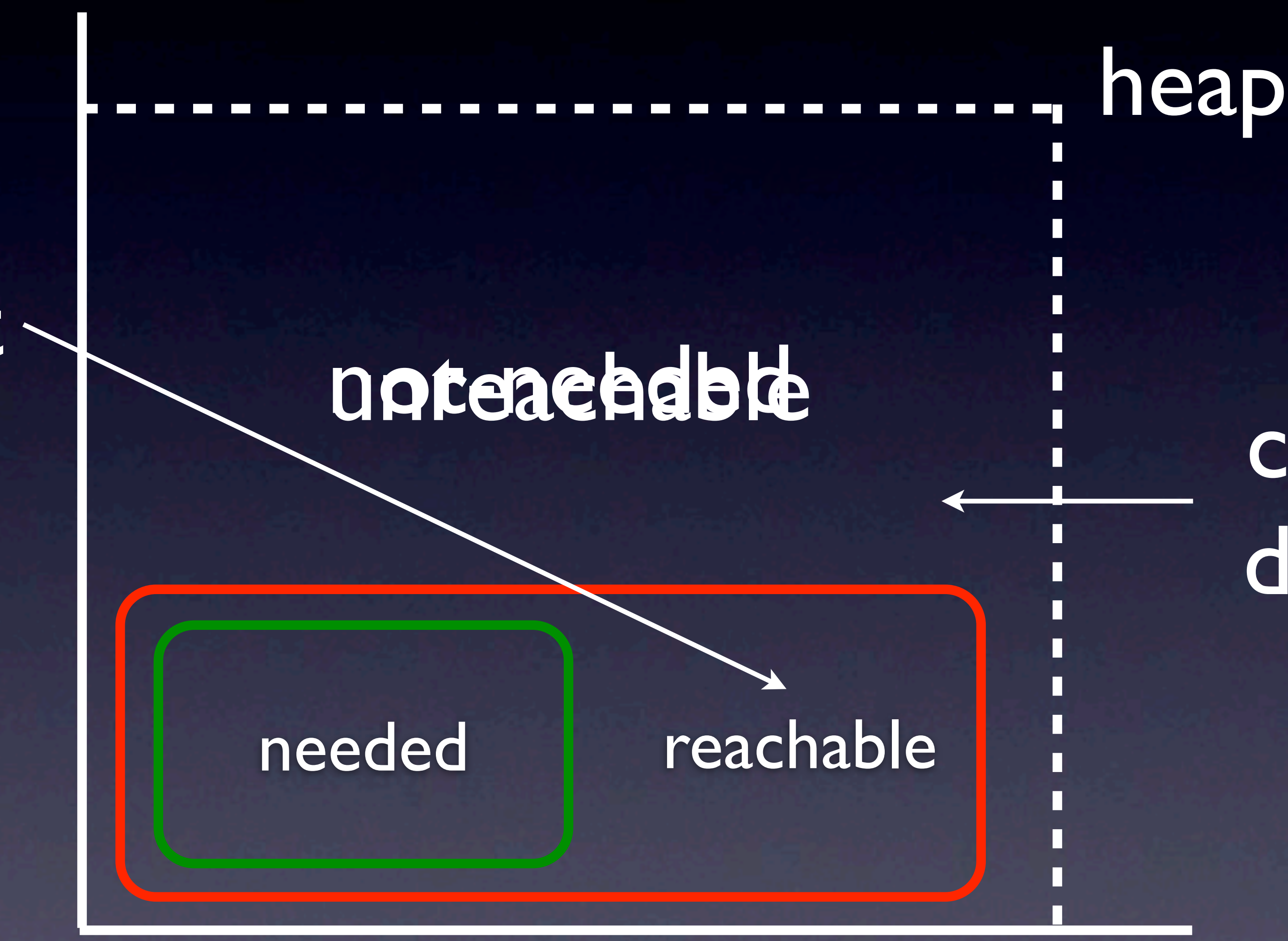
Short-term Memory for Self-collecting Mutators

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

CHESS Seminar, UC Berkeley, September 2010

Heap Management

explicit memory management deallocates here



garbage collectors deallocate here

- memory leaks
- dangling pointers

- tracing
- reference-counting
  - reachable memory leaks

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!

Full Compile-Time Knowledge

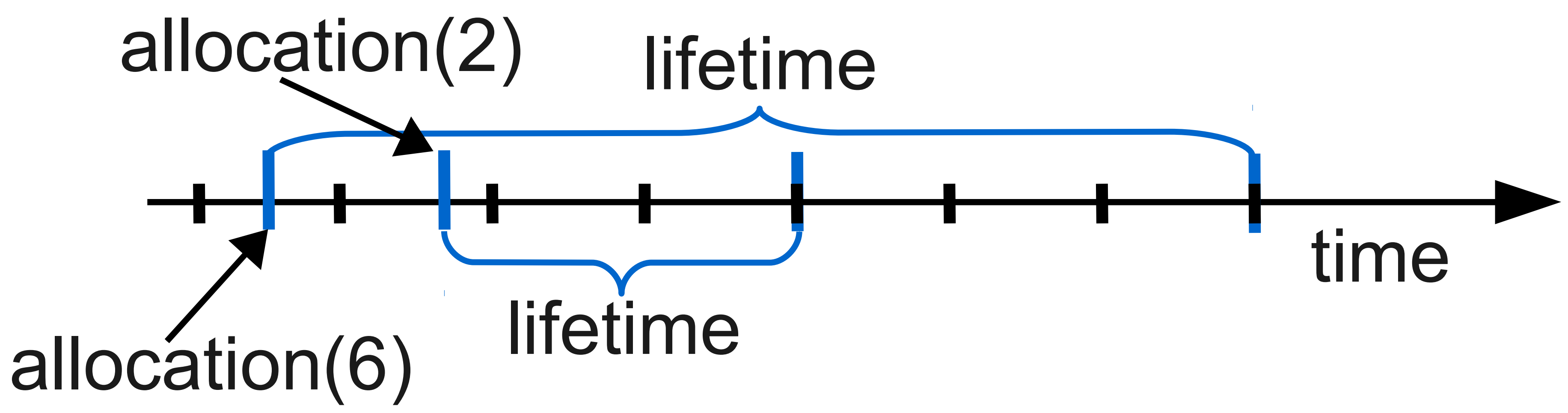


Figure 2. Allocation with known expiration date.

Maximal Memory Consumption

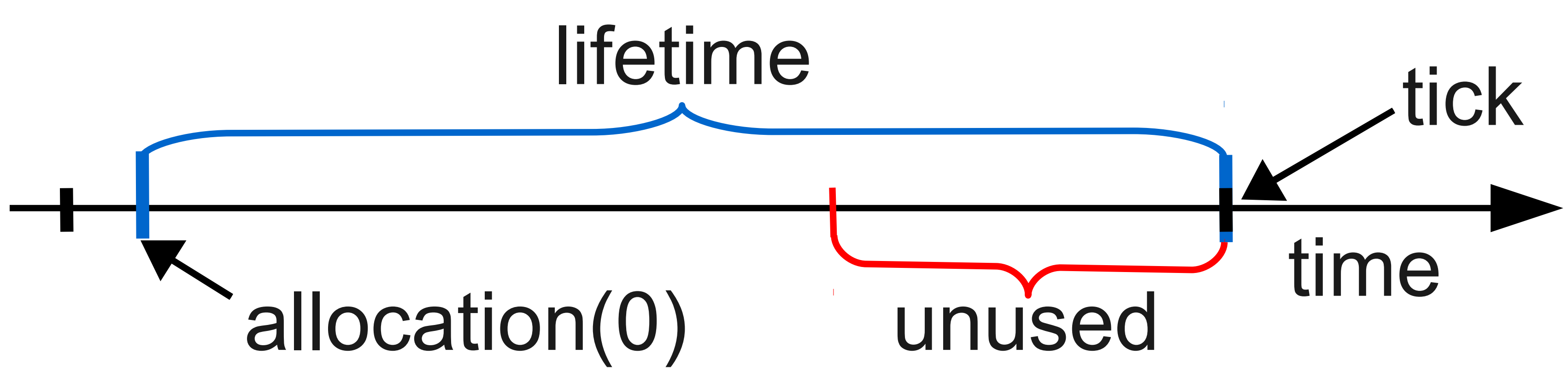


Figure 3. All objects are allocated for one time unit.

