SSMalloc: A Low-latency, Locality-conscious Memory Allocator with Stable Performance Scalability *

Ran Liu† ‡, Haibo Chen †
†Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
‡Software School, Fudan University
http://ipads.se.sjtu.edu.cn/ssmalloc

1 INTRODUCTION

Allocation latency, access locality and performance scalability are three key factors affecting the efficiency of a memory allocator for many cores. However, many previous state-of-the-art memory allocators focus one or two of them, making the application performance not satisfactory enough when the other factors become dominant.

Moreover, our investigation (detailed data in section 3) indicates that most state-of-art memory allocators cannot maintain stable scalability and locality when the number of threads exceeds the number of cores, which may be common in nowadays software. We believe that performance stability is equally important to the above three factors as it can provide more predictable performance for many applications requiring frequent memory allocation.

This paper presents a new memory allocator (called SSMalloc) that provide low-latency and locality-conscious memory management with stable performance scalability even with a large number of application threads. The key design decisions underlying SSMalloc include: 1) providing low and predictable latency for memory management operations through carefully minimized critical path; 2) minimizing *mmap* system calls that might be contended in kernel; 3) adopting lock-free and mostly wait-free algorithms.

APSys '12, July 23-24, 2012, Seoul, S. Korea

We have implemented a prototype version of SSMalloc and evaluated its performance and scalability against state-of-the-art memory allocators. Experiments using a number of commonly used allocation-benchmarks running on a 48-core machines show that for most cases SSMalloc outperforms prior systems in allocation latency and access locality, and provides more stable and scalable performance.

2 DESIGN OF SSMALLOC

Design Principles: The most important design goal of SSMalloc is stable scalability. Hence, SSMalloc should be lock-free. However, pure lock-free memory allocator is impossible in nowadays operating system. Multiple threads that perform VM management operations (e.g. mmap, munmap) simultaneously will contend in kernel [1]. In the OS kernel, these operations are usually being executed with a global lock held. The design of SSMalloc strictly limits the number of VM management system calls, whereby most of the possible kernel-level contention are avoided.

Another requirement is low and predictable latency. To achieve predictable latency, SSMalloc is designed to be nearly wait-free. Use of loops is strictly limited in lock-free synchronization. The other part of the algorithm could be done in bounded steps, where the critical path of SSMalloc is very short.

To achieve stable cache locality, all the data structures are carefully arranged. Fields frequently used together locally are grouped into the same cache line. Fields that may be read or write by multiple threads are separated in different cache lines to

*. This work was supoorted by China National Natural Science Foundation under grant numbered 61003002.

avoid false sharing.

Overall Structure: The structure of SSMalloc is showed in Figure 1. SSMalloc maintains a private heap for each application thread. A thread operates on its own heap for most requests, thus eliminates most of the synchronization. Each private heap holds several fixed-size memory chunks. A memory chunk contains many objects of the same size class.

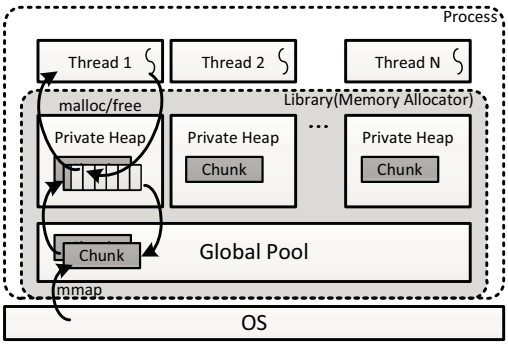


Figure 1: Overall Architecture of SSMalloc

When the thread private heap space is insufficient to fulfill a subsequent request, a free memory chunk is fetched from a global pool. If the thread private heap holds too many free memory chunks, some chunks will be reclaimed back to the global pool. Most of the operations on global pool is done purely by lock-free operations except for global pool enlargement. Note that we could easily eliminate this only lock by allowing different threads to enlarge the global pool simultaneously, However, it is useless since memory map operations will contend in kernel.

2.1 Memory Chunks

Memory chunk is the basic unit of the memory exchange between the private heap and the global pool. It is composed of a 65536-bytes data area and a 256-bytes header. Each memory chunk stores memory objects of the same size class.

Allocations requests smaller than 64KB are considered as small allocations, which are directly handled in the private heap without interfering with other threads. Small allocations are subdivided into several predefined size classes. A memory request is served with a memory block of the nearest size class. SSMalloc chooses three sets of size classes: Tiny (8B, 12B, 16B, 20B, ... , 60B, 64B); Medium (80B, 96B, ...,256B); and Large (384B, 512B, ... ,32768B, 49152B, 65536B).

