Experience With an Efficient Parallel Kernel Memory Allocator

Paul E. McKenney
Jack Slingwine
Phil Krueger

Sequent Computer Systems, Inc.
15450 SW Koll Parkway
Beaverton, OR 97006
(503) 578-5131
Email: {mckenney,jacks,philk}@sequent.com
FAX: (503) 578-5271

August 6, 1995

Abstract

There has been great progress from the traditional allocation algorithms designed for small memories to more modern algorithms exemplified by McKusick's and Karel's allocator [MK88]. Nonetheless, none of these algorithms have been designed to meet the needs of UNIX kernels supporting commercial data-processing applications in a shared-memory multiprocessor environment.

On a shared-memory multiprocessor, memory is a global resource. Therefore, allocator performance depends on synchronization primitives and manipulation of shared data as well as on raw CPU speed.

Synchronization primitives and access to shared data depend on system bus interactions. The speed of system busses has not kept pace with that of CPUs, as witnessed by the ever-larger caches found on recent systems. Thus, the performance of synchronization primitives and of memory allocators that use them have not received the full benefit of increased CPU performance.

An earlier paper [MS93] describes an allocator designed to meet this situation. This paper reviews the motivation for and design of the allocator and the presents experience gained during the three years that the allocator has been in production use.

1 Introduction

Parallel implementations of UNIX have been quite successful at meeting the the needs of online transaction-processing (OLTP) applications. Nonetheless, one weakness of previous implementations has been the general-purpose kernel memory allocator.

The old version of the allocator is a straightforward global allocator whose critical sections are protected by spinlocks. Although this worked quite well on older platforms, this allocator's performance is less than optimal on newer platforms, primarily because the speed of synchronization primitives (such as spinlocks) has not increased as rapidly as the speed of other instructions.

There has also been great progress in the area of multiprocessor synchronization primitives (citations [BK91, Her91, Her93, SSHT93] give an overview of several areas of recent progress). However, synchronization requires global processing. Global processing is very costly in comparison to local processing and can be expected to become even more expensive as technology advances [HJ91, SC91]. We therefore decided to abandon the search for ever-more sophisticated synchronization primitives in favor of a search for an algorithm that does not depend so heavily on synchronization. This search bore fruit in the form of an algorithm that runs fifteen times as fast as the

old allocator on a single processor and that exhibits linear speedup on shared-memory multiprocessors, resulting in more than a three-orders-of-magnitude increase in performance, while adding online coalescing.

Section 2 analyzes the behavior of the old algorithm. Section 3 presents the new algorithm, Section 4 evaluates an implementation, and Section 5 presents experience gathered during three years production use of the new allocator.

2 Analysis

Our investigation into kernel-memory-allocation performance began when we found that the STREAMS [Rit84] buffer allocator was running four to five times more slowly than predicted by instruction counts. We quickly realized that the general-purpose kernel-memory allocator suffered from the same problem, which motivated us to develop the algorithm described in this paper.

The remainder of this section presents the results of the investigation, describing the initial behavior of *allocb* (the STREAMS buffer allocator) and *freeb* (the STREAMS buffer deallocator) and showing how the current allocation algorithm's interaction with the shared-memory multiprocessor environment leads to this behavior. All measurements presented in this section were taken on a Sequent S2000/200 with a pair of 25 MHz 80486 CPUs running DYNIX/ptx, a parallel variant of UNIX.

2.1 Behavior of *allocb*

The *allocb* function returns a pointer to a message, which consists of a message block, data block, and STREAMS buffer. To do this, it must find a buffer capable of holding the specified number of bytes, allocate a message block and data block, and initialize them so that the message block points to the data block that points to the STREAMS buffer. The caller may then link several messages together to form a segmented message, add the message to a queue, allocate a new message block to form a second reference to some data (for example, in order to retain the data for possible later retransmission), or free up the message.

When sufficient memory is available, *allocb* executes a nearly fixed code sequence[1] that would require 12.5 microseconds in the absence of cache misses. However, measured times ranged from 28 to 198 microseconds, with the average at 64.2 microseconds. We captured a 64.76 microsecond trace on a logic analyzer and found that the worst 19 of the 304 off-chip accesses (6.3%) accounted for 57.6% of the elapsed time and that the worst 31 (10.2%) accounted for 68.4% of the elapsed time.

2.2 Behavior of *freeb*

The *freeb* function typically executes a fixed code sequence that would require 8.8 microseconds in the absence of cache misses. Measured times ranged from 16 to 176 microseconds, with the average at 48.7 microseconds. We captured a 102.8 microsecond trace on a logic analyzer representing a back-to-back pair of *freeb*s invoked from *freemsg*, and found that the worst 28 of the 322 off-chip accesses (8.6%) accounted for 50.6% of the elapsed time, while the worst 74 (23.0%) accounted for 80.3% of the elapsed time.

In both *allocb* and *freeb* the worst accesses were cache misses, either to main memory, to the other processor's cache, or to uncacheable device registers. Note that this behavior is not peculiar to *allocb* or *freeb*; any allocator that consisted of a traditional allocator protected by a simple mutual-exclusion scheme (such as the general-purpose kernel memory allocator) would suffer from the same problem. Other investigators [TGH92] have independently demonstrated some of the difficulties with use of simple mutual exclusion to protect data structures used by traditional algorithms.

An improved version of *allocb* is presented in [MG92]. This paper describes an improved version of the general-purpose kernel memory allocator.

3 Memory Allocator Design

This section presents the design goals that we set out for the new memory allocator, followed by the design itself.

[1] There is a small loop that selects the proper freelist given the block size, but the maximum execution time for this loop is only a few percent of the total runtime. There are also variations in the number of TLB misses.

3.1 Design Goals

The design goals for the new allocator are: (1) to implement full System V semantics, (2) to support high allocation/deallocation rates, (3) to scale well with increasing processor speeds, (4) to exhibit linear speedup on shared-memory multiprocessors, (5) to be capable of allocating all available buffers to any or all CPUs, and (6) to be capable of coalescing blocks so as to reallocate the memory to different-sized requests.

Implementing full System V semantics adds some overhead. A more efficient interface would allow the caller to request that a given block size be encoded into a "magic cookie" for use in subsequent allocation requests for that size, greatly reducing the number of translations from block size to freelist address. In addition, it is permissible to take the address of the System V allocation (*kmem_alloc*) and deallocation (*kmem_free*) functions. A more efficient interface would also provide C preprocessor macros to perform these functions, thereby avoiding function-call overhead. This paper reports the performance of both the standard version and an optimized version.

An important goal is to exceed the performance of simple global mutual-exclusion. An allocator that meets this goal is faster than any possible ad-hoc allocator based on mutual exclusion; thus, it almost entirely eliminates any motivation to create such ad-hoc allocators. One situation in which ad-hoc allocators are still beneficial is when the structures being allocated are subject to some complex but reusable initialization. The STREAMS buffer allocator described earlier provides an example of this situation. Three different structures (the message block, data block, and data buffer) must be linked together and allocated as a unit. However, the memory allocator's code may be reused for special-purpose allocators such as the STREAMS buffer allocator. This reuse occurs at the binary level,[2] so that a proliferation of special-purpose allocators can be accomodated, if need be, without undue kernel bloat.

A good allocator will scale with the processor speeds as opposed to interconnect latencies. This requires that the allocator exhibit good locality of reference in order to avoid cache-thrashing and that it avoid use of instructions such as read-modify-write operations and branches that can result in CPU pipeline stalls.

Read-modify-write instructions can result in pipeline stalls because they are required to be executed as if they are atomic. Modern microprocessors operate in a pipelined fashion, overlapping the execution of several instructions. The execution of atomic operations may be overlapped with that of other instructions only under very restricted conditions. Further advances in the art of CPU design might well ease these restrictions. However, superscalar techniques (execution of several parallel pipelines within a single CPU) will increase the penalty associated with stalling for atomic operations.

