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

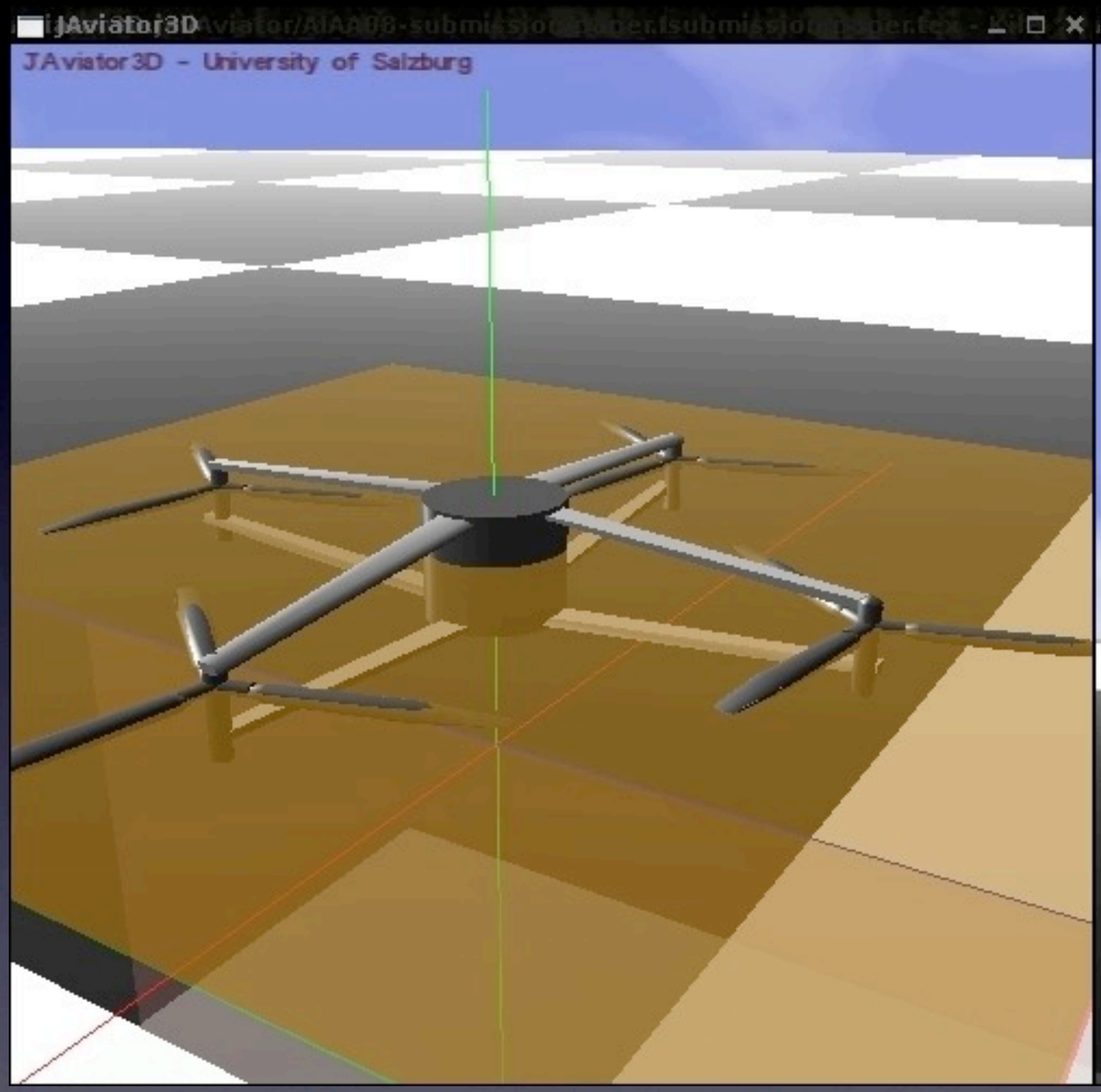
Martin Aigner, Andreas Haas, Christoph Kirsch,
Hannes Payer, Andreas Schoenegger, Ana Sokolova
Universität Salzburg

