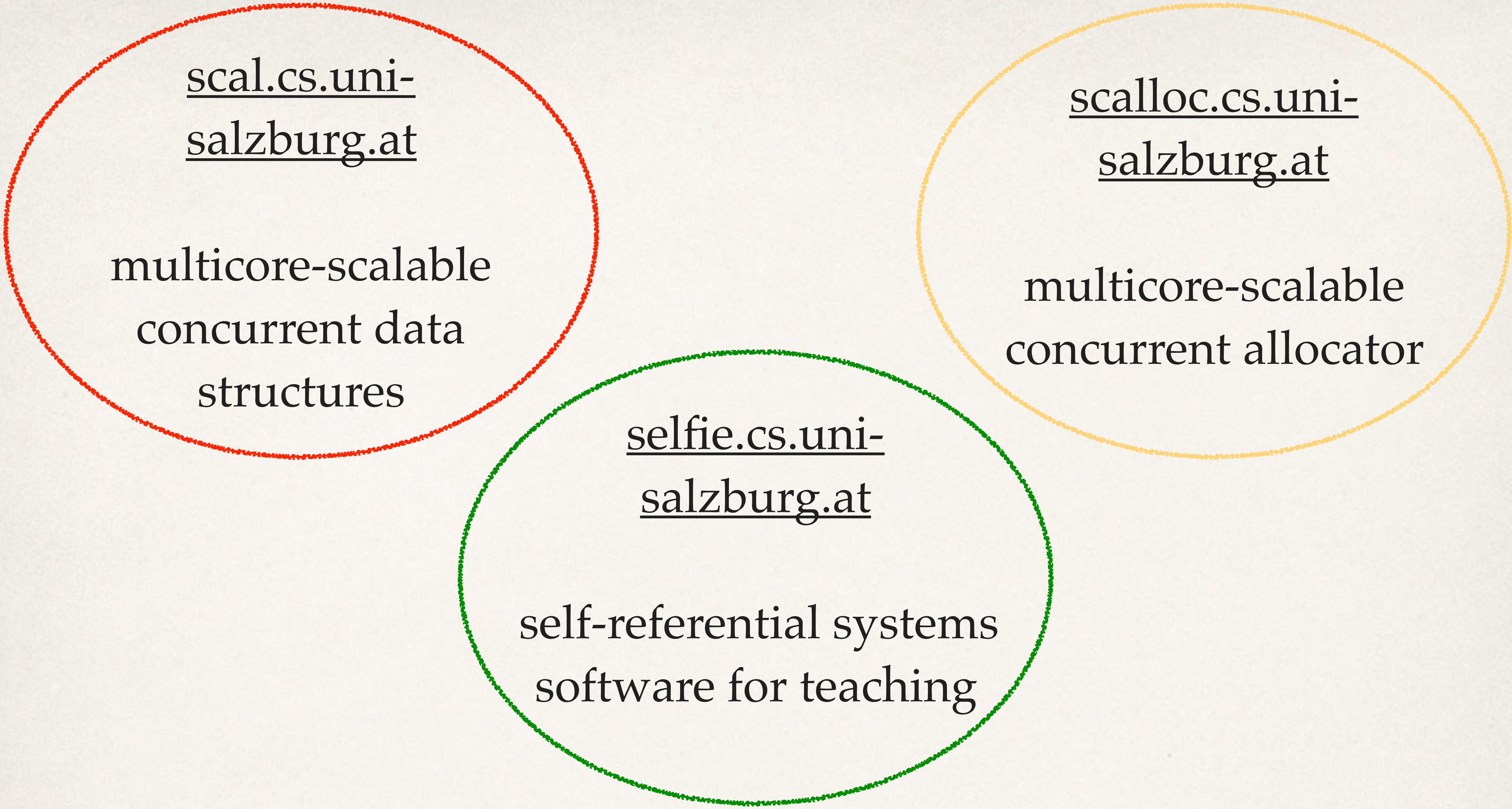


scal.cs.uni-salzburg.at

multicore-scalable concurrent data structures

scalloc.cs.uni-salzburg.at

multicore-scalable concurrent allocator

selfie.cs.uni-salzburg.at

self-referential systems software for teaching

Scal, Scalloc, and Selfie

Christoph Kirsch, University of Salzburg, Austria

North Eastern University, November 2015

Joint Work

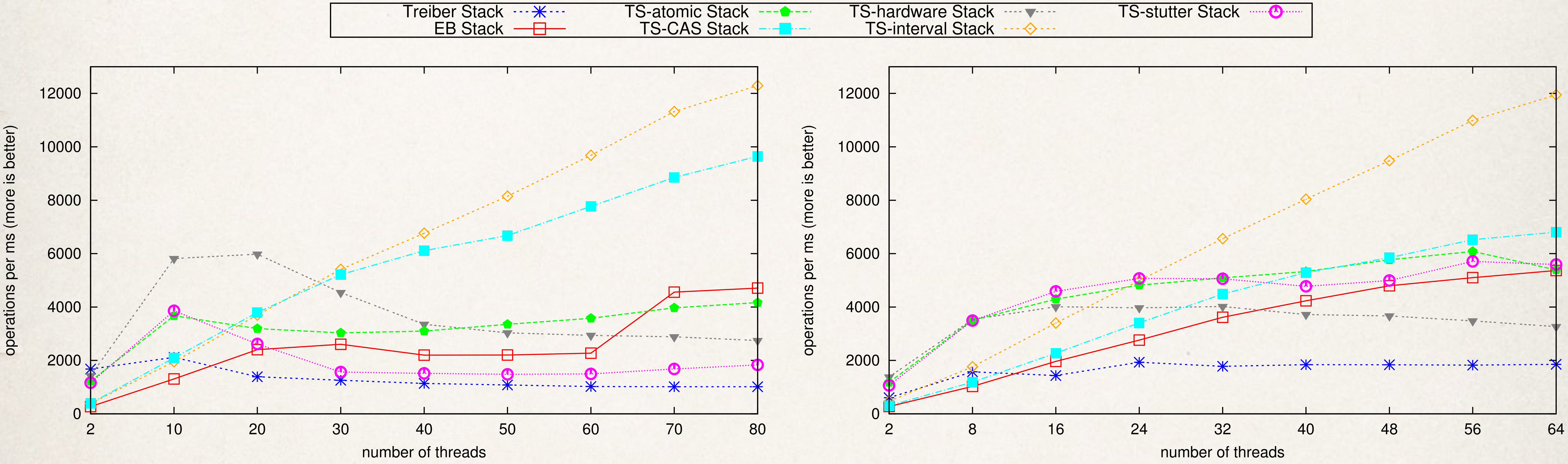
- Martin Aigner
- Christian Barthel
- Mike Dodds
- Andreas Haas
- Thomas Henzinger
- Andreas Holzer
- Thomas Hütter
- Michael Lippautz
- Alexander Miller
- Simone Oblasser
- Hannes Payer
- Mario Preishuber
- Ana Sokolova
- Ali Szegin

Infrastructural Software

- We are interested in designing and implementing concurrent data structures that are <u>fast</u> and <u>scale</u> on multicore hardware.