Trading-off Compile-Time, Runtime, Memory

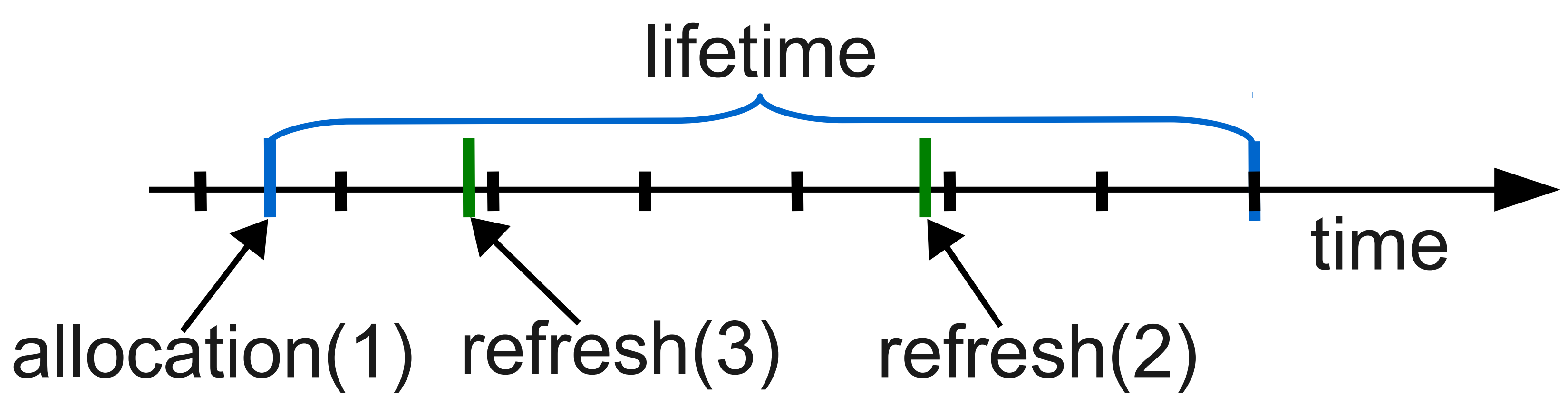
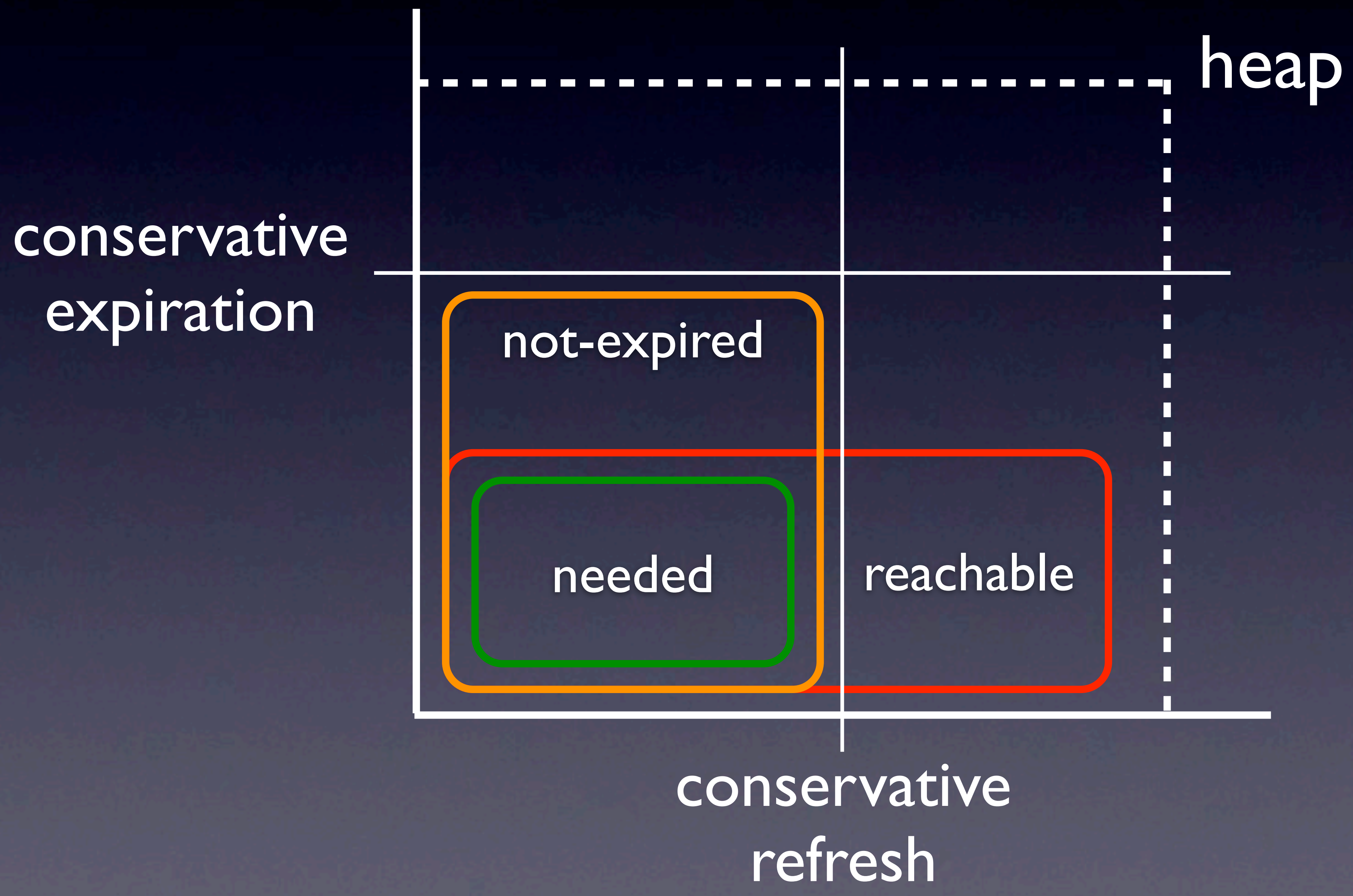


Figure 4. Allocation with estimated expiration date. If the object is needed longer, it is refreshed.

Heap Management



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 `tick()` call
- Each object receives upon its allocation an expiration date that is initialized to the thread-local time
- An explicit `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

benchmark	LoC	tick	refresh	free	aux	total
mpg123	16043	1	0	(-)43	0	44
JLayer	8247	1	6	0	2	9
Monte Carlo	1450	1	3	0	2	6
LuIndex	74584	2	15	0	3	20

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!

