DeNoVo malloc: Validating Data in Persistent Memory

Arthur Martens[†]
TU Braunschweig

Rouven Scholz[*‡]
TU Braunschweig

Phil Lindow[*‡]
TU Braunschweig

Marc A. Kastner[‡]
TU Braunschweig

Rüdiger Kapitza
TU Braunschweig

Extended Abstract

① Fast, byte addressable persistent memory that maintains its state across power cycling events has already hit the market. Products based on battery-backed DRAM, have been available for more than two years and recently MRAM, FRAM or ReRAM memory is available for embedded devices. This new type of memory demands specialized system support to ensure consistency in case of power failures and data retrieval after a system restart. Moreover, due to the longevity of data stored in persistent memory, this data is also more susceptible to soft errors such as bit flips caused by radiation, wear out, or temperature effects. Being already persistent, the data won't be stored on secondary storage. Consequently, data corruptions in main memory cannot be resolved by simply restarting the system. The latter, however, limits the usability of persistent memory and poses a high risk of a permanent inconsistent system state.

② In this poster we present *DeNoVo malloc*, a dependable non-volatile memory allocator that protects the entire persistent memory in the system from power failures as well as transient faults. While the kernel components are protected with a tailored solution, the entire user space, including all memory allocated by the user, is guarded by an approach that combines the allocator with software transactional memory (STM).

③ The API of *DeNoVo malloc* is similar to the *malloc()* function provided by LibC. The only difference to traditional C or C++ code is that the programmer has to wrap all access to persistent memory into transaction atomic blocks provided by GCC. These blocks are executed with STM semantics. Using a write-back strategy, no changes are made to persistent memory if a power failure interrupts a transaction. To provide data validation on top of the traditional STM approach, we exploit the implicit code instrumentation provided by GCC's transaction atomic blocks. This allows us to control and manipulate every word access.

④ To be able to validate persistent data, whenever it is accessed for the first time during a transaction, *DeNoVo malloc* allocates twice the amount of requested memory. The additional space is used to store error correction code (ECC) words in between the original data words. In principle any ECC implementation is usable here. However, speed is an important factor to keep the performance reasonable. Therefore, our solution uses CRC32c for error detection. It has a hamming distance of 8 for a word length of 64 bit, and is implemented in hardware on x86 CPUs supporting the SSE 4.2 extension. Error correction relies on multiple trials. This is aided by an error correction halfword C to reduce the amount of possible correction candidates. Although we always have to try multiple solutions, the error correction still remains reasonable fast and any injected error, up to 8 bits, was correctly repaired by this approach.

Since all original data words are interleaved with ECC now, any access to this data outside a transaction will yield corrupted results. This kind of bug is very common especially when implementing the STM paradigm into existing applications. Moreover, accessing persistent data without transactions is unsafe in the presence of power failures. In order to mitigate this kind of bug, all addresses to persistent memory that are exposed by *DeNoVo malloc* are placed in a special Transaction Staging (TxStaging) section that has no access rights. Only if these addresses are accessed within a transaction, the addresses are transformed to the actual section where the persistent memory is mapped.

⑤ To evaluate the performance of *DeNoVo malloc*, we looked at the traversal times of a transactional linked list, several STAMP Benchmarks and Memcached. Compared to a plain STM solution, *DeNoVo malloc* achieves 50% to 90% performance, whereby transactions with a small write set and short commit period are impacted the most. For this type of transactions, however, our approach highly outperforms pmemobj, an object store for persistent memory provided by http://pmem.io.

---

[*] These authors will present the poster
[†] PhD Student
[‡] Student

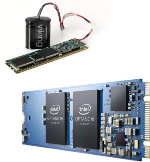
DeNoVo malloc: Validating Data in Persistent Memory

Arthur Martens, Rouven Scholz, Phil Lindow, Marc A. Kastner, Rüdiger Kapitza

TU Braunschweig | Institute of Operating Systems and Computer Networks

1 Persistent Memory is Available

- Battery-backed DRAM
- MRAM, FRAM, ReRAM
- PCM (Intel Optane) (to be released this year)

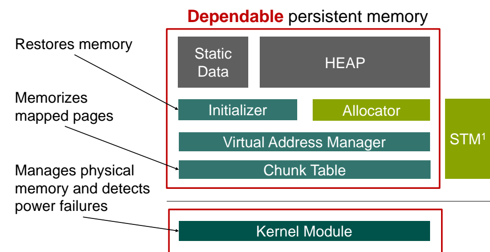


Need for System Support

- Persistent data must be:
  - Restorable
  - Consistent in case of power failures
  - **Dependable** (since data is stored for a long time)

2 Architecture

- Goal: protect entire persistent memory in the system
- Lightweight: minimize data that needs protection



1): STM is based on TinySTM library from http://tmware.org/tinystm

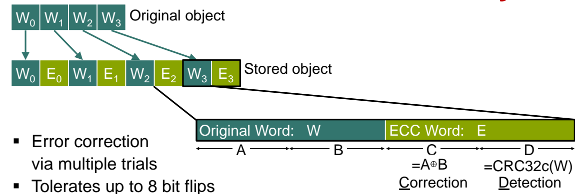
3 Programming Model

```
void push_front(widged_t widged) {
  __transaction_atomic {
    node_t* node  = dnv_malloc(sizeof(node_t));
    node->payload = widged ;
    node->next    = head_ ;
    head_         = node ;
  } // transaction commit
}
                    TM_STORE(node->next,
                             TM_LOAD(head_));
```

- State of the art:
  - Utilize STM for power failure tolerance
- DeNoVo malloc STM extension:
  - Persistent data validation on access

STM: Software Transactional Memory

4 Fault Tolerant Persistent Memory



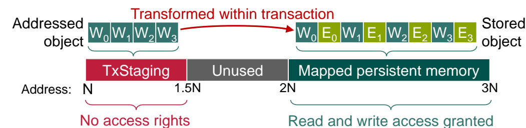
- Error correction via multiple trials
- Tolerates up to 8 bit flips

Programming Support

- Access to persistent memory without transactions:
  - …is easily overseen
  - …will almost always yield corrupted data
  - …is unsafe in case of power failures

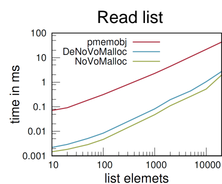
  } needs to be prevented

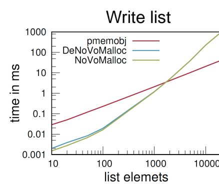
➔ Keep addresses to persistent memory always within **TxStaging** section

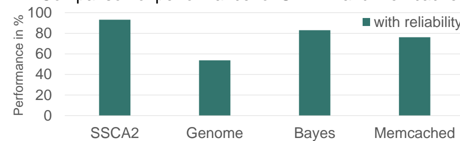


5 Evaluation

- Dependable access achieves 50% to 90% of performance compared to pain STM
- Most performance impact on read intensive workload
- Always outperforms pmemobj for transactions with a small write set







STAMP: Stanford Transactional Applications for Multi-Processing

pmemobj: persistent object store provided by http://pmem.io

Technische Universität Braunschweig

DFG

danceOS

Institute of Operating Systems and Computer Networks