Short-term Memory for Self-collecting Mutators: Towards Time- and Space-predictable Virtualization

Andreas Haas, Christoph Kirsch, Hannes Payer, Andreas Schoenegger, Ana Sokolova
Time-predictable virtualization: process response times and jitter are bounded per process, independently of any other processes.
Space-predictable virtualization: (shared) memory usage and fragmentation are bounded per process, independently of any other processes.
Time- and space-predictable virtualization enables time- and space-compositional software processes.
Giotto

HTL
[EMSOFT 2006, RTSS 2009]

Exotasks
[LCTES 2007, TECS 2009]

Variable-Bandwidth Servers
[IIES 2009, SIES 2009, RTAS 2010, Submitted]

Compact-fit
[USENIX ATC 2008]

Short-term Memory
[Submitted]

Time

Space
Short-term Memory
Traditional Memory Model

- Allocated memory objects are guaranteed to exist until deallocation

- **Explicit** deallocation is fast but not safe and error-prone (dangling pointers and memory leaks)

- **Implicit** deallocation (unreachable objects) is safe but its performance is proportional to heap size and still correctness is not guaranteed (memory leaks)
Short-term Memory

- Memory objects are only guaranteed to exist for a finite amount of time
- Memory objects are allocated with a given expiration date
- Memory objects are neither explicitly nor implicitly deallocated but may be refreshed to extend their expiration date
Short-term Memory

The diagram illustrates the concept of short-term memory, showing objects that are either safe or require refreshment over time. Objects labeled 14, 15, and 16 are interconnected, indicating their relationship and the process of refreshment and expiration.
With short-term memory, programmers specify which memory objects are still needed and not which memory objects are not needed anymore!
Full Compile-Time Knowledge

Figure 1. Allocation with known expiration date.
Maximal Memory Consumption

Figure 2. All objects are allocated for one time unit.
Trading-off Compile-Time, Runtime, Memory

Figure 3. Allocation with estimated expiration date. If the object is needed longer, it is refreshed.
Self-collecting Mutators
SCM

- Self-collecting mutators (SCM) is an explicit memory management system:
  - `new(Class)`
  - `refresh(Object, Extension)`
  - `tick()`
Memory Reuse

• When an object expires, its memory may be reused but only by an object allocated at the same allocation site:
  ▶ type-safe but not necessarily safe!

• Objects allocated at the same site are stored in a buffer (insert, delete, select-expired)
1. *Select* an *expired* object, if there are any, and *delete* it from the *buffer*, or else, if there are none, allocate memory from *free memory*

2. Assign the *current logical system time* to the object as *expiration date* and *insert* it into the buffer

- Free memory is handled by a bump pointer
Refresh

1. **Delete** object from its buffer
2. Assign **new** expiration date
3. **Insert** object back into the buffer

- Expiration extensions are **bounded** by a constant in our implementation
- Side-effect: objects allocated at allocation sites that are only executed once are **permanent** and do not require refreshing
Time Advance

- The current logical system time is implemented by a global counter
- Time advance: increment the counter by one modulo a wrap-around
- We also support multi-threaded applications
Implementation
Complexity Trade-off

Table 2. Comparison of buffer implementations. The number of objects in a buffer is \( m \), the maximal expiration extension is \( n \).
Insert-pointer buffer
(with bounded expiration extension $n=3$)

Figure 6. Insert-pointer buffer implementation.
Segregated buffer
(with bounded expiration extension $n=3$
and unsorted select-expired)

Figure 7. Segregated buffer implementation.
Experiments
part of the main loop can be code which has the structure contain code which has the same structure. For example,

gram works on the data generated in the main loop. is opy

amount of memory. The finalization phase in which the proy

zation phase. In addition, the main loop can allocate any

loop the program works on the data created in the initialy

not share any code with the succeeding phases. In the main

permanant data is allocated. The initialization phase must

Section 7.

our benchmarks from this class of programs as described in

cient compiler test can be provided to guarantee correctnessx

introduced by the memory management system. A suffiy

memory consumption is constant and no pause times are

for such programs, the time performance improves the

mutators. Moreover, when self-collecting mutators are used

which need only few changes in order to use self-collecting

There exists a class of programs, called suitable programs, which can be achieved.

randomly in the time unit. Therefore, fine-grained incremeny

time unit does not matter. The

each time unit. However, the exact time of refreshing in a

As just discussed, objects only have to be refreshed once

bounded by the number of refreshed objects. The complexity of refreshing is therefore

redundant refreshes by checking whether an object was aly

gle thread. Moreover, in a multi-threaded setting, we avoid

ever, an object is only refreshed once in a time unit by a siny

in Section 7.

the effect of the choice of tick frequency in our experiments

freshing and space overhead of unused memory. We present

the programmer can tradeoff time overhead of additional rey

which implies time overhead. With the number of

sumption decreases, but more

would be full and nothing can be allocated any more.

garbage collection runs in a mark-sweep garbage collector

Table 4.

Monte Carlo       JLayer MP4

\begin{tabular}{|l|l|}
\hline
CPU & 2x AMD Opteron DualCore, 2.0 GHz \\
\hline
RAM & 4GB \\
\hline
OS & Linux 2.6.24-16 \\
\hline
Java VM & Jikes RVM 3.1.0 \\
\hline
initial heap size & 50MB \\
\hline
\end{tabular}

\textbf{Table 3.} System configuration.
The program starts with an initialization phase in which any suitable program consists of three phases. To adapt such a program for using self-collecting mutators, the space overhead of unused memory is bounded by the number of refreshed objects. The complexity of refreshing is therefore not share any code with the succeeding phases.

