Reducing Active False-Sharing in TCMalloc

T. D. Crundall
Australian National University

Supervisor: S. Blackburn (ANU)
Collaborators: L. Angove (ANU)

TCMalloc

TCMalloc is an implementation of malloc. Malloc is used in C code to allocate memory. TCMalloc also handles calls to new and delete in C++ code. TCMalloc is optimised for high concurrency. It was written around a decade ago in order to better utilise mulit-processor machines running multi-threaded programs. The TC stands for thread cache. TCMalloc uses thread caches to allow concurrent threads to perform certain allocations without locks in shared memory. In practice these allocations generally make up the majority of allocations.

TCMalloc was a great solution for the time and we have found that despite not being actively maintained TCMalloc remains competitive with other concurrent malloc implementations.

Background

The word 'object' will be used throughout this report to represent the unit of memory passed around in the internal workings of TCMalloc and ultimately passed to the program through malloc calls. The word 'slot' will be used to specifically describe the memory passed around only in the internal workings of TCMalloc. A slot is a chunk of memory set aside for the use of an object but nothing is yet allocated to it.

False-sharing results when multiple threads share words on the same cache-line without actually sharing data. This seriously impacts performance on multi-core machines. Sometimes this is unavoidable, such as in the case of one thread allocating a number of objects which end up on the same cache-line only for the thread to pass them off to different threads. This specific flavour of false-sharing is unavoidable by the allocator and can only be guaranteed not to happen by padding every object allocated out to the cache-line size. This is unwise as it leads to massive levels of internal fragmentation.

Other times false-sharing is induced by the allocator when different threads allocate objects at different times only for those objects to be placed in such away that they share a cache-line. This is called active false-sharing.

Passive false-sharing is when the deallocation of an object being handled in such a way that the reallocation of the now free memory results in false-sharing. Both passive and active false-sharing can be avoided through the design of the allocator.

TCMalloc Implementation

TCMalloc carries out most allocations lock-free by providing each thread with a cache of allocatable memory for use by small objects. Small in this context means smaller than a page. TCMalloc has a different path for objects larger than a page and a lock on the central-heap is always required. A thread-cache's memory is kept track of through a set of free-lists. Each free list is associated with a size-class. A mapping is provided to take an object of a certain size to a size-class in such a way as to balance minimising internal fragmentation as well as reducing the overhead of the number of free-lists required to be kept track of.

TCMalloc is lazily initialised, hence the thread-caches are initialised as empty. As the user program allocates memory, the free-lists are populated with usable slots from the central heap. The central heap has a set of pages per size-class with slots already carved out. Each page only provides slots one size-class. When a thread tries to allocate an object but the associated free-list is empty, the thread grabs the lock for the central heap and grabs a number of slots and adds them to the free-list to await future allocations.

Other Implementations

Other implementations such as Hoard and Scalloc differ from TCMalloc in how memory is handled internally. In TCMalloc the central heap is responsible for carving raw memory into object sized chunks which are then passed to thread-caches for local use. With Hoard and Scalloc raw memory is passed to threads for local use; the threads are responsible for carving up the memory as required. We shall see that this is the major contributing factor to TCMalloc's false-sharing and a reduction of locality with small objects.

