International Journal of Innovative Computing, Information and Control
Volume **14**, Number **5**, October **2018**
ICIC International ©2018 ISSN 1349-4198
pp. 1603–1615

OPTIMIZED MEMORY ALLOCATOR FOR HOT-SIZE ALLOCATIONS

Xiuhong Li and Gulila · Altenbek
College of Information Science and Engineering
Xinjiang University
No. 666, Shengli Road, Urumqi 830046, P. R. China
lixiuhong2016@163.com; gla@xju.edu.cn

Received December 2017; revised April 2018

Abstract. *Dynamic memory allocator is crucial to the overall performance of programs. This paper shows that current state-of-the-art memory allocators all adopt certain similar designs which are suboptimal and leave us great potential to do optimization and thus achieves better performance. By introducing a new memory chunk management strategy that puts priority on hot size classes, to postpone de-allocating free memory chunks in hot size classes and to build customized size classes for hot allocations, we are able to increase the rate of fast-path hit in malloc and at the same time reduce memory fragmentation. Experimental results show that our memory allocator can achieve an average speedup of 30% over Hoard, and 126% over TCmalloc, showing great potential to be a drop-in replacement of current state-of-the-art memory allocators.*
Keywords: Memory allocator, Hot size, Fragmentation, Hit rate

1. **Introduction.** Native programs (C or C++ programs) heavily rely on dynamic memory allocation [1, 2, 3]. In decades, *malloc* has become a widely adopted programming paradigm and it is often used very intensively in modern programs [4, 5, 16, 19]. Many studies have shown that dynamic memory allocation has been a bottle neck in programs and this calls for deep optimization [6].

As dynamic memory allocation (*malloc*) is a highly frequent operation that is executed in programs widely such as scientific programs [7] and data processing programs [8], its efficiency is thought to be extremely critical [6, 9]. In this paper we mainly show that two common mechanisms adopted by state-of-the-art memory allocators (such as Hoard [9] and TCmalloc [10]) are suboptimal and leave us great potential to optimize them.

- Current state-of-the-art memory allocators all adopt a similar hierarchy design that consists of a fast path allocation and a slow path allocation [6, 11]. For example, in Hoard, every thread has its own local heap and all threads share a global heap. Any memory allocation first goes to the local heap, and if the space of the local heap is not sufficient, the local heap will ask for new chunk of memory from the global heap. When an allocation is served in the local heap, its speed is fast (we call this fast path allocation, or fast path hit), while when an allocation cannot be served directly by the local heap, it is a time-consuming allocation due to the successive allocation of new chunks from the global heap (we call this slow path allocation, or fast path miss). Under this hierarchy design, trying to maximize the fast path hit rate is a core way to improve the allocation speed and thus the overall performance of the programs. However, all previous memory allocators all adopted a simple and static strategy to manage empty chunks between local and global heaps, which is suboptimal and can be improved much. In Hoard [9], for example, if the local heap

DOI: 10.24507/ijicic.14.05.1603

is less than half-full, it then tries to return free chunks to global heap. The half-full threshold is set fixed during the whole program execution period and there may be situations that the memory chunks are frequently moved between the local heap and the global heap, causing severe performance lost.

- All state-of-the-art memory allocators adopt a size class design to manage memory chunks in each thread's local heap [6]. Memory allocations are rounded up to the next nearest size classes to be served. For example, in TCmalloc [10], each threads local heap has 88 size classes. Each size class is actually a free list of memory objects of certain size. Common sizes are 8 bytes, 16 bytes, 32 bytes, 64 bytes, 128 bytes and so on, which means, if, for example, an upper application wants to allocate 60 bytes, it will be rounded up to 64 bytes and be served by the free list of 64 bytes. This design can effectively reduce the metadata and improve allocation performance (poping and pushing a free list is so cheap [6]). However, as it will be shown in this paper, rounding up memory allocations to the next size class will cause waste of memory and thus lead to a performance lost due to memory fragmentation and TLB (Translation Lookaside Buffer) miss [12], especially in allocation-intensive applications.

Facing the two problems described above, this paper introduces an optimized memory management strategy which dynamically adjusts itself during execution. Our mechanism is motivated by a simple and very important finding: all applications are intensively using only a small number of size classes. For example, TCmalloc [10] has 88 size classes but only 3-5 of them are used intensively. Though different applications intensively use different size classes, we can always observe that there are hot size classes that are often used. Statistically, over 60% of memory allocations go to the hottest 3 size classes in the benchmarks we tested. Based on this observation, we can implement a dynamic chunk management strategy that puts priority on serving these hot-size allocations. When we find that if a thread allocates much of the hot-size objects, we tend to give it more free memory chunks and postpone recycling them, even if there is much free space in that thread's local heap. Moreover, for the hot-size allocations, we tend to construct customized size classes for them to reduce memory fragmentation. For example, the benchmark *shbench* allocates more than 12,000,000 objects and most of them are 64-180 bytes. These objects allocation will be served in the 128 and 256 bytes size classes and will cause a huge waste of averagely 90 bytes of memory per object, which may lead to severe memory fragmentation and performance degradation. To solve this problem, we can just construct customized size classes for the hottest allocations; thus, we can reduce the fragmentation problem effectively. In all, our optimized memory allocator design can achieve better memory management and performance by prioritizing the hot-size allocations. Experimental results show that our memory allocator can achieve an average speedup of 30% over Hoard, and 126% over TCmalloc, showing great potential to be a drop-in replacement of current state-of-the-art memory allocators.

The rest of this paper is organized as follows. In Section 2, we introduce the background of current memory allocator design and our motivations. We show the design and implementation of our optimized memory allocator in Section 3. Experiments are conducted in Section 4 and we conclude in Section 5.