Data area of the memory chunk is divided into clean area and dirty area. Memory in clean area has never been touched in the memory chunk. The header maintains a pointer to the clean area and a list of deallocated objects. During an allocation, SSMalloc first tries to pop a free object in this list. If the list is empty, memory will be allocated from the clean area by increasing the clean pointer. Designing the memory chunk includes several other considerations:

Uniform Memory Chunk Size for Memory Reuse: Memory chunk size is crucial to SSMalloc's performance. Smaller chunk size forces private heaps to perform more synchronization with the global pool to acquire the same amount of memory, thus suppress the scalability of the allocator, while larger chunks introduce more memory fragmentation, which result in poor spatial locality and comparably higher memory consumption.

A typical approach adapted by many other allocators is using larger chunks for larger classes. It ensures that memory chunks for every class contains enough available memory objects. In contrast, memory chunks in SSMalloc are of the same size for all the size classes. In most applications, small object allocation is far more frequently than large ones. This design naturally guarantees that a memory chunk for smaller size class contains more available memory objects. Further, this design brings several more benefits: 1) In the private heap, the size class for memory objects could be changed instantly without additional coalescing and splitting of memory areas. It allows memory chunks to be easily reused as other size classes within the private heap. 2) The design of the global pool is greatly simplified, which makes a lock-free global pool possible. 3) Indexing the metadata of a memory object became straightforward and can be done by a simple alignment operation, as described below.

Per-chunk Metadata to Ease Locating: Memory allocators often cluster metadata in a centralized area, which improves spatial memory locality by eliminating per-object header but make it harder to locate metadata during deallocation. Special hashtable or page-table-like structures are often used to index the metadata, which is usually costly.

In SSMalloc, metadata are placed in the per-chunk header. Spatial locality is still preserved since there is no metadata in the data area of a memory chunk. As memory chunks are placed one after another in heap space, locating metadata is as simple as aligning the address of memory object to the chunk boundary, which is a simple O(1) operation.

Unaligned Chunk to Reduce Cache Conflict: As described above, the size of memory chunks in SSMalloc is 65536 bytes + 256 bytes, which is not aligned to page size. Although it would be easier to manage memory resources if memory chunk size is aligned, all the headers would be placed at the beginning of memory pages, thus occupying the same cache sets. Accessing these headers will result in severe cache conflicts. We address this problem by assigning an unaligned size to memory chunks, which staggers cache lines used by different headers and consequently balances the load of cache access.

2.2 Private Heap

Each private heap belongs to an application thread. It is allocated when a thread performs its first allocation. All the small allocations are directly handled in it. This design is an important guarantee of overall scalability.

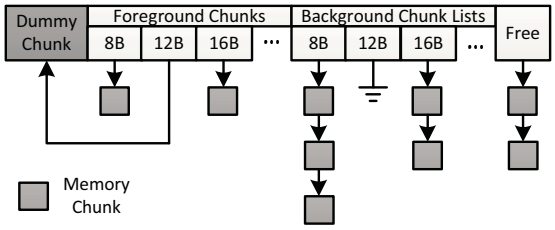


Figure 2: Private Heap in SSMalloc.

Figure 2 shows the basic structure of a private heap. A private heap maintains several memory chunks in different states as listed below.

Foreground Chunks are chunks currently in use, which are responsible for the next allocation. Hence there is at least one free object in each chunk. A private heap maintains one Foreground Chunk for each size class.

Full Chunks are chunks with no available memory objects. After all the objects are allocated, a Foreground Chunk becomes a Full Chunk and a new Foreground Chunk is selected.

Background Chunks contains one or more free objects. Freeing a single object in a Full Chunk makes it to be a Background Chunk. They are maintained in lists in private heaps.

Local Free Chunks are completely free chunks that are temporarily cached in the private heap. A Local Free Chunk could be directly converted into a memory chunk of any size class and reused locally. When the free chunk cache in private heap is overpopulated, additional free chunks are released to the global memory pool and become *Global Free Chunks*.

Dummy Chunk is a special pseudo chunk that contains a valid header and only one available memory object. When a private heap is allocated, all the Foreground Chunk pointers points to a single Dummy Chunk that resides in the private heap header. During the first allocation in a size class, the dummy chunk will be replaced by a real memory chunk. This trick eliminates the validity check of memory chunk in critical path.

2.3 Global Pool

Global pool manages memory chunks and private heaps in the lowest level of SSMalloc. All the memory chunks and private heaps are allocated from the Global Pool. Detailed structure of the Global Pool is illustrated in Figure 3.

