How to Tune Malloc to Optimize Serving Side Applications

Naredula Janardhana Reddy, Mahadevan Iyer and Srigurunath Chakravarthi
Performance Engineering, Yahoo!
{njana,mkiyer,sriguru}@yahoo-inc.com

Abstract

The steady increase in CPU cores in commodity servers, evolving memory hierarchies including the appearance of NUMA in Intel architecture, and decreasing cost of RAM have introduced several opportunities to optimize application memory (de)allocation and access performance. This paper describes various such performance opportunities that developers of performance-critical applications at Yahoo! must pay special attention to. We present a performance overview of three high-performance malloc libraries: TCMalloc, Jemalloc and Y-jemalloc – our own enhancement of Jemalloc that contributed up to 18% latency reduction for the Ad-serving application NGD. Most of this improvement was achieved by avoidance of locks to synchronize concurrent malloc/free calls and minimizing system calls to allocate/free address space regions and free individual memory pages. This paper puts in one place all considerations related to picking and tuning the malloc library to deliver best performance for your application type and hardware configuration.

1. Introduction

Efficient application memory management has become increasingly important due to rapid advances in multi-core processor architecture and memory hierarchies and ever increasing in-memory processing.
This paper addresses the following important issues:

1. Online apps and RAM cost: For "serving" apps at Yahoo!, latency is an important requirement and often SLA bound. RAM has become progressively cheaper and 16-24 GB servers are now common. There is a big opportunity to systematically exploit *CPU-Mem tradeoffs* to achieve better latencies at the cost of higher memory footprint. The malloc library is the "agent" that facilitates such trade-offs and it is important to understand how.
2. On multi-core systems, efficient concurrent memory operations across threads with minimal or no lock contention is important since it both reduces latency and increases total throughput of the threads. Concurrency optimizations can help improve both the latency as well as the *capacity* of typical "serving" apps at Yahoo!, leading to *CapEx* and *OpEx* savings.
3. With Intel introducing Non-Uniform Memory Access (NUMA) into its new generation of processors (Nehalem and Westmere), performance can be significantly improved by making malloc libraries "NUMA-aware". This is a newly unfolded opportunity for Yahoo! applications to exploit.

Jemalloc [1] and TCMalloc [2] are two commonly used malloc libraries meant for multi-threaded applications that need to perform a fairly high rate of (de)allocations of a wide range of sizes. Previous related works [3-9] did not comprehensively address all the multi-threaded performance issues addressed in this paper.

Despite several thread-aware optimizations in Jemalloc and TCMalloc, it was observed that both these libraries imposed a fairly large (de)allocation performance overhead on NGD (Non-Guaranteed Delivery). TCMalloc provided better speed but its memory footprint was unacceptably high and not controllable. This led the primary author of this paper to make targeted improvements to Jemalloc (called "Y-Jemalloc" in this paper). Y-Jemalloc demonstrated significant NGD response time improvement compared to what was achievable with JeMalloc or TCMalloc. These optimizations are generic, tunable and usable for a broad class of multi-threaded applications, and particularly benefit request-response applications at Yahoo!. We believe that malloc optimization and tuning considerations that we have presented in this paper will be useful to all Yahoo! developers that are writing high-performing "serving" applications. Many of these optimizations will also benefit "offline" or "content processing" applications, but this paper does not focus on that offline class of applications.

As related work, the Multi Processor Communications (MPC) project [13] implements a per-thread heap for lock-free malloc and is also NUMA-aware but unlike Y-Jemalloc it does not address 1) tunable heap caching and control of system call overheads, 2) cost of calloc page clearing, and 3) use of HugeTLB.

2. Multi-Core Malloc Libraries

This section presents the high level design of two popular malloc libraries design with performance on multi-core / multi-CPU systems in mind.

2.1. *Jemalloc*

Design: Objects are managed by splitting entire address space into memory pools called arenas where a group of threads share a single arena [1]. There is a chance of lock contention if more threads than arenas exist. The objects are maintained using a binary buddy algorithm inside each arena. This is different from pre-created objects in TCMalloc.