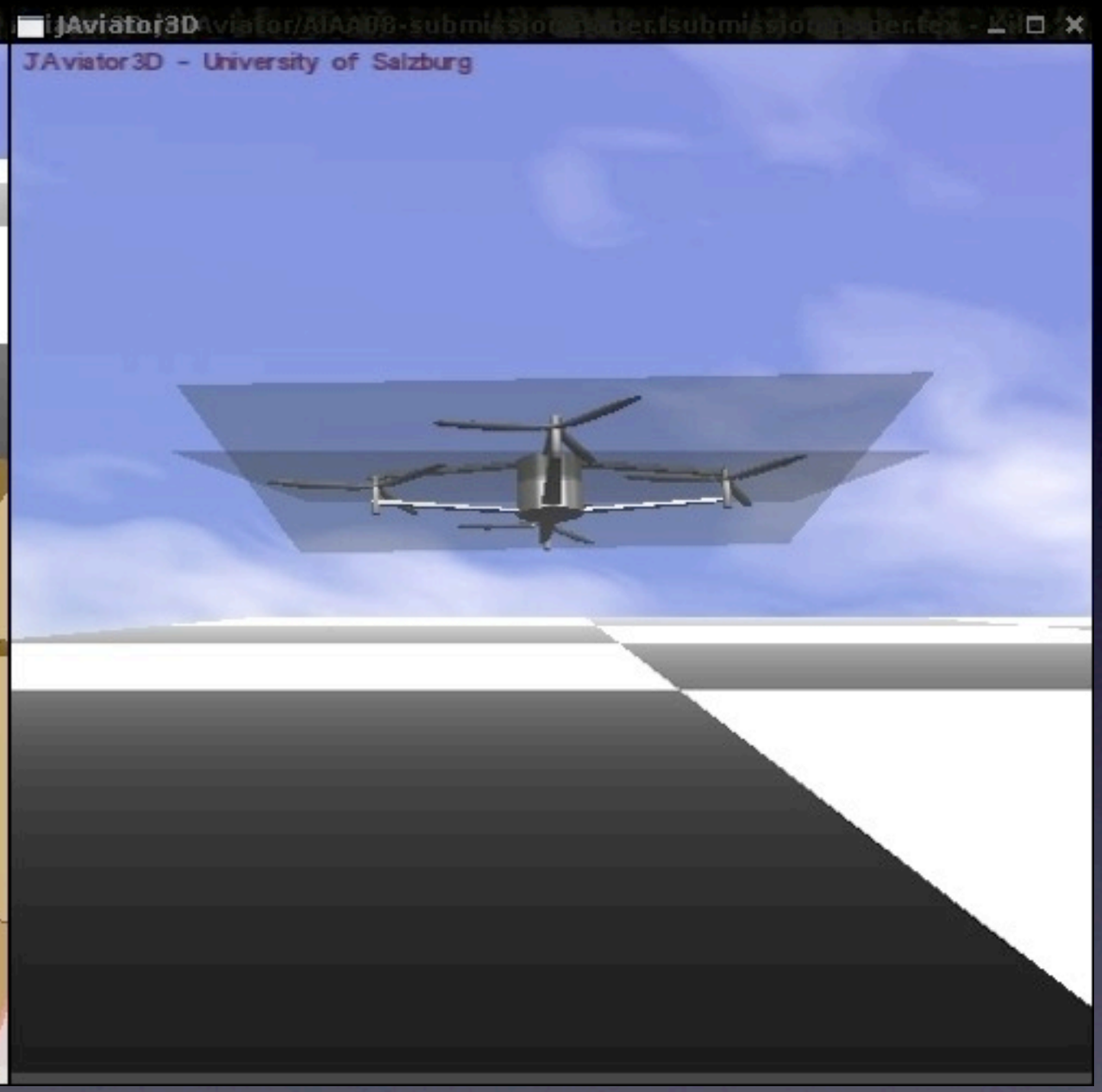
Computer Science Symposium, May 2010
IST Austria

