Network Buffer Allocation in the FreeBSD Operating System [May 2004]

Bosko Milekic <bmilekic@FreeBSD.org>

Abstract

This paper outlines the current structure of network data buffers in FreeBSD and explains their allocator's initial implementation. The current common usage patterns of network data buffers are then examined along with usage statistics for a few of the allocator's key supporting API routines. Finally, the improvement of the allocation framework to better support Symmetric-Multi-Processor (SMP) systems in FreeBSD 5.x is outlined and an argument is made to extend the general purpose allocator to support some of the specifics of network data buffer allocations.

1 Introduction

The BSD family of operating systems is in part reputed for the quality of its network subsystem implementation. The FreeBSD operating system belongs to the broader BSD family and thus contains an implementation of network facilities built on top of a set of data structures still common to the majority of BSD-derived systems. As the design goals for more recent versions of FreeBSD change to better support SMP hardware, so too must adapt the allocator used to provide network buffers. In fact, the general-purpose kernel memory allocator has been adapted as well and so the question of whether to continue to provide network data buffers from a specialized or general-purpose memory allocator naturally arises.

2 Data Structures

FreeBSD's network data revolves around a data structure called an Mbuf. The allocated Mbuf data buffer is fixed in size and so additional larger buffers are also defined to support larger packet sizes. The Mbuf and accompanying data structures have been defined with the idea that the usage patterns of network buffers substantially differ from those of other general-purpose buffers [1]. Notably, network protocols all require the ability to construct records widely varying in size and possibly throughout a longer period of time.

Not only can records of data be constructed on the fly (with data appended or prepended to existing records), but there also exist well-defined consumer-producer relationships between network-bound threads executing in parallel in the kernel. For example, a local thread performing network inter-process-communication (IPC) will tend to allocate many buffers, fill them with data, and enqueue them. A second local thread performing a read may end up receiving the record, consuming it (typically copying out the contents to user space), and promptly freeing the buffers. A similar relationship will arise where certain threads performing receive operations (consuming) largely just read and free buffers and other threads performing transmit operations (producing) largely allocate and write buffers.

The Mbuf and supporting data structures allow for both the prepending and appending of record data as well as data-locality for most packets destined for DMA in the network device driver (the goal is for entire packet frames from within records to be stored in single contiguous buffers). Further justification for the fixed-size buffer design is provided by the results in Section 4.

2.1 Mbufs

In FreeBSD, an Mbuf is currently 256 Bytes in size. The structure itself contains a number of general-purpose fields and an optional internal data buffer region. The general-purpose fields vary depending on the type of Mbuf being allocated.

Every Mbuf contains an *m_hdr* substructure which includes a pointer to the next Mbuf, if any, in an Mbuf chain. The *m_hdr* also includes a pointer to the first Mbuf in the next packet or record's Mbuf chain, a reference to the data stored or described by the Mbuf, the length of the stored data, as well as flags and type fields.

In addition to the *m_hdr*, a standard Mbuf contains a data region which is *(256 Bytes - sizeof(struct m_hdr))* long. The structure differs

for a *packet-header* type Mbuf which is typically the first Mbuf in a packet Mbuf chain. Therefore, the *packet-header* Mbuf usually contains additional generic header data fields supported by the *pkthdr* substructure. The *pkthdr* substructure contains a reference to the underlying receiving network interface, the total packet or record length, a pointer to the packet header, a couple of fields used for checksumming, and a header pointer to a list of Mbuf meta-data link structures called *m_tags*. The additional overhead due to the *pkthdr* substructure implies that the internal data region of the *packet-header* type Mbuf is *(256 Bytes - sizeof(struct m_hdr) - sizeof(struct pkthdr))* long.

The polymorphic nature of the fixed-size Mbuf implies simpler allocation, regardless of type, but not necessarily simpler construction. Notably, *packet-header* Mbufs require special initialization at allocation time that other ordinary Mbufs do not, and so care must be taken at allocation time to ensure correct behavior. Further, while it is theoretically possible to allocate an ordinary Mbuf and then construct it at a later time in such a way that it is used as a *packet-header*, this behavior is strongly discouraged. Such attempts rarely lead to clean code and, more seriously, may lead to leaking of Mbuf meta-data if the corresponding *packet-header* Mbuf's *m_tag* structures are not properly allocated and initialized (or properly freed if we're transforming a *packet-header* Mbuf to an ordinary Mbuf).

Sometimes it is both necessary and advantageous to store packet data within a larger data area. This is certainly the case for large packets. In order to accommodate this requirement, the Mbuf structure provides an optional *m_ext* substructure that may be used instead of the Mbuf's internal data region to describe an external buffer. In such a scenario, the reference to the Mbuf's data (in its *m_hdr* substructure) is modified to point to a location within the external buffer, the *m_ext* header is appropriately filled to fully describe the buffer, and the M_EXT bit flag is set within the Mbuf's flags to indicate the presence of external storage.

2.2 Mbuf External Buffers

FreeBSD's Mbuf allocator provides the shims required to associate arbitrarily-sized external buffers with existing Mbufs. The optionally-used *m_ext* substructure contains a reference to the base of the externally-allocated buffer, its size, a type descriptor, a pointer to an unsigned integer for reference counting, and a pointer to a caller-specified free routine along with an optional pointer to an arbitrarily-typed data structure that may be passed by the Mbuf code to the specified free routine.

The caller who defines the External Buffer may wish to provide an API which allocates an Mbuf as well as the External Buffer and its reference counter and takes care of configuring the Mbuf for use with the specified buffer via the *m_extadd()* Mbuf API routine. The *m_extadd()* routine allows the caller to immediately specify the free function reference which will be hooked into the allocated Mbuf's *m_ext* descriptor and called when the Mbuf is freed, in order to provide the caller with the ability to then free the buffer itself. Since it is possible to have multiple Mbufs refer to the same underlying External Buffer (for shared-data packets), the caller-provided free routine will only be called when the last reference to the External Buffer is released. It should be noted that once the reference counter is specified by the caller via *m_extadd()*, a reference to it is immediately stored within the Mbuf and further reference counting is entirely taken care of by the Mbuf subsystem itself without any additional caller intervention required.

