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

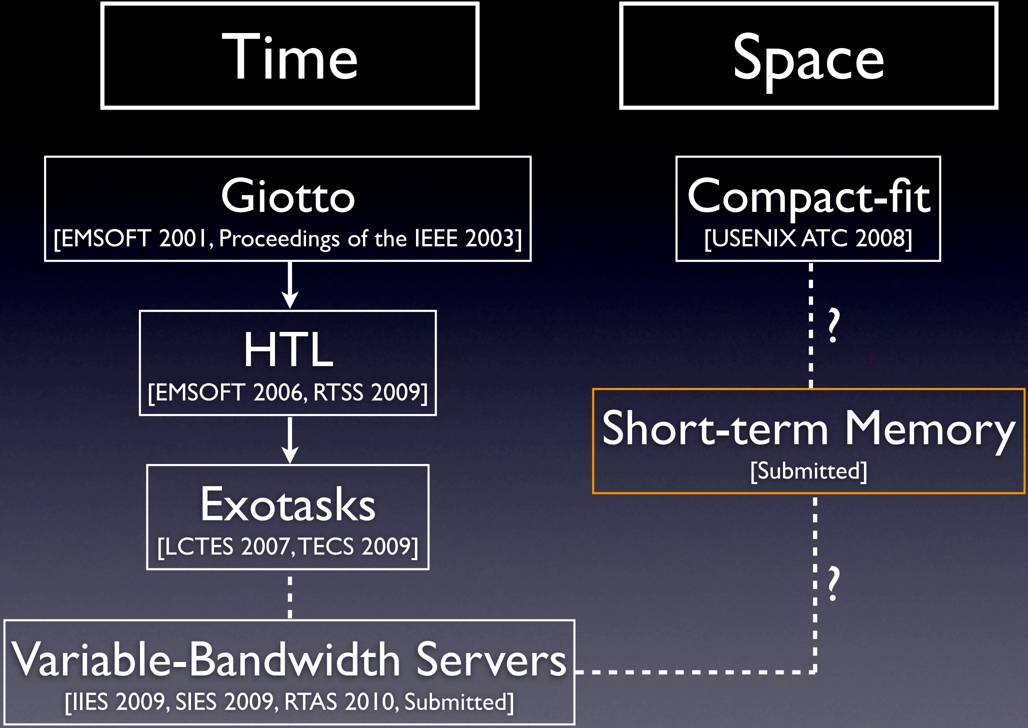
Andreas Haas, Christoph Kirsch, Hannes Payer, Andreas Schoenegger, Ana Sokolova

RiSE Workshop, March 2010
TU Graz, Austria

Time-predictable virtualization: process <u>response times</u> and <u>jitter</u> 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


Time
Space
Giotto
[EMSOFT 2001, Proceedings of the IEEE 2003]
Compact-fit
[USENIX ATC 2008]
?
HTL
[EMSOFT 2006, RTSS 2009]
Short-term Memory
[Submitted]
Exotasks
[LCTES 2007, TECS 2009]
?
Variable-Bandwidth Servers
[IIES 2009, SIES 2009, RTAS 2010, Submitted]

Short-term Memory

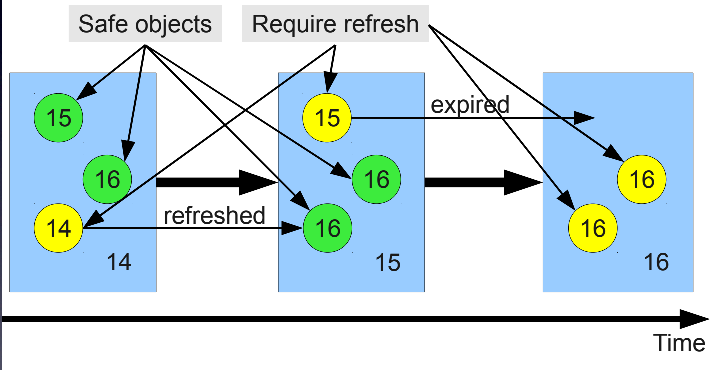
Traditional Memory Model

- 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



With short-term memory programmers specify which memory objects are still needed and <u>not</u> which memory objects are not needed anymore!

Full Compile-Time Knowledge

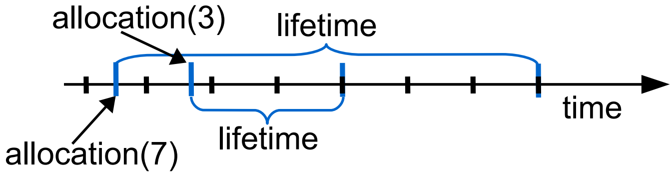


Figure 1. Allocation with known expiration date.

Maximal Memory Consumption

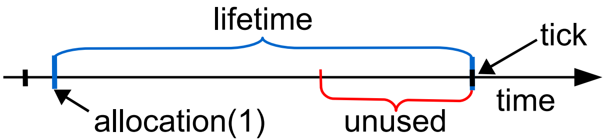


Figure 2. All objects are allocated for one time unit.

Trading-off Compile-Time, Runtime, Memory

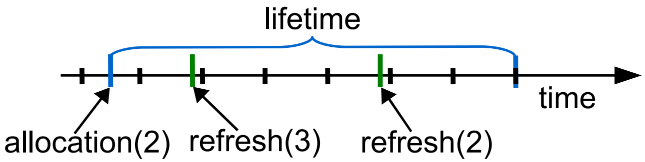


Figure 3. Allocation with estimated expiration date. If the object is needed longer, it is refreshed.