The JAviator

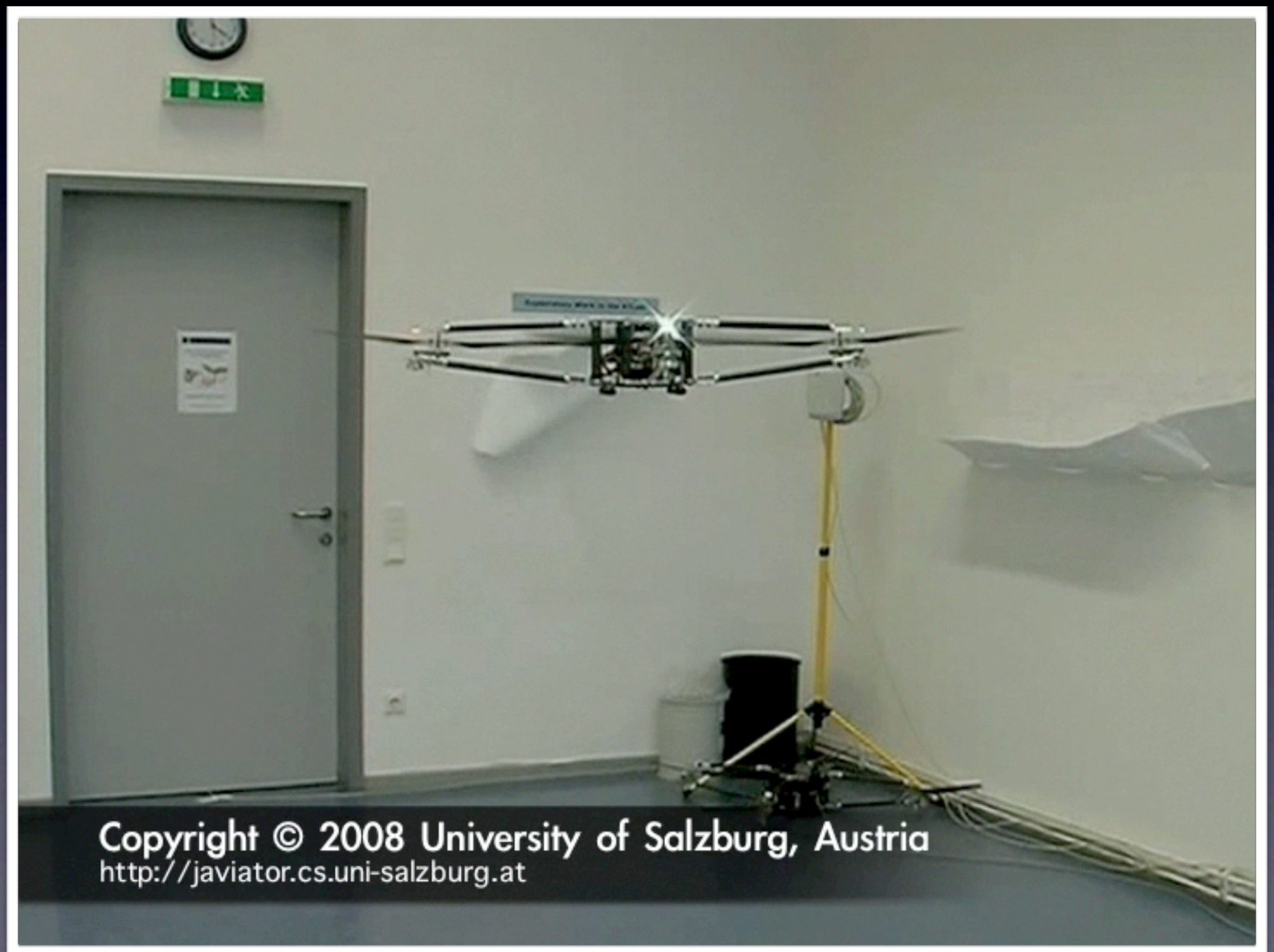
javiator.cs.uni-salzburg.at

Quad-Rotor Helicopter





Indoor Flight
STARMAC Controller



Outdoor Flight Salzburg Controller

Short-term Memory

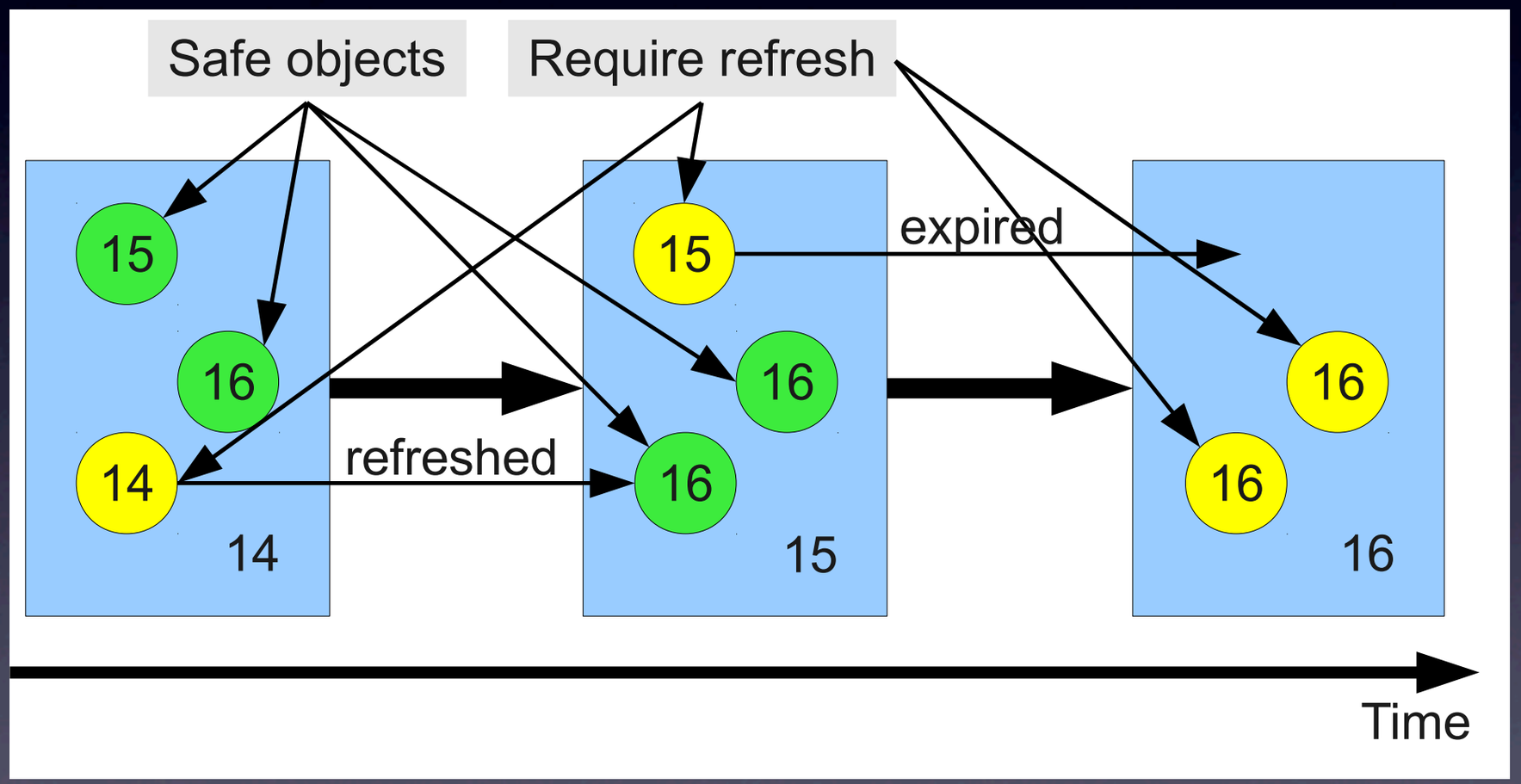
Traditional Memory Model

- 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

Example



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

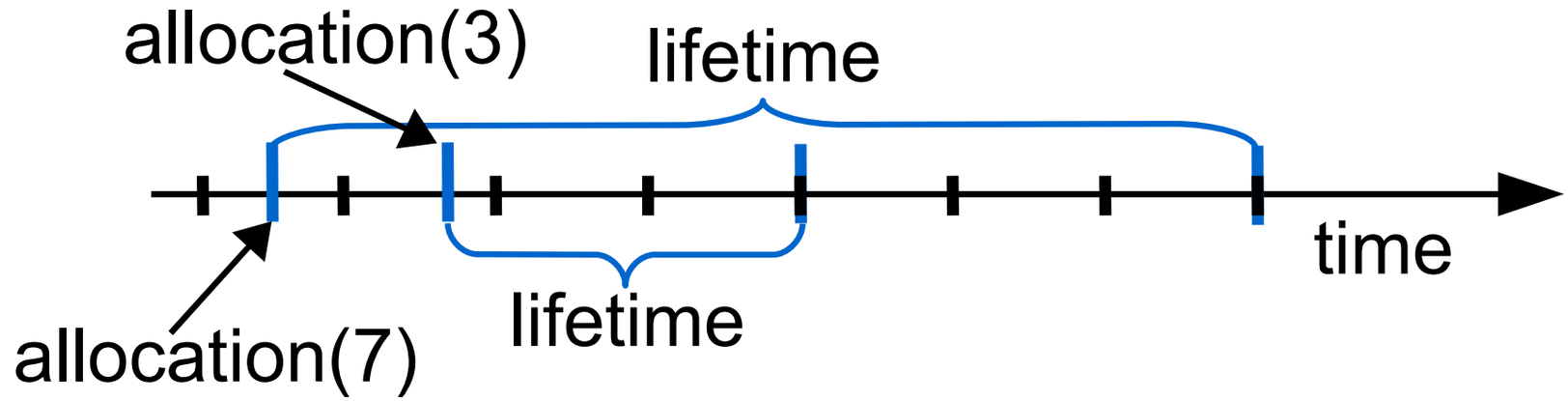


Figure 1. Allocation with known expiration date.

