A Scalable Concurrent `malloc(3)` Implementation for FreeBSD

Jason Evans <jasone@FreeBSD.org>

April 16, 2006

Abstract

The FreeBSD project has been engaged in ongoing work to provide scalable support for multi-processor computer systems since version 5. Sufficient progress has been made that the C library's `malloc(3)` memory allocator is now a potential bottleneck for multi-threaded applications running on multi-processor systems. In this paper, I present a new memory allocator that builds on the state of the art to provide scalable concurrent allocation for applications. Benchmarks indicate that with this allocator, memory allocation for multi-threaded applications scales well as the number of processors increases. At the same time, single-threaded allocation performance is similar to the previous allocator implementation.

Introduction

FreeBSD's previous `malloc(3)` implementation by Kamp (1998), commonly referred to as *phkmalloc*, has long been considered one of the best available, and has fared well in published comparisons (Feng and Berger, 2005; Berger et al., 2000; Bohra and Gabber, 2001). However, it was designed at a time when multi-processor systems were rare, and support for multi-threading was spotty. The FreeBSD project is engaged in an ongoing effort to provide scalable performance on SMP systems, and has made sufficient progress that `malloc(3)` had become a scalability bottleneck for some multi-threaded applications. This paper presents a new `malloc(3)` implementation, informally referred to here as *jemalloc*.

On the surface, memory allocation and deallocation appears to be a simple problem that merely requires a bit of bookkeeping in order to keep track of in-use versus available memory. However, decades of research and scores of allocator implementations have failed to produce a clearly superior allocator. In fact, the best known practices for measuring allocator performance are remarkably simplistic, as are the summary statistics for measured performance. Wilson et al. (1995) provide an excellent review of the state of the art as of a decade ago. Multi-processors were not a significant issue then, but otherwise the review provides a thorough summary of the issues that face modern allocators. Following are brief mentions of various issues that particularly impact *jemalloc*, but no attempt is made to discuss all of the issues that must be considered when designing an allocator.

Allocator performance is typically measured via some combination of application execution time and average or peak application memory usage. It is not sufficient to measure the time consumed by the allocator code in isolation. Memory layout can have a significant impact on how quickly the rest of the application runs, due to the effects of CPU cache, RAM, and virtual memory paging. It is now commonly accepted that synthetic traces are not even adequate for measuring the effects of allocation policy on fragmentation (Wilson et al., 1995). The only definitive measures of allocator performance are attained by measuring the execution time and memory usage of real applications. This poses challenges when qualifying the performance characteristics of allocators. Consider that an allocator might perform very poorly for certain allocation patterns, but if none of the benchmarked applications manifest any such patterns, then the allocator may appear to perform well, despite pathological performance for some work loads. This makes testing with a wide variety of applications important. It also motivates an approach to allocator design that minimizes the number and severity of degenerate edge cases.

Fragmentation can be thought of in terms of internal fragmentation and external fragmentation. Internal fragmentation is a measure of wasted space that is associated with individual allocations, due

to unusable leading or trailing space. External fragmentation is a measure of space that is physically backed by the virtual memory system, yet is not being directly used by the application. These two types of fragmentation have distinct characteristic impacts on performance, depending on application behavior. Ideally, both types of fragmentation would be minimal, but allocators have to make some tradeoffs that impact how much of each type of fragmentation occurs.

RAM has become dramatically cheaper and more plentiful over the past decade, so whereas *phk-malloc* was specially optimized to minimize the working set of pages, *jemalloc* must be more concerned with cache locality, and by extension, the working set of CPU cache lines. Paging still has the potential to cause dramatic performance degradation (and *jemalloc* doesn't ignore this issue), but the more common problem now is that fetching data from RAM involves huge latencies relative to the performance of the CPU.

An allocator that uses less memory than another does not necessarily exhibit better cache locality, since if the application's working set does not fit in cache, performance will improve if the working set is tightly packed in memory. Objects that are allocated close together in time tend also to be used together, so if the allocator can allocate objects contiguously, there is the potential for locality improvement. In practice, total memory usage is a reasonable proxy for cache locality; *jemalloc* first tries to minimize memory usage, and tries to allocate contiguously only when it doesn't conflict with the first goal.

Modern multi-processor systems preserve a coherent view of memory on a per-cache-line basis. If two threads are simultaneously running on separate processors and manipulating separate objects that are in the same cache line, then the processors must arbitrate ownership of the cache line (Figure 1). This false cache line sharing can cause serious performance degradation. One way of fixing this issue is to pad allocations, but padding is in direct opposition to the goal of packing objects as tightly as possible; it can cause severe internal fragmentation. *jemalloc* instead relies on multiple allocation arenas to reduce the problem, and leaves it up to the application writer to pad allocations in order to avoid false cache line sharing in performance-critical code, or in code where one thread allocates objects and hands them off to multiple other threads.

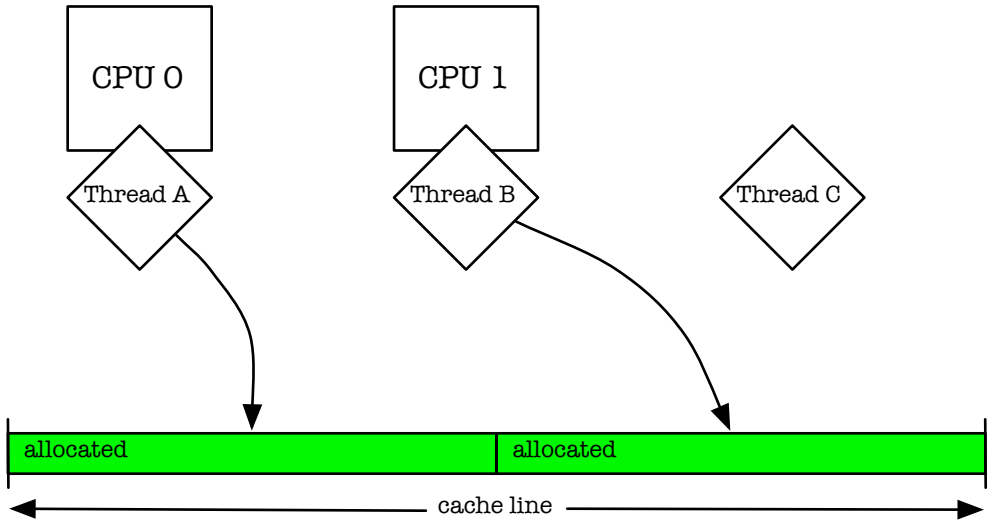

Figure 1: Two allocations that are used by separate threads share the same line in the physical memory cache (false cache sharing). If the threads concurrently modify the two allocations, then the processors must fight over ownership of the cache line.

