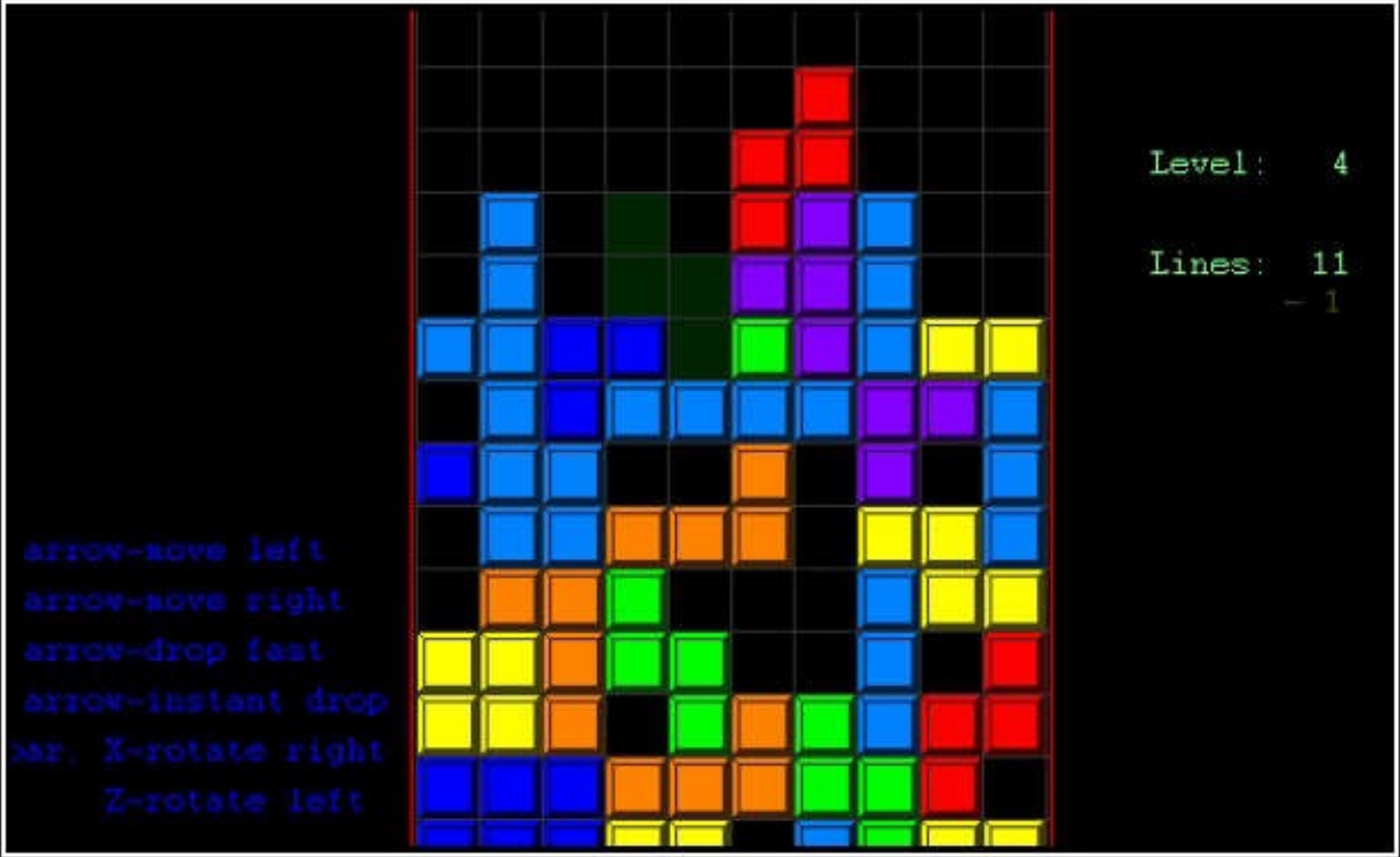
Memory by the Slab

The Tale of Jeff Bonwick's Slab Allocator

Ryan Zezeski // Sep 2015 // Papers We Love, NYC

STOP
MURANO



TETRIS AND LIFE

ERRORS PILE UP, ACCOMPLISHMENTS DISAPPEAR

Best Fit

Fastest

vs

General Allocator?

malloc(3C) & free(3C)

Have no a priori knowledge of size or lifetime.

General Allocators

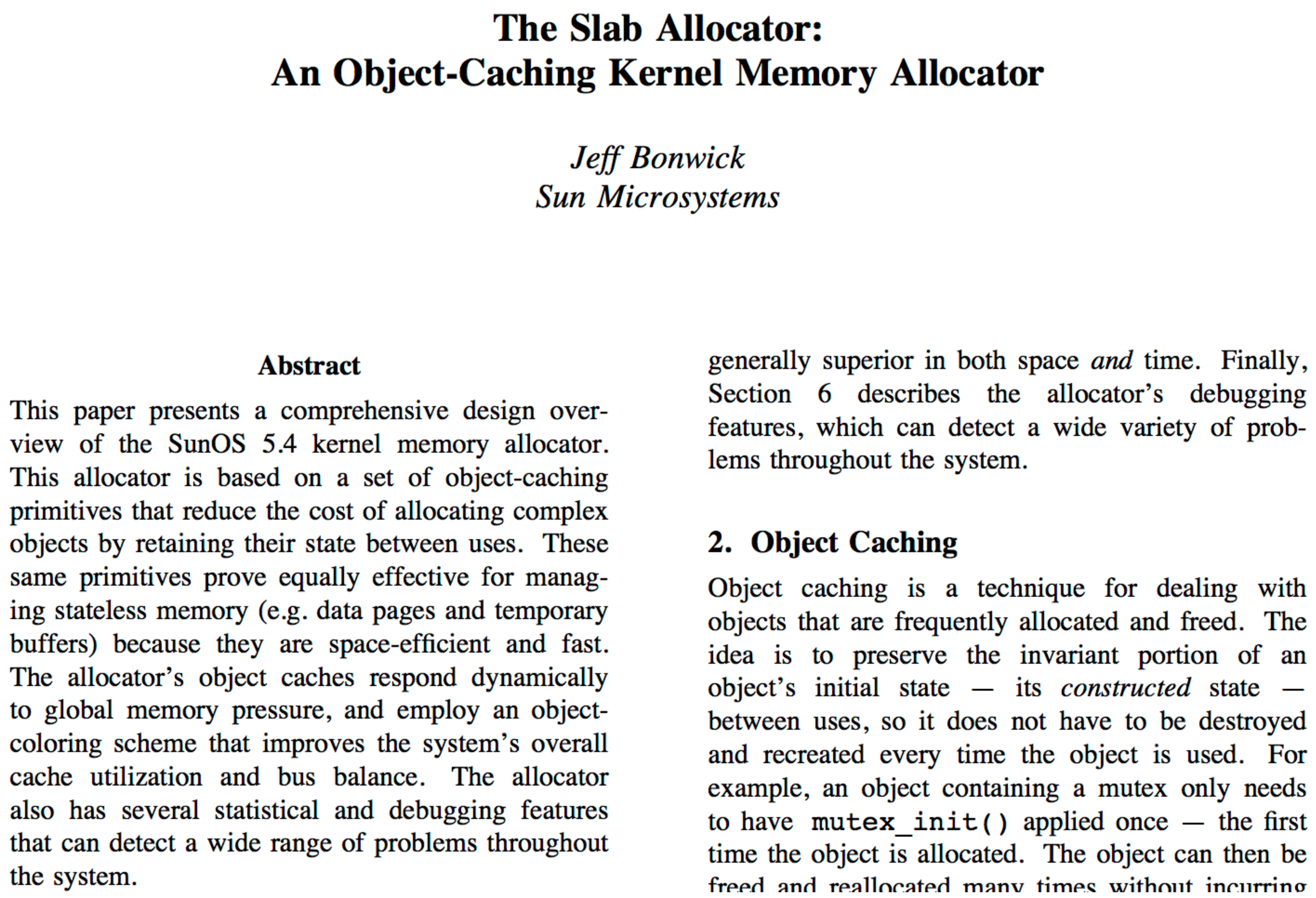
- dlmalloc — Doug Lea malloc
- ptmalloc — multithreaded dlmalloc (GNU libc?)
- TCmalloc — Google's Thread Caching malloc
- jemalloc — FreeBSD libc (not the kernel alloc, that's slab)

The Slab Allocator

- Created by Jeff Bonwick
- Solaris 2.4 (SunOS 5.4)
- USENIX 1994 (Same year as Speed. Shoot the hostage!)

"The slab allocator is operationally similar to the "CustoMalloc" [Grunwald93A], "QuickFit" [Weinstock88], and "Zone" [VanSciver88] allocators, all of which maintain distinct freelists of the most commonly requested buffer sizes. The Grunwald and Weinstock papers each demonstrate that a customized segregated-storage allocator—one that has ***a priori*** knowledge of the most common allocation sizes—is usually optimal in **both space and time.**"

–Bonwick94, Sec. 3.1, Par. 6



The Slab Allocator:
An Object-Caching Kernel Memory Allocator

Jeff Bonwick
Sun Microsystems

Abstract

This paper presents a comprehensive design overview of the SunOS 5.4 kernel memory allocator. This allocator is based on a set of object-caching primitives that reduce the cost of allocating complex objects by retaining their state between uses. These same primitives prove equally effective for managing stateless memory (e.g. data pages and temporary buffers) because they are space-efficient and fast. The allocator's object caches respond dynamically to global memory pressure, and employ an object-coloring scheme that improves the system's overall cache utilization and bus balance. The allocator also has several statistical and debugging features that can detect a wide range of problems throughout the system.

generally superior in both space *and* time. Finally, Section 6 describes the allocator's debugging features, which can detect a wide variety of problems throughout the system.