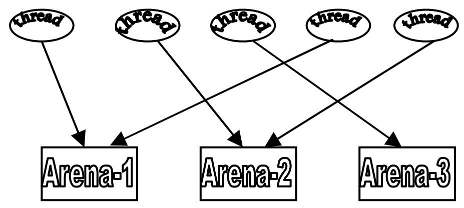

Figure 1. Jemalloc: Arena based design

2.2. TCMalloc

Design: Objects are managed at two levels, one is at central cache and other is a per-thread cache [2]. When malloc is called, if a request can be satisfied with the free objects in the per-thread cache it is a quick lockless operation. Otherwise, it will search the central cache for free objects. In addition to satisfying the current request, it will also pre-fetch a bunch of objects of the same size as the request and add those into the thread cache.

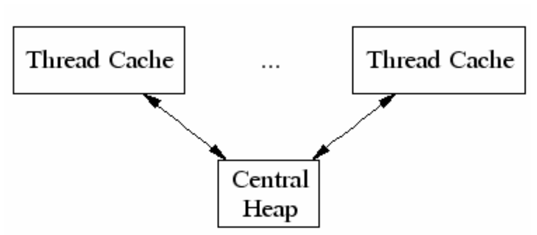

Figure 2. TCMalloc: Thread cache and central heap

The thread cache maintains one linked list of buckets for each size in contrast to jemalloc's binary buddy objects.

2.3. Pros and Cons of TCMalloc and jemalloc

TCMalloc's performs better when the app's heap growth or reduction is moderate such as when allocation and matching de-allocation operations are well interleaved. Most malloc requests can then be satisfied from the per-thread cache without locking overhead from other threads even when there are a large number of threads. Jemalloc in contrast always uses a lock even with single thread per arena. The performance degrades further with increase in the number of threads per arena, but this can be mitigated by increasing the arenas at application launch.

On the other hand, when heap growth or reduction is large such as due to a sequence of malloc's followed by matching free's, TCMalloc suffers frequent movement of objects from the local cache to central and vice versa. This causes contention among threads for the single central lock. This is where TCMalloc performs worse than jemalloc.

Common Disadvantage: Both TCMalloc and jemalloc potentially incur a large number of system calls that degrade performance when the heap stretches or shrinks beyond specific limits:

1. When *malloc* is called, *mmap* is called to allocate address space if it can't be satisfied from memory available in the library.
2. When *free* is called, *madvice* or *munmap* are called for releasing memory from the library back to the OS.

Ideally the total cache of free memory chunks maintained inside the malloc library must be large enough to avoid too many such system calls but small enough to limit the total system memory usage and avoid swapping. Jemalloc and TCMalloc do not provide full control of the size limits of this free memory which is however fully implemented in Y-Jemalloc.

3. Malloc Performance Considerations

At a high-level, the performance of malloc libraries on multi-core systems can be viewed in terms of three metrics:

1. Latency of *malloc* and *free* calls.
2. Level of concurrency across threads. This is a measure of how contention-free it is.
3. Memory Footprint. This is a measure of how efficiently the library manages the available memory.

3.1. Performance Issues and Opportunities

We have identified 9 important issues affecting memory allocation and access performance. Some of these are dealt with by existing malloc libraries but others have only been addressed fully in Y-Jemalloc..

1. **Locking Overhead**: In a typical *malloc* implementation supporting multi-threading, for every *malloc/free* call a global lock is used at least once to synchronize access to internal malloc data structures such as free lists. This can impose a big latency overhead e.g. for glibc's ptmalloc, the lock and unlock operations accounted for ~33% (100 ns) out of ~300 ns for malloc+free on a 2.8 GHz Xeon server. Two factors contribute to this:
   i. *Memory bus locking:* CPU needs to lock the memory bus to perform atomic operations required by both mutex lock and unlock. E.g., in our Ad server application, it contributed ~3% of total CPU time.
   ii. *Contention*: The wait-for-lock latency grows with the number threads contending for the lock. This will lead to a system call.