Implement

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

- Java patch under EPL
  - based on Jikes RVM, GN 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

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

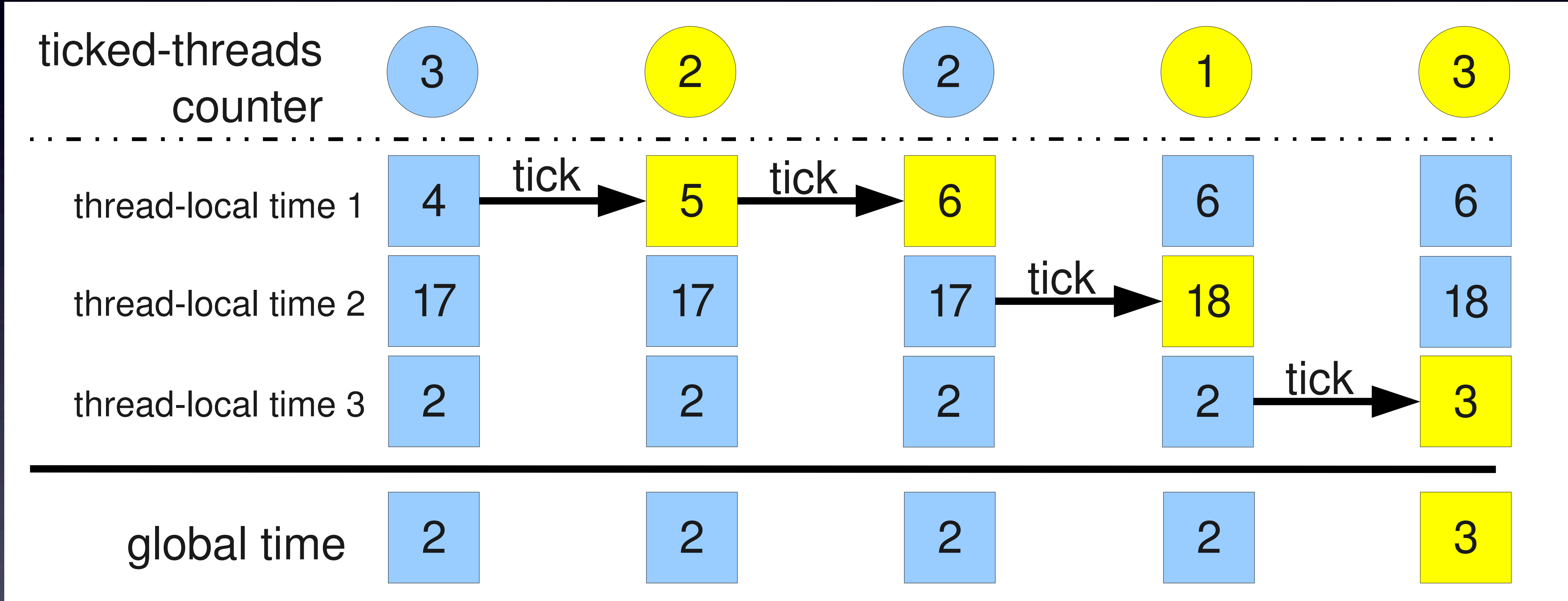
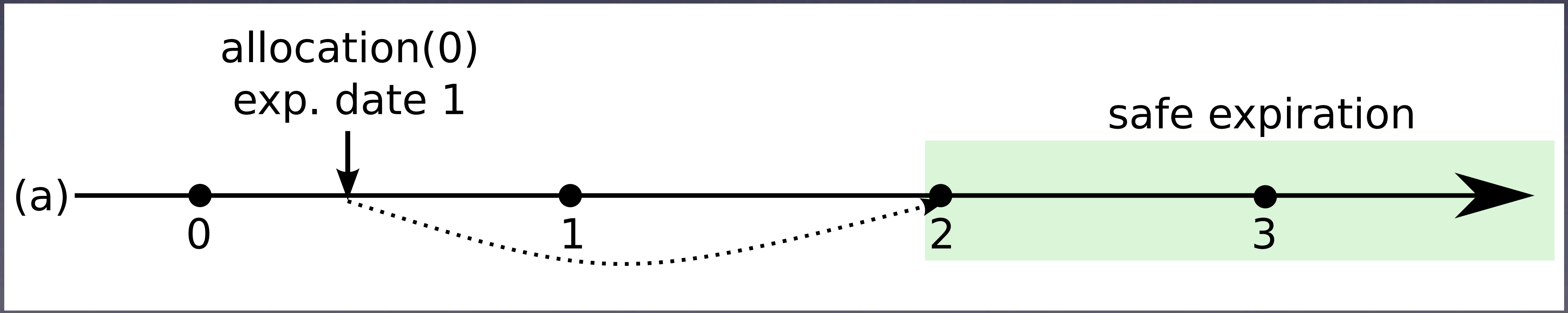


Figure 5. Global time calculation.