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*)

Allocation

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

	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	$O(1)$	$O(1)$	$O(\log n)$

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*)

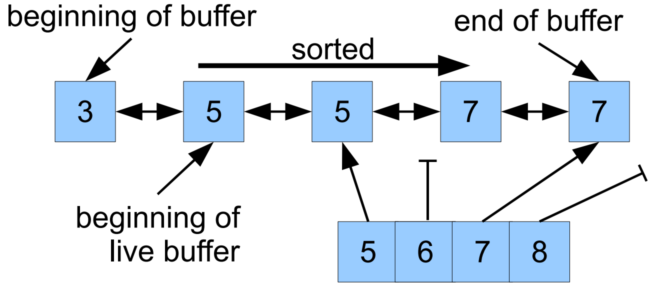


Figure 6. Insert-pointer buffer implementation.

Segregated buffer

(with bounded expiration extension *n*=3 and unsorted select-expired)

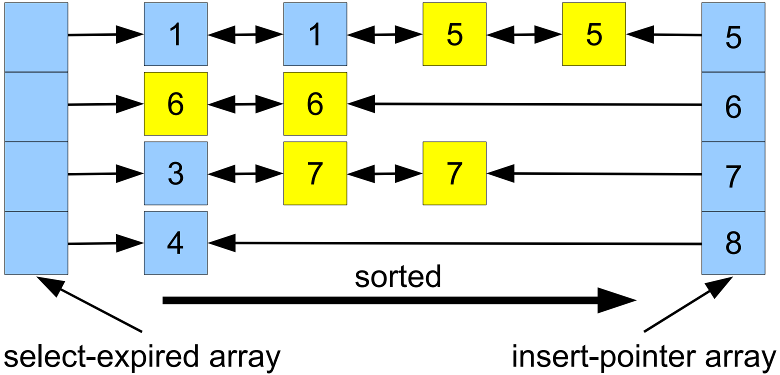


Figure 7. Segregated buffer implementation.

Experiments

Setup

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 3. System configuration.

Benchmarks

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

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

Runtime Performance

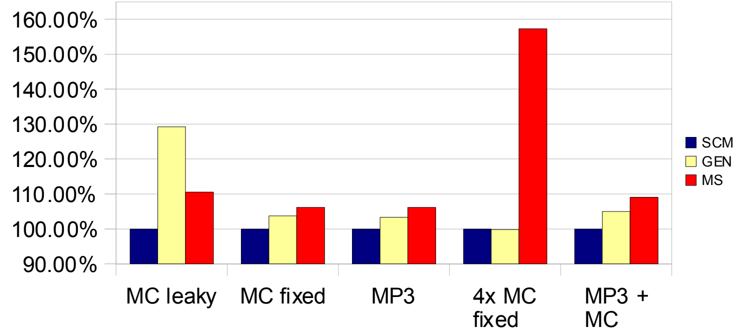


Figure 8. Total runtime of the benchmarks in percent of the runtime of the benchmark using self-collecting mutators.

Latency & Memory

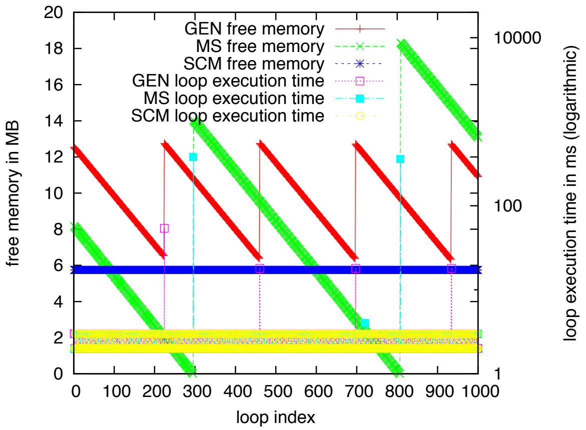


Figure 9. Free memory and loop execution time of the fixed Monte Carlo benchmark.

Latency with Refreshing

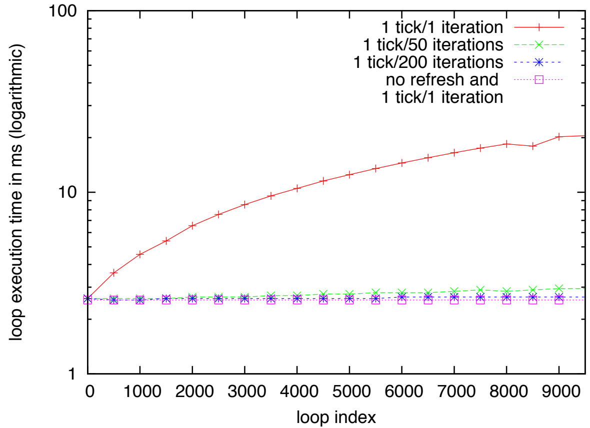


Figure 11. Loop execution time of the Monte Carlo benchmark with different tick frequencies.

Memory with Refreshing

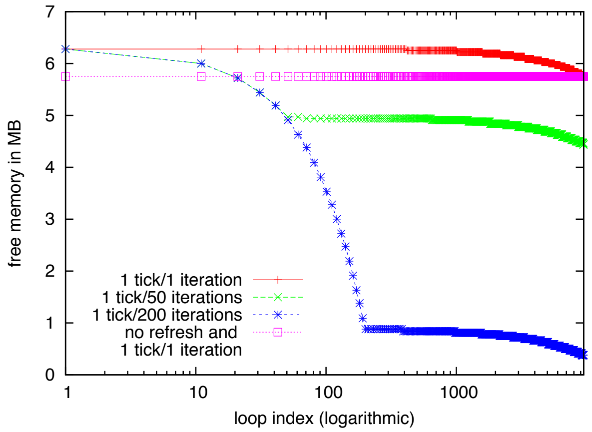


Figure 12. Free memory of the Monte Carlo benchmark with different tick frequencies.


Thank you