   Solution (Less Locking or Lockless): Reduce contention by using different memory areas per thread or thread-group e.g. Jemalloc's *arenas* or Tcmalloc's per-thread caches. Malloc calls from threads assigned to

different arenas will access different lock variables thereby reducing the lock contention among threads.
In fact at one thread per arena, we *can fully eliminate use of lock*s. This 'lockless' optimization can be exploited by applications such as NGD with a known bound on the number of threads.

2. **Excess Unused Pages**: Malloc implementations maintain a pool of memory pages that have been freed by the application but are cached for satisfying future malloc requests. Some like jemalloc make *madvice*() system calls to release the physical memory for individual pages from this pool when the pool size crosses a threshold. This threshold has to be tuned to the application characteristics: a low value will cause a) more frequent madvice() calls or b) more frequent allocation requests to the next layer i.e. global cache common to all threads or the OS, upon malloc calls from the application. These operations incur system calls or locking and hence raise the average latency. On the other hand a high threshold will increase the total memory usage potentially leading to swapping to/from disk.
   Solution (Page-freeing Threshold Tuning): Maintaining a user-configurable or adaptive threshold e.g. Jemalloc's "F" flag.
3. **Excess Address Space Pruning**: This is similar to the free pages latency problem described above, except that the malloc makes munmap() or sbrk() system calls to free entire chunks of mapped virtual address spaces rather than releasing individual physical pages using madvice().
   Solution (Address-Space-Pruning Threshold): Maintaining a threshold parameter for total size of app-freed address space chunks of all arenas. Release one or more of those chunks to OS (munmap) only when it increases above the threshold. JeMalloc does not provide such a threshold but Y-JeMalloc does as explained in the next section.
4. **Inter-Thread Alloc-Dealloc**: In arena-based implementations [1] when objects are allocated by one thread (owner) and freed by another thread (non-owner) the latter suffers lock access latency.
   Solution: It is possible to avoid lock contention by queuing up cross-thread free-s and allow for them to be freed by a subsequent malloc/free operation by the owner thread. This can be implemented as a multiple-producer single-consumer queue with fine-grain locking. To prevent memory leak and to limit unusable memory (associated with unprocessed 'free's), enqueued objects are processed in the following instances: a) while inserting and queue size exceeds a limit (in this case, lockless mode is turned off) b) when the owner thread calls malloc/free, and c) when the owner thread exits.
5. **HugeTLB:** There are many mission-critical commercial applications in use today such as large-scale databases and web servers that cache large amounts of disk data in memory to satisfy latency requirements. For global-scale servers used in global Internet firms this memory usage easily reaches several GB which with the normal 4KB page size requires of the order of a million page table entries. Thus only a tiny fraction of the application's memory accesses will be satisfied by the TLB and the rest have to hit the page table. This can significantly slow down most of those memory accesses and hence degrade latency of server response to users.
   Solution: By using HugeTLB feature of paging hardware we can lower TLB misses e.g., On Intel one can set the page size to 2MB instead of 4KB which increases by 500-fold the memory footprint covered by the TLB. See section 4 for the results of a HugeTLB benchmarking test.
6. **Page Clearing Overhead**: Applications that frequently use *calloc* instead of *malloc* suffer higher latency due to *calloc*'s clearing of memory bits to zeros. This is typically done via *memset* which incurs measurable latency and uses CPU cycles.
   Solution: The clearing could be done in a background house-keeping thread that clears and maintains a pool of zero-ed pages. This can significantly reduce the latency of *calloc* at the cost of higher memory consumption required to store the pool of zero-ed memory. None of the libraries discussed in this paper implement this optimization. Alternately, if hardware features (instruction set or special hardware) to reset memory are available, the malloc library can use them to save on CPU cycles.
7. **False Cache Sharing** : On multi-core CPUs with shared cache, when different memory objects located in the same cache-line are concurrently accessed by different threads running on different cores, due to the cache synchronization mechanism in hardware, other threads will end up having to wait to access their objects in that cache-line while one thread is writing to it. This causes multi-threaded program execution speed to in fact degrade with increase in the number of cores and threads [11]. As [12] points out the situation of different objects allocated in the same cache-line can be quite common in multi-threaded programs due to there being a number of reasons: 1) the objects are nearby in same array or are fields in the same objects 2) objects are allocated close together in time or by same thread 3) static objects laid out close together by the linker in memory 4) during compacting garbage collection when other intervening garbage objects between two objects got collected causing the latter two objects to become adjacent.
   Solution: Optimize malloc to allocate cache-line aligned blocks to prevent the malloc library from returning objects sharing a cache-line to different

threads. This comes at the cost of memory wastage for smaller sized allocations.

