. The thread-local time of a fast thread cannot be later than the thread-local time of the slowest thread plus one.

The implementation of a tick-call with concurrency support is illustrated in Figure 4. The first tick of Thread1 increases its thread-local time. The next tick does not change its thread-local time because Thread2 has not yet advanced time. When Thread2 ticks the global time advances as well. Thereafter, the thread-local time of Thread1 can increase again. We exploit the bounded difference of thread-local times in our buffer implementation, which is described in Section 3.2.

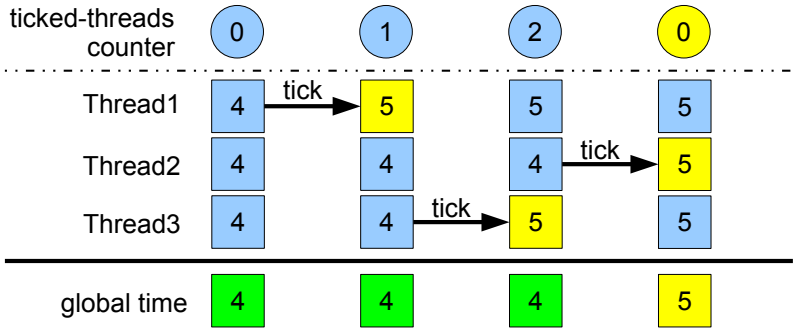

Fig. 5. The calculation of the global time.

Since the time difference of threads is bounded, the global time can be calculated in constant time at every tick. An example of the algorithm synchronizing the time of three threads is shown in Figure 5. We count the number of threads which are faster than the global time. When the number of fast threads is the same as the total number of threads, the global time advances and the counter is reset. The time complexity of the time-advance algorithm is constant. At a tick, a thread has to check if its thread-local time is the same as the global time. If it is the same, it increases its thread-local time and the ticked-threads counter. If all threads have ticked, the global time advances and the counter is reset to zero.

Note that no memory can be reused if the thread-local time of a thread stands still caused by inactivity or unintended thread behavior. This problem also appears in barrier implementations for concurrent applications [14]. Timeouts may be a solution although the objects of the blocking threads must still be handled somehow. At the moment we do not support threads which block for extended periods of time.

In the single-threaded version of self-collecting mutators the new expiration date of an object is the current time of the thread plus the expiration extension.

In the multi-threaded version, the new expiration date of an object must be set to the global time plus one plus the expiration extension. The global time plus one corresponds to the thread-local time of any fast thread and not necessarily to the thread-local time of the executing thread. Using the global time plus one is a conservative approximation of object expiration to ensure that shared objects do not expire prematurely.

3.2 Implementation

We implemented the concepts described above for the programming language Java. We extended the Jikes Research Virtual Machine [1], version 3.1.0, with our system, and we use Gnu Classpath[2] 0.97.2 as class library implementation.

The Jikes RVM is written in Java and uses the same memory management as the program it executes. Nevertheless, we do not handle the objects allocated by Jikes with self-collecting mutators. The memory of Jikes objects is never reused. The reason for this decision is that the memory usage of Jikes is difficult to port to self-collecting mutators while keeping its efficiency. In our benchmarks, Jikes only allocates objects when a program is started and therefore reusing memory would not improve its performance.

We provide two additional functions in the interface of the virtual machine: refresh and tick. The refresh function takes two parameters, a reference to the object which should be refreshed and the expiration extension. The function tick has no parameters. For allocation, we modified the original allocation procedures of Jikes.

Buffers. The buffer implementation is most important for the performance of self-collecting mutators. It has to provide three operations: *insert*, *select expired*, and *delete*. A singly-linked list implementation would provide constant-time *insert*, but *select expired* and *delete* would depend on the size of the buffer. A doubly-linked list improves *delete* to constant time, but *select expired* remains linear in the size of the list.

When sorting by expiration date is applied to the list, the complexity of *select expired* drops to constant time. However, the complexity of *insert* becomes linear in the size of the buffer. By imposing an upper bound on the expiration extension, more efficient implementations are possible. We present two such buffer implementations which exploit the fact of bounded expiration extensions: insert-pointer buffers and segregated buffers.

Insert-pointer buffer. The following observation is the basis for the insert-pointer buffer implementation. If n is the bound of the expiration extension, there are exactly $n+1$ possible insert positions in the buffer to keep it sorted. Of these, n positions are for refreshing and one position is for allocation in the current time unit. For concurrency support one additional insert position is needed, which we discuss later.