Branches can result in pipeline stalls because it is not always possible to determine the branch's outcome early enough to do sufficient instruction prefetching. Therefore, the pipeline can stall, waiting for instructions to be fetched from memory or from cache. This effect can be clearly seen in logic-analyzer traces; instruction prefetch will continue along the wrong path when the outcome of a branch is not correctly predicted. The exact magnitude of this effect varies with architecture and with the exact circumstances of the mispredicted branch. However, the amount of effort that has been expended to cause compilers to more accurately predict branches gives some hint of the importance of this effect. Further advances in the arts of compiler and CPU design may make this issue less important, but algorithms such as fully-inlined binary search will likely remain problematic when presented with random input.

Near-linear speedups are needed in order to support configurations with large numbers of processors and communications interfaces. To achieve this goal, the allocator must avoid operations that require coordination between CPUs. An analogy drawn from traffic engineering may be helpful. Within cities, cars must frequently cross each other's paths. Drivers must coordinate their actions (with varying degrees of aggression) in order to avoid collision, and the speed limits are set low to allow for this coordination. In contrast, on rural freeways cars rarely cross each other's paths, and a much lower degree of coordination is required, thereby allowing speed limits to be set higher. Likewise, multiprocessor allocators that avoid the need for coordination avoid inconveniently-low speed limits.

[2] In other words, special-purpose allocators such as *allocb* invoke the same functions as does the general-purpose *kmem_alloc* allocator.

It is clearly important that any given CPU be able to allocate the last remaining buffer, although the allocator is permitted to incur more overhead in this hopefully infrequent low-memory situation.

It is not uncommon for machines in commercial environments to be presented a cyclic workload. For example, the machine might be used for data entry and queries as part of a distributed database during the day, and for backups and database reorganization at night. These different activities often require different sizes of memory allocations, e.g., the data entries and queries might require huge numbers of small blocks of memory to track database locking while the backups and database reorganization might require massive amounts of memory dedicated to user processes.

Consequently, the allocator must be able to coalesce adjacent free blocks of memory into larger blocks, allowing memory to be used to satisfy requests of different sizes or to be returned to the system for use by user processes. Allocators must recover from problems such as overallocation of memory to a given blocksize without a reboot. Coalescing should not interfere with normal system operation, since a one-minute pause caused by an offline coalescing algorithm can be just as disruptive as a reboot.

3.2 Roads Not Taken

We considered a number of possible allocation schemes.

Although the McKusick-Karels (MK) algorithm [MK88] is extremely efficient on uniprocessors when presented with requests whose sizes can be determined at compile-time, it does not meet goals 3 and 4 on multiprocessors. In particular, its fully-inlined binary search results in pipeline stalls because no reasonable instruction prefetch strategy can correctly predict all of the branches. As presented, the MK algorithm also fails to meet goal 6, but could be modified to do the required coalescing. Nonetheless, the large number of algorithms that are directly and indirectly derived from the MK algorithm (including the one presented in this paper) form an impressive testament to its strengths.

One such algorithm is the watermark-based lazy buddy system [LB89], which attempts to combine high-speed allocation with high-quality coalescing. However, it requires global synchronization on each operation and fails to maintain good locality of reference (since each block is sent singly to be coalesced, rather than being sent in large groups), thereby failing to meet goals 3 and 4 on multiprocessors.

Another MK-derived algorithm is Rogue Wave's C++ memory allocator [Mye92]. This allocator also attempts to combine high-speed allocation with high-quality coalescing, but intentionally degrades its ability to coalesce in favor of decreasing the resident set size of the program. This is a laudable goal within a user process, but is largely irrelevant within the kernel. Furthermore, the algorithm is not designed for use on multiprocessors and so does not meet goals 3 and 4 in this environment.

Algorithms designed specifically to promote high-quality coalescing [Knu73] are quite slow [KV85] and thus fail to meet goal 2. It is quite difficult to exceed the performance of removing the first element from a simple, singly-linked, linear list–particularly when that list is accessed from only one of the CPUs.

3.3 Allocator Design

The requirements for high speed and for coalescing conflict to a large degree. Very little coalescing can be performed within the 9-VAX-instruction budget of the McKusick-Karels allocator. It is nevertheless possible to do both high-speed allocation and high-quality, online coalescing by introducing the concept of layering to the allocator.

The allocator consists of four layers: (1) a per-CPU caching layer, (2) a global layer, (3) a coalesce-to-page layer, and (4) a coalesce-to-"vmblk" layer. The lower-numbered layers are optimized for speed, while the the higher-numbered layers are optimized for coalescing, as illustrated in Figure 1. The following sections describe each of these layers in turn. A final section describes how "cookies" are used to efficiently encapsulate request-size information.

3.3.1 Per-CPU Caching Layer

The only purpose of the per-CPU caching layer is to support high-speed allocation and deallocation in the common case. Each CPU maintains a local cache of buffers for each of a small fixed set of buffer sizes, much as the McKusick-Karels algorithm does. Consequently, there is

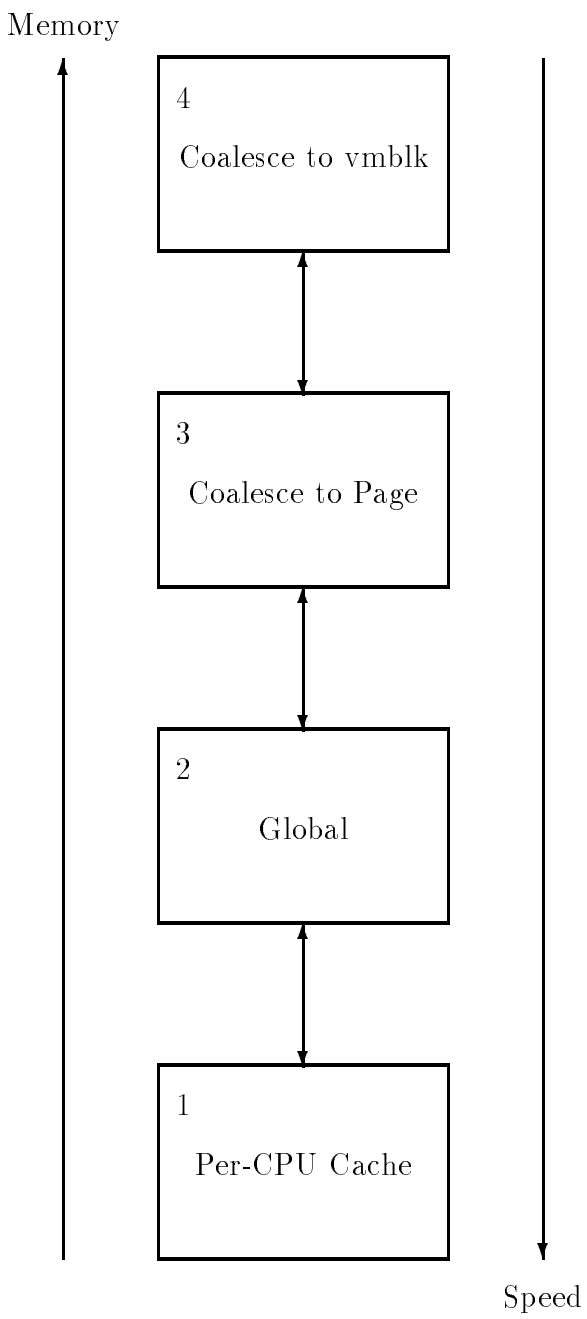

Figure 1: Allocator Layering

one instance of a per-CPU cache for each possible CPU-buffer-size combination. For example, a four-CPU system that managed the default set of nine power-of-two block sizes (16, 32, 64, 128, 254, 512, 1024, 2048, and 4096 bytes) would have 36 per-CPU caches.