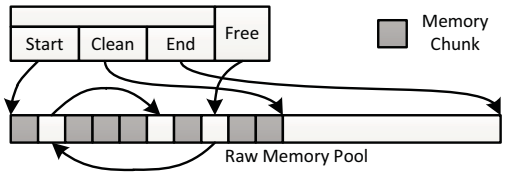


Figure 3: Global Pool in SSMalloc

Contiguous Raw Memory Pool to Reduce VM Footprint: Raw Memory Pool occupies a contiguous area in the address space, in which memory chunks are allocated next to each other. This design is TLB-friendly since it keeps virtual memory footprint of allocated memory as smaller as possible.

There's a pointer recording the begin of unused memory area of the Raw Memory Pool. Memory chunks are allocated by increasing this pointer by the size of a memory chunk, which could be done in one atomic instruction.

Raw Memory Pool is enlarged via mmap if its spare space is not enough. A lock must be acquired

before a thread tries to enlarge it. This is the only lock in SSMalloc. To reduce the number of mmap calls, the size of the Raw Memory Pool is enlarged exponentially each time. Besides, since the designated virtual address is specified in a mmap call, its latency is much lower. Though this approach seems not very economic in VM space, the physical memory consumption will not be affected since physical pages are allocated on demand the first time it is touched. In this way, the total time spent in mmap calls can be greatly reduced.

Global Memory Reuse: Globally freed memory chunks are maintained in a lock-free Global Free Chunk List. When a private heap tries to allocate a memory chunk from the Global Pool, it first tries to pop a free memory chunk from this list. In case the list is empty, a new memory chunk will be allocated from the Raw Memory Pool. Since all the memory chunks are of the same size, no further processing is needed on the returned memory chunks. In this way memory chunks can be reused among different local heaps.

Private Heap Reuse: After a thread is terminated, there may be several memory chunks remaining in the private heap. Rather than returning these memory chunks to the Global Pool, SSMalloc directly caches the whole private heap header of terminated threads in a lock-free list.

During the allocations of new private heaps, the lock-free list is checked first. Private heaps allocated from this list are comparatively mature ones as they have already owned a series of memory chunks. Reusing these private heaps reduces synchronization needed to allocate memory chunks to fresh private heaps. This mechanism minimizes the overhead of thread creation in SSMalloc and further guarantees the scalability of applications creating and destroying a large number of threads.

2.4 Large Allocations

Large allocations in SSMalloc are directly forwarded to OS via mmap. A pseudo data chunk header (whose owner is a special value) is added before the beginning of each large memory object to record its related metadata. Since large allocations rarely happen in real applications, it seldom becomes a bottleneck of scalability.

2.5 Allocation Algorithm

Based on the discussion above, we summarize the allocation algorithm as the following pseudo-code.

Algorithm 1: Allocation Algorithm

```
   Input: size of the requested object
   Output: ptr to the allocated object
 1 class ← SizeToClass(size);
 2 if IsSmallClass(class) then
 3 |   cph ← CurrentPrivateHeap;
 4 |   Retry :;
 5 |   fc ← ForegroundChunk(cph,class);
 6 |   ptr ← AllocFromChunk(fc);
 7 |   if Full(fc) then
 8 |   |   ReplaceForegroundChunk(cph,class);
 9 |   |   if IsDummyChunk(fc) then
10 |   |   |   ResetDummyChunk(cph);
11 |   |   |   goto Retry;
12 |   |   end
13 |   end
14 else
15 |   ptr ← LargeAlloc();
16 end
17 return ptr
```

The critical path in the algorithm is very short and contains no synchronization. Furthermore, all operations in the private heap contains no loops. For an allocation request, the size class is first calculated from the requested size. If it is a small allocation, a memory object is allocated from the Foreground Chunk of that class. Otherwise the request is forwarded to OS via mmap.

2.6 Deallocation Algorithm

The algorithm is summarized as the following pseudo-code. Note that all chunks in SSMalloc are organized in a FIFO fashion so that data locality can be preserved.

Algorithm 2: Deallocation Algorithm

```
   Input: ptr of the object to be deallocated
 1 cph ← CurrentPrivateHeap;
 2 chunk ← ExtractChunk(ptr);
 3 if IsOwner(cph,chunk) then
 4 |   LocalFree(chunk,ptr);
 5 else
 6 |   if IsLarge(chunk) then
 7 |   |   LargeFree(ptr);
 8 |   else
 9 |   |   RemoteFree(chunk,ptr);
10 |   end
11 end
```

Memory object may be freed by threads other than its owner thread (i.e. Remote Free). These threads could not directly operate on the corresponding memory chunk. Similar to Streamflow [2], each memory chunks maintains an additional lock-free list in to temporarily store memory

objects deallocated be other threads. Other threads can insert memory objects to this list via a single atomic instruction. Even so, there's still one atomic instruction in each remote free, whose overhead is a little higher. Unlike Streamflow, SSMalloc caches remotely freed objects in the private heap of the freer thread. Therefore, multiple objects could be returned to the owner thread with a single instruction. This design improves scalability significantly for those programs who perform a lot of remote frees. SFMalloc [3] also caches remotely freed objects in the freeing thread, but its remote free algorithm is much more complex than SSMalloc, resulting in a much higher CPU consumption.