Maximal Memory Consumption

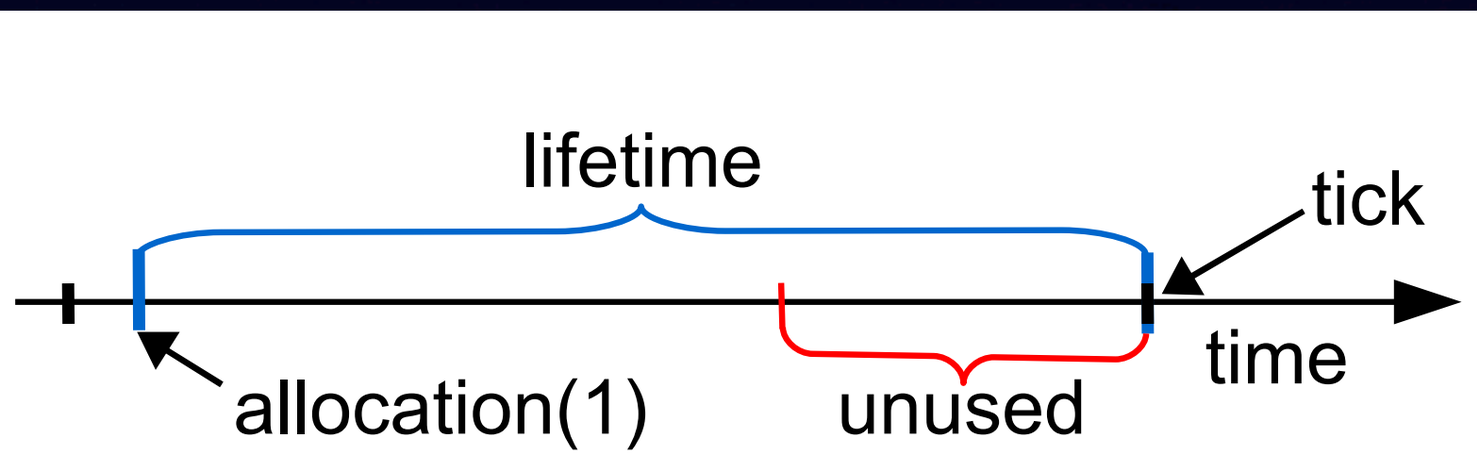


Figure 2. All objects are allocated for one time unit.

Trading-off Compile-Time, Runtime, Memory

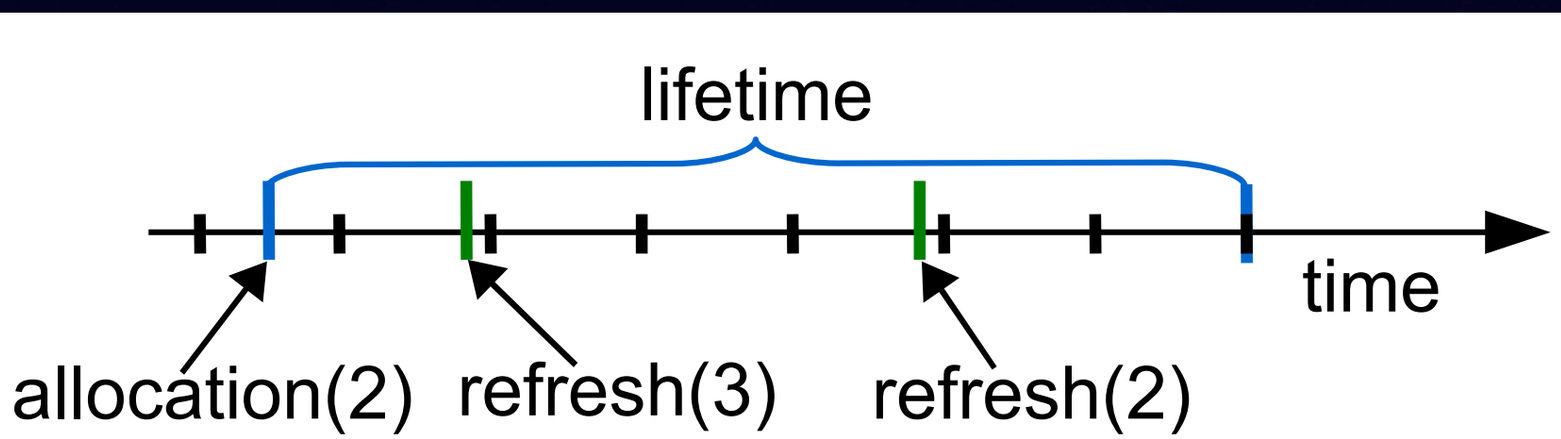


Figure 3. Allocation with estimated expiration date. If the object is needed longer, it is refreshed.

Our Conjecture:

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

Benchmark

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.