The *kmem_alloc* function first attempts to satisfy a request for a given size of block from the appropriate cache on the current CPU. For example, an interrupt routine running on CPU 2 needing a 50-byte block would first check CPU 2's cache of 64-byte blocks. CPUs are prohibited from accessing other CPUs' per-CPU caches, thus removing the need for any synchronization primitives (other than the disabling of interrupts) guarding the per-CPU caches.

When a per-CPU cache is exhausted, it is replenished from the global layer; when it becomes too full (as determined by a kernel parameter named *target*), the excess is put back into the global layer. Blocks are moved in *target*-sized groups, preventing unnecessary linked-list operations. This is accomplished by maintaining a split freelist in the per-CPU cache as shown in Figure 2. The maximum size

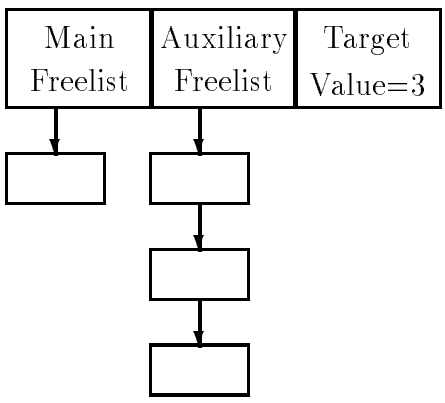

Figure 2: Per-CPU Data Structures

of each half of the per-CPU freelist is *target*, so that the total number of blocks in a per-CPU freelist may range up to twice *target*. Blocks are normally allocated from and freed to the *main* list. If adding another block would cause the *main* list to exceed *target*, *main* is moved to *aux*. If *aux* is not empty, its contents are first returned to the global layer. Thus, as shown in Figure 2, up to two additional blocks may be

freed onto *main*. Freeing a third block would cause the contents of *aux* to be returned to the global pool, the contents of *main* to be moved to *aux*, and the newly-freed block to be added to *main*. At this point, the configuration would again be as shown in Figure 2.

If *main* is empty upon allocation, the contents of *aux*, if any, are moved to *main*. If *aux* is also empty, *main* is instead replenished from the global layer. In the situation shown in Figure 2, one more block may be allocated from *main*, at which point *main* will be empty. A second allocation will result in the contents of *main* being moved to *aux* and one of the blocks being used to satisfy the allocation request. At this point, *main* will contain two more blocks and *aux* will be empty, allowing two additional allocations to be made directly from *main*. The next allocation would find both *main* and *aux* empty, causing *main* to be refilled from the global layer.

Note that the global layer will be accessed at most one time per *target*-number of accesses. This means that the per-allocation overhead incurred in the global layer may be reduced to any desired level simply by increasing the value of *target*. The only penalty for increasing *target* is the increased amount of memory that will reside in the per-CPU caches. In practice, there is no motivation to increase *target* beyond the point at which the global-layer overhead becomes an insignificant portion of the per-allocation overhead.

3.3.2 Global Layer

The only purpose of the global layer is to support reasonable performance in cases when one CPU allocates buffers of a given size, which are then passed to other CPUs that free them. The global layer allows the freed buffers to move back to the allocating CPU without incurring the overhead of coalescing.

There is a separate instance of the global layer for each block size. Each instance maintains free blocks in lists of *target*-sized lists, as shown in Figure 3. This technique allows *target*-sized blocks of data to be passed to and from the per-CPU layers with a minimum number of linked-list operations. Odd-sized lists of blocks may be passed into the global layer during low-memory operation or during per-CPU cache flushes. These lists are added to the *bucket* list, which is used to group the blocks back into *target*-sized lists.

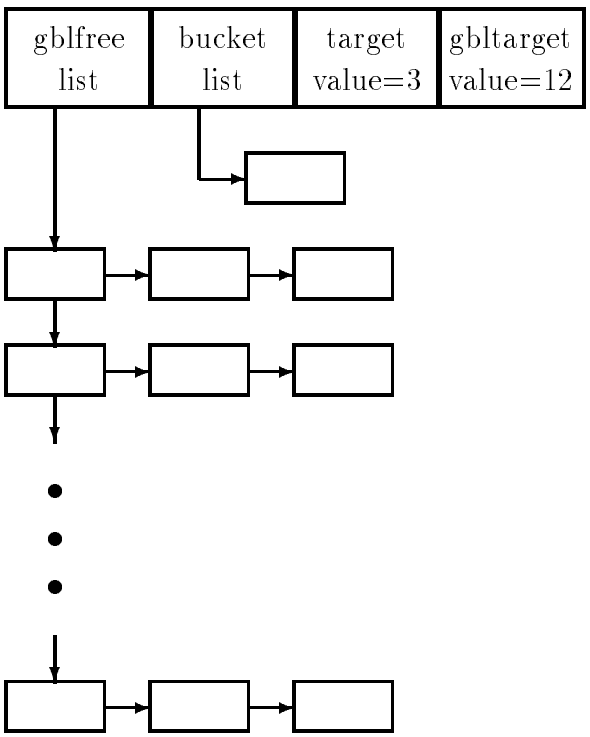

Figure 3: Global Layer Data Structures

When the global layer becomes too full, the excess buffers are sent up to the coalesce-to-page layer. When the global layer is empty, it is replenished from the coalesce-to-page layer. The number of blocks in the global layer ranges up to twice a parameter named *gbltarget*. There is no reason to maintain a split freelist at the global layer, since each block must be individually examined by the coalesce-to-page layer (described in the following section) in order to determine which page's freelist it belongs on.

A schematic view of the data structures implementing the per-CPU and global layers is shown in Figure 4. Each CPU has a pointer to an array of its per-CPU caches, and each per-CPU cache maintains a pointer to the global pool serving its blocksize. Request sizes are converted to indexes into the array of caches through use of a table indexed by size.

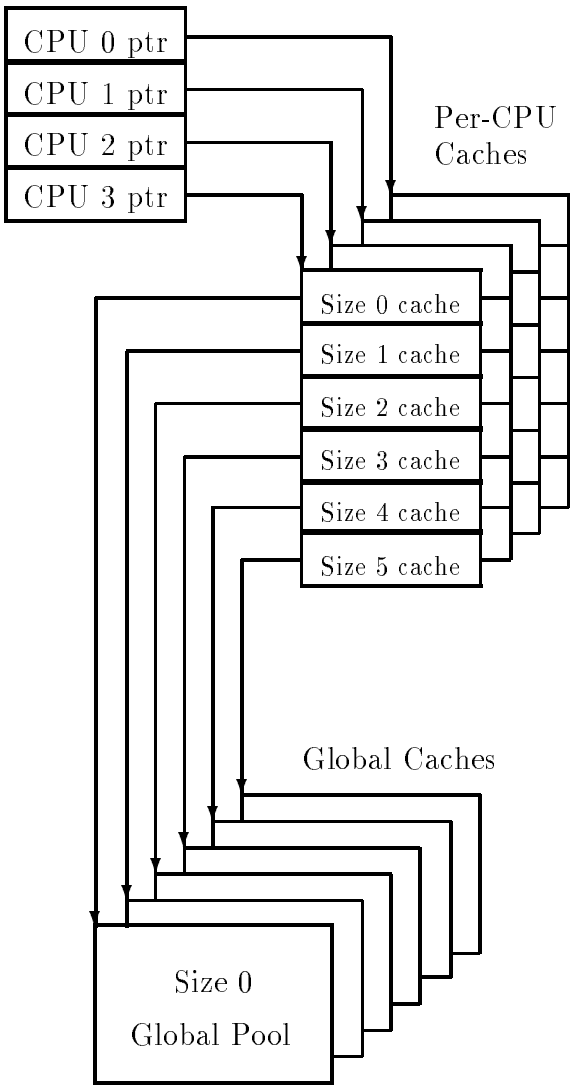

Figure 4: Per-CPU and Global Layers

3.3.3 Coalesce-to-Page Layer

The coalesce-to-page layer gathers blocks of a given size and coalesces them into pages. This layer maintains an auxiliary data structure for each page, which contains the per-page freelist and a count of the number of blocks in the page that are currently free (this per-page data structure is described in more detail in the discussion of the coalesce-to-vmblk layer below). When the count equals the total number of blocks in the page, the entire page may be given back to the system; in other words, the coalesce-to-page layer can immediately determine when all of the blocks in a given page have been freed up. This eliminates the need for a computationally-expensive mark-and-sweep algorithm or an offline sorting algorithm. Pages that have some blocks in use are placed on a radix-sorted freelist so that pages with the fewest free blocks will be allocated from most frequently, as shown in Figure 5. This sorting has the benefit of allowing pages that have only a few in-use blocks more time to gather them. In turn, this allows the page to be used for allocations of other sizes and for user processes.