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

Pointers to these positions are stored in an additional pointer array. When time advances, the pointer array needs to be updated. However, any insert-pointer can only get one of the following values: the beginning of the live part of the buffer, the value of another existing pointer in the pointer array, or the end of the buffer. For the update, we keep a pointer to the beginning of the live buffer, that is a pointer to the first unexpired object if such exists.

At each time advance objects may expire, which imposes the need of updating the beginning of the live buffer. This is done at the first *insert* after time advance. The new value is one of the existing pointers in the insert-pointer array, or the end of the buffer. This update is linear in n. Using a bitmap reduces the complexity to $\log n$. The new values of the insert pointers can be determined in $\log n$ time as well.

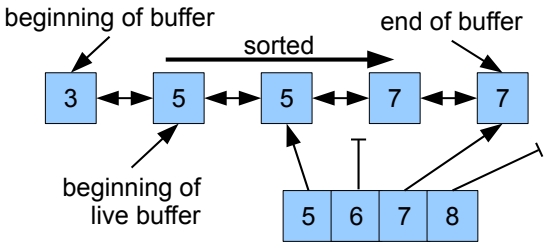

Fig. 6. Insert-pointer buffer implementation.

Figure 6 illustrates the implementation of an insert-pointer buffer. The maximal expiration extension is three, so the insert-pointer array has four positions. The current time is 5. There are pointers to the beginning and to the end of the buffer, the pointer to the beginning of the live buffer and the pointer array with pointers pointing to the insert positions. An insert pointer for a given time value points to the last object in the buffer with this expiration date. Objects with expiration date 6 do not exist in the buffer and therefore the insert pointer 6 has no value. However, the correct insert position for new objects with expiration date 6 is right after the insert position of objects with expiration date 5. After time advance, at time 6, the beginning of the live buffer points to the successor of where pointer 5 points to. In the new time unit, pointer 5 is not needed any longer for objects with expiration date 5. Instead, the pointer is used for objects with expiration date 9, which can now be inserted into the buffer. Hence, the insert pointers correspond to time units modulo the size of the array.

The complexity of *delete* is constant time because the data structure is still a doubly-linked list, and *select expired* is constant time because of the sorting (only the beginning of the buffer needs to be checked). The complexity of *insert* is constant time if the insert pointer is set, or linear in the size of the array if a correct insert position has to be determined. When bitmaps are used to find existing pointers in the array, the worst-case complexity of *insert* drops to $\log n$.

Segregated buffer. The insert-pointer buffer allows to get the oldest object from the buffer. However, this is not necessary. It is enough to find any expired object. This insight is used to get constant-time *insert* at the cost of logarithmic *select expired* in the segregated buffer implementation.

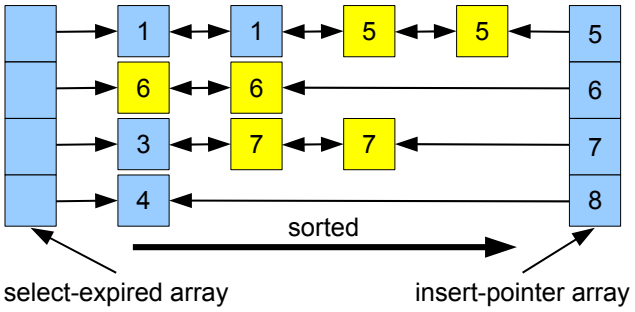

Fig. 7. Segregated buffer implementation.

Segregated buffers are shown in Figure 7. Here, not all objects are in the same doubly-linked list, there are $n+1$ doubly-linked lists. There exists one list for every insert position we had in the insert-pointer buffer before. Segregated buffers use two pointer arrays of size $n+1$. The first array contains pointers to the heads of the lists, this is the select-expired array. The second array is the insert-pointer array which contains pointers to the tails of the lists. Hence, an object is added at the tail of a list. The correct list for inserting an object with a given expiration date is determined modulo the size of the array.

Every doubly-linked list is sorted because when an object is inserted to a list, it is at least as young as the youngest object in the list and it is inserted at the tail. A list can contain objects with different expiration dates but only its youngest objects are not expired.

The complexity of *insert* is constant time because the correct list is determined by a modulo calculation, and objects are only added at the tail. The *delete* operation is again constant time because objects are stored in a doubly-linked list. The complexity of *select expired* is linear in n because in the worst case the head of every list has to be searched for old objects. This complexity again can be reduced to $\log n$ by using a bitmap. The bitmap indicates the buffers which contain expired objects. To select an expired object faster, we store the index of the list where the last expired object was found. The next search for an expired object starts at this position.

