PRmalloc: Leveraging Predictability for Deep Learning Memory Allocation

Wencong Xiao, Shiru Ren, Tongxuan Liu, Yong Li
{wencong.xwc,shiru.rsr,tongxuan.ltx,jiufeng.ly}@alibaba-inc.com
Alibaba Group

Abstract

We introduce PRmalloc, a new memory allocator that optimizes the memory management of deep learning. Benefiting from the uniquely predictable feature of repeated iterations in deep learning training, PRmalloc leverages the domain-specific knowledge to better cache and reuse large memory blocks, cutting down the system overhead of memory allocation and saving memory footprint significantly. We have implemented PRmalloc in TensorFlow, a popular deep learning framework. Our evaluation shows that PRmalloc improves the end-to-end training performance of the state-of-art recommendation model up to 1.8×.

ACM Reference format:
Wencong Xiao, Shiru Ren, Tongxuan Liu, Yong Li. 2019. PRmalloc: Leveraging Predictability for Deep Learning Memory Allocation. In *Proceedings of Workshop on AI Systems at SOSP 2019, Ontario, Canada, Oct 27, 2019 (AI Systems '19),* 3 pages.

1 Introduction

An increasingly popular trend over the last few years is deep learning for artificial intelligence. Recommendation, as a crucial artificial intelligence application to uncover user intents, is widely deployed in many companies, such as Amazon and Alibaba. The industrial-scale deep learning recommenders usually build on computational frameworks (*e.g.,* TensorFlow [3]) to train models with massive sparse user behavior data on CPU, consuming a large amount of host memory.

Even though with the state-of-art memory management module embedded, we observe that recommendation applications on TensorFlow usually occupy more than 50% memory compared with the real requirement. Surprisingly, detailed profiling reports up to 900K/s minor page fault during job execution, which hurts the overall system performance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
AI Systems '19, Oct 27, 2019, Ontario, Canada
© 2019 Association for Computing Machinery.
ACM ISBN .

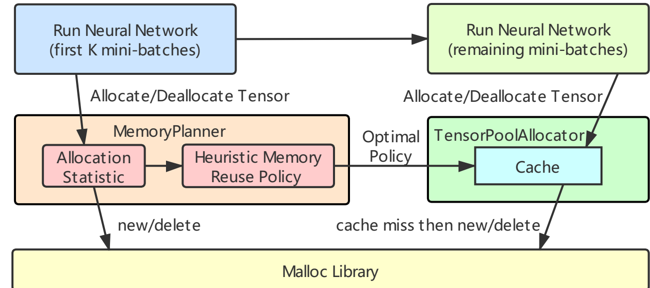

Figure 1. Architecture overview.

More specially, TensorFlow relies on third-party memory libraries *ptmalloc* or *jemalloc* to manage the CPU memory. Those memory libraries prominently improve the performance for web server [2], benefiting from a per-thread memory pool design for small memory blocks (*e.g.,* < 32KB).

However, we argue that existing memory allocators are a poor fit to deep learning. The tensors used in training usually require large memory blocks (~MB), and the summary of total required memory can be tens of GB [6]. Our statistic on a typical recommendation model (*i.e.,* DeepCTR [1]) shows that a sub-second mini-batch training triggers more than 1500 large memory allocation (>32KB), counting for more than 98% of the total memory request.

To address the challenges mentioned above, we exploit the predictability of deep learning training. A deep learning training job includes millions of *mini-batches*, each mini-batch is a traversal on a deterministic data flow graph for computation. Therefore, in the view of an application, most of the allocate/deallocate requests are consistent among mini-batches. Furthermore, the dependency relationship of computation can also be utilized to schedule memory block recycle [5].

We introduce PRmalloc, a brand new *predictable reusable memory allocator* taking advantage of such remarkable predictability. As indicated in Figure 1, in contrast to conventional memory allocators, PRmalloc adapts to the memory usage characteristics of different deep learning training applications, by learning from the memory block life-cycle at the early stages through *MemoryPlanner* module. The collected domain knowledge helps to generate heuristic policy into *TensorPoolAllocator* to better schedule the memory recycle. Such a design has two vital benefits. First, memory allocator can cache the large memory block, saving the minor page fault introduced by memory allocation system call. Second, a better memory reuse plan can be learned to reduce the memory footprint, minimizing the overall resource usage.

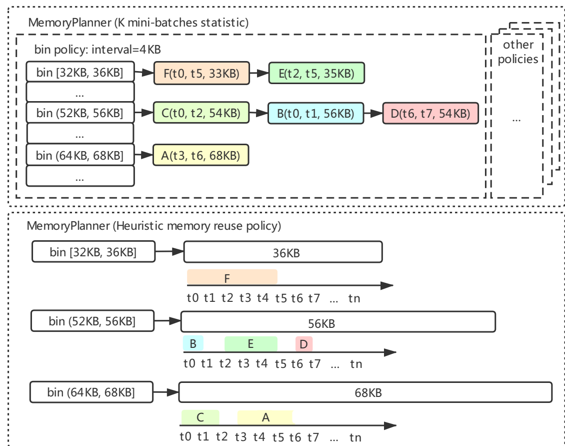

Figure 2. A memory reuse example.