Once all of the blocks in a page have been freed, the physical memory is returned to the system. The virtual memory is retained and passed up to the coalesce-to-vmblk layer. This process illustrates a key difference between kernel- and user-level memory allocators. Kernel-level allocators must manage the virtual address space and physical memory explicitly

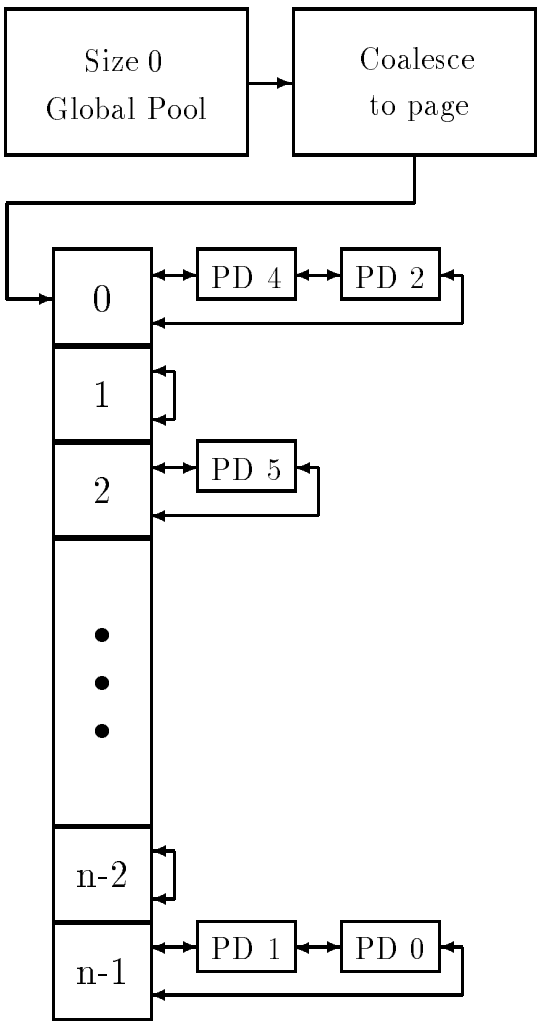

Figure 5: Coalesce-to-Page Layer

and separately. In contrast, user-level allocators need not and typically cannot easily distinguish between virtual and physical memory.

3.3.4 Coalesce-to-vmblk Layer

This layer manages large vmblks of virtual memory (4 megabytes in size for the current implementation). Pages of virtual-address space are allocated from vmblks as needed and are mapped onto physical memory. Requests for blocks of memory larger than one page bypass layers 1 through 3 and are handled directly by the coalesce-to-vmblk layer. Adjacent spans of free pages in a vmblk are coalesced as they are freed; a boundary-tag-like scheme uses per-page auxiliary data structures (called page descriptors) to track the sizes and locations of free spans of virtual memory.

The system must be able to locate the page descriptor corresponding to a particular block given only that block's address. This is accomplished with a two-level scheme using a sparse array as shown in Figure 6. In the first level the upper bits of the block's address are used to index into a dope vector, which contains the address of the vmblk containing that block. The vmblk consists of a group of page descriptors followed by the corresponding data pages. In the second level, the index of the block's page descriptor within the vmblk is obtained by subtracting the vmblk's address from the block's address, shifting off the lower bits to get the page index within the vmblk, and finally subtracting the number of pages occupied by the page descriptors.

This two-level scheme allows overhead information to be kept only for those pages controlled by the allocator. Other pages (such as those used by processes) require no such overhead. The performance penalty associated with this two-level scheme is incurred only at the coalesce-to-page and coalesce-to-vmblk layers, and therefore has a minimal effect on overall system performance.

Page descriptors corresponding to pages that have been split into blocks contain the block size, a freelist pointer, and the number of free blocks. Page descriptors corresponding to spans contain the boundary-tag information and free-list pointers needed to allocate and coalesce large blocks.

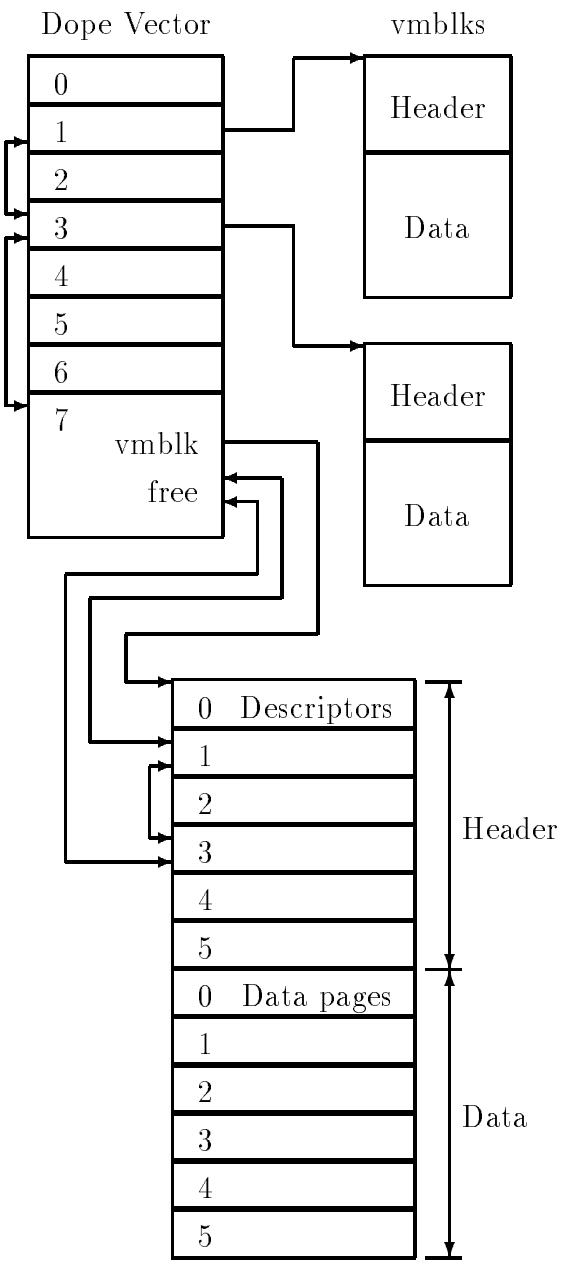

Figure 6: Sparse Array of Page Descriptors

3.3.5 Cookies

As noted earlier, there is significant overhead associated with inlined binary searches given widely-varying inputs that defeat branch-prediction schemes. Hence, the inline binary search used by the MK algorithm is most effective when the size is known at compile time. Otherwise, a subroutine call combined with a table lookup can be just as efficient.

Explicitly requiring that the request size be known at compile time allows the overhead of freeing to be further reduced (cases where the request size is not known at compile time may be handled by the standard function interface). The caller invokes *kmem_alloc_get_cookie* to translate a request size into an opaque "cookie" that is passed to subsequent expansions of the *KMEM_ALLOC_COOKIE* and *KMEM_FREE_COOKIE* macros. The cookie contains pointers to the proper per-CPU pools, removing the need for the free operation to determine the block size given only its address.