Self-collecting Mutators

Goals

- Competitive performance to GC systems
- Constant-time operations
  - Predictable execution times
- No additional threads
  - No read/write barriers

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
- 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

Single-threaded 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* at time 5)

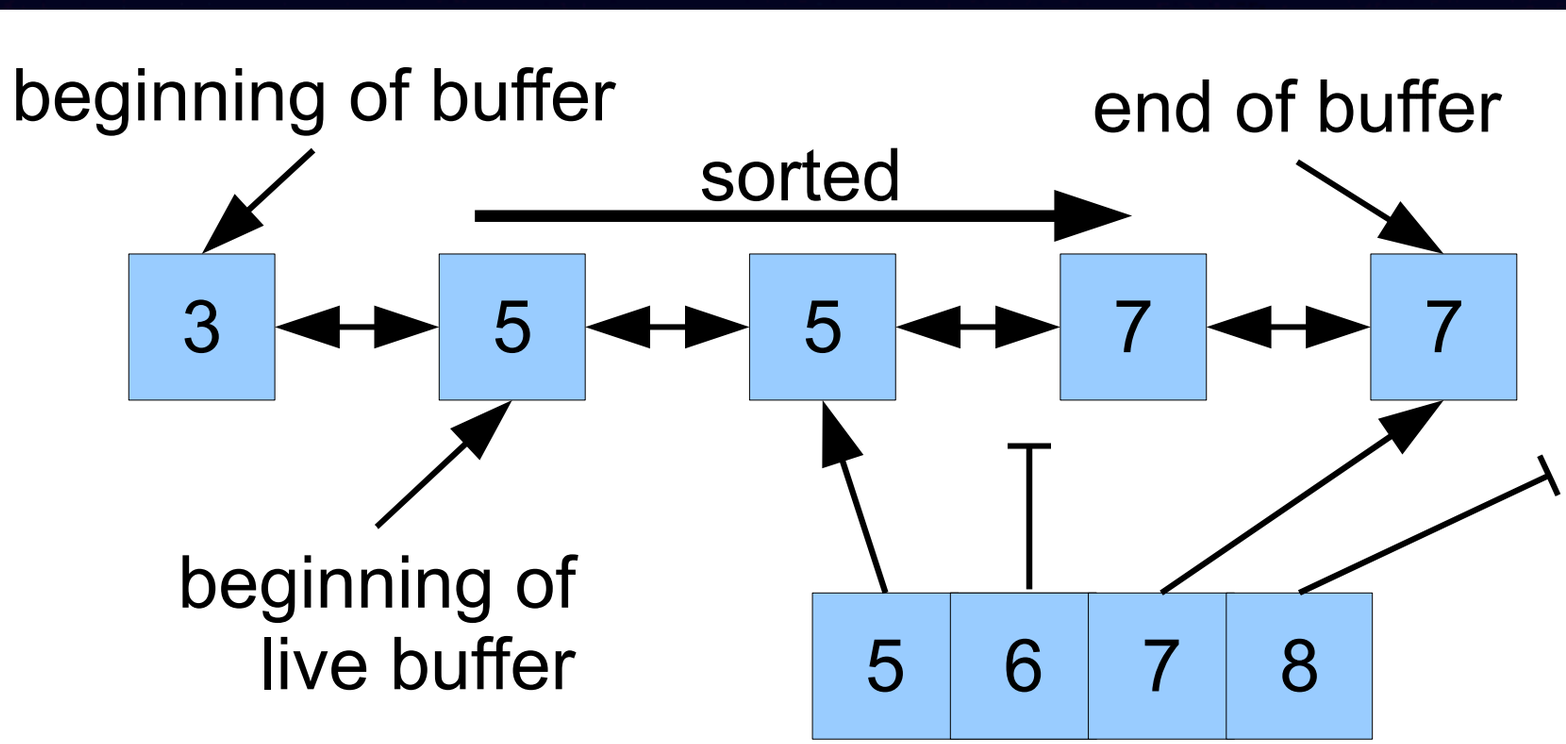


Figure 6. Insert-pointer buffer implementation.

Segregated buffer
(with bounded expiration extension *n*=3 at time 5)