2. Background and Motivation.

In this section we first give a brief introduction of the common design of current state-of-the-art memory allocators and their problems, and then we show the observation that motivates our optimized design.

2.1. **Common design in memory allocator.** Dynamic memory allocator is basically a runtime system that manages pools of free virtual memory for upper applications. It first requires continuous blocks of memory (here we call chunks) from the underlying operating system and then organizes and manages the chunks to serve upper allocations of small pieces. As the allocation speed is very important, current state-of-the-art memory allocators all adopt some similar designs to maximize the allocation speed.

First, as shown in Figure 1, all the memory chunks are organized hierarchically. Each thread has its own local heap which is the first level of the hierarchy and which may contain several memory chunks. Any allocation will first try to be served at the local heap. If the local heap has enough memory to serve that allocation, then it returns the allocated memory and we define this process as fast path. If the local heap has no enough memory to serve this allocation, it will first require a free memory chunk from the second level of the hierarchy which is the global heap, and then serve the memory allocation using the new allocated memory chunk. We define this as slow path. Here the memory chunk is a continuous block of memory that is allocated and deallocated between thread's local heap and the global heap, and between the global heap and the underlying operating system. This hierarchy design could accelerate memory allocation by isolating different threads and by serving most memory allocations via the fast path.

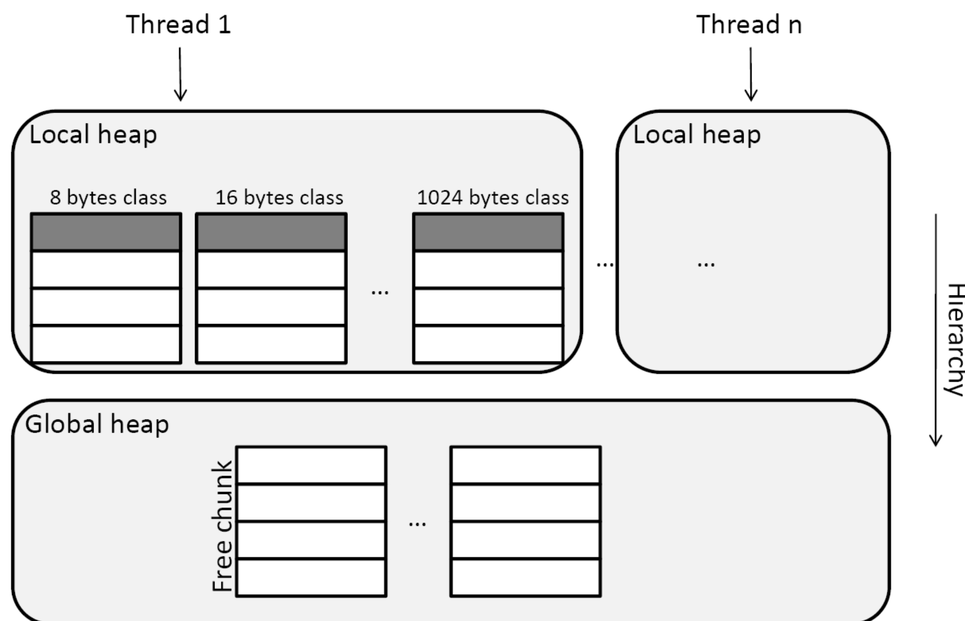

FIGURE 1. Hierarchy organization

Moreover, as shown in Figure 1, free memory chunks are organized into different size classes in the thread's local heap. A size class is actually a free list of pieces of memory that are all of the same size and that are waiting to be allocated. In the case shown, the sizes range from 8 bytes to 1024 bytes. Any allocation will be rounded up to the next nearest size to be served by one of the free lists. For example, an allocation of 10 bytes will be served by the 16 bytes size class. This design could greatly accelerate memory allocation of small objects.

2.2. **Problems in the common design.**

2.2.1. *Ping-pong effect.* As a hierarchy design is adopted, maximizing the fast path hit rate is core to performance. All previous memory allocators adopted similar heuristic

strategies to move memory chunks between thread's local heap and the global heap to try to maximize the hit rate of the local heap. However, as we will show in this section, previous heuristic strategies are all static which are simple and could not fit for general cases. For example, the popular memory allocator Hoard [9] adopted a strategy that if a thread's local heap is less than half full, it will then try to return free chunks to the global heap, which could then be used by other threads.

However, the static strategy could sometimes cause huge performance lost due to an effect we referred to as *ping-pong effect*. Figure 2 shows an example. A current state of the heap is shown in Figure 2(a). The size class of 16 bytes occupies two memory chunks. Then it serves a serial of *free* operations shown in Figures 2(b) and 2(c). After the *free* operations done in Figure 2(c), the free memory in the size class of 16 bytes is below a threshold (half full) so it tries to return free memory chunk to the global heap. This process is shown in Figure 2(d). However, just after returning the free chunk to the global heap, there may be a serial of *malloc* that forces it to allocate the free chunk back again (shown in Figures 2(e) and 2(f)). Thus a lot of time is wasted on moving the free chunk between the thread's local heap and the global heap and we call this *ping-pong effect*. Moreover, we have found that for the frequently allocated objects of certain sizes, this *ping-pong effect* is severe. In this case, the size class of 16 bytes is hot, and it suffers severe performance lost due to the *ping-pong effect*.