- We then apply the best designs in other <u>infrastructural</u> software such as a memory allocator.

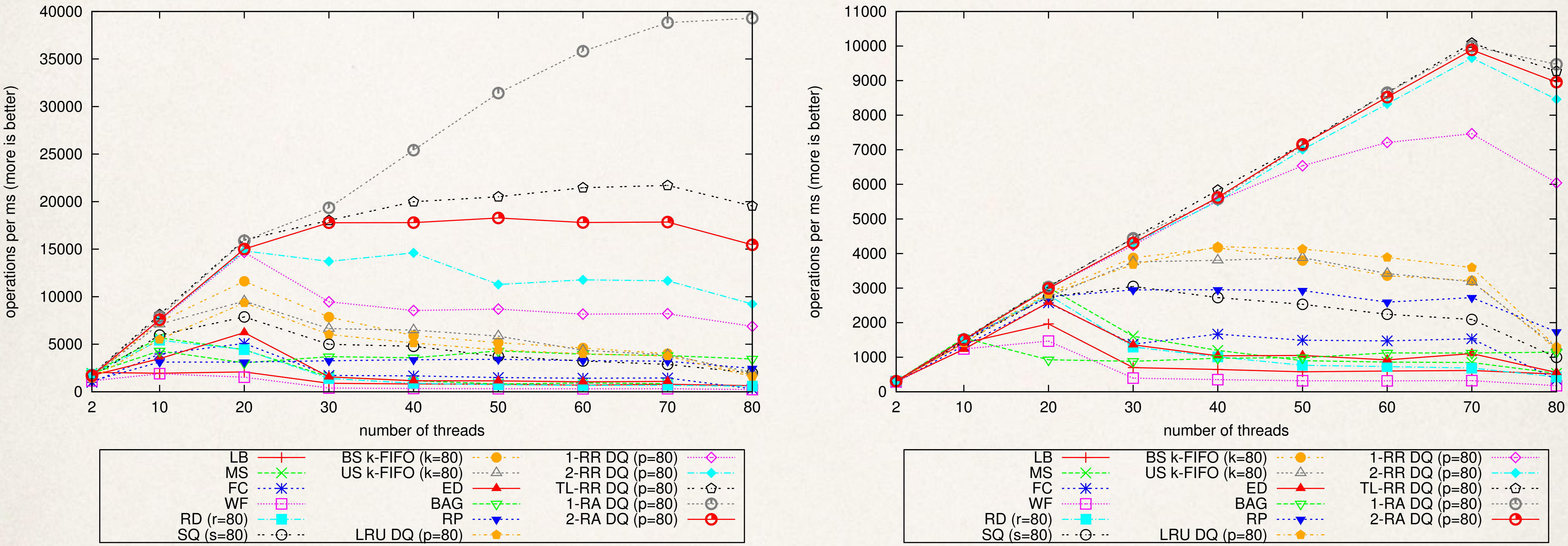
Timestamped (TS) Stack [POPL15]



(a) Producer-consumer benchmark, 40-core machine.

(b) Producer-consumer benchmark, 64-core machine.

Distributed Queues [CF13]



(a) High contention producer-consumer microbenchmark ($c = 250$) (b) Low contention producer-consumer microbenchmark ($c = 2000$)

Figure 1: Performance and scalability of producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) server machine

Concurrent Data Structures: scal.cs.uni-salzburg.at [POPL13, CF13, POPL15, NETYS15]

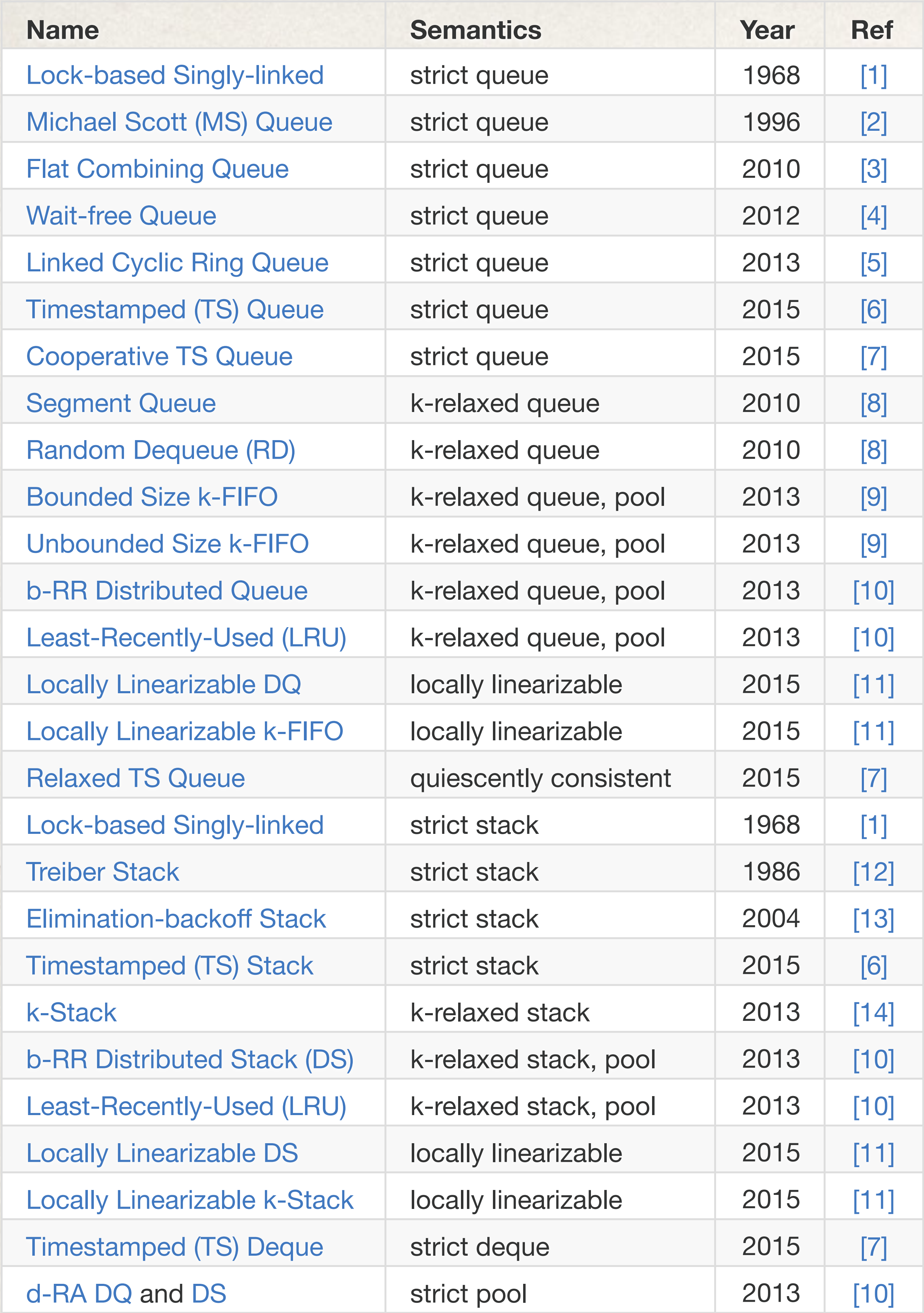
- Scal is an open-source benchmarking framework that provides