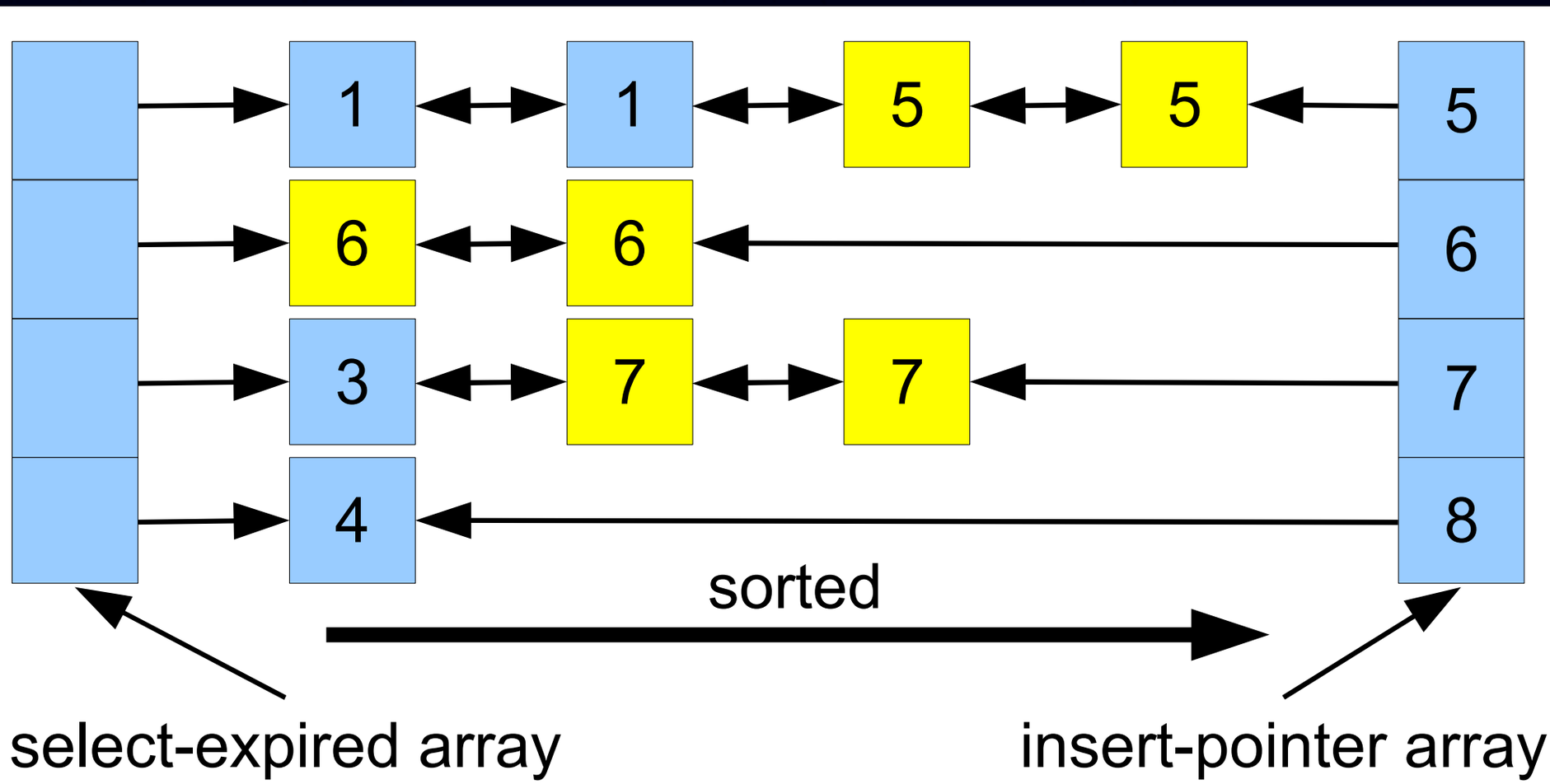


Figure 7. Segregated buffer implementation.

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.

Runtime Performance

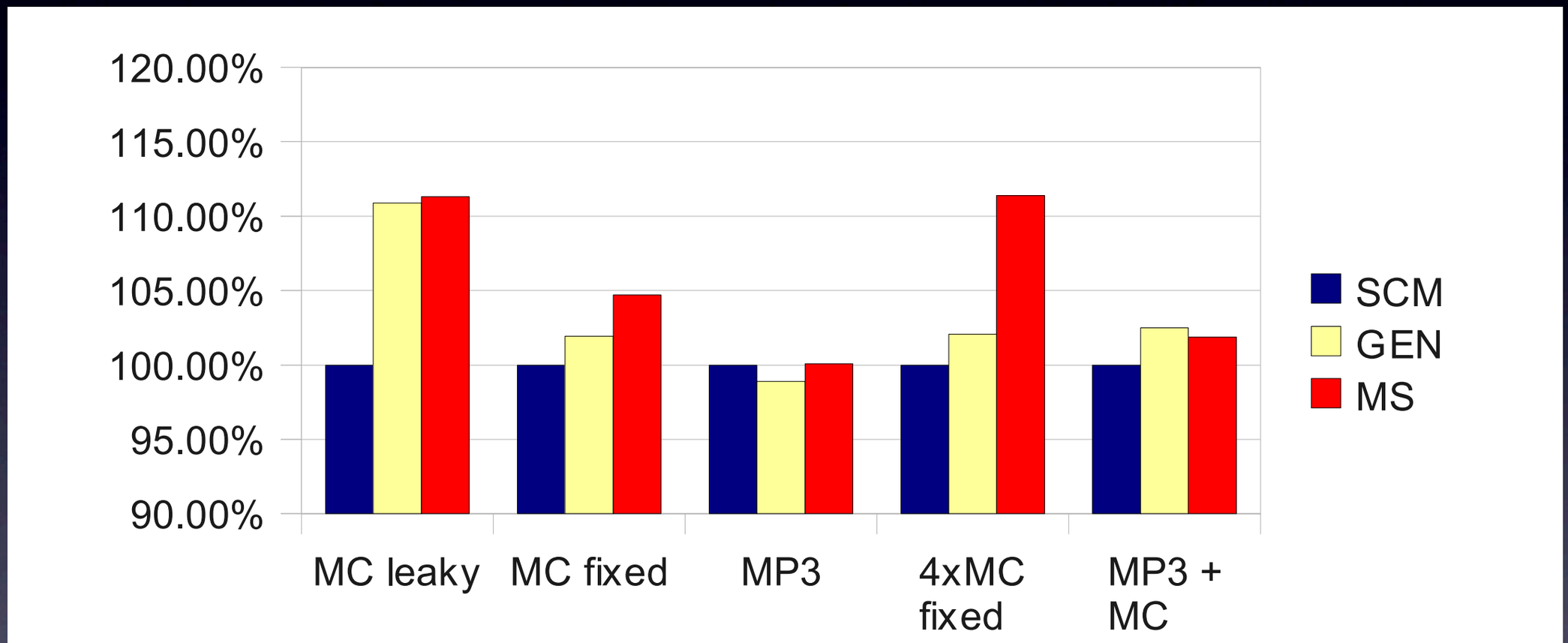


Fig. 8. Total runtime of the benchmarks in percent of the runtime of the benchmark using self-collecting mutators. The production configuration of Jikes is used.