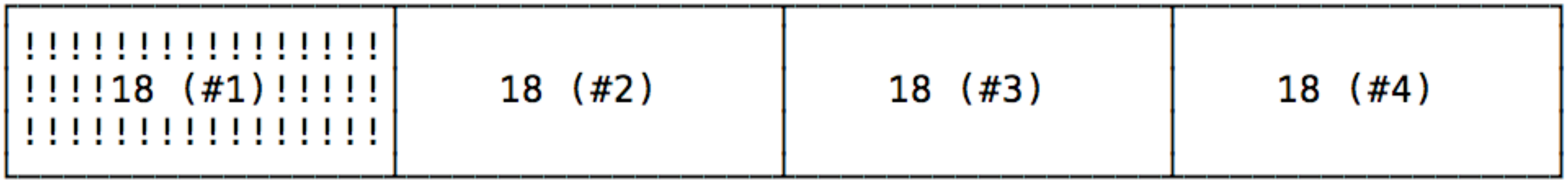
2. Object Caching

Object caching is a technique for dealing with objects that are frequently allocated and freed. The idea is to preserve the invariant portion of an object's initial state — its *constructed* state — between uses, so it does not have to be destroyed and recreated every time the object is used. For example, an object containing a mutex only needs to have `mutex_init()` applied once — the first time the object is allocated. The object can then be freed and reallocated many times without incurring

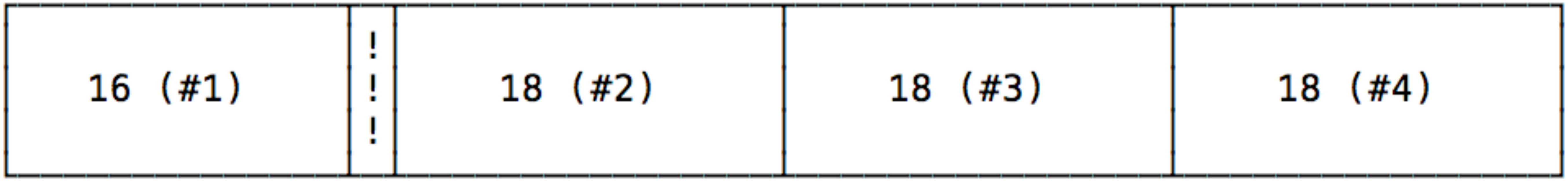
Object Caching

Your memory is no longer a blob. It has a name! It has a type!

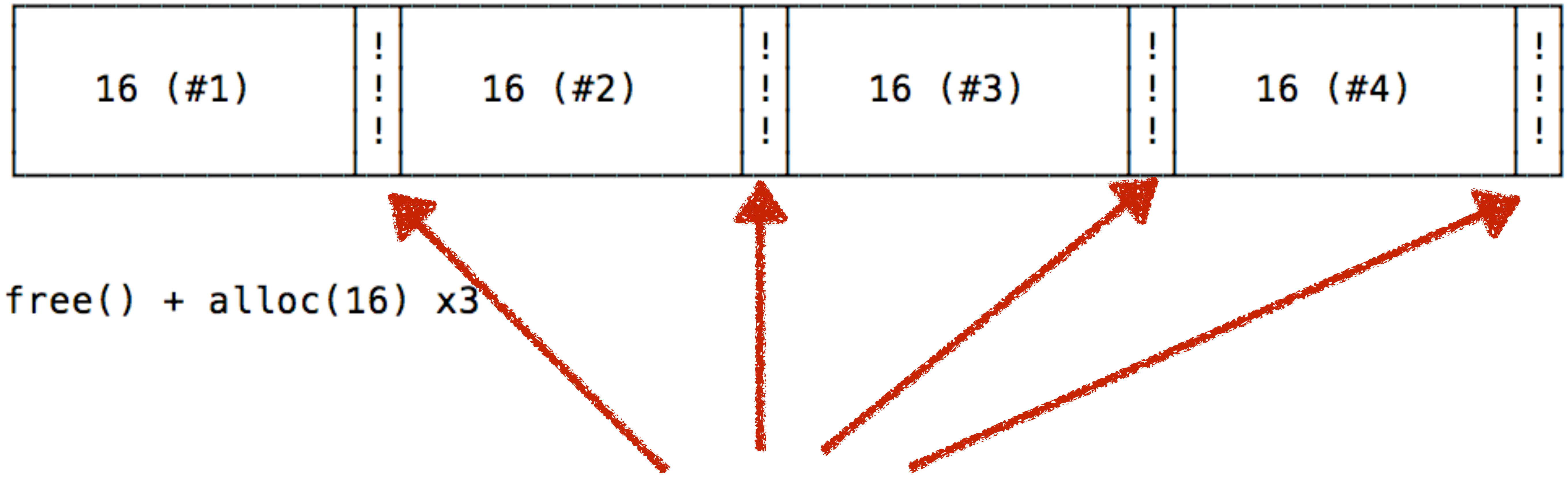
18 (#1)	18 (#2)	18 (#3)	18 (#4)



```
free(#1)
```

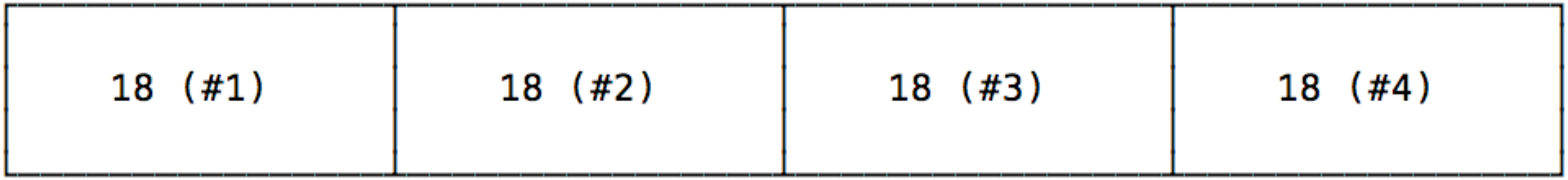


```
alloc(16)
```

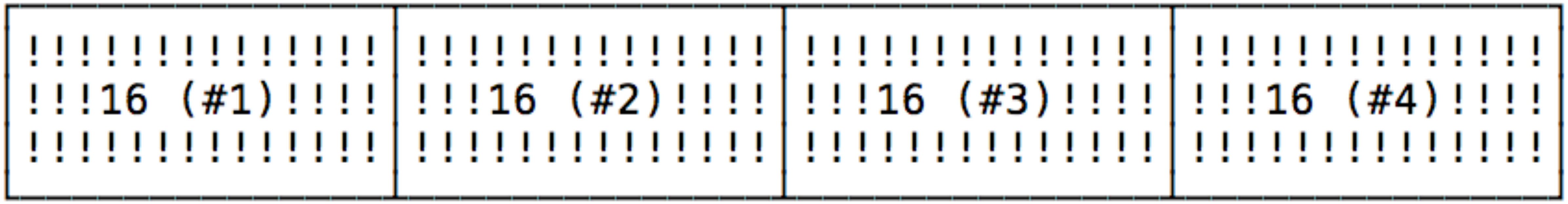


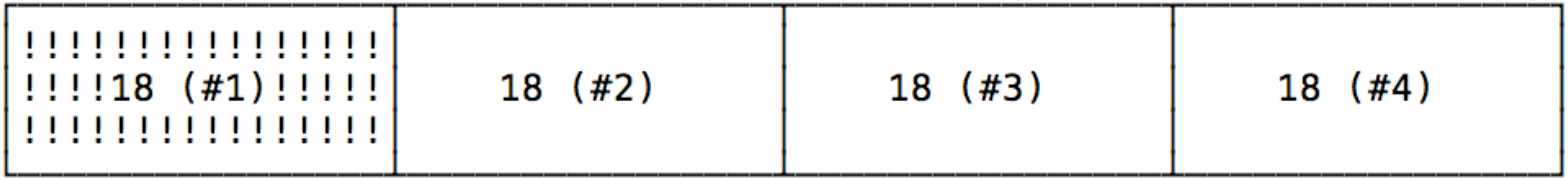
```
free() + alloc(16) x3
```

External Fragmentation

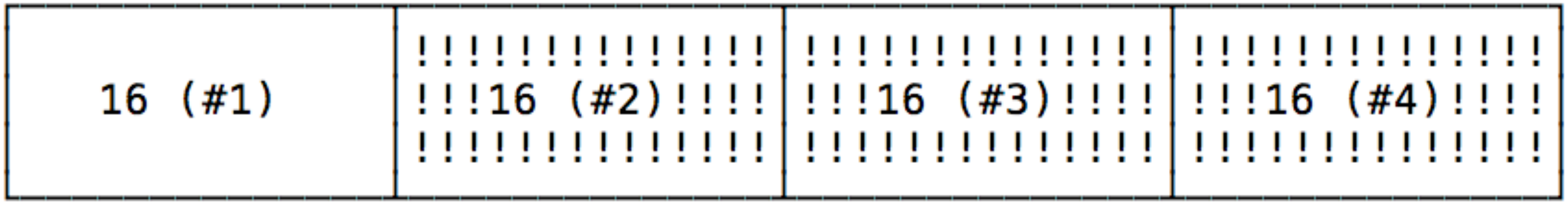


```
kmem_cache_alloc(obj18_cache) x4
```





```
kmem_cache_free(obj18_cache, #1);
kmem_cache_alloc(obj16_cache);
```



alloc(size_t)
TO
cache_alloc(cache_t *)

```
cache_t *
cache_create(
    char *name,
    size_t size,
    int align,
    void (*ctor)(void *, size_t),
    void (*dtor)(void *, size_t));

void
cache_destroy(cache_t *);
```

cache_alloc()

```
if (obj in slab?)
    return obj; /* ctor() not called */
else {
    claim free buffer in slab;
    obj = ctor(buf); /* create obj */
    return obj;
}
```

cache_free()

```
return obj to cache;
```

"The idea is to preserve the invariant portion of an object's initial state—its *constructed* state—between uses, so it does not have to be destroyed and recreated every time the object is used."

