Efficient Memory Allocator with Better Performance and Less Memory Usage

Xiuhong Li, Altenbek·Gulila

***Abstract*—Dynamic memory allocator is critical for native (C and C++) programs (malloc and free for C; new and delete for C++). Current memory allocator adopts a common design that organizes memory into several bins of fixed sizes to achieve fast memory allocation and de-allocation. This paper shows that this design would essentially introduce large ratio of memory fragmentation and thus lead to memory waste and degraded performance, especially for allocation intensive applications. To address this problem, we introduce a dynamic strategy to create customized bins of special sizes. Unlike the standard bins in previous general memory allocators, the customized bins created in our memory allocator can better fit allocations of special sizes and reduce memory fragmentation and usage, thus improve performance and space efficiency. Experimental results show that our new memory allocator can achieve up to 1.7x speedup (averagely over 18.8%) over the state-of-the-art memory allocator Hoard with 22% less memory usage.**

***Index Terms*—Dynamic memory allocator, Customized, Fragmentation, Better performance, Space efficiency.**

I. Introduction

C and C++ programs heavily rely on dynamic memory allocation [1]–[5] and the related key functions have been an essential part of standard libraries (malloc and free for C programs, new and delete for C++ programs) [1]–[5]. Figure 1 shows a common case of using dynamic memory allocation in C/C++ programs (the malloc at line 3 and line 7). Memory allocators (e.g. tcmalloc in glibc) provide this neat interface (malloc and free) to ease programming with allocation of dynamic memory and mask all the memory management details. However, studies show that memory allocator greatly impacts the efficiency of programs [1], [2] in terms of performance and memory space, especially when the memory allocation and de-allocation are performed in an intense way (e.g. performed in a loop as shown at line 7 in Figure 1). More efficient memory allocators are in constant need for various situations.

Dynamic memory allocation has always been a hot area of research and there raised a lot of different allocation strategies focused on time (allocation performance) or space efficiency (less fragments) [1]–[4]. In recent decades, new allocation strategies have been raised to focus on parallel allocation [1], with the prevalence of multi-core architecture and multi-threaded applications. Popular memory allocators or allocator frameworks such as Hoard [1], Heaplayers [2], jemalloc [3], and Tcmalloc [4] all exhibit good performance as well as good scalability when executing with multiple threads. However, as we will show in this paper, after entering into the new era of 64 bits systems and with the widely adoption of big memory applications (or allocation intensive applications) [6], the fragmentation problem [7] which has not drawn much attention in allocator design is emerging to be a severe problem that will degrade both space and time efficiency.

More specifically, current memory allocators all focus on serving memory allocating and de-allocation at a fast speed and they all adopt a common mechanism that organizes virtual memory into several bins of fixed sizes. For example, the Hoard memory allocator [1] has bins of 32, 64, 128, 256, 512 bytes and so on. If we want to allocate a space of 33 byes, it will give us 64 bytes from the 64 bytes bin. Easily we can tell that this memory allocation mechanism is fast but there are 64-33=31 bytes wasted. Also, if we want to allocate a space of 100 bytes, the memory allocator will give us 128 bytes from the 128 bytes bin and leads to a waste of 28 bytes (128 bytes 100 bytes). This design behaves fine in the past for applications that allocate small amount of memory. However, if we have an allocation intensive application that allocates a lot of small objects like this, the waste is huge. This huge waste not only leads to poor space efficiency, but most importantly, it introduces virtual memory explosion and thus introduces much more TLB (Translation Lookaside Buffer) miss [8]–[10] which will greatly hinder performance (see details in Section 2).

