Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems

Ismail Oukid*
TU Dresden & SAP SE

Daniel Booss*
SAP SE

Adrien Lespinasse°
Independent

Wolfgang Lehner+
TU Dresden

Thomas Willhalm*
Intel Deutschland GmbH

Grégoire Gomes°
Grenoble INP

*first.last@{sap,intel}.com +first.last@tu-dresden.de °first.last@gmail.com

ABSTRACT

Storage-Class Memory (SCM) is a novel class of memory technologies that promise to revolutionize database architectures. SCM is byte-addressable and exhibits latencies similar to those of DRAM, while being non-volatile. Hence, SCM could replace both main memory and storage, enabling a novel single-level database architecture without the traditional I/O bottleneck. Fail-safe persistent SCM allocation can be considered *conditio sine qua non* for enabling this novel architecture paradigm for database management systems. In this presentation, we present PAllocator[1], a fail-safe persistent SCM allocator whose design emphasizes high concurrency and capacity scalability. Contrary to previous works, PAllocator thoroughly addresses the important challenge of persistent memory fragmentation by implementing an efficient defragmentation algorithm. We show that PAllocator outperforms state-of-the-art persistent allocators by up to one order of magnitude, both in operation throughput and recovery time, and enables up to 2.39x higher operation throughput on a persistent B-Tree.

1. INTRODUCTION

Memory allocation has always been a topic in database systems. In the disk-based database systems era, memory management was mostly synonymous of buffer pool management. In the era of main-memory database systems, we switched to using transient allocators, such as jemalloc [1] and tcmalloc [9]. However, these allocators are general-purpose and database systems have all but general-purpose needs. Hence, to no surprise, several database system vendors, such as SAP, IBM, and Oracle implement their own memory management [4, 5, 3].

SCM can be architected as universal memory, i.e., as main memory and storage at the same time. This architecture enables a novel single-level database architecture that is able to scale to much larger main memory capacities while removing the traditional I/O bottleneck. Databases that implement this paradigm start to emerge, such as Peloton [6], FOEDUS [10], and our database SOFORT [13]. Persistent memory allocation is a fundamental building block for enabling this novel database architecture. In this presentation we present SOFORT's memory management component, called PAllocator, a highly scalable, fail-safe, and persistent allocator for SCM, specifically designed for databases that require very large main memory capacities. PAllocator uses internally two different allocators: SmallPAllocator, a small block persistent allocator that implements a segregated-fit strategy; and BigPAllocator, a big block persistent allocator that implements a best-fit strategy and uses hybrid SCM-DRAM trees to persist and index its metadata. The use of hybrid SCM-DRAM trees [12] enables PAllocator to also offer a fast recovery mechanism. Besides, PAllocator addresses fragmentation in persistent memory, which we argue is an important challenge, and implements an efficient defragmentation algorithm that is able to reclaim the memory of fragmented blocks by leveraging the hole punching feature of sparse files. To the best of our knowledge, PAllocator is the first SCM allocator that proposes a transparent defragmentation algorithm as a core component for SCM-based database systems. Our evaluation shows that PAllocator improves on state-of-the-art persistent allocators by up to two orders of magnitude in operation throughput, and by up to three orders of magnitude in recovery time. Besides, we integrate PAllocator and a state-of-the-art persistent allocator in the FPTree [12], a persistent hybrid SCM-DRAM B-Tree, and show that PAllocator enables up to 2.39x better operation throughput than its counterpart.

Presentation Outline. First, we present an analysis of the challenges posed by SCM that we drew from our experience in designing and implementing SOFORT. Then, we discuss different design dimensions of state-of-the-art persistent allocators, with a focus on the SCM-relevant ones such as the pool structure, garbage collection, and defragmentation. Thereafter, we present our PAllocator, its different components, its defragmentation algorithm, and its recovery mechanism. Then, we present a performance evaluation overview of PAllocator against state-of-the-art SCM allocators. Finally, we conclude with a list of future SCM management challenges to foster discussion.

2. SCM PROGRAMMING CHALLENGES

SCM is no panacea as it raises unprecedented programming challenges. Given its byte-addressability and low latency, processors can access, read, modify, and persist data in SCM using load/store instructions at a CPU cache line granularity. The path from CPU registers to SCM is long and mostly volatile, including store buffers and CPU caches, leaving the programmer with little control over when data is persisted, thereby causing a plethora of programming challenges: data consistency, data recovery, partial writes, and persistent

[1]This work has been published in PVLDB 2017 [11].

memory leaks – all of which we will detail in this presentation. Therefore, there is a need to enforce the order and durability of SCM writes using persistence primitives, such as cache line flushing instructions. This in turn creates new failure scenarios, such as missing persistence primitives. As an example, we discuss below the challenge of data recovery.