Latency & Memory

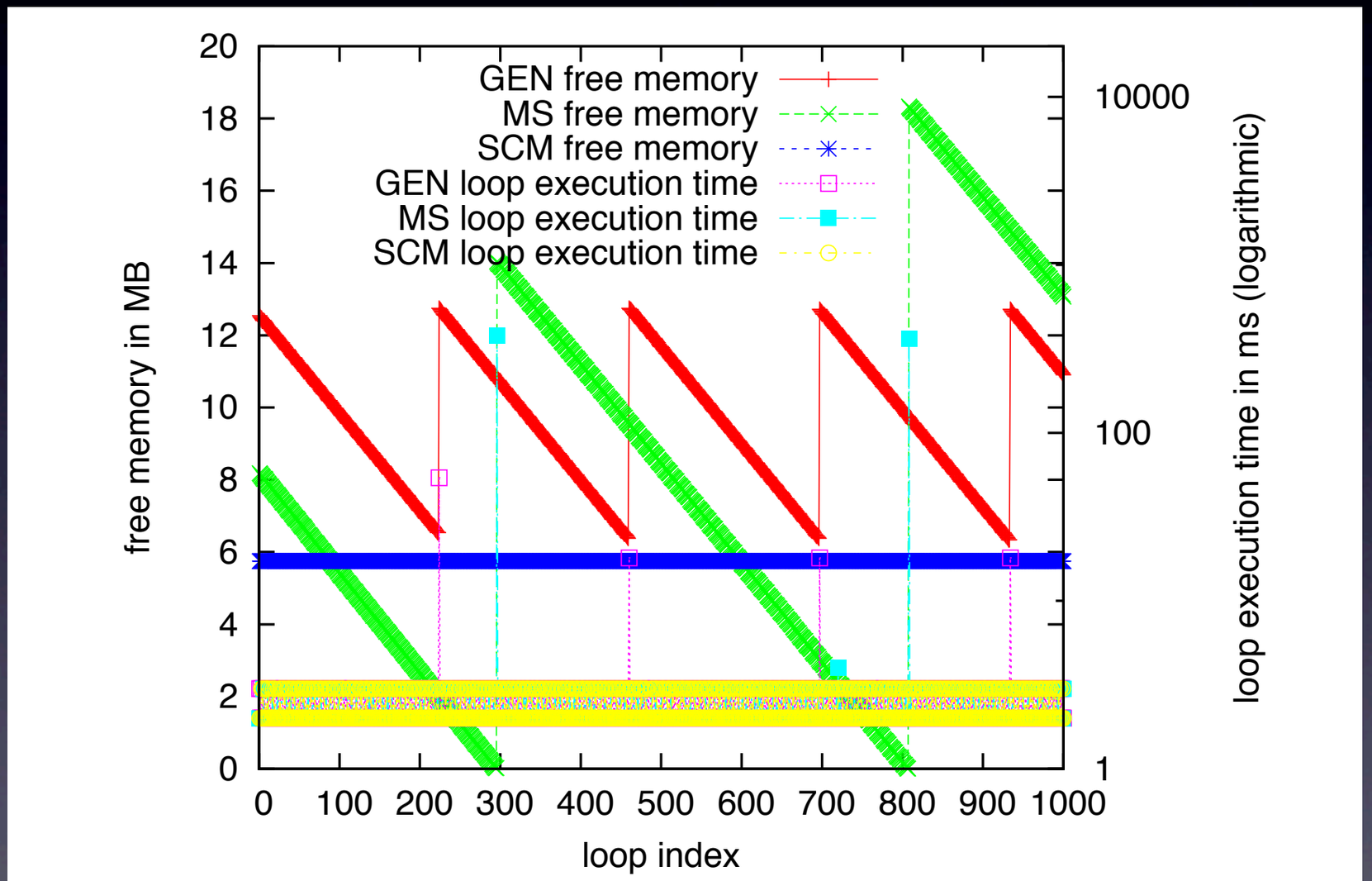


Figure 9. Free memory and loop execution time of the fixed Monte Carlo benchmark.