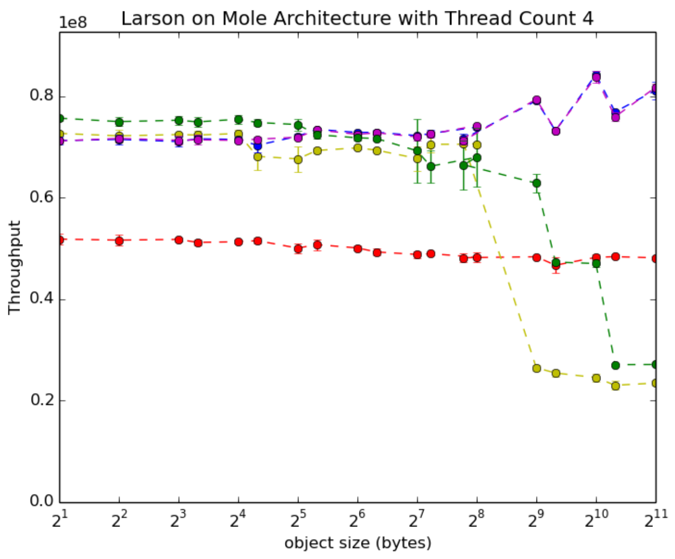
Benchmarks

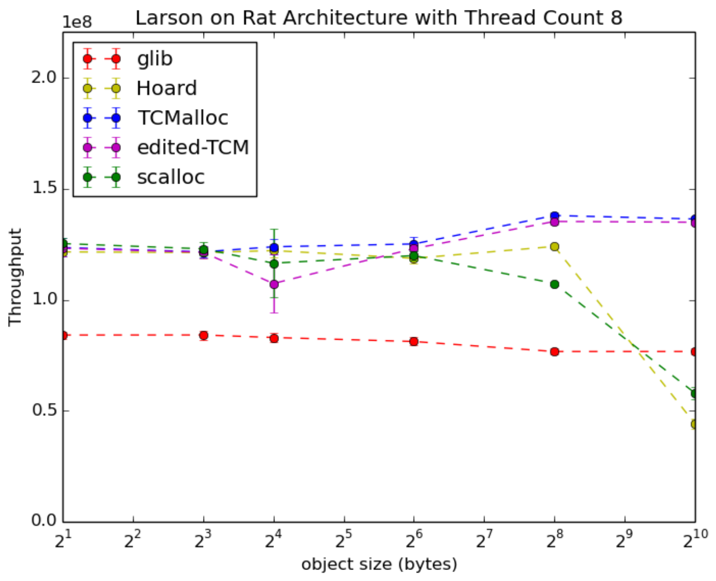
A few key benchmarks were used to compare the relative performances of the various implementations of malloc. Of particular use were cache-thrash, cache-scratch and larson. Larson is courtesy of Paul Larson at Microsoft Research. The other benchmarks used are part of the suite of benchmarks used in the paper introducing Hoard.

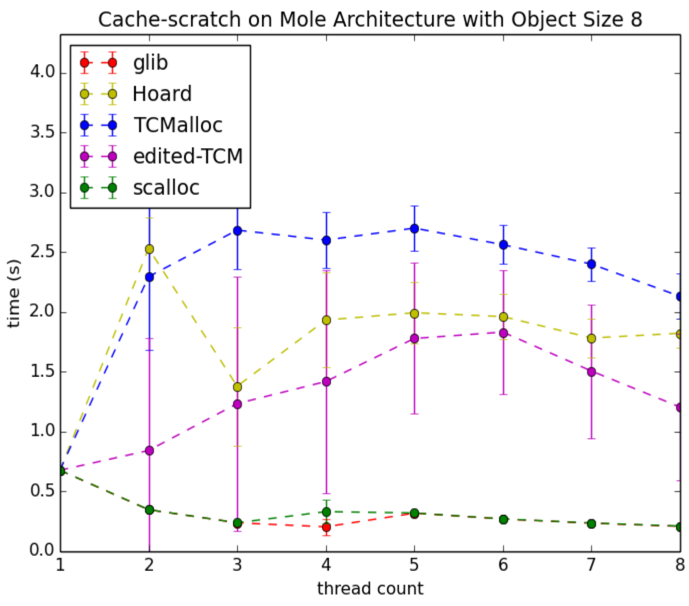
Cache-thrash tests active false-sharing. For a given object size, thread count and iterations, it allocates an object to each thread and soon deallocates it. Each thread thus only ever has at maximum one object allocated at any time. If the allocator is not inducing false-sharing then this all happens concurrently with high performance. However if an allocator is putting all the objects in the same cache-lines then the multi-core machine wastes time moving cache-lines between various cores' L1 caches.

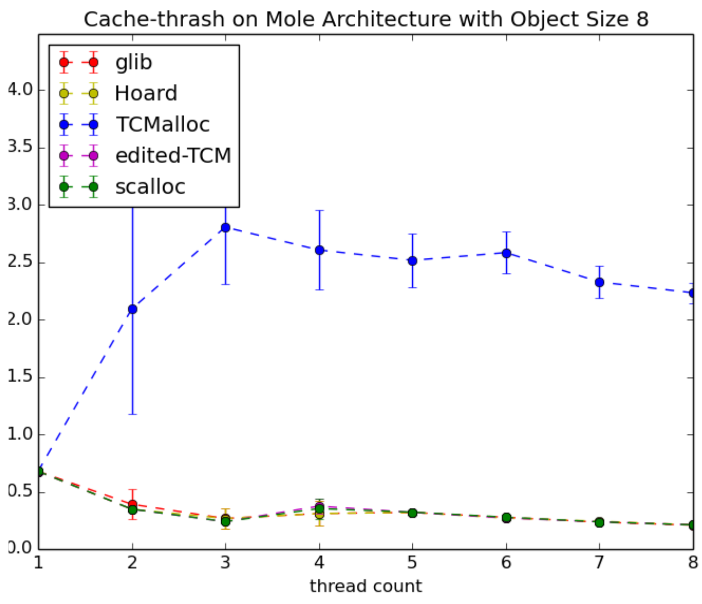
Larson simulates a server load with random allocations and deallocations of randomly sized objects within a certain range. We used Larson as an approximation of general performance.