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

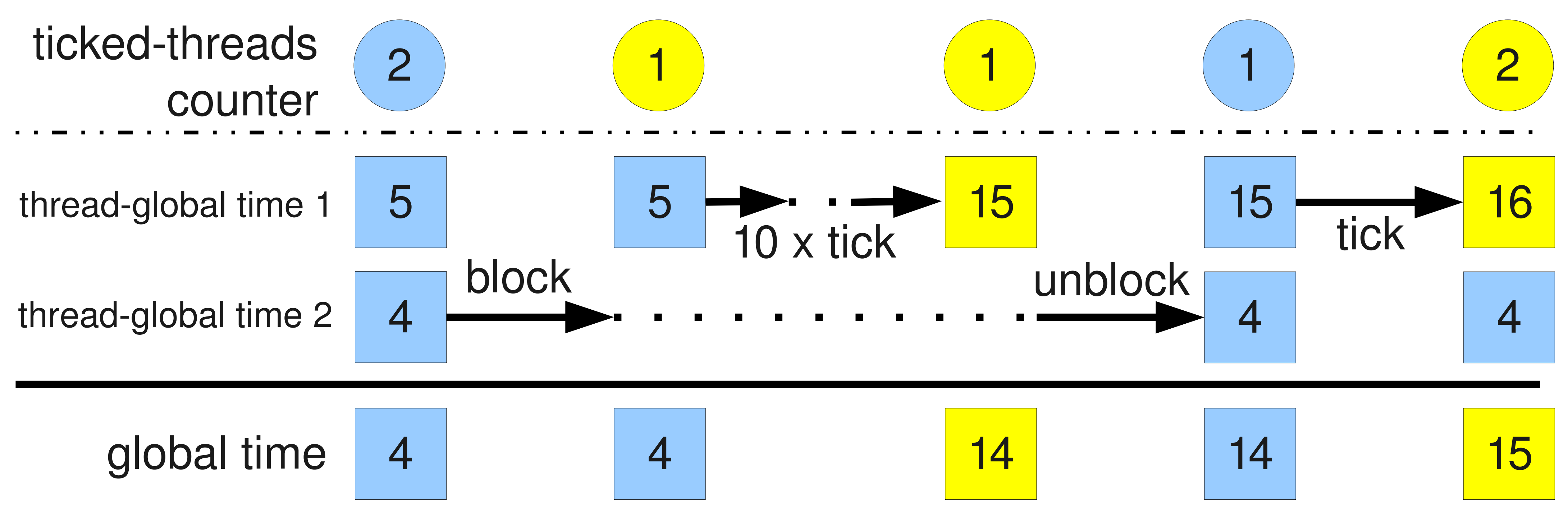
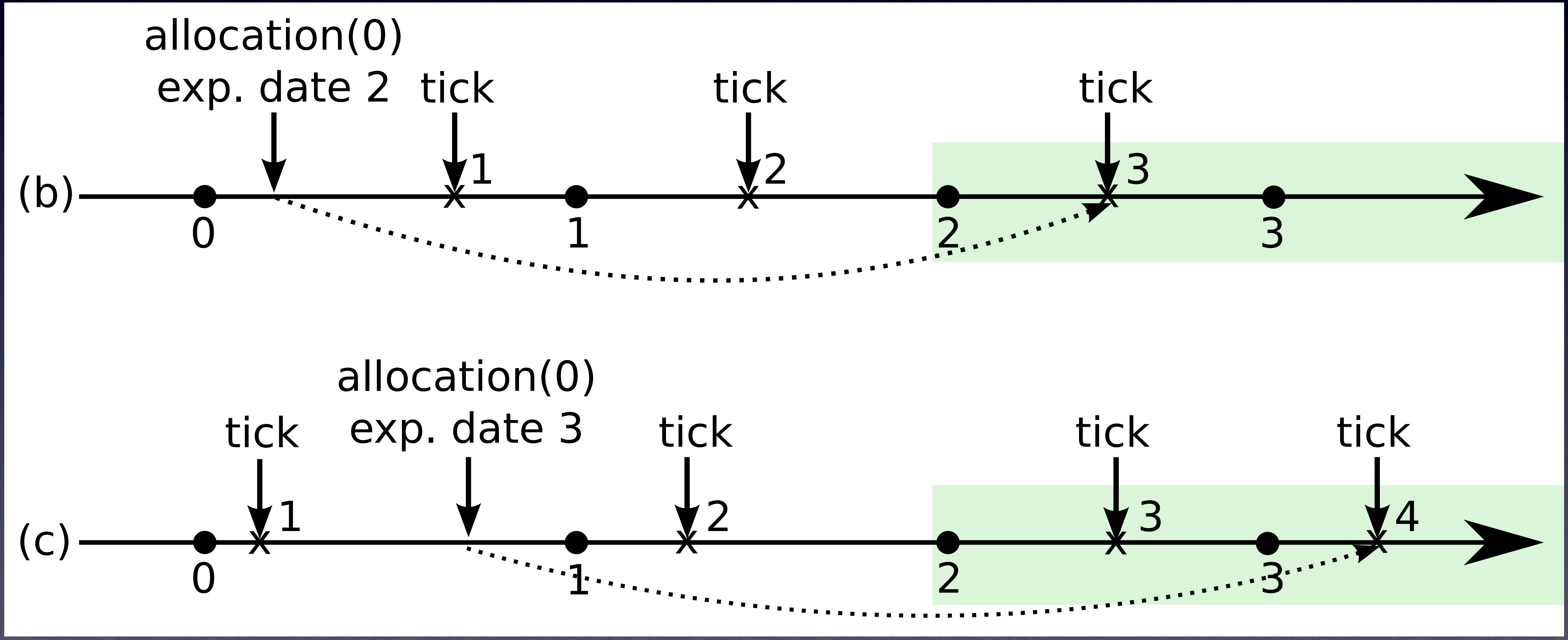


Figure 7. Thread-global times.

Multiple Expiration Dates

- Allocation:
  - first expiration date = thread-global time + 2
- Refresh:
  - new expiration date = thread-global time + 2 + extension
- Expiration:
  - for all threads *t* and expiration dates *d* of *t*: expiration date *d* < thread-global time of *t*

Multiple Expiration Dates



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

	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.

Segregated buffer

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

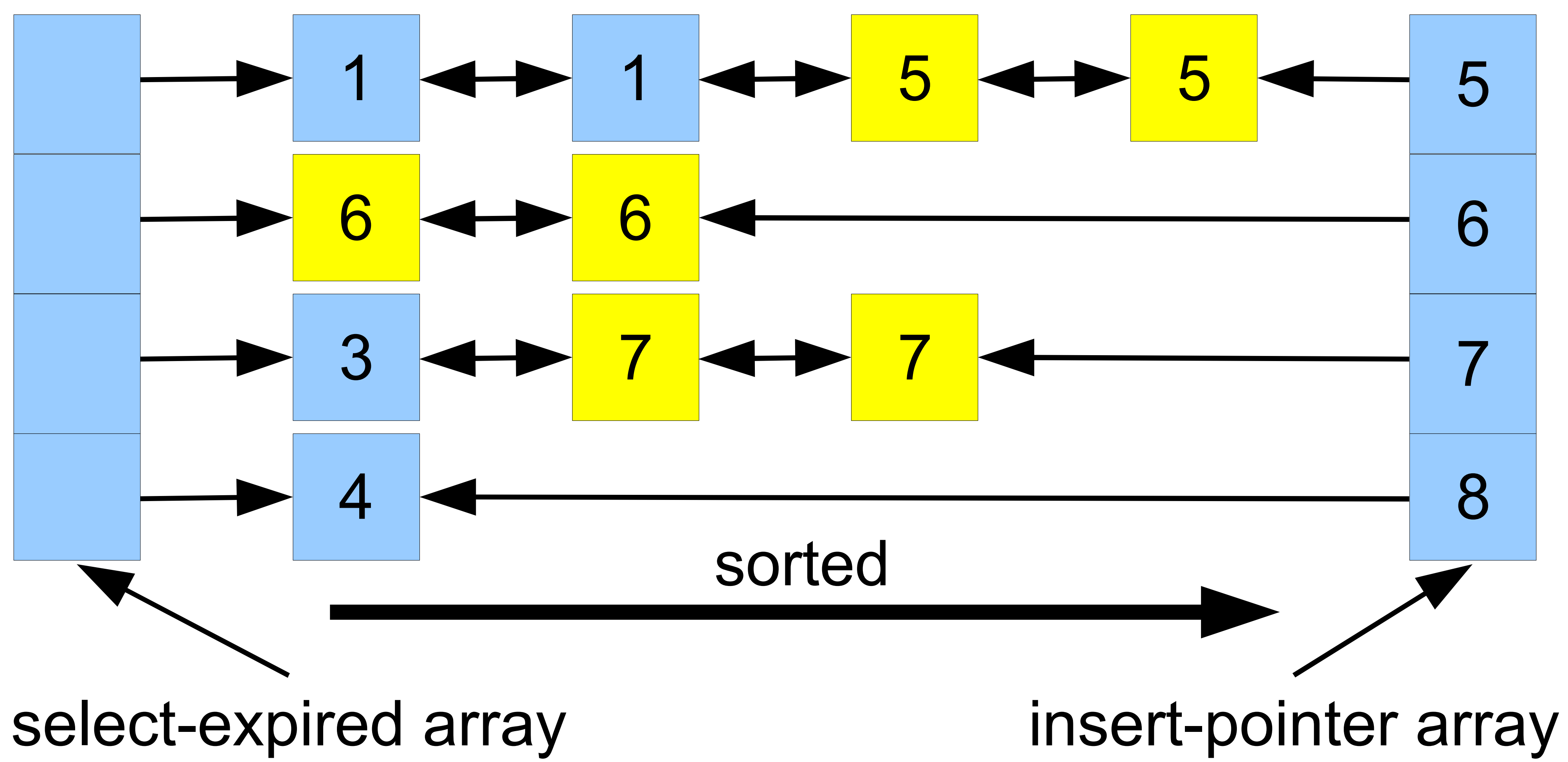


Figure 7. Segregated buffer implementation.