Reference counting for External Buffers has long been a sticky issue with regards to the network buffer facilities in FreeBSD. Traditionally, FreeBSD's support for reference counting consisted of providing a large sparse vector for storing the counters from which there would be a one-to-one mapping to all possible Clusters allocated simultaneously (this limit has traditionally been NMBCLUSTERS). Thus, accessing a given Cluster's reference count consisted of merely indexing into the vector at the Cluster's corresponding offset. Since the mapping was one-to-one, this always worked without collisions. As for other types of External Buffers, their reference counters were to be provided by the caller via *m_extadd()* or auto-allocated via the general-purpose kernel *malloc()* at ultimately higher cost.

Since the traditional implementation, various possibilities have been considered. FreeBSD 4.x was modified to allocate counters for all types of

External Buffers from a fast linked list, similar to how 4.x still allocates Mbufs and Mbuf Clusters. The advantage of the approach was that it provided a general solution for all types of buffers. The disadvantage was that allocating a counter was required for each allocation requiring external buffers. Another possible implementation would mimick the NetBSD [2] and OpenBSD [10] Projects' clever approach, which is to interlink Mbufs referring to the same External Buffer. This was both a reasonably fast and elegant approach, but would never find its way to FreeBSD 5.x due to serious complications that result because of lock ordering (with potential to lead to deadlock).

The improvements to reference counting made in later versions of FreeBSD (5.x) are discussed in sections 5.2 and 5.4.

2.3 Mbuf Clusters

As previously noted, Mbuf Clusters are merely a general type of External Buffer provided by the Mbuf subsystem. Clusters are 2048 bytes in size and are both virtual-address and physical-address contiguous, large enough to accommodate the ethernet MTU and allow for direct DMA to occur to and from the underlying network device.

A typical socket storing connection data references one or more Mbuf chains some of which usually have attached Mbuf Clusters. On a send, data is usually copied from userland to the Mbuf's Cluster (or directly to the Mbuf's internal data region, if small enough). The Mbuf chain is sent down through the network stack and finally handed off to the underlying network device which takes care of uploading (usually DMA) the contents directly from the Cluster, or Mbuf internal data region, to the device. A typical receive usually involves an interrupt from a network device which then results in an interrupt handler being run. The device's interrupt handler downloads a frame into a Cluster attached to an Mbuf and queues the packet. The traditional FreeBSD implementation schedules a software interrupt handler to dequeue the packet (Mbuf chain) and run it through the stack only to finally buffer the data into the socket (this is implemented by hooking the now-manipulated Mbuf chain into the socket's receive buffer). The usual send and receive data flows are illustrated in *Figure 1*.

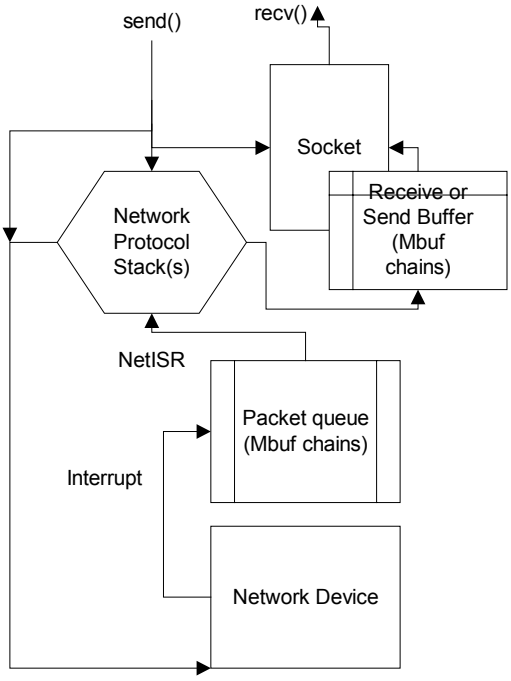

Figure 1: Data Flows for Typical Send & Receive

Several optimizations have been made to improve performance in certain scenarios. Notably, a sendfile-buffer, another External Buffer type, is designed to allow data to be transferred without requiring the relatively expensive userland-to-kernel copy. The sendfile buffer implementation is beyond the scope of this paper. The curious reader should refer to [3].

3 Implementation in Pre-5.x FreeBSD

The traditional pre-5.x Mbuf allocator, although simple, had a number of important shortcomings, all the result of optimizations which only considered single-processor performance and ignored the possibility of multi-processor scalability within the OS kernel.

The FreeBSD 4.x (and prior) network buffer allocator used global singly-linked lists for maintaining free Mbufs, Clusters, and reference counters for External Buffers. While the advantages of this approach are simplicity and speed in the single-processor case, its inability to reclaim resources and potential to bottleneck in

SMP setups in particular makes it a less appealing choice for FreeBSD 5.x.

Traditionally, protection needed to be ensured only from interrupt handlers which could potentially preempt in-kernel execution. This was accomplished via optimized interrupt class masking. All Mbuf and Cluster allocations requiring the use of the allocator's freelists thus temporarily masked all network device and stack interrupts via the *splimp()* call in FreeBSD 4.x. The *spl*()* interface has been deprecated in FreeBSD 5.x.

4 Usage Patterns

A common argument, when comparing BSD's Mbuf and Cluster model to other designs such as for example Linux's SKBs (Socket Buffers, see [4]) is that BSD's Mbufs result in more wasted space than SKBs and that SKBs allow for faster operations by always ensuring to pre-allocate worst-case amount of headspace prior to their data regions. The observations presented in this section aim to dispell these myths.

Linux's SKB is a structure with additional buffer space allocated at its end according to initial size requirements. The linear SKB requires the specification of total buffer length at allocation time. The suggestion (notably by [4]) is to allocate additional space at the beginning of the SKB in order to accommodate any future headspace that may be required as the SKB is pushed up or down the stack. For a request of *len* bytes, for example, we may wish to actually allocate *len* + *headspace* bytes for the SKB. If the requested buffer sizes (*len*) are all relatively close in magnitude to each other, the variously-sized SKB allocation requests will result in calls to the underlying Linux slab allocator for allocation lengths differing by small amounts. The allocator's general object caches and slabs are typically organized according to requested object size ([5], [8]) and, depending on the requested length and on how accurate the initial estimate of required headspace was, may actually result in a comparable amount of wasted space to the scenario where a fixed-size Cluster is used. Wastage is a natural side-effect of using general purpose caches not designed to store objects of a given fixed size, but which must accommodate objects of a wide variety of sizes. The difference is that in the Linux SKB case, the wastage is not immediately apparent because it is accounted for by tradeoffs in the underlying allocator's design.