Facing this new problem, this paper introduces a new heap or memory allocator design that focuses on reducing the fragmentation. We focus on intense memory allocations and giving them the exact size they want to reduce the fragmentation and thus reduce the related TLB miss to improve performance. To maintain a fast allocation speed, we introduce a mechanism to predict the allocation sizes of future allocations thus we can prepare bins of the exact size in advance for allocation. Experiments show that for big-memory-footprint benchmarks which allocate a lot of objects, our new memory allocator design can gain performance benefit of up to 1.7x (averagely over 18.8%) with less

```
1: int main(…){
2:    …
3:    some_struct * p = (some_struct *)malloc(sizeof(some_struct));
4:    …//operate on p
5:    free(p);
6:    …
6:    for(…){
7:       void* tmp = malloc(…//various sizes);
8:       …//operate on tmp
9:       free(tmp)
10:   }
11:}
```

Fig. 1. Use Case of Dynamic Memory Allocation

Manuscript received April 22nd, 2017; revised August 4th, 2017. This work was partially supported by China NSF under Grant No.61363062.

Xiuhong Li and Altenbek·Gulila are with Information Science and Engineering Colleges, Xinjiang University, Urumq, Xinjiang, P.R. China. (e-mail: lixiuhong2016@163.com, gla@xju.edu.cn).

memory usage (22%) compared with the state-of-the-art allocator Hoard [1], exhibiting great potential to be adopted widely. Our memory allocator is a general memory allocator that could be used widely in many fields such as scientific computing [11], image processing [12], and logic control programs.

In the rest of this paper, we first introduce the common design in current memory allocators and its fragmentation problem (Section 2). Then we give our optimized design to reduce fragmentation to achieve better performance (Section 3). We conduct experiment in Section 4 and conclude in Section 5.

II. Fragments in Memory Allocator

Memory allocator is generally a runtime system that offers interfaces to help upper applications perform dynamic memory allocation. It normally pre-allocates several large memory blocks (superblock called in Hoard [1]) from the underlying operating system via system calls such as mmap in Linux [13] or VirualAlloc in Windows [14] and then serves subsequent allocations from upper applications (malloc and free). As shown in Figure 2, most memory allocators adopt a bin-based mechanism which organizes superblocks into several classes of fixed sizes to support fast small objects allocation. For example, if in this case an upper application calls malloc(33), then the memory allocator will find and return a 64 byes free space in the 64-byte bin. Organizing and searching for space of fixed size could be done very efficiently based on this design. However, as the upper application will only use the 33 bytes of the allocated space, the rest 31 bytes is left unused and thus wasted. This consumes more virtual and physical memory. This problem, we call fragmentation, can be easily noticed in some allocation intensive applications.

Figure 3 shows the fragmentation ratio in Hoard [1], a state-of-the-art memory allocator for multi-threaded programs when running with some allocation intensive benchmarks [1], [2] (Here we define the fragmentation ratio to be the actual memory amount allocated to programs divided by the memory amount required by programs, as discussed above to indicate the waste situation). Taking the benchmark shbench for an example, it allocates more than 12,000,000 objects and most objects are of size between 64-180 bytes. These objects will all be allocated in the 128 and 256 bytes bins and will cause a huge waste of averagely 90 bytes per object. The result is that shbench uses about 3 times more memory than it requires. This 3 times more virtual memory usage will cause much more TLB (Translation Lookaside Buffer) miss due to the paging mechanism [8]–[10] and impact performance hugely (see details in Section 4).