3 Preliminary Evaluation

All performance tests are performed on an 8 Six-Core (2.4 Ghz) AMD x64 system (48 cores in total) running Debian-Linux with kernel version 3.2.10. The machine has 128 GB of memory.

Other memory allocators we used for comparison include (1) the thread-safe allocator of glibc [4]; (2) TCMalloc from Google's performance tools [5]; (3) jemalloc from Facebook [6]; (4) Streamflow [2]; (5) SFMalloc [3].

We use several different types of workloads to study the performance of these allocators: (1) 197.parser from SPECINT-2000, a CPU intensive single thread benchmark; (2) espresso [7], an optimizer for programmable logic array; (3) splint [8], a tool for statically checking C programs; (4) shbench, a memory allocator stress test tool from MicroQuill [9]; (5) Larson benchmark from Larson and Krishnan [10]; (6) Recycle, a custom synthetic microbenchmark that allocate and deallocate objects simultaneously in multiple threads [2]; (7) WordCount, a MapReduce application from Phoenix 2.0.0 [11].

Most of the objects allocated in these benchmarks are small objects. It is a common characteristic of most applications using dynamic memory allocation. The benchmarks we chose represent several different memory reuse patterns.

Sequential Performance: We choose 3 applications for sequential performance evaluation. The result is summarized in Figure 4(a). All the execution times are normalized to the one with glibc malloc. SSMalloc outperforms all other memory allocators in *espresso*, *splint* and *197.parser*, due to the optimization for latency in SSMalloc. The critical path of SSMalloc is very short and contains no synchronization. Although SFMalloc, Streamflow, TCMalloc and jemalloc also eliminate synchronizations in critical path, the length of their critical path is longer than SSMalloc, especially for the deallocation routine.

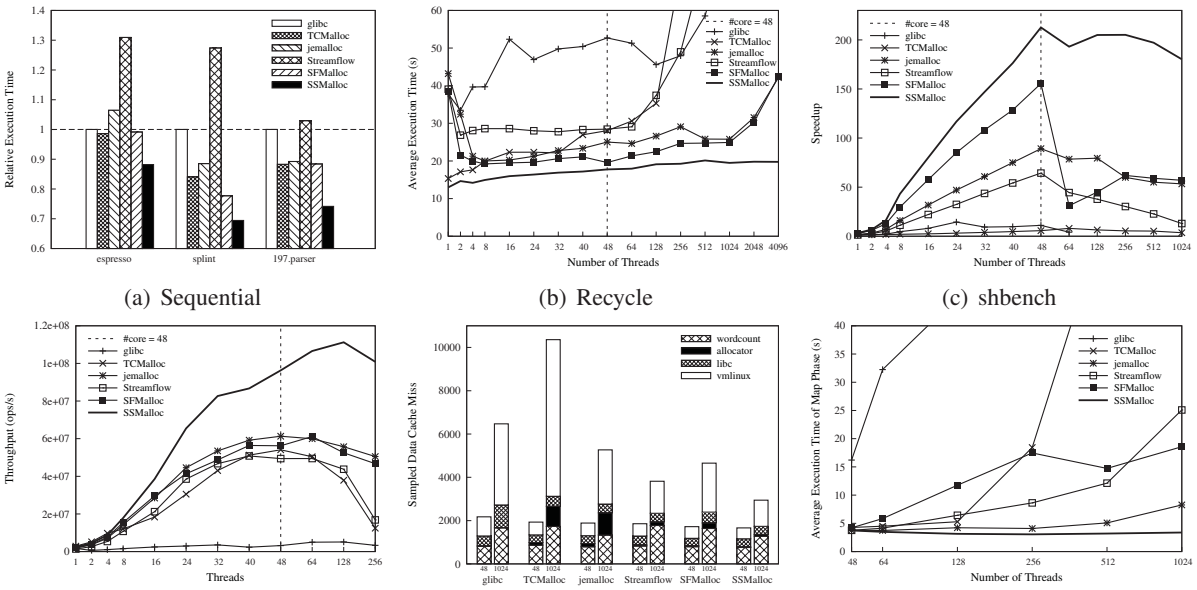
Multithreading Scalability and Stableness: Figure 4 show the performance scalability of all four multithreading benchmarks. We can see that SSMalloc notably outperforms other allocators in most cases. For *Recycle*, SSMalloc consistently outperform all the other memory allocators for at least 8.9% . It is mainly because SSMalloc has much lower allocation latency. Besides, since SSMalloc is designed to avoid kernel-level contention, the performance of SSMalloc stands stable when there are 4096 threads, while the performance of jemalloc and SFMalloc drop dramatically.