8. **Memory Footprint Control**: The Free Pages and Address Pruning optimizations described above come at the cost of increased system memory usage, and fundamentally involve a memory footprint versus speed trade-off. Often times, applications may incur a brief but anticipated increase in memory usage. As an example, Ad-serving applications constantly refresh specific files into memory. In such cases, speed optimizations that typically come at the cost of a high memory footprint can lead to a spike in memory usage and cause swapping. On the other hand, turning off these optimizations to avoid swapping during such situations will slow down allocation speed during normal serving activity.
   Solution (Speed-for-Memory Switch): Provide the ability to temporarily turn off some of the above optimizations and purge unused memory back to the OS. Y-JeMalloc implements this by providing two API functions to disable optimization & purge unused memory and to re-enable optimizations. The application will need to be modified to make use of this feature.
9. **Overhead of Consecutive Free calls**: For many request-response server applications, while serving a request it is common for applications to use a pattern of successive malloc calls followed by matching free calls. Such frees can be grouped together in one shot thereby saving CPU cycles.
   Solution: **Memory Contexts**: The allocator can provide the notion of a "memory context" in which all related (de)allocations occur. In such a paradigm, the application will create a memory context before performing all its allocations. Calls to free are replaced by one call that destroys the context. Destroying the context releases all its allocated memory in one shot. This saves CPU cycles and helps avoid memory leaks. PostgreSQL [10] uses a similar strategy to manage all memory (de)allocations for a transaction.
10. **NUMA Awareness**: For Non-Uniform Memory Access (NUMA) system architectures the OS, malloc library and application should be NUMA aware to reduce traffic on the shared memory bus. A NUMA aware OS typically allocates physical pages local to each CPU and the scheduler tries to avoid moving threads from one CPU to another. Applications can be made NUMA aware by not moving memory objects from one thread to another. Malloc libraries can be made NUMA aware as described here. The "first-touch" policy improves the probability that a thread will be assigned physical memory from the local node by the NUMA aware OS, but if the application releases the memory object and malloc library re-allocates the same object to thread running on a remote NUMA node, then this increases traffic on the shared bus. This is because typical malloc libraries are not NUMA aware; they do not keep track of memory locality and can interfere with the Operating System's effort to assign local physical memory to each thread. This type of pattern should be avoided by the malloc library.
    Solution: Maintain separate memory pools for each NUMA node so that released objects will be re-assigned only to threads attached to the same NUMA node. Note that Jemalloc and TCMalloc are currently *not* NUMA aware, and we need to address this short-coming in future.

3.2. Y-Jemalloc

Y-Jemalloc is an enhancement of jemalloc that implements the following subset of the above listed performance optimizations.:

- **Lockless mode** (S option): This is the optimization 1 explained in the previous section. Locks are used in *malloc* and *free only* when the number of threads of the application increases beyond the number of arenas. The latter is a fixed constant that can be chosen based on knowledge of the maximum threads likely to be launched in the application execution. E.g. in Ad-serving applications the number of worker threads is a configured constant.
- **Minimizing system calls**:
  - Page-Freeing Threshold (F option): This is optimization 2 of the previous section. It is already present in the original jemalloc. It is retained in Y-Jemalloc.and in fact the benefit of tuning it is clearly demonstrated in our experimental results in the next section. The default setting (no F option specified) corresponds to a threshold of 2MB only above which madvice() calls are made. A setting of nF corresponds to a threshold of 2^n times the default.
  - Per-Thread Address-Space Threshold (T) option: This is a new addition over jemalloc. Here a per-thread pool of address-space chunks obtained using mmap() or sbrk() is maintained. Specifying 'nT' as the option will limit the pool size to no more than 2^n chunks. Specifying n=0 will make it the same as jemalloc i.e. only one free chunk is maintained per thread. Typically n=0 and a chunk size of 1 MB is enough for typical apps that alternately allocate and free small objects. But in apps where there are many medium sized objects > 1MB, its worth tuning n to a positive value or increasing the chunk size.
  - Common Address-Space Threshold (W) option: This too is a new addition over jemalloc. Here a central pool of address space chunks available for all threads is maintained. Specifying 'nW' as the option will limit the pool size to no more than 2^n chunks. Typically it is advisable to maintain a larger central