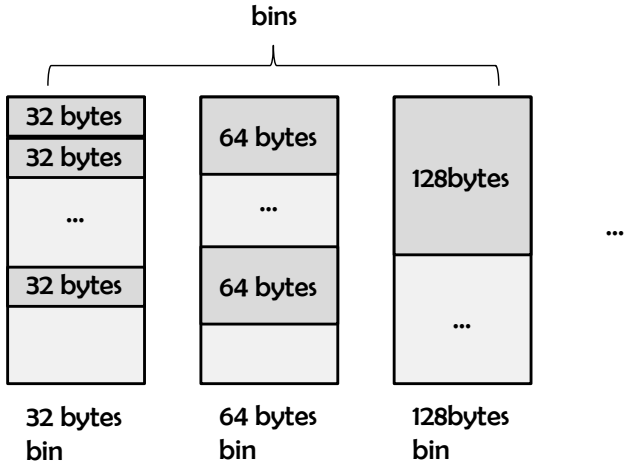

Fig. 2. Heap Management

III. Efficient Memory Allocator Design

It is a common view that customized memory allocator is the best performance choice over all general memory allocators [1]. In most performance-critical programs, the programmers always choose to write their own memory allocators instead of using the standard ones (e.g. glibc or Hoard). This is because the programmers know the memory usage patterns of upper allocations. For example, in the shbench benchmark, if the programmers know in advance that it will allocate a lot of objects of 70 bytes, they will then design the memory allocator to have a bin of 70 bytes instead of 128 bytes (bins shown in Figure 4). In this case the 70-byte bin design could effectively reduce memory usage and TLB miss and thus outperforms the 128-byte bin design. Above all, the key point to design an efficient general memory allocator is to try to analyze and predict the upper allocation patterns. We will introduce our method in this section.

First, we need to focus on intense memory allocations, especially those allocations in loops. There have been a lot of studies showing that most allocations in the same loop all tend to be the same sizes [15]. Our goal is to tackle these allocations well. For example, in the benchmark shbench, a lot of its memory allocation is in loops and most of them fall into the same size groups (70 bytes). According to this we need to create a special bin for the special size instead of the standard bins (shown in Figure 2). It also allocates some objects outside loops. However, as these objects only account for a small part and are not the main source of fragmentation, we just ignore them and leave them to be served in the standard bins (shown in Figure 2).

A. Intense Memory Allocation Detection

Intense memory allocation usually happens in loops. Loop detection is an old topic and there have been a lot of tools to achieve this [16]. However, the mainstream methods for loop detection all require the source code of programs for static code analysis [16], [17]. The adoption of these static mechanisms may compromise the practicability of our memory allocator. Instead, we introduce a simple threshold-based mechanism to detect intense memory allocation at runtime. The memory allocation process is shown in Figure 5. When serving each memory allocation, we first record and get the number of how many times this certain size has been allocated. Then we test if the number exceeds a certain threshold. If not, which means memory allocation of this size is not intense and thus it does not require special attention, then this memory allocation is served by the standard routine (standard_malloc in Figure 5, the standard memory allocator contains bins just like Figure 5). Otherwise, we need to construct new special bin for this special size of allocation to reduce the memory fragmentation (malloc_from_new_bin in

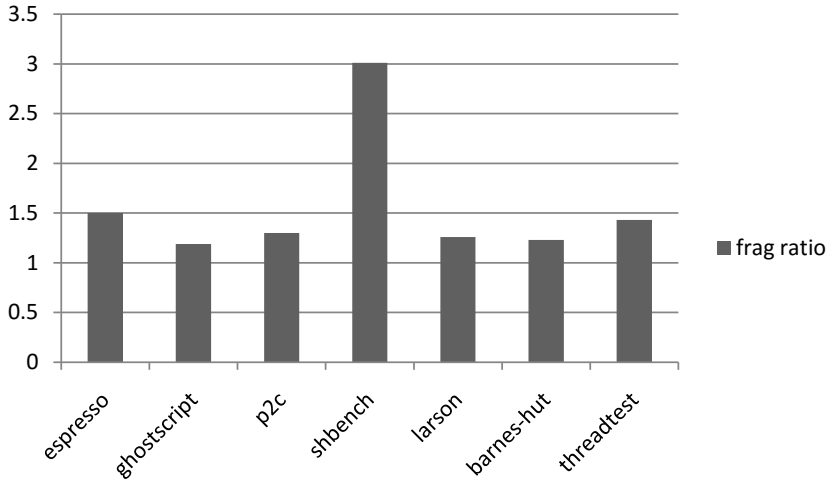

Fig. 3. Fragment Ratio

Figure 5). Currently the threshold is set to be 100 according to our experiment tests, which means if one objects of a certain size is allocated more than 100 times, it is most likely to be allocated in a loop and this represents an intense memory allocation situation. According to our test, averagely if the same size of object is allocated more than 100 times, we can start to gain performance benefit by creating a new special bin.

