Scalable Memory Allocation

New York University

Rahul Manghwani and Tao He

Abstract

Memory allocation performance in single and multithreaded environments is an important aspect of many applications. Allocators such as malloc in the Solaris Operating System or ptmalloc in the GNU libc library work good with single-threaded applications. However, different approaches must be taken when designing new allocators optimized for a multithreaded application.

The creators of Google's Thread-Caching Malloc (TCMalloc) and Lockless's LLAlloc all claim they are delivering better performance in malloc / realloc / free speeds compared to glibc's malloc, especially under multithreaded environment. In this report, we developed the traditional malloc as well as a simplified thread-caching malloc. Further, we compared their performance with Google’s TCMalloc. Then, our focus switched to studying how TCMalloc and LLAlloc made memory allocating more scalable. Finally, we benchmarked those allocators to elucidate their strength and bottlenecks.

Contact: Tao He (th1133@nyu.edu), Rahul Manghwani (rahulmanghwani@gmail.com)

❖ Introduction

A good memory allocator needs to balance a number of goals. Two of these most prominent goals are minimizing time and minimizing space usage. Speed is important for any malloc implementation because if malloc can get faster, thousands of existing applications having bottlenecks on dynamic memory allocation will get significant performance boost without the need to change any code.
To understand how TCMalloc and other memory allocators designed for multithreaded applications outperform traditional allocator such as glibc's malloc, we need to explore its internal mechamisms first, identify key drawbacks in traditional glibc's malloc, then design the new more efficient scalable memory allocators.

❖ Working of Traditional Malloc (glibc):

- Imagine the memory space as large continuous buffer.

1000 bytes free

The allocator needs to maintain a data structure to keep track of how much space is allocated and how much space is free. This data structure is tagged to the beginning of the block and therefore amount of actual free space is size of the block minus size of the data structure.

```
struct free_list {
  struct free_list* next;
  struct free_list* prev;
};
```

```
struct Header {  // Footer is the same structure as Header
   size_t size;
   int    flag;  // Can be omitted, use last bit of size as flag
};
```

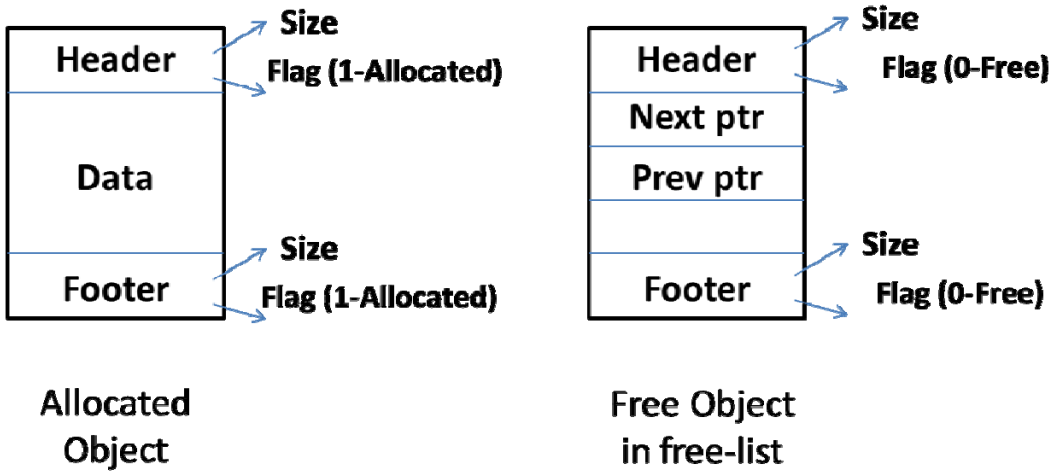
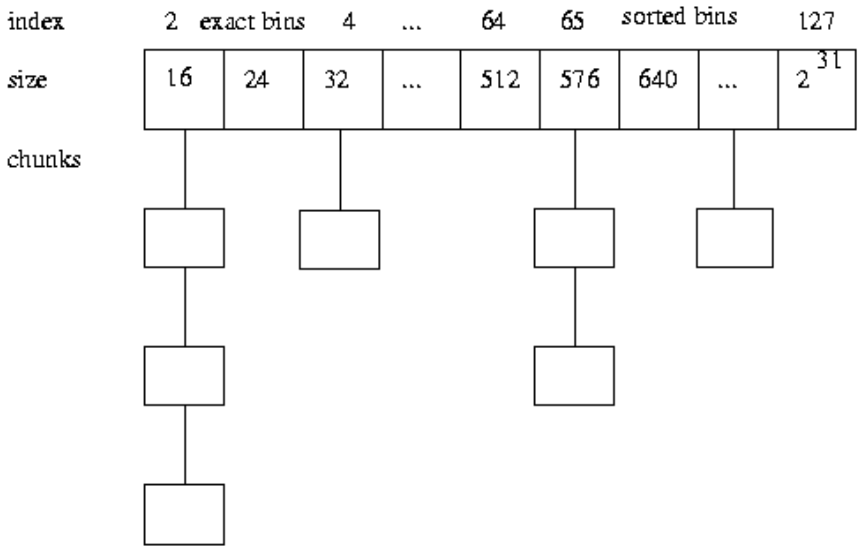


Figure 1

- **Structure of Free Lists**
  In order to reduce fragmentation, the memory is divided into fragments where each bin contains the chunks group by size. Each list will be a doubly-linked list. Not all the lists may be populated at any given time.

