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.

1. Introduction

The allocation and freeing of objects are among the most common operations in the kernel. A fast kernel memory allocator is therefore essential. However, in many cases the cost of initializing and destroying the object exceeds the cost of allocating and freeing memory for it. Thus, while improvements in the allocator are beneficial, even greater gains can be achieved by caching frequently used objects so that their basic structure is preserved between uses.

The paper begins with a discussion of object caching, since the interface that this requires will shape the rest of the allocator. The next section then describes the implementation in detail. Section 4 describes the effect of buffer address distribution on the system's overall cache utilization and bus balance, and shows how a simple coloring scheme can improve both. Section 5 compares the allocator's performance to several other well-known kernel memory allocators and finds that it is 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 the expense of `mutex_destroy()` and `mutex_init()` each time. An object's embedded locks, condition variables, reference counts, lists of other objects, and read-only data all generally qualify as constructed state.

Caching is important because the cost of constructing an object can be significantly higher than the cost of allocating memory for it. For example, on a SPARCstation-2 running a SunOS 5.4 development kernel, the allocator presented here reduced the cost of allocating and freeing a stream head from 33 microseconds to 5.7 microseconds. As the table below illustrates, most of the savings was due to object caching:

Stream Head Allocation + Free Costs (μsec)			
allocator	construction + destruction	memory allocation	other init.
old	23.6	9.4	1.9
new	0.0	3.8	1.9

Caching is particularly beneficial in a multithreaded environment, where many of the most frequently allocated objects contain one or more embedded locks, condition variables, and other constructible state.

The design of an object cache is straightforward:

To allocate an object:

```
if (there's an object in the cache)
    take it (no construction required);
else {
    allocate memory;
    construct the object;
}
```

To free an object:

```
return it to the cache (no destruction required);
```

To reclaim memory from the cache:

```
take some objects from the cache;
destroy the objects;
free the underlying memory;
```

An object's constructed state must be initialized only once — when the object is first brought into the cache. Once the cache is populated, allocating and freeing objects are fast, trivial operations.

2.1. An Example

Consider the following data structure:

```
struct foo {
    kmutex_t   foo_lock;
    kcondvar_t foo_cv;
    struct bar *foo_barlist;
    int        foo_refcnt;
};
```

Assume that a `foo` structure cannot be freed until there are no outstanding references to it (`foo_refcnt == 0`) and all of its pending `bar` events (whatever they are) have completed (`foo_barlist == NULL`). The life cycle of a dynamically allocated foo would be something like this:

```
foo = kmem_alloc(sizeof (struct foo),
    KM_SLEEP);
mutex_init(&foo->foo_lock, ...);
cv_init(&foo->foo_cv, ...);
foo->foo_refcnt = 0;
foo->foo_barlist = NULL;

use foo;

ASSERT(foo->foo_barlist == NULL);
ASSERT(foo->foo_refcnt == 0);
cv_destroy(&foo->foo_cv);
mutex_destroy(&foo->foo_lock);
kmem_free(foo);
```

Notice that between each use of a foo object we perform a sequence of operations that constitutes nothing more than a very expensive no-op. All of this overhead (i.e., everything other than ''use foo'' above) can be eliminated by object caching.

2.2. The Case for Object Caching in the Central Allocator

Of course, object caching can be implemented without any help from the central allocator — any subsystem can have a private implementation of the algorithm described above. However, there are several disadvantages to this approach:

(1) There is a natural tension between an object cache, which wants to keep memory, and the rest of the system, which wants that memory back. Privately-managed caches cannot handle this tension sensibly. They have limited insight into the system's overall memory needs and *no* insight into each other's needs. Similarly, the rest of the system has no knowledge of the existence of these caches and hence has no way to ''pull'' memory from them.

(2) Since private caches bypass the central allocator, they also bypass any accounting mechanisms and debugging features that allocator may possess. This makes the operating system more difficult to monitor and debug.

(3) Having many instances of the same solution to a common problem increases kernel code size and maintenance costs.

Object caching requires a greater degree of cooperation between the allocator and its clients than the standard `kmem_alloc(9F)/kmem_free(9F)` interface allows. The next section develops an interface to support constructed object caching in the central allocator.

2.3. Object Cache Interface

The interface presented here follows from two observations:

(A) Descriptions of objects (name, size, alignment, constructor, and destructor) belong in the clients — not in the central allocator. The allocator should not just ‘‘know’’ that `sizeof (struct inode)` is a useful pool size, for example. Such assumptions are brittle [Grunwald93A] and cannot anticipate the needs of third-party device drivers, streams modules and file systems.

(B) Memory management policies belong in the central allocator — not in its clients. The clients just want to allocate and free objects quickly. They shouldn’t have to worry about how to manage the underlying memory efficiently.

It follows from (A) that object cache creation must be client-driven and must include a full specification of the objects:

(1)
```
struct kmem_cache *kmem_cache_create(
    char *name,
    size_t size,
    int align,
    void (*constructor)(void *, size_t),
    void (*destructor)(void *, size_t));
```

Creates a cache of objects, each of size `size`, aligned on an `align` boundary. The alignment will always be rounded up to the minimum allowable value, so `align` can be zero whenever no special alignment is required. `name` identifies the cache for statistics and debugging. `constructor` is a function that constructs (that is, performs the one-time initialization of) objects in the cache; `destructor` undoes this, if applicable. The constructor and destructor take a `size` argument so that they can support families of similar caches, e.g. streams messages. `kmem_cache_create` returns an opaque descriptor for accessing the cache.