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

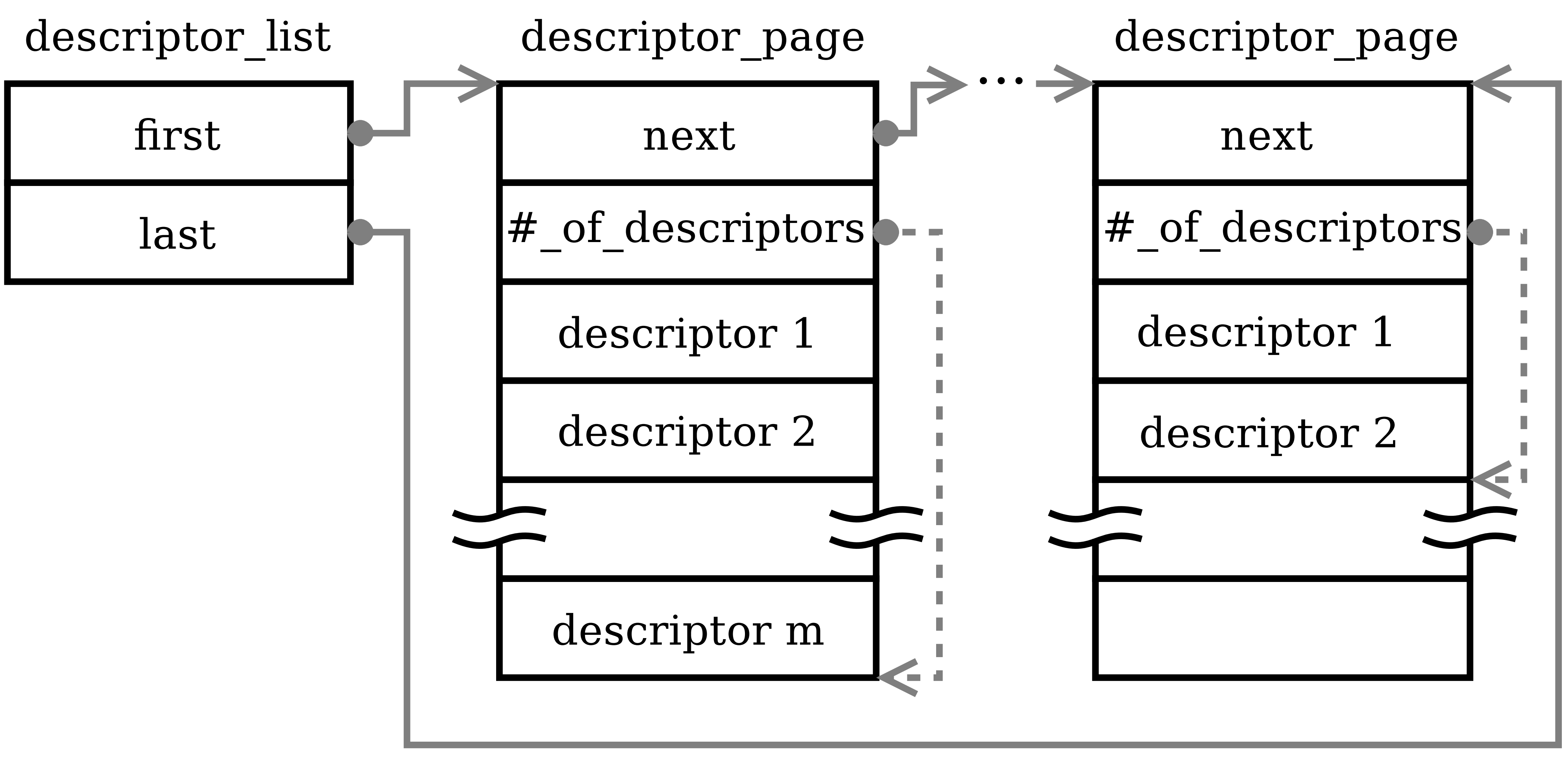


Figure 8. The design of the descriptor list.

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

Setup

CPU	2x AMD Opteron DualCore, 2.0 GHz
RAM	4GB
OS	Linux 2.6.32-21-generic
Java VM	Jikes RVM 3.1.0
C compiler	gcc version 4.4.3
C allocator	ptmalloc2-20011215 (glibc-2.10.1)

Table 3. System configuration.

Java: Memory

	MC leaky	MC fixed	4×MC fixed	JLayer	LuIndex
SCM(1,1)	40MB	40MB	60MB	95MB	370MB
SCM (50,20)	50MB	40MB	70MB	/	/
aggressive SCM(1,1)	/	/	/	90MB	250MB
GEN	95MB	40MB	70MB	95MB	370MB
MS	100MB	40MB	70MB	95MB	370MB

