Short-term Memory for Self-collecting Mutators

Martin Aigner, Andreas Haas, Christoph Kirsch, Ana Sokolova
Universität Salzburg

CHESS Seminar, UC Berkeley, September 2010
Heap Management

- memory leaks
- dangling pointers
- tracing
- reference-counting
- reachable memory leaks

explicit memory management deallocates here

unreachable

heap

garbage collectors deallocate here

eeded

reachable
Short-term Memory
Traditional (Persistent) Memory Model

- Allocated memory objects are guaranteed to exist until deallocation

- **Explicit** deallocation is **not safe** (dangling pointers) and can be **space-unbounded** (memory leaks)

- **Implicit** deallocation (unreachable objects) is **safe** but may be **slow** or **space-consuming** (proportional to size of live memory) and can still be **space-unbounded** (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
With short-term memory programmers or algorithms specify which memory objects are still needed and not which memory objects are not needed anymore!
Figure 2. Allocation with known expiration date.
Maximal Memory Consumption

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

© C. Kirsch 2010
Trading-off Compile-Time, Runtime, Memory

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

conservative expiration

not-expired

needed

reachable

conservative refresh

heap

© C. Kirsch 2010
Sources of Errors:

1. *not-needed* objects are continuously refreshed or *time* does not advance (memory leaks)
2. *needed* objects expire (dangling pointers)
Explicit Programming Model

• Each thread advances a thread-local clock by invoking an explicit \textit{tick()} call

• Each object receives upon its allocation an expiration date that is initialized to the thread-local time

• An explicit \textit{refresh}(Object, Extension) call sets the expiration date of the Object to the current thread-local time plus the given Extension
Explicit, Concurrent Programming Model

- Each object (logically!) receives expiration dates for all threads that are initialized to the respective thread-local times.

- Refreshing an object (logically!) sets its already expired expiration dates to the respective thread-local times.

  All threads must `tick()` before a newly allocated or refreshed object can expire!
Our Conjecture:

It is easier to say which objects are still needed than which objects are not needed anymore!
Use Cases

<table>
<thead>
<tr>
<th>benchmark</th>
<th>LoC</th>
<th>tick</th>
<th>refresh</th>
<th>free</th>
<th>aux</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg123</td>
<td>16043</td>
<td>1</td>
<td>0</td>
<td>(-)43</td>
<td>0</td>
<td>44</td>
</tr>
<tr>
<td>JLayer</td>
<td>8247</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>1450</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>LuIndex</td>
<td>74584</td>
<td>2</td>
<td>15</td>
<td>0</td>
<td>3</td>
<td>20</td>
</tr>
</tbody>
</table>

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.
Self-collecting Mutators
Goals

- **Explicit**, thread-safe memory management system
- **Constant time** complexity for all operations
  - predictable execution times, incrementality
- **Constant space** consumption by all operations
  - small, bounded space overhead
- **No additional threads** and no read/write barriers
  - self-collecting mutators!
Implementations

- **Java** patch under EPL
  - based on Jikes RVM, GNU Classpath class library
- Dynamic **C** library (libscm) under GPL
  - based on POSIX threads, ptmalloc2 allocator
- **Available** at:
  - tiptoe.cs.uni-salzburg.at/short-term-memory

works with any legacy code (1-word space overhead per memory block)
Two Approximations

- **Single-expiration-date approximation (for Java)**
  - one expiration date for all threads
  - recursive refresh is easy but blocking threads are a problem

- **Multiple-expiration-date approximation (for C)**
  - expiration dates for all threads that refreshed an object
  - recursive refresh is difficult but blocking threads can be handled
Global Time

Figure 5. Global time calculation.
Single Expiration Date

- Allocation: expiration date = global time + 1
- Refresh:
  - expiration date = global time + 1 + extension
  - unless the result is less than the old date
- Expiration: expiration date < global time
Thread-Global Time

- Threads are partitioned into active and passive
- Global time is computed over active threads