Next, it follows from (B) that clients should need just two simple functions to allocate and free objects:

(2)
```
void *kmem_cache_alloc(
    struct kmem_cache *cp,
    int flags);
```

Gets an object from the cache. The object will be in its constructed state. `flags` is either `KM_SLEEP` or `KM_NOSLEEP`, indicating whether it’s acceptable to wait for memory if none is currently available.

(3)
```
void kmem_cache_free(
    struct kmem_cache *cp,
    void *buf);
```

Returns an object to the cache. The object must still be in its constructed state.

Finally, if a cache is no longer needed the client can destroy it:

(4)
```
void kmem_cache_destroy(
    struct kmem_cache *cp);
```

Destroys the cache and reclaims all associated resources. All allocated objects must have been returned to the cache.

This interface allows us to build a flexible allocator that is ideally suited to the needs of its clients. In this sense it is a ‘‘custom’’ allocator. However, it does not have to be built with compile-time knowledge of its clients as most custom allocators do [Bozman84A, Grunwald93A, Margolin71], nor does it have to keep guessing as in the adaptive-fit methods [Bozman84B, Leverett82, Oldehoeft85]. Rather, the object-cache interface allows clients to specify the allocation services they need on the fly.

2.4. An Example

This example demonstrates the use of object caching for the ‘‘foo’’ objects introduced in Section 2.1. The constructor and destructor routines are:

```
void
foo_constructor(void *buf, int size)
{
    struct foo *foo = buf;

    mutex_init(&foo->foo_lock, ...);
    cv_init(&foo->foo_cv, ...);
    foo->foo_refcnt = 0;
    foo->foo_barlist = NULL;
}

void
foo_destructor(void *buf, int size)
{
    struct foo *foo = buf;

    ASSERT(foo->foo_barlist == NULL);
    ASSERT(foo->foo_refcnt == 0);
    cv_destroy(&foo->foo_cv);
    mutex_destroy(&foo->foo_lock);
}
```

To create the foo cache:

```
foo_cache = kmem_cache_create("foo_cache",
    sizeof (struct foo), 0,
    foo_constructor, foo_destructor);
```

To allocate, use, and free a foo object:

```
foo = kmem_cache_alloc(foo_cache, KM_SLEEP);
use foo;
kmem_cache_free(foo_cache, foo);
```

This makes foo allocation fast, because the allocator will usually do nothing more than fetch an already-constructed foo from the cache. `foo_constructor` and `foo_destructor` will be invoked only to populate and drain the cache, respectively.

The example above illustrates a beneficial side-effect of object caching: it reduces the instruction-cache footprint of the code that *uses* cached objects by moving the rarely-executed construction and destruction code out of the hot path.

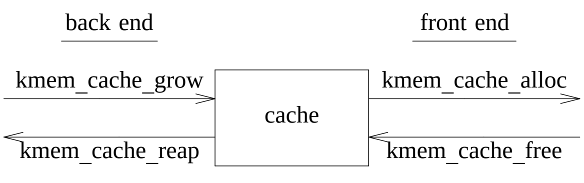
3. Slab Allocator Implementation

This section describes the implementation of the SunOS 5.4 kernel memory allocator, or ‘‘slab allocator,’’ in detail. (The name derives from one of the allocator’s main data structures, the *slab*. The name stuck within Sun because it was more distinctive than ‘‘object’’ or ‘‘cache.’’ Slabs will be discussed in Section 3.2.)

The terms *object*, *buffer*, and *chunk* will be used more or less interchangeably, depending on how we’re viewing that piece of memory at the moment.

3.1. Caches

Each cache has a *front end* and *back end* which are designed to be as decoupled as possible:



The front end is the public interface to the allocator. It moves objects to and from the cache, calling into the back end when it needs more objects.

The back end manages the flow of real memory through the cache. The influx routine (`kmem_cache_grow()`) gets memory from the VM system, makes objects out of it, and feeds those objects into the cache. The outflux routine (`kmem_cache_reap()`) is invoked by the VM system when it wants some of that memory back — e.g., at the onset of paging. Note that all back-end activity is triggered solely by memory pressure. Memory flows in when the cache needs more objects and flows back out when the rest of the system needs more pages; there are no arbitrary limits or watermarks. Hysteresis control is provided by a working-set algorithm, described in Section 3.4.

The slab allocator is not a monolithic entity, but rather is a loose confederation of independent caches. The caches have no shared state, so the allocator can employ per-cache locking instead of protecting the entire arena (kernel heap) with one global lock. Per-cache locking improves scalability by allowing any number of distinct caches to be accessed simultaneously.

Each cache maintains its own statistics — total allocations, number of allocated and free buffers, etc. These per-cache statistics provide insight into overall system behavior. They indicate which parts of the system consume the most memory and help to identify memory leaks. They also indicate the activity level in various subsystems, to the extent that allocator traffic is an accurate approximation. (Streams message allocation is a direct measure of streams activity, for example.)