B. Context-based Allocation

Memory allocations from different loops may allocate objects of same size. In order to be more precise to predict the memory allocation pattern, we introduce a context-based method to distinguish different allocation contexts. That is, when serving each memory allocation (malloc in Figure 5), we record the allocation size information along with the current execution context. We can get the execution context by back-tracing the running stacks (e.g. backtrace in glibc). By doing this we are able to distinguish allocations from different execution functions and thus achieve better accuracy when predicting intensive allocations. To better map execution context to certain records, we built a hash table and use the addresses of the caller functions as the hash keys. More, we give each context a special id to identify that context. The code is shown in Figure 5.

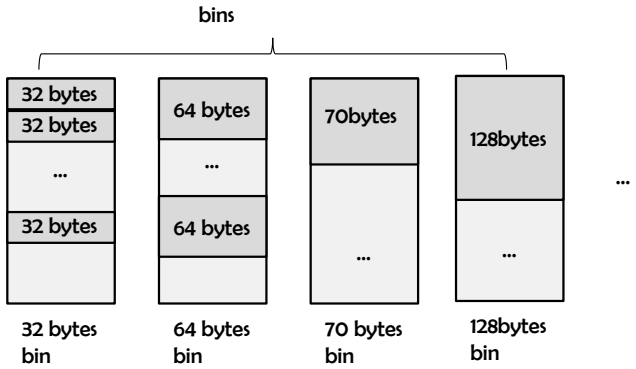

Fig. 4. Customized Heap Management

```
1: void * malloc(size_t sz){
2:   int n;
3:   int context_id = backtrace();  //get the current context
4:   n = record_the_sz(sz, context_id);  //record and get the number of how many
5:                                       //times this sized object is allocated under this context
6:   if(n >= threshold || is_intense(context_id))
7:       return standard_malloc(sz);
8:   else
9:       return malloc_from_new_bin(sz);
10:}
11:
12:void * malloc_from_new_bin(size_t sz){
13:    bin b = get_bin_with_size(sz);
14:    if(!b)
15:        create_new_bin(sz);
16:    return malloc_from_bin(b, sz);
17:}
```

Fig. 5. Code of Memory Allocation

C. Optimization

The context-based allocation scheme can be used further to help adjust the threshold (currently fixed to be 100) as introduced in Section III-A. If in a certain execution context we have found that a certain sized object is allocated for more than 100 times, we then mark this execution context as intense context, for it may contain loops to make intense memory allocations. Then next time if we re-enter this execution context and do memory allocations again, the threshold is set accordingly to be 1, which means we start to create special bins for special sizes immediately (the line 6 shown in the code in Figure 5). As we can know from the history that this execution context will do intense memory allocations. With this dynamic scheme we can start our customized memory allocation early and thus we are likely to get some performance benefit.

Overall, our memory allocator is implemented based on the general memory allocator Hoard which contains a run-time analysis part and a special part for intense memory allocation. If we find the memory allocation is not intense, we fall back to standard memory allocation with the standard bins shown in Figure 2. Otherwise we create new special bins to try to reduce the memory fragments and thus achieve better performance.

