Critique for Hoard: A Scalable Memory Allocator for Multithreaded Applications

Xin Wang

1 Summary

Many memory allocators have been bottlenecks for parallel, multithreaded C and C++ applications in limiting their program scalability and performance. Problems with memory allocators include low speed and scalability, false sharing caused by heap organizations, and increase in memory consumption when confronting producer-consumer object allocation. The paper Hoard: A Scalable Memory Allocator for Multithreaded Applications has proposed a better allocator with rapid speed and good scalability, that can avoid false sharing and is memory efficient. Hoard is noticable superior to other techniques in solving blowup and false sharing problems, and achieves barely any synchronization costs. [3]

Hoard is designed to overcome the problems with previous allocators such as false sharing that occurs in heap objects when cache lines are divided into small objects. It also keeps blowup to a bounded constant factor, while other allocators are unable to bound blowups. First, in order to bound blowup, Hoard maintains a global heap as well as heaps for each processor. It transfers a large chunk of memory from per-processor heap to global heap, which can be reused by other processors. Each thread can not access global heap and per-processor heaps at the same time. Hoard maintains usage statistics for each heap and allocates memory in superblocks. All blocks in a superblock are in the same size class. Whenever a per-processor heap reaches the emptiness threshold, Hoard moves superblocks from that heap to the global heap. To improve locality, when a block in a superblock is freed, the superblock is moved to the front of its fullness group. Second, Hoard combines superblocks together with multiple-heaps to avoid false sharing. Hoard ensures that only one thread can allocate from a given superblock. If more than one thread make requests from memory, these requests should be satisfied from different superblocks. Once a block is deallocated, Hoard returns the block to its superblock.

A set of experiments have be designed to prove the effectiveness of Hoard. These experiments demonstrate the fast speed, good scalability, false sharing avoidance ability and low fragmentation of Hoard and make comparison between Hoard and other memory allocators such as Solaris, Ptmalloc, MTmalloc etc. The experiments are carried both on uniprocessors and multiprocessors. The results are stable and robust.

To sum up, this paper has introduced a new memory allocator Hoard, which is able to improve performance such as speed, scalability, false sharing avoidance and low fragmentation, compared to previous allocators. Its per-processor and global heaps as well as moving superblocks across heaps have enabled these features and help to bound blowup.

2 Positive reactions

Hoard is designed to overcome the disadvantages of previous memory allocators. It is the first allocator that has significantly avoided the false sharing and memory inefficient problems. Hoard uses novel techniques of organizing per-processor heaps and global heap and moving superblocks among heaps. These techniques have helped Hoard to perform better than others not only in the aspects mentioned above, but also in bounding blowup and reach low expected case synchronization.

The experiments in the paper are convincing and the results are successful. The improvement in experiments in speed, scalability, false sharing avoidance and low fragmentation has proved Hoard a better memory allocator than other allocators, which are all useful and prevalent tools, involved in the experiments. Robustness has also be proved from the results. The sensitivity study helps to examine the effect of altering empty fraction on runtime and fragmentation for multithreaded benchmarks.

According to the current requiring for scalable architecture and runtime system support, Hoard has provided researchers with a good direction to this field.

3 Negative reactions

Hoard is designed to deal with problems of memory allocators for parallel and multithreaded C and C++ programs. In C and C++ programs, when allocating, deallocating or reallocating memory, pointers need to be stored appropriately. However, Hoard does not take copying collectors and pointers storage into consideration.

As indicated in [1], Hoard may rely heavily on CAS(compare and swap) hardware instruction. CAS instructions can be several orders of magnitude slower than non-atomic operations. Therefore, it is meaningful to try to avoid atomic operations in common malloc and free routines and to use them in non-blocking way.

4 Extension proposal

If we take transaction memory into consideration, Hoard can be extended to support transaction that enable transactional use of the system memory allocator.

5 Conclusions

To sum up, Hoard has contributed much in improving the performance of memory allocation and has successfully dealt with many problems with other previous allocators, which is proved by experiments in the paper. This paper has big influence to future research. For example, according to similar ideas, Olszewski et al. have implemented JudoSTM [2], a novel dynamic binary-rewriting technique to implemnt STM supporting C/C++ programs.

References

[1] Richard L Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C Hertzberg. Mcrt-malloc: a scalable transactional memory allocator. In *Proceedings of the 5th international symposium on Memory management*, pages 74–83. ACM, 2006.

[2] Marek Olszewski, Jeremy Cutler, and J Gregory Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 365–375. IEEE Computer Society, 2007.

[3] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197. ACM, 2006.