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. The slab allocator is in this category, but has the advantage that its customizations are client-driven at run time rather than being hard-coded at compile time. (This is also true of the Zone allocator.)

The standard non-caching allocation routines, `kmem_alloc(9F)` and `kmem_free(9F)`, use object caches internally. At startup, the system creates a set of about 30 caches ranging in size from 8 bytes to 9K in roughly 10-20% increments. `kmem_alloc()` simply performs a `kmem_cache_alloc()` from the nearest-size cache. Allocations larger than 9K, which are rare, are handled directly by the back-end page supplier.

3.2. Slabs

The *slab* is the primary unit of currency in the slab allocator. When the allocator needs to grow a cache, for example, it acquires an entire slab of objects at once. Similarly, the allocator reclaims unused memory (shrinks a cache) by relinquishing a complete slab.

A slab consists of one or more pages of virtually contiguous memory carved up into equal-size chunks, with a reference count indicating how many of those chunks have been allocated. The benefits of using this simple data structure to manage the arena are somewhat striking:

(1) Reclaiming unused memory is trivial. When the slab reference count goes to zero the associated pages can be returned to the VM system. Thus a simple reference count replaces the complex trees, bitmaps, and coalescing algorithms found in most other allocators [Knuth68, Korn85, Standish80].

(2) Allocating and freeing memory are fast, constant-time operations. All we have to do is move an object to or from a freelist and update a reference count.

(3) Severe external fragmentation (unused buffers on the freelist) is unlikely. Over time, many allocators develop an accumulation of small, unusable buffers. This occurs as the allocator splits existing free buffers to satisfy smaller requests. For example, the right sequence of 32-byte and 40-byte allocations may result in a large accumulation of free 8-byte buffers — even though no 8-byte buffers are ever requested [Standish80]. 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. Any sequence of 32-byte and 40-byte allocations — no matter how complex — can only result in population of the 32-byte and 40-byte freelists. Since prior allocation is a good predictor of future allocation [Weinstock88] these buffers are likely to be used again.

The other reason that slabs reduce external fragmentation is that all objects in a slab are of the same type, so they have the same lifetime distribution.* The resulting segregation of short-lived and long-lived objects at slab granularity reduces the likelihood of an entire page being held hostage due to a single long-lived allocation [Barrett93, Hanson90].

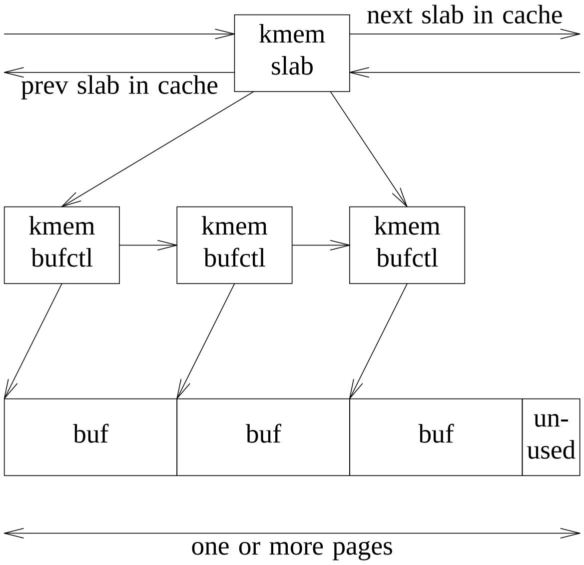
* The generic caches that back `kmem_alloc()` are a notable exception, but they constitute a relatively small fraction of the arena in SunOS 5.4 — all of the major consumers of memory now use `kmem_cache_alloc()`.

(4) Internal fragmentation (per-buffer wasted space) 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. For example, assuming 4096-byte pages, the slabs in a 400-byte object cache would each contain 10 buffers, with 96 bytes left over. We can view this as equivalent 9.6 bytes of wasted space per 400-byte buffer, or 2.4% internal fragmentation.

In general, 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 the slab size. However, larger slabs are more likely to cause external fragmentation, since the probability of being able to reclaim a slab decreases as the number of buffers per slab increases. The SunOS 5.4 implementation limits internal fragmentation to 12.5% (1/8), since this was found to be the empirical sweet-spot in the trade-off between internal and external fragmentation.

3.2.1. Slab Layout — Logical

The contents of each slab are managed by a `kmem_slab` data structure that maintains the slab's linkage in the cache, its reference count, and its list of free buffers. In turn, each buffer in the slab is managed by a `kmem_bufctl` structure that holds the freelist linkage, buffer address, and a back-pointer to the controlling slab. Pictorially, a slab looks like this (bufctl-to-slab back-pointers not shown):



3.2.2. Slab Layout for Small Objects

For objects smaller than 1/8 of a page, a slab is built by allocating a page, placing the slab data at the end, and dividing the rest into equal-size buffers:

buf	buf	...	buf	buf	un-used	kmem slab

one page

Each buffer serves as its own bufctl while on the freelist. Only the linkage is actually needed, since everything else is computable. These are essential optimizations for small buffers — otherwise we would end up allocating almost as much memory for bufctls as for the buffers themselves.