  1. software infrastructure for executing concurrent data structure algorithms,

  2. workloads for benchmarking their performance and scalability, and

  3. implementations of a large set of concurrent data structures.

Scal: A Benchmarking Suite for Concurrent Data Structures [NETYS15]



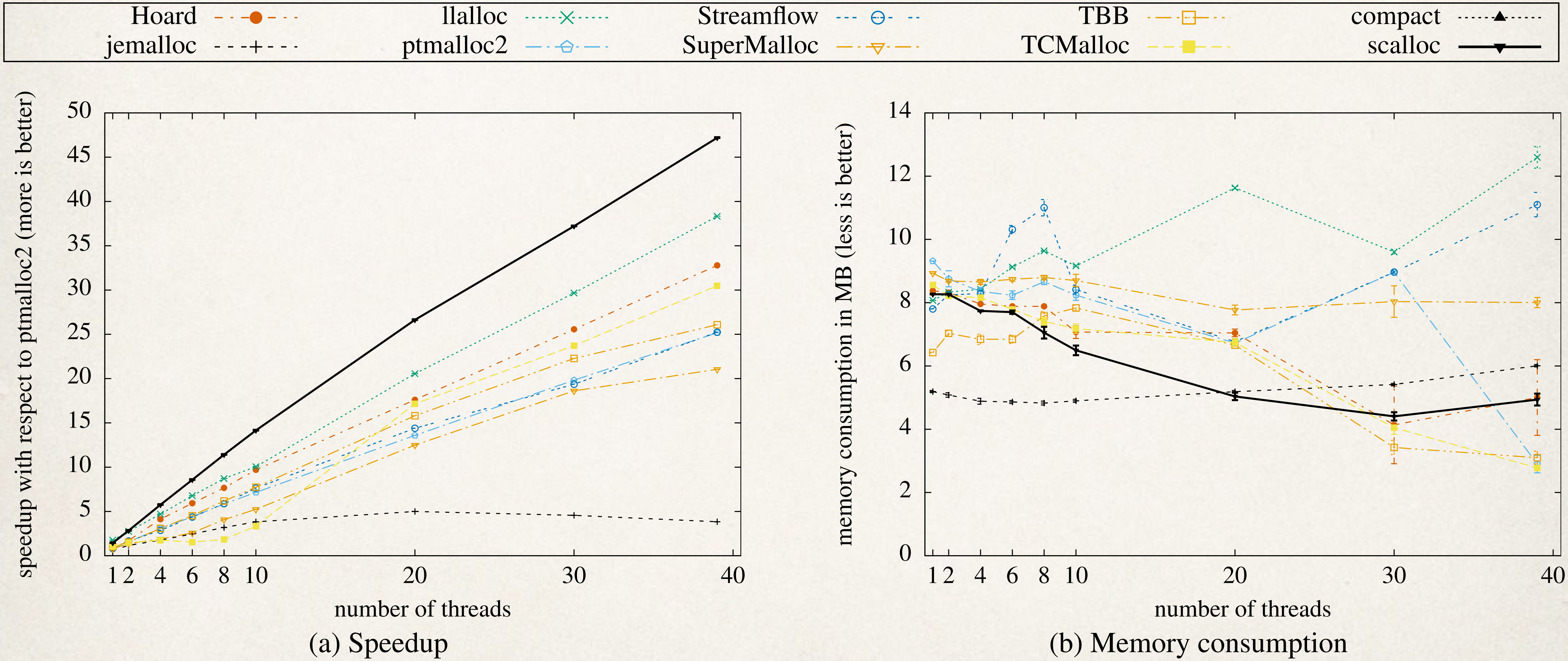
Name	Semantics	Year	Ref
Lock-based Singly-linked	strict queue	1968	[1]
Michael Scott (MS) Queue	strict queue	1996	[2]
Flat Combining Queue	strict queue	2010	[3]
Wait-free Queue	strict queue	2012	[4]
Linked Cyclic Ring Queue	strict queue	2013	[5]
Timestamped (TS) Queue	strict queue	2015	[6]
Cooperative TS Queue	strict queue	2015	[7]
Segment Queue	k-relaxed queue	2010	[8]
Random Dequeue (RD)	k-relaxed queue	2010	[8]
Bounded Size k-FIFO	k-relaxed queue, pool	2013	[9]
Unbounded Size k-FIFO	k-relaxed queue, pool	2013	[9]
b-RR Distributed Queue	k-relaxed queue, pool	2013	[10]
Least-Recently-Used (LRU)	k-relaxed queue, pool	2013	[10]
Locally Linearizable DQ	locally linearizable	2015	[11]
Locally Linearizable k-FIFO	locally linearizable	2015	[11]
Relaxed TS Queue	quiescently consistent	2015	[7]
Lock-based Singly-linked	strict stack	1968	[1]
Treiber Stack	strict stack	1986	[12]
Elimination-backoff Stack	strict stack	2004	[13]
Timestamped (TS) Stack	strict stack	2015	[6]
k-Stack	k-relaxed stack	2013	[14]
b-RR Distributed Stack (DS)	k-relaxed stack, pool	2013	[10]
Least-Recently-Used (LRU)	k-relaxed stack, pool	2013	[10]
Locally Linearizable DS	locally linearizable	2015	[11]
Locally Linearizable k-Stack	locally linearizable	2015	[11]
Timestamped (TS) Deque	strict deque	2015	[7]
d-RA DQ and DS	strict pool	2013	[10]

Scalloc: Concurrent Memory Allocator scalloc.cs.uni-salzburg.at [OOPSLA15]

- fast, multicore-scalable, low-memory-overhead allocator
- three key ideas:

1. backend: single global concurrent data structure for reclaiming memory effectively and efficiently
2. virtual spans: single algorithm for small and big objects
3. frontend: constant-time (modulo synchronization) allocation and eager deallocation