The use of cookies allows the common case of the free operation to consume only thirteen 80x86 instructions, as compared to the 16 VAX instructions consumed by the MK algorithm.

4 Measurements

The following sections present instruction counts for the allocator, measurements on a simple benchmark that exhibits best-case performance, measurements on another simple benchmark that exhibits worst-case performance, and finally measurements taken from a more sophisticated benchmark that makes more typical use of the allocator.

All measurements were taken on a Symmetry 2000 system with 50MHz 80486 processors.

4.1 Instruction Counts

The efficient "cookie" version of the allocator executes thirteen 80x86 instructions each for the allocation and free operations. Allocation overhead is comparable to that of MK when differences between the VAX and 80x86 instruction set are taken into account (in particular, the 80x86 lacks a memory-to-memory move instruction). A single additional memory reference is required in order to handle multiple processors. The overhead of freeing is somewhat

less than that of MK even without considering instruction-set differences. The difference is due to the use of the cookie-based scheme. MK must look up the block's size and use this information to index into the list of freelist, while the cookie allows direct access to the proper per-CPU cache.

Note that the efficient version is nonstandard and is useful only when the size of the request is known at compile-time.

The less efficient but standard interface executes 35 instructions for allocation and 32 instructions for freeing, assuming that the each of the actual arguments can be evaluated and stored with a single instruction. The additional overhead is caused by the function call and by the need to map from the request size to the proper per-CPU cache. Currently, all variable-sized structures have large initialization overheads that overwhelm the performance difference between the standard and cookie-based interfaces.[3] Therefore, there is currently little motivation to provide a third interface that provides speedier allocation of variable-length structures.

4.2 Best-Case Benchmark

We measured best-case performance by constructing a system call containing a loop that is run for a user-specified length of time. Each pass through the loop invokes *kmem_alloc* to allocate a buffer, then invokes *kmem_free* to immediately deallocate this same buffer. When the specified length of time has passed, the system call returns the number of *kmem_alloc*/*kmem_free* pairs that were executed. Thus, the measurements include the overhead of the loop which invokes *kmem_alloc* and *kmem_free*; this overhead amounts to as much as 40% for the faster algorithms. This system call is invoked from a user program, which is forced to run on a specified CPU. Multiple-CPU data is collected by running multiple instances of the program, each on its own CPU.

The performance was highly linear as shown in Figure 7. The x-axis shows the number of CPUs and the y-axis shows the number of pairs of allocation and freeing accomplished per second. The top trace shows the performance of the non-standard cookie-based macro, the next trace shows the performance of the standard functional interface, and the bottom two traces show the performance of naive parallelizations of the MK algorithm and of the "oldkma" algorithm, which resembles "Fast Fits" [Ste83] (algorithm "S" in Korn's and Vo's survey [KV85]).

Figure 8 displays the same data on a semilog plot so that the traces for the two slower algorithms may be more easily distinguished from the x-axis. The irregularities in the trace of the naive parallelization of the MK algorithm are due to second-order effects resulting from the extreme lock contention exhibited by this algorithm. These effects are largely masked by the greater overhead of the slower "oldkma" algorithm.

The cookie-based allocator ranges from 15 times the performance of the "oldkma" allocator on a single CPU to more than 1,000 times the performance on 25 CPUs.[4] The standard interface is roughly half as fast as the cookie-based allocator, but note that this dramatic-seeming difference in performance amounts to only about 20 instructions per operation.

In contrast, the other two schemes simply did not scale with increasing numbers of CPUs. In fact, in both cases, the best performance was observed when running on a single CPU.

Hardware monitors indicate that the common case of the two fast algorithms are free from the cache-thrashing that accounted for so much of the original algorithm's execution time. We therefore expect that the allocator will continue to scale well with increasing processor speeds.

4.3 Worst-Case Benchmark

The best-case benchmark exercises only the per-CPU caching layer. The worst-case benchmark exercises not only all the layers, but takes care to exercise the upper layers to the greatest extent possible, thereby incurring the worst possible per-allocation overhead. This is accomplished by allocating blocks of a given size until memory is exhausted, freeing them all, then repeating the process with the next-larger size.

The benchmark is implemented as a shell

[3] The only exception to this rule is the communications subsystem, for which a special-purpose allocator (*allocb* and *freeb*) already exists.

[4] Although the machine we were using had 26 CPUs, we cannot reliably measure the performance of all 26 CPUs simultaneously because the script that coordinates the tests must use one of the CPUs.