The freelist linkage resides at the *end* of the buffer, rather than the beginning, to facilitate debugging. This is driven by the empirical observation that the beginning of a data structure is typically more active than the end. If a buffer is modified after being freed, the problem is easier to diagnose if the heap *structure* (freelist linkage) is still intact.

The allocator reserves an additional word for constructed objects so that the linkage doesn't overwrite any constructed state.

3.2.3. Slab Layout for Large Objects

The above scheme is efficient for small objects, but not for large ones. It could fit only *one* 2K buffer on a 4K page because of the embedded slab data. Moreover, with large (multi-page) slabs we lose the ability to determine the slab data address from the buffer address. Therefore, for large objects the physical layout is identical to the logical layout. The required slab and bufctl data structures come from their own (small-object!) caches. A per-cache self-scaling hash table provides buffer-to-bufctl conversion.

3.3. Freelist Management

Each cache maintains a circular, doubly-linked list of all its slabs. The slab list is partially sorted, in that the empty slabs (all buffers allocated) come first, followed by the partial slabs (some buffers allocated, some free), and finally the complete slabs (all buffers free, refcnt == 0). The cache's freelist pointer points to its first non-empty slab. Each slab, in turn, has its own freelist of available buffers. This two-level freelist structure simplifies memory reclaiming. When the allocator reclaims a slab it doesn't have to unlink each buffer from the cache's freelist — it just unlinks the slab.

3.4. Reclaiming Memory

When `kmem_cache_free()` sees that the slab reference count is zero, it does not immediately reclaim the memory. Instead, it just moves the slab to the tail of the freelist where all the complete slabs reside. This ensures that no complete slab will be broken up unless all partial slabs have been depleted.

When the system runs low on memory it asks the allocator to liberate as much memory as it can. The allocator obliges, but retains a 15-second working set of recently-used slabs to prevent thrashing. Measurements indicate that system performance is fairly insensitive to the slab working-set interval. Presumably this is because the two extremes — zero working set (reclaim all complete slabs on demand) and infinite working-set (never reclaim anything) — are both reasonable, albeit suboptimal, policies.

4. Hardware Cache Effects

Modern hardware relies on good cache utilization, so it is important to design software with cache effects in mind. For a memory allocator there are two broad classes of cache effects to consider: the distribution of buffer addresses and the cache footprint of the allocator itself. The latter topic has received some attention [Chen93, Grunwald93B], but the effect of buffer address distribution on cache utilization and bus balance has gone largely unrecognized.

4.1. Impact of Buffer Address Distribution on Cache Utilization

The address distribution of mid-size buffers can affect the system's overall cache utilization. In particular, power-of-two allocators — where all buffers are 2^n bytes and are 2^n-byte aligned — are pessimal.* Suppose, for example, that every inode (≈300 bytes) is assigned a 512-byte buffer, 512-byte aligned, and that only the first dozen fields of an inode (48 bytes) are frequently referenced. Then the majority of inode-related memory traffic will be

* Such allocators are common because they are easy to implement. For example, 4.4BSD and SVr4 both employ power-of-two methods [McKusick88, Lee89].

at addresses between 0 and 47 modulo 512. Thus the cache lines near 512-byte boundaries will be heavily loaded while the rest lie fallow. In effect only 9% (48/512) of the cache will be usable by inodes. Fully-associative caches would not suffer this problem, but current hardware trends are toward simpler rather than more complex caches.

Of course, there's nothing special about inodes. The kernel contains many other mid-size data structures (e.g. 100-500 bytes) with the same essential qualities: there are many of them, they contain only a few heavily used fields, and those fields are grouped together at or near the beginning of the structure. This artifact of the way data structures evolve has not previously been recognized as an important factor in allocator design.

4.2. Impact of Buffer Address Distribution on Bus Balance

On a machine that interleaves memory across multiple main buses, the effects described above also have a significant impact on bus utilization. The SPARCcenter 2000, for example, employs 256-byte interleaving across two main buses [Cekleov92]. Continuing the example above, we see that any power-of-two allocator maps the first half of every inode (the hot part) to bus 0 and the second half to bus 1. Thus almost all inode-related cache misses are serviced by bus 0. The situation is exacerbated by an inflated miss rate, since all of the inodes are fighting over a small fraction of the cache.

These effects can be dramatic. On a SPARCcenter 2000 running LADDIS under a SunOS 5.4 development kernel, replacing the old allocator (a power-of-two buddy-system [Lee89]) with the slab allocator reduced bus imbalance from 43% to just 17%. In addition, the primary cache miss rate dropped by 13%.