For *shbench*, the glibc allocator times out after 64 threads. Since the execution time varies largely among different allocators, performance is showed in speedup relative to the performance of glibc allocator with one thread. SSMalloc achieves at least 2X speedup to other allocators except SFMalloc. Again, SSMalloc is the most stable one when the number of threads exceeds the number of cores. SFMalloc manages to scale well before 48 threads since it is lock-free in user space. But its performance drops dramatically with more threads because of kernel-level contention. TCMalloc and the glibc allocator cannot scale in this benchmark because of lock contention on global data structures.

For *Larson*, SSMalloc scales better than all other allocator and achieves at least 24% higher throughput compared to other allocators when the number of threads is larger than 4. When the number of threads is larger than the number of cores, SSMalloc is much more stable than other allocators. It outperforms other allocators by 74% to 99% in this case.

Multithreading Locality Stableness: Figure 4(e) also confirms that SSMalloc experience much less cache misses that other memory allocators with both 48 and 1024 threads. This further contributes to a much less execution time of the Map phase, as shown in Figure 4(f).

Memory Space Efficiency: We measure the



(a) Sequential (b) Recycle (c) shbench

(d) Larson (e) WordCount cache-miss breakdown (f) WordCount time

Figure 4: Evaluation results of 5 benchmarks and the cache locality of WordCount.

Benchmark	glibc	TCMalloc	jemalloc	Streamflow	SFMalloc	SSMalloc
Recycle	242M	252M	272M	438M	227M	320M
shbench	33809M	21555M	21709M	24864M	21673M	21673M
Larson	285M	278M	448M	792M	1200M	915M
WordCount	1728M	1680M	1709M	1703M	1691M	1694M
espresso	1044K	2588K	3928K	6932K	2164K	1652K
splint	33172K	24860K	23964K	28500K	25868K	25580K
197.parser	10084K	10932K	8864K	21604K	9876K	10068K

Table 1: Physical memory consumption of each memory allocator

peak physical memory footprint of all the benchmarks for 48 threads, as shown in Table 1. SSMalloc's memory consumption for most programs are similar with other allocators. The memory footprint for Larson is larger as Larson measures the throughput in a fixed time. SSMalloc as well as SFMalloc caches remotely freed objects locally and thus incurs greater memory consumption.

4 Conclusion

In this paper, we have introduced an memory allocator SSMalloc. SSMalloc explored the design space of memory allocator for many-thread programs on many-core system. It simultaneously provided very low and predictable latency, stable high scalability, and stable cache locality. We have evaluated SSMalloc using 7 different workloads. Exprimental results showed that SSMalloc had notable performance advantage with sequential workloads and scaled much better with many threads than previous memory allocators. SSMalloc also provided stable cache locality with many threads.

References

[1] A.T. Clements, M.F. Kaashoek, and N. Zeldovich. Scalable address spaces using rcu balanced trees. 2012.

[2] Scott Schneider. Scalable locality-conscious multithreaded memor allocation. In *Proc. ISMM*, 2006.

[3] S. Seo, J. Kim, and J. Lee. Sfmalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *Proc. PACT*, 2011.

[4] W. Gloger. Dynamic memory allocator implementations in linux system libraries. `http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html`.

[5] Google. Tcmalloc: Thread-caching malloc. `http://google-perftools.googlecode.com/svn/trunk/doc/tcmalloc.html`.

[6] J. Evans. A scalable concurrent malloc (3) implementation for freebsd. *Proceedings of BSDCan*, 2006.

[7] R. Rudell and A. Sangiovanni-Vincentelli. Espresso-mv: Algorithms for multiple-valued logic minimization. In *Proc. IEEE Custom Integrated Circuits Conf*, pages 230–234, 1985.

[8] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *Software, iEEE*, 19(1):42–51, 2002.

[9] MicroQuill, Inc. `http://www.microquill.com`.

[10] P.Å. Larson and M. Krishnan. Memory allocation for long-running server applications. In *ACM SIGPLAN Notices*, volume 34, pages 176–185. ACM, 1998.

[11] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proc. HPCA*, 2007.