One of the main goals for this allocator was to reduce lock contention for multi-threaded applications running on multi-processor systems. Larson and Krishnan (1998) did an excellent job of presenting and testing strategies. They tried pushing locks down in their allocator, so that rather than using a single

allocator lock, each free list had its own lock. This helped some, but did not scale adequately, despite minimal lock contention. They attributed this to "cache sloshing" – the quick migration of cached data among processors during the manipulation of allocator data structures. Their solution was to use multiple arenas for allocation, and assign threads to arenas via hashing of the thread identifiers (Figure 2). This works quite well, and has since been used by other implementations (Berger et al., 2000; Bonwick and Adams, 2001). *jemalloc* uses multiple arenas, but uses a more reliable mechanism than hashing for assignment of threads to arenas.

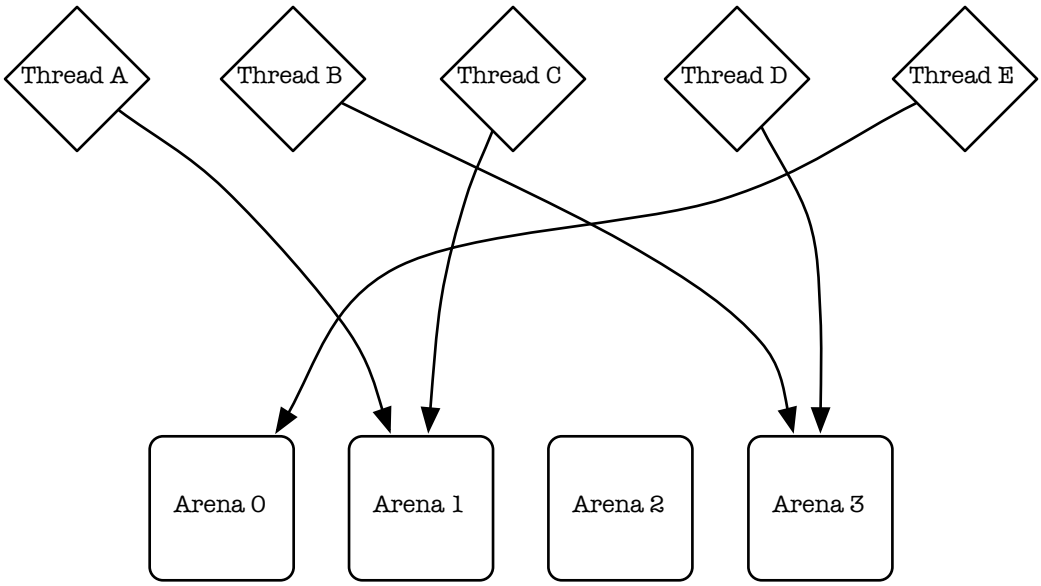

Figure 2: Larson and Krishnan (1998) hash thread identifiers in order to permanently assign threads to arenas. This is a pseudo-random process, so there is no guarantee that arenas will be equally utilized.

The rest of this paper describes the primary *jemalloc* algorithms and data structures, presents benchmarks that measure performance and scalability of multi-threaded applications on a multi-processor system, as well as performance and memory usage of single-threaded applications, and discusses measurement of memory fragmentation.

Algorithms and data structures

FreeBSD supports run-time configuration of the allocator via the `/etc/malloc.conf` symbolic link, the `MALLOC_OPTIONS` environment variable, or the `_malloc_options` global variable. This provides for a low-overhead, non-intrusive configuration mechanism, which is useful both for debugging and performance tuning. *jemalloc* uses this mechanism to support the various debugging options that *phkmalloc* supported, as well as to expose various performance-related parameters.

Each application is configured at run-time to have a fixed number of arenas. By default, the number of arenas depends on the number of processors:

Single processor: Use one arena for all allocations. There is no point in using multiple arenas, since contention within the allocator can only occur if a thread is preempted during allocation.

Multiple processors: Use four times as many arenas as there are processors. By assigning threads to a set of arenas, the probability of a single arena being used concurrently decreases.

The first time that a thread allocates or deallocates memory, it is assigned to an arena. Rather than hashing the thread's unique identifier, the arena is chosen in round-robin fashion, such that arenas

are guaranteed to all have had approximately the same number of threads assigned to them. Reliable pseudo-random hashing of thread identifiers (in practice, the identifiers are pointers) is notoriously difficult, which is what eventually motivated this approach. It is still possible for threads to contend with each other for a particular arena, but on average, it is not possible to do initial assignments any better than round-robin assignment. Dynamic re-balancing could potentially decrease contention, but the necessary bookkeeping would be costly, and would rarely be sufficiently beneficial to warrant the overhead.

Thread-local storage (TLS) is important to the efficient implementation of round-robin arena assignment, since each thread's arena assignment needs to be stored somewhere. Non-PIC code and some architectures do not support TLS, so in those cases, the allocator uses thread identifier hashing. The thread-specific data (TSD) mechanism that is provided by the pthreads library (Butenhof, 1997) would be a viable alternative to TLS, except that FreeBSD's pthreads implementation allocates memory internally, which would cause infinite recursion if TSD were used by the allocator.

All memory that is requested from the kernel via `sbrk(2)` or `mmap(2)` is managed in multiples of the "chunk" size, such that the base addresses of the chunks are always multiples of the chunk size (Figure 3). This chunk alignment of chunks allows constant-time calculation of the chunk that is associated with an allocation. Chunks are usually managed by particular arenas, and observing those associations is critical to correct function of the allocator. The chunk size is 2 MB by default.