Preliminary experiments show PRmalloc boosts the state-of-art recommendation models training by 1.8×, benefiting from lower memory footprint and fewer page fault.

2 PRmalloc

Motivated by the unique memory characteristics, we design PRmalloc, a statistically-based heuristic memory management library especially for deep learning applications. First, we outline the architecture of PRmalloc. Second, we discuss how it can effectively boost the end-to-end system performance by identifying and utilizing the memory reuse information of deep learning between and within mini-batches.

2.1 System architecture and workflow

We implement PRmalloc in TensorFlow. As illustrated in Figure 1, the architecture of PRmalloc consists of three principal components, namely, the MemoryPlanner, the TensorPoolAllocator, and the malloc library.

MemoryPlanner is in charge of the memory management policy. It first collects the tensor allocation information during execution. Based on this information, MemoryPlanner can heuristically search for the optimal memory allocation scheme for the running deep learning application. By leveraging the malloc library, TensorPoolAllocator uses a lock-free queue to manage the memory blocks' allocation and deallocation according to the optimal memory allocation scheme.

Thanks to the predictability, we observe that the read/write sequences of the tensors remain quite stable between successive mini-batches, which means they incline to allocate/deallocate the same set of tensors at roughly the same time, offering a possibility of reducing memory usage through reusing the allocated memory blocks between mini-batches. As illustrated in Figure 1, to find the optimal memory allocation scheme, the MemoryPlanner first leverages the allocation statistic to record all tensors' allocate and deallocate operations in the first K (typically 10) mini-batches, capturing the lifetime of the required memory blocks to formulate the memory block recycle dependency. It then estimates the amount of memory usage under different bin policies

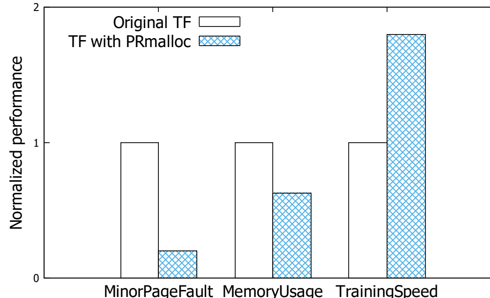

Figure 3. Deep-CTR Performance.

(PRmalloc adopts a bin-based mechanism [4] which organizes memory blocks into several fixed-size bins). Based on the estimations, it heuristically searches for the optimal bin policy to minimize the memory usage and minor page fault.

TensorPoolAllocator considers the identified heuristic policy to manage a memory pool for the rest of mini-batches. Ideally, the memory blocks it provides each time are exactly match the ones requested by the deep learning application, especially for large memory requests. Therefore, it avoids extra minor page faults from the OS memory allocation. Note that, sometimes the actual memory requests can validate the found policy, TensorPoolAllocator will roll back to the original memory allocator instead.

2.2 Heuristic memory reuse within mini-batches

Apart from the across mini-batches reuse, to further cut down the memory footprint, PRmalloc also reuses memory blocks by utilizing the memory dependency information within a mini-batch. Specifically, during allocation statistics recording, PRmalloc first sorts all memory allocation requests within a mini-batch from largest to smallest according to the size of the request (*e.g.*, $A \longrightarrow F$ in Figure 2). Then, PRmalloc tries to match each request with the appropriate-sized block (which is available for the requested time period) by ascending order of the bin size. For instance, for the request C in Figure 2, PRmalloc tries to match it with 52KB blocks first. However, due to there is no available block for requested time period, PRmalloc matches it with a 68KB block finally. As the example shown in Figure 2, PRmalloc reduces the memory usage by up to 47% by reusing the allocated memory blocks within a mini-batch for request C, D, and E.

3 Evaluation

We use a real production workload from Taobao Search of Alibaba to evaluate PRmalloc. We implement our system prototype on TensorFlow r1.8 and evaluate the DeepCTR model [1] in a configuration of 200 workers and 40 parameter servers running in a hybrid could environment. Figure 3 shows the normalized statistic/performance of minor page fault, total memory usage, and training speed (*i.e.,* step per second). According to the profiling result, the memory usage and page fault drop significantly, thus the performance boosts by 1.8× in the end-to-end training.

References

[1] 2019. DeepCTR. (2019). Retrieved Sept 2, 2019 from https://github.com/shenweichen/DeepCTR

[2] 2019. Scalable memory allocation using jemalloc. (2019). Retrieved Sept 11, 2019 from https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919/

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[4] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of the BSDCan conference*.

[5] Hal A Kierstead. 1991. A polynomial time approximation algorithm for dynamic storage allocation. *Discrete Mathematics* 88, 2-3 (1991), 231–237.

[6] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 595–610.