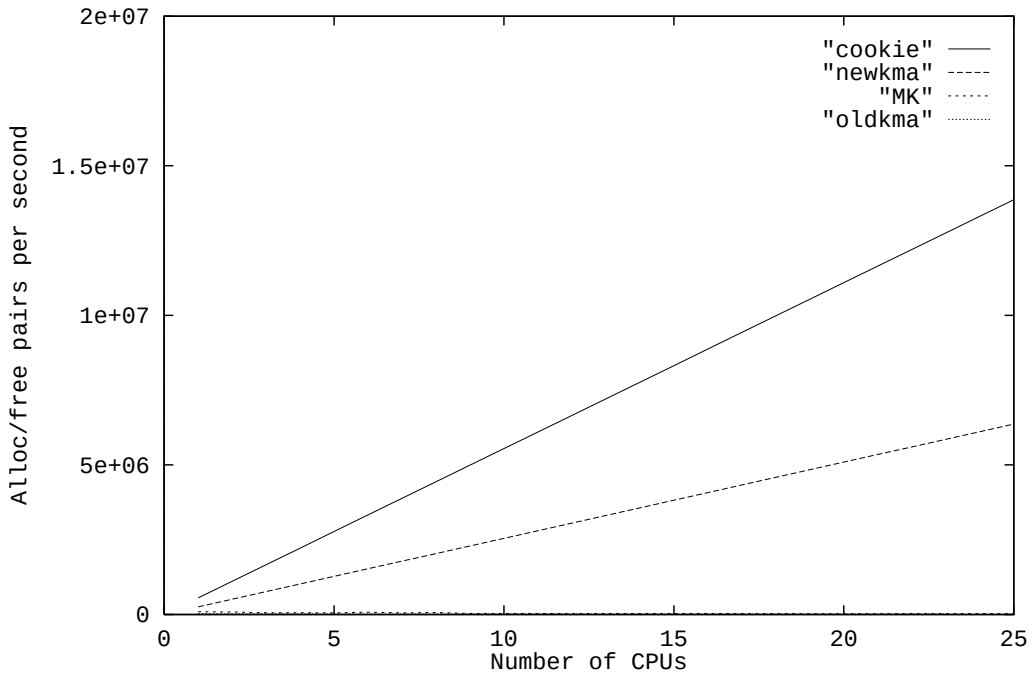

Figure 7: Performance of New *kmem_alloc* and *kmem_free*

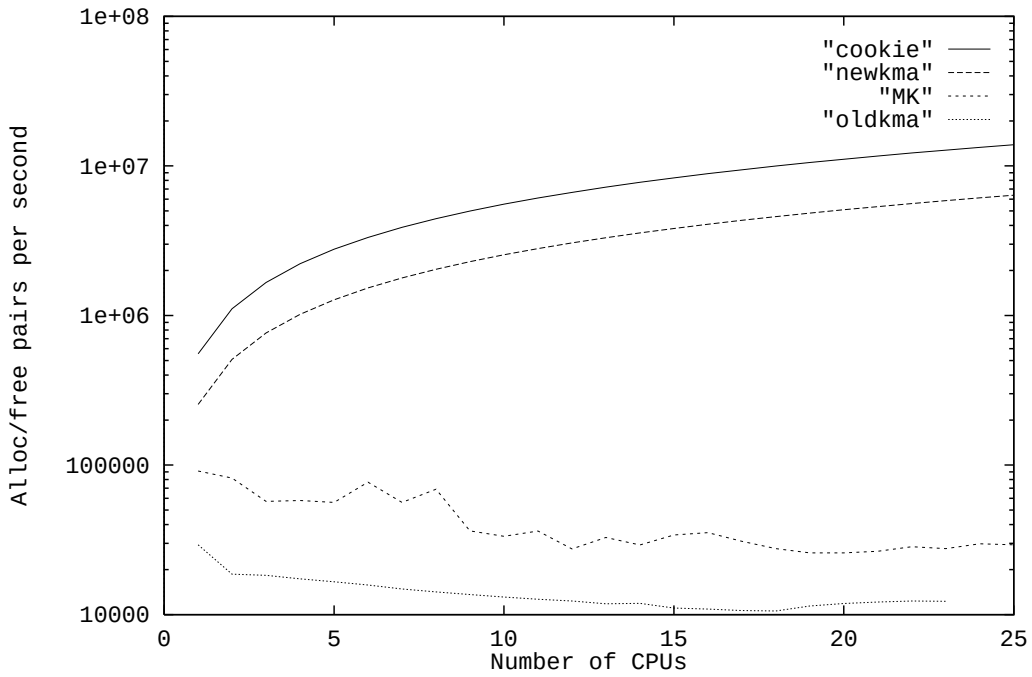

Figure 8: Performance of New *kmem_alloc* and *kmem_free*

script which uses a set of special-purpose system calls which allow the user to explicitly specify sequences of allocation and free operations. A *syscall_kma*() system call causes the system to allocate a specified number of blocks of a given size, placing them on a linked list in the kernel. A companion *syscall_kmf*() system call causes the system to free a specified number of blocks from the linked list.

Note that an allocator that does no coalescing would fail to complete this benchmark, having permanently fragmented all available memory into the smallest possible blocks. It would be necessary to reboot the system between runs of each block size. An allocator that does periodic offline coalescing would require that appropriate *sleep* commands be placed in the script in order to ensure that the newly-freed blocks of the previous size were fully coalesced before advancing to the next size. The fact that our allocator required neither reboots nor delays of any sort demonstrates the effectiveness of the coalescing scheme.

The results are shown on Figure 9. Note that the x-axis is in units of block size rather than number of CPUs. Large blocks showed decreased performance because they require physical memory to be allocated from the virtual-memory system more frequently, and the *target* value is set by a heuristic that limits the amount of memory that is tied up in per-CPU caches. This value ranges from 10 for 16-byte blocks to just 2 for 4096-byte blocks. Although this heuristic may be overridden to increase performance, there is usually little reason to. The overhead of initializing large blocks of memory typically overshadows the virtual-memory system's overhead.

Freeing small blocks is more expensive than allocating them because of the overhead of mapping from the block's address to its per-page freelist. Normally, this overhead would be infrequently incurred, but the worst-case benchmark forces it to occur on each and every free.

4.4 Distributed Lock Manager Benchmark

The best-case benchmark is effectively measuring only the performance of the per-CPU layer, while the worst-case benchmark overstates the overhead of the upper layers. Realistically evaluating the overall performance requires measuring an application that makes more sophisticated use of the memory allocator than did the simple benchmarks presented in the previous sections. The application we selected was a distributed lock manager, which makes heavy use of *kmem_alloc* in order to build data structures needed to track lock requests and ownership. This lock manager is used by OLTP applications to maintain a consistent view of data among a cooperating cluster of machines.

Unfortunately, it is not possible to directly measure the *kmem_alloc* overhead in this benchmark. The microsecond counters used to measure the overhead for the two simple benchmarks do not have enough resolution to accurately measure an isolated invocation of these allocators. However, the degree by which the upper layers will degrade performance can be expressed in terms of miss rates. We define the miss rate at a given layer as the fraction of accesses to that layer that require the services of a higher layer. For the value of 10 used for *target* for small blocks, at most one of every ten allocations will require the services of the global layer. Hence, the maximum miss rate from the per-CPU caching layer is 10%. The value of 15 used for *gbltarget* for small blocks results in a maximum miss rate of 6.7% from the global layer to the coalescing layer. The maximum combined miss rate from the per-CPU and global layers is 0.67%. In other words, at most one out of every 150 allocations will require service from the coalescing layer. Real applications will fall somewhere between the best- and worst-case benchmarks. Measuring a particular application's miss rates allows us to estimate that application's allocation overhead without the need for special-purpose hardware.

The miss rate from the per-CPU layer into the global layer ranged from 2.1% (for frees of 256-byte blocks) to 7.8% (for allocations of 512-byte blocks). Note that the 7.8% figure is fairly close to the worst-case figure of 10%. Again, if need be, the value of *target* can be increased to reduce both the worst-case and the real-world miss rates.

The miss rate from the global layer to the coalesce-to-page layer ranged from 1.2% (for frees of 256-byte blocks) to 3.0% (for allocations of 512-byte blocks). Both these figures compare favorably to the worst-case figure of 6.7%.

The combined miss rate of the per-CPU and global layers to the coalesce-to-page layer ranged from 0.02% (for frees of 256-byte blocks)

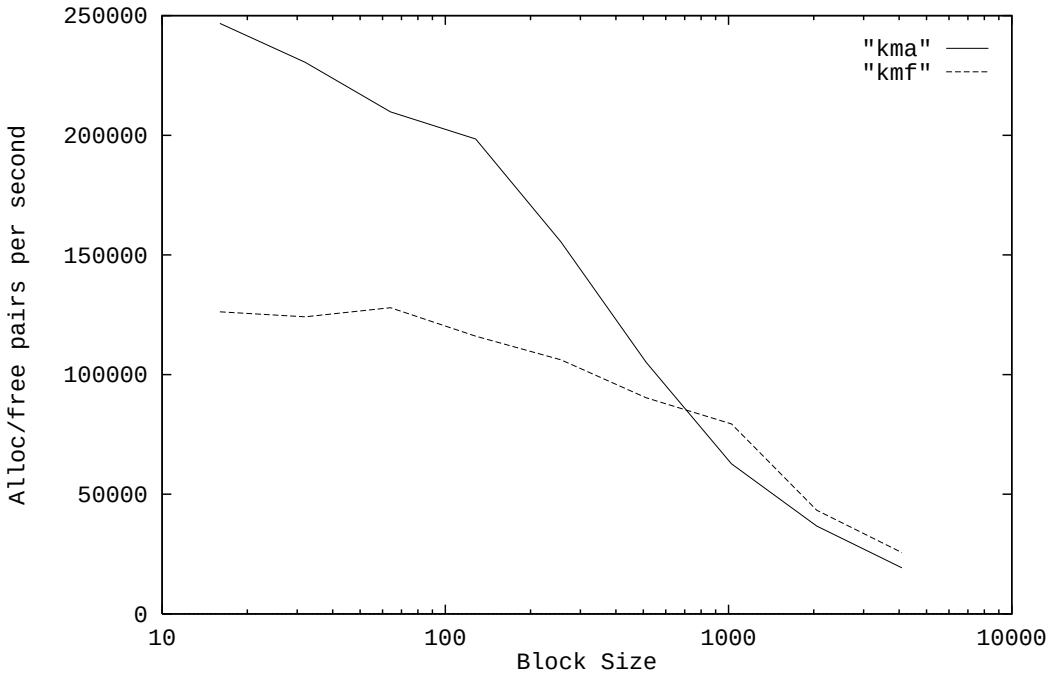

Figure 9: Worst-Case Performance

to 0.14% (for allocations of 512-byte blocks), both of which compare favorably to the worst case of 0.67%. These combined miss rates ensure that the overhead of coalescing is diluted by a factor ranging from 700 to 5000, thereby maintaining an acceptable per-block overhead.