A different solution could be based on priority queues [5], which already provide the operations *select expired* and *insert*. However, without the restrictions used by the insert-pointer buffer and the segregated buffer it is not possible to implement a priority queue which provides both *select expired* and *insert* with

a complexity independent on the number of objects in the queue. If it were possible, then sorting in linear time would also be possible.

	insert	delete	select expired
Singly-linked list	$O(1)$	$O(m)$	$O(m)$
Doubly-linked list	$O(1)$	$O(1)$	$O(m)$
Sorted doubly-linked list	$O(m)$	$O(1)$	$O(1)$
Insert-pointer buffer	$O(\log n)$	$O(1)$	$O(1)$
Segregated buffer	$O(1)$	$O(1)$	$O(\log n)$
Priority queue	$O(1)$ / $O(\log m)$	$O(1)$	$O(\log m)$ / $O(1)$

Table 3. Comparison of buffer implementations. The number of objects in a buffer is m, the maximal expiration extension is n.

Table 3 gives an overview of the complexity of the buffer operations in the different buffer implementations.

For the benchmarks we use the segregated buffer implementation with a maximal expiration extension of one time unit. Only few objects exist longer than one time unit, and most objects need not be refreshed since their allocation sites are never called again. An expiration extension of one time unit already allows for incremental usage of refresh.

For concurrency support one additional insert position exists in a sorted buffer to compensate for the time difference between fast and slow threads. The possible insert positions of the fast thread are shifted by one. Therefore, the size of the pointer arrays for both the insert-pointer buffer and the segregated buffer has to be extended to $n + 2$ to support concurrency.

Each doubly-linked list is implemented with a next pointer and a previous pointer in every object header. For refreshing we have to know in which buffer the object is contained. Therefore, the buffer identifier is stored in the header of each object.

Concurrency Support. For the multi-threaded implementation, one has to filter threads which cannot do appropriate tick-calls. For example, Jikes initializes several administrative threads at start-up. Most of the time these threads are inactive. Therefore, they cannot tick appropriately.

To distinguish application threads from such administrative threads, we let threads register during which they get their thread-local time. Registration happens automatically at the first memory operation of the thread. At registration, a thread gets the global time as its thread-local time. A thread is automatically unregistered upon termination, or it can unregister explicitly. A problem with thread unregistrations is how to secure its objects. One solution would be that a thread adds a special tag to its objects so that they cannot be reused. The thread removes the tag when it is activated again.

Memory overhead. Our system has the following sources of memory overhead:

- The expiration date and the buffer identifier are stored in one word in the header of every object. When the highest possible global time is reached, the time wraps around and starts with zero again.
- The next pointer and the previous pointer for the doubly-linked list require two more words in the object header.
- Every allocation site has one segregated buffer with a maximal expiration extension of n. For multi-threading every segregated buffer consists of $n+2$ doubly-linked lists with two words overhead each (for the pointers in the arrays), and one word for the index of the next buffer to search. We therefore have a total overhead of seven words per allocation site in our implementation ($n = 1$).
- For time definition, we have one word for the global time, one word for the ticked-thread counter, and one word for the thread-local times.
- For multi-threading, we need one lock per allocation site and one lock for time synchronization.

Memory consumption of concurrent programs. The memory consumption per thread in a concurrent program increases if another thread is slower. Nevertheless, the memory consumption per thread is bounded by the amount of memory it can allocate between two ticks of the slowest thread. After a tick of the slowest thread, fast threads reuse their objects again.

Runtime overhead for arbitrary programs. The runtime overhead of self-collecting mutators consists of the overhead of tick-calls and the overhead of refreshing. Since ticking is fast, tick-calls do not introduce much overhead.

There is no upper bound on the number of tick-calls in a program, but there is a lower bound. Namely, the number of garbage collection runs in a mark-sweep garbage collector [12] provides the lower bound since otherwise the heap would be full and nothing could be allocated any more.

When the number of tick-calls increases the memory consumption decreases, but more refresh-calls are necessary, which implies time overhead. With the number of tick-calls the programmer can trade-off time overhead of additional refreshing and space overhead of unused memory. We present the effect of the choice of tick frequency in our experiments in Section 5.

Refreshing adds time overhead in every time unit. However, an object is only refreshed once in a time unit by a single thread. Moreover, in a multi-threaded setting, we avoid redundant refreshes by checking whether an object was already refreshed. The complexity of refreshing is therefore bounded by the number of refreshed objects.

Incrementality. As just discussed, objects only have to be refreshed once each time unit. However, the exact time of refreshing in a time unit does not matter. The refresh-calls can be distributed randomly in the time unit. Therefore, fine-grained incrementality can be achieved.