It is certainly true that for every 2048-Byte Cluster, the corresponding Mbuf's internal data region is left unused and thus results in wasted space. For a Cluster of 2048 bytes, the internal Mbuf wastage is approximately 6%. This 6% wastage is however constant regardless of the accuracy of headspace estimates and selected object cache, because all Mbufs and Clusters are allocated from specific Mbuf and Cluster zones (with their own object-specific slabs), significantly reducing internal fragmentation and wastage associated with general purpose schemes. It is also true that for a wide-range of requested lengths, the unused space left in fixed-sized Clusters would increase the proportion of wastage. However, if packet size distributions are bimodal, then this is much less of an issue than one may be led to expect.

The second common argument is that the preallocation of headspace within SKBs leads to better performance due to not having to "pullup" an Mbuf chain which may have had a small Mbuf prepended to it while on its way through the stack. Indeed, because the BSD model does not require the initial reservation of space within the Mbuf (as one is always allowed to prepend another Mbuf to the chain, if required), should the stack at any point need to ensure a certain length of data to be contiguous at the beginning of the chain, a call to *m_pullup()* must be made and may occasionally result in data copying. However, if the calls to *m_pullup()* are kept out of the common case, then the total number of pullups is small and overall performance does not suffer. Protocol-specific socket send routines often also prepend space within the first allocated Cluster if they know that it will be needed, and thus significantly reduce the number of calls to *m_pullup()* requiring data copying.

Some real-world statistics on packet size distributions and m_pullup() usage are presented below.

4.1 Packet Size Distributions

As mentioned above, using fixed-size small and large buffers (Mbufs and larger External Buffers such as Clusters) is appropriate when most of the packets are either large or small.

Packet size distributions tend to be bimodal. A typical packet size distribution collected on a NAT gateway machine running FreeBSD is shown in *Figure 2*.

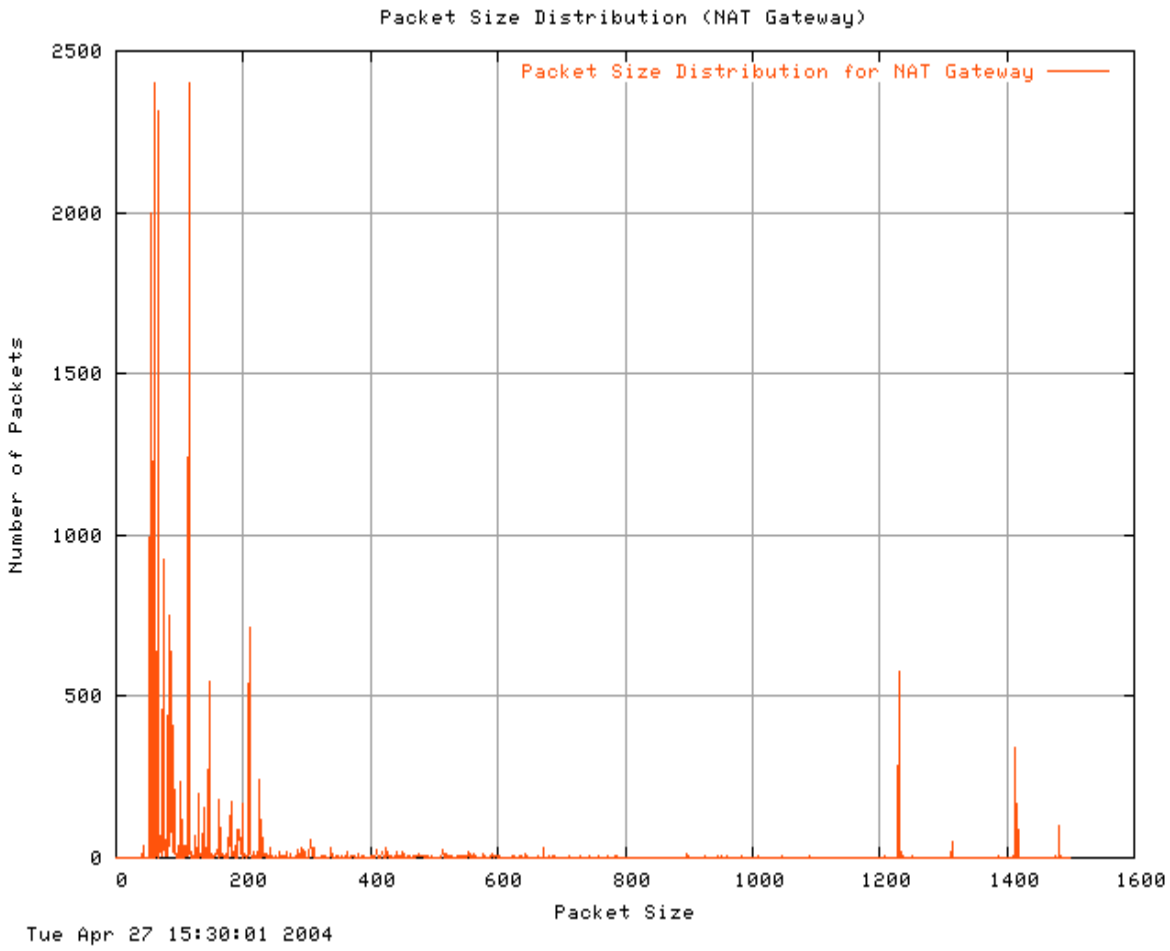

Figure 2: Packet Size Distribution For Typical NAT Gateway

4.2 API Usage Statistics

The number of *m_pullup()* calls resulting in data copying may affect the effectiveness of the Mbuf model. The results in *Figure 3* show API usage statistics for *m_pullup()*.

m_pullup()	m_clget()	m_getcl()
0	954	14239

Figure 3: Number of Calls to Various API Routines for Several Days of Uptime (no INET6)

Additionally, the same set of statistics reveal that the number of calls to *m_getcl()* (to allocate both an Mbuf and a Cluster atomically) is much greater than the number of calls to *m_clget()* (to allocate a single Cluster when an Mbuf is already available). The results shown are from a system where the *sosend()* code was modified to use *m_getcl()* whenever possible and take advantage of the new allocator's Packet cache (see section 5.4).