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

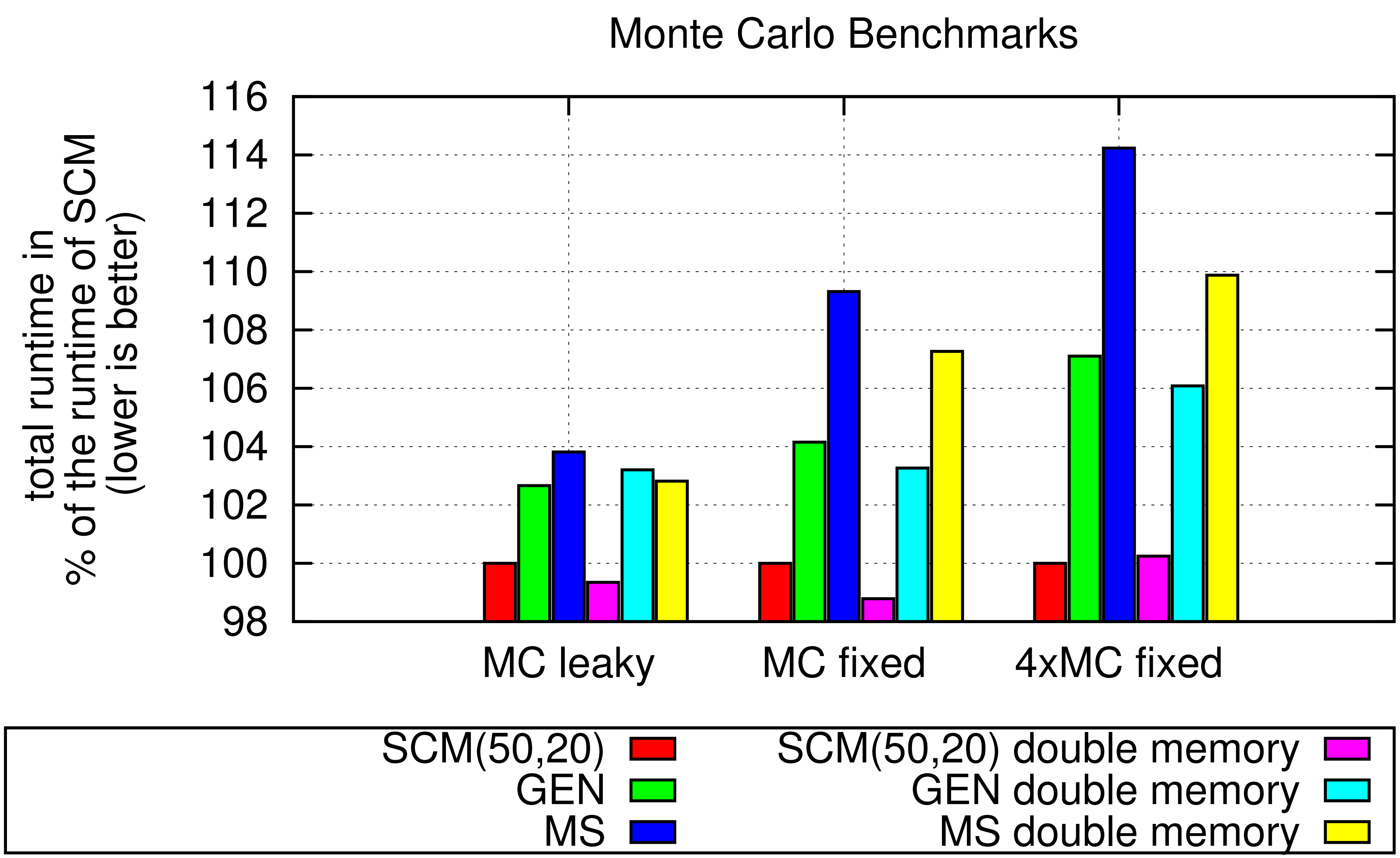


Figure 9. Total execution time of the Monte Carlo benchmarks in percentage of the total execution time of the benchmark using self-collecting mutators.

Java: Throughput

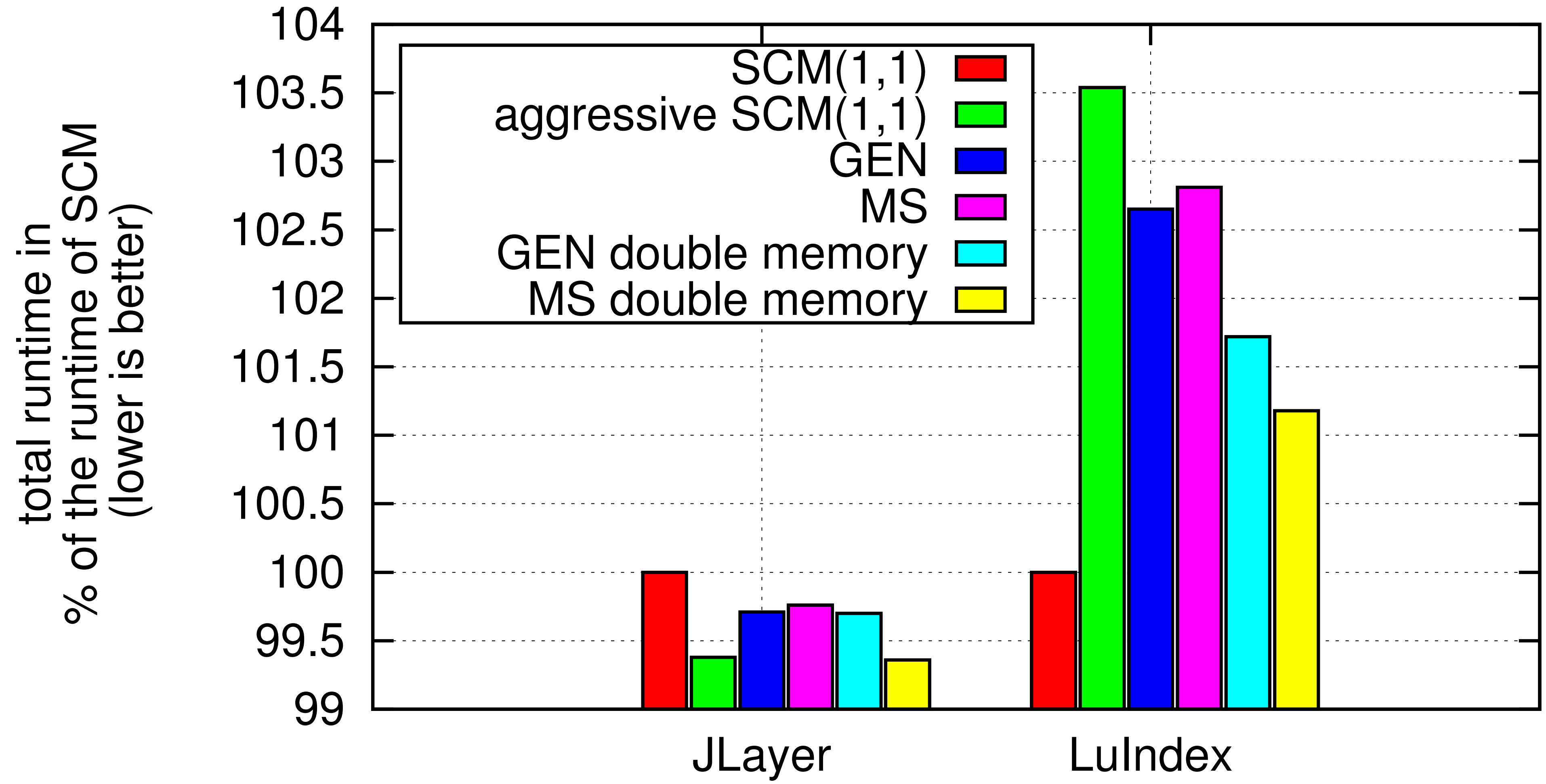


Figure 10. Total execution time of the JLayer and the LuIndex benchmarks in percentage of the total execution time of the benchmark using self-collecting mutators.