3.3 Related Work

As already stated in the related work of short-term memory, the work presented in [13] also describes the use of buffers per allocation site with the intention to eliminate memory leaks. There, cyclic buffers whose size is determined in experiments are used. Self-collecting mutators determine buffer sizes dynamically depending on tick-calls. Moreover, they provide refresh-calls to trade-off space consumption caused by sparse tick-calls and time consumption caused by required refresh-calls.

The memory management system described in [13] maintains type safety as self-collecting mutators do. Other work which provides memory management type safety to support the design of non-blocking thread synchronization algorithms is reported on in [8]. In [7] the authors propose the use of type-safe pool allocation to support program analysis.

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

4 Suitable Programs

There exists a class of programs, called suitable programs, which need only few changes in order to use self-collecting mutators. Moreover, when self-collecting mutators are used for such programs, the time performance improves, the memory consumption is constant, and no pause times are introduced by the memory management system. A sufficient compiler test can be provided to guarantee correctness, meaning that no dangling pointers are created. We chose our benchmarks from this class of programs as described in Section 5. To some extend these properties result from the side effect of allocation site buffers that the memory of objects is not reused when their allocation site is not called again.

Any suitable program consists of three phases:

1. Initialization phase;
2. Main loop;
3. Finalization phase.

The program starts with an initialization phase in which permanent data is allocated. The initialization phase must not share any code with the succeeding phases. In the main loop the program works on the data created in the initialization phase. In addition, the main loop can allocate any amount of memory. The finalization phase, in which the program works on the data generated in the main loop, is optional.

This basic structure can be extended. All three phases can contain code which has the same structure. For example, part of the main loop can be code which has the structure described above, with an inner loop serving as a sub-main loop.

To adapt such a program for using self-collecting mutators, a tick-call has to be added at the end of the main loop to ensure time advance. When objects

which are allocated in the main loop do not exist longer than one loop iteration, no refreshing is required. Objects which have been allocated in the initialization phase do not have to be refreshed, although they are expired. Their memory will not be reused anyway. This is the side-effect of our implementation already described in Section 3.2. Only those objects have to be refreshed which are allocated in the main loop and which should exist longer than one loop iteration. For best performance, the number of refresh-calls should be low.

Multi-threaded applications are suitable if all threads are suitable. For better memory efficiency, the execution time of one iteration of the main loop of each thread should be similar.

The correctness of a program can be checked by the compiler. A sufficient test using escape analysis can be provided to check if objects which exist longer than one loop iteration are refreshed correctly.

The described class of programs also performs well with other memory management systems like region-based memory management [15]. However, in region-based memory management it can happen that a single live object keeps a whole region alive. In such a case, self-collecting mutators would only refresh the single live object whereas the other objects expire.

5 Experimental Setup and Evaluation

CPU	2x AMD Opteron DualCore, 2.0 GHz
RAM	4GB
OS	Linux 2.6.24-16
Java VM	Jikes RVM 3.1.0
initial heap size	50MB

Table 4. System configuration.

We ran the benchmarks on a platform described in Table 4. We used the production configuration of Jikes for the measurements of the total runtime. For the other measurements we used the baseline configuration without runtime optimization. The reason is that we want to show the effects of the memory management without side-effects of pause times introduced by the optimizer. We compare self-collecting mutators with a 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 [3].

We executed two benchmarks explained in Section 2.2, the Monte Carlo benchmark and the JLayer MP3 converter. Because of the allocation site buffers we were able to reduce the number of refresh-calls. To reduce the number of refreshes even more in the Monte Carlo benchmark we preallocated the result set so that no refresh has to be done for it. Table 5 shows the number of allocation sites and the imposed space overhead for system management. The system

benchmark	LoC	total number of changed LoC	allocation sites	system overhead
Monte Carlo	1450	10	101	811 words
JLayer MP3 converter	8247	1	312	2499 words

Table 5. Lines of code of the benchmarks, the effort of adapting them for self-collecting mutators, and the space overhead.

overhead only contains the overhead at program startup, but not the overhead of the object headers. The system overhead consists of 7 words per allocation site for buffer handling, plus 1 word per allocation site for the buffer lock, plus 3 words for the global time, the global time lock, and the ticked-thread counter. Hence it amounts to 8 words per allocation site plus 3 additional words.