4.3. Slab Coloring

The slab allocator incorporates a simple coloring scheme that distributes buffers evenly throughout the cache, resulting in excellent cache utilization and bus balance. The concept is simple: each time a new slab is created, the buffer addresses start at a slightly different offset (color) from the slab base (which is always page-aligned). For example, for a cache of 200-byte objects with 8-byte alignment, the first slab's buffers would be at addresses 0, 200, 400, ... relative to the slab base. The next slab's buffers would be at offsets 8, 208, 408, ... and so on. The maximum slab color is determined by the amount of unused space in the slab. In this example, assuming 4K pages, we can fit 20 200-byte buffers in a 4096-byte slab. The buffers consume 4000 bytes, the `kmem_slab` data consumes 32 bytes, and the remaining 64 bytes are available for coloring. Thus the maximum slab color is 64, and the slab color sequence is 0, 8, 16, 24, 32, 40, 48, 56, 64, 0, 8, ...

One particularly nice property of this coloring scheme is that mid-size power-of-two buffers receive the maximum amount of coloring, since they are the worst-fitting. For example, while 128 bytes goes perfectly into 4096, it goes near-pessimally into 4096 - 32, which is what's actually available (because of the embedded slab data).

4.4. Arena Management

An allocator's arena management strategy determines its dynamic cache footprint. These strategies fall into three broad categories: sequential-fit methods, buddy methods, and segregated-storage methods [Standish80].

A sequential-fit allocator must typically search several nodes to find a good-fitting buffer. Such methods are, by nature, condemned to a large cache footprint: they have to examine a significant number of nodes that are generally nowhere near each other. This causes not only cache misses, but TLB misses as well. The coalescing stages of buddy-system allocators [Knuth68, Lee89] have similar properties.

A segregated-storage allocator, such as the slab allocator, maintains separate freelists for different buffer sizes. These allocators generally have good cache locality because allocating a buffer is so simple. All the allocator has to do is determine the right freelist (by computation, by table lookup, or by having it supplied as an argument) and take a buffer from it. Freeing a buffer is similarly straightforward. There are only a handful of pointers to load, so the cache footprint is small.

The slab allocator has the additional advantage that for small to mid-size buffers, most of the relevant information — the slab data, bufctls, and buffers themselves — resides on a single page. Thus a single TLB entry covers most of the action.

5. Performance

This section compares the performance of the slab allocator to three other well-known kernel memory allocators:

> **SunOS 4.1.3**, based on [Stephenson83], a sequential-fit method;
>
> **4.4BSD**, based on [McKusick88], a power-of-two segregated-storage method;
>
> **SVr4**, based on [Lee89], a power-of-two buddy-system method. This allocator was employed in all previous **SunOS 5.x** releases.

To get a fair comparison, each of these allocators was ported into the same SunOS 5.4 base system. This ensures that we are comparing just allocators, not entire operating systems.

5.1. Speed Comparison

On a SPARCstation-2 the time required to allocate and free a buffer under the various allocators is as follows:

Memory Allocation + Free Costs		
allocator	time (µsec)	interface
slab	3.8	kmem_cache_alloc
4.4BSD	4.1	kmem_alloc
slab	4.7	kmem_alloc
SVr4	9.4	kmem_alloc
SunOS 4.1.3	25.0	kmem_alloc

Note: The 4.4BSD allocator offers both functional and preprocessor macro interfaces. These measurements are for the functional version. Non-binary interfaces in general were not considered, since these cannot be exported to drivers without exposing the implementation. The 4.4BSD allocator was compiled *without* `KMEMSTATS` defined (it's on by default) to get the fastest possible code.

A `mutex_enter()/mutex_exit()` pair costs 1.0 µsec, so the locking required to allocate and free a buffer imposes a lower bound of 2.0 µsec. The slab and 4.4BSD allocators are both very close to this limit because they do very little work in the common cases. The 4.4BSD implementation of `kmem_alloc()` is slightly faster, since it has less accounting to do (it never reclaims memory). The slab allocator's `kmem_cache_alloc()` interface is even faster, however, because it doesn't have to determine which freelist (cache) to use — the cache descriptor is passed as an argument to `kmem_cache_alloc()`. In any event, the differences in speed between the slab and 4.4BSD allocators are small. This is to be expected, since all segregated-storage methods are operationally similar. Any good segregated-storage implementation should achieve excellent performance.

The SVr4 allocator is slower than most buddy systems but still provides reasonable, predictable speed. The SunOS 4.1.3 allocator, like most sequential-fit methods, is comparatively slow and quite variable.

The benefits of object caching are not visible in the numbers above, since they only measure the cost of the allocator itself. The table below shows the effect of object caching on some of the most frequent allocations in the SunOS 5.4 kernel (SPARCstation-2 timings, in microseconds):

Effect of Object Caching			
allocation type	without caching	with caching	improvement
allocb	8.3	6.0	1.4x
dupb	13.4	8.7	1.5x
shalloc	29.3	5.7	5.1x
allocq	40.0	10.9	3.7x
anonmap_alloc	16.3	10.1	1.6x
makepipe	126.0	98.0	1.3x

All of the numbers presented in this section measure the performance of the allocator in isolation. The allocator's effect on overall system performance will be discussed in Section 5.3.

5.2. Memory Utilization Comparison

An allocator generally consumes more memory than its clients actually request due to imperfect fits (internal fragmentation), unused buffers on the freelist (external fragmentation), and the overhead of the allocator's internal data structures. The ratio of memory requested to memory consumed is the allocator's *memory utilization*. The complementary ratio is the *memory wastage* or *total fragmentation*. Good memory utilization is essential, since the kernel heap consumes physical memory.