Java: Latency & Memory

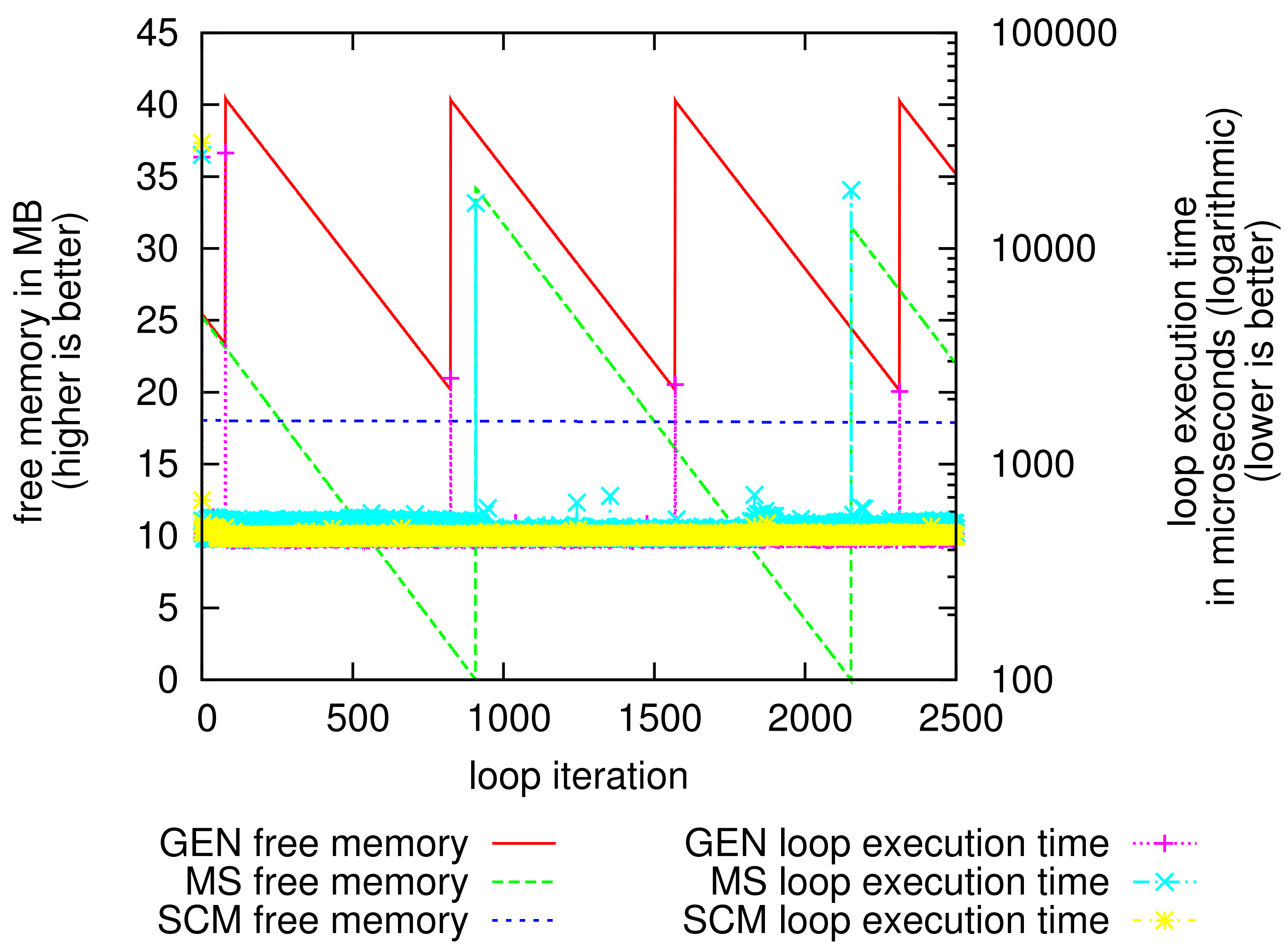


Figure 11. Free memory and loop execution time of the fixed Monte Carlo benchmark.

Java: Latency w/ Refreshing

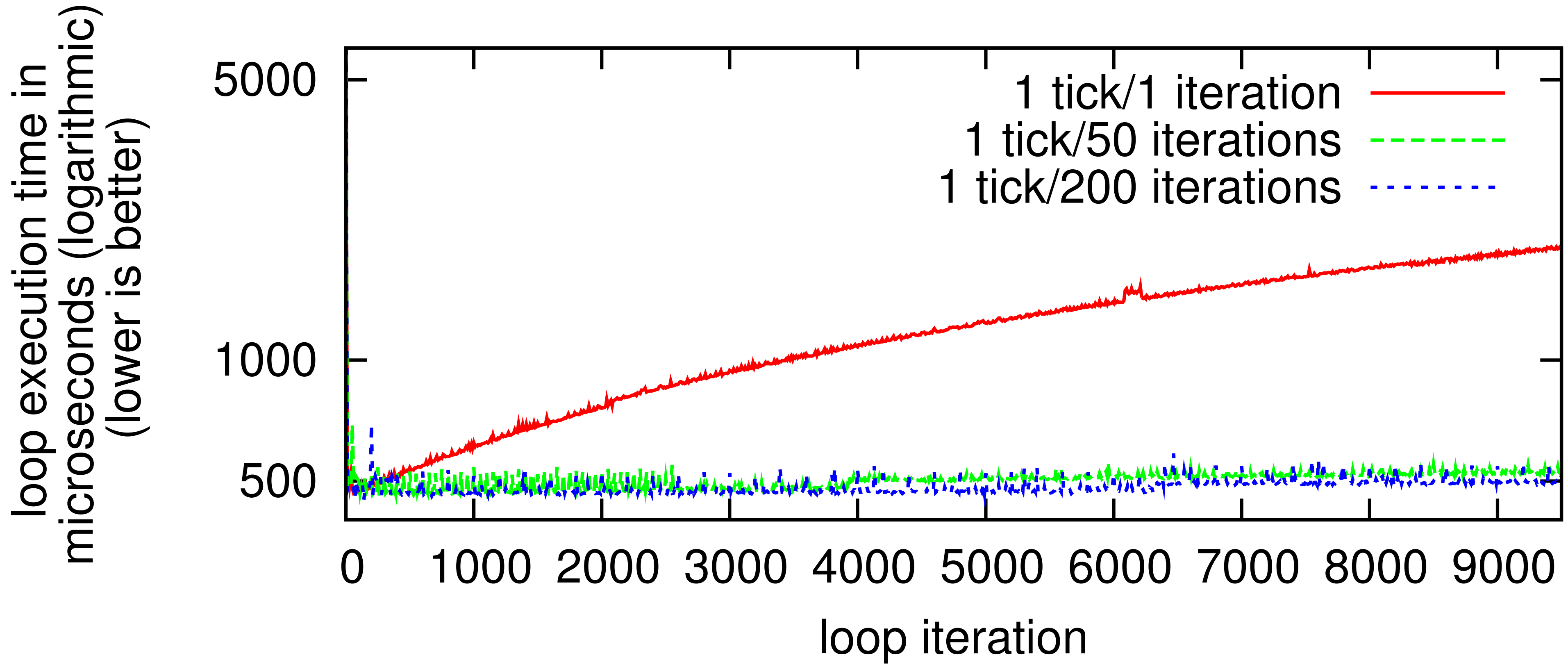


Figure 13. Loop execution time of the Monte Carlo benchmark with different tick frequencies. Self-collecting mutators is used.