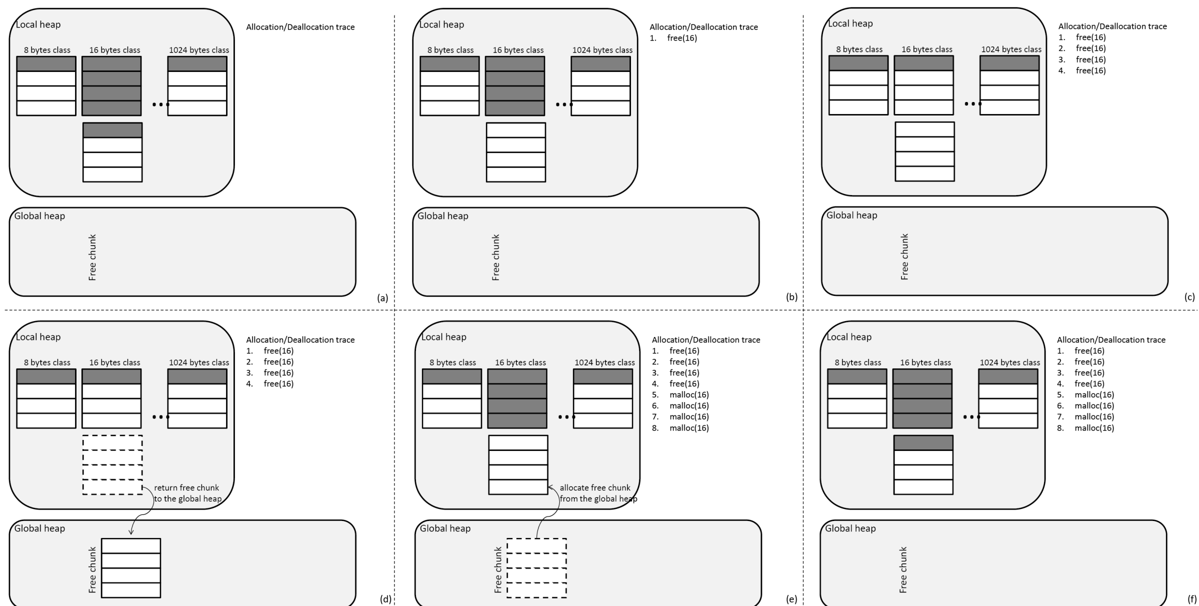

FIGURE 2. *Ping-pong effect*

2.2.2. *Memory fragmentation.* As shown in Figure 1, free memory chunks are organized into different size classes in the thread's local heap. Any allocation will be rounded up to the next size class to be served, and this is the main source of memory fragmentation that leads to performance degradation and extensive memory usage, especially for allocation-intensive applications (application that does a lot of *malloc*). Figure 3 shows the memory fragmentation problem in common memory benchmarks. Here we define the fragmentation ratio to be the amount of memory allocated to the applications divided by the amount of memory the applications actually use. We can see the fragmentation problem is severe in all benchmarks, especially in the benchmark *shbench*, which consumes 3 times more memory than it actually uses. We argue that this memory fragmentation

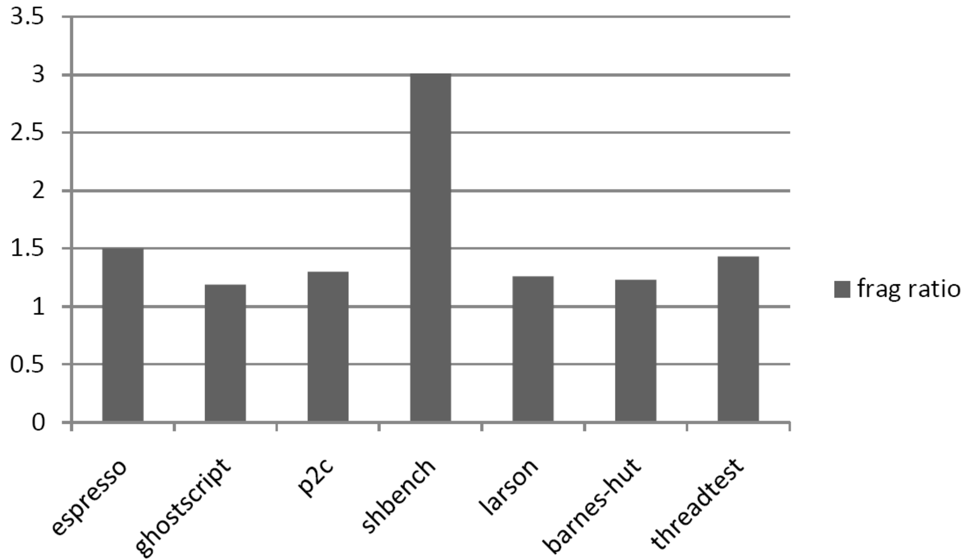

Figure 3. Memory fragmentation

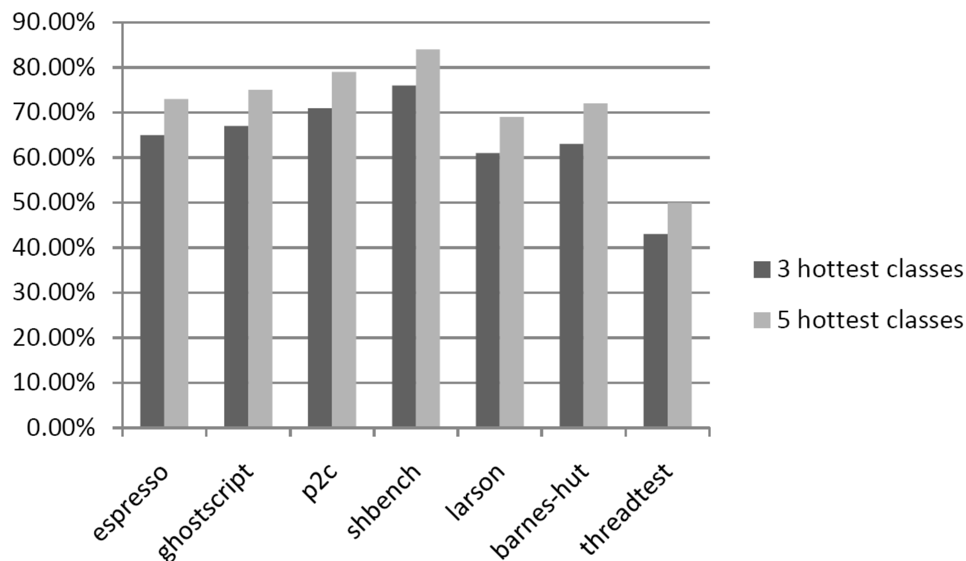

Figure 4. Hot size ratio

problem is becoming increasingly severe in modern 64-bit systems, where programmers always see virtual memory as an unlimited resource. Unlimited use of virtual memory will not only cause memory waste, but it will also lead to severe TLB miss, degrading the overall performance.