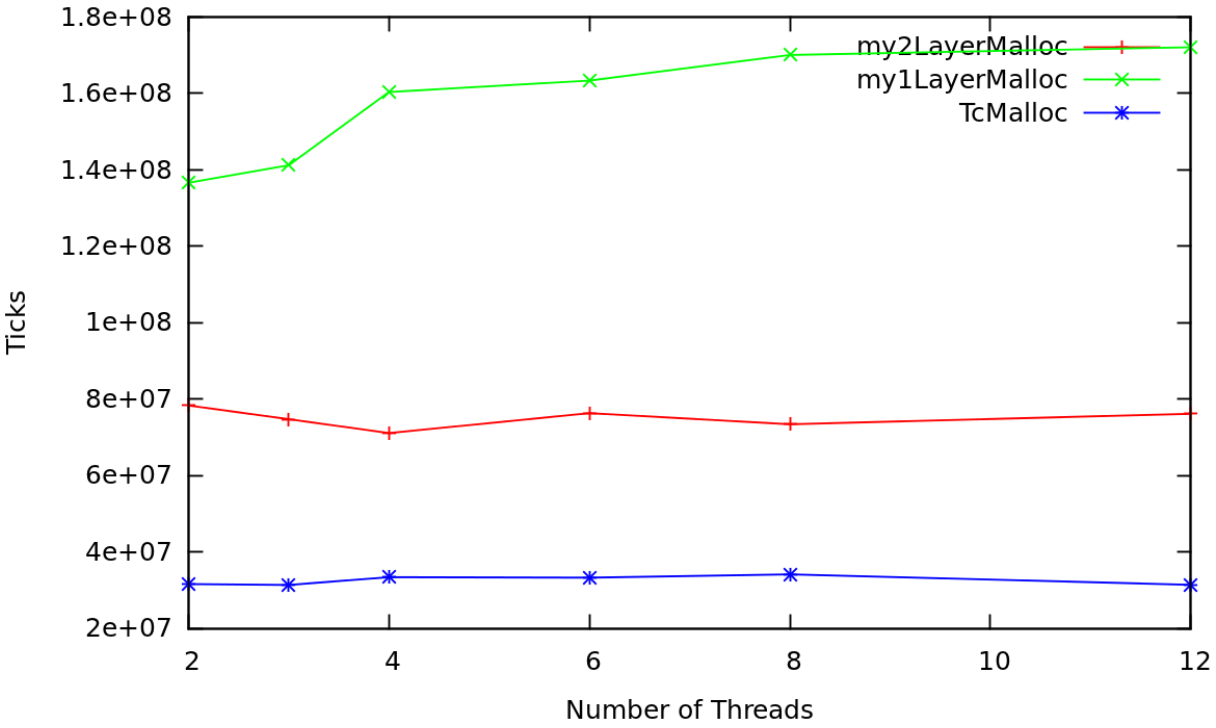


  Searches for available chunks are processed in smallest-first, *best-fit* order. Freed chunks are coalesced with neighboring ones, and held in bins that are searched in size order. Thus the general categorization of the algorithm is best first with coalescing.

- **Disadvantages of Traditional malloc and an improvement:**

- Lots of wasted space especially for small allocation objects: each Header / Footer occupies 4 bytes (in a 32 bit machine), if coalescing are adopted (without coalescing, only Header is required), every object must be surrounded by Header and Footer, so N-8 byte objects will account for 16*N bytes.
- If the size of a free object is bigger than required, but not big enough to carve into smaller objects, then there will be an internal fragmentation.
- No mechanism to separate small allocations with large (eg. requesting more than 200 MB memory) ones. Cannot adjust the bin size to speed up all kinds of allocations.
- In a multithreaded application, these data structures need to be protected with locks. As memory is being allocated concurrently in multiple threads, all the threads must wait in line while requests are handled once at a time. Therefore, all threads are competing for access to the same heap causing a problem known as heap contention.
- Adding a Second Layer (as a thread-cache) on top of the “Base Allocator” can greatly increase the scalability of the allocator (check the benchmark below).

- **A Comparison between 1 and 2 Layer traditional mallocs as well as TCMalloc:**
  The testing problem let N threads allocate a total of MAXALLOC bytes, each thread will work on its private allocation/free of (MAXALLOC / N) bytes:



❖ **Google's Thread-Caching Malloc (TCMalloc)**

➢ **TCMalloc's design to counter glibc malloc's shortages:**

- Memories are allocated by a run of pages instead of arbitrary sizes, thus greatly reduced sbrk or mmap system call overhead.
- Instead of using Header / Footer, a global page map was used to map between a given page to the location containing info about this page. For 64-bit architecture with 4K pages, 2^52 items have to be mapped. Therefore, a three-level radix tree was used to minimize memory cost, at the beginning, about 4.5 MB are used for the mapping.
- Separating the small allocation (<= 32KB) with big ones. Also, each thread gets a private thread cache, lock free multi-threaded allocation can be achieved if there is enough space in thread cache.
- Large object allocations are satisfied by the central page heap, the central heap is NOT thread-safe, so a spinlock has to be taken when allocating from central page heap.