Optimization	Y-Jemalloc	Jemalloc	TCMalloc	Benefiting Applications	Cost/Limitation of optimization
Lockless Mode	Available. Tunable.	Partial. Locking within arena.	Partial Locking can happen for access to central cache.	Any app with frequent or regular malloc/free calls; ~6% latency reduction in NGD	App must either increase # arenas to match # threads
Free-Pages Threshold	Available. Tunable.	Available. Tunable.	Available. Tunable.	Apps with successive malloc-s without interleaving free-s; 3-4% latency reduction in NGD	Increased memory usage..
Pruning Address Space	Available. Tunable.	Not available	Unknown	Same as above	Increased memory usage.
Inter-Thread	Available	Not available	Not available	Apps that allocate objects in one thread but free them in another.	Slightly increased memory usage. Dependence on owner thread to call malloc/free to piggy back queued free-s.
HugeTLB	Not available	Not available	Available	1. Faster access for large memory apps. Smaller page table memory for multi-process apps.	OS reserves memory ahead for Huge Pages.
Page-Cleaning	Available	Not available	Not available	Apps with frequent calloc's; 42% latency reduction due to avoiding byte-clearing	Need special h/w or h/w instructions, or extra mem for pre-cleared pools.
Cache-line Alignment	Available	Available	Available	Multi-threaded apps with many small-sized allocations. These apps have a high chance of false cache-line sharing.	Some mem wastage due to cache-line sized blocks for small requests.
Memory Footprint Control	Available	Not available	Not available	Apps that temporarily need large memory can avoid excess mem usage and swapping by using the speed-for-memory switch API.	Need to modify the apps to insert non-standard API.
Memory Context	Not available	Not available	Not available	Apps with a large number of successive malloc-ed objects that are all freed successively. e.g., transaction based allocations.	New paradigm. Apps need to be rewritten. More mem usage due to delayed frees.
NUMA Aware	Partial: With Lockless mode ON (without W option.)	Not NUMA Aware: Due to many threads sharing same arena.	Not NUMA aware : Due to Central cache	Any multi threaded application on NUMA machine.	1. Memory object may be moved from one thread to another. 2. Thread may be moved from CPU to another.

Table 1 Summary of optimizations, their costs, and benefiting applications.

pool than the per-thread T pool since it will hold chunks released by all the threads. In jemalloc, there was no such central pool i.e. upon a free() call from app for a large object more than the chunk size the memory was immediately returned back to OS.

- **Inter-Thread Alloc-Dealloc**: This is optimization 4 of the previous section. Y-Jemalloc adds a queue of pending frees to each arena to reduce the contention of cross-thread de-allocations.
- **Page Clearing Overhead**: Y-Jemalloc has prototyped the offline page-clearing mechanism using a house-keeping thread (optimization 6 of previous section).
- **Speed-for-Memory API**: Y-Jemalloc provides an API to control the level of optimizations on-the-fly (optimization 8 of previous section).
- **NUMA Awareness**: By default, the lockless mode described above without the use of the 'W' option is fully NUMA aware. For NUMA systems, it is recommended that the 'T' option be used instead of the 'W' option, but this will cost extra memory consumption.

To make 'W' fully NUMA aware instead of a central queue, a separate queue per NUMA node is needed. In such a design, if the malloc library cannot get an object from the same NUMA node's queue it will get it from another NUMA node and release the physical memory of that memory object by calling *madvise* (to release physical memory but not the virtual address space associate with that object). The OS will then reallocate the physical page from the local node upon “first-touch”. Y-Jemalloc does not implement this optimization since its end-use is currently not on NUMA systems.