We measured the total runtime of the benchmarks, the latency of the memory management system, and the memory consumption over time. To test the runtime properties of the concurrency support we execute both the Monte Carlo benchmark and the JLayer benchmark in parallel. Moreover, we start four instances of the Monte Carlo benchmark at the same time to show that the shared use of allocation sites is possible. Finally, we show the overhead of refreshing and the effect of the number of tick-calls on the memory consumption of a program.

5.1 Total runtime

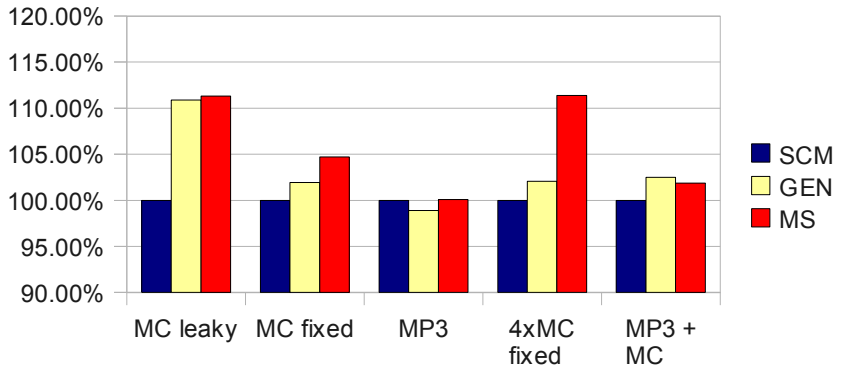

Fig. 8. Total runtime of the benchmarks in percent of the runtime of the benchmark using self-collecting mutators. The production configuration of Jikes is used.

The total runtime of the benchmarks is presented in Figure 8. We executed the benchmarks ten times and calculated the average of the execution times. The

original Monte Carlo benchmark has a memory leak which is not collected by a garbage collector because it is still reachable. Self-collecting mutators (SCM) reuse the memory objects in the memory leak when they expire. With the memory leak, self-collecting mutators is 10% faster than the generational garbage collector (GEN), and 11% faster than the mark-sweep garbage collector (MS). We also modified the Monte Carlo benchmark and removed the memory leak. Self-collecting mutators remain (slightly) faster than the garbage-collected systems in the modified Monte Carlo benchmark. Self-collecting mutators are also competitive in the execution time of the MP3 converter and the parallel execution of the MP3 converter and the fixed Monte Carlo benchmark. When four instances of the Monte Carlo benchmark are executed in parallel, garbage collection is triggered often. This results in a performance drop of the mark-sweep garbage collector. The garbage collection overhead of the generational garbage collector is nearly the same as the buffer management overhead of self-collecting mutators. Note that in the multi-threaded benchmarks we used 100MB initial heap size because of the increased memory consumption.

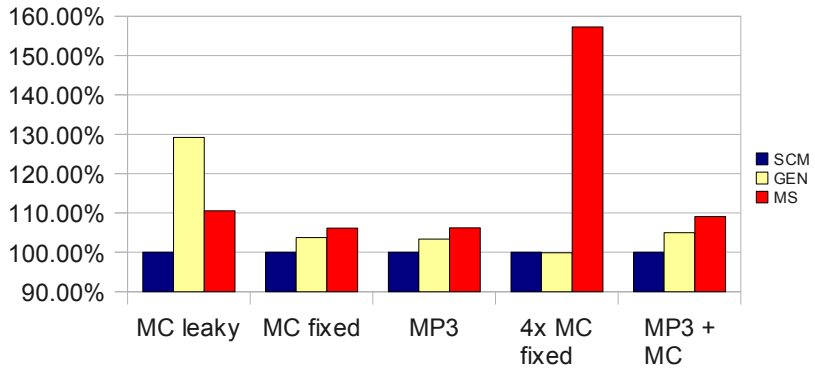

Fig. 9. Total runtime of the benchmarks in percent of the runtime of the benchmark using self-collecting mutators. The baseline configuration is used.

Figure 9 shows the same benchmarks with the baseline configuration of Jikes. The garbage collection algorithms benefit more from the optimizations than the self-collecting mutators. The main difference is visible in the leaky Monte Carlo benchmark for the generational garbage collector, and in the multi-threaded Monte Carlo benchmark for the mark-sweep garbage collector. For the other benchmarks the results are similar.