➢ **Components of TCMalloc and our Implementation approach:**

- **Base class, system allocator:**
  This class controls the directly memory allocation from the underline system, in our approach, we used sbrk() call to allocate memory by 16K each time, the interface:

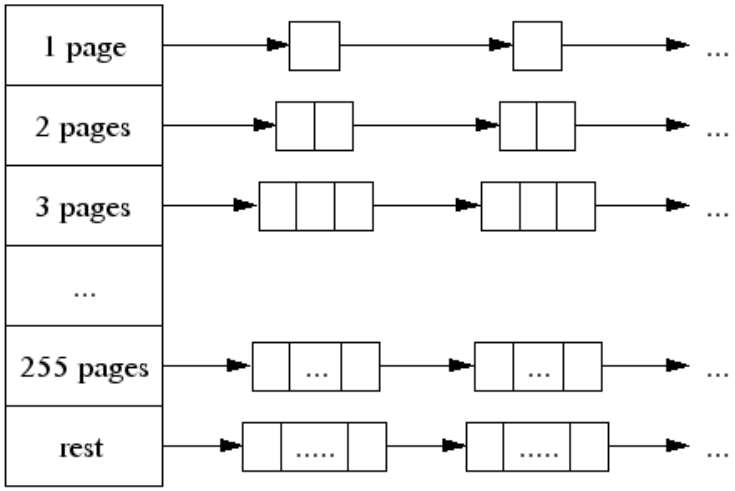
```
class sys_alloc {
public:
  void* getMemFromOS(size_t size, int alignment = 0);
```

```
  void releaseToOS(void* startpos, size_t      length);
};
```

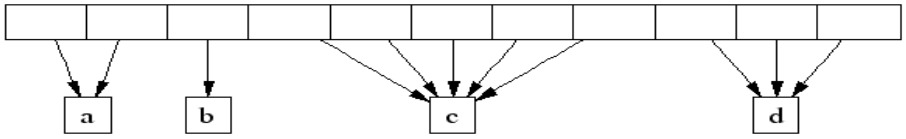
// “releaseToOS” should be called less frequently, as if later requests run out of memory from central page heap, the allocator has to fetch memory from OS again, involving more overhead.

- **Central Page Allocator:**

  Central page heap is the second lowest level of the tcmalloc, it allocates / deallocates in the granularity of pages. However, it's not thread-safe, multi-threaded malloc request has to modify the status of this heap under a spinlock.
  The internal structure of this heap is an array of free lists. A kth entry in the array represents the list of runs consisting of k pages. The 256th entry is a free list of runs whose length is >= 256 (Here, only this list is sorted in non-decreasing order).



- **Span Objects**
  Span contains information of a run of pages (whether it's being allocated or not. If allocated, whether this span belongs to a single larger object (>32K) or has been split up into several small objects (then store the correspond size-class info in the span as well))

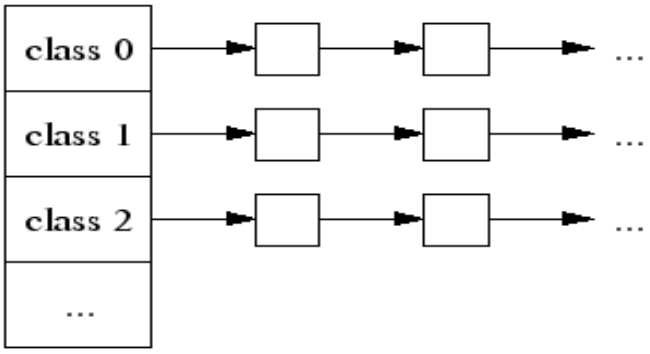


eg. Span a,b,c,d occupy 2 pages, 1 page, 5 pages and 3 pages respectively.

- **Heap Map**
  Heap map maps pages to locations containing the information of this page (a span object). For 32 bit machines, this map can be maintained using an array of size 4 MB (2^20 entries, each 4 bytes). However, 64 bit machines need a specialized logarithm structure (3 level radix tree, with 2^16 nodes on root level and 2^18 nodes on the second

and third level, thus at start up, this heap map will accounts for 4.5MB memory usage) to fit the representation in the memory.

- **Thread Specific Free List:**
  Thread Caching Malloc reduces heap contention by allocation each thread a thread local cache.
  A thread cache is divided into various size classes where a size class represents objects of the given size. These size-classes are spaced so that small sizes are separated by 8 bytes each, larger sizes are represented 16 bytes and so on. Each size-class contains a singly linked list of free object per size class.



Thread - Specific Free List

- **Central Free List:**
  All the threads share a common central free List. Each central free list is organized as a two-level data structure: a set of span, and a linked list of free objects per span.
  An object is returned to a central free list by adding it to the linked list of its containing span. If the linked list length now equals the total number of small objects in the span, this span is now completely free and is returned to the page heap.
  A central free list is guarded by spinlock for concurrent accesses.

- **Allocation :**

  TCMalloc treats objects with size <= 32K as "small" objects: Below is the Small Object Allocation Algorithm:

  (1) Map size of the object to the corresponding size-class.
  (2) Look in the corresponding free list in the thread cache for the current thread.
  (3) If the Thread local free list is not empty, we remove the first object from the list and return it. Till this stage TCMalloc acquires no locks.
  (4) On the other hand if the Thread local free list is empty, then
    (a) Acquire the lock on central free list
    (b) Fetch a bunch of objects from a central free list for this size-class.
    (c) Place them in the thread-local free list.

(d) Return one of the newly fetched objects to the applications.