–Bonwick94, Sec. 2, Par. 1

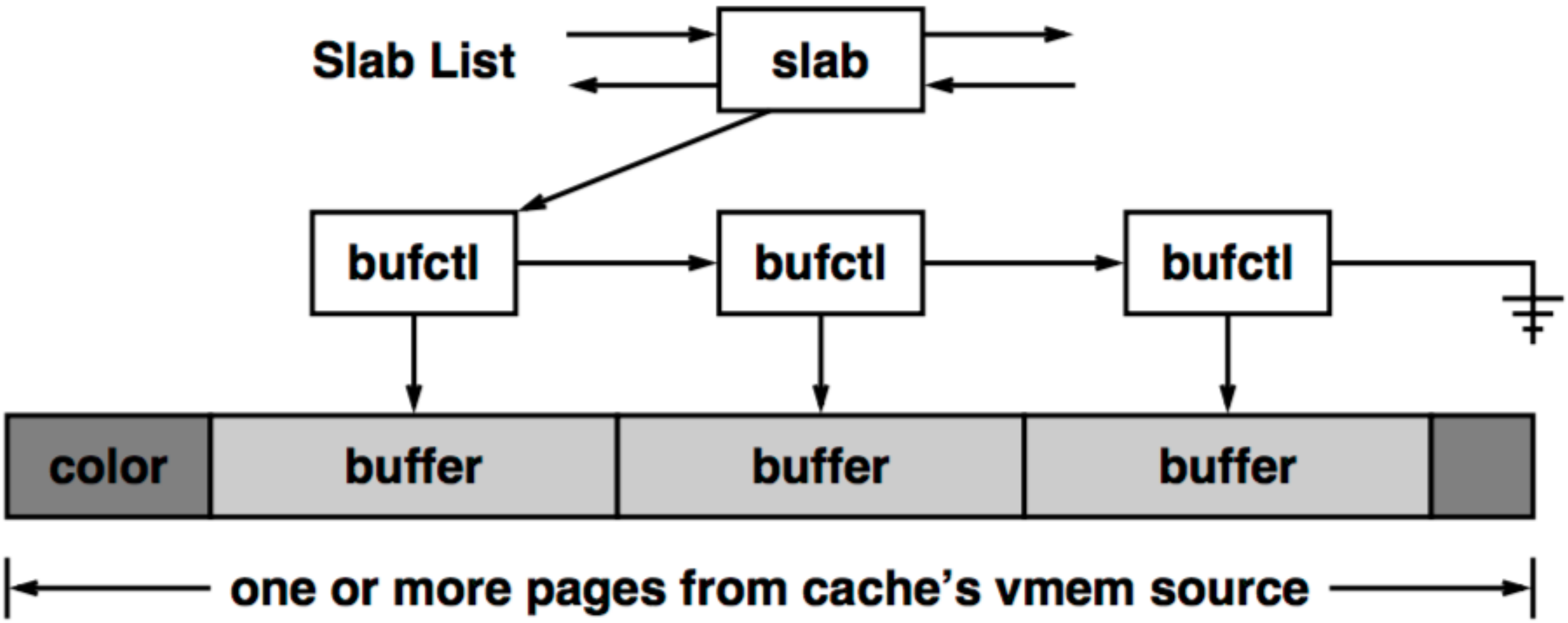
"Caching is important because the cost of constructing an object can be significantly higher than the cost of allocating memory for it."

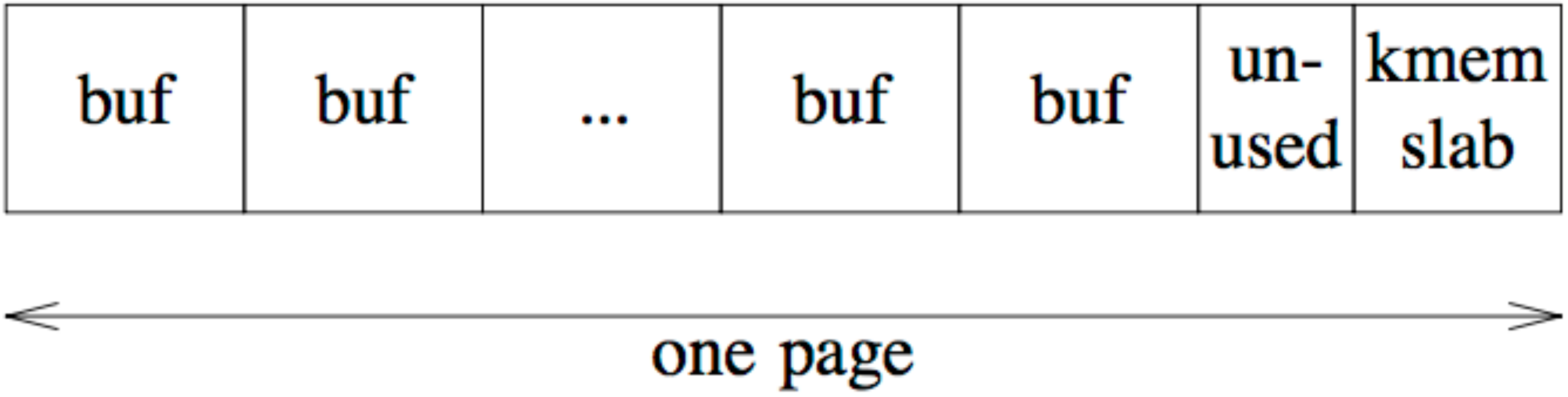
–Bonwick94, Sec. 2, Par. 2

general allocators can’t do this

they lack the API for it

The Slab


Slab List
slab
bufctl
bufctl
bufctl
color
buffer
buffer
buffer
one or more pages from cache's vmem source


buf
buf
...
buf
buf
un-used
kmem slab
one page

The Slab is the unit of currency.

1. Reclaim is trivial

"Thus a simple reference count replaces the complex trees, bitmaps, and coalescing algorithms found in most other allocators."

–Bonwick94, Sec. 3.2, Par. 3

2. Alloc and free are fast

“All we have to do is move an object to or from a freelist and update a reference count.”

–Bonwick94, Sec. 3.2, Par. 4

3. Severe external fragmentation unlikely

“A segregated-storage allocator cannot suffer this fate, since the only way to populate its 8-byte freelist is to actually allocate and free 8-byte buffers.”

–Bonwick94, Sec. 3.2, Par. 5

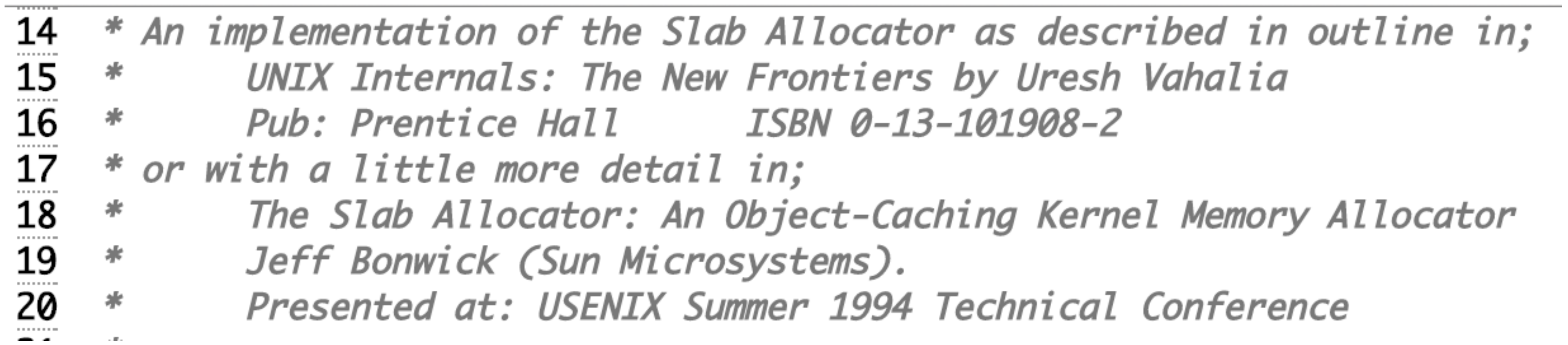
4. Internal fragmentation is minimal

"Each buffer is exactly the right size (namely, the cache's object size), so the only wasted space is the unused portion at the end of the slab…if a slab contains *n* buffers, then the internal fragmentation is at most 1/*n*; thus the allocator can actually control the amount of internal fragmentation by controlling slab size…The SunOS 5.4 implementation limits internal fragmentation to 12.5% (1/8)."