Local Allocation & Deallocation

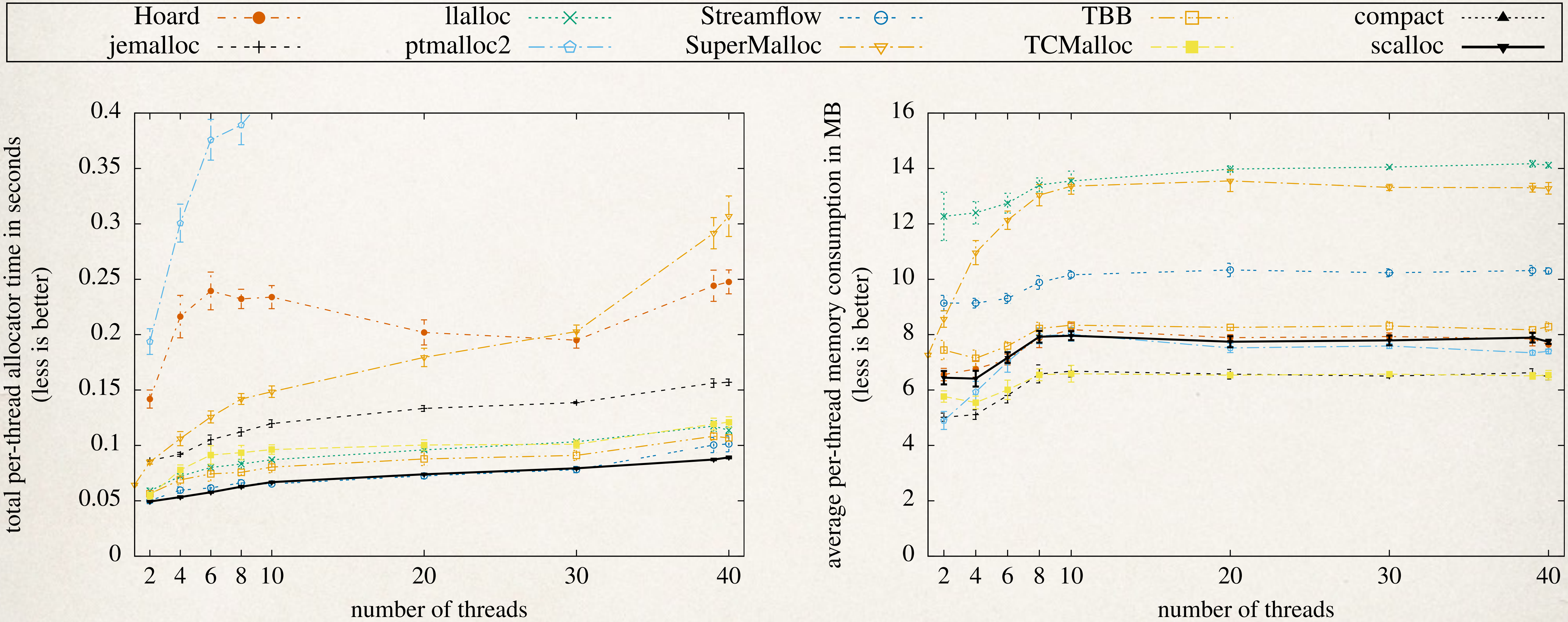


(a) Speedup

(b) Memory consumption

Figure 6: Thread-local workload: Threadtest benchmark

Remote Deallocation

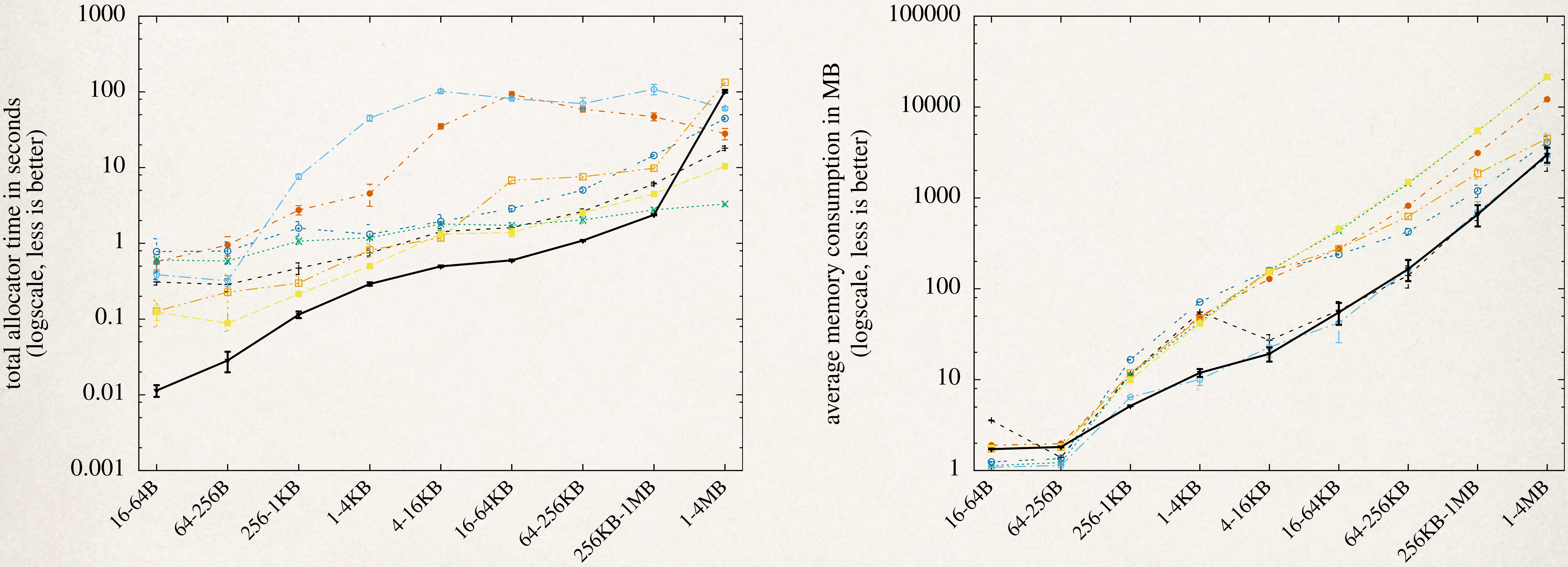


(a) Total per-thread allocator time

(b) Average per-thread memory consumption

Figure 9: Temporal and spatial performance for the producer-consumer experiment

Object Size



(a) Total allocator time

(b) Average memory consumption

Figure 10: Temporal and spatial performance for the object-size robustness experiment at 40 threads