(5) If the central free list is also empty:

(a) We allocate a run of pages from the central page allocator.

(b) Split the run into a set of objects of this size-class.

(c) Place the new objects on the central free list.

(d) As before, move some of these objects to the thread-local free list

Large Object Allocation:

Allocation for k pages is satisfied by looking in the kth free list in the central heap. If that free list is empty, we look in the next free list, and so forth. Eventually, we look in the last free list if necessary. If that fails, we fetch memory from the system. If an allocation for k pages is satisfied by a run of pages of length > k, the remainder of the run is re-inserted back into the appropriate free list in the central page allocator.

- **Deallocation:**

The following algorithm is used for the deallocation of the object.

(1) Compute its page no

(2) Map the page no to one of the span object.

(3) Determine the type of object(small or large object).

(4a) If the object is small insert it into the appropriate free list of the current thread's cache. If the thread cache now exceeds a predetermined size (2MB by default), a garbage moves the unused objects from the thread cache into the central free lists. We also record the minimum length of the list 'L' since the last garbage collection. We use this past history as a predictor of future accesses and move L/2 objects from the thread cache free list to the corresponding central free list. This algorithm has the nice property that if a thread stops using a particular size, all objects of that size will quickly move from the thread cache to the central free list where they can be used by other threads.

(4b) If the object is large, we look at the neighbouring spans associated with this object and if those are free, then we coalesce them with the current span and insert into the appropriate position in the central page allocator.

❖ Further Improvements (Lockless's LLAlloc)

➢ LLAlloc's design to further improve TCMalloc:

- LLAlloc divides allocation into three categories by size (TCMalloc only introduced 2 groups). Only the largest allocation sizes (those above 256MB) are allocated directly in a mutual exclusive way.
- For Allocations above 512 bytes and below 256MB, the allocation was done by searching free blocks represented by a B-tree. This log(n)-time for lookup of b-tree blocks is fast and is being further improved by adding a cache in front of the b-tree

algorithms. Therefore, there are great improvements in allocating medium size (32K to 1MB) objects comparing to TCMalloc.

- LLAlloc also utilized a wait-free queue for synchronization. There is one queue for each thread. A thread freeing a block from another thread will place the block on the correct queue and quickly return. Thus, even in a producer-consumer alloc-free scenario, it won't have one thread cache starve but another thread cache overfull problem as in the TCMalloc allocator.

❖ Benchmarking

The test program forks a number of threads and performs a series of allocations and deallocations in each thread; the threads do not communicate other than by synchronization in the memory allocator. The memory allocators were benchmarked for execution time and memory usage.

➢ **Benchmark on basis of Execution Time:**

The graph shows the performance of different memory allocators for varying number of threads.

The X-axis denotes the maximum memory allocation size (in a single malloc call) in the form of (2^number) bytes. The Y- axis denotes the execution time in Ticks (TicksClock::Ticks).