–Bonwick94, Sec. 3.2, Par. 5

Slab Accomplishments

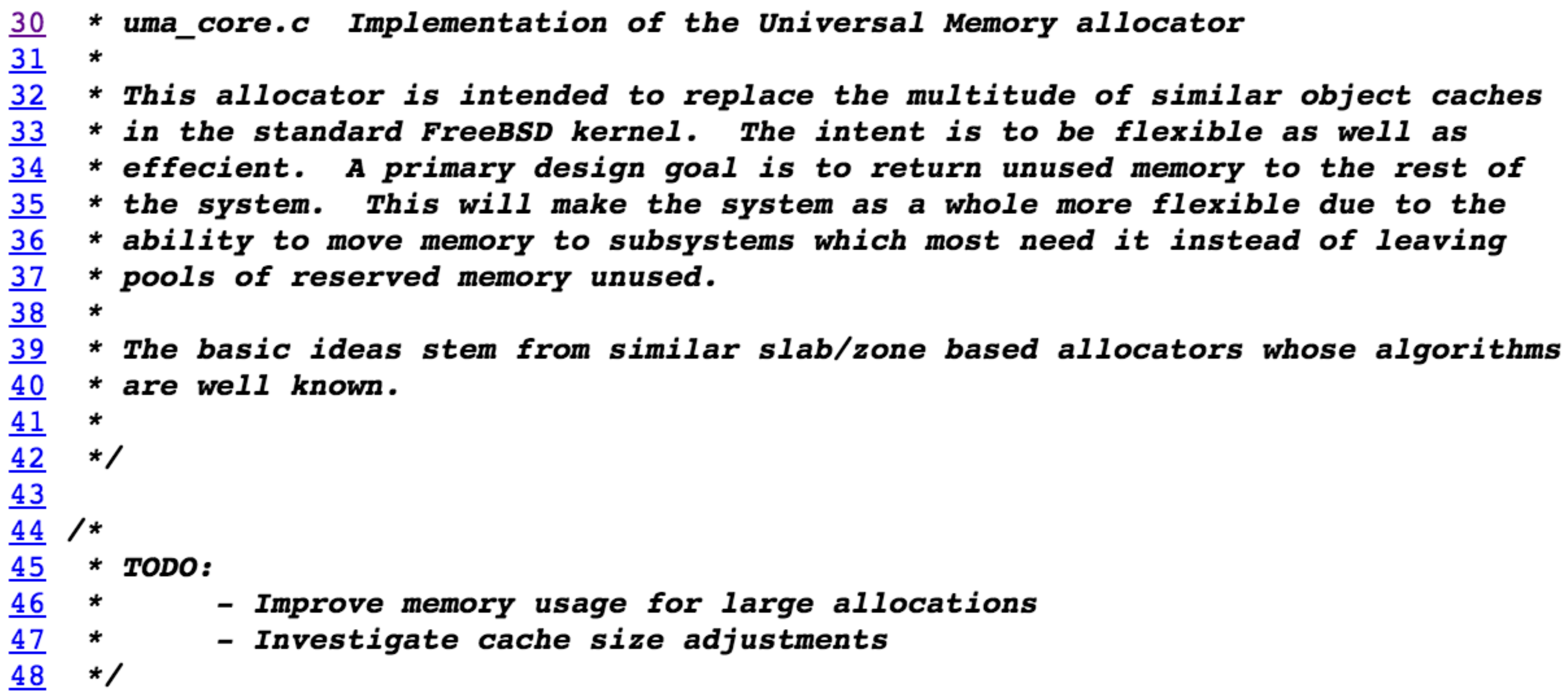
- Reduction of 3,000 LoC
- Simpler and More Understandable Design
- Much faster
- Less fragmentation: 46% to 14% (Solaris 2.4)
- Cache Friendly (very important in modern CPUs)
- Multi-core Friendly (2001 Additions)



```
 * An implementation of the Slab Allocator as described in outline in;
 *      UNIX Internals: The New Frontiers by Uresh Vahalia
 *      Pub: Prentice Hall      ISBN 0-13-101908-2
 * or with a little more detail in;
 *      The Slab Allocator: An Object-Caching Kernel Memory Allocator
 *      Jeff Bonwick (Sun Microsystems).
 *      Presented at: USENIX Summer 1994 Technical Conference
```

Linux 4.2

http://lxr.free-electrons.com/source/mm/slab.c?v=4.2#L14

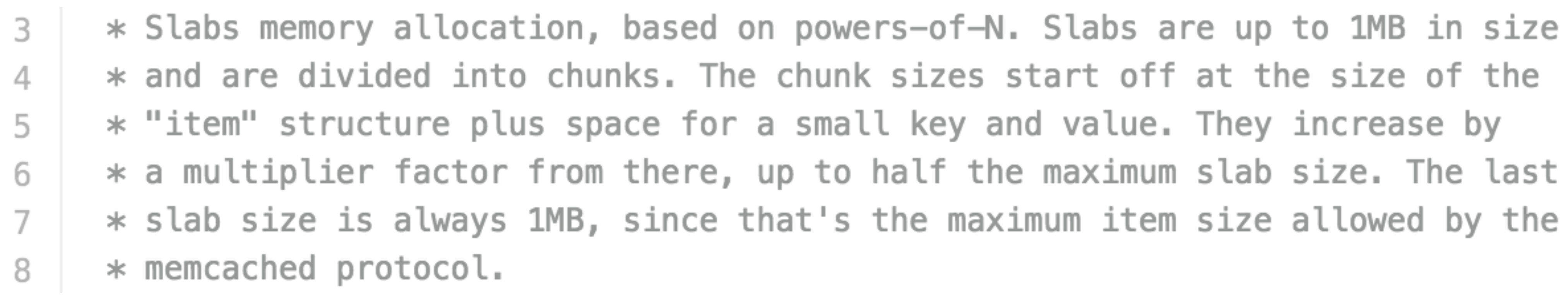


```
 * uma_core.c  Implementation of the Universal Memory allocator
 *
 * This allocator is intended to replace the multitude of similar object caches
 * in the standard FreeBSD kernel.  The intent is to be flexible as well as
 * effecient.  A primary design goal is to return unused memory to the rest of
 * the system.  This will make the system as a whole more flexible due to the
 * ability to move memory to subsystems which most need it instead of leaving
 * pools of reserved memory unused.
 *
 * The basic ideas stem from similar slab/zone based allocators whose algorithms
 * are well known.
 *
 */

/*
 * TODO:
 *      - Improve memory usage for large allocations
 *      - Investigate cache size adjustments
 */
```

FreeBSD 10

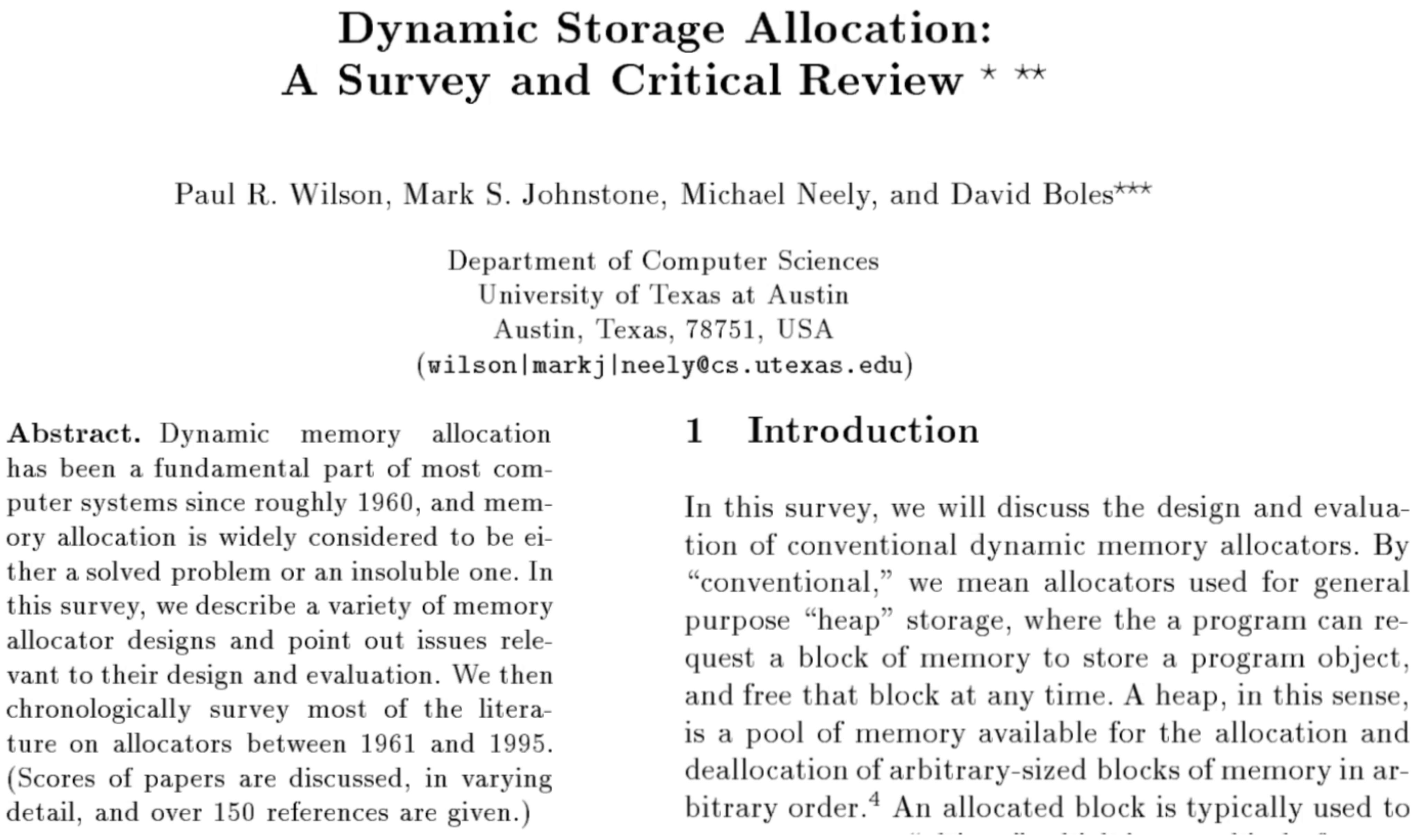
http://fxr.watson.org/fxr/source/vm/uma_core.c?v=FREEBSD10#L30



```
* Slabs memory allocation, based on powers-of-N. Slabs are up to 1MB in size
* and are divided into chunks. The chunk sizes start off at the size of the
* "item" structure plus space for a small key and value. They increase by
* a multiplier factor from there, up to half the maximum slab size. The last
* slab size is always 1MB, since that's the maximum item size allowed by the
* memcached protocol.
```

memcached?

https://github.com/memcached/memcached/blob/master/slabs.c



Dynamic Storage Allocation:
A Survey and Critical Review * **

Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles***

Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78751, USA
(wilson|markj|neely@cs.utexas.edu)

Abstract. Dynamic memory allocation has been a fundamental part of most computer systems since roughly 1960, and memory allocation is widely considered to be either a solved problem or an insoluble one. In this survey, we describe a variety of memory allocator designs and point out issues relevant to their design and evaluation. We then chronologically survey most of the literature on allocators between 1961 and 1995. (Scores of papers are discussed, in varying detail, and over 150 references are given.)