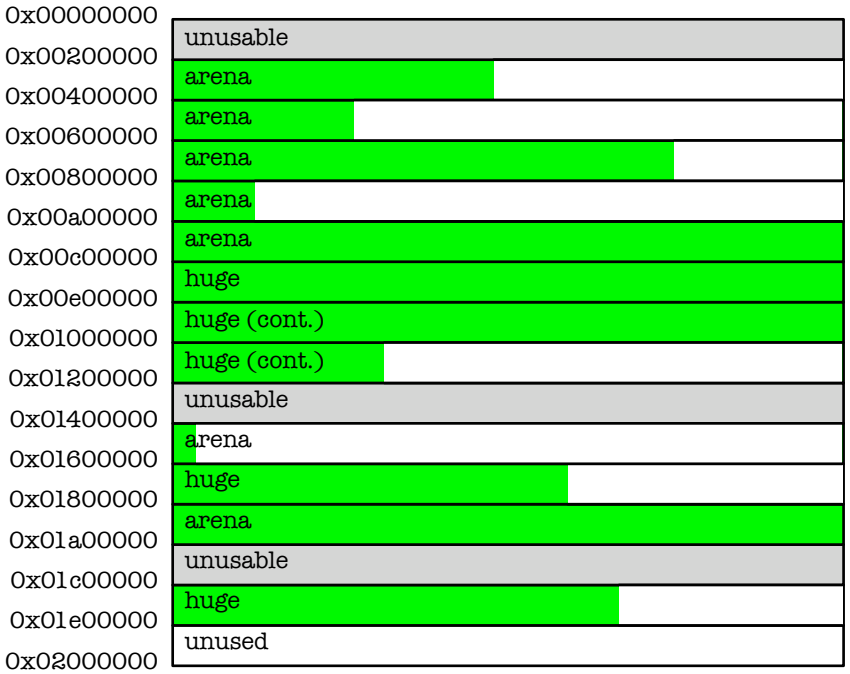

Figure 3: Chunks are always the same size, and start at chunk-aligned addresses. Arenas carve chunks into smaller allocations, but huge allocations are directly backed by one or more contiguous chunks.

Allocation size classes fall into three major categories: small, large, and huge. All allocation requests are rounded up to the nearest size class boundary. Huge allocations are larger than half of a chunk, and are directly backed by dedicated chunks. Metadata about huge allocations are stored in a single red-black tree. Since most applications create few if any huge allocations, using a single tree is not a scalability issue.

For small and large allocations, chunks are carved into page runs using the binary buddy algorithm. Runs can be repeatedly split in half to as small as one page, but can only be coalesced in ways that

reverse the splitting process. Information about the states of the runs is stored as a page map at the beginning of each chunk. By storing this information separately from the runs, pages are only ever touched if they are used. This also enables the dedication of runs to large allocations, which are larger than half of a page, but no larger than half of a chunk.

Small allocations fall into three subcategories: tiny, quantum-spaced, and sub-page. Modern architectures impose alignment constraints on pointers, depending on data type. `malloc(3)` is required to return memory that is suitably aligned for any purpose. This worst case alignment requirement is referred to as the quantum size here (typically 16 bytes). In practice, power-of-two alignment works for tiny allocations since they are incapable of containing objects that are large enough to require quantum alignment. Figure 4 shows the size classes for all allocation sizes.

Category	Subcategory	Size
Small	Tiny	2 B
		4 B
		8 B
	Quantum-spaced	16 B
		32 B
		48 B
		...
		480 B
		496 B
		512 B
	Sub-page	1 kB
		2 kB
Large		4 kB
		8 kB
		16 kB
		...
		256 kB
		512 kB
		1 MB
Huge		2 MB
		4 MB
		6 MB
		...

Figure 4: Default size classes, assuming runtime defaults, 4 kB pages and a 16 byte quantum.

It would be simpler to have no subcategories for small allocations by doing away with the quantum-spaced size classes. However, most applications primarily allocate objects that are smaller than 512 bytes, and quantum spacing of size classes substantially reduces average internal fragmentation. The larger number of size classes can cause increased external fragmentation, but in practice, the reduced internal fragmentation usually more than offsets increased external fragmentation.

Small allocations are segregated such that each run manages a single size class. A region bitmap is stored at the beginning of each run, which has several advantages over other methods:

- The bitmap can be quickly scanned for the first free region, which allows tight packing of in-use regions.
- Allocator data and application data are separate. This reduces the likelihood of the application corrupting allocator data. This also potentially increases application data locality, since allocator data are not intermixed with application data.
- Tiny regions can be easily supported. This would be more difficult if, for example, a free list were embedded in the free regions.

There is one potential issue with run headers: they use space that could otherwise be directly used by the application. This could cause significant external fragmentation for size classes that are larger

than the run header size. In order to limit external fragmentation, multi-page runs are used for all but the smallest size classes. As a result, for the largest of the small size classes (usually 2 kB regions), external fragmentation is limited to approximately 3%.

Since each run is limited in how many regions it can manage, there must be provisions for multiple runs of each size class. At any given time, there is at most one "current" run for each size class. The current run remains current until it either completely fills or completely empties. Consider though that it would be possible for a single malloc/free to cause the creation/destruction of a run if there were no hysteresis mechanism. To avoid this, runs are categorized according to fullness quartile, and runs in the `QINIT` category are never destroyed. In order for a run to be destroyed, it must first be promoted to a higher fullness category (Figure 5).

Fullness categories also provide a mechanism for choosing a new current run from among non-full runs. The order of preference is: `Q50`, `Q25`, `Q0`, then `Q75`. `Q75` is the last choice because such runs may be almost completely full; routinely choosing such runs can result in rapid turnover for the current run.