2.3. **Key findings and motivation.** This paper aims to address the two problems in current memory allocators that are discussed above. Our work is based on a key finding: applications tend to allocate many objects of the same sizes. Figure 4 shows the statistics. For all the benchmark we tested, we have found that they all make intense allocation of certain sizes.

As Figure 4 shows, for all the benchmarks, more than 40-60% of the memory allocations fall into 3 hottest size classes. Moreover, more than 50-70% of the memory allocations fall into 5 hottest size classes. We argue that most applications fall into this memory allocation

pattern and thus it indicates great opportunity for optimization. What we need is just to focus on the several hottest size classes and make optimizations accordingly.

3. Design and Implementation.

In this section we give the design and implementation of our optimized memory allocator that prioritizes the hot size classes and thus achieves better performance and less memory fragmentation. Based on our key findings introduced in Section 2.3, we can easily tackle the two problems introduced in Section 2.2 and thus behave better in terms of both performance and memory fragmentation.

3.1. Hot size identification.

As introduce in Section 2.3, most applications adopt the same memory allocation pattern that they are likely to allocate objects of the same sizes. Thus, in order to use this finding to do optimization, the first problem is to identify the hot size classes dynamically at runtime.

We added a special sampling step in the memory allocation process. The *malloc* steps are shown in Figure 5. The steps 1, 3, 4, 5 are original *malloc* steps that are adopted by previous work. The step 2 is our special sampling step that tries to identify those hot size classes dynamically.

For detecting the hot size classes, a naive way is to do the sampling all the time and calculate the ratio of every allocation's size. However, doing this may introduce two types of overhead. (1) The sampling overhead may be introduced constantly due to the constant sampling and ratio calculation. (2) It may take a long time to determine the top several hottest size classes and thus we lose opportunities to do optimization early. Here what

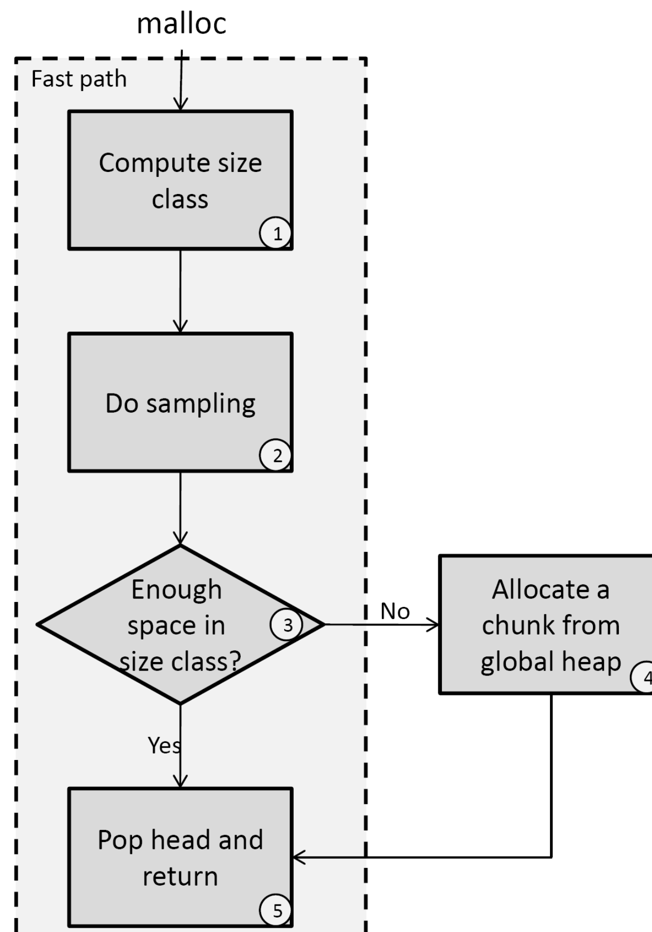

FIGURE 5. *Malloc* flow

we want is to identify the hottest size classes as early as possible and thus we can tackle the two overhead discussed above.

Our sampling method here is based on an experimental result of previous work [13]: intense memory allocations of the same size mainly happen in loops, and in the same execution context. Thus, we do not do sampling all the time. Instead, we focus on detecting intense memory allocations of the same sizes in loops, and in the same execution context. In detail, for each memory allocation, we record its size along with its execution context. We can use the backtrace mechanism [14, 15] to get the execution context easily. Only when we find the same size of memory allocated intensively in the same execution context, we mark that size hot.

To determine if the memory allocation is executed in a loop, we adopted a simple threshold method. We do recording that, for the first 1000 memory allocations, if an allocation of the same size in the same execution context takes up to 30%, we then regard that allocation as hot and in a loop. For all the applications, we only do sampling for the first 1000 allocations to determine the hottest several size classes. This method can help us reduce the sampling overhead and do optimization as early as possible. A problem is that this method may be not precise theoretically. However, as we tested in the experiment, it detects the hottest size classes perfectly. After all, if a memory allocation of the same size is executed more than 300 times in the same context, it is most likely in a loop. We have not seen exception for this case. Finally, in our current implementation, we only detect and record the hottest three size classes. As introduced in Section 2.3, they account for more than 40-60% of the whole memory allocations in programs and thus leave us great potential to do optimization.