Refreshing adds time overhead in every time unit. However, the exact time of refreshing in a tick unit does not matter. The programmer can tradeoff time overhead of additional refreshing and space overhead of unused memory. We present a sufficient test using escape analysis to check if objects which exist longer than one loop iteration are refreshed correctly. The correctness of a program can be checked by the compiler test can be provided to guarantee correctness. We chose to implement an Immix collector [4] which has the structure copying collector where the higher generation is handled by the standard garbage collector of Jikes, a two-generation collector. Mutators are refreshed which are allocated in the main loop and which have been allocated in the initialization phase do not have to be refreshed, although they are expired. Their memory will not be reused in the initialization phase. In the main loop which need only few changes in order to use self-collecting mutators, the number of tick calls increases the memory consumption decreases, but more redundant refreshes by checking whether an object was already refreshed. This basic structure can be extended. All three phases can be code which has the same structure. For example, the finalization phase works on the data generated in the main loop.

Section 7

As just discussed, objects only have to be refreshed once. The effect of the choice of tick frequency in our experiments has the same structure. For better memory efficiency the execution time of one iteration of the main loop of each thread should be measured. Any suitable program consists of three phases.

### 4.2 Incrementality

As just discussed, objects only have to be refreshed once. Therefore, fine-grained incrementality can be achieved.

### 5. Suitable Programs

Time optimization is the main focus of this work. The mutators are ready refreshed. The complexity of refreshing is therefore bounded by the number of refreshed objects. The system overhead is 811 words for Monte Carlo and 2499 words for JLayer MP3 converter. The memory consumption is constant and no pause times are introduced by the memory management system.

### 6. Experimental Setup and Evaluation

We discuss these in more detail in Section 8.

#### Table 4

<table>
<thead>
<tr>
<th>benchmark</th>
<th>LoC</th>
<th>added LoC</th>
<th>allocation sites</th>
<th>system overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monte Carlo</td>
<td>1450</td>
<td>10</td>
<td>101</td>
<td>811 words</td>
</tr>
<tr>
<td>JLayer MP3 converter</td>
<td>8247</td>
<td>1</td>
<td>312</td>
<td>2499 words</td>
</tr>
</tbody>
</table>

**Table 4.** Lines of code of the benchmarks, the effort of adapting them for self-collecting mutators, and the space overhead.
Runtime Performance

Figure 8. Total runtime of the benchmarks in percent of the runtime of the benchmark using self-collecting mutators.
Latency & Memory

We executed two benchmarks: the Monte Carlo benchmark of the Grande Java Benchmark Suite and the JLayer MP: converter. 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. This result set is then processed in the finalization phase. To reduce memory consumption, we changed the Monte Carlo benchmark to preallocate the result objects before the main loop. Empty objects are then filled with data. Refreshing can be used instead of preallocation, but this implies significant runtime overhead. We present the overhead of refreshing at the end of the section.

The MP: encoder did not require additional modifications. The added number of lines of code per benchmark is shown in Table 2, which also shows the number of allocation sites and the imposed space overhead for system management.

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.

**Figure 8.** Total runtime of the benchmarks in percent of the runtime of the benchmark using self3collecting mutators.

We modified the Monte Carlo benchmark and removed the memory leak. Self3collecting mutators remain slightly faster than the garbage-collected systems in the modified Monte Carlo benchmark. The same applies to the MP: converter and the parallel execution of the MP: 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 mark3sweep garbage collector. The garbage collection overhead of the generational garbage collector is nearly the same as the locking overhead of self3collecting mutators.

**Figure 9.** Free memory and loop execution time of the fixed Monte Carlo benchmark.

The amount of free memory is constant when the benchmark is executed with self3collecting 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 self3collecting mutators except for the iterations in which garbage collection is triggered. The loop execution time is much larger. The free memory curve of the garbage-collected systems looks like a sawtooth 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. Next, we measure the memory consumption and the loop execution times of self3collecting mutators of the parallel Monte Carlo benchmark. Figure 9 shows the first 9 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 but the system requires some loop iterations to find its steady state. Thereafter, the buffers of all allocation sites are large.

**Figure 9.** Free memory and loop execution time of the fixed Monte Carlo benchmark.


Latency with Refreshing

Figure 11. Loop execution time of the Monte Carlo benchmark with different tick frequencies.

At last we analyze the time-space tradeoff 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 --7 the free memory over time is visualized in Figure -(. With preallocation and tick at every loop iteration4 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 iteration7 tick at every \(\frac{1}{50}\)th loop iteration and tick at every \(\frac{1}{200}\)th iteration. 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 \(\frac{1}{200}\)th loop iteration, the memory consumption is maximal, but the performance is nearly as good as the performance of the system with preallocation. At most fifty refresh calls are executed every loop iteration. Moreover, 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 \(\frac{1}{200}\)th loop iteration needs nearly the whole heap.

Figure 10. Free memory and loop execution time of the parallel Monte Carlo benchmark.

Related Work

To the best of our knowledge we are the first to propose the use of short-term memory. However, there is a resemblance between short-term memory and other memory management systems. A semi-space garbage collector [1] for example, refreshes objects by copying them from one part of the heap to another part. Time advance is implicit when the heap is full.

In [2] the authors also describe the use of buffers per allocation site with the intention to eliminate memory leaks. They use cyclic buffers whose size is determined in experiments. Self-collecting mutators determine buffer sizes dynamically depending on tick calls. Moreover, they provide refresh calls to tradeoff space consumption caused by sparse tick calls and time consumption caused by required refresh calls. The memory management system described in [2] 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 [3]. In [4] the authors propose the use of type-safe pool allocation to support program analysis.
Memory with Refreshing

Figure 12. Free memory of the Monte Carlo benchmark with different tick frequencies.
Thank you