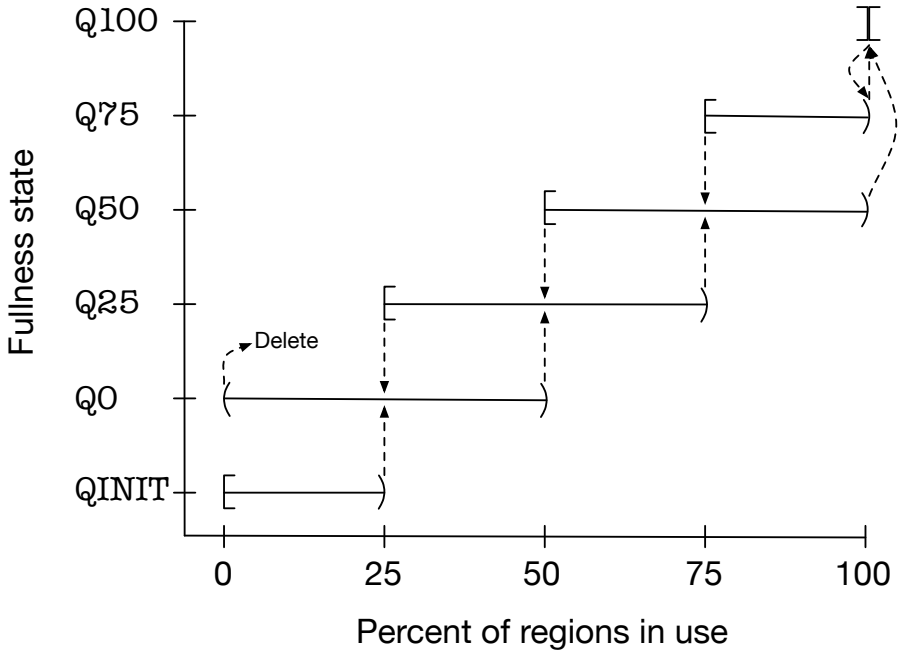

Figure 5: Runs are categorized according to fullness, and transition between states as fullness increases/decreases past various thresholds. Runs start in the `QINIT` state, and are deleted when they become empty in the `Q0` state.

Experiments

In order to quantify the scalability of *jemalloc* as compared to *phkmalloc*, I ran two multi-threaded benchmarks and five single-threaded benchmarks on a four-processor system (dual-dual Opteron 275), using FreeBSD-CURRENT/amd64 (April 2006). I investigated several other thread-safe allocators before performing the benchmarks, but ultimately only included Doug Lea's *dlmalloc*, due to various portability issues. I modified *dlmalloc*, version 2.8.3, so that I was able to integrate it into a custom libc that included libc-based spinlock synchronization, just as *phkmalloc* and *jemalloc* use. All benchmarks were invoked using the `LD_PRELOAD` loader mechanism in order to choose which allocator implementation was used. `libthr` was used rather than `libpthread` for all benchmarks, since there was a thread

switching performance issue in `libpthread` that had not been resolved as of the time this paper was written.

It should be noted that *dlmalloc* is not intended as a scalable allocator for threaded programs. It is included in all benchmarks for completeness, but the primary reason for including it in these experiments was for the single-threaded benchmarks.

Multi-threaded benchmarks

malloc-test

`malloc-test` is a microbenchmark that was created by Lever and Boreham (2000) to measure the upper bound on allocator scalability for multi-threaded applications. The benchmark executes a tight allocation/deallocation loop in one or more threads. On a multi-processor system, allocator throughput will ideally scale linearly as the number of threads increases to match the number of processors, then remain constant as the number of threads continues to increase. Figure 6 shows the results.

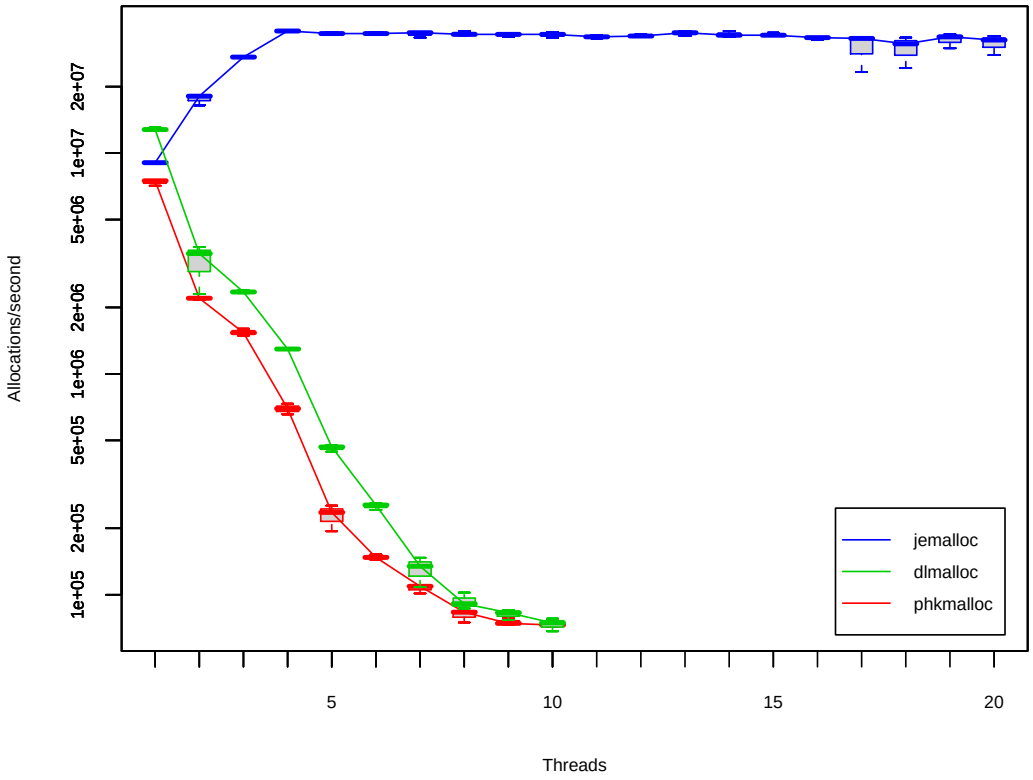

Figure 6: Allocator throughput, measured in allocations/second, for increasing numbers of threads. Each run performs a total of 40,000,000 allocation/deallocation cycles, divided equally among threads, creating one 512-byte object per cycle. All configurations are replicated three times, and the results are summarized by box plots, where the central lines denote the medians and the whiskers represent the most extreme values measured.

phkmalloc and *dlmalloc* suffer severe performance degradation if more than one thread is used, and performance continues to degrade dramatically as the number of threads increases. No results are presented for more than ten threads for these allocators, due to to the extremely long run times required.

jemalloc scales almost perfectly up to four threads, then remains relatively constant above four threads, with an interesting exception above sixteen threads. *jemalloc* uses sixteen arenas by default

on a four-processor system, so contention is minimal until at least seventeen threads are running. Above sixteen threads, some threads contend over arenas, and once the other threads finish, those that are contending for arenas exhibit worst case performance until they complete.

super-smack

The second multi-threaded benchmark uses a database load testing tool called Super Smack (`http://vegan.net/tony/supersmack/`), version 1.3, that runs one or more client threads that access a server — MySQL 5.0.18 in this case. Super Smack comes with two pre-configured load tests, and I used one of them (`select-key.smack`). Each run of `super-smack` performed approximately 200,000 queries, divided equally among client threads.