**Figure 7.** Thread-global times.
Multiple Expiration Dates

- Allocation:
  - first expiration date = \text{thread-global time} + 2

- Refresh:
  - new expiration date = \text{thread-global time} + 2 + extension

- Expiration:
  - for all threads $t$ and expiration dates $d$ of $t$: expiration date $d < \text{thread-global time of } t$
Multiple Expiration Dates

In this paper, we present two new concurrent implementations of short-term memory. For example, as stated before, the mark phase of garbage collectors may be used to implement special cases of short-term memory. For instance, the explicit programming model can be approximated by keeping multiple expiration dates for an object by a subset of threads using a notion of global time that advances upon events determined by an offline analysis tool. General refreshing is not possible but could be done at the expense of increased memory consumption since all expiration dates must be stored per-code-block regions, which allow to deallocate unreachable memory leaks for expiring reachable but actually not-freshly allocated objects.

Figure 6.

- Allocation
- Expiration date
- Ticking
- Tick of threads

The system is self-collections, there are no additional threads for implementing a special case of short-term memory where the refreshment time advances in a per-code-block region. The algorithm always uses expiration dates, which are related to the more general problem of properly refreshing multiple objects in constant time. The algorithm always uses expiration dates, which are related to the more general problem of properly refreshing multiple objects in constant time. The algorithm always uses expiration dates, which are related to the more general problem of properly refreshing multiple objects in constant time.

Recall the explicit programming model for short-term memory, called self-collections, which conserves memory and refreshing requires updating all expiration dates of an object. 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 upon events determined by an offline analysis tool. General refreshing is not possible but could be done at the expense of increased memory consumption since all expiration dates must be stored per-code-block regions, which allow to deallocate unreachable memory leaks for expiring reachable but actually not-freshly allocated objects.

The system is self-collections, there are no additional threads for implementing a special case of short-term memory where the refreshment time advances in a per-code-block region. The algorithm always uses expiration dates, which are related to the more general problem of properly refreshing multiple objects in constant time. The algorithm always uses expiration dates, which are related to the more general problem of properly refreshing multiple objects in constant time. The algorithm always uses expiration dates, which are related to the more general problem of properly refreshing multiple objects in constant time.

Constant memory consumption by all operations
Constant time complexity for all operations
No additional threads and no read-write barriers
All memory management operations are constant-time
Implementation
Java Object Model

• Jikes objects are extended by a 3-word object header:
  • 16-bit integer for expiration date
  • 2 references for doubly-linked list of objects sorted by expiration dates
  • 16-bit allocation-site identifier
• Three list operations:
  • insert, remove, select-expired
### Complexity Trade-off

<table>
<thead>
<tr>
<th>Buffer Implementation</th>
<th>Insert</th>
<th>Delete</th>
<th>Select Expired</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singly-linked list</td>
<td>O(1)</td>
<td>O(m)</td>
<td>O(m)</td>
</tr>
<tr>
<td>Doubly-linked list</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(m)</td>
</tr>
<tr>
<td>Sorted doubly-linked list</td>
<td>O(m)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Insert-pointer buffer</td>
<td>O(\log n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Segregated buffer</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(\log n)</td>
</tr>
</tbody>
</table>

**Table 2.** Comparison of buffer implementations. The number of objects in a buffer is \( m \), the maximal expiration extension is \( n \).
Segregated buffer
(with bounded expiration extension $n=3$ at time 5)

Figure 7. Segregated buffer implementation.
C Memory Block Model

- An expiration date for a given memory block is represented by a descriptor, which is a pointer to the block.
- Memory blocks are extended by a 1-word descriptor counter, which counts the descriptors pointing to a given block.
- Descriptors representing a given expiration date are gathered in a per-thread descriptor list.
Descriptor List

Figure 8. The design of the descriptor list.
Descriptor Buffer

- A **descriptor buffer** is an array of size $n+3$ of descriptor lists where $n$ is a compile-time bound on the maximal extension for refreshing.