1 Introduction

In this survey, we will discuss the design and evaluation of conventional dynamic memory allocators. By "conventional," we mean allocators used for general purpose "heap" storage, where the a program can request a block of memory to store a program object, and free that block at any time. A heap, in this sense, is a pool of memory available for the allocation and deallocation of arbitrary-sized blocks of memory in arbitrary order.[4] An allocated block is typically used to

The Moby Dick of allocator papers

Published in 1995, one year *after* Bonwick's Slab Allocator.
But…not once is Bonwick or Slab mentioned.

"A major point of this section is that the mainstream of allocator research over the last several decades has focused on oversimplified (and unrealistic) models of program behavior, and that little is actually known about how to design allocators, or what performance to expect."

–Wilson95, p. 3

"Locality of reference is increasingly important, as the difference between CPU speed and main memory speeds has grown dramatically, with no sign of stopping."

–Wilson95, p. 2

“There are regularities in program behavior that allocators exploit, a point that is often insufficiently appreciated even by professionals who design and implement allocators. ”

–Wilson95, p. 6

Bonwick's Insights

- Object initialization is more expensive than allocation.
- Objects of the same type often have the same lifetime.
- Many mid-sized or larger objects receive a majority of their access on a minority of fields.
- The beginning of a data structure is typically more active than the end. Put important information at the end for maximizing chance of debugging.

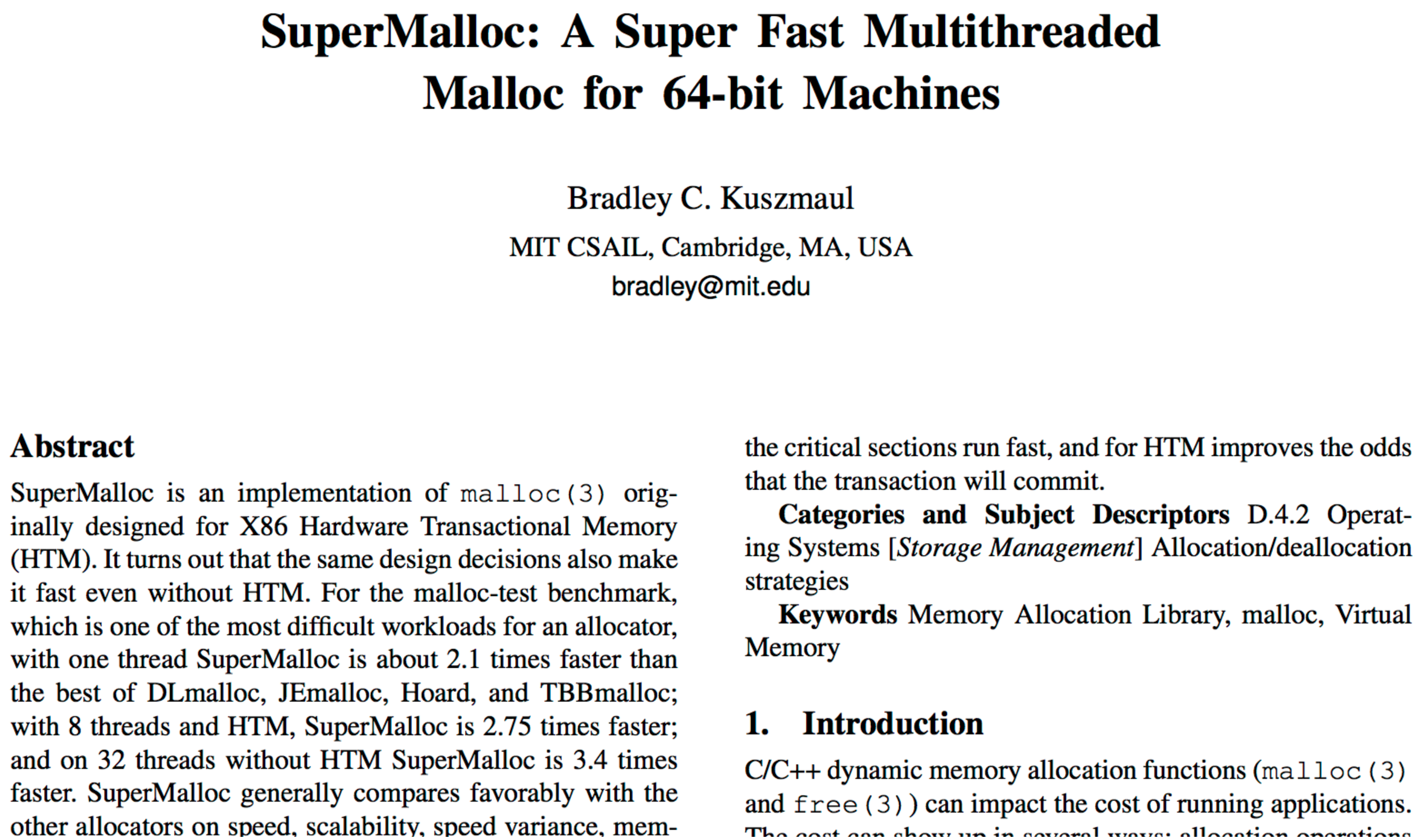
"Fragmentation is caused by isolated deaths…An allocator that can predict which objects will die at approximately the same time can exploit that information to reduce fragmentation, by placing those objects in contiguous memory."

–Wilson95, p. 15

"If the application requests the same size block soon after one is freed, the request can be satisfied by simply popping the pre-formatted block of a free list in a very small constant time."

–Wilson95, p. 23

Bonwick passes with flying colors



SuperMalloc: A Super Fast Multithreaded Malloc for 64-bit Machines

Bradley C. Kuszmaul
MIT CSAIL, Cambridge, MA, USA
bradley@mit.edu

Abstract

SuperMalloc is an implementation of `malloc(3)` originally designed for X86 Hardware Transactional Memory (HTM). It turns out that the same design decisions also make it fast even without HTM. For the malloc-test benchmark, which is one of the most difficult workloads for an allocator, with one thread SuperMalloc is about 2.1 times faster than the best of DLmalloc, JEmalloc, Hoard, and TBBmalloc; with 8 threads and HTM, SuperMalloc is 2.75 times faster; and on 32 threads without HTM SuperMalloc is 3.4 times faster. SuperMalloc generally compares favorably with the other allocators on speed, scalability, speed variance, mem-

the critical sections run fast, and for HTM improves the odds that the transaction will commit.

Categories and Subject Descriptors D.4.2 Operating Systems [*Storage Management*] Allocation/deallocation strategies

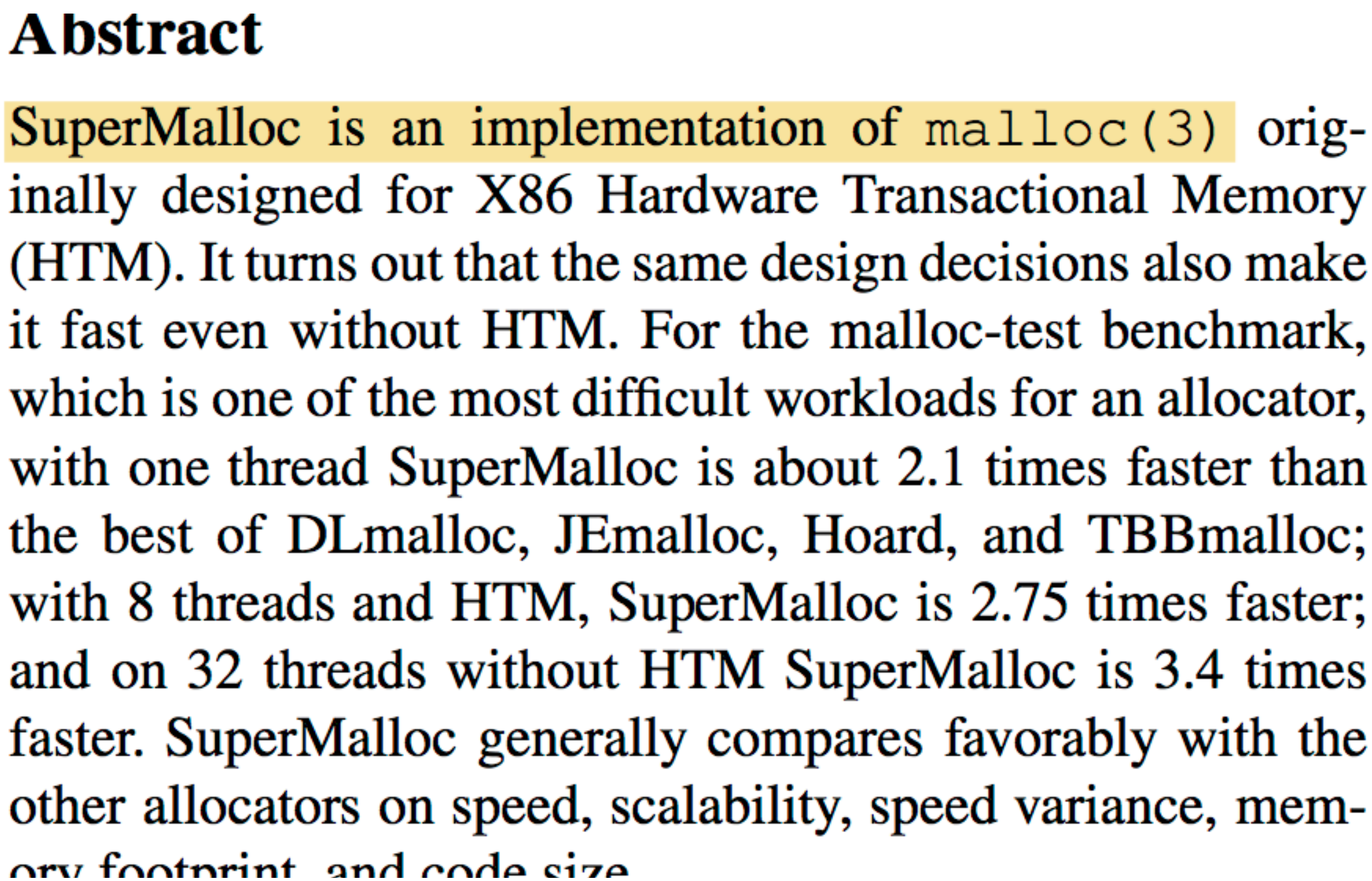
Keywords Memory Allocation Library, malloc, Virtual Memory