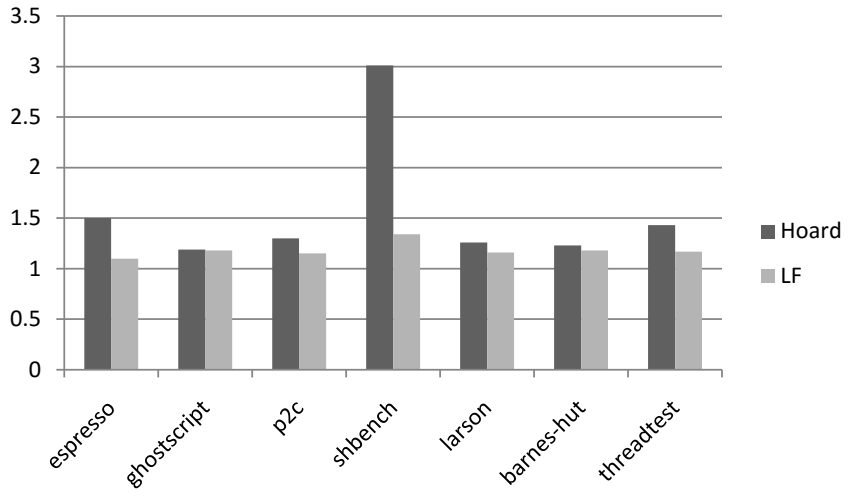

Fig. 6. Fragment Ratio

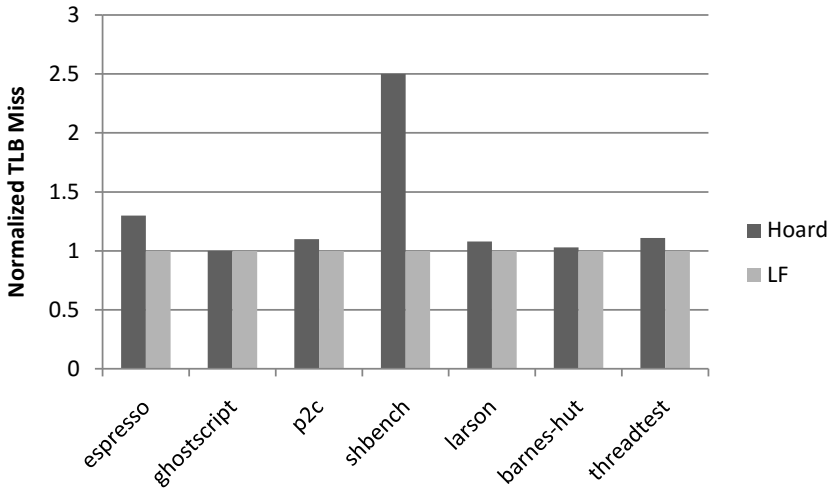

Fig. 7. Normalized TLB Miss

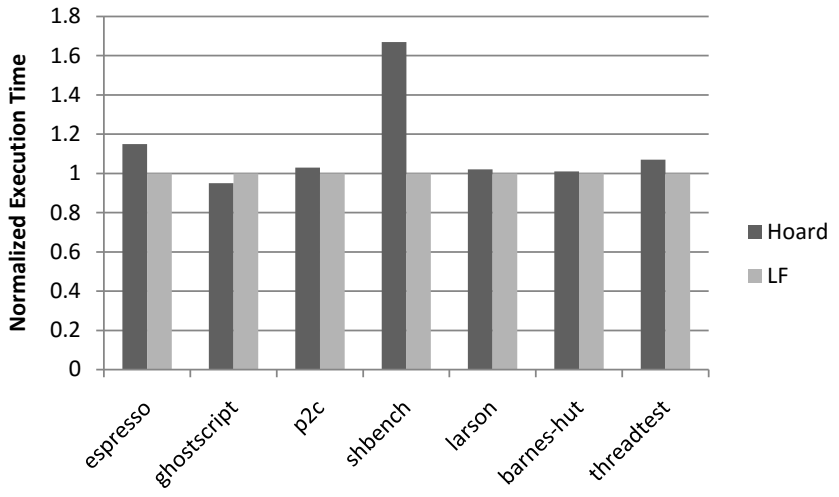

Fig. 8. Normalized Execution Time

IV. Experiment

A. Methodology

In this paper we propose an efficient memory allocator to try to reduce memory fragments and achieve better performance. In this section we will mainly show the performance and space benefit of our memory allocator compared with the state-of-the-art memory allocator Hoard [1].