Results are summarized in Figure 7. When *jemalloc* is used, performance degrades gracefully as the number of client threads increases, and variability is low, especially for worst case performance. When *phkmalloc* is used, median performance is typically about the same as for *jemalloc*, but worst case performance variability is extreme. Additionally, there is a sudden dramatic drop in performance between 75 and 80 client threads. *dlmalloc* performs well on average, but as with *phkmalloc*, worst case variability is extreme.

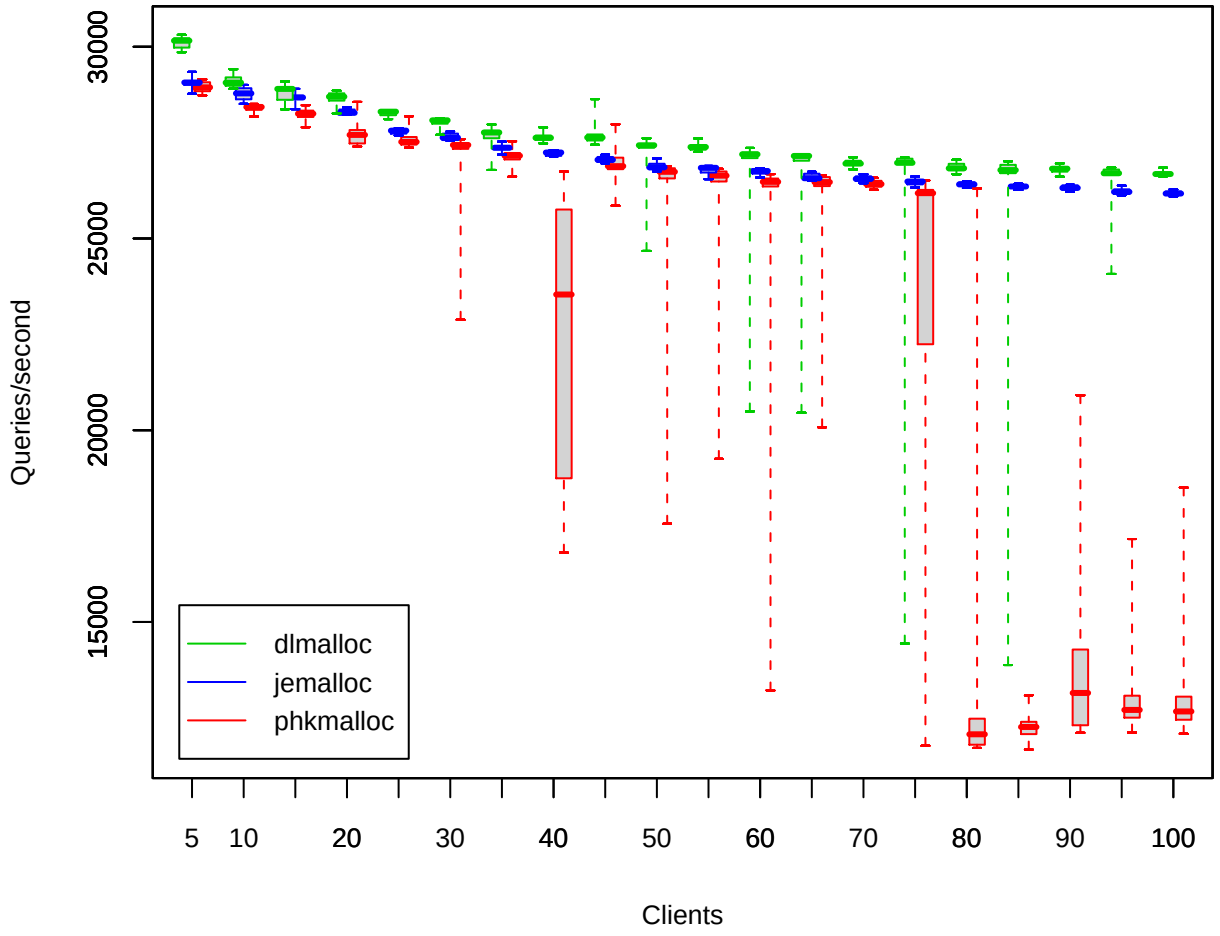

Figure 7: MySQL query throughput, measured in queries/second, for increasing numbers of client threads. A total of approximately 100,000 queries are performed during each run, divided evenly among client threads. All configurations are replicated ten times, and the results are summarized by box plots.

Single-threaded benchmarks

I performed benchmarks of five single-threaded programs. These benchmarks should be taken with a grain of salt in general, since I had to search pretty far and wide to find repeatable tests that showed significant run time differences. The run times of most real programs simply do not depend significantly on malloc performance. Additionally, programs that make heavy use of networks or filesystems tend to be subject to high variability in run times, which requires many replicates when assessing significance of results. As a result, there is an inherent selection bias here, and these benchmarks should not be interpreted to be representative of any particular class of programs.

Figure 8 summarizes the results for the single-threaded benchmarks. More details about the benchmarks follows.

