Understanding and Optimizing Persistent Memory Allocation

Wentao Cai, Haosen Wen, H. Alan Beadle, Mohammad Hedayati, Michael L. Scott

University of Rochester

Challenges

Resizable durable data structures on nonvolatile memory require an allocator that manages the persistent memory region. However, this poses additional challenges compared to transient allocators:

1. Consistency of allocator's metadata after a crash (**failure atomicity**).
2. **Memory leaks** due to failures between allocation and attachment, or between detachment and deallocation:

```
Node* t = list->tail();
t->next = malloc(sz); // sz == sizeof(Node)
          ↑
        Crash
```

3. **Pointer** reusability across executions.

Failure-induced Memory Leaks

State-of-the-art persistent allocators typically rely on one of two strategies to handle failure-induced memory leaks:

1. Failure-atomic **malloc-to** and **free-from** operations, which effectively makes (de-)allocations *durably linearizable* (and thus expensive).

```
t->next = malloc(sz) ⟹ malloc_to(t->next, sz)
```

2. Traditional malloc and free, with offline tracing *garbage collection* (GC) from *persistent roots*.

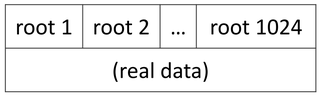


Figure 1. Example of a Memory Region

Recoverability

An allocator is **recoverable** if, in the wake of a crash, it is able to bring its metadata to a state in which all and only the *"in use" blocks* are allocated. In use blocks are defined as those that are:

- Malloc-to-ed but not yet free-from-ed, or
- Reachable from persistent roots via tracing.

Observation: Given a proper tracing mechanism, almost any transient allocator can be made recoverable by identifying the in-use blocks during recovery and reconstructing the allocator's transient metadata.

Allocator methods must also be failure-atomic, either via logging or by adding appropriate flushes and fences to nonblocking code.

This work was supported in part by NSF grants CCF-1422649, CCF-1717712, and CNS-1900803, and by a Google Faculty Research award.

Ralloc

Ralloc is the first nonblocking, recoverable allocator.[¤] Based on the (transient) LRMalloc,[1] it retains the traditional malloc/free interface and adds three key innovations:

1. **Minimum run-time overhead** by persisting just enough information to permit GC and metadata reconstruction after a crash.
2. **Filter functions** to avoid false positives in conservative GC for type-unsafe languages (e.g., C++). These enumerate internal pointers for a given type of block:

```
class TreeNode
{
   … // content fields
   TreeNode* left, *right;
}
```

```
void filter(TreeNode* ptr)
{
   visit(ptr->left);
   visit(ptr->right);
}
```

3. **Position-independent pointers** to allow persistent memory regions to be mapped at an arbitrary address.

[¤] Source code available at https://github.com/qtcwt/ralloc

Comparison to State of the Art

	PMDK[2]	Makalu[3]	Ralloc
Memory Leaks	malloc-to & free-from	malloc & free w/ Conservative GC	**malloc & free** w/ Conservative GC & Optional **Filter Function**
Atomicity	Logging	Undo Logging	**Reconstruction**
Pointers	Position Independent ID + Offset (128 bits)	Fixed Mapping Address	**Position Independent** *Off-holder** (**64 bits**)
Liveness	Lock-based	Lock-based	**Lock-free**

*An *off-holder*[4] holds the offset to the target *from the pointer itself.*

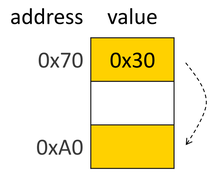
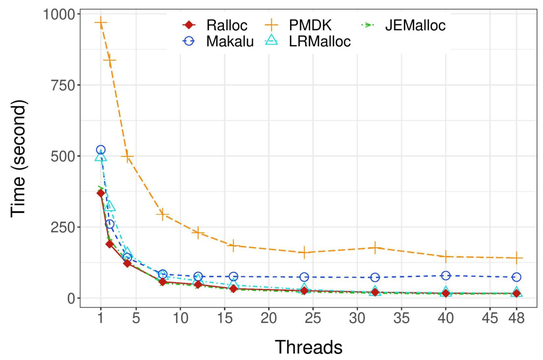


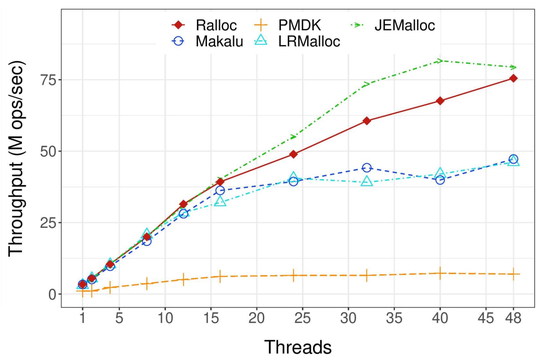
Figure 2. Example of Off-holders

Experiments

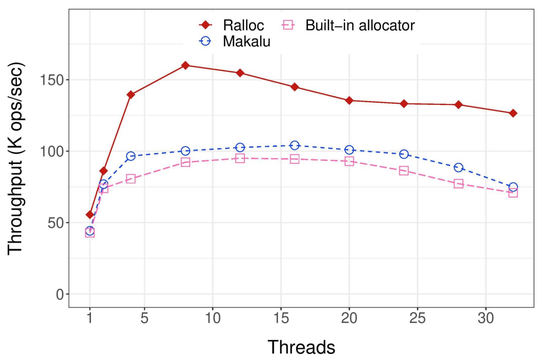
Environment: Two Intel Xeon Silver 4114 CPUs (total of 20 physical cores / 40 hyper-threads) and emulated persistent memory on DRAM.



(a) Threadtest[5] (lower is better)



(b) Larson[5] (higher is better)



(c) Memcached[6] (higher is better)

References

[1] R. Leite and R. Rocha. LRMalloc: A Modern and Competitive Lock-Free Dynamic Memory Allocator. In *VECPAR*, São Pedro, Brazil, Sept. 2018.
[2] A. Rudoff and M. Slusarz. Persistent memory development kit, Sept. 2014. http://pmem.io/pmdk/.
[3] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In OOPSLA, Amsterdam, The Netherlands, Oct. 2016.
[4] G. Chen, L. Zhang, R. Budhiraja, X. Shen, and Y. Wu. Efficient support of position independence on non-volatile memory. In *MICRO*, Cambridge, MA, Oct. 2017.
[5] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In ASPLOS, Cambridge, MA, Nov. 2000.
[6] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. An analysis of persistent memory use with WHISPER. In ASPLOS, Xi'an, China, 2017.