An allocator's space efficiency is harder to characterize than its speed because it is workload-dependent. The best we can do is to measure the various allocators' memory utilization under a fixed set of workloads. To this end, each allocator was subjected to the following workload sequence:

(1) System boot. This measures the system's memory utilization at the console login prompt after rebooting.

(2) A brief spike in load, generated by the following trivial program:

```
fork(); fork(); fork(); fork();
fork(); fork(); fork(); fork();
fd = socket(AF_UNIX, SOCK_STREAM, 0);
sleep(60);
close(fd);
```

This creates 256 processes, each of which creates a socket. This causes a temporary surge in demand for a variety of kernel data structures.

(3) Find. This is another trivial spike-generator:

```
find /usr -mount -exec file {} \;
```

(4) Kenbus. This is a standard timesharing benchmark. Kenbus generates a large amount of concurrent activity, creating large demand for both user and kernel memory.

Memory utilization was measured after each step. The table below summarizes the results for a 16MB SPARCstation-1. The slab allocator significantly outperformed the others, ending up with half the fragmentation of the nearest competitor (results are cumulative, so the ‘‘kenbus’’ column indicates the fragmentation after all four steps were completed):

	Total Fragmentation (waste)				
allocator	boot	spike	find	kenbus	s/m
slab	11%	13%	14%	14%	233
SunOS 4.1.3	7%	19%	19%	27%	210
4.4BSD	20%	43%	43%	45%	205
SVr4	23%	45%	45%	46%	199

The last column shows the kenbus results, which measure peak throughput in units of scripts executed per minute (s/m). Kenbus performance is primarily memory-limited on this 16MB system, which is why the SunOS 4.1.3 allocator achieved better results than the 4.4BSD allocator despite being significantly slower. The slab allocator delivered the best performance by an 11% margin because it is both fast *and* space-efficient.

To get a handle on real-life performance the author used each of these allocators for a week on his personal desktop machine, a 32MB SPARCstation-2. This machine is primarily used for reading e-mail, running simple commands and scripts, and connecting to test machines and compute servers. The results of this obviously non-controlled experiment were:

Effect of One Week of Light Desktop Use		
allocator	kernel heap	fragmentation
slab	6.0 MB	9%
SunOS 4.1.3	6.7 MB	17%
SVr4	8.5 MB	35%
4.4BSD	9.0 MB	38%

These numbers are consistent with the results from the synthetic workload described above. In both cases, the slab allocator generates about half the fragmentation of SunOS 4.1.3, which in turn generates about half the fragmentation of SVr4 and 4.4BSD.

5.3. Overall System Performance

The kernel memory allocator affects overall system performance in a variety of ways. In previous sections we considered the effects of several individual factors: object caching, hardware cache and bus effects, speed, and memory utilization. We now turn to the most important metric: the bottom-line performance of interesting workloads. In SunOS 5.4 the SVr4-based allocator was replaced by the slab allocator described here. The table below shows the net performance improvement in several key areas.

System Performance Improvement with Slab Allocator		
workload	gain	what it measures
DeskBench	12%	window system
kenbus	17%	timesharing
TPC-B	4%	database
LADDIS	3%	NFS service
parallel make	5%	parallel compilation
terminal server	5%	many-user typing

Notes:

(1) DeskBench and kenbus are both memory-bound in 16MB, so most of the improvement here is due to the slab allocator’s space efficiency.

(2) The TPC-B workload causes very little kernel memory allocation, so the allocator’s speed is not a significant factor here. The test was run on a large server with enough memory that it never paged (under either allocator), so space efficiency is not a factor either. The 4% performance improvement is due solely to better cache utilization (5% fewer primary cache misses, 2% fewer secondary cache misses).

(3) Parallel make was run on a large server that never paged. This workload generates a lot of allocator traffic, so the improvement here is attributable to the slab allocator's speed, object caching, and the system's lower overall cache miss rate (5% fewer primary cache misses, 4% fewer secondary cache misses).

(4) Terminal server was also run on a large server that never paged. This benchmark spent 25% of its time in the kernel with the old allocator, versus 20% with the new allocator. Thus, the 5% bottom-line improvement is due to a 20% reduction in kernel time.

6. Debugging Features

Programming errors that corrupt the kernel heap — such as modifying freed memory, freeing a buffer twice, freeing an uninitialized pointer, or writing beyond the end of a buffer — are often difficult to debug. Fortunately, a thoroughly instrumented kernel memory allocator can detect many of these problems.

This section describes the debugging features of the slab allocator. These features can be enabled in any SunOS 5.4 kernel (not just special debugging versions) by booting under kadb (the kernel debugger) and setting the appropriate flags.* When the allocator detects a problem, it provides detailed diagnostic information on the system console.

6.1. Auditing

In audit mode the allocator records its activity in a circular transaction log. It stores this information in an extended version of the bufctl structure that includes the thread pointer, hi-res timestamp, and stack trace of the transaction. When corruption is detected by any of the other methods, the previous owners of the affected buffer (the likely suspects) can be determined.