4. Performance Results

This section presents performance results from NGD as well as micro-benchmarks showing the performance gain obtainable on modern multi-core Intel based systems (quad-core, dual-CPU “Harpertown” systems with 16 GB of RAM).

4.1. Y-Jemalloc in the NGD Ad Serving Application

In this experiment, we present Y-Jemalloc performance numbers for NGD an ad serving application whose primary performance metric is response-time to user queries. Incoming requests cause several memory allocations to occur successively before the corresponding deal locations occur. This. was seen to alternately grow and shrink the heap size by large amounts of up to 500 MB. Using a profiler, malloc and free were seen to contribute a measurable component of the overall CPU time spent with both TCMalloc and jemalloc that was attributable to locking and system call overheads.

Test Identifier	Y-Jemalloc Configuration W=address space pruning F=Free pages tuning, S=Enable lockless mode (Numerical prefix increases threshold)	Extra Memory Footprint (MB)	Approximate system calls per request: madvice + mmap + munmap	Latency Gain (%)
T0	(no optimizations)	-	10+5+5	(0%)
T1	S5W	100	10+0.3+0.3	13%
T2	S5W7F	200	0+0.3+0.3	16%
T3	S6W8F	500	0+0.05+0.01	18%
T4	S9W9F	1000	0+0+0	18%

Table 2 – NGD performance via Y-Jemalloc tuning

We measured the combined effect of implementing “Lockless Mode” (S option), the Page-Freeing Threshold (F option), and the Address-Space-Pruning Threshold via W option. Table 2 shows the results. Up to 18% reduction in user response latency was achieved at the cost of memory footprint increase of 500MB, i.e., only !4% over the basic no-optimization value of 3.6 GB. Note that the max memory overhead is 1 GB as shown in row T4 below. This corresponds to zero system calls which means that the max application heap usage was 3.6+1 GB where the extra 1GB was always allocatable from the per-thread and common malloc pools..

Though it is not shown in this table the lockless (S) option alone contributed 6% to the latency reduction.. Another roughly 3% was contributed by the Page-Freeing threshold (F) tuned to 2^7=128 pages as seen from the difference between rows 2 and 1. The remaining and largest improvement of 9% came from tuning Address-Space threshold (W) to 2^8=256 chunks. See [14] for the experiment details and logs.

4.2. False cache-line sharing micro-benchmark

Here two threads concurrently keep reading and writing continuously to two different elements of an array spaced *d* bytes apart in the array. When *d* was set to the cache-line size (>=64 bytes), the total test time was seen to drop ~3.5 times compared to that for *d* < 64 (table 3.). This shows that performance can be significantly improved if malloc allocates memory in cache-line-sized blocks. This prevents two threads from accessing locations in the same cache-line-sized block.

Test Description	Execution time	Improvement
Without Cache-line alignment	7.92 sec	-
With Cache-line Alignment	2.52 sec	214%

Table 3 – Cache-line Benchmark Results

4.3. Calloc micro-benchmark

The intention here is to measure the cost of clearing the page bytes in a *calloc* call. Table 4 below shows that the malloc call time averaged over a large sequence of back-to-back *malloc* calls is 42% faster than a same-sized sequence of c*alloc* calls. This indicates significant gain possible from calling *memset* in an offline housekeeping thread instead of from *calloc*.

Description	Malloc	Calloc	Improvement
Run time	9 sec	14sec	42%

Table 4 - Calloc vs. Malloc Test Results

4.4. HugeTLB Results

Micro-benchmark:

Here we measure the improvement from using HugeTLB pages of size 2MB. Memory of a given size ‘footprint’) was allocated for a data array on which a fixed number of reads (300 million) was sequentially done on equi-spaced locations i.e. separated by fixed offset (5000 bytes.) A single thread was used to avoid TLB flushes from context switches. Table 5 shows the test results. As the footprint size was increased from 160MB to 600MB, the use of HugeTLB showed an increasing reduction in run-time compared to using 4K pages and reached ~5X reduction at 320MB which is the max TLB table size. Huge pages covers 160*2 = 320MB of memory. Thus up to 320MB mem usage all the read accesses will use the TLB and never hit the page table. That explains why in table 5 the total time for 300M reads remains fixed at 5 sec for up to 320MB. In contrast with the normal 4K page size, the TLB only covers 160*4=640KB memory and hence more and more of the read accesses suffer a TLB miss and have to access the page table in memory as the footprint size increases. This explains the increasing execution time for normal page size.