Memory Access

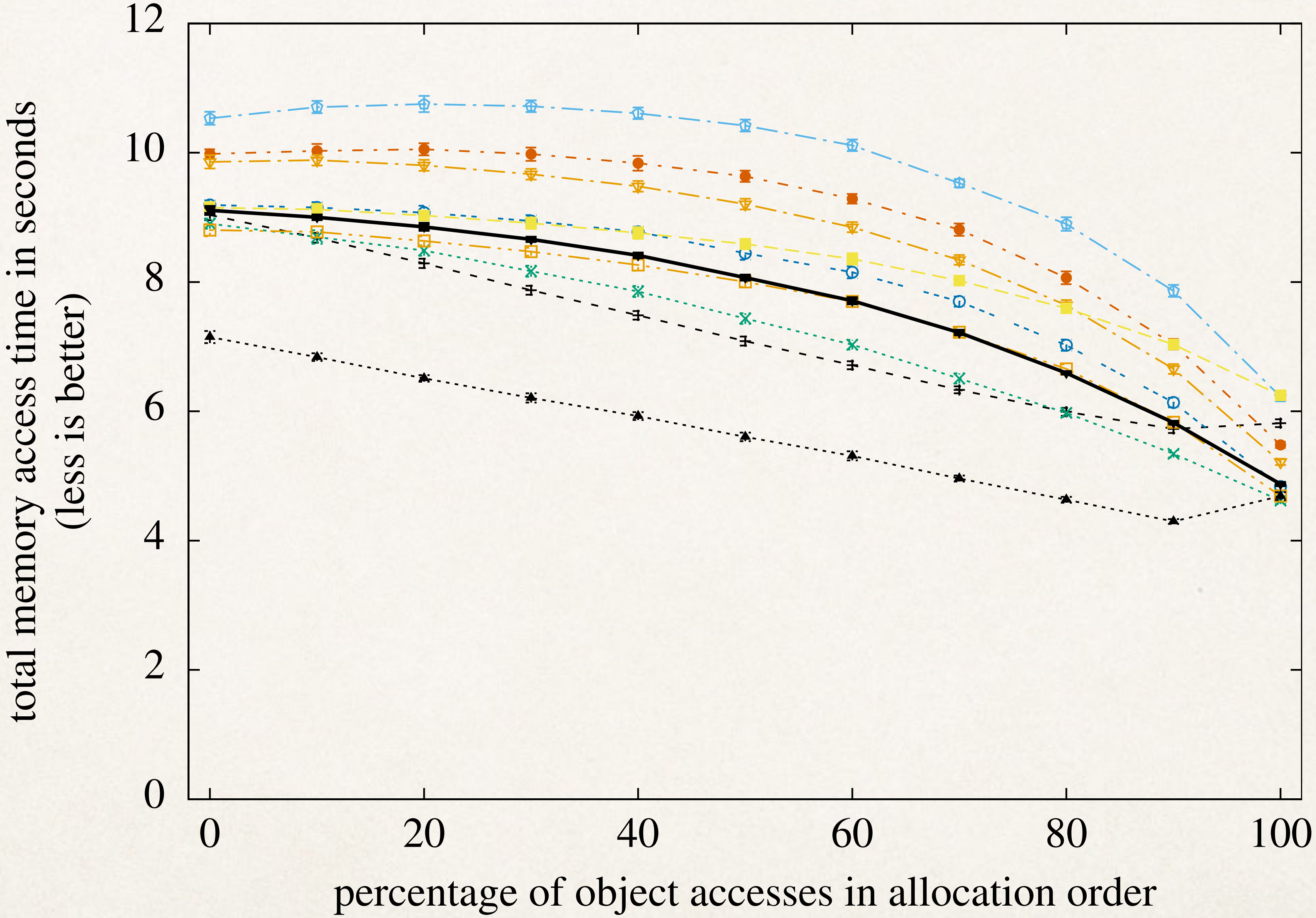


Figure 11: Memory access time for the locality experiment

Virtual Spans: 64-bit Address Space

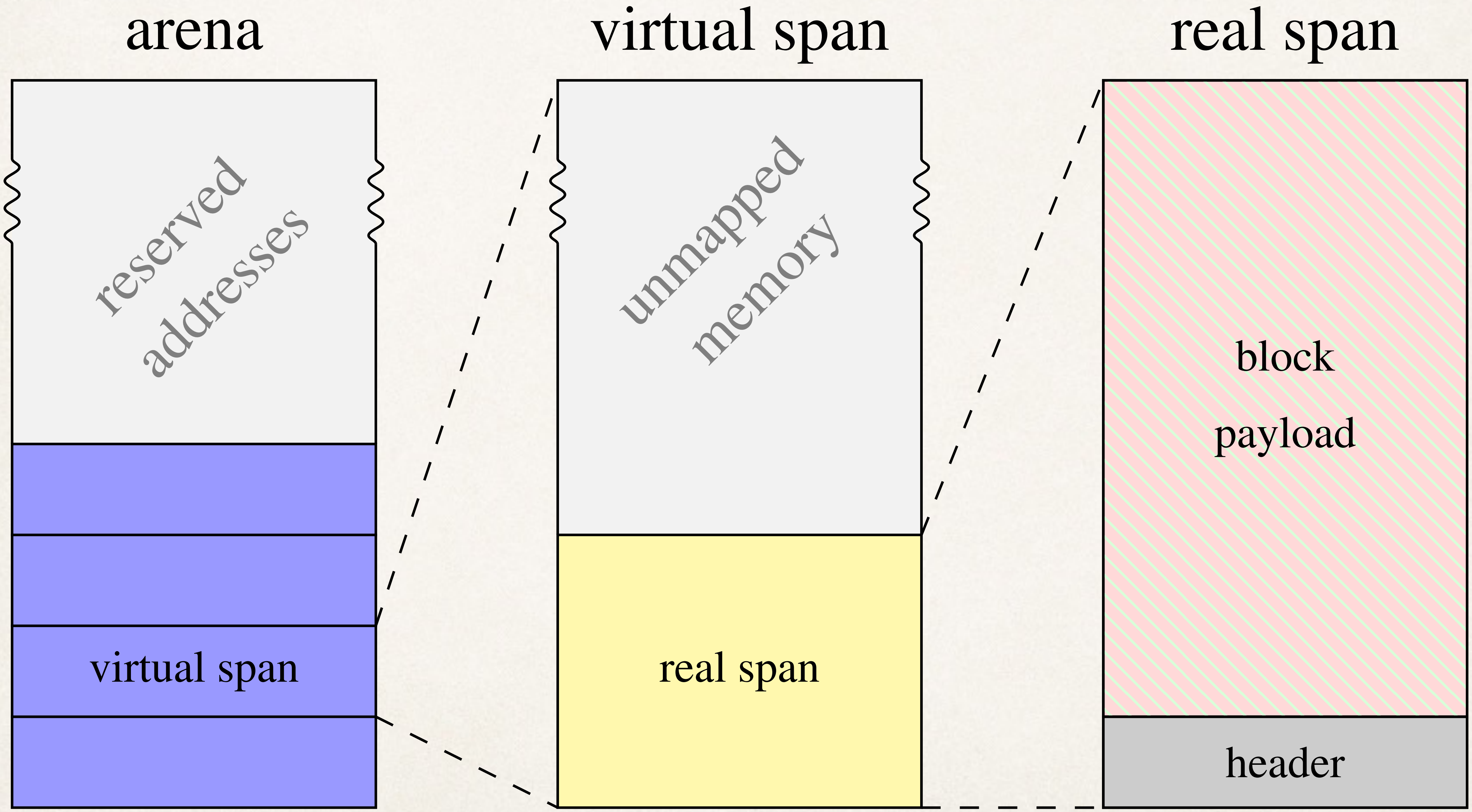


Figure 1: Structure of arena, virtual spans, and real spans

Backend: Double Segregation

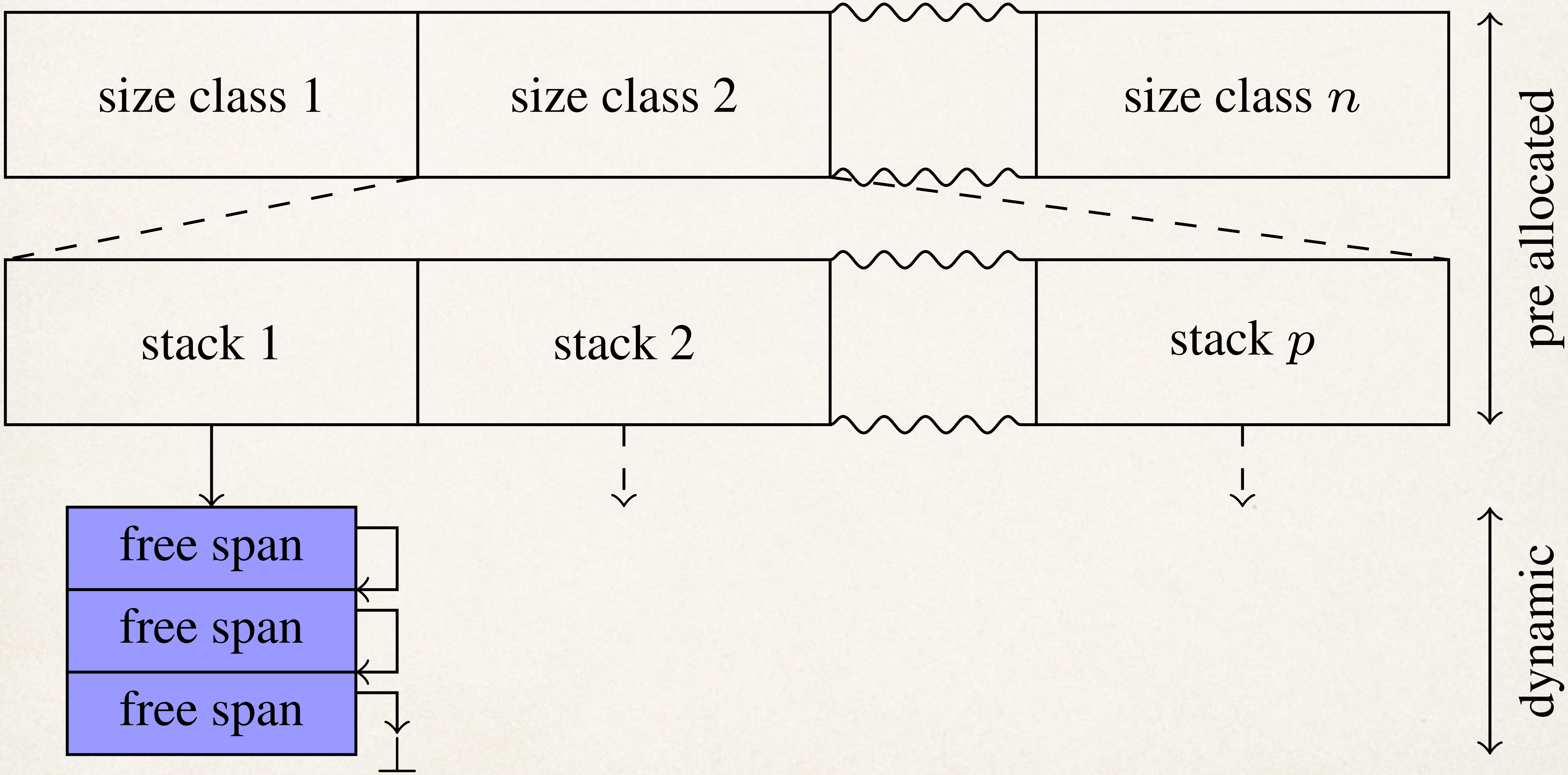


Figure 2: Span pool layout

Frontend: Eager Memory Reuse

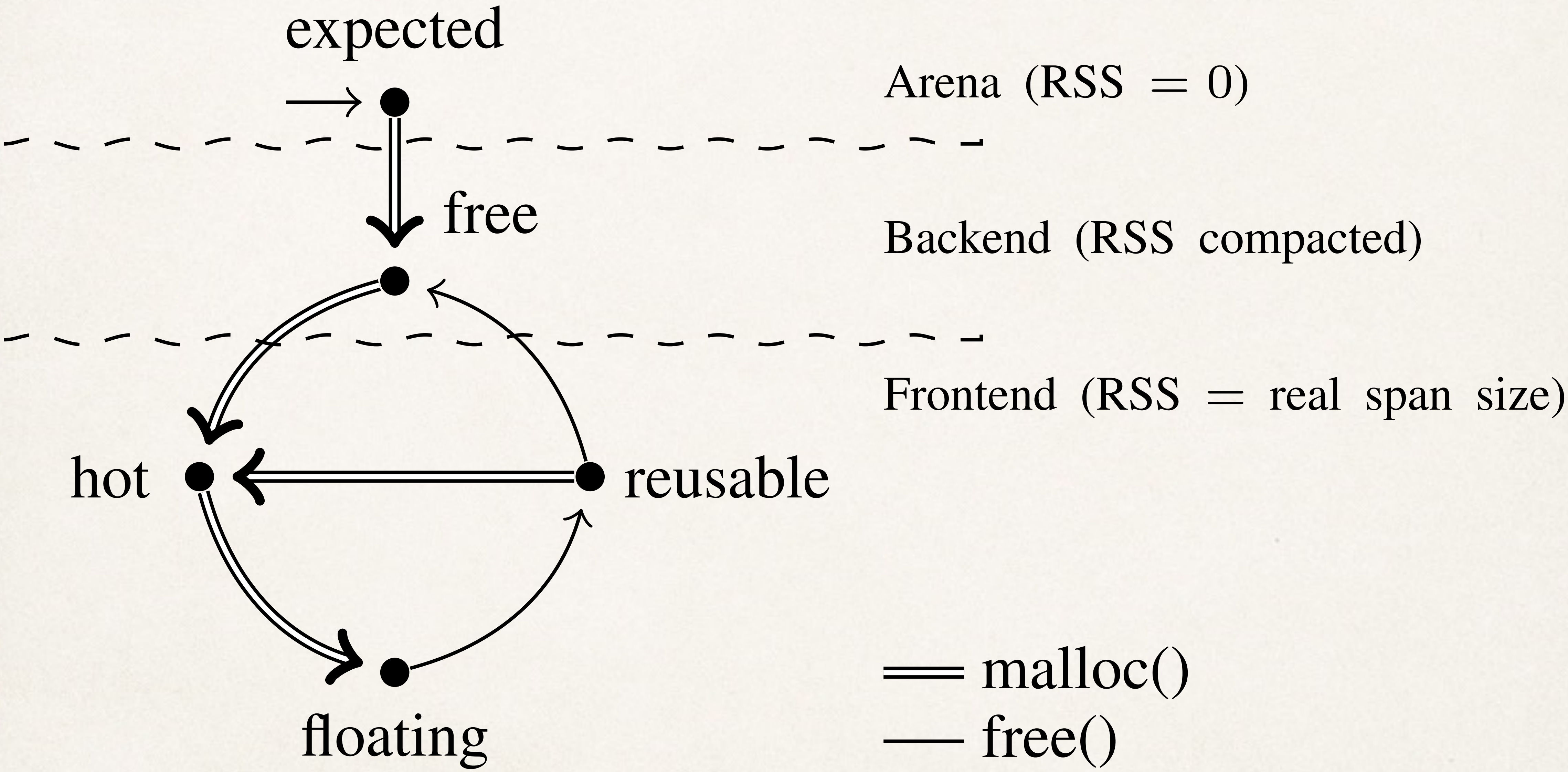


Figure 3: Life cycle of a span

Selfie: Teaching Systems Engineering [selfie.cs.uni-salzburg.at]

- Selfie is a fully self-referential 4k-line C implementation of:

  1. a self-compiling compiler called cstarc that compiles a tiny subset of C called C Star (C*) to a tiny subset of MIPS32 called MIPSter,