5 FreeBSD 5.x and Beyond: Integrating Network Buffer Allocation in UMA

A primary goal of FreeBSD 5.x is multi-processor scalability. The ability to run kernel code in parallel from different process contexts offers some exciting possibilities with regards to scalability. Whereas the previous Mbuf-specific allocator performed very adequately in the single-processor scenario, it was not designed with multi-processor scalability in mind, and so as the number of processors is increased, it has the potential to bottleneck high network performance. The primary purpose of the new allocator was, therefore, to allow network buffer allocations to scale as the number of CPUs is increased while maintaining comparable performance for the single-processor case which is still extremely common in today's server configurations.

The development that led to the present-day network buffer allocator in FreeBSD was essentially done in two steps. The first step (from now on referred to as *mballoc*) consisted in the rewriting of the network buffer-specific (Mbuf and Mbuf Cluster) allocator to better handle scalability in an SMP setup. The second and latest step (from now on referred to as *mbuma*) is a merging of those initial efforts with a new Universal Memory Allocation (UMA) framework initially developed by Jeff Roberson for the FreeBSD Project. While the initial effort to write an SMP-friendly Mbuf allocator was succesful, the most recent change significantly minimizes code duplication and instead extends UMA to handle network buffer allocations with requirements initially intended for Mbuf and Mbuf Cluster specific allocations, while only minimally interfering with general purpose UMA allocations.

5.1 Facilities in 5.x and Later

FreeBSD 5.x provides several facilities for ensuring mutual exclusion within the kernel. Their implementation details are beyond the scope of this paper (for a good description see [6]).

The mutex locks currently provided by FreeBSD require sleep-blocking through the scheduler at worst and a bus-locked or serialized instruction at best. The cost of a non-pipelined or serializing or, even worse (on older processors), bus-locked instruction can be much higher than that of a typical and easily pipelined instruction, particularly on architectures with long pipelines (Refer to Intel [7], for example).

In addition to sleepable mutex locks, FreeBSD 5.x provides a spinlock which also requires an atomic instruction in the best case and spinning with a hard interrupt disable in the worst case. A way to ensure mutual exclusion local to a single CPU also exists and consists of a basic critical section which, unless nested, currently requires a hard interrupt disable.

There is currently ongoing work on optimizing the common case of a critical section to a simple and easily pipelined interlock against the executing CPU which would defer the hard interrupt disable to a first occurance of an interrupt. Along with this work and FreeBSD 5.x scheduler's ability to temporarily pin down a thread to a CPU (an operation which should also only involve a simple pipelined interlock on the executing processor), per-CPU structures such as those involved in a common-case UMA allocation can be made lock-less and therefore faster in the common case.

5.2 Mballoc: an Mbuf-Specific SMP-Aware Allocator

The first major step in the development of an SMP-aware Mbuf allocator consisted in a major re-write of the existing Mbuf-specific allocator (described in Section 3).

The advantages of a Mbuf-specific allocator were more significant at the implementation time of *mballoc* over two years ago. At that time, the existing general-purpose memory and zone allocators were themselves not SMP-aware and used heavily contended global structures. Therefore, the move to per-CPU allocation buckets for at least Mbuf and Cluster allocations was significant for what concerns the SMP efforts at the time, although its true scalability advantages remained (although today much less so) on the unwinding of the kernel BGL (Big Giant Lock), a lock forcing synchronization on kernel entry points until the underlying structures have been finer-grain locked.

The *mballoc* allocator implemented slab-like structures which it called buckets, as well as both global and per-CPU caches of buckets for Mbufs

and Mbuf Clusters. Since UMA itself implements similar structures, but with additional flexibility -- such as the ability to reclaim resources following large demand spikes -- additional work on an Mbuf-specific allocator such as *mballoc* would be retired only if UMA could be extended to properly support Mbuf and Cluster allocations which are significantly different from other system structures.

Therefore, the primary goal of the second development effort which resulted in *mbuma* (described in section 5.4) was to provide new functionality to the Mbuf system, in particular resource reclamation, while minimizing code duplication, providing a better architecture, and while not significantly sacrificing overall network performance.

5.3 The UMA Framework

In order to better understand the extensions and changes made to UMA to adequately support network buffer allocations, it is important to conceptually understand UMA's original design. In particular, the relationships amongst objects within the UMA framework are a key component to understanding how to efficiently setup UMA Zones.

UMA is in its simplest form an implementation of a Slab allocator (see [8]). In addition to providing a low-level slab allocator, UMA additionally provides per-CPU caches as well an intermediate bucket-cache layer separating the per-CPU caches from the backing slab cache for a given object type.

Most objects allocated from UMA come from what are called UMA Zones. In fact, the only current exception are certain types of general-purpose kernel *malloc()* allocations which bypass the UMA Zone structure (and associated bucket caches) and directly access UMA's backing slab structures containing references to allocated pages. A UMA Zone is thus created for a specific type of object with a specific size such as, for example, a Socket. Allocations for Sockets will then occur from the created Socket Zone and should result, in the common case, in a simple allocation from a per-CPU cache.

The per-CPU caches for all Zones are currently protected by a common set of per-CPU mutex locks. As described in section 5.1, mutex locks can be quite expensive even in the common case and work is being done to evaluate whether replacing them with an optimized critical section and short-term CPU-pinning of threads will result in noticeable performance improvement. The UMA Zone itself is protected by a per-Zone mutex lock. Since the common case allocations are supposed to occur from per-CPU caches, contention on the Zone lock is not a major concern.