1. Introduction

C/C++ dynamic memory allocation functions (`malloc(3)` and `free(3)`) can impact the cost of running applications.

Super Malloc

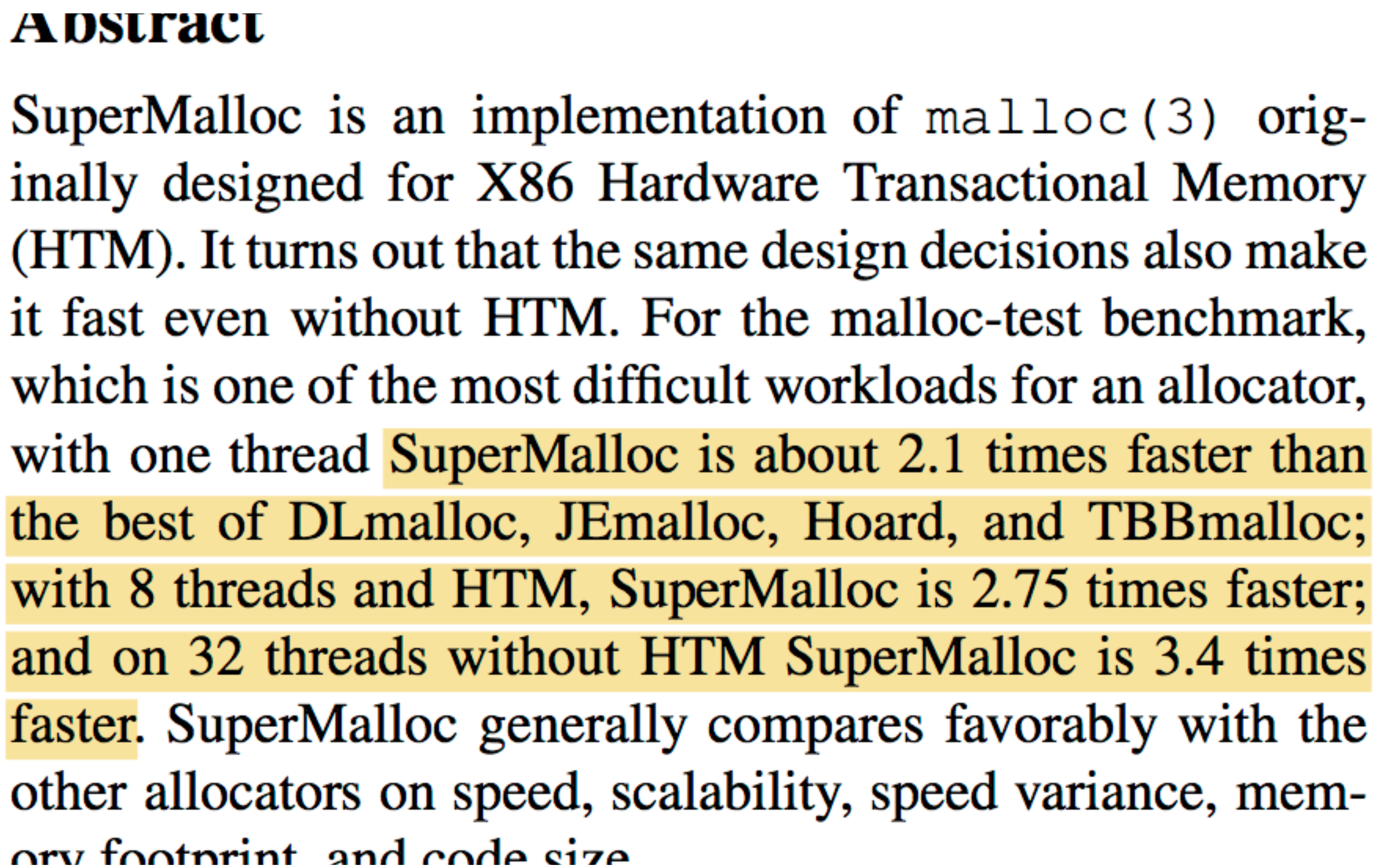
A general allocator designed in 2015, 21 years after Bonwick's Slab Allocator.



Abstract

SuperMalloc is an implementation of `malloc(3)` originally designed for X86 Hardware Transactional Memory (HTM). It turns out that the same design decisions also make it fast even without HTM. For the malloc-test benchmark, which is one of the most difficult workloads for an allocator, with one thread SuperMalloc is about 2.1 times faster than the best of DLmalloc, JEmalloc, Hoard, and TBBmalloc; with 8 threads and HTM, SuperMalloc is 2.75 times faster; and on 32 threads without HTM SuperMalloc is 3.4 times faster. SuperMalloc generally compares favorably with the other allocators on speed, scalability, speed variance, memory footprint, and code size.

It's a general allocator.

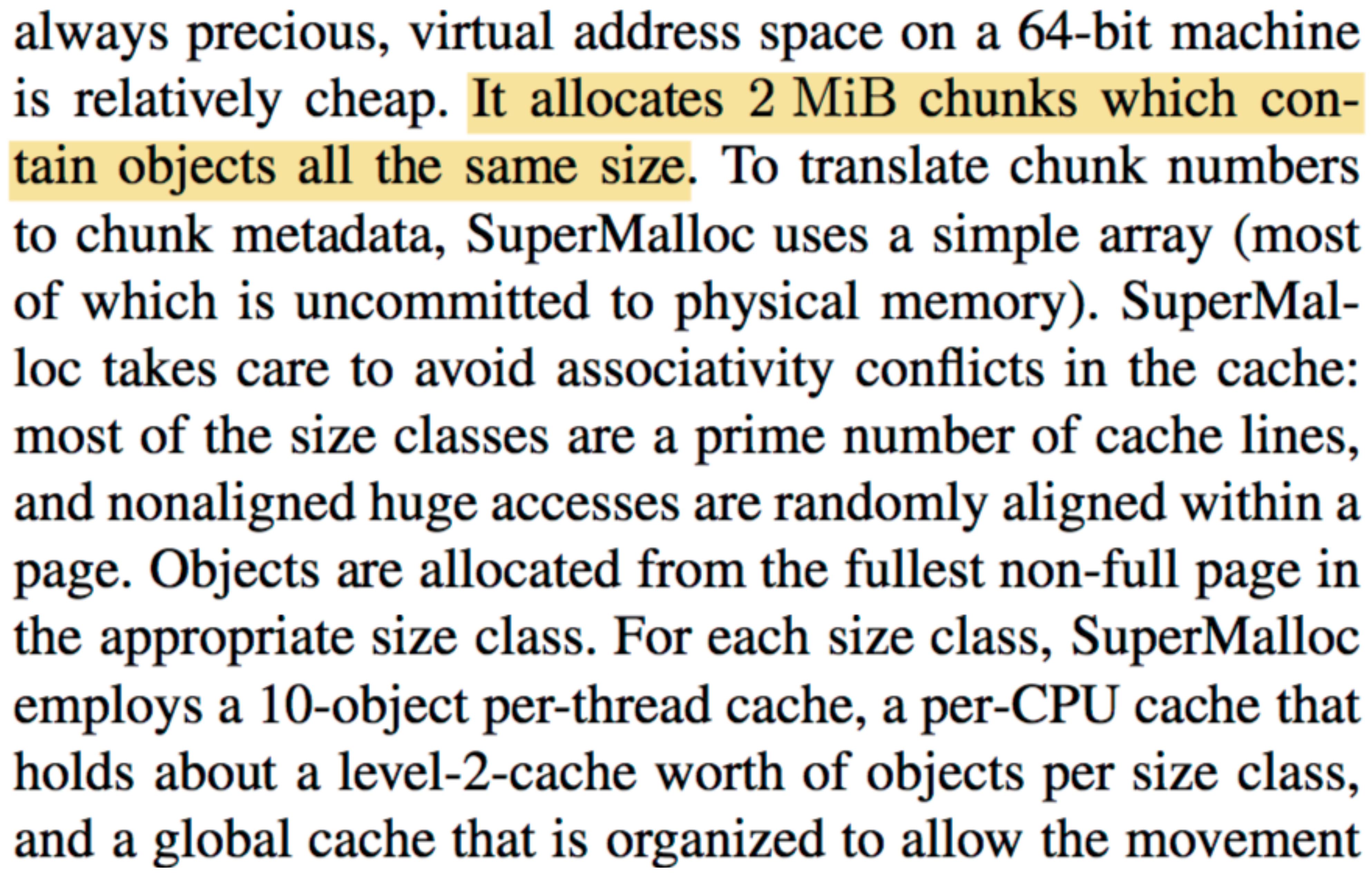


Abstract

SuperMalloc is an implementation of `malloc(3)` originally designed for X86 Hardware Transactional Memory (HTM). It turns out that the same design decisions also make it fast even without HTM. For the malloc-test benchmark, which is one of the most difficult workloads for an allocator, with one thread SuperMalloc is about 2.1 times faster than the best of DLmalloc, JEmalloc, Hoard, and TBBmalloc; with 8 threads and HTM, SuperMalloc is 2.75 times faster; and on 32 threads without HTM SuperMalloc is 3.4 times faster. SuperMalloc generally compares favorably with the other allocators on speed, scalability, speed variance, memory footprint, and code size