  2. a self-executing emulator called mipster that executes MIPSter code including itself when compiled with cstarc, and

  3. a tiny C* library called libcstar utilized by cstarc and mipster.

5 statements:
assignment
while
if
return
procedure()

```
int atoi(int *s) {
    int i;
    int n;
    int c;

    i = 0;
    n = 0;
    c = *(s+i);

    while (c != 0) {
        n = n * 10 + c - '0';
        if (n < 0)
            return -1;

        i = i + 1;
        c = *(s+i);
    }

    return n;
}
```

no data structures,
just `int` and `int*`
and dereferencing:
the * operator

character constants
string constants

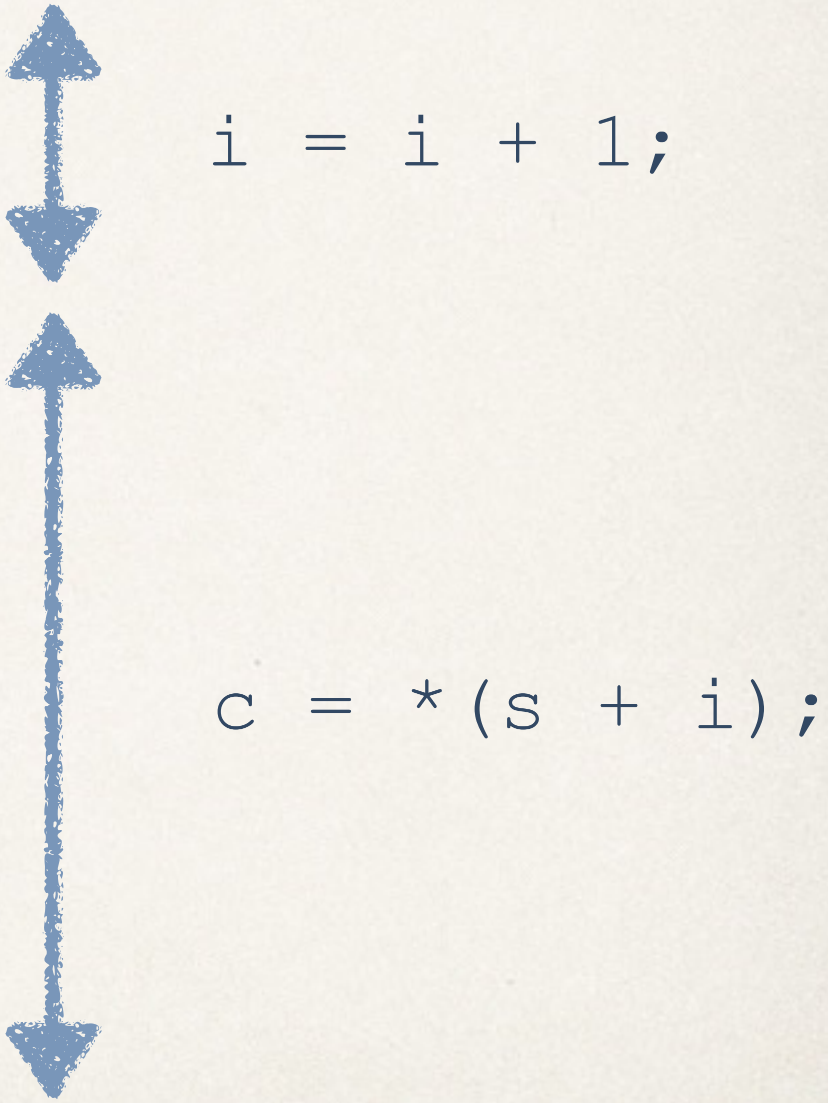
integer arithmetics
pointer arithmetics

no bitwise operators
no Boolean operators

library: `exit`, `malloc`, `open`, `read`, `write`

MIPSter: 15 out of 43 Instructions

```
atoi.c: $pc=0x000001CC: lw $t0,-4($fp)
atoi.c: $pc=0x000001D0: addiu $t1,$zero,1
atoi.c: $pc=0x000001D4: addu $t0,$t0,$t1
atoi.c: $pc=0x000001D8: sw $t0,-4($fp)
atoi.c: $pc=0x000001DC: lw $t0,8($fp)
atoi.c: $pc=0x000001E0: lw $t1,-4($fp)
atoi.c: $pc=0x000001E4: addiu $t2,$zero,4
atoi.c: $pc=0x000001E8: multu $t1,$t2
atoi.c: $pc=0x000001EC: mflo $t1
atoi.c: $pc=0x000001F0: nop
atoi.c: $pc=0x000001F4: nop
atoi.c: $pc=0x000001F8: addu $t0,$t0,$t1
atoi.c: $pc=0x000001FC: lw $t0,0($t0)
atoi.c: $pc=0x00000200: sw $t0,-12($fp)
```