3.2. **Optimized chunk organization and memory allocation.** After we got the hottest size classes, we can do certain optimizations accordingly to tackle the two performance hazards in the state-of-the-art memory allocators (discussed in Section 2.2). Here we introduce two chunk management strategies to tackle the two problems respectively.

3.2.1. *Postponed free.* Previous memory allocators all adopt statistic heuristics to allocate and de-allocate memory chunks from the global heap, in the hope of maximizing the fast path hit rate and at the same time reducing blowup (using memory unlimitedly). For example, the popular memory allocator Hoard [9] de-allocates a memory chunk in a thread's local heap if the corresponding size class is less than half full (case shown in Figures 2(c) and 2(d)). However, we have shown that this static heuristic may cause *ping-pong effect* and thus lead to a low fast path hit rate and severe performance lost (case shown in Figures 2(e) and 2(f)).

Here we argue that we can treat the hot size allocations specially to abate the *ping-pong effect*. The idea is, the hot size allocations are the main source of *ping-pong effect*. Those hot size objects are allocated intensively, and also de-allocated intensively. Thus they causes severe *ping-pong effect*. When we got the hot size classes, we can just postpone de-allocating free chunks of them. Thus, further intense allocations of these sizes are more likely to be served directly in the thread's local heap (via fast path). Here our strategy is:

- We do not initiatively de-allocate any free memory chunks in the hot size classes in a thread's local heap.
- If other threads or other size classes need memory chunks and the global heap has no memory chunks, we then choose one free memory chunk in the hot size classes and give it out.

- We choose the free memory chunks based on its hot rate. For example, if we maintained three hottest size classes, we always first choose one free memory chunk in the third hottest size classes and then the second, and so on.
- For other size classes that have not been marked as hot, we treat them as usual.

We will show later in experiment that our optimized chunk management strategy will greatly reduce *ping-pong effect* and increase the first path hit rate, improving performance of the whole program.

3.2.2. *Customized size classes.* As shown in Figure 1, current state-of-the-art memory allocators all adopt a design to organize free memory chunks into several size classes. The size classes have standard sizes ranging from 8, 16, 32, to 1024 or more. These sizes are chosen to be a standard and this is because it may be good for cache-level locality (they can be easily fit into a cache line or aligned by cache lines [17, 18]). Any allocation from the upper applications will be rounded up to the next size class to be served. However, as we discussed in Section 2.2, these standard size classes may cause severe memory fragmentation, which leads to excessive use of memory and performance degradation due to more TLB (Translation Lookaside Buffer) miss. For example, the benchmark *shbench* allocates a lot of objects of the 72 bytes and they are all served by the 128-byte size classes, which introduces 55 bytes (128-72) of memory waste per allocation. This problem gets more severe when the upper application makes intensive allocations of that unstandard size. We argue that this situation exists widely in most programs (as shown in Figure 3).

Here we offer a customized memory chunk design to solve this problem. When we are detecting the hottest size classes, we also record the hottest allocation sizes. Then when a thread's local heap allocates new memory chunks due to allocations of the hottest special sizes, we build customized size classes for them. For example, for the benchmark *shbench*, we will build a special size class of 72-byte. Thus, all the allocations of 72 bytes will be served directly by the 72-byte size class instead of the 128-byte size class and thus we reduce memory fragmentation. The code of *malloc* is shown in Figure 6.

```
void * malloc(size_t sz){
    do_sampling(sz);            //sampling

    if(size_hot(sz)){
        if(no_space_in_local_heap(sz)){
             \\get a free chunk from global heap
             \\then build a customized size class
             \\for this size
             get_customized_size_class(sz);
        }

        return malloc_customized(sz)
    }
    else
        return standard_malloc(sz);
```

Figure 6. Code of *malloc*

We will show in experiment that our customized size class design will greatly reduce memory fragmentation and thus improve performance and at the same time reduce memory usage.

Above all, the main novelty and contribution of this paper are as follows.

- We identify the *ping-pong effect* caused by eager de-allocation of such memory chunks, which is a common situation in previous memory allocators. We solve this problem by postponing the de-allocation to certain safe points, and thus reduce the performance overhead caused by *ping-pong effect*.
- We show that previous memory allocators all adopt fixed-sized memory bins to serve memory allocation, which may cause large portion of memory fragmentation. By introducing customized bins dynamically at runtime, we can reduce memory fragmentation significantly, and thus improve efficiency in terms of both time and space.

By the two new designs of our work, we make our memory allocator faster and at the same time consume less memory, which can be a drop-in replacement of the current state-of-the-art memory allocators.

3.3. **Implementation details.** We build our memory allocator based on the previous memory allocator Hoard [9] and we altered the key chunk management strategy in it. We argue that our design is not limited to certain implementations. It can be applied to other existing memory allocators as a drop-in replacement of the chunk management part. Here why we chose Hoard is because it is a quite popular memory allocator and thus is representative of the state-of-the-art technology. In the experiment we will do comparison with unmodified Hoard and other popular memory allocator such as TCmalloc [10].

In the implementation we adopt the traditional size of 65536 bytes for a memory chunk. This is the granularity for a thread's local heap to do allocation and de-allocation from the global heap, and for the global heap to do allocation and de-allocation from the underlying operating system. We adopt the Hoard's design to build 8 standard size classes (8, 16, 32, 64, 128, 256, 512, 1024) for each thread's local heap initially (TCmalloc has 88 size classes of the similar pattern), and then add our dynamic strategy that if we find any hot sizes, we build customized size classes for them.

Other unmentioned implementation choices are by default in Hoard.