Java: Memory w/ Refreshing

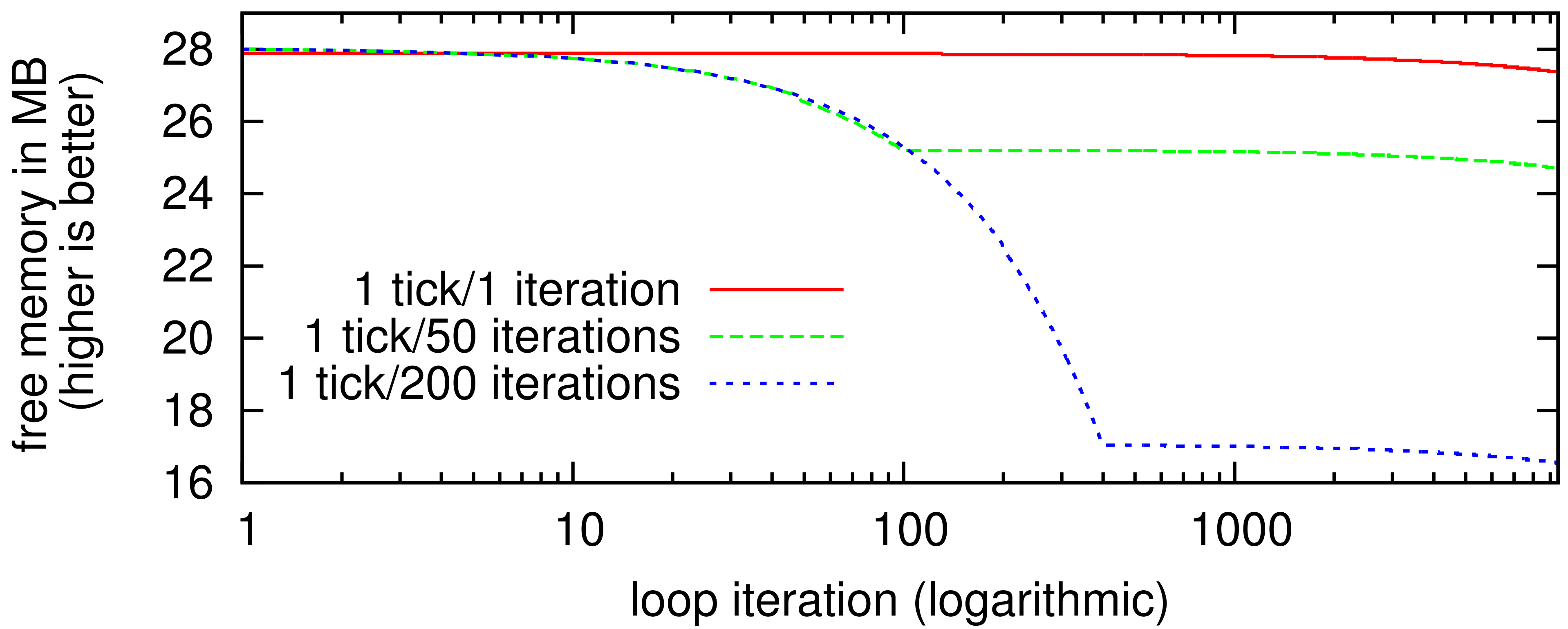


Figure 14. Free memory of the Monte Carlo benchmark with different tick frequencies. Self-collecting mutators is used.

C: Overhead

	persistent MM	short-term MM
malloc of ptmalloc2	166 (78 / 199k)	/
free of ptmalloc2	86 (14 / 169k)	/
malloc of SCM	172 (82 / 267k)	138 (75 / 271k)
free of SCM	91 (10 / 157k)	/
local-refresh(1, 256B)	/	227 (131 / 548k)
local-refresh(10, 256B)	/	225 (131 / 548k)
local-refresh(1, 4KB)	/	228 (131 / 548k)
local-refresh(10, 4KB)	/	230 (131 / 548k)
global-refresh(1, 256B)	/	226 (116 / 551k)
global-refresh(10, 256B)	/	224 (116 / 551k)
global-refresh(1, 4KB)	/	227 (116 / 551k)
global-refresh(10, 4KB)	/	228 (116 / 551k)
local-tick(1, 256B)	/	378 (277 / 164k)
local-tick(10, 256B)	/	359 (277 / 71k)
local-tick(1, 4KB)	/	375 (277 / 164k)
local-tick(10, 4KB)	/	366 (277 / 164k)
global-tick(1, 256B)	/	367 (229 / 169k)
global-tick(10, 256B)	/	352 (229 / 151k)
global-tick(1, 4KB)	/	365 (229 / 169k)
global-tick(10, 4KB)	/	361 (229 / 169k)

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.

C: Throughput

ptmalloc2	895.25ms	100.00%
ptmalloc2 through SCM	899.43ms	100.47%
local-SCM(1, 256B)	890.18ms	99.43%
local-SCM(10, 256B)	898.28ms	100.34%
local-SCM(1, 4KB)	892.18ms	99.66%
local-SCM(10, 4KB)	892.28ms	99.67%
global-SCM(1, 256B)	893.76ms	99.83%

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.

C: Memory

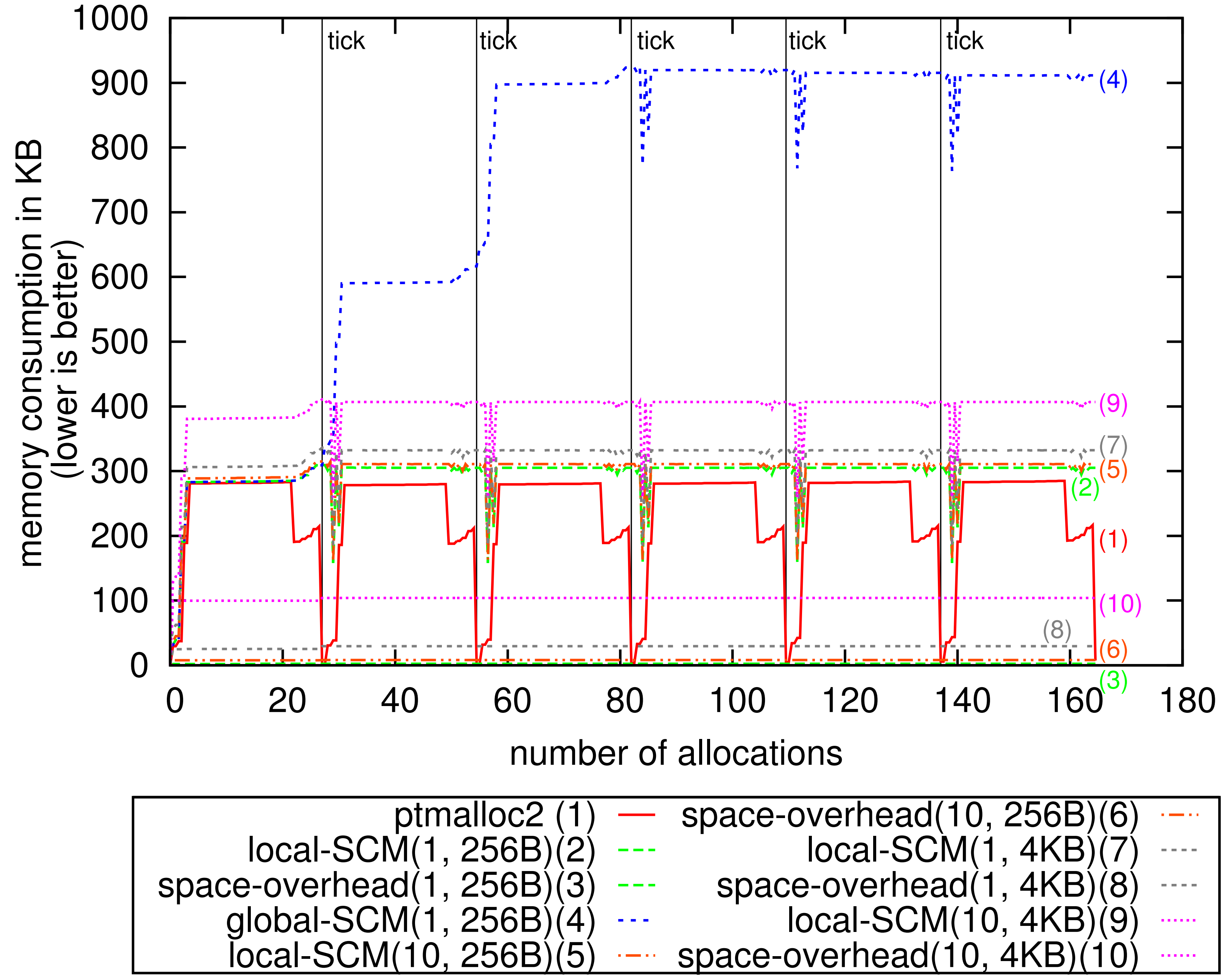


Figure 15. Memory overhead and consumption of the mpg123 benchmark. Again, local/global-SCM(n, m) stands for self-collecting mutators with a maximal expiration extension of n and descriptor page size m, using local/global-refresh. We write space-overhead(n, m) to denote the memory overhead of the local-SCM(n, m) configurations for storing descriptors and descriptor counters.


Thank you
Check out:
eurosys2011.cs.uni-salzburg.at