`i = i + 1;`

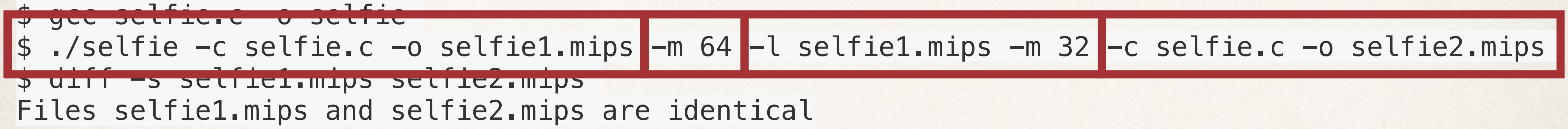
`c = *(s + i);`

Self-Compilation

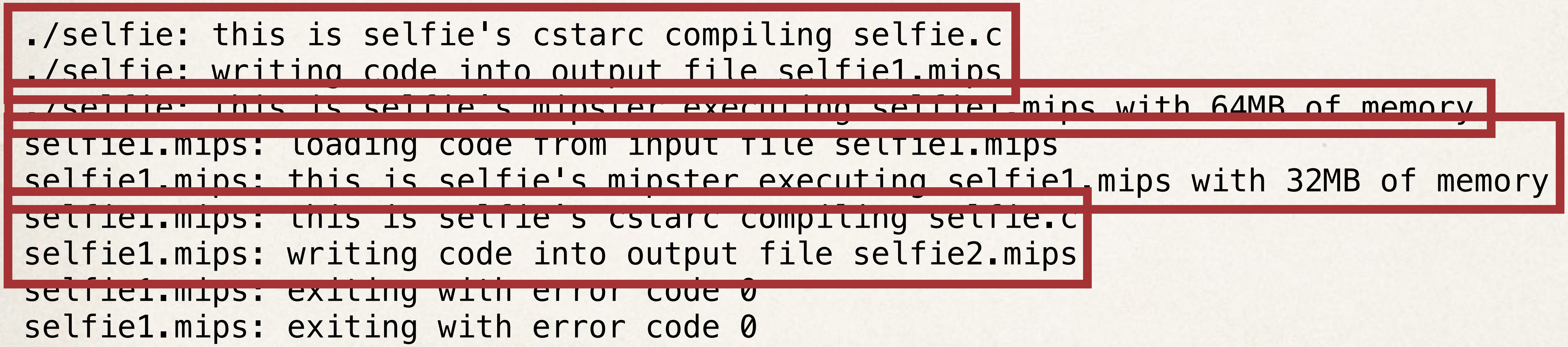
```
$ gcc selfie.c -o selfie
$ ./selfie -c selfie.c -o selfie1.mips -m 32 -c selfie.c -o selfie2.mips
$ diff -s selfie1.mips selfie2.mips
Files selfie1.mips and selfie2.mips are identical
```

```
./selfie: this is selfie's cstarc compiling selfie.c
./selfie: writing code into output file selfie1.mips
./selfie: this is selfie's mipster executing selfie1.mips with 32MB of memory
selfie1.mips: this is selfie's cstarc compiling selfie.c
selfie1.mips: writing code into output file selfie2.mips
selfie1.mips: exiting with error code 0
```

Self-Execution



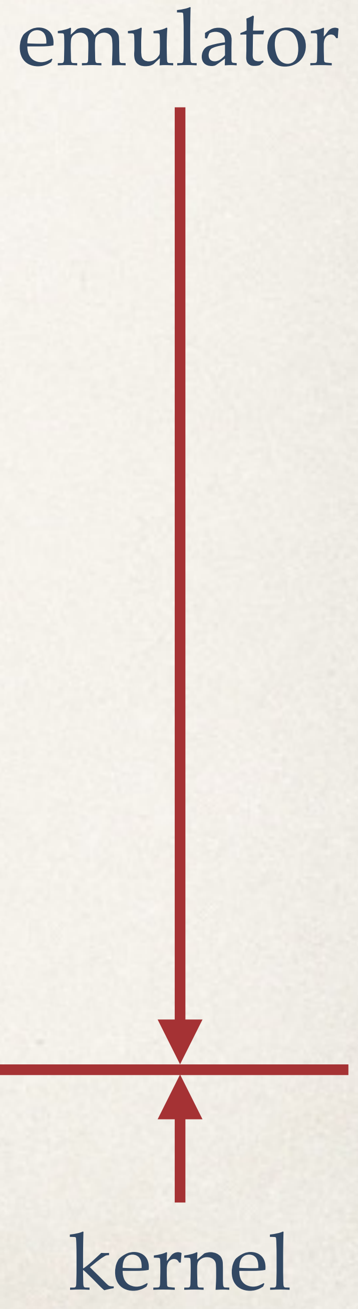
```
$ gcc selfie.c -o selfie
$ ./selfie -c selfie.c -o selfie1.mips -m 64 -l selfie1.mips -m 32 -c selfie.c -o selfie2.mips
$ diff -s selfie1.mips selfie2.mips
Files selfie1.mips and selfie2.mips are identical
```



```
./selfie: this is selfie's cstarc compiling selfie.c
./selfie: writing code into output file selfie1.mips
./selfie: this is selfie's mipster executing selfie1.mips with 64MB of memory
selfie1.mips: loading code from input file selfie1.mips
selfie1.mips: this is selfie's mipster executing selfie1.mips with 32MB of memory
selfie1.mips: this is selfie's cstarc compiling selfie.c
selfie1.mips: writing code into output file selfie2.mips
selfie1.mips: exiting with error code 0
selfie1.mips: exiting with error code 0
```

Selfie OS Kernel: Controlling Self-Referentiality

- “system calls”: `exit`, `malloc`, `open`, `read`, `write`
- assignment 0: sorted, singly-linked list in C*
- assignment 1: preemptive scheduling, process switching
- assignment 2: cooperative scheduling (`yield` “system call”), memory segmentation
- assignment 3: locking (`lock` and `unlock` “system calls” to protect I/O)
- assignment 4: processes (`fork` and `wait` “system calls”), threads, shared memory
- assignment 5: locking (to protect data)
- assignment 6: virtual memory, paging
- assignment 7: bootstrapping (`switch` “system call”)
- assignment 8: user and kernel mode



emulator

kernel

Future Work with Selfie et al.

- I/O
- file systems
- virtualization
- memory allocation
- garbage collection
- concurrency: <u>semantics</u>
- volatility: <u>persistent</u> memory

Thank you!