5 Experience

This section describes experience gained in the three years we have been using the new `kmem_alloc`. The following sections show how well the new algorithm has scaled with increasing CPU speeds, present per-layer miss rates observed under a timesharing code-development workload, analyze the performance of the coalescing layer, describe why cookies are not used, and present analysis of expected performance on CC-NUMA architectures.

5.1 Scaling with Faster CPUs

Table 1 shows the overhead of an allocation/free pair on various configurations of CPU and compiler. The Model E uses 50MHz 80486s and the Model 10 uses 100MHz Pentiums. The Pentiums run 1.91 times faster than the 486s, which is close to the difference in clock rate. In principle, it might be possible to get an additional doubling in speed due to the Pentium's dual-pipeline architecture, but we did not take any extreme measures (e.g., hand-coded assembly) to make `kmem_alloc` take advantage of this feature.

5.2 Miss Rates Under Timesharing Code-Development Workload

This algorithm relies heavily on low miss rates at the per-CPU layer to get good performance.

Table 2 displays allocation miss-rate statistics. The "Allocs" column shows the number of allocations for a given size, the "Global" column shows the number of references to the global layer. The "Worst%" column shows the "worst-case" miss rate to the global layer expected from the target value for the corresponding block size. The "%" column shows the actual miss rate. The "Splits" column gives the number of pages that were split into the blocks, and the "%" column gives the miss rate to the

Configuration	af/s	μs/af	speedup
Model E, old comp -O	254704	3.93	
Model E, new comp	290653	3.44	1.14 (comp)
Model E, new comp -O	315062	3.17	1.09 (opt) 1.24 (opt+comp)
Model 10, new comp	541145	1.85	1.86 (CPU)
Model 10, new comp -O	604164	1.66	1.91 (CPU) 1.11 (opt) 2.37 (CPU+comp)

Table 1: Allocation Miss Rates

coalescing layer (in other words, the percentage of global-layer allocation requests that resulted in a page being split). The "Allocs/Split" column gives then number of per-CPU layer requests for each page split.[5]

The measured miss rates are excellent, particularly for the 32-byte requests that make up well over 50% of the requests. These 32-byte request literally have a one-in-a-million chance of resulting in an expensive page-split operation. The unexpectedly-high miss rates for 512-byte requests are discussed below.

Table 3 shows the analogous free-side miss-rate statistics. The "Frees" column gives the number of free requests, the "Global" column gives the number that reach the global layer, the "Worst%" column gives the worst-case miss rate based on the target value, the "%" column gives the actual measured miss rate, the "Coalesces" column gives the number of times that blocks are coalesced back into pages, the "%" column gives the fraction of global-layer frees the result in coalescing, and the "Frees/Coalesce" gives that number of per-CPU layer requests per page coalesced.

Again, the miss rates are excellent, particularly for the 32-byte requests.

Note that a list of elements is freed to the global layer, and that a page's worth of elements are coalesced. It is therefore possible for more pages to be coalesced than lists freed to the global layer, the average list freed to the global layer contains more than one page's worth of memory. This situation actually occurred in the 4096-byte case, where roughly four pages were "coalesced" for each list freed to the global layer.

Note also that the actual miss rate to the global layer exceeded the worst-case for allocation of 512-byte blocks. The reason for this is that the per-engine caches are periodically flushed. If the free rate is low enough, this flushing will significantly increase the allocation-side miss rate to the global layer, while decreasing the free-side miss rate to the global layer. Normally, increased miss rate would be cause for concern. However, Table 4 shows that increased miss rate is seen only for those sizes that have low free rates. Therefore, the increased miss rates cause insignificant degradation of overall system throughput.

size	frees/s	target	%flushes	s/gblfree
16	9.1	20	99.6	77.8
32	225.7	20	66.1	50.6
64	25.1	20	25.0	19.0
128	12.1	20	72.2	54.2
256	13.0	20	69.6	51.7
512	1.7	16	99.8	89.8
1024	30.6	8	30.7	35.8
2048	1.7	4	68.4	65.6
4096	2.4	2	20.8	21.6
total	321.4			

Table 4: Per-CPU Cache Flush Rate

Since a given CPU's cache is flushed every 72 seconds, at least one target's worth of blocks must be freed up on a given CPU in 72 seconds to have any chance to successfully free to the global layer rather than flushing. In particular, the 512-byte case rarely gets a chance to free.

However, simply freeing at least a target's worth of memory blocks every 72 seconds is not enough to ensure that most blocks will be freed to the global instead of being flushed. In fact the 512- and 2048-byte pools have identical free rates, but very different flush rates. A very high hit rate in the per-CPU caches (such as enjoyed by the 16- and 32-byte sizes) can result in a high flush rate. The 2048-byte pools has a low target

[5] This is not given as a percentage because the author did not want to wear out the "0" key on his terminal.

Size	Allocs	Global	Worst%	%	Splits	%	Allocs/Split
16	5,798,363	68,089	5.00	1.17	75	0.11	77,311.5
32	143,785,499	255,802	5.00	0.18	112	0.04	1,283,799.1
64	15,997,147	310,406	5.00	1.94	1,661	0.54	9,631.0
128	7,679,703	279,929	5.00	3.65	479	0.17	16,032.8
256	8,315,199	372,936	5.00	4.49	3,559	0.95	2,336.4
512	1,070,258	117,040	6.25	10.94	1,595	1.36	671.0
1024	19,502,765	417,908	12.5	2.14	86,486	20.70	225.5
2048	1,056,950	217,144	25.00	20.54	11,811	5.44	89.5
4096	1,547,864	353,740	50.00	22.85	319,141	90.22	4.9
Total	204,753,748	2,392,994	N/A	1.17	424,919	17.76	481.9

Table 2: Allocation Miss Rates

Size	Frees	Global	Worst%	%	Coalesces	%	Frees/Coalesce
16	5,784,730	65,134	5.00	1.13	19	0.03	304,459.5
32	143,777,631	104,939	5.00	0.07	10	0.01	14,377,763.1
64	15,996,564	269,449	5.00	1.68	1,631	0.61	1,631.8
128	7,677,711	93,402	5.00	1.22	332	0.36	23,125.6
256	8,309,413	98,731	5.00	1.19	3,056	3.10	2,719.0
512	1,069,240	55,987	6.25	5.24	1,347	2.41	793.8
1024	19,501,125	143,115	12.5	0.73	85,761	59.92	227.4
2048	1,056,100	75,674	25.00	7.17	11,232	14.84	94.0
4096	1,547,833	238,700	50.00	15.42	319,105	133.68	4.9
Total	204,720,347	1,145,131	N/A	0.56	422,493	36.89	484.6

Table 3: Free Miss Rates

value, which results in a low hit rate, and thus a relatively low flush rate.

The best predictor of flush rate is the time required for a given CPU to free to global, shown in the "s/gblfree" column of Table 4. The sizes requiring more than 72 seconds to free to global have very high flush rates.

Since 32-byte allocations have the highest per-CPU cache hit rates and account for over 70% of the total allocations, we have best performance where most important.

5.3 Performance of Coalescing Layer

The coalescing layer splits pages into smaller blocks, and does not release the page back to the system until all of its blocks have been freed up. Knuth concluded long ago [Knu73] that this type of allocator is not particularly memory efficient in the general case.

Although, this allocator was not designed for the general case (it was designed for a cyclical workload with drastic changes in memory usage), it is interesting to measure its performance for a timesharing code-development workload.

size	Kwaste	Kinuse	%waste
16	10.4	224	4.63867
32	160.1	408	39.2387
64	68.2	120	56.8229
128	319.4	588	54.3155
256	524.8	2012	26.081
512	479.5	992	48.3367
1024	1260.0	2900	43.4483
2048	560.0	2316	24.1796
4096	0.0	144	0
total	3382.3	9706	34.8

Table 5: Coalescing Layer Performance–Low Load

At first glance, the 34.8% overall memory waste shown in Table 5 does little to dispute Knuth's view. However, this 34.8% wastage is nowhere near the worst-case 50% wastage for 2048-bytes blocks, to say nothing of the 99.6% wastage possible with 16-byte blocks. Further, this data was taken during a period of low load immediately following a high-load period.

