HEAPPERF: Diagnosing Heap Related Performance Bugs

Tongping Liu

Abstract

Keywords Performance, Synchronization, Multithreaded Programs, Profiling, Instrumentation

1. Introduction

Heap memory related performance bugs can be from the following categories, if only think about applications.

- : Too many allocations and deallocations. Based on [6], the total run time spending in memory allocation and liberation may take up to 30% execution time.
- : Too many memory usages: this can actually affect the performance when there are too much memory that has been allocated but not used.
- :

What we can do for heap memory management?

First, we can point out the unnecessary memory allocations and deallocations. For example, we can malloc a large object, and then assign to different small projects. Some of them may be called inside the internal level of loop functions, we can move up. By reducing unnecessary memory allocations, we expect to improve the performance.

Second, we can give a statistics on the life-span of objects. Whether we can find out some problems inside? For example, we can use stack variables instead of heap.

Third, we can actually give the statistics on each callsite. Some callsites may have larger number of allocations. Check whether we can use .

Can we evaluate the performance related to heap allocations? For example, how much time is spending on memory allocation. We can approximate the time of spending on each allocation. Then we can distribute the time to different statements, just similar to gprof. Then maybe it is obvious that we can reduce the overhead by reducing the memory allocations.

In the end, although not every interested, it is to check the overhead of every memory allocation on each popular memory allocation. Thus, pointing out that the memory allocation actually should pay attention to the level of stacks. Thus, it is possible that we can design a new memory allocator by reducing the level of memory allocation. This is a reverse to HeapLayer. It is great to have an survey paper on this:

A. How is the overhead of memory allocation in large applications? How we can evaluate it? B. How is the overhead that comes from memory management? We evaluate this on some popular benchmarks. C. Whether the overhead comes from different cache uses? or other things. D. It will shed a light whether we need to re-design the memory allocator. It will be a perfect paper for ISMM or other places.

2. Related Work

[2] develops two simply analytical model to evaluate the performance impact on large application, based on an application's interaction with the memory system. The observation is that a regular application has continuous and stride memory accesses, while an irregular application has three memory access types: continuous accesses, accesses within the same L1/L2 cache line, and random accesses. This is actually not that related to our system.

[1]: This paper presents a detailed performance study of three important classes of commercial workloads: online transaction processing (OLTP), decision support systems (DSS), and Web index search. This study characterizes the memory system behavior of these workloads through a large number of architectural experiments on Alpha multiprocessors augmented with full system simulations to determine the impact of architectural trends. We also identify a set of simplifications that make these workloads more amenable to monitoring and simulation without affecting representative memory system behavior. We observe that systems optimized for OLTP versus DSS and index search workloads may lead to diverging designs, specifically in the size and speed requirements for off-chip caches.

Mtrace++ [6] is a source code level instrumentation that traces the memory allocations and deallocations. Mtrace++ identifies originations of allocated memories and life spans of objects. But it can not point out whether problems may occur inside programs.

[5]

[4]

LeakPoint[3] shares the similar target as HEAPPERF on memory leak detection that memory leaks of larger size are more important than leaks of smaller area. However, LeakPoint imposes between 100× and 300× overhead, while only less than 5% overhead for HEAPPERF. It do not focus on another source of performance problems, those unnecessary memory allocations and deallocations.

3. Conclusion

References

[1] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 3–14, Washington, DC, USA, 1998. IEEE Computer Society.

[2] I. Chihaia and T. Gross. Effectiveness of simple memory models for performance prediction. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on - ISPASS*, pages 98–105, 2004.

[3] J. Clause and A. Orso. Leakpoint: Pinpointing the causes of memory leaks. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 515–524, New York, NY, USA, 2010. ACM.

[4] Y. Hasan and J. Chang. A hybrid allocator. In *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*, pages 214–222, March 2003.

[5] W. H. Lee, J. Chang, and Y. Hasan. Evaluation of a high-performance object reuse dynamic memory allocation policy for c++ programs. In *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, volume 1, pages 386–391 vol.1, May 2000.

[6] W. H. Lee, J. M. Chang, and Y. Hasan. A dynamic memory measuring tool for c++ programs. In *Proceedings of the 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'00)*, ASSET '00, pages 155–, Washington, DC, USA, 2000. IEEE Computer Society.