We conduct our experiment on a server with Intel proces-

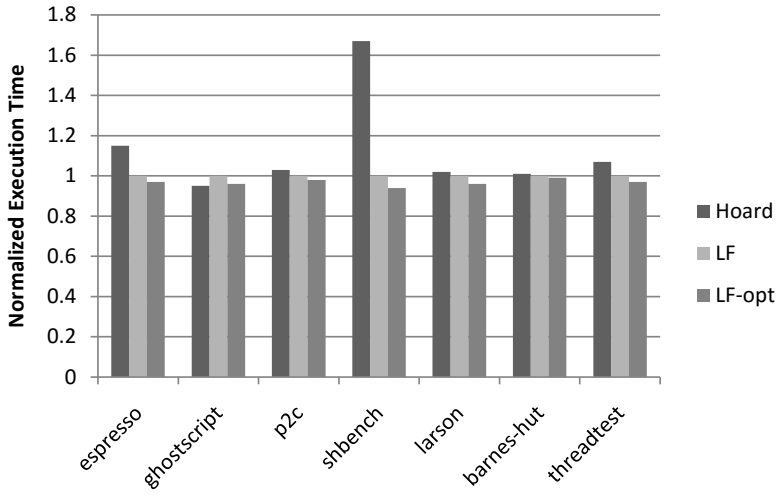

Fig. 9. Normalized Execution Time

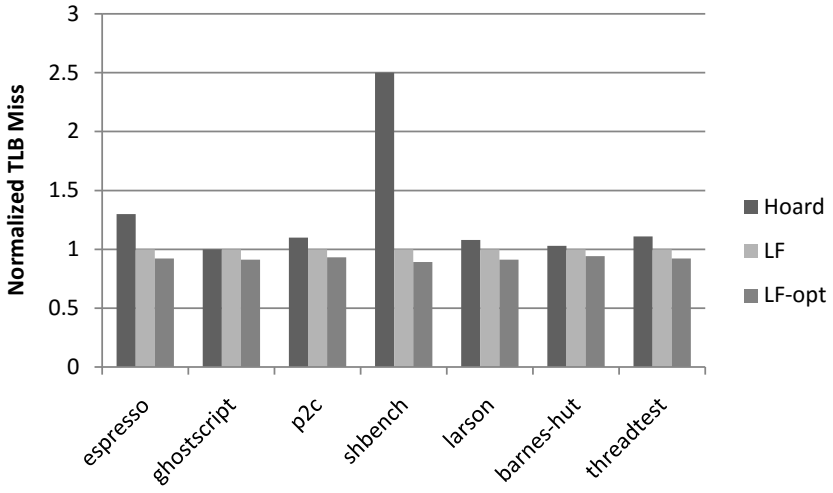

Fig. 10. Normalized TLB Miss

sor E3 e1230 v3 equipped with 32GB of physical memory. The benchmarks we selected are all allocation intensive benchmarks with default input [1]. We built our memory allocator based on Hoard (version number 3.101). The operating system kernel is Linux 3.10.

B. Results

First we test our work without the optimization of dynamic adjustment introduced in Section III-C. The experimental results are shown in Figure 6 - 8.

Figure 6 shows the fragmentation ratio of our work (LF for low fragmentation) compared with Hoard (Here we define the fragmentation ratio to be the actual memory amount allocated to programs divided by the memory amount required by programs). We can see that our LF can greatly reduce the fragmentation and thus reduce the memory usage, especially for the benchmark shbench. In the benchmark shbench, Hoard allocates 3 times more memory than it actually uses while our LF only allocates less than 30% more. Also for the benchmark espresso, p2c, larson, and threadtest, we can see obvious improvements that our LF achieves over Hoard. For the benchmark ghostscript and barnes-hut, LF and Hoard achieve similar result. This is because the origin fragmentation problem in these two benchmarks is not so severe.