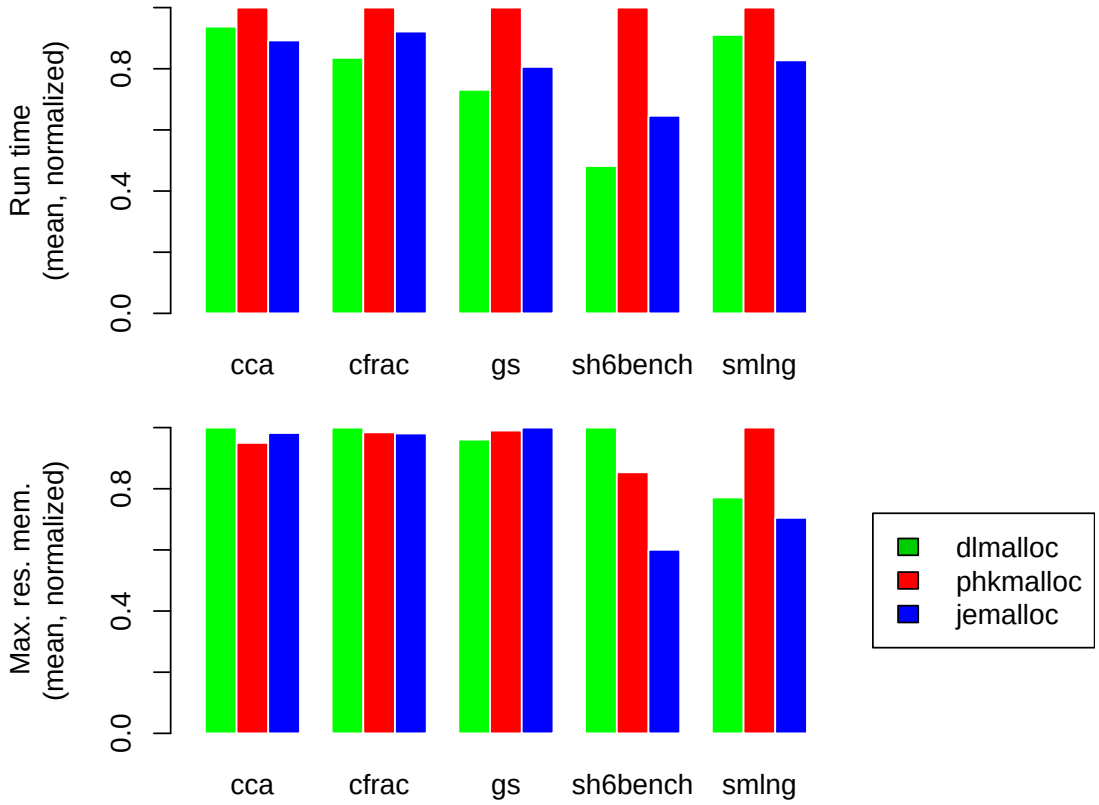

Figure 8: Scaled run time and maximum resident memory usage for five single-threaded programs. Each graph is linearly scaled such that the maximum value is 1.0.

cca

`cca` is a Perl script that makes heavy use of regular expressions to extract cpp logic from C code, and generate various statistics and a PostScript graph of the cpp logic. For these experiments, I concatenated all `.c` and `.h` files from FreeBSD's libc library, and used the resulting file as input to `cca`. I used version 5.8.8 of Perl from the ports tree, and compiled it to use the system malloc. Following is a summary of three replicates (also see Figure 8):

`cca`	*dlmalloc*	*phkmalloc*	*jemalloc*
Run time (mean)	72.88 s	77.73 s	69.37 s
Max. res. mem. (mean)	11244 kB	10681 kB	11045 kB

cfrac

`cfrac` is a C program that factors large numbers. It is included with the Hoard memory allocator (`http://www.cs.umass.edu/~emery/hoard/`). I used 4758260277435811572216749275547536776 7 as input. Following is a summary of ten replicates (also see Figure 8):

`cfrac`	*dlmalloc*	*phkmalloc*	*jemalloc*
Run time (mean)	9.57 s	11.45 s	10.55 s
Max. res. mem. (mean)	2105 kB	2072 kB	2064 kB

`cfrac` overwhelmingly allocates 16- and 32-byte objects, but its overall memory usage is not high. *dlmalloc* likely uses slightly more memory than the other allocators due to higher internal fragmentation for small objects.

gs

`gs` (GhostScript) is a PostScript interpreter. I used AFPL GhostScript 8.53 from the ports tree. I used a 37 MB input file, and ran `gs` as:

```
gs -dBATCH -dNODISPLAY PS3.ps
```

Following is a summary of three replicates (also see Figure 8):

`gs`	*dlmalloc*	*phkmalloc*	*jemalloc*
Run time (mean)	31.06 s	42.48 s	34.24 s
Max. res. mem. (mean)	15001 kB	15464 kB	15616 kB

gs overwhelmingly allocates either 240-byte objects or large objects, since it uses a custom allocator internally. As a result, this benchmark stresses performance of large object allocation.

phkmalloc appears to suffer system call overhead due to the large memory usage fluctuations, whereas the other allocators have more hysteresis built in, which reduces the total number of system calls. *jemalloc* does not perform quite as well as *dlmalloc* because of the additional overhead of managing runs, as opposed to merely calling `mmap()`.

sh6bench

`sh6bench` is a microbenchmark by MicroQuill, available for download from their website (`http://www.microquill.com/`). This program repeatedly allocates groups of equal-size objects, retaining some portion of the objects from each cycle, and freeing the rest, in various orders. The number of objects in each group is larger for small objects than for large objects, so that there is a bias toward allocating small objects.

I modified `sh6bench` to call `memset()` immediately after each `malloc()` call, in order to assure that all allocated memory was touched. For each run, `sh6bench` was configured to do 2500 iterations, for object sizes ranging from 1 to 1000. Following is a summary of ten replicates (also see Figure 8):

`sh6bench`	*dlmalloc*	*phkmalloc*	*jemalloc*
Run time (mean)	3.35 s	6.96 s	4.50 s
Max. res. mem. (mean)	105467 kB	90047 kB	63314 kB

The fact that allocations aren't actually used for anything in this microbenchmark penalizes *jemalloc* as compared to *dlmalloc*. *jemalloc* has to touch the per-run region bitmap during every allocation/deallocation, so if the application does not make significant use of the allocation, *jemalloc* suffers decreased cache locality. In practice, it is unrealistic to allocate memory and only access it once. *jemalloc* can actually improve cache locality for the application, since its bitmap does not spread out regions like the region headers do for *dlmalloc*.

As mentioned in the introduction, synthetic traces are not very useful for measuring allocator performance, and `sh6bench` is no exception. It is difficult to know what to make of the huge differences in memory usage among the three allocators.

smlng

`smlng` is an Onyx (`http://www.canonware.com/onyx/`) program. It implements an SML/NG parser and optimizer, for the ICFP 2001 Programming Contest (`http://cristal.inria.fr/ICFP2001/prog-contest/`). I used a single-threaded build of Onyx-5.1.2 for this benchmark. For input, I repeatedly concatenated various SML/NG examples and test cases to create a 513 kB file. Following is a summary of ten replicates (also see Figure 8):

`smlng`	*dlmalloc*	*phkmalloc*	*jemalloc*
Run time (mean)	12.65 s	13.89 s	11.49 s
Max. res. mem. (mean)	71987 kB	93320 kB	65772 kB

jemalloc appears to use less memory than *phkmalloc* because the predominantly used size classes are not all powers of two: 16, 48, 96, and 288. *jemalloc* also appears to do a good job of recycling regions as memory use fluctuates, whereas *dlmalloc* experiences some external fragmentation due to the allocation pattern.