Additionally, the UMA Zone provides access to two pairs of function pointers to which the caller is free to hook custom functions. The first pair is the constructor/destructor pair (*ctor/dtor*) and the second is the initializer/finalizer (*init/fini*) pair. Together, the *ctor/dtor* and *init/fini* pairs allow the caller to preemptively configure objects as they make their way through the allocator's caches. UMA ensures that no UMA-related locks are held when the *ctor/dtor* or *init/fini* are called.

The initializer is called to optionally configure an object as it is first allocated and placed within a slab cache within the Zone. The finalizer is called when the object is finally freed from the slab cache and its underlying page is handed back to the Virtual-Memory (VM) subsystem. Thus, the objects that reside within the Zone (whether in the slabs or buckets) are all custom configured prior to entering or leaving the caches.

The constructor is called on the object once it has left one of the Zone's caches and is about to be handed off to the caller. Finally, the destructor is called on the object as it is returned by the caller and prior to it entering any of the Zone's caches.

The *ctor/dtor* and *init/fini* facilities allow the caller to optionally allocate additional structures and cache objects in already preconfigured or partially configured states thus minimizing the common case allocation which merely results in a constructor (but not initializer) call or the common case free which merely results in a destructor (but not finalizer) call.

A typical Zone setup is shown in *Figure 4*.

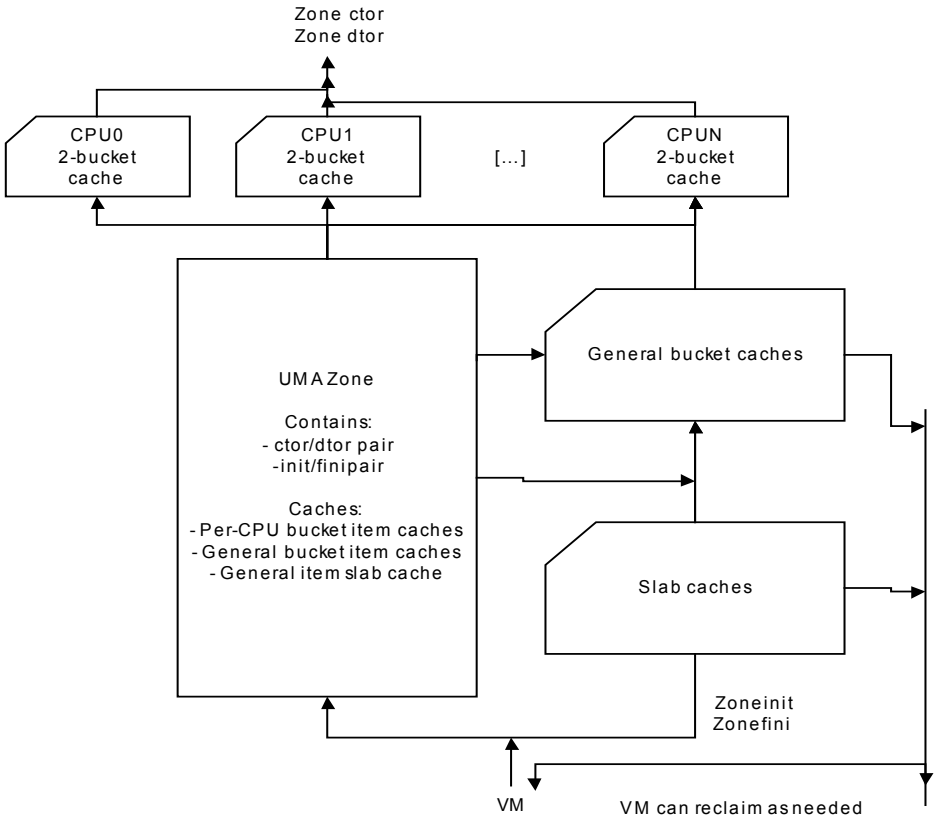

Figure 4: A Typical UMA Zone

5.4 Extending UMA to Accommodate Common-Case Usage

An approach to configuring Mbuf and Cluster allocations via UMA using the unmodified UMA framework would consist in the creation of at least two Zones. The first Zone would provide the system with Mbufs. The second Zone would provide the required 2K Clusters. There would be no *init/fini* for either Zone. The *ctor/dtor* for the Mbuf Zone would involve initializing Mbuf header information for the constructor, and possibly freeing an External Buffer for the destructor. The Cluster Zone would only require a constructor responsible for initializing a reference counter and hooking the Cluster to an Mbuf who's reference would have to be provided through the constructor.

Figure 5 shows a possible setup for network buffer allocation using the original UMA implementation.

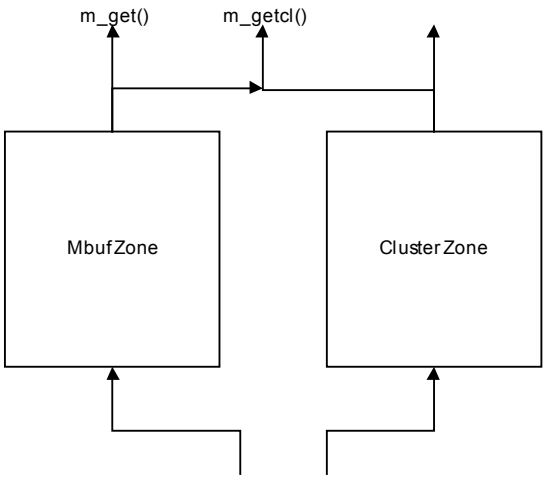

Figure 5: Mbuf & Cluster Allocation Setup Within Original UMA

Although the obvious approach is straightforward, it lacks support for the very common scenario where both an Mbuf and a Cluster are required at the same time (see section 4, API usage). Support for an atomic Mbuf and Cluster allocation was added in the Mbuf allocation API in FreeBSD 5.x in the form of *m_getcl()*. The existing *m_clget()* (also known as MCLGET()) which only allocates a Cluster and attaches it to an already allocated Mbuf is becoming much less common than the new *m_getcl()*. Therefore, it would make sense to provide a cache of Mbufs with pre-attached Clusters and thus allow faster atomic Mbuf and Cluster allocations.