Test #	**Footprint size (MB)**	**HugeTLB run time**	**Normal Page size run time (sec)**	**Perf Gain (%)**
T1	160	5	6	16%
T2	200	5	8	50%
T3	250	5	14	64%
T4	280	5	20	75%
T5	300	5	25	80%
T6	320	5	27	81%
T7	340	14	30	53%
T8	400	31	34	9%
T9	600	33	36	8%

Table 5 - TLB Test Results

4.5. NUMA micro benchmark

This micro-benchmark demonstrates the difference in memory access speed for a local vs. remote NUMA node. First, *n* threads are created, each of which malloc's *n* objects of 500MB each. Thus there is a matrix of n^2 objects of which each column corresponds to objects created by one thread. We run the test in two modes:

Mode 1: The ith thread accesses in a tight loop the bytes in the *n* objects of the ith column, i.e., its own objects. Hence, all accesses will be to the local NUMA node.

Mode 2: The ith thread accesses in a tight loop the bytes in the *n* objects of the ith row. For our test hardware containing 2 NUMA nodes, this corresponds to 50% local NUMA node access and 50% remote.

Hardware	**Mode1 (sec)**	**Mode2 (sec)**	**Gain**
E5530	280	357	21%
E5620	315	370	15%

Table 6 NUMA Micro-benchmark Results

Table 6 above shows the execution time for $n=16$ threads on an Intel(R) Xeon(R) CPU E5530/E5620 @ 2.40GHz (with 16 virtual cores).

5. Conclusions and Future Work

This paper described several problems and opportunities to be addressed by malloc libraries to provide high performance to multi-threaded applications on modern multi-core systems. The proposed solutions included optimizations for lowering the overhead of (de)allocations & accesses, minimizing locking & contention across threads, trading-off memory for improved latency, avoidance of system calls, and providing better control of memory foot-print. By demonstrating performance gains for these optimizations this paper established that

1. A good malloc library should strike the right balance between latency, concurrency and extra memory consumption.
2. Equally, *in the Yahoo! context, online applications can and should make conscious choices on optimizations* that are most relevant to their performance metrics, allocation and access patterns, hardware configuration and resource availability.

Making Jemalloc or Y-Jemalloc fully NUMA-aware is our primary future work.

References

[1] Evans, J, "A Scalable Concurrent malloc(3) Implementation for FreeBSD," *BSDCan 2006*, Ottawa, Canada, May 2006.

[2] Ghemawat S, "TCMalloc : Thread-Caching Malloc," http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[3] Bohra A, Gabber E, "Are Mallocs Free of Fragmentation?," In USENIX 2001 Annual Technical Conference: FREENIX Track.

[4] Feng Y, Berger ED, "A Locality-Improving Dynamic Memory Allocator," MSP 2005

[5] Berger ED, McKinley KS, Blumofe RD, Wilson PR, "Hoard: A Scalable Memory Allocator for Multithreaded Applications," ASPLOS 2000

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

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

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

[9] 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

[10] PostgreSQL Database Memory Management: http://www.postgresql.org/docs/8.2/static/spi-memory.html

[11] Torrellas J, Lam M.S., Hennessy J.L., "False Sharing and Spatial Locality in Multiprocessor Caches," IEEE Transactions in Computers, vol. 43, No. 6, June 1994.

[12] Sutter, H., "Eliminating False Sharing," Dr Dobbs Journal online (www.drdobbs.com), May 2009.

[13] Perache M., Jourdren H., Namyst R., "MPC: A Unified Parallel Runtime for Clusters for NUMA Machines," EuroPar 2008.

[14]. Reddy, J., "JeMalloc Optimization," http://twiki.corp.yahoo.com/view/Yst/JeMallocOptimization.