- **Two** (constant-time) buffer operations:
  - insert, move-expired

- **Two** buffers per thread:
  - locally-clocked and globally-clocked
Memory Operations
(are all constant-time modulo the underlying allocator)

- `malloc(s)` returns a pointer to a memory block of size s plus one word for the descriptor counter, which is set to zero
- `free(Block)` frees the given `Block` if its descriptor counter is zero
- `local_refresh(Block, Extension)`
- `global_refresh(Block, Extension)`
- `tick()`
Experiments
ptmalloc2y We measure the average execution time of
vocations on latency and memory consumption.
benchmarkw the effect of using different frequencies of
collected systems and showw for the single-instance Monte Carlo
marky Moreover, for the Monte Carlo benchmark, we measure lax
a two-generation copying collector where the mature space is hanx
tators and two garbage collectors available with Jikesw the markx
ble 3y For the Java benchmarks we compare selfcollecting mux
4. Experimental Setup and Evaluation
convenient to use and does not require many code changes.
visible from the benchmarks in Table 2, selfcollecting mutators is
tators can also be achieved with static preallocation.
However, as
the periodic behavior of the benchmarky
Table 4.
The same heap size is then used for the garbagexcollected systems
necessary to execute each benchmark with selfcollecting mutators.
We execute each measurement 3fi times and calculate the average
4.1 Java Benchmarks
4.1.1 Total Execution Time and Memory Consumption
The original Monte Carlo benchmark sMC leakyt is the only
MC fixedty The execution of 4
sMC leaky benchmark can be executed in 4fiMBy We
MC fixed requires 6fiMB heap size. Both of them do not re
x
x
Table 3. System configuration.

<table>
<thead>
<tr>
<th>CPU</th>
<th>2x AMD Opteron DualCore, 2.0 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>4GB</td>
</tr>
<tr>
<td>OS</td>
<td>Linux 2.6.32-21-generic</td>
</tr>
<tr>
<td>Java VM</td>
<td>Jikes RVM 3.1.0</td>
</tr>
<tr>
<td>C compiler</td>
<td>gcc version 4.4.3</td>
</tr>
<tr>
<td>C allocator</td>
<td>ptmalloc2-20011215 (glibc-2.10.1)</td>
</tr>
</tbody>
</table>

© C. Kirsch 2010
Java: Memory

<table>
<thead>
<tr>
<th></th>
<th>MC leaky</th>
<th>MC fixed</th>
<th>4×MC fixed</th>
<th>JLayer</th>
<th>LuIndex</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCM(1,1)</td>
<td>40MB</td>
<td>40MB</td>
<td>60MB</td>
<td>95MB</td>
<td>370MB</td>
</tr>
<tr>
<td>SCM (50,20)</td>
<td>50MB</td>
<td>40MB</td>
<td>70MB</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>aggressive SCM(1,1)</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>90MB</td>
<td>250MB</td>
</tr>
<tr>
<td>GEN</td>
<td>95MB</td>
<td>40MB</td>
<td>70MB</td>
<td>95MB</td>
<td>370MB</td>
</tr>
<tr>
<td>MS</td>
<td>100MB</td>
<td>40MB</td>
<td>70MB</td>
<td>95MB</td>
<td>370MB</td>
</tr>
</tbody>
</table>

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.
Java: Throughput

![Graph showing performance comparison of different systems and benchmarks.](image)

**Figure 9.** Total execution time of the Monte Carlo benchmarks in percentage of the total execution time of the benchmark using self-collecting mutators.
Java: Throughput

Figure 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.
Figure 11. Free memory and loop execution time of the fixed Monte Carlo benchmark.
Java: Latency w/ Refreshing

Figure 13. Loop execution time of the Monte Carlo benchmark with different tick frequencies. Self-collecting mutators is used.
Java: Memory w/ Refreshing