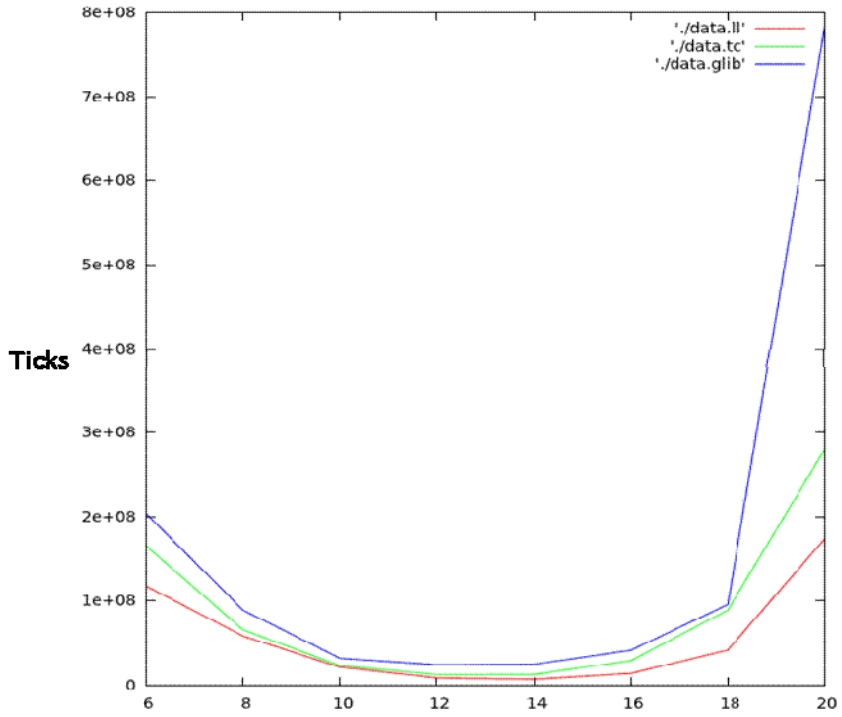
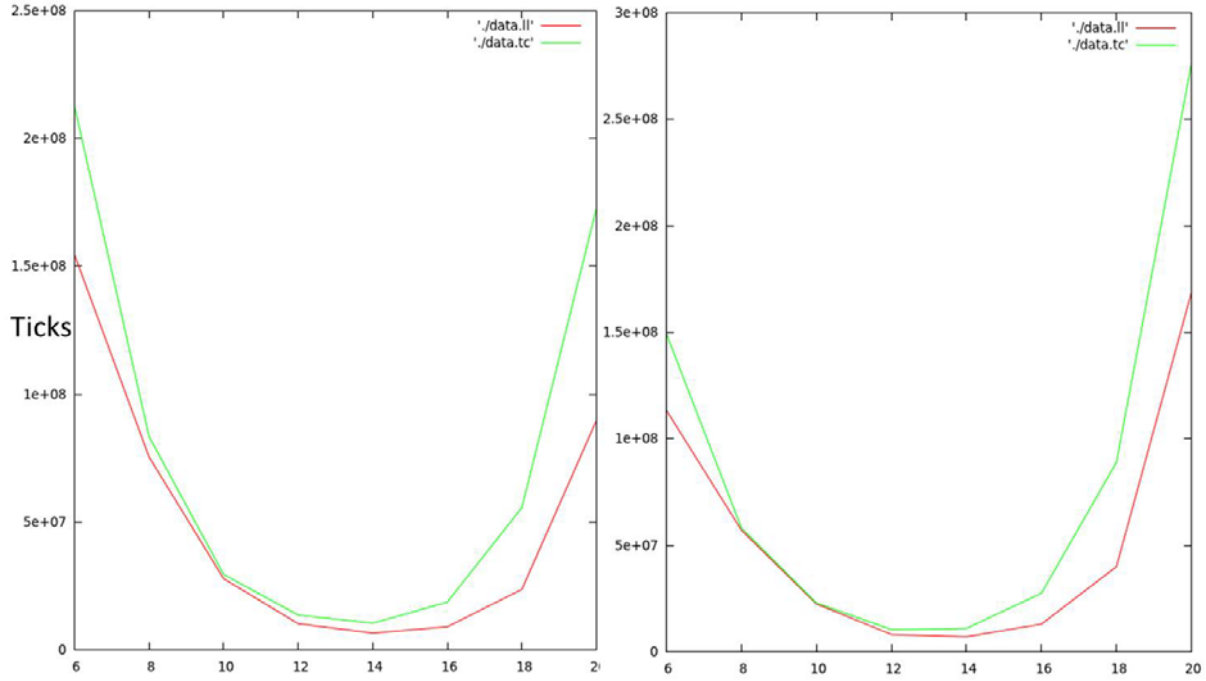
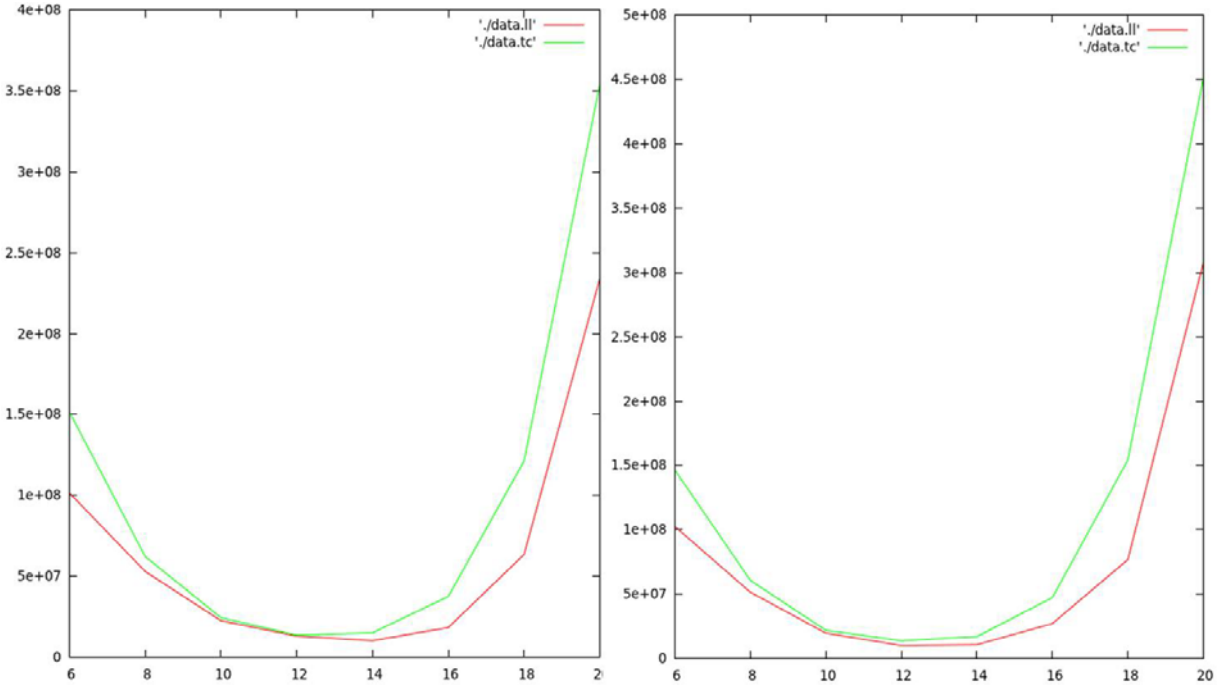