5.2 Pause times 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 beginning of every loop iteration.

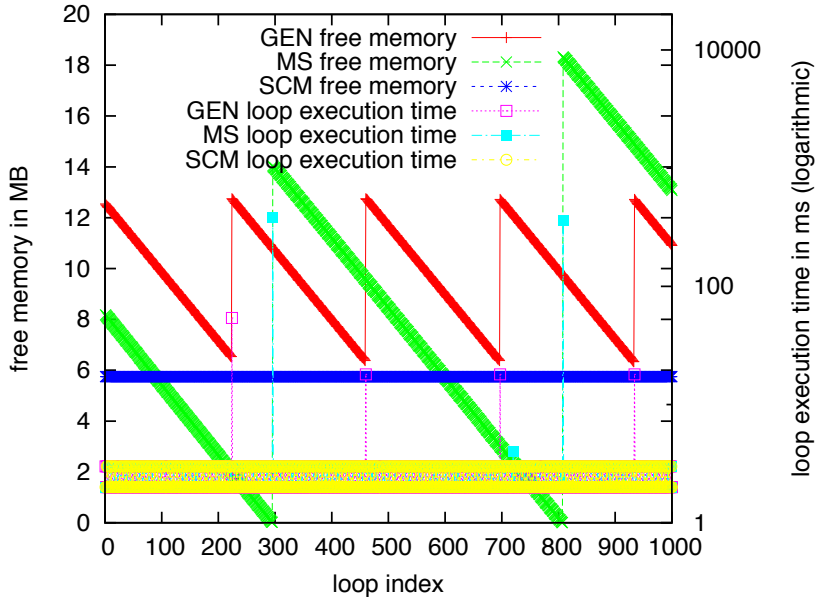

Fig. 10. Free memory and loop execution time of the fixed Monte Carlo benchmark using the baseline configuration.

Figure 10 shows the free memory and the loop execution time of the fixed Monte Carlo benchmark. The amount of free memory is constant when the benchmark is executed with self-collecting mutators, and the loop execution time is nearly constant. It has a jitter of one millisecond. Both garbage-collected systems have the same loop execution time 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 only shows the first thousand loop iterations, further iterations show the same pattern.

The measurements were done with the baseline configuration because in the production configuration of Jikes the virtual machine itself also requires memory. This can be seen in Figure 11. When the virtual machine only executes the code of the benchmark the memory consumption is constant, also in the production configuration. However, when Jikes runs optimization routines, it allocates mem-

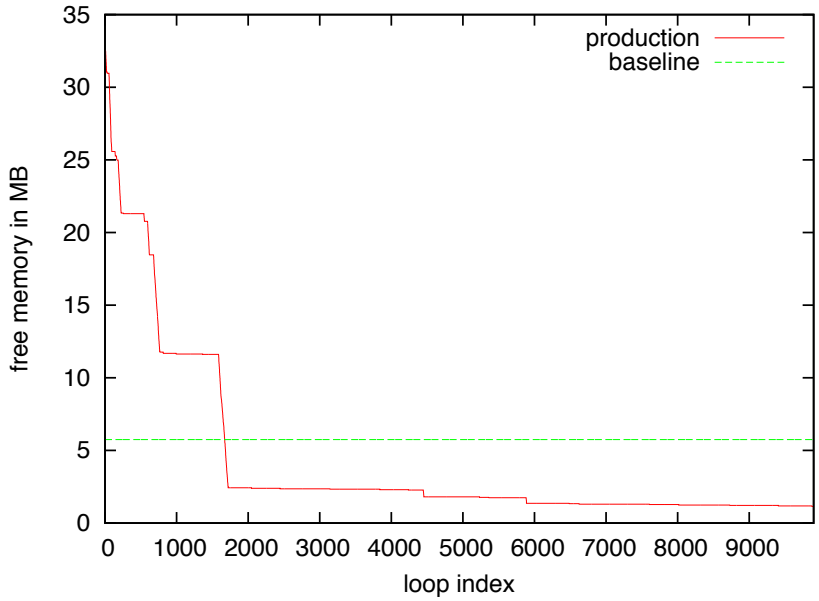

Fig. 11. Comparison of the free memory of the production configuration and the baseline configuration, for the fixed Monte Carlo benchmark.

ory that cannot be reused since it does not itself use self-collecting mutators. In the baseline configuration no optimizations are executed, so the amount of free memory is constant over the whole execution time.