In order to accomplish a Packet (Mbuf with Cluster attached) cache, two approaches were implemented and considered. The first consisted of overlaying small single-bucket per-CPU caches on top of two or more existing Zones and filling the overlayed Packet caches from the existing Mbuf and Cluster Zones. The per-CPU caches are accompanied by their own *ctor/dtor* pair along with a *back-allocator/back deallocator* pair which is called to replenish or drain the caches when required. However, early performance testing, in particular as FreeBSD code was modified to use *m_getcl()* (atomic Mbuf and Cluster allocate) where permitted, indicated that larger Packet caches tend to perform better because they allow, on average, more Packet allocations to occur from the cache. The need for larger caches meant that either buckets could be artificially grown large or that a cache of multiple buckets would be better suited for Packets.

The second approach, which is the adopted approach, consists in a more pronounced slab to bucket-cache interface. In this solution, the traditional UMA Zone is made to only hold the general and per-CPU bucket caches. The slab cache itself is defined and held in a separate structure called a Keg. The Keg structure now holds the slab cache and handles back-end (VM) allocations. The traditional UMA Zone then becomes a single Zone, now called the Master Zone, backed by a single back-end Keg. The Packet Zone is then implemented as a Secondary Zone to the Mbuf Zone. Therefore, there are caches for Mbufs, Clusters, and Packets, and the cache sizes are scaled by UMA according to demand. That is, UMA may decide to allocate additional buckets, if needed, and grow Zone caches. The slab cache for Mbufs is shared between the Mbuf Master Zone and the Packet Secondary Zone.

The applications of this solution may extend beyond merely Mbufs and Clusters. In fact, it is worth investigating whether other commonly-used variations of Mbufs, such as for example Mbufs with pre-attached Sendfile Buffers, or pre-allocated *m_tag* metadata for TrustedBSD support [9], deserve their own Secondary Zone caches as well. This, and other similar investigations, are left as future work (section 5.7).

A second problem with the obvious UMA Zone setup is addressing the reference counting issue for Mbuf Clusters. There are a couple of ways to address reference counting within the UMA framework. The first is to hook custom back-end allocation and deallocation routines to the UMA Zone and force Mbuf Clusters to come from a separate virtual address map which would be pre-allocated on startup according to the traditional NMBCLUSTERS compile-time or boot-time tunable. The solution then consists of providing a sparse mapping of Clusters to reference counters by virtual address and thus each Cluster, upon indexing into the sparse map, would be able to retrieve its reference counter (as in *mballoc*). The sparse map works because Clusters would come from their own virtual address map and thus ensure a very simple way to index into a sparse array of counters. Unfortunately, the idea implies that the maximum number of Mbuf Clusters would still need to be specified at compile or boot-time.

Instead, the adopted solution consists of altering the UMA framework to allow the creation of Zones with a special Reference-Counter flag that ensures that the underlying object slabs contain space for object reference counters. The reference counter can then be looked up from the Cluster constructor by referring to the allocated Cluster's underlying slab structure. The slab itself may be looked up via a hash table or through a hidden slab reference within the underlying page structure. An implication of the adopted solution is that the NMBCLUSTERS as a compile-time or boot-time option can finally be removed and replaced instead with a variable that is runtime-tunable by the System Administrator. The applications of this solution may also extend beyond merely Clusters.

The final but easily solved problem with the UMA Zone setup is that since some of the constructors are asked to allocate additional memory, there is a possibility that the memory allocation may fail. In fact, this is a more general problem and the solution is to allow for constructors and initializers to fail by indicating failure to the mainline UMA code. Thus, a failed constructor or initializer will be permitted to force allocation failure by returning NULL to the caller. Finally, the applications of this solution, like for the other above-described solutions, may also extend beyond merely Mbufs and Clusters.

The FreeBSD Network Buffer implementation within this modified framework is shown in *Figure 6*.

