Scal, Scalloc, and Selfie

Christoph Kirsch, University of Salzburg, Austria
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
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.
<table>
<thead>
<tr>
<th>Name</th>
<th>Semantics</th>
<th>Year</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock-based Singly-linked</td>
<td>strict queue</td>
<td>1968</td>
<td>[1]</td>
</tr>
<tr>
<td>Michael Scott (MS) Queue</td>
<td>strict queue</td>
<td>1996</td>
<td>[2]</td>
</tr>
<tr>
<td>Flat Combining Queue</td>
<td>strict queue</td>
<td>2010</td>
<td>[3]</td>
</tr>
<tr>
<td>Wait-free Queue</td>
<td>strict queue</td>
<td>2012</td>
<td>[4]</td>
</tr>
<tr>
<td>Linked Cyclic Ring Queue</td>
<td>strict queue</td>
<td>2013</td>
<td>[5]</td>
</tr>
<tr>
<td>Timestamped (TS) Queue</td>
<td>strict queue</td>
<td>2015</td>
<td>[6]</td>
</tr>
<tr>
<td>Cooperative TS Queue</td>
<td>strict queue</td>
<td>2015</td>
<td>[7]</td>
</tr>
<tr>
<td>Segment Queue</td>
<td>k-relaxed queue</td>
<td>2010</td>
<td>[8]</td>
</tr>
<tr>
<td>Random Dequeue (RD)</td>
<td>k-relaxed queue</td>
<td>2010</td>
<td>[8]</td>
</tr>
<tr>
<td>Bounded Size k-FIFO</td>
<td>k-relaxed queue, pool</td>
<td>2013</td>
<td>[9]</td>
</tr>
<tr>
<td>Unbounded Size k-FIFO</td>
<td>k-relaxed queue, pool</td>
<td>2013</td>
<td>[9]</td>
</tr>
<tr>
<td>b-RR Distributed Queue</td>
<td>k-relaxed queue, pool</td>
<td>2013</td>
<td>[10]</td>
</tr>
<tr>
<td>Least-Recently-Used (LRU)</td>
<td>k-relaxed queue, pool</td>
<td>2013</td>
<td>[10]</td>
</tr>
<tr>
<td>Relaxed TS Queue</td>
<td>quiescently consistent</td>
<td>2015</td>
<td>[7]</td>
</tr>
<tr>
<td>Lock-based Singly-linked</td>
<td>strict stack</td>
<td>1968</td>
<td>[1]</td>
</tr>
<tr>
<td>Treiber Stack</td>
<td>strict stack</td>
<td>1986</td>
<td>[12]</td>
</tr>
<tr>
<td>Elimination-backoff Stack</td>
<td>strict stack</td>
<td>2004</td>
<td>[13]</td>
</tr>
<tr>
<td>Timestamped (TS) Stack</td>
<td>strict stack</td>
<td>2015</td>
<td>[6]</td>
</tr>
<tr>
<td>k-Stack</td>
<td>k-relaxed stack</td>
<td>2013</td>
<td>[14]</td>
</tr>
<tr>
<td>b-RR Distributed Stack (DS)</td>
<td>k-relaxed stack, pool</td>
<td>2013</td>
<td>[10]</td>
</tr>
<tr>
<td>Least-Recently-Used (LRU)</td>
<td>k-relaxed stack, pool</td>
<td>2013</td>
<td>[10]</td>
</tr>
<tr>
<td>Timestamped (TS) Deque</td>
<td>strict deque</td>
<td>2015</td>
<td>[7]</td>
</tr>
<tr>
<td>d-RA DQ and DS</td>
<td>strict pool</td>
<td>2013</td>
<td>[10]</td>
</tr>
</tbody>
</table>
**k-FIFO Queues [PaCT13]**

(a) Very high contention \((c = 1000, i = 0)\)

(b) High contention \((c = 4000, i = 0)\)
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 hyper-threads per core) server machine
Timestamped (TS) Stack [POPL15]

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

(b) Producer-consumer benchmark, 64-core machine.
Local Linearizability [CONCUR16]

“queue-like” data structures

“stack-like” data structures

Figure 5 Performance and scalability of producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) machine
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

Figure 6: Thread-local workload: Threadtest benchmark

Figure 7: Thread-local workload: Shbench benchmark

Figure 8: Thread-local workload (including thread termination): Larson benchmark
Remote Deallocaton

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

(a) Total per-thread allocator time

(b) Average per-thread memory consumption

7.4 Robustness against False Sharing

False sharing occurs when objects that are allocated in the same cache line are read from and written to by different threads. In cache coherent systems this scenario can lead to performance degradation as all caches need to be kept consistent. An allocator is prone to active false sharing if objects that are allocated by different threads (without communication) end up in the same cache line. It is prone to passive false sharing if objects that are remotely deallocated by one thread and allocated by another end up in the same cache line.
Object Size

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

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

Two threads causes on average 50% remote frees and running 40 threads causes on average 97.5% remote frees.

Figure 9a presents the total time each thread spends in the allocator for an increasing number of producers/consumers. Up to 30 threads scalloc and Streamflow provide the best temporal performance and for more than 30 threads scalloc outperforms all other allocators.