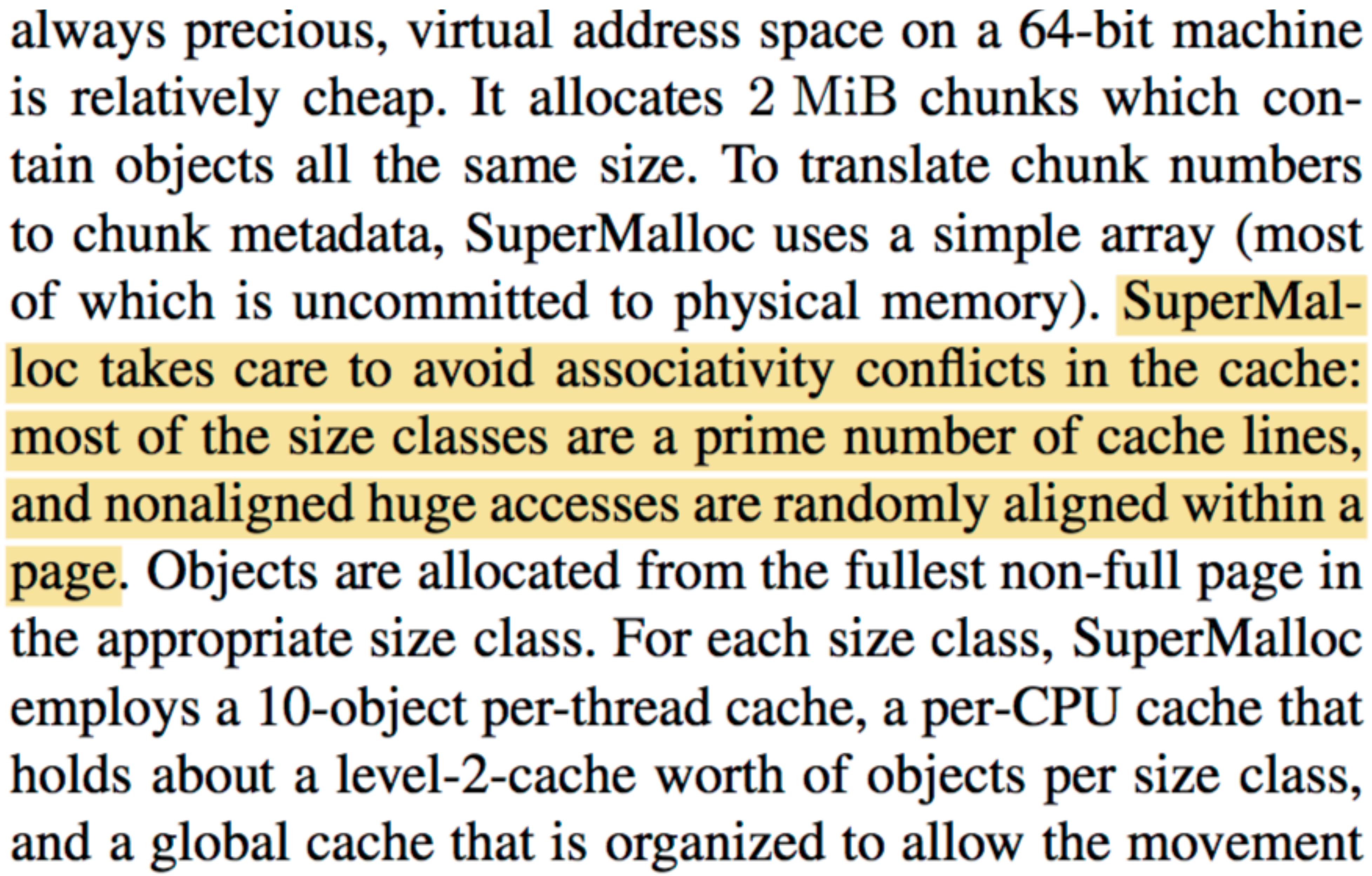
Faster than popular allocators.

Especially as thread count grows.



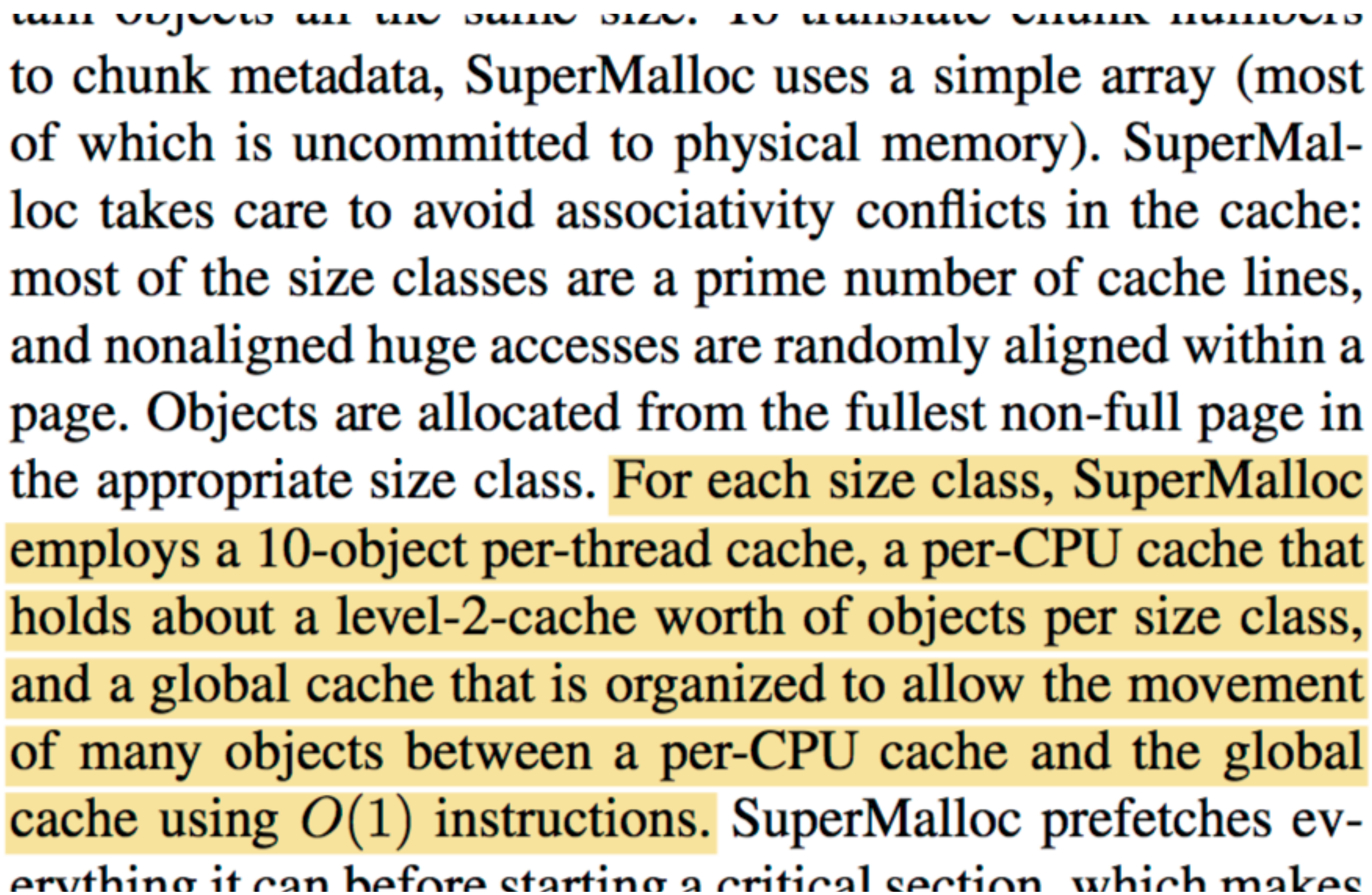
always precious, virtual address space on a 64-bit machine
is relatively cheap. It allocates 2 MiB chunks which con-
tain objects all the same size. To translate chunk numbers
to chunk metadata, SuperMalloc uses a simple array (most
of which is uncommitted to physical memory). SuperMal-
loc takes care to avoid associativity conflicts in the cache:
most of the size classes are a prime number of cache lines,
and nonaligned huge accesses are randomly aligned within a
page. Objects are allocated from the fullest non-full page in
the appropriate size class. For each size class, SuperMalloc
employs a 10-object per-thread cache, a per-CPU cache that
holds about a level-2-cache worth of objects per size class,
and a global cache that is organized to allow the movement

So…basically a slab.