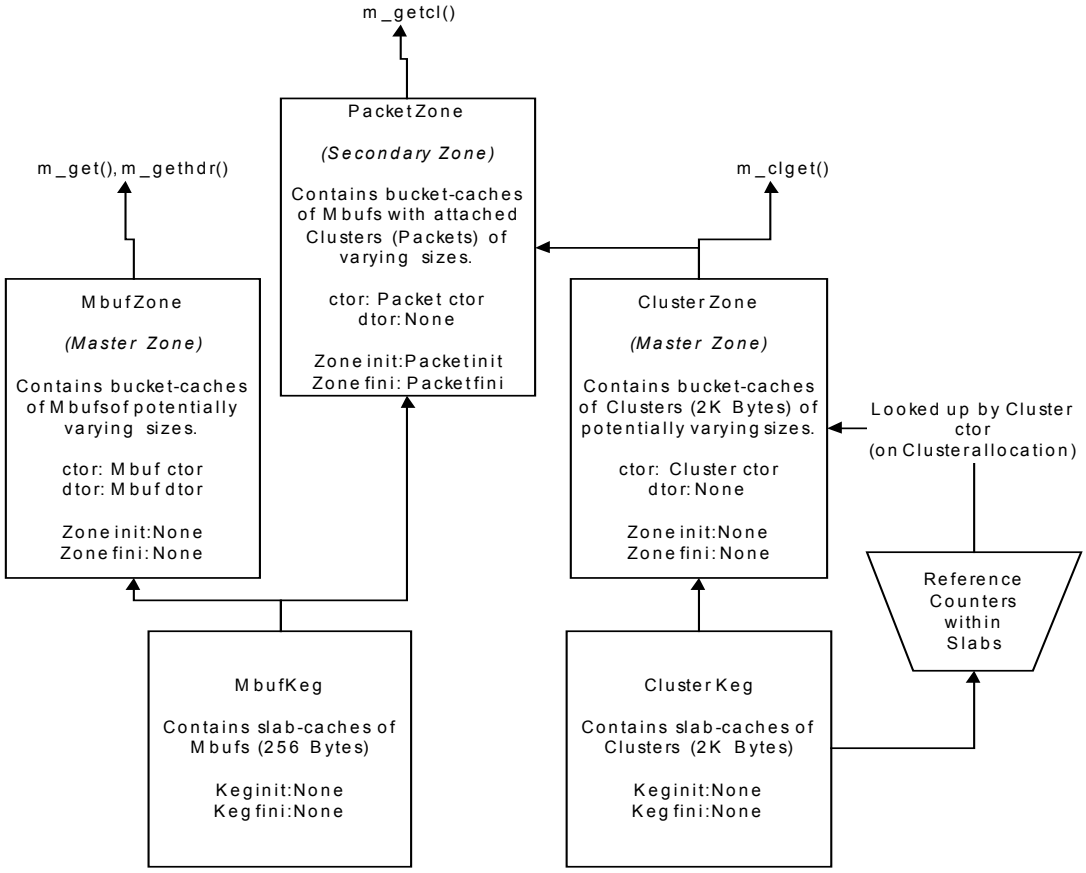

Figure 6: The mbuma Zone Setup

5.5 Performance Considerations

An important advantage of the modified network buffer allocation scheme is its ability to recover unused memory resources following, for example, a large spike in network-related activity. Neither the network buffer allocator in FreeBSD 4.x (and earlier) nor *mballoc* in earlier FreeBSD 5.x implemented memory reclamation and instead kept all freed Mbuf and Clusters cached for fast future allocation. While large caches are undoubtedly beneficial for common-case allocations during future equally heavy network activity, overall system performance is not merely affected by the system's ability to provide fast network buffer allocations.

Unreclaimed memory leads to an increase in swapping even though network activity may have

returned to normal. *Figure 7* shows the total number of pages swapped out on a FreeBSD 5.x machine over time. The graph compares *mballoc* (stock FreeBSD 5.x) to *mbuma* and shows how each affects swap use following a large network spike exhausting a maximum number of Mbuf Clusters. The superior response of *mbuma* is in this case entirely due to UMA's ability to hand unused pages from its caches back to the VM system, if required.

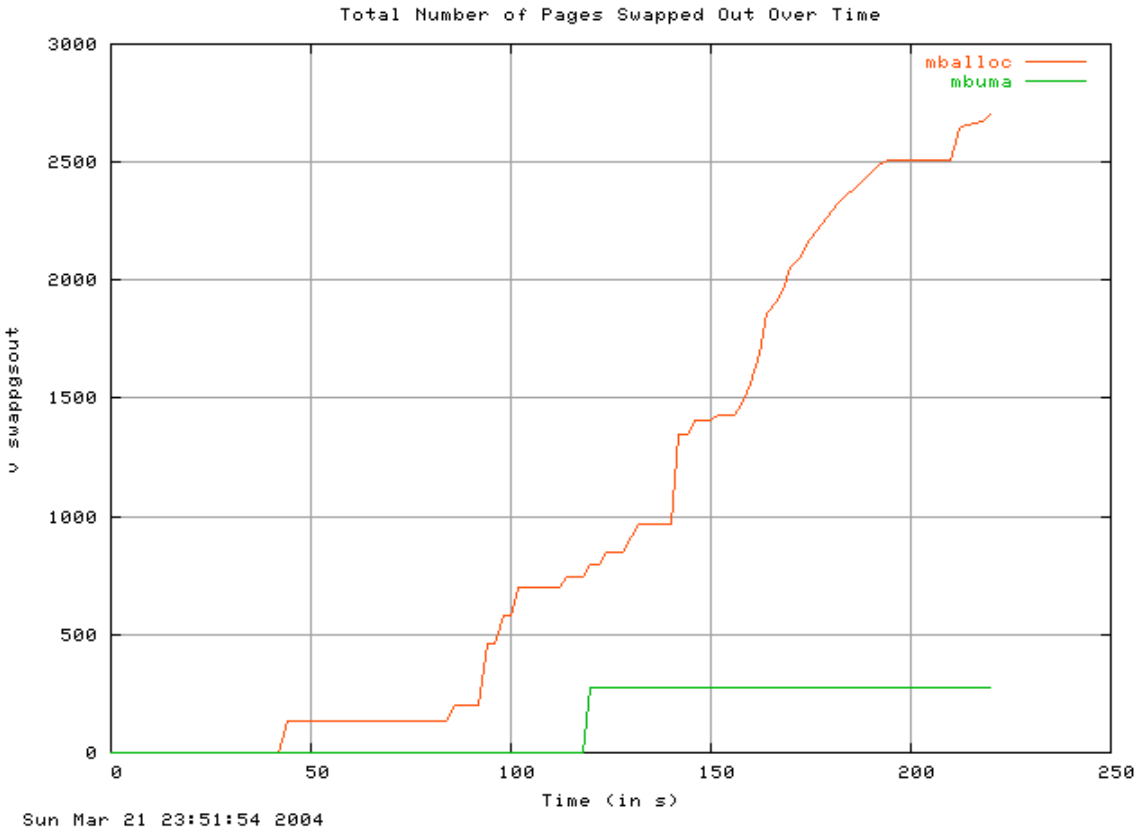

Figure 7: mballoc & mbuma Swap Usage Following Network Spike
(the top line shows mballoc results, the lower line shows mbuma results)

An additionally important performance consideration is *mbuma*'s ability to handle high packet throughput, in particular relative to the stock implementation in present-day FreeBSD 5.x. *Netperf* [11] performance data for both TCP and UDP stream tests is shown in *Figure 8*.

The data in *Figure 8* was collected on a two-machine setup with a gigabit-Ethernet connection serving the link. The first machine was configured to send packets with *netperf* and was running FreeBSD 4.x which is considered to provide network buffers optimally in a single-processor setup. The second machine ran FreeBSD 5.x (both the stock and the patched *mbuma* versions), was SMP (dual-processor), contained an Intel 1000 (FreeBSD's *em* driver) controller, and was configured as the *netperf* receiver for all tests.

Unfortunately, the controller on the sender side was attached to a 32-bit PCI bus, which likely significantly influenced throughput. Despite this limitation, the tests reveal that throughput was about equivalent to present-day FreeBSD 5.x without modification, notably differing in small magnitudes, likely due to the allocators replenishing their caches at different times throughout execution.