4. **Experiment.** This paper introduces a better memory allocator design that tries to maximize the fast path hit rate of *malloc* and reduce memory fragmentation, which both help improve performance. In this section we mainly show every main aspect of our memory allocator and compare it with current state-of-the-art memory allocators.

4.1. **Methodology.** In the experiment we mainly test and show the main aspect of our memory allocator (here we call it HSmalloc for Hot Size malloc). We mainly show its fast path hit rate, memory fragmentation, memory usage, and the overall performance. As contrast set, we chose the memory allocators Hoard [9] and TCmalloc [10]. Hoard is a popular memory allocator that is specialized for parallel programs. TCmalloc is the standard memory allocator shipped with *glibc*. These two memory allocators can represent the current state-of-the-art technology.

The benchmarks we chose are all typical benchmarks (as shown in Figure 3) that have used to test memory allocators in previous work [9]. They all make intensive memory allocations and thus are proper to be benchmarks for testing memory allocators.

All the experiments are conducted on an Intel sever equipped with 32GB of memory running Linux 3.12. The Hoard we chose is at version 3.10.

4.2. **Result.** We first show the fast path hit rate of our HSmalloc compared with Hoard and TCmalloc. If a memory allocation is served directly in a thread's local heap, it is fast and we call this fast path hit. Higher fast path hit rate could imply better performance. The result is shown in Figure 7. We can see that all the three memory allocators achieve

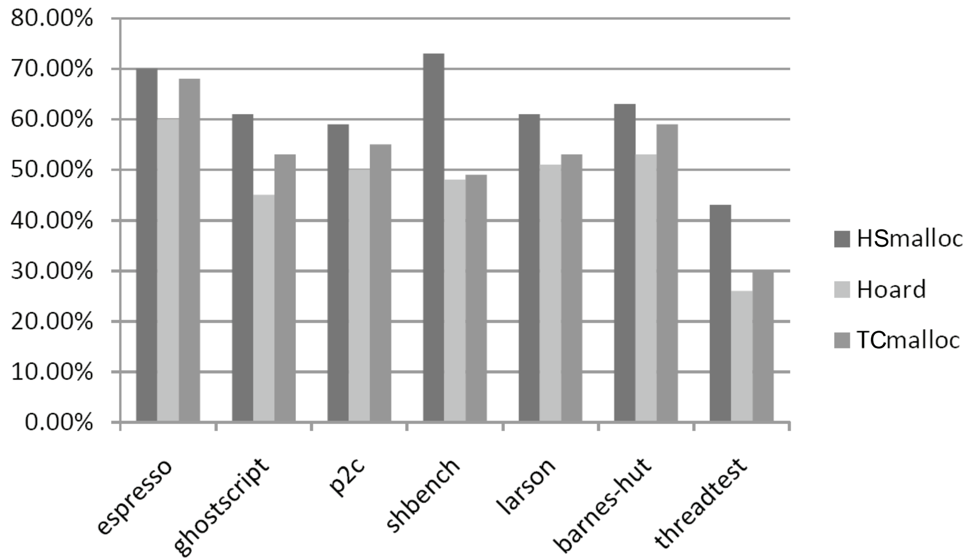

Figure 7. Fast path hit rate

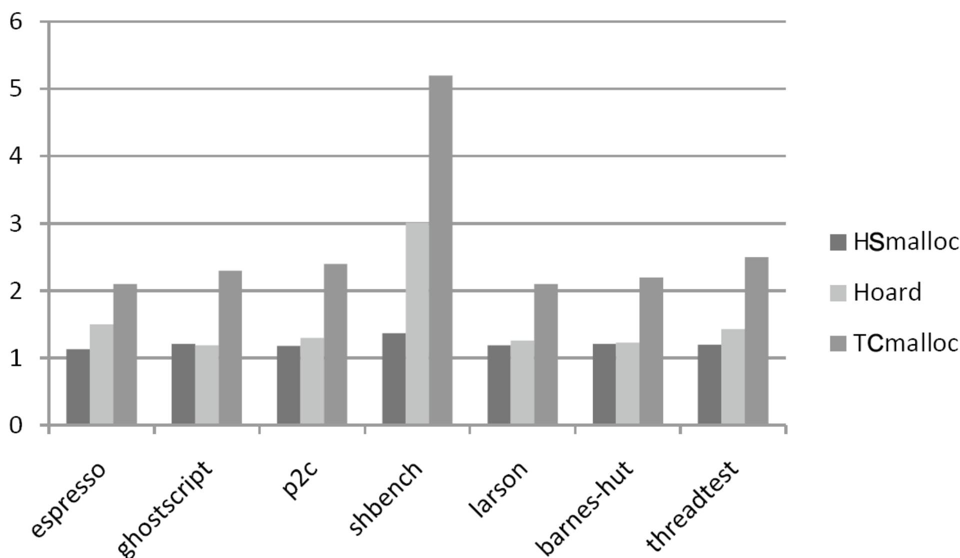

Figure 8. Memory fragmentation ratio

quite high fast path hit rate. Specifically, our HSmalloc achieves the highest rate of fast path hit. This is due to our dynamic adjustment strategy that puts priority on hot size classes. TCmalloc behaves better than Hoard and this is because TCmalloc organizes more size classes than Hoard (88 versus 8). Organizing more size classes is better for fast path hit rate but it may introduce more memory fragmentation. We will discuss this later. In all, our HSmalloc improves fast path hit rate by 10% to 25%, averagely 15% over Hoard and 10% over TCmalloc.