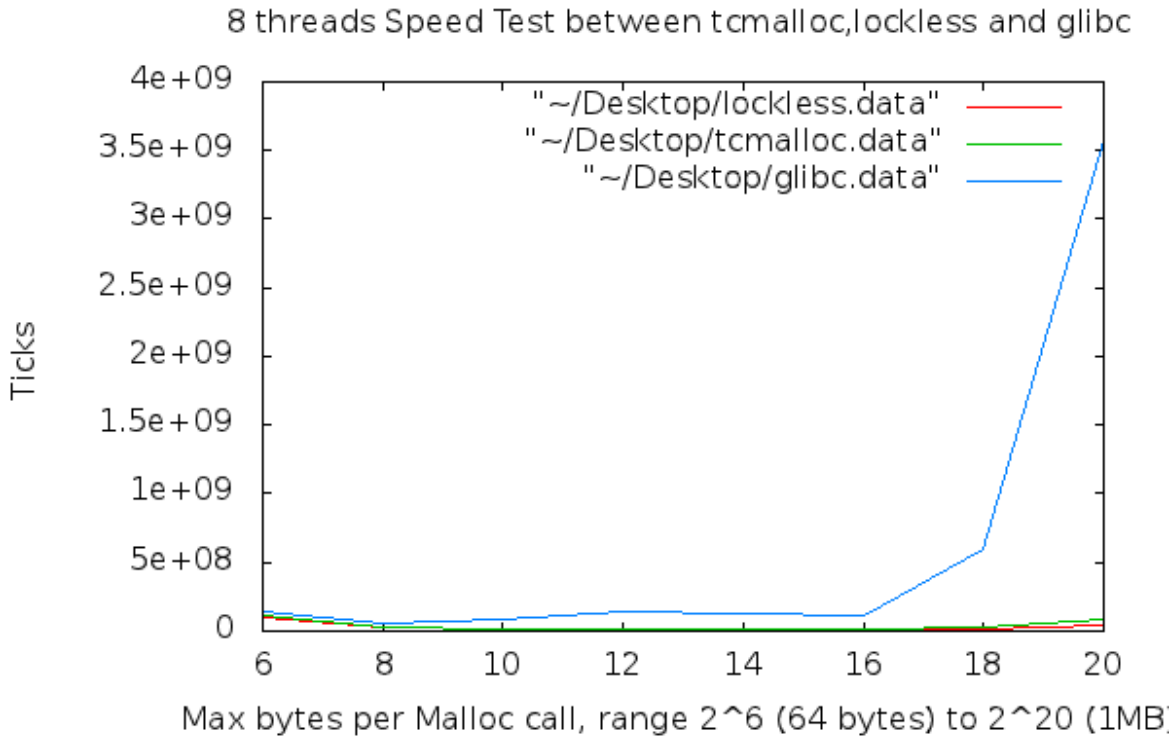
Figure 2: Two threads Speed test among glibc, tcmalloc(tc), llalloc(ll)