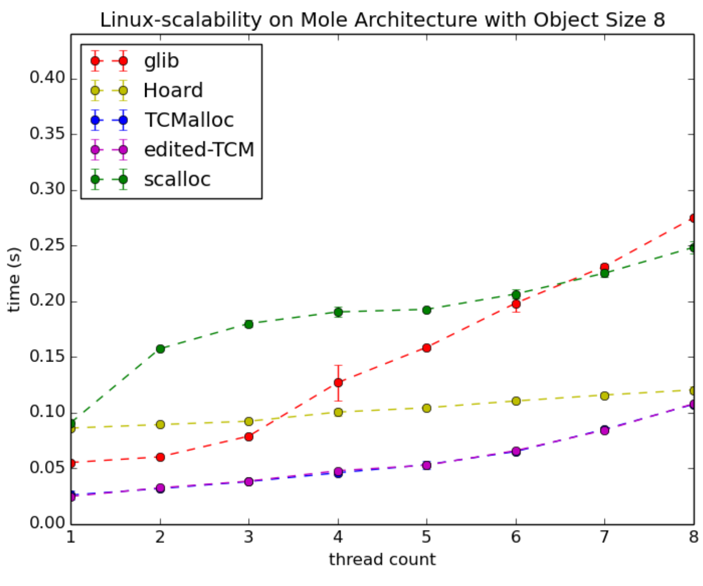
Cache-scratch looked at passive false-sharing. An allocator can be quite good at avoiding active false-sharing but still fall victim to passive false-sharing as we will see.

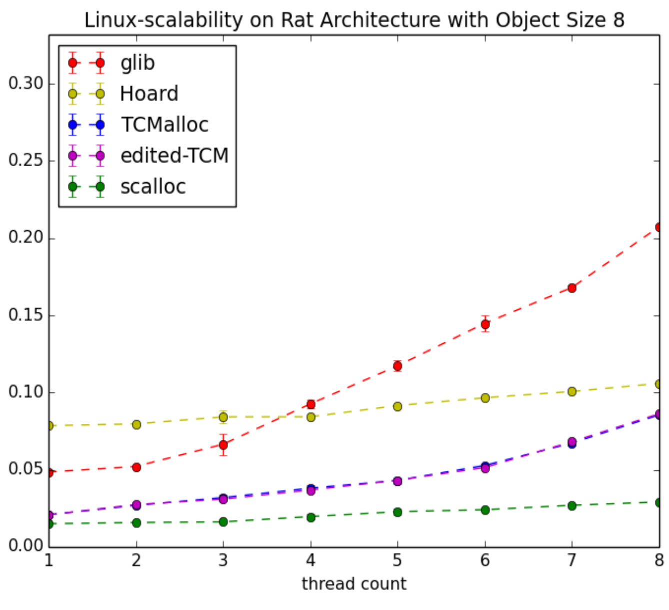

Larson on Mole Architecture with Thread Count 4
1e8
Throughput
object size (bytes)


Larson on Rat Architecture with Thread Count 8
1e8
glib
Hoard
TCMalloc
edited-TCM
scalloc
Throughput
object size (bytes)


Cache-scratch on Mole Architecture with Object Size 8
glib
Hoard
TCMalloc
edited-TCM
scalloc
time (s)
thread count


Cache-thrash on Mole Architecture with Object Size 8
glib
Hoard
TCMalloc
edited-TCM
scalloc
thread count


Linux-scalability on Mole Architecture with Object Size 8
glib
Hoard
TCMalloc
edited-TCM
scalloc
time (s)
thread count


Linux-scalability on Rat Architecture with Object Size 8
glib
Hoard
TCMalloc
edited-TCM
scalloc
thread count

My Efforts

As the Larson plot shows TCMalloc performs particularly badly with small objects. It outperforms the other implementations with large objects however. The focus of my benchmarking was around the various implementations relative performance under loads of predominantly small objects.