6.2. Freed-Address Verification

The buffer-to-bufctl hash table employed by large-object caches can be used as a debugging feature: if the hash lookup in `kmem_cache_free()` fails, then the caller must be attempting to free a bogus address. The allocator can verify *all* freed addresses by changing the ''large object'' threshold to zero.

* The availability of these debugging features adds no cost to most allocations. The per-cache flag word that indicates whether a hash table is present — i.e., whether the cache's objects are larger than 1/8 of a page — also contains the debugging flags. A single test checks all of these flags simultaneously, so the common case (small objects, no debugging) is unaffected.

6.3. Detecting Use of Freed Memory

When an object is freed, the allocator applies its destructor and fills it with the pattern 0xdeadbeef. The next time that object is allocated, the allocator verifies that it still contains the deadbeef pattern. It then fills the object with 0xbaddcafe and applies its constructor. The deadbeef and baddcafe patterns are chosen to be readily human-recognizable in a debugging session. They represent freed memory and uninitialized data, respectively.

6.4. Redzone Checking

Redzone checking detects writes past the end of a buffer. The allocator checks for redzone violations by adding a guard word to the end of each buffer and verifying that it is unmodified when the buffer is freed.

6.5. Synchronous Unmapping

Normally, the slab working-set algorithm retains complete slabs for a while. In synchronous-unmapping mode the allocator destroys complete slabs immediately. `kmem_slab_destroy()` returns the underlying memory to the back-end page supplier, which unmaps the page(s). Any subsequent reference to any object in that slab will cause a kernel data fault.

6.6. Page-per-buffer Mode

In page-per-buffer mode each buffer is given an entire page (or pages) so that every buffer can be unmapped when it is freed. The slab allocator implements this by increasing the alignment for all caches to the system page size. (This feature requires an obscene amount of physical memory.)

6.7. Leak Detection

The timestamps provided by auditing make it easy to implement a crude kernel memory leak detector at user level. All the user-level program has to do is periodically scan the arena (via /dev/kmem), looking for the appearance of new, persistent allocations. For example, any buffer that was allocated an hour ago and is still allocated now is a *possible* leak.

6.8. An Example

This example illustrates the slab allocator's response to modification of a free snode:

```
kernel memory allocator: buffer modified after being freed
modification occurred at offset 0x18 (0xdeadbeef replaced by 0x34)
buffer=ff8eea20  bufctl=ff8efef0  cache: snode_cache
previous transactions on buffer ff8eea20:

thread=ff8b93a0  time=T-0.000089  slab=ff8ca8c0  cache: snode_cache
kmem_cache_alloc+f8
specvp+48
ufs_lookup+148
lookuppn+3ac
lookupname+28
vn_open+a4
copen+6c
syscall+3e8

thread=ff8b94c0  time=T-1.830247  slab=ff8ca8c0  cache: snode_cache
kmem_cache_free+128
spec_inactive+208
closef+94
syscall+3e8

(transaction log continues at ff31f410)
kadb[0]:
```

Other errors are handled similarly. These features have proven helpful in debugging a wide range of problems during SunOS 5.4 development.

7. Future Directions

7.1. Managing Other Types of Memory

The slab allocator gets its pages from segkmem via the routines `kmem_getpages()` and `kmem_freepages()`; it assumes nothing about the underlying segment driver, resource maps, translation setup, etc. Since the allocator respects this firewall, it would be trivial to plug in alternate back-end page suppliers. The ''getpages'' and ''freepages'' routines could be supplied as additional arguments to `kmem_cache_create()`. This would allow us to manage multiple types of memory (e.g. normal kernel memory, device memory, pageable kernel memory, NVRAM, etc.) with a single allocator.

7.2. Per-Processor Memory Allocation

The per-processor allocation techniques of McKenney and Slingwine [McKenney93] would fit nicely on top of the slab allocator. They define a four-layer allocation hierarchy of decreasing speed and locality: per-CPU, global, coalesce-to-page, and coalesce-to-VM-block. The latter three correspond closely to the slab allocator's front-end, back-end, and page-supplier layers, respectively. Even in the absence of lock contention, small per-processor freelists could improve performance by eliminating locking costs and reducing invalidation traffic.

7.3. User-level Applications

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)`.

8. Conclusions

The slab allocator is a simple, fast, and space-efficient kernel memory allocator. The object-cache interface upon which it is based reduces the cost of allocating and freeing complex objects and enables the allocator to segregate objects by size and lifetime distribution. Slabs take advantage of object size and lifetime segregation to reduce internal and external fragmentation, respectively. Slabs also simplify reclaiming by using a simple reference count instead of coalescing. The slab allocator establishes a push/pull relationship between its clients and the VM system, eliminating the need for arbitrary limits or watermarks to govern reclaiming. The allocator's coloring scheme distributes buffers evenly throughout the cache, improving the system's overall cache utilization and bus balance. In several important areas, the slab allocator provides measurably better system performance.

Acknowledgements

Neal Nuckolls first suggested that the allocator should retain an object's state between uses, as our old streams allocator did (it now uses the slab allocator directly). Steve Kleiman suggested using VM pressure to regulate reclaiming. Gordon Irlam pointed out the negative effects of power-of-two alignment on cache utilization; Adrian Cockcroft hypothesized that this might explain the bus imbalance we were seeing on some machines (it did).