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>persistent MM</th>
<th>short-term MM</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc of ptmalloc2</td>
<td>166 (78 / 199k)</td>
<td>/</td>
</tr>
<tr>
<td>free of ptmalloc2</td>
<td>86 (14 / 169k)</td>
<td>/</td>
</tr>
<tr>
<td>malloc of SCM</td>
<td>172 (82 / 267k)</td>
<td>138 (75 / 271k)</td>
</tr>
<tr>
<td>free of SCM</td>
<td>91 (10 / 157k)</td>
<td>/</td>
</tr>
<tr>
<td>local-refresh(1, 256B)</td>
<td>/</td>
<td>227 (131 / 548k)</td>
</tr>
<tr>
<td>local-refresh(10, 256B)</td>
<td>/</td>
<td>225 (131 / 548k)</td>
</tr>
<tr>
<td>local-refresh(1, 4KB)</td>
<td>/</td>
<td>228 (131 / 548k)</td>
</tr>
<tr>
<td>local-refresh(10, 4KB)</td>
<td>/</td>
<td>230 (131 / 548k)</td>
</tr>
<tr>
<td>global-refresh(1, 256B)</td>
<td>/</td>
<td>226 (116 / 551k)</td>
</tr>
<tr>
<td>global-refresh(10, 256B)</td>
<td>/</td>
<td>224 (116 / 551k)</td>
</tr>
<tr>
<td>global-refresh(1, 4KB)</td>
<td>/</td>
<td>227 (116 / 551k)</td>
</tr>
<tr>
<td>global-refresh(10, 4KB)</td>
<td>/</td>
<td>228 (116 / 551k)</td>
</tr>
<tr>
<td>local-tick(1, 256B)</td>
<td>/</td>
<td>378 (277 / 164k)</td>
</tr>
<tr>
<td>local-tick(10, 256B)</td>
<td>/</td>
<td>359 (277 / 71k)</td>
</tr>
<tr>
<td>local-tick(1, 4KB)</td>
<td>/</td>
<td>375 (277 / 164k)</td>
</tr>
<tr>
<td>local-tick(10, 4KB)</td>
<td>/</td>
<td>366 (277 / 164k)</td>
</tr>
<tr>
<td>global-tick(1, 256B)</td>
<td>/</td>
<td>367 (229 / 169k)</td>
</tr>
<tr>
<td>global-tick(10, 256B)</td>
<td>/</td>
<td>352 (229 / 151k)</td>
</tr>
<tr>
<td>global-tick(1, 4KB)</td>
<td>/</td>
<td>365 (229 / 169k)</td>
</tr>
<tr>
<td>global-tick(10, 4KB)</td>
<td>/</td>
<td>361 (229 / 169k)</td>
</tr>
</tbody>
</table>
Self-collecting mutators is also competitive to explicit deallocation of memory in terms of total execution time, as measured in the mpg678 benchmark. We then compare self-collecting mutators with different tick frequencies.

Table 6 shows the execution times in CPU clock cycles of the self-collecting mutators with a maximal expiration extension of 0.5 on different loop iteration counts. The memory overhead and consumption of descriptors and descriptor pages varies with the tick frequency and the use of SCM.

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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Loop Execution Time (ms)</th>
<th>Throughput (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptmalloc2</td>
<td>895.25</td>
<td>100.00%</td>
</tr>
<tr>
<td>ptmalloc2 through SCM</td>
<td>899.43</td>
<td>100.47%</td>
</tr>
<tr>
<td>local-SCM(1, 256B)</td>
<td>890.18</td>
<td>99.43%</td>
</tr>
<tr>
<td>local-SCM(10, 256B)</td>
<td>898.28</td>
<td>100.34%</td>
</tr>
<tr>
<td>local-SCM(1, 4KB)</td>
<td>892.18</td>
<td>99.66%</td>
</tr>
<tr>
<td>local-SCM(10, 4KB)</td>
<td>892.28</td>
<td>99.67%</td>
</tr>
<tr>
<td>global-SCM(1, 256B)</td>
<td>893.76</td>
<td>99.83%</td>
</tr>
</tbody>
</table>
C: Memory

Figure 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.
Thank you

Check out:
eurosyst2011.cs.uni-salzburg.at