Next we measure the memory consumption and the loop execution times of self-collecting mutators of the parallel Monte Carlo benchmark. Figure 12 shows the first 20 loop iterations. The values representing free memory for a thread correspond to the overall free memory measured at the end of a loop iteration for the considered thread. The memory consumption is constant (also for all further iterations), but the system 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 tick-calls. We started a Monte Carlo benchmark which does not preallocate the result set and compared it with the benchmark which does preallocation. The loop execution times are shown in Figure 13, the free memory over time is visualized in Figure 14. With preallocation (and tick at every loop iteration) we get the best memory consumption at the end of the benchmark execution and the lowest loop execution time. Without preallocation all result objects have to be refreshed in every time unit. For the measurements we considered three scenarios: tick at every loop iteration, tick at every 50th loop iteration and tick at every 200th iter-

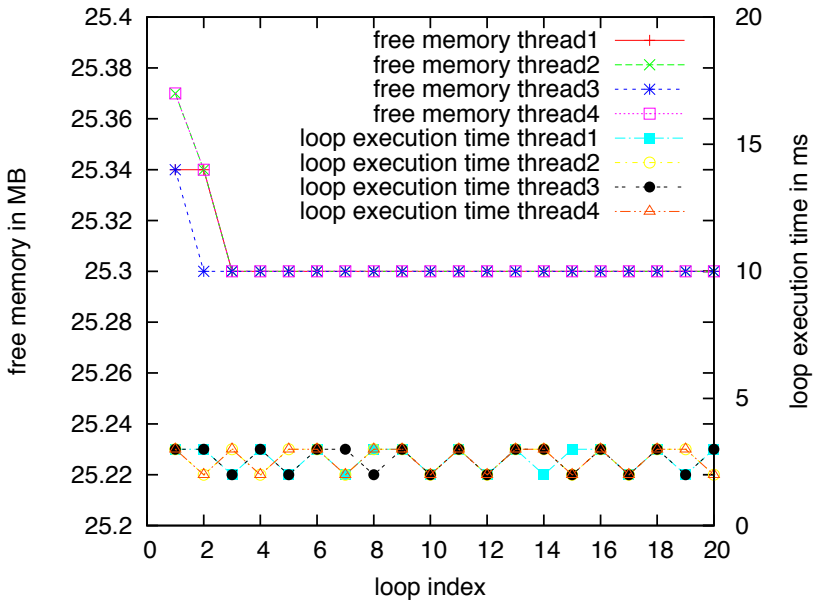

Fig. 12. Free memory and loop execution time of the parallel Monte Carlo benchmark using the baseline configuration.

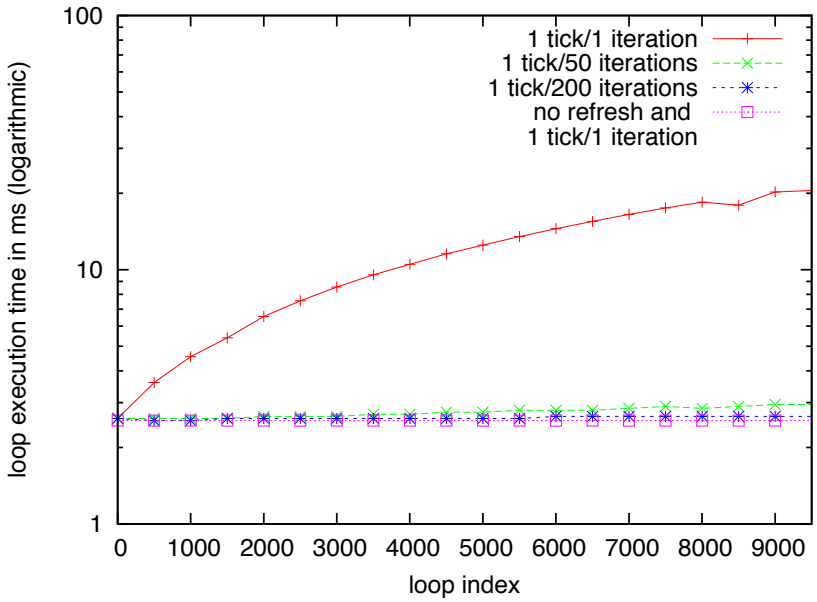

Fig. 13. Loop execution time of the Monte Carlo benchmark with different tick frequencies. The baseline configuration is used.