The average per-thread memory consumption illustrated in Figure 9b suggests that all allocators deal with blowup fragmentation, i.e., we do not observe unbounded growth in memory consumption. However, the absolute differences among different allocators are significant. Scalloc provides competitive spatial performance where only jemalloc and ptmalloc2 require less memory at the expense of higher total per-thread allocator time.

This experiment demonstrates that the approach of scalloc to distributing contention across spans with one remote free list per span works well in a producer-consumer workload and that using a lock-based implementation for reusing spans is not a performance bottleneck.

7.4 Robustness against False Sharing

False sharing occurs when objects that are allocated in the same cache line are read from and written to by different threads. In cache coherent systems this scenario can lead to performance degradation as all caches need to be kept consistent. An allocator is prone to active false sharing if objects that are allocated by different threads (without communication) end up in the same cache line. It is prone to passive false sharing if objects that are remotely allocated end up in the same cache line.
Memory Access

Figure 11: Memory access time for the locality experiment
1. Introduction

Like other allocators (e.g., [17] and [14]), scalloc can be divided into two main parts: (1) a mutator-facing frontend that manages memory in so-called spans, and (2) a backend for managing the spans (ideally returning them to the operating system when empty).

Scalloc maintains scalability with respect to performance and memory consumption by:

- introducing virtual spans that enable unified treatment of variable-size objects;
- providing a scalable backend for managing spans;
- providing a frontend with constant time malloc and free calls that only consider live heap (no garbage collection cycles).

The following subsections describe these crucial concepts of scalloc.

3.1 Real Spans and Size Classes

A (real) span is a contiguous portion of memory partitioned into blocks of the same size. The size of blocks in a span determines which size class the span belongs to. All spans in a given size class have the same number of blocks. Hence, the size of a span is fully determined by its size class: it is the product of the block size and the number of blocks, plus a span header containing administrative information. In scalloc, there are 29 size classes but only 9 distinct real-span sizes which are all multiples of 4KB (the size of a system page).

The first 16 size classes, with block sizes ranging from 16 bytes to 256 bytes in increments of 16 bytes, are taken from TCMalloc [6]. This design of small size-classes limits block internal fragmentation. All these 16 size classes have the same real-span size. Size classes with larger blocks range from 512 bytes to 1MB, in increments that are powers of two. These size classes may have different real-span size, explaining the difference between 29 size classes and 9 distinct real-span sizes.

Objects of size larger than any size class are not managed by spans, but rather allocated directly from the operating system using `mmap`.

3.2 Virtual Spans

A virtual span is a span allocated in a very large portion of virtual memory (32TB) which we call arena. All virtual spans have the same fixed size of 2MB and are 2MB-aligned in the arena. Each virtual span contains a real span, of one of the available size classes. By the size class of the virtual span we mean the size class of the contained real span. Typically, the real span is (much) smaller than the virtual span that contains it. The maximal real-span size is limited by the size of the virtual span. This is why virtual spans are suitable for big objects as well as for small ones. The structure of the

Figure 1: Structure of arena, virtual spans, and real spans
and the corresponding page frames can be unmapped. By madvise we always mean madvise with up to 256TB arena space and 2TB addressable physical memory with up to 48 bits for virtual addresses, this would enable support larger arenas in future work. On current hardware, and spatial performance for the benchmarks that were not the physical memory addressable with scalloc is virtual spans, if real spans are the smallest possible (16KB), bytes, i.e., 224TB.

The virtual address space still left is mapped I/O. The virtual address space still left is tem. It is still possible to allocate additional virtual memory overhead as the memory is not mapped by the operating system call on Linux. This call does not introduce any significant virtual memory addresses, the upper limit for a single mmap memory allocation. Upon initialization, scalloc mmaps a virtual span gets empty, it is inserted into the free-list section. The disadvantages of using virtual spans are:

- minimizes the chances that a virtual span changes its real span size.
- Note that since virtual spans are of the same size and of virtual spans, i.e., the span-pool discussed in the next section. The span-pool is a locally linearizable [Michael: TODO: size class to real span bin in figure 3.3 Backend: Span-Pool]

The span-pool is a global concurrent data structure that logically corresponds to a real-span-size.

Local treatment of small and big objects; lower latency of push and pop compared to enqueue and dequeue; and stacks rather than queues for the following reasons: spatial locality, especially on thread-local workloads; lower latency of arrays and stacks.

Figure 2: Span pool layout
Frontend: Eager Memory Reuse

---

Expected

<table>
<thead>
<tr>
<th>Arena (RSS = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backend (RSS compacted)</td>
</tr>
<tr>
<td>Frontend (RSS = real span size)</td>
</tr>
</tbody>
</table>

Expected

| malloc() |
| free() |

Figure 3: Life cycle of a span
Selfie is a self-referential 6k-line C implementation (in a single file) of:

1. a **self-compiling** compiler called *starc* 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 starc,

3. a **self-hosting** hypervisor called *hypster* that virtualizes mipster and can host all of selfie including itself, and

4. a tiny C* library called *libcstar* utilized by starc and mipster.
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;
}
MIPSter: 17 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);
Future Work with Selfie et al.

- I/O
- file systems
- memory allocation
- garbage collection
- concurrency: semantics
- parallelism: multicore
- volatility: persistent memory
Thank you!