About the previous experiment on fragmentation, we discuss that for a memory allocator to introduce less fragments, it is key that the memory allocation strategy matches the upper allocation needs of applications. For example, Hoard always uses standard allocation bins of 64, 128, 256, etc. Thus it will behave good with applications that only make allocations of these standard sizes (like the benchmark ghostscript and barnes-hut do, thus Hoard performs well on these two benchmarks as shown in Figure 6.). Our work LF uses standard allocation bins at first and adjusts itself to the allocations of special sizes made by upper applications, thus it behaves well in all cases (as shown in Figure 6). We show that this is the key that our work LF can outperform traditional memory allocators.

Figure 7 shows the normalized TLB miss Hoard introduces over our LF. We can see the TLB miss is highly related to the memory fragmentation ratio. For the benchmark shbench, Hoard introduces about 2.5 times more TLB miss than our LF. Also for the benchmark espresso, p2c, larson, and

threadtest, our LF shows obvious advantage of less TLB miss than Hoard. For the benchmark ghostscript and barnes-hut, Hoard achieves similar result with our LF (1 and 1.03, respectively.). In theory, if an application is using more memory, it will put more pressure on TLB for address translation (which is because the limited TLB slot cannot cache all the memory footprint of the application). When a TLB miss happens, it impacts the performance accordingly, for more time will be spent on searching the page table to perform virtual-physical address translation. Thus, much memory fragment in an application which will lead to more consumption of memory is bad in terms of both time and space efficiency. Thus it is crucial for a memory allocator to try to reduce the memory fragment and thus reduce the memory consumption and improve performance.

Figure 8 shows the normalized execution time. We can see or LF achieves an obvious improvement on performance for most benchmarks, especially in the shbench, Hoard is 1.7 times slower than our LF. This is mainly due to the more fragments and more TLB miss it introduces. For the benchmark ghostscript, our LF is slower than Hoard, this is due to the management overhead and the runtime analysis overhead in LF. We point out that the main performance overhead comes from more time spent on dealing with TLB miss, which is caused by more memory consumption introduced by memory allocators.

From the Figure 7 and 8 we can tell that the overall performance of applications is highly coupled with the overall tlb miss the applications may incur. The overall performance is determined by the tlb miss. Thus to improve performance, it is key to reduce tlb miss. A step forward, we can tell from the Figure 6 and 7 that the tlb miss is determined by the overall fragmentation situation introduced during memory allocation. Thus to draw a conclusion here, controlling the memory fragmentation is the key for a memory allocator to perform well in terms of space and time efficiency. This is the key finding and guiding idea of this paper.

Second, we show the improvement that our optimization (introduced in Section III-C) of dynamic adjustment could achieve.

Figure 9 shows the result of execution time. We can see that our optimization (LF-opt) could effectively improve the overall performance. Averagely, LF-opt achieves 3.3% performance improvement over LF, and 18.8% over Hoard. We argue that this improvement mainly comes from the early construction of customized bins for memory allocations in intensive execution contexts. By marking certain execution contexts as intense, we are able to accelerate the further memory allocations in such contexts and thus achieve improved performance. The corresponding results about normalized tlb miss is shown in Figure 10, our optimized scheme achieves better result on TLB miss (averagely 9% less than LF and 39% less than Hoard).

In all, a good memory allocator will achieve less memory fragmentation, which brings benefit for both time and space efficiency. Our memory allocator achieves better performance (up to 1.7x, averagely over 18.8%) and at the same time uses less memory (22%), which shows great potential to be a directly drop-in replacement of current state-of-the-art memory allocators, especially for allocation intensive applications.