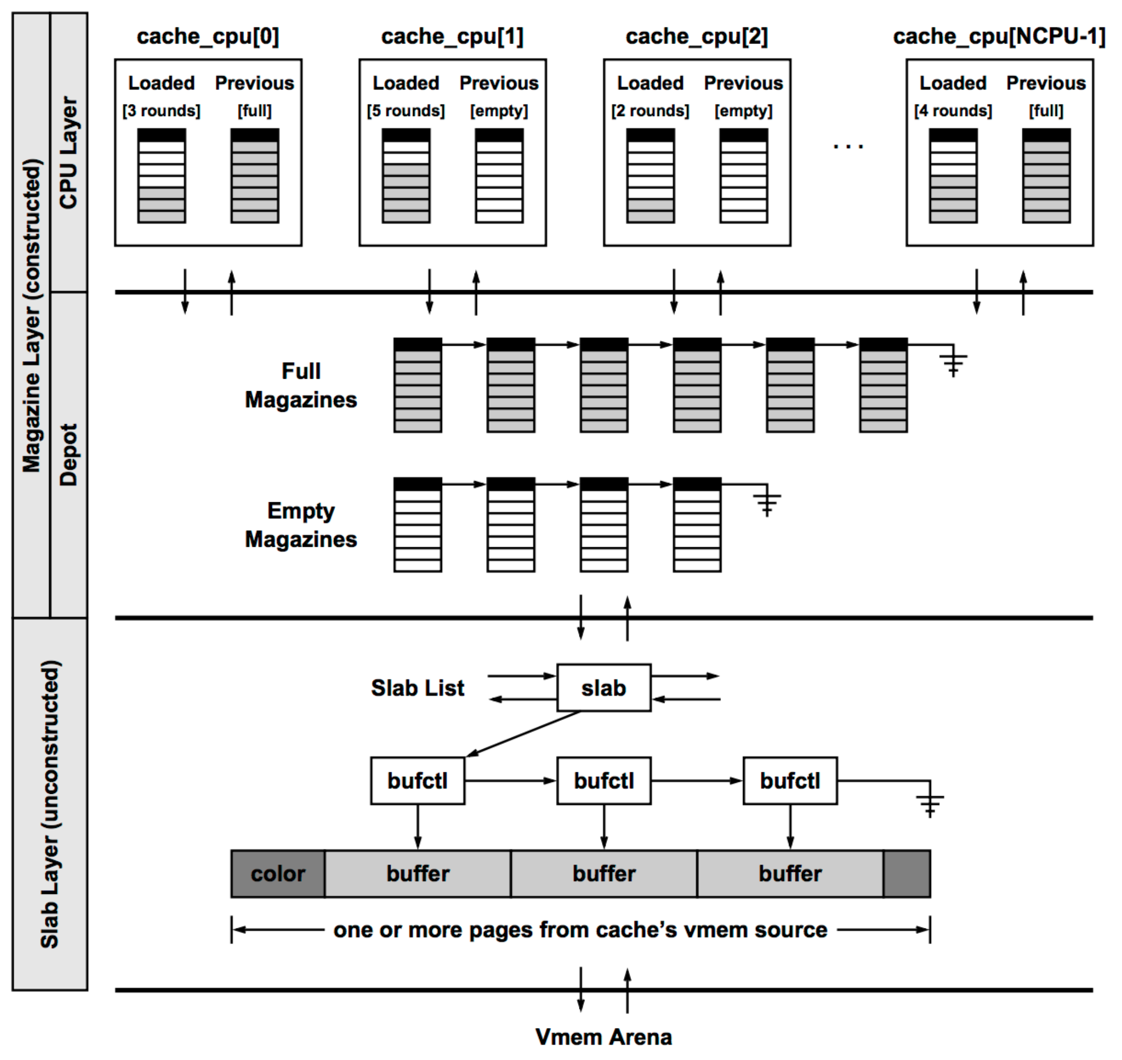
always precious, virtual address space on a 64-bit machine is relatively cheap. It allocates 2 MiB chunks which contain objects all the same size. To translate chunk numbers to chunk metadata, SuperMalloc uses a simple array (most of which is uncommitted to physical memory). SuperMalloc takes care to avoid associativity conflicts in the cache: most of the size classes are a prime number of cache lines, and nonaligned huge accesses are randomly aligned within a page. Objects are allocated from the fullest non-full page in the appropriate size class. For each size class, SuperMalloc employs a 10-object per-thread cache, a per-CPU cache that holds about a level-2-cache worth of objects per size class, and a global cache that is organized to allow the movement

So…slab coloring


tain objects all the same size. To translate chunk numbers to chunk metadata, SuperMalloc uses a simple array (most of which is uncommitted to physical memory). SuperMalloc takes care to avoid associativity conflicts in the cache: most of the size classes are a prime number of cache lines, and nonaligned huge accesses are randomly aligned within a page. Objects are allocated from the fullest non-full page in the appropriate size class. For each size class, SuperMalloc employs a 10-object per-thread cache, a per-CPU cache that holds about a level-2-cache worth of objects per size class, and a global cache that is organized to allow the movement of many objects between a per-CPU cache and the global cache using $O(1)$ instructions. SuperMalloc prefetches everything it can before starting a critical section, which makes

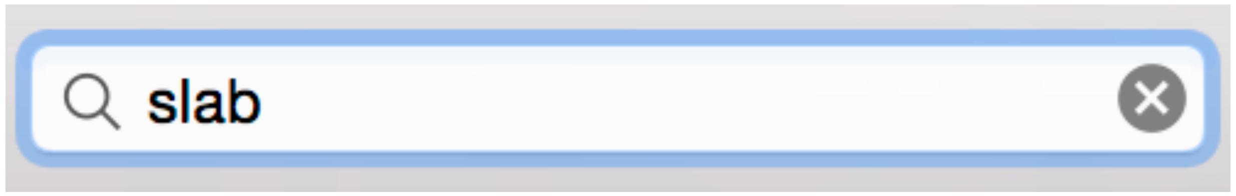
This sounds familiar…

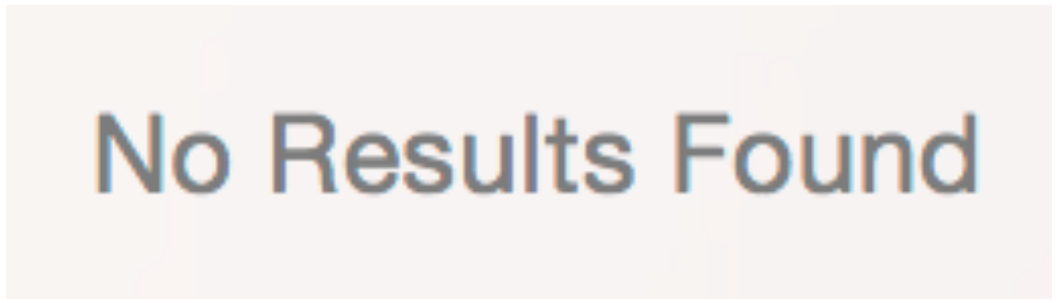
Figure 3: Structure of an Object Cache – The Magazine and Slab Layers



Slab Magazine Layer

(USENIX 2001)




No Results Found

WAT?

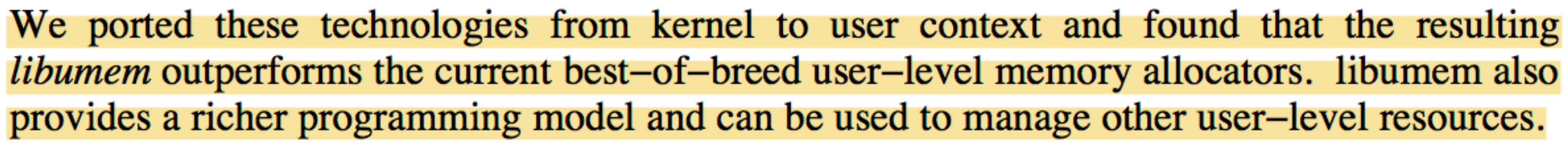
Seriously…really?

This is actually kind of cool. Independent discovery of the same ideas.

Where do we go from here?

“The slab allocator could also be used as a user-level memory allocator. The back-end page supplier could be mmap(2) or sbrk(2).”

–Bonwick94, Sec. 7.3


We ported these technologies from kernel to user context and found that the resulting *libumem* outperforms the current best–of–breed user–level memory allocators. libumem also provides a richer programming model and can be used to manage other user–level resources.

libumem

Bonwick & Adams, USENIX 2001

time for POSIX slab:
cache_alloc(3C)
cache_free(3C)

What have we learned?

Thank You

ryan@zinascii.com

References

[**Bonwick94**] Jeff Bonwick. *The Slab Allocator: An Object-Caching Kernel Memory Allocator.* USENIX Summer Technical Conference, 1994.

[**Wilson95**] Paul R. Wilson et al. *Dynamic Storage Allocation: A Survey and Critical Review.* International Workshop on Memory Management, 1995.

[**Bonwick01**] Jeff Bonwick & Jonathan Adams. *Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources*. USENIX Annual Technical Conference, 2001.

[**Kuszmaul15**] Bradley C. Kuszmaul. *SuperMalloc: a super fast multithreaded malloc for 64-bit machines.* International Symposium on Memory Managment, 2015.

Further Reading

[**Ross67**] Douglas T. Ross. *The AED free storage package.* CACM Vol. 10 Issue 8, 1967.

[**Korn85**] D. G. Korn & K. P Vo. *In Search of a Better Malloc.* USENIX Summer Conference, 1985.

[**Lea00**] Doug Lea. *A Memory Allocator.* 2000.

[**Gorman07**] Mel Gorman. *Understanding The Linux Virtual Memory Manager.* 2007.

[**Stone12**] Adrian Stone. *The Hole That dlmalloc Can't Fill.* Game Angst, 2012.