Analysis

Exhaustive benchmarking of allocators is not feasible, and the benchmark results should not be interpreted as definitive in any sense. Allocator performance is highly sensitive to application allocation patterns, and it is possible to construct microbenchmarks that show any of the three allocators tested here in either a favorable or unfavorable light, at the benchmarker's whim. I made every attempt to avoid such skewing of results, but my objectivity should not be assumed by the reader. That said, the benchmarks should be sufficient to convince the reader of at least the following:

- *jemalloc*'s run time performance scales well for multi-threaded programs running on multi-processor systems.
- *jemalloc* exhibits similar performance to *phkmalloc* and *dlmalloc* for single-threaded programs, both in terms of run time and memory usage.

In point of fact, *jemalloc* performed very well for the presented benchmarks, and I have found no reasons to suspect that *jemalloc* would perform substantially worse than *phkmalloc* or *dlmalloc* for anything but specially crafted microbenchmarks.

None of the benchmarks were designed to measure performance under memory pressure. Such benchmarks would be of interest, but I did not include them here mainly because *phkmalloc* has been shown to perform well under memory pressure (Kamp, 1998; Feng and Berger, 2005), and *jemalloc* uses sufficiently similar algorithms that it should exhibit similar performance.

Fragmentation was quite difficult to analyze, since it is mainly a qualitative issue, and standard tools only provide quantitative metrics. Maximum resident memory usage results are only moderately useful when analyzing fragmentation, since differences between allocators indicate differences in fragmentation at the point of maximum memory utilization. Obviously, an allocator could cause much worse data locality for the application without having a substantial impact on maximum resident memory usage.

In order to better understand fragmentation, I wrote a program that processes `kdump(1)` output that was generated by using the "`U`" `malloc(3)` option in conjunction with `ktrace(1)`. An example graph is shown in Figure 9. With this program I was able to observe the effects of various layout policies, and as a result *jemalloc* uses memory much more effectively than in earlier development versions.

Understanding allocator performance for various allocation patterns is a constant challenge. Mechanisms for gaining insight into what the allocator is actually doing are important both during allocator development and application development. To this end, *jemalloc* outputs detailed statistics at exit if optional code is enabled at libc compile time, and if the "`P`" `malloc(3)` option is specified. Following is output from running the `cca` benchmark. Detailed interpretation is left to the reader who is interested enough to read the allocator source code. However, most of the statistics should have obvious meanings after having read this paper.

```
___ Begin malloc statistics ___
Number of CPUs: 4
Number of arenas: 16
Chunk size: 2097152 (2^21)
Quantum size: 16 (2^4)
Max small size: 512
Pointer size: 8
Assertions enabled
Allocated: 7789600, space used: 14680064

chunks:
       nchunks   highchunks    curchunks
          1279            7            7

huge:
     nmalloc      ndalloc    allocated
           0            0            0

arenas[0] statistics:
allocated: 7789600
calls:
      nmalloc      ndalloc     nmadvise
     76908219     76890620            0
large requests: 10681
bins:
          bin   size nregs run_sz nrequests nruns hiruns curruns npromo  ndemo
            0 T    2  1892   4096       184     1      1       1      0      0
            1 T    4   946   4096      2824     1      1       1      1      0
            2 T    8   985   8192  64656199  9318     21      14 201961 319453
            3 Q   16  1004  16384   5431302   169      3       2  12121  16369
            4 Q   32  1014  32768   2058948     1      1       1      3      0
            5 Q   48  1358  65536   4518045     4      4       4   1827   1889
            6 Q   64  1019  65536    101063     2      2       2      7      1
            7 Q   80   815  65536     53401     1      1       1      0      0
            8 Q   96   679  65536     55408     2      2       2      5      1
            9 Q  112   582  65536       609     1      1       1      0      0
           10 Q  128   509  65536      5261     1      1       1      0      0
           11 Q  144   452  65536      3564     1      1       1      0      0
           12 Q  160   407  65536      1206     1      1       1      0      0
           13 Q  176   370  65536       326     1      1       1      0      0
           14 Q  192   339  65536       549     1      1       1      0      0
           15 Q  208   313  65536       100     1      1       1      0      0
           16 Q  224   291  65536        53     1      1       1      0      0
           17 Q  240   271  65536        90     1      1       1      0      0
           18 Q  256   254  65536        58     1      1       1      0      0
           19 Q  272   239  65536      7550     1      1       1      0      0
           20 Q  288   226  65536        44     1      1       1      0      0
           21 Q  304   214  65536        52     1      1       1      0      0
           22 Q  320   203  65536        36     1      1       1      0      0
           23 Q  336   194  65536        38     1      1       1      0      0
           24 Q  352   185  65536        26     1      1       1      0      0
           25 Q  368   177  65536        49     1      1       1      0      0
           26 Q  384   169  65536        16     1      1       1      0      0
           27 Q  400   163  65536        36     1      1       1      0      0
           28 Q  416   156  65536        13     1      1       1      0      0
           29 Q  432   150  65536        19     1      1       1      0      0
           30 Q  448   145  65536        19     1      1       1      0      0
           31 Q  464   140  65536        12     1      1       1      0      0
           32 Q  480   135  65536        29     1      1       1      0      0
           33 Q  496   131  65536        45     1      1       1      0      0
           34 Q  512   127  65536        14     1      1       1      0      0
           35 S 1024    63  65536       230     1      1       1      0      0
           36 S 2048    31  65536       120     1      1       1      3      0
--- End malloc statistics ---
```