As can be seen TCMalloc performs particularly badly in the cache-thrash benchmark implying that TCMalloc results in active false-sharing. An instrumented version of cache-thrash was used to display where exactly the various objects were getting stored on the heap. It turns out TCMalloc crams the objects allocated by different threads right up against themselves. Hoard and Scalloc behaved sensibly by storing objects belonging to different threads many cache-lines apart.

Upon inspection of the TCMalloc algorithm it became clear that when a thread-cache is still small each time the thread-cache needs to be increased its length will only be incremented by one. In order to counter this I edited the algorithm to make sure the smallest amount of memory being passed from the central-heap to a thread-cache for any size-class was one cache-line. This is achieved by the central-heap passing a certain number of slots for that size-class calculated such that at least 64 bytes in total is being passed.

This edit greatly improved TCMalloc's performance on cache-thrash. However minimal change is observed in the edited TCMalloc's performance on Larson, which is used to approximate the general performance on a simulated real world load.

The lack of performance difference is attributed to the following fact. Even though the central-heap passes at least 64 bytes of memory to thread-caches, there is no guarantee that the memory is contiguous. Occasionally slots are returned from thread-caches to the central heap. Since the central-heap is structured as a free-list, the returned slots are simply prepended to the existing list, effectively shuffling the once ordered memory addresses into a random set.

Since memory isn't being passed as contiguous blocks, there is minimal control on locality and a high risk of false-sharing.

Future Work

A potential line of effort for significantly improving TCMalloc's performance with regards to small objects is to ensure memory is passed around internally as contiguous blocks at least the size of a cache-line. This problem can be solved by addressing two different parts.

The first part is ensuring memory is returned to the central-heap as contiguous blocks of memory. The second part is ensuring memory is handed out by the central-heap as contiguous blocks of memory. If the first part is achieved the second part becomes trivial. The solution is to ensure the total number of bytes handed out by the central-heap is exactly that of a cache-line size. In fact the second part is precisely what is implemented in the edited version of TCMalloc.

In order to ensure memory is handed to the central-heap as contiguous chunks an intermediate structure could be built that sits in between thread-caches and the central-heap. When a thread-cache is reduced in size the objects which are taken from the thread-cache are not sent directly to the central-heap. Instead they are sent to the intermediate structure. This intermediate structure gradually gathers the objects into cache-line sizes of memory. Only once a cache-line is complete would the intermediate structure return the now contiguous memory chunk to the central-heap.

A hash function could be used to send objects to certain bins where each bin corresponds to a certain cache-line of memory. The objects would wait there until their bin was full. A bit map could keep track of whether any bin is full and awaiting return to the central-heap.

The positives of this approach is reduced false-sharing and greater control of locality – the minimum size need not be a cache-line but could be increased to any size deemed appropriate and the intermediate stage could be expanded to handle larger objects as required.

The drawbacks include more internal time spend passing objects through various structures as well as more memory used on keeping track of everything. There is also a risk that a thread could hold onto one object while the remainder of the corresponding cache-line is sitting in the intermediate state unusable. This could be countered by encouraging thread-caches to hold their memory for longer.

Conclusion

There are many aspects of TCMalloc which I was unable to address. Particularly which aspects of TCMalloc result in its dominance in large object allocation.

Performance is always about trade-offs and as more and more computing is being performed on machines with greater number of cores, a high-performing concurrent malloc is an invaluable resource.

Footnotes

Specs of machines:

Mole: Core i7 2600, 3.4GHz, 4 processors
Rats: Intel i7-4770, 3.4GHz, 8 processors, 8GB RAM

Parameters of benchmarks:

Cache-scratch parameters:	[P 100 8 1000000]
Cache-thrash parameters:	[P 100 8 1000000]
Larson parameters:	[10 7 8 1000 10000 P]
Linux-scalability parameters:	[8 1000000 P]

References

Hoard: a scalable memory allocator for multithreaded applications; Berger, Emery D (2000)

Fast, Multicore-Scalable, Low-Fragmentation Memory Allocation through Large Virtual Memory and Global Data Structures; Aigner M, Kirsch C M, et al (2015)

TCMalloc documentation; Ghemawat S. and Menage P.

Computer Architecture: A Quantitative Approach; Patterson D. and Hennessy J. L. (1990)