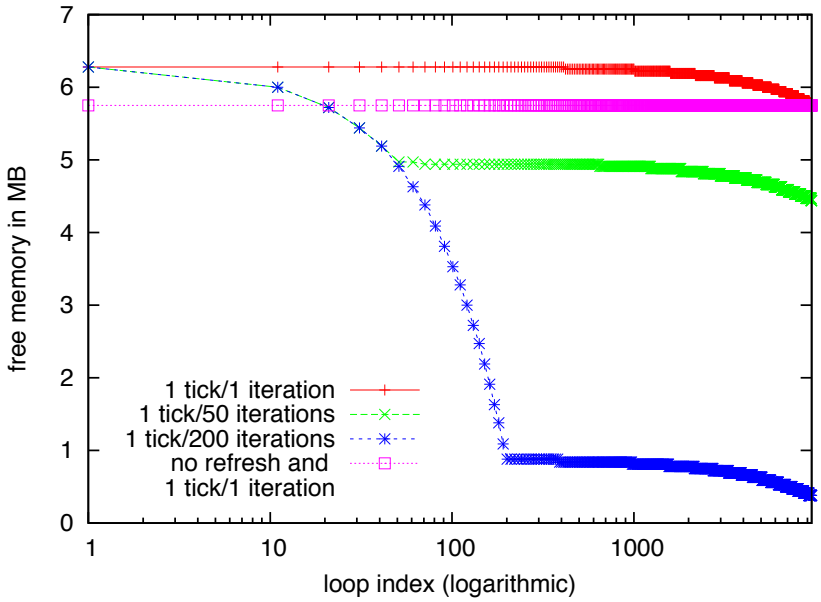

Fig. 14. Free memory of the Monte Carlo benchmark with different tick frequencies. The baseline configuration is used.

ation. We distributed the required refresh-calls uniformly over all time units. As a result, the loop execution time has only small variance. The results show that the more ticks, and thus more refreshing, the longer the loop execution time. However, with less ticks the memory consumption increases. When a tick-call is executed only every 200th loop iteration, the memory consumption is maximal, but the performance is nearly as good as the performance of the system with preallocation. In the last iterations the number of allocated objects is large and therefore exactly fifty refresh-calls are executed per iteration. This explains the slight increase in the loop execution time. The memory consumption of the benchmarks without preallocation increases as time elapses since a new result object is allocated in every loop iteration. After the last iteration the memory consumption of the Monte Carlo benchmark with preallocation and the Monte Carlo benchmark with a tick-call at every loop iteration is the same. The execution which ticks only once every 200th loop iteration needs nearly the whole heap.

6 Conclusion and Future Work

We proposed the short-term memory model and a memory management system that uses it, self-collecting mutators. In short-term memory objects are allocated with an expiration date. Self-collecting mutators achieve constant-time memory

operations. Moreover, the memory consumption becomes constant after an initial period of time. The self-collecting mutators algorithm also provides concurrency support. We present benchmarks that confirm the properties of self-collecting mutators.

We identified a class of programs for which programmer support is easy and correctness is guaranteed. In one of the benchmarks we only had to insert one line of code to start it with self-collecting mutators. For this class of programs, using self-collecting mutators is almost as easy as programming in a garbage-collected system, yet with decreased runtime overhead and improved predictability.

As future work we aim at providing a program analysis tool which helps the programmer to write correct and efficient programs for self-collecting mutators. Other issues are the implementation of buffer shrinking so that unused memory can be shifted between buffers of different allocation sites. As discussed in the introduction of short-term memory, the key of using short-term memory for multi-threaded applications is time synchronization between threads. In the future we plan to explore different time definitions that may allow for better time synchronization, which may improve the handling of inactive threads.

References

1. 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.
2. Bacon, D. F., Cheng, P., and Rajan, V. T. A unified theory of garbage collection. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2004), ACM, pp. 50–68.
3. 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.
4. Cheney, C. J. A nonrecursive list compacting algorithm. *Commun. ACM 13*, 11 (1970), 677–678.
5. 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.
6. Craciunas, S., Kirsch, C., Payer, H., Sokolova, A., Stadler, H., and Staudinger, R. A compacting real-time memory management system. In *Proc. USENIX ATC* (2008).
7. Dhurjati, D., Kowshik, S., Adve, V., and Lattner, C. Memory safety without runtime checks or garbage collection. In *Proc. LCTES* (2003), ACM.
8. Greenwald, M. Non-blocking synchronization and system design. Tech. rep., Stanford University, 1999.
9. Knuth, D. E. *Fundamental Algorithms*, second ed., vol. 1 of *The Art of Computer Programming.* Addison-Wesley, 1973.
10. Lea, D. A memory allocator. Unix/Mail/, 6/96, 1996.

11. Mathew, J. A., Coddington, P. D., and Hawick, K. A. Analysis and development of Java Grande benchmarks. In *Proc. JAVA* (1999), ACM.
12. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM 3*, 4 (1960), 184–195.
13. Nguyen, H. H., and Rinard, M. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proc. ISMM* (2007), ACM.
14. Tanenbaum, A. S. *Modern Operating Systems*. Prentice Hall, 2001.
15. Tofte, M., and Talpin, J.-P. Region-based memory management. *Inf. Comput. 132*, 2 (1997), 109–176.