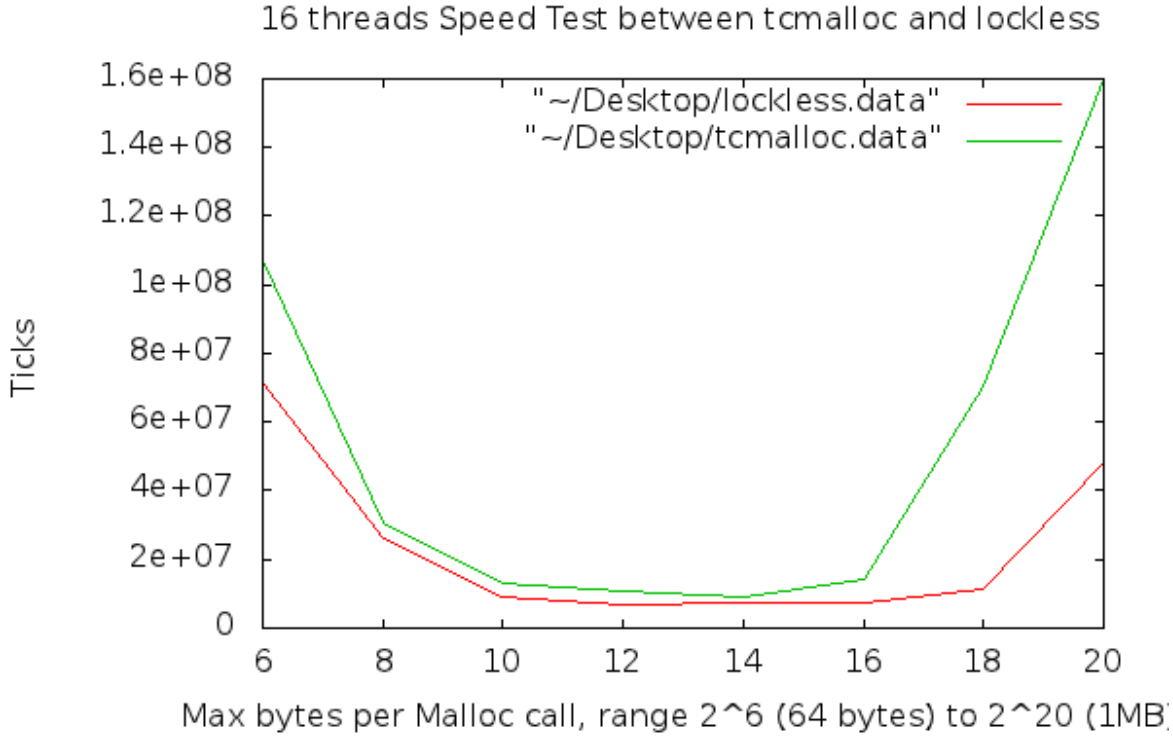


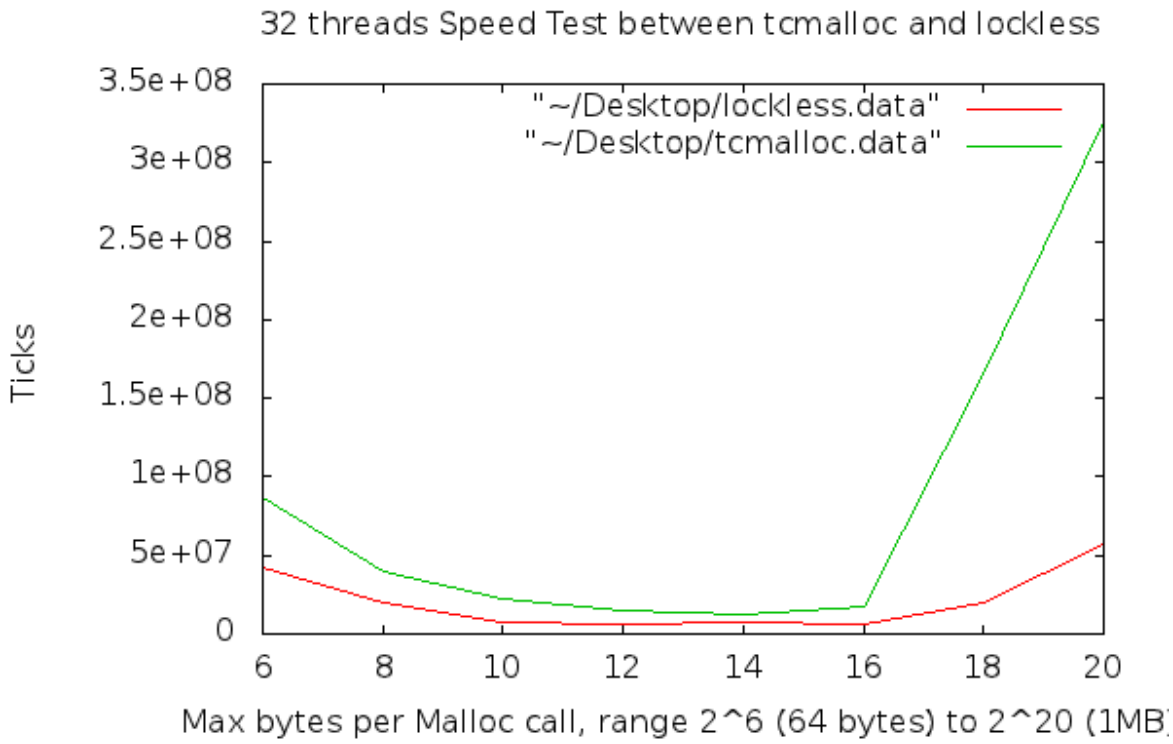
Max bytes per malloc call, range: 2^6 (64) bytes to 2^20 (1MB)
Figure 3: 1/ 2 threads Speed test among tcmalloc(tc), llalloc(ll)

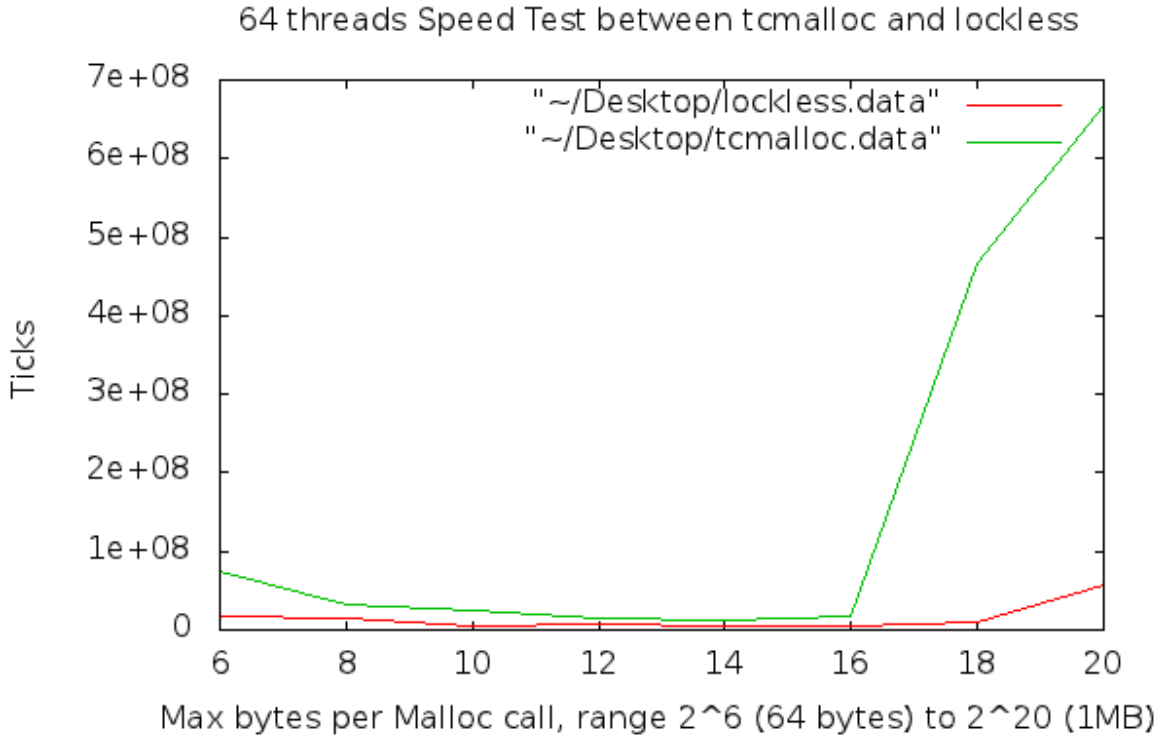


Max bytes per malloc call, range: 2^6 (64) bytes to 2^20 (1MB)
Figure 4: 3 / 4 threads Speed test among tcmalloc(tc), llalloc(ll)