SCM is handled using a file system. User space access to SCM is granted via memory mapping using *mmap*. The mapping behaves like *mmap* for traditional files, except that the persistent data is directly mapped to the virtual address space, instead of to a `DRAM`-cached copy. When a program crashes, its pointers become invalid since the program gets a new address space when it restarts. This implies that these pointers cannot be used to recover persistent data structures. To solve this issue, we devise a programming model that relies on a new pointer type, denoted Persistent Pointer (PPtr), that remains valid across restarts. It consists of a base, which is a file ID, and an offset within that file that indicates the start of the block pointed to. Persistent pointers can easily be translated into regular pointers by adding the offset to the start address of the memory-mapped file. To perform recovery, we need to keep track of an entry point. One entry point is sufficient for the whole storage engine since every structure is encapsulated into a larger structure up to the full engine. An entry point can be two PPtrs: the first points to the root object of `SOFORT` and the second to the root object of `PAllocator`. The entry point is kept in `SCM` in a special file at a known offset.

3. DESIGN GOALS AND DECISIONS

We identify the following design goals for persistent memory allocators tailored for large-scale SCM-based systems:

- The ability to adapt to changes in memory resources; This is particularly important in a cloud environment.
- High concurrency scalability; Large main-memory systems usually run on multi-socket systems with up to 1000 cores. Thus, it is important for the persistent allocator to provide robust and scalable performance.
- Provide robust performance for all sizes of allocations, as database-system allocation sizes cover a wide range, from a few bytes to hundreds of gigabytes.
- Fast recovery; Currently there are instances of single-node main-memory database systems such as SAP HANA [8] with up to 64 `TB` of main memory. With SCM this capacity will quickly exceed 100 `TB`. Thus, the persistent allocator must exhibit fast recovery and should not rely on scanning memory to recover its metadata.
- Defragmentation ability; Database systems run for a very long time, much longer than general-purpose applications, making fragmentation much more likely to happen.

So far, state-of-the-art persistent allocators, such as `NVML` [2] and `Makalu` [7] have been engineered as general-purpose allocators, taking inspiration from existing general-purpose transient allocators. We argue that they are unfit for large-scale SCM-based database systems because:

- They use a single pool (file), which is difficult to both grow and shrink.
- They put an emphasis on the scalability of small-block allocations (from a few bytes up to a few kilobytes), and neglect that of middle-sized and large-block allocations.
- They do not provide defragmentation capabilities.
- They often rely on scanning memory to recover their meta-data during recovery.

`PAllocator` is not a general-purpose allocator. It fulfills the above design goals following radically different design decisions than state-of-the-art persistent allocators:

- We use multiple files instead of a single pool, which allows us to easily grow and shrink our pool of persistent memory.
- We use large files to avoid having a large number of them which would hit the limitations of current file systems.
- We use three different allocation strategies for small, big, and huge allocation sizes, mostly independent from each other, to ensure robust performance for all allocation sizes.
- We aggressively cache free memory by not removing free files. Instead, we keep them to speed up future allocations. This is acceptable since main-memory database systems usually have dedicated resources.
- Instead of thread-local pools, we use one allocator object per physical core. Using thread-local pools might hurt performance and complicates fail-safety management as the local pool has to be given back and integrated in the global pool upon thread termination. Using striping per physical core combined with aggressive caching provides a stable, robust, and scalable performance.
- To defragment memory, we leverage the hole punching feature of sparse files. This is an additional advantage of using multiple files.
- To provide fast recovery, we persist most of `PAllocator`'s metadata and rely on hybrid `SCM-DRAM` trees to trade off between performance and recovery time when necessary.

4. REFERENCES

[1] jemalloc Memory Allocator. http://jemalloc.net/ (last accessed: september 24, 2017).

[2] NVML Library. http://pmem.io/nvml/ (last accessed: september 24, 2017).

[3] Oracle Database Administrator's Guide: Managing Memory. https://docs.oracle.com/database/121/ADMIN/memory.htm (last accessed: september 24, 2017).

[4] SAP HANA Memory Usage Explained. https://www.sap.com/documents/2016/08/205c8299-867c-0010-82c7-eda71af511fa.html (last accessed: september 24, 2017).

[5] The DB2 UDB memory model. https://www.ibm.com/developerworks/data/library/techarticle/dm-0406qi/ (last accessed: september 24, 2017).

[6] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *PVLDB*, 10(4):337–348, 2016.

[7] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *OOPSLA 2016*, pages 677–694. ACM, 2016.

[8] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[9] S. Ghemawat and P. Menage. TCMalloc: Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html (last accessed: september 24, 2017).

[10] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 691–706. ACM, 2015.

[11] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes. Memory management techniques for large-scale persistent-main-memory systems. *Proceedings of the VLDB Endowment*, 10(11):1166–1177, 2017.

[12] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*, pages 371–386. ACM, 2016.

[13] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant Recovery for Main-Memory Databases. In *CIDR*, 2015.