Second, we show the memory fragmentation situation of our HSmalloc compared with Hoard and TCmalloc. Here we define the memory fragmentation ratio to be the amount of memory allocated to the applications divided by the amount of memory the applications actually use. Higher fragmentation ratio means more usage of memory and implies more TLB miss, which will cause performance lost. The result is shown in Figure 8. Hoard and TCmalloc both introduce severe memory fragmentation, especially in the benchmark

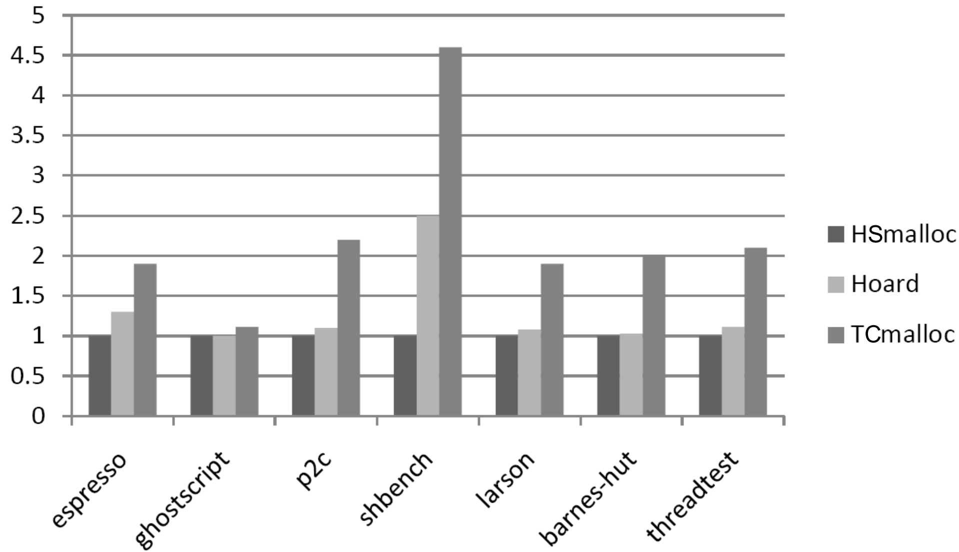

Figure 9. Normalized TLB miss

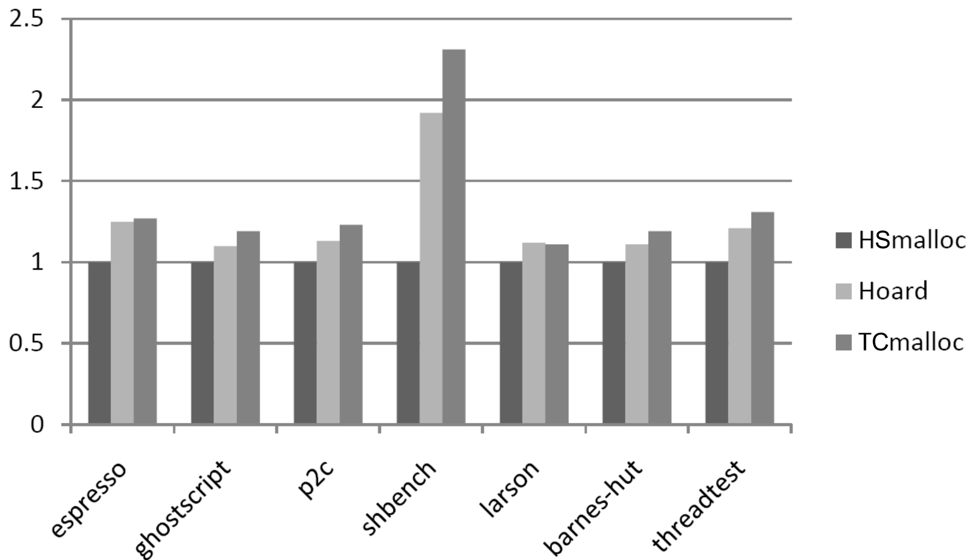

Figure 10. Overall normalized performance

shbench. We argue that the standard size classes are the main source of memory fragmentation. Notice that TCmalloc introduces obviously severe fragmentation, which is mainly because it organizes many standard size classes. Last, we can see our HSmalloc achieves a good control of memory fragmentation. It reduces memory fragmentation by 2% to 73%, averagely 22% over Hoard and 55% over TCmalloc.

Third, we show the normalized TLB miss. The result is shown in Figure 9. We can easily tell that the TLB miss is highly coherent with the memory fragmentation ratio. Hoard and TCmalloc both introduce times of more TLB miss over our HSmallc due to their high memory fragmentation ratio.

Fourth, we show the overall performance of our HSmalloc compared with Hoard and TCmalloc. Our HSmalloc could achieve better performance due to its higher fast path hit rate and less memory fragmentation. Figure 10 shows the result. Our HSmalloc performs better than Hoard and TCmalloc. Averagely, it outperforms Hoard by 30% and outperforms TCmalloc by 126%. Here the performance of TCmalloc being so poor is