8 threads Speed Test between tcmalloc,lockless and glibc
"~/Desktop/lockless.data"
"~/Desktop/tcmalloc.data"
"~/Desktop/glibc.data"
Ticks
4e+09
3.5e+09
3e+09
2.5e+09
2e+09
1.5e+09
1e+09
5e+08
0
6
8
10
12
14
16
18
20
Max bytes per Malloc call, range 2^6 (64 bytes) to 2^20 (1MB)


16 threads Speed Test between tcmalloc and lockless
"~/Desktop/lockless.data"
"~/Desktop/tcmalloc.data"
Ticks
1.6e+08
1.4e+08
1.2e+08
1e+08
8e+07
6e+07
4e+07
2e+07
0
6
8
10
12
14
16
18
20
Max bytes per Malloc call, range 2^6 (64 bytes) to 2^20 (1MB)


32 threads Speed Test between tcmalloc and lockless
"~/Desktop/lockless.data"
"~/Desktop/tcmalloc.data"
Ticks
3.5e+08
3e+08
2.5e+08
2e+08
1.5e+08
1e+08
5e+07
0
6
8
10
12
14
16
18
20
Max bytes per Malloc call, range 2^6 (64 bytes) to 2^20 (1MB)


64 threads Speed Test between tcmalloc and lockless
"~/Desktop/lockless.data"
"~/Desktop/tcmalloc.data"
Ticks
7e+08
6e+08
5e+08
4e+08
3e+08
2e+08
1e+08
0
6
8
10
12
14
16
18
20
Max bytes per Malloc call, range 2^6 (64 bytes) to 2^20 (1MB)

- **Observations:**

(1) TCMalloc is faster than ptmalloc and hence it is more scalable because there is contention problem between threads.
(2) Lockless allocator uses lock free techniques to minimize latency and memory allocation, therefore it outperforms TCMalloc, particularly for larger allocation sizes.
(3) TCMalloc's performance drops off as the allocation size increases. This is because the per-thread cache is garbage-collected when it hits a threshold (defaulting to 2MB). With larger allocation sizes, fewer objects can be stored in the cache before it is garbage-collected.

- **Benchmark on basis of Heap Usage:**

Memory Allocator	Maximum heap usage / kbytes
Glibc-malloc	3908
TCMalloc	4128
LLAlloc	3988

- **Observations:**
  (1) For the same memory allocation requests, TCMalloc's heap usage is higher than lockless because there is less contention for heap space and more heap request can be satisfied.
  (2) However, both TCMalloc and LLAlloc needs more heap space than traditional glibc malloc. This may due to the extra space required by the heap map for TCMalloc or LLAlloc's B-tree search structure. However, if we are allocating very small objects (eg. 8 bytes objects), then glibc's Header / Footer memory waste will show up, leading to more virtual memory usage.

❖ Other thoughts:

- **A good memory allocator also has to maximize locality:**
  That is, to allocate chunks of memory that are typically used together near each other. This helps to minimize page and cache misses during program execution. We will provide a real example in our presentation.

- **Security maybe another concern:**
  To make a good memory allocator for a sensitive system, security may be also a concern. Because the heap may contain secret information which should be inaccessible to the user, which requires the memory allocated
  (1) Cannot be paged to disk.

(2) Incredibly hard to access through an attached debugger.
Josh[5] suggested to use “Virtual Alloc” in windows to set protection on the memory space, therefore, making more restricted memory allocator.

❖ References:

1. http://gperftools.googlecode.com/svn/trunk/doc/tcmalloc.html, “TCMalloc: Thread Caching Malloc”, by Sanjay Ghemawat
2. http://locklessinc.com/downloads/, “Lockless’ LLAlloc”
3. “The Foundations for Scalable Multi-core Software in Intel® Threading Building Blocks”, Alexey Kukanov, Performance, Analysis and Threading Lab, Intel Corporation
4. http://gee.cs.oswego.edu/dl/html/malloc.html, “A Memory Allocator”, by Doug Lea
5. http://stackoverflow.com/questions/8451/secure-memory-allocator-in-c#27194, “Secure Memory Allocator in C++”
6. http://msdn.microsoft.com/en-us/library/windows/desktop/aa366887%28v=vs.85%29.aspx, “VirtualAlloc function”, Windows, Dev Center
7. "Computer Organization and Design, The Hardware / Software Interface", by Patterson and Hennessy, 4th edition.
8. https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919
9. “Hoard: A Scalable Memory Allocator for Multithreaded Applications”, Emery D. Berger Kathryn S. McKinleyy Robert D. Blumofe Paul R. Wilson, ACM 089791886/97/05