V. Conclude

This paper introduces an efficient memory allocator that achieves better performance and at the same time consumes less memory. We focus on intensive memory allocation and introduce a dynamic strategy to create customized bins for memory allocation of special sizes. Unlike the standard bins in previous general memory allocators, the customized bins created in our memory allocator can better fit intensive memory allocation and reduce memory fragmentation. By reducing the memory usage, we can reduce the TLB miss and thus achieve better performance. Experimental results show that our new memory allocator can achieve up to 1.7x speedup (averagely over 18.8%) over the state-of-the-art memory allocator Hoard with 22% less memory usage, which shows great potential to be a directly drop-in replacement of current state-of-the-art memory allocators, especially for allocation intensive applications. Our future work includes adopting and testing our memory allocator to modern large data base systems.

Acknowledgment

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

Many thanks to Prof. Li who has helped with the experiment and his comments on the memory fragmentation problem.

References

[1] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *ACM Sigplan Notices*, vol. 35, no. 11, pp. 117–128, 2000.

[2] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Building high-performance custom and general-purpose memory allocators," in *Proceedings of the SIGPLAN 2001 Conference on Programming Language Design and Implementation*, 2000, pp. 114–124.

[3] P. Argyroudis and C. Karamitas, "Exploiting the jemalloc memory allocator: Owning firefoxs heap," *Blackhat USA*, 2012.

[4] S. Ghemawat and P. Menage, "Tcmalloc: Thread-caching malloc," 2009.

[5] X. Chen, A. Slowinska, and H. Bos, "On the detection of custom memory allocators in c binaries," *Empirical Software Engineering*, vol. 21, no. 3, pp. 753–777, 2016.

[6] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn, "Archipelago: trading address space for reliability and security," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2. ACM, 2008, pp. 115–124.

[7] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: solved?" in *ACM SIGPLAN Notices*, vol. 34, no. 3. ACM, 1998, pp. 26–36.

[8] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *Acm sigplan notices*, vol. 41, no. 6. ACM, 2006, pp. 158–168.

[9] G. Hildesheim, C. K. Tan, R. S. Chappell, and R. Bhatia, "Concurrent page table walker control for tlb miss handling," Jun. 30 2015, uS Patent 9,069,690.

[10] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal, "Range translations for fast virtual memory," *IEEE Micro*, vol. 36, no. 3, pp. 118–126, 2016.

[11] Z. Sun, A. Li, and H. Qin, "The neutrix limit of the hurwitz zeta function and its application." *IAENG International Journal of Applied Mathematics*, vol. 47, no. 1, pp. 56–65, 2017.

[12] V. A. Kulyukin and S. K. Reka, "Toward sustainable electronic beehive monitoring: Algorithms for omnidirectional bee counting from images and harmonic analysis of buzzing signals." *Engineering Letters*, vol. 24, no. 3, pp. 317–327, 2016.

[13] P. Daniel, C. Marco *et al.*, "Understanding the linux kernel," 2007.

[14] L. Li, J. E. Just, and R. Sekar, "Address-space randomization for windows systems," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006, pp. 329–338.

[15] R. Guo, X. Liao, H. Jin, J. Yue, and G. Tan, "Nightwatch: Integrating lightweight and transparent cache pollution control into dynamic memory allocation systems." in *USENIX Annual Technical Conference*, 2015, pp. 307–318.
[16] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.
[17] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder, "Static analysis of energy consumption for llvm ir programs," in *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. ACM, 2015, pp. 12–21.

Xiuhong Li is a full-time researcher and lecturer at Information Science and Engineering Colleges, Xinjiang University, Urumq, Xinjiang, P.R. China. Her research interests include operating system, management runtime system, and compiler.

Altenbek·Gulila is a professor at Information Science and Engineering Colleges, Xinjiang University, Urumq, Xinjiang, P.R. China. Her research field includes operating system, natural language processing, and artificial intelligence. She has published more than 35 papers in these fields.