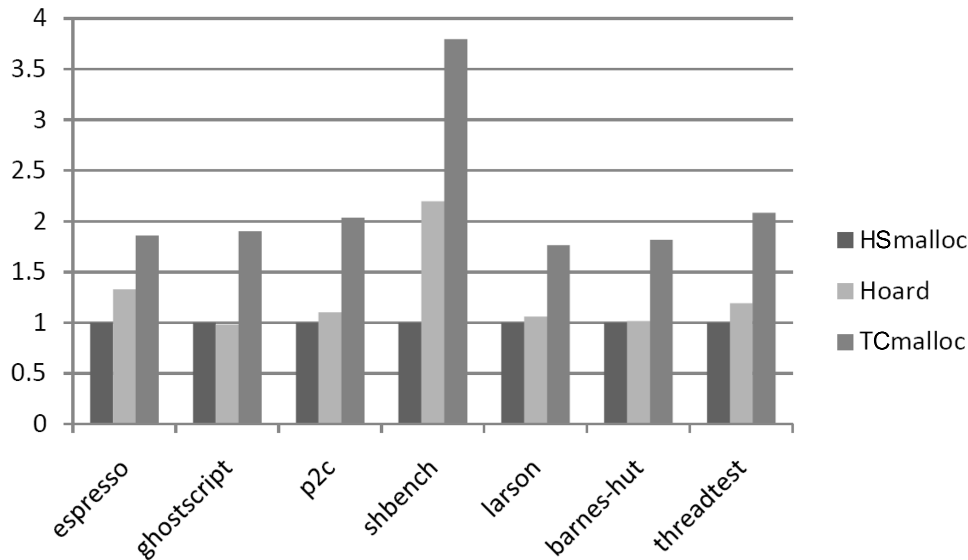

Figure 11. Normalized memory consumption over HSmalloc

mainly because it introduces much more memory fragmentation, leading to severe TLB miss.

Last, we show the memory consumption situation of the three memory allocators. The experimental result is shown in Figure 11. As we can see, Hoard and TCmalloc both consume much more memory than our HSmalloc, which is mainly because they introduce much more memory fragmentation. Comparing Figure 11 and Figure 8 we can find a direct relationship of this. Averagely, our HSmalloc consumes 26% less memory compared with Hoard and 117% less memory compared with TCmalloc.

Above all, our new memory allocator design can greatly increase the rate of fast path hit and reduce memory fragmentation, showing great potential to improve the overall performance of programs.

5. **Conclusion.** This paper introduces an optimized memory allocator design that increases the rate of fast path hit in *malloc* and at the same time reduce memory fragmentation. We introduce a new memory chunk management strategy that puts priority on hot size classes, to postpone de-allocating free memory chunks in hot size classes and build customized size classes for hot allocations. Experimental results show that our memory allocator can achieve an average speedup of 30% over Hoard, and 126% over TCmalloc, showing great potential to be a drop-in replacement of current state-of-the-art memory allocators. Future research will include adopting more sophisticated mechanism (such as some deep learning methods) to predict the memory access and allocation patterns of upper application and thus make better memory allocations accordingly.

Acknowledgment. The authors also gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the presentation.

REFERENCES

[1] S. Widmer, D. Wodniok, N. Weber and M. Goesele, Fast dynamic memory allocator for massively parallel architectures, *Proc. of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pp.120-126, 2013.

[2] B. C. Xing, M. Shanahan and R. Leslie-Hurd, Intel® software guard extensions (intel® sgx) software support for dynamic memory allocation inside an enclave, *Proc. of the Hardware and Architectural Support for Security and Privacy*, 2016.

[3] M. Danelutto, G. Mencagli and M. Torquati, Efficient dynamic memory allocation in data stream processing programs, *Intl. IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, pp.1181-1188, 2016.

[4] D. Dice, T. Harris, A. Kogan and Y. Lev, The influence of malloc placement on tsx hardware transactional memory, *arXiv:1504.04640*, 2015.

[5] G. M. Winn, *Heap Management Using Dynamic Memory Allocation*, US Patent 9,389,997, 2016.

[6] S. Kanev, S. L. Xi, G.-Y. Wei and D. Brooks, Mallacc: Accelerating memory allocation, *Proc. of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.33-45, 2017.

[7] H. Song, M. Yi, J. Huang and Y. Pan, Numerical solutions of fractional partial differential equations by using legendre wavelets, *Engineering Letter*, vol.24, no.3, pp.358-364, 2016.

[8] O. Adeyeye and Z. Omar, Maximal order block method for the solution of second order ordinary differential equations, *International Journal of Applied Mathematics*, vol.46, no.4, 2016.

[9] E. D. Beger, *The Hoard Memory Allocator*, 2014.

[10] S. Ghemawat and P. Menage, *Tcmalloc: Thread-Caching Malloc*, 2009.

[11] J. Evans, A scalable concurrent malloc (3) implementation for freebsd, *Proc. of the BSDCan Conference*, Ottawa, Canada, 2006.

[12] A. Bhattacharjee and M. Martonosi, *Inter-core Cooperative TLB Prefetchers*, US Patent 9,524,232, 2016.

[13] R. Guo, X. Liao, H. Jin, J. Yue and G. Tan, Nightwatch: Integrating lightweight and transparent cache pollution control into dynamic memory allocation systems, *USENIX Annual Technical Conference*, pp.307-318, 2015.

[14] D. V. Payne, D. A. Steffen, H. M. Ong, J. Molenda, K. S. Orr and K. B. Stone, *Queue Debugging Using Stored Backtrace Information*, US Patent 9,378,117, 2016.

[15] C. Liu, X. Yan, H. Yu, J. Han and P. S. Yu, Mining behavior graphs for "backtrace" of noncrashing bugs, *Proc. of the 2005 SIAM International Conference on Data Mining*, pp.286-297, 2005.

[16] H. H. Dam, Design of variable phase filter with minimax criterion and iterative method, *International Journal of Innovative Computing, Information and Control*, vol.13, no.3, pp.741-750, 2017.

[17] K. F. Munoz, H. McLeod, C. Pitt, E. Preston, T. Shelton et al., Do you know if your clients are having challenges coping, *Hearing Journal*, vol.70, no.1, 2017.

[18] R. F. Olanrewaju, A. Baba, B. U. I. Khan, M. Yaacob, A. W. Azman and M. S. Mir, A study on performance evaluation of conventional cache replacement algorithms: A review, *The 4th International Conference on Parallel, Distributed and Grid Computing (PDGC)*, pp.550-556, 2016.

[19] S. Liao, Q. Zhu and R. Liang, An efficient approach of test-cost-sensitive attribute reduction for numerical data, *International Journal of Innovative Computing, Information and Control*, vol.13, no.6, pp.2099-2111, 2017.