It should be noted that further throughput performance testing may reveal more interesting

results and should be performed, in particular with faster network controller configurations (such as on-board high-bandwidth gigabit Ethernet controllers). Due to hardware availability requirements, this was left as future work.

Netperf TCP Stream Tests: Relative Comparison of Stock -CURRENT and mbuma (+/- 2.5% with 99% conf.)

				-CURRENT stock (SMP)	*mbuma (SMP)*
	Send/Recv Socket Size (Bytes)	**Send Message Size (Bytes)**	**Elapsed Time (secs)**	**Throughput (10^6 bps)**	**Throughput (10^6 bps)**
	57,344	4,096	60.01	241.47	**250.75**
	57,344	8,192	60.01	269.39	269.08
	57,344	32,768	60.01	284.86	284.16
	32,768	4,096	60.01	**258.93**	246.13
	32,768	8,192	60.01	273.35	272.67
	32,768	32,768	60.01	280.07	278.72

Netperf UDP Stream Tests: Relative Comparison of Stock –CURRENT and mbuma (+/- 5.0% with 99% conf.)

			-CURRENT stock (SMP)	*mbuma (SMP)*
			(Send/Receive)	*(Send/Receive)*
Socket Size (Bytes)	**Message Size (Bytes)**	**Elapsed Time (secs)**	**Throughput (10^6 bps)**	**Throughput (10^6 bps)**
32,768	64	60.01	21.25 / 6.88	21.28 / 7.69
32,768	1,024	60.01	255.06 / 66.19	254.05 / 68.38
32,768	1,472	60.01	318.45 / 96.48	320.38 / 97.66

Figure 8: Netperf TCP and UDP Stream Test (Relative Comparison)

5.6 Future Work

As previously mentioned, there is current ongoing work on replacing the common-case UMA allocation with code that merely requires a simple critical section, where the critical section itself does not typically require a hard interrupt disable. Whether this proves to be significantly superior to per-CPU mutex locks remains to be seen, but current evidence regarding the cost of acquiring a mutex lock seems to suggest that it will be.

Additionally, it would be worth considering whether other types of External Buffers (besides for Clusters) would benefit from the UMA Keg approach, with a Secondary Zone configuration. Similarly, Mbufs with pre-attached *m_tags* for TrustedBSD [9] extensions might benefit from a Secondary Zone configuration as well. However, one should be careful to not segment the Mbuf Keg too much, as there is a parallel between cache fragmentation and having pre-initialized objects available in UMA's caches.

For what concerns space wastage within Mbufs when External Buffers are used, it is worth investigating the implementation of the so-called mini-Mbuf, a smaller Mbuf without an internal data region used merely to refer to an External Buffer of a given type. The idea of the mini-Mbuf was first brought to the author's attention by Jeffrey Hsu, who noted that mini-Mbufs

would likely result in less wasted space when External Buffers are used.

For what concerns network performance, it would be worth investigating the scalability of *mbuma* versus that of FreeBSD 4.x as the number of CPUs is varied in an SMP configuration. In particular, this evaluation should occur once most of the Giant lock if unwound off the FreeBSD 5.x network stacks.

Finally, it would be interesting to consider the implementation of Jumbo Buffers for large MTU support for gigabit-Ethernet controllers. In particular, integration with UMA is highly desirable, as a Secondary Zone could be configured for Jumbo Buffers as well.

6 Conclusion & Acknowledgements

FreeBSD 5.x's development continues today but with the advent of most recent changes, scalability and performance will soon again become the ultimate priority of the FreeBSD developer.

In order to facilitate the task of tuning and optimizing, from both organizational and framework perspectives, it is imperative to design a clean architecture. The purpose of the work presented in this paper was therefore not to only make network buffer allocations more SMP-friendly (in the scalability sense), but to also eliminate code and concept duplication and provide a clean and better understood network buffer allocation framework.

This balance of achieving solid performance while maintaining a consistent design has been the goal of many of the present and past BSD projects and is the single most important motivation for the work presented in this paper.

The author would like to thank the BSD communities for their lofty aspirations, the FreeBSD SMPng team (all those involved with FreeBSD 5.x's development in one way or another), and in particular Jeff Roberson for implementing a fine general-purpose memory allocation framework for FreeBSD 5.x and offering valuable suggestions during early *mbuma* attempts. Robert Watson (McAfee Research, TrustedBSD, FreeBSD Core Team) and Andrew Gallatin (FreeBSD Project, Myricom, Inc.) deserve special mention for their help with performance analysis. Jeffrey Hsu (DragonFlyBSD Project, FreeBSD Project) also deserves a special mention for taking the time to discuss issues pertaining to this work on numerous occasions and making many important recommendations.

Finally, the author would like to thank his friends and family, who have offered him the possibility of that fine balance in everyday life, who through their presence and support have deeply influenced the way the author lives, and without whom this work would never have been conceivable.

7 References

[1] M. K. McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System,* Addison-Wesley Longman, Inc. (1996)

[2] The NetBSD Project, http://www.netbsd.org/

[3] sendfile(2) man page, sf_bufs implementation in arch/arch/vm_machdep.c, CVS Repository: http://www.freebsd.org/cgi/cvsweb.cgi

[4] Alan Cox, *Kernel Korner: Network Buffers and Memory Management*, The Linux Journal, Issue 30 (1996). URL: http://www.linuxjournal.com/article.php?sid=1312

[5] Brad Fitzgibbons, *The Linux Slab Allocator* (2000). URL: http://www.cc.gatech.edu/people/home/bradf/cs7001/proj2/index.html

[6] John H. Baldwin, *Locking in the Multithreaded FreeBSD Kernel*, Proceedings of the BSDCon 2002 Conference (2002).

[7] Intel, *IA-32 Intel Architecture Software Developer's Manual, Vols. I, II, III.* (2001).

[8] Jeff Bonwick, *The Slab Allocator: An Object-Caching Kernel Memory Allocator.* USENIX Summer 1994 Conference Proceedings. (1994).

[9] The TrustedBSD Project, http://www.trustedbsd.org/

[10] The OpenBSD Project, http://www.openbsd.org/

[11] Netperf, a Network Performance Analyzer. URL: http://www.netperf.org/