I'd like to thank Cathy Bonwick, Roger Faulkner, Steve Kleiman, Tim Marsland, Rob Pike, Andy Roach, Bill Shannon, and Jim Voll for their thoughtful comments on draft versions of this paper. Thanks also to David Robinson, Chaitanya Tikku, and Jim Voll for providing some of the measurements, and to Ashok Singhal for providing the tools to measure cache and bus activity.

Most of all, I thank Cathy for putting up with me (and without me) during this project.

References

[Barrett93] David A. Barrett and Benjamin G. Zorn, *Using Lifetime Predictors to Improve Memory Allocation Performance.* Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation, pp. 187-196 (1993).

[Boehm88] H. Boehm and M. Weiser, *Garbage Collection in an Uncooperative Environment.* Software - Practice and Experience, v. 18, no. 9, pp 807-820 (1988).

[Bozman84A] G. Bozman, W. Buco, T. Daly, and W. Tetzlaff, *Analysis of Free Storage Algorithms -- Revisited.* IBM Systems Journal, v. 23, no. 1, pp. 44-64 (1984).

[Bozman84B] G. Bozman, *The Software Lookaside Buffer Reduces Search Overhead with Linked Lists.* Communications of the ACM, v. 27, no. 3, pp. 222-227 (1984).

[Cekleov92] Michel Cekleov, Jean-Marc Frailong and Pradeep Sindhu, *Sun-4D Architecture.* Revision 1.4, 1992.

[Chen93] J. Bradley Chen and Brian N. Bershad, *The Impact of Operating System Structure on Memory System Performance.* Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, v. 27, no. 5, pp. 120-133 (1993).

[Grunwald93A] Dirk Grunwald and Benjamin Zorn, *CustoMalloc: Efficient Synthesized Memory Allocators.* Software - Practice and Experience, v. 23, no. 8, pp. 851-869 (1993).

[Grunwald93B] Dirk Grunwald, Benjamin Zorn and Robert Henderson, *Improving the Cache Locality of Memory Allocation.* Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation, pp. 177-186 (1993).

[Hanson90] David R. Hanson, *Fast Allocation and Deallocation of Memory Based on Object Lifetimes.* Software - Practice and Experience, v. 20, no. 1, pp. 5-12 (1990).

[Knuth68] Donald E. Knuth, *The Art of Computer Programming, Vol I, Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1968.

[Korn85] David G. Korn and Kiem-Phong Vo, *In Search of a Better Malloc.* Proceedings of the Summer 1985 Usenix Conference, pp. 489-506.

[Lee89] T. Paul Lee and R. E. Barkley, *A Watermark-based Lazy Buddy System for Kernel Memory Allocation.* Proceedings of the Summer 1989 Usenix Conference, pp. 1-13.

[Leverett82] B. W. Leverett and P. G. Hibbard, *An Adaptive System for Dynamic Storage Allocation.* Software - Practice and Experience, v. 12, no. 3, pp. 543-555 (1982).

[Margolin71] B. Margolin, R. Parmelee, and M. Schatzoff, *Analysis of Free Storage Algorithms.* IBM Systems Journal, v. 10, no. 4, pp. 283-304 (1971).

[McKenney93] Paul E. McKenney and Jack Slingwine, *Efficient Kernel Memory Allocation on Shared-Memory Multiprocessors.* Proceedings of the Winter 1993 Usenix Conference, pp. 295-305.

[McKusick88] Marshall Kirk McKusick and Michael J. Karels, *Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel.* Proceedings of the Summer 1988 Usenix Conference, pp. 295-303.

[Oldehoeft85] Rodney R. Oldehoeft and Stephen J. Allan, *Adaptive Exact-Fit Storage Management.* Communications of the ACM, v. 28, pp. 506-511 (1985).

[Standish80] Thomas Standish, *Data Structure Techniques.* Addison-Wesley, Reading, MA, 1980.

[Stephenson83] C. J. Stephenson, *Fast Fits: New Methods for Dynamic Storage Allocation.* Proceedings of the Ninth ACM Symposium on Operating Systems Principles, v. 17, no. 5, pp. 30-32 (1983).

[VanSciver88] James Van Sciver and Richard F. Rashid, *Zone Garbage Collection.* Proceedings of the Summer 1990 Usenix Mach Workshop, pp. 1-15.

[Weinstock88] Charles B. Weinstock and William A. Wulf, *QuickFit: An Efficient Algorithm for Heap Storage Allocation.* ACM SIGPLAN Notices, v. 23, no. 10, pp. 141-144 (1988).

[Zorn93] Benjamin Zorn, *The Measured Cost of Conservative Garbage Collection.* Software - Practice and Experience, v. 23, no. 7, pp. 733-756 (1993).

Author Information

Jeff Bonwick is a kernel hacker at Sun. He likes to rip out big, slow, old code and replace it with small, fast, new code. He still can't believe he gets paid for this. The author received a B.S. in Mathematics from the University of Delaware (1987) and an M.S. in Statistics from Stanford (1990). He can be flamed electronically at *bonwick@eng.sun.com*.