Latency with Refreshing

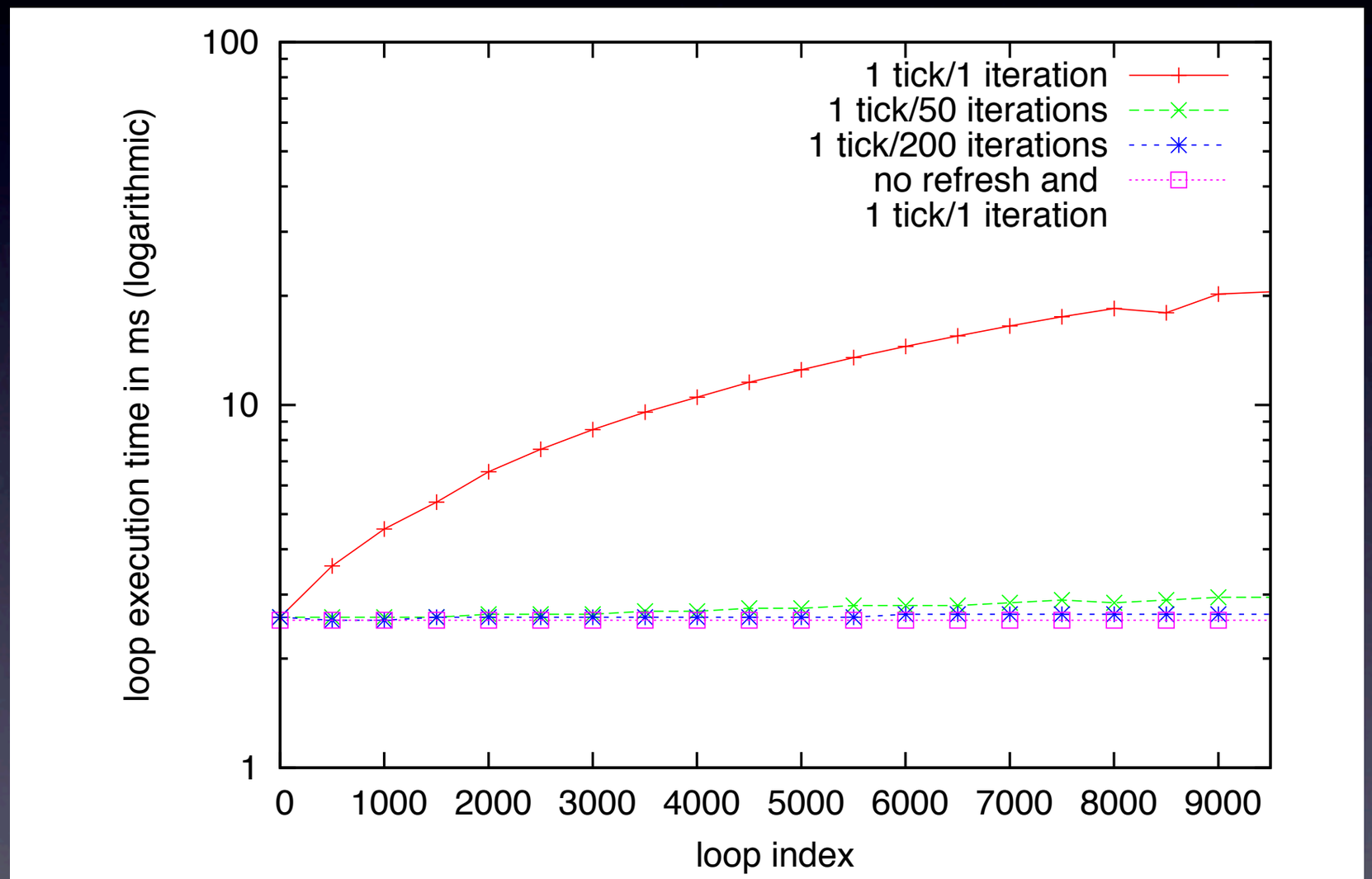


Figure 11. Loop execution time of the Monte Carlo benchmark with different tick frequencies.

Memory with Refreshing

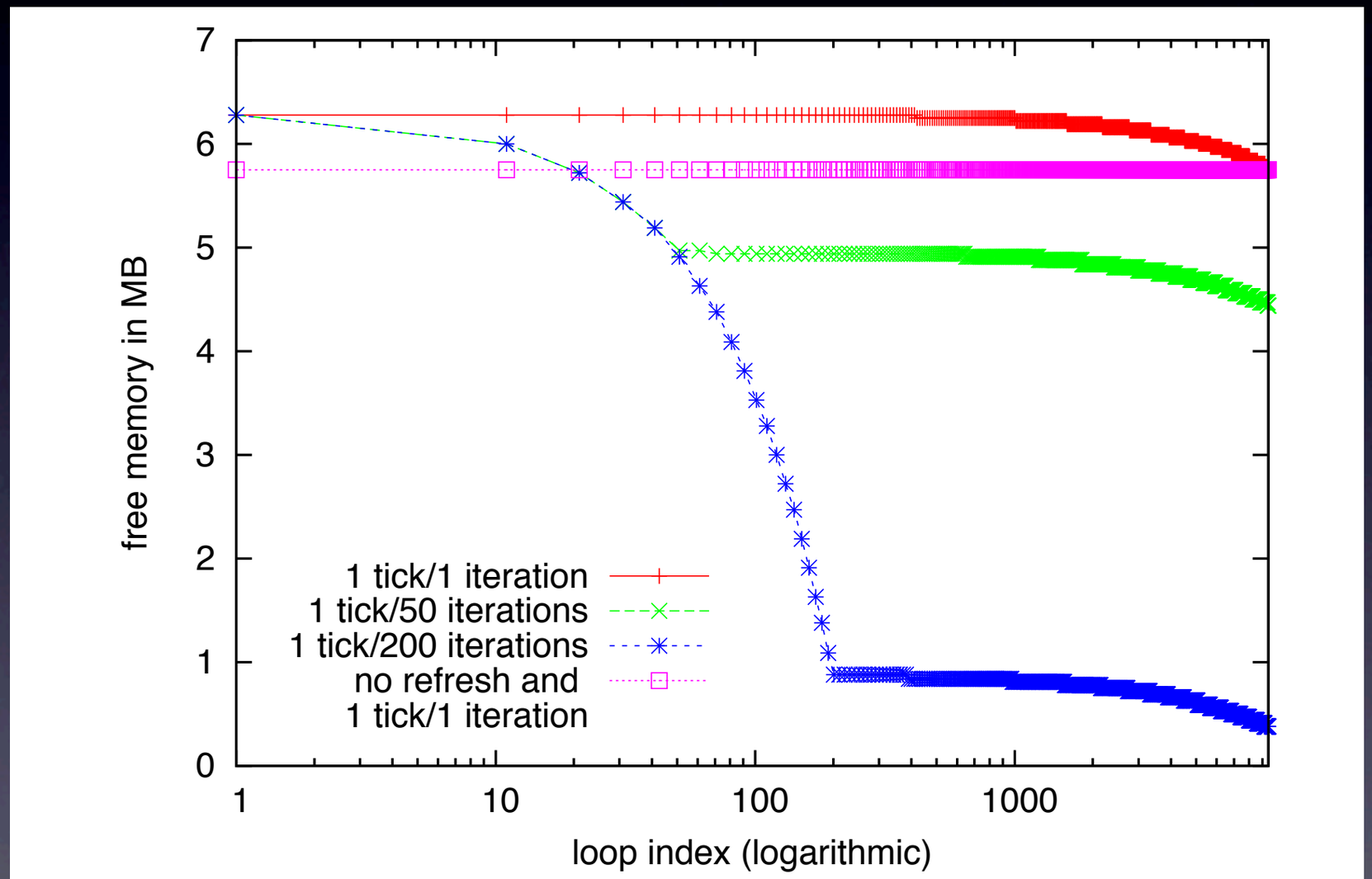


Figure 12. Free memory of the Monte Carlo benchmark with different tick frequencies.

Thank you