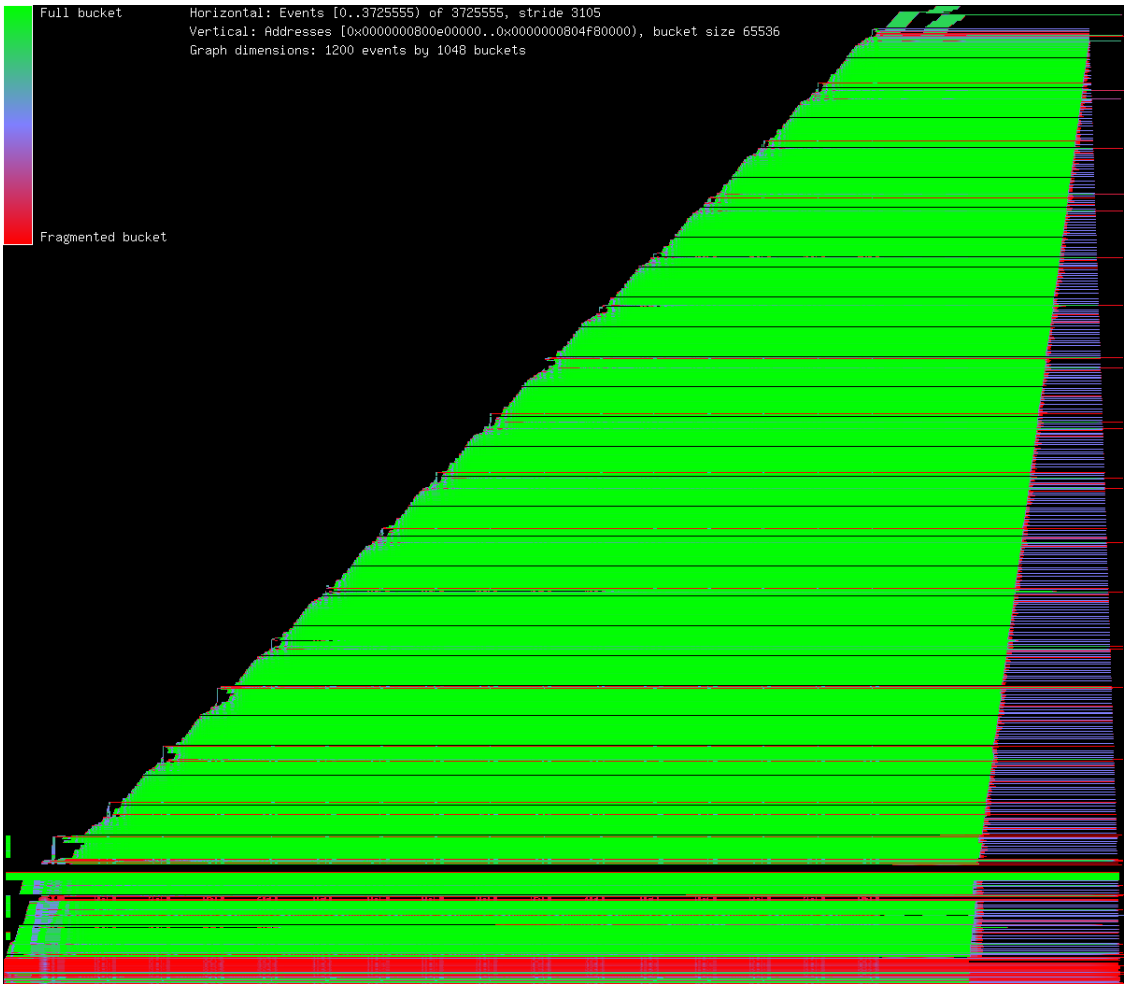

Figure 9: Graph of memory usage on amd64 when running the `smlng` benchmark. Each position along the horizontal axis represents a snapshot of memory at an instant in time, where time is discretely measured in terms of allocation events. Each position along the vertical axis represents a memory range, and the color indicates what proportion of that memory range is in use.

Discussion

One of the constant frustrations when developing *jemalloc* was the observation that even seemingly innocuous additional features, such as the maintenance of a per-arena counter of total allocated memory, or any division, caused measurable performance degradation. The allocator started out with many more features than it ended up with. Removed features include:

- `malloc_stats_np()`. Just adding a per-arena counter of total allocated memory has a measurable performance impact. As a result, all statistics gathering is disabled by default at build time. Since the availability of statistics isn't a given, a C API isn't very useful.
- Various sanity checks. Even simple checks are costly, so only the absolute minimum checks that are necessary for API compliance are implemented by default.

On a more positive note, the runtime allocator configuration mechanism has proven to be highly flexible, without causing a significant performance impact.

Allocator design and implementation has strong potential as the subject of career-long obsession, and indeed there are people who focus on this subject. Part of the allure is that no allocator is superior

for all possible allocation patterns, so there is always fine tuning to be done as new software introduces new allocation patterns. It is my hope that *jemalloc* will prove adaptable enough to serve FreeBSD for years to come. *phkmalloc* has served FreeBSD well for over a decade; *jemalloc* has some big shoes to fill.

Availability

The *jemalloc* source code is part of FreeBSD's libc library, and is available under a two-clause BSD-like license. Source code for the memory usage graphing program that was used to generate Figure 9 is available upon request, as are the benchmarks.

Acknowledgments

Many people in the FreeBSD community provided valuable help throughout this project, not all of whom are listed here. Kris Kennaway performed extensive stability and performance testing for several versions of *jemalloc*, which uncovered numerous issues. Peter Wemm provided optimization expertise. Robert Watson provided remote access to a four-processor Opteron system, which was useful during early benchmarking. Mike Tancsa donated computer hardware when my personal machine fell victim to an electrostatic shock. The FreeBSD Foundation generously funded my travel so that I could present this work at BSDcan 2006. Aniruddha Bohra provided data that were used when analyzing fragmentation. Poul-Henning Kamp provided review feedback that substantially improved this paper. Finally, Rob Braun encouraged me to take on this project, and was supportive throughout.

References

Berger ED, McKinley KS, Blumofe RD, Wilson PR (2000) Hoard: A Scalable Memory Allocator for Multithreaded Applications. ASPLOS 2000

Bohra A, Gabber E (2001) Are Mallocs Free of Fragmentation? In USENIX 2001 Annual Technical Conference: FREENIX Track

Bonwick J, Adams J (2001) Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In Proceedings of the 2001 USENIX Annual Technical Conference

Butenhof DR (1997) Programming with POSIX Threads. Addison–Wesley, Reading, Massachusetts

Feng Y, Berger ED (2005) A Locality-Improving Dynamic Memory Allocator. MSP

Kamp PH (1998) Malloc(3) revisited. In USENIX 1998 Annual Technical Conference: Invited Talks and FREENIX Track, 193–198

Larson P, Krishnan M (1998) Memory allocataion for long-running server applications. In Proceedings of the International Symposium on Memory Management (ISSM), 176–185

Lever C, Boreham D (2000) `malloc()` Performance in a Multithreaded Linux Environment. In USENIX 2000 Annual Technical Conference: FREENIX Track

Wilson PR, Johnstone MS, Neely M, Boles D (1995) Dynamic Storage Allocation: A Survey and Critical Review. In Proceedings of the 1995 International Workshop on Memory Management