Data taken during a period of high load,

size	Kwaste	Kinuse	%waste
16	2.5	336	0.7
32	12.0	452	2.7
64	69.4	136	51.0
128	150.4	744	20.2
256	295.0	2244	13.1
512	58.5	1116	5.2
1024	609.0	3804	16.0
2048	66.0	3908	1.7
4096	0.0	224	0.0
total	1262.4	12,964	9.7

Table 6: Coalescing Layer Performance–Heavy Load

shown in Table 6, has a much better 9.7% wastage. Although the amount of memory wasted increased when the load decreased, over a megabyte of memory was returned to the system.

This data shows that the allocator works quite well in a traditional timesharing environment. However, more work will be required to produce an algorithm suitable for systems that have insufficient memory.

5.4 Cookies

Although cookie-base allocation is considerably faster than the standard subroutine-call allocation, it was never implemented or used in a released version of PTX. The reasons for this are quite interesting–it turns out that the measured speedup for cookie-based allocation comes only at the price of an overall reduction in system performance!

The reason for this is that cookie-based allocation increases the size of the kernel, resulting in more cache misses on instruction fetches. Cookie-based allocation requires 24 bytes of code, compared to just 9 bytes for a `kmem_alloc` call,[6] for a 15-byte increase. Similarly, cookie-base free requires 20 bytes of code, compared to just 8 bytes for a `kmem_free` call for a 12-byte increase.

The most recently-released version of PTX has 130 calls to `kmem_alloc` and 257 calls to `kmem_free`. Use of cookie-based allocation would therefore result in a 5034-byte increase in kernel size, which in turn results in increase instruction-fetch miss rates that cancel any performance advantage from cookie-based allocation.

Applications or kernels which do almost all of their allocation/deallocation through a few specific allocation/free calls and which spend much of their time allocating and deallocating are more likely to benefit from cookie-based allocation.

[6]This assumes that the size of the block being allocated is 255 bytes or less, larger allocations require an additional byte up to 65535 bytes. Larger allocations are extremely rare and require an additional two bytes.

5.5 CC-NUMA Expectations

Cache-coherent non-uniform-memory-access machines such as the Stanford FLASH [HKO+94] use a hierarchical bus structure that gives a particular CPU low-overhead access to memory "close" to that CPU. This suggests that an allocator that attempts to give CPUs memory that is "close" might boost performance. This section analyzes such an allocator, showing that any significant improvement is likely to required that either the deallocator or the users of the allocator must also be modifed to promote locality.

A simple mixing model is shown in Figure 10. Arc "A" represents memory being allocated, arc "B" represents memory moving from the global to the per-CPU caches, arc "C" represents memory moving from the coalescing layer to the global cache, arc "D" represents memory being freed, arc "E" represents memory moving from the per-CPU to the global caches, and arc "F" represents memory moving from the global cache to the coalescing layer.

Memory from the coalescing layer is assumed to consist purely of memory closest to the CPU making the allocation request. Memory freed by the callers of `kmem_free` is assumed to have "purity" p_f, in other words, p_f fraction of it is memory close to the CPU doing the `kmem_free` and $1-p_f$ is from memory close to other CPUs. The allocation and free rates λ are assumed equal, as they must be over long time periods. The hit rate at both the per-CPU and global caches is assumed to be r for both allocation and free. Then the purity of the per-CPU caches p_e and the purity of the global pool p_g is given by the following system of equations:

$$p_e = \frac{\lambda r p_g + \lambda p_f}{\lambda r + \lambda} \quad (1)$$

$$p_g = \frac{\lambda r p_e + \lambda r^2}{\lambda r + \lambda r^2} \quad (2)$$

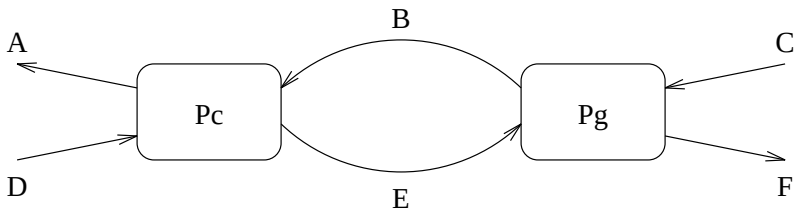

Figure 10: CC-NUMA Memory Mixing Model

The λs cancel:

$$p_e = \frac{rp_g + p_f}{r+1} \tag{3}$$

$$p_g = \frac{p_e + r}{r+1} \tag{4}$$

Solving for p_e and p_g:

$$p_e = \frac{r^2 + p_f r + p_f}{r^2 + r + 1} \tag{5}$$

$$p_g = \frac{r^2 + r + p_f}{r^2 + r + 1} \tag{6}$$

An efficiently-configured allocator will have a low miss rate r, which results in both the per-engine caches and global pool having purities roughly equal to that of the memory being `kmem_free`ed. Therefore, if the memory allocated from the per-CPU caches is to be pure, the memory freed to these caches must also be pure.

6 Conclusions

The new *kmem_alloc* and *kmem_free* functions meet their design goals. These goals are achieved by avoiding synchronization, by taking advantage of cache locality (rather than through use of sophisticated synchronization schemes), and by maintaining low miss rates at the per-CPU and global layers so as to dilute the overhead inherent in coalescing.

These functions are more than capable of meeting the challenge of commercial data processing. They also clearly demonstrate that the problem of efficient resource allocation on a shared-memory multiprocessor is quite tractable.

Experience has shown that these functions scale reasonably well with increasing processor generations, and that they work well in a time-sharing software-development environment.

References

[BK91] N. S. Barghouti and G. E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys:*, September 1991.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 11(1):124–149, January 1991.

[Her93] Maurice Herlihy. Implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[HJ91] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, pages 18–28, September 1991.

[HKO+94] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The performance impact of flexibility in the stanford flash multiprocessor. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[Knu73] Donald Knuth. *The Art of Computer Programming.* Addison-Wesley, 1973.

[KV85] David G. Korn and Kiem-Phong Vo. In search of a better malloc. In *USENIX Conference Proceedings*, Berkeley CA, June 1985.

[LB89] T. Paul Lee and R. E. Barkley. Design and evaluation of a watermark-based lazy buddy system. *Performance Evaluation Review*, 17(1), May 1989.

[MG92] Paul E. McKenney and Gary Graunke. Efficient buffer allocation on shared-memory multiprocessors. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Tucson, AZ, February 1992.

[MK88] Marshall Kirk McKusick and Michael J. Karels. Design of a general purpose memory allocator for the 4.3BSD UNIX kernel. In *USENIX Conference Proceedings*, Berkeley CA, June 1988.

[MS93] Paul E. McKenney and Jack Slingwine. Efficient kernel memory allocation on shared-memory multiprocessors. In *USENIX Conference Proceedings*, Berkeley CA, February 1993.

[Mye92] Nathan Myers. C++ memory management: An overview. Message-ID 9210131855.AA24066@rwave.roguewave.com, October 1992.

[Rit84] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, October 1984.

[SC91] Harold S. Stone and John Cocke. Computer architecture in the 1990s. *IEEE Computer*, pages 30–38, September 1991.

[SSHT93] Janice S. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple reservations and the oklahoma update. *IEEE Parallel and Distributed Technology Systems and Applications*, 1(4):58–71, November 1993.

[Ste83] C. J. Stephenson. Fast fits: New methods for dynamic storage allocation. *SIGOPS Operating System Review*, 17(5), 1983.

[TGH92] Josep Torrellas, Anoop Gupta, and John Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *ASPLOS V*, October 1992.