Clobber-NVM: Log Less, Re-execute More

Yi Xu, Joseph Izraelevitz, Steven Swanson

UC San Diego & University of Colorado, Boulder

Published on ASPLOS 2021
Persistent Memory Programming

Application

Cache

PMEM

DRAM
Persistent Memory Programming

- Persistent memory is byte-addressable.
- Persistent over power failures.
- Delivers DRAM-class latency/BW

Application → Cache → PMEM → DRAM
Persistent Memory Programming

Cache

The cache is volatile. **Cached updates will be dropped** after a power loss.
Persistent Memory Programming

Application

PMEM application needs crash consistency

PMEM  DRAM

NVSL
Persistent Memory Programming

PMEM libraries provide the means to apply sets of writes to persistent memory atomically.

Unfortunately, most current libraries impose significant overhead.
### PMEM Program with Undo Logging

#### DRAM program

```c
void list_push(list_t *list, char* value){
    memcpy(list->buf[list->size], value, strlen(value);
    list->size++;
}
```

#### PMEM program with undo logging

```c
void list_push(list_t *list, char* value){
    undo_log(value, strlen(value));
    persist_barrier();
    memcpy(list->buf[list->size], value, strlen(value));
    undo_log(list->size, sizeof(size_t));
    persist_barrier();
    list->size++;
}
```
PMEM Program with Undo Logging

DRAM program

```c
void list_push(list_t *list, char* value)
{
    memcpy(list->buf[list->size], value, strlen(value);
    list->size++;
}
```

PMEM program with undo logging

```c
void list_push(list_t *list, char* value){
    undo_log(value, strlen(value));
    persist_barrier();
    memcpy(list->buf[list->size], value, strlen(value);
    undo_log(list->size, sizeof(size_t));
    persist_barrier();
    list->size++;
}
```
PMEM Program with Undo Logging

DRAM program

```c
void list_push(list_t *list, char* value){
    memcpy(list->buf[list->size], value, strlen(value);
    list->size++;
}
```

PMEM program with undo logging

```c
void list_push(list_t *list, char* value){
    undo_log(value, strlen(value));
    persist_barrier();
    memcpy(list->buf[list->size], value, strlen(value);
    undo_log(list->size, sizeof(size_t));
    persist_barrier();
    list->size++;
}
```
PMEM Program with Undo Logging

void list_push(list_t *list, char* value){
    undo_log(value, strlen(value));
    persist_barrier();
    memcpy(list->buf[list->size], value, strlen(value));
    list->size++;
}

Are the logs and barriers necessary?
Recover Through Re-execution

- PMEM
  - Writes
  - Reads
Recover Through Re-execution

PMEM
Recover Through Re-execution

PMEM
Recover Through Re-execution

PMEM
Recover Through Re-execution

PMEM
Recover Through Re-execution

PMEM
Recover Through Re-execution

PMEM
Recover Through Re-execution

PMEM
Recover Through Re-execution

PMEM
Recover Through Re-execution

PMEM
Recover Through Re-execution
Recover Through Re-execution

Re-execute on data that could be inconsistent?
Use **Clobber-NVM**!
Clobber-NVM Program

```c
void list_push(list_t *list, char* value){
    txbegin();
    memcpy(list->buf[list->size], value, strlen(value));
    list->size++;
    txend();
}
```
void list_push(list_t *list, char* value){
  txbegin();
  memcpy(list->buf[list->size], value, strlen(value));
  list->size++;
  txend();
}
void list_push(list_t *list, char* value) {
    txbegin();
    memcpy(list->buf[list->size], value, strlen(value));
    list->size++;
    txend();
}
Clobber-NVM Program

PMEM program with Clobber-NVM

```c
void list_push(list_t *list, char* value)
{
    txbegin();
    memcpy(list->buf[list->size], value, strlen(value));
    list->size++;
    txend();
}
```
void list_push(list_t *list, char* value){
    txbegin();
    memcpy(list->buf[list->size], value, strlen(value));
    list->size++;
    txend();
}
A transaction input is a clobbered input if it may be overwritten within this transaction, and this write is a clobber write.
Clobbered inputs are a problem for re-execution.
Clobbered inputs are a problem for re-execution.
Clobbered inputs are a problem for re-execution.
Clobber_Log before Clobber Writes

Clobber_Log --- undo logs before clobber writes.
Clobber_log before Clobber Writes
Clobber_log before Clobber Writes
Re-execute Based on Clobber_log

PMEM
Re-execute Based on Clobber_log
Re-execute Based on Clobber_log
Re-execute Based on Clobber_log
Re-execute Based on Clobber_log
Handle DRAM Accesses
Handle DRAM Accesses
Handle DRAM Accesses
Handle DRAM Accesses on v_log

v_log stores volatile transaction inputs
Handle DRAM Accesses

PMEM

Clobber Log

DRAM

V Log
Handle DRAM Accesses

PMEM

DRAM
Handle DRAM Accesses
Handle DRAM Accesses

PMEM

DRAM
Evaluation Setup

- Platform: two 24-core Intel Cascade Lake SP processors, running at 2.2 GHz. The platform has a total of 192 GB of DRAM and 1.5 TB (6 ×256 GB) of Intel Optane DC Persistent Memory directly attached to each processor.
- Configured Optane DCPMM in 100% App Direct mode.
- All experiments use Ext4 to manage persistent pools and directly access NVM pages via DAX.
Data structure Benchmarks

![Data structure Benchmarks Diagram]

Throughput (Mops)

- **Bptree**
- **Hashmap**
- **Rbtree**
- **Skiplist**

- **PMDK**
- **Clobber-NVM**
Memcached Performance

![Memcached Performance Diagram](image)
Conclusion

- Clobber-NVM: Recovers by re-executing interrupted transactions.
- Clobber-NVM compiler: Identifies necessary log entries, and automatically adds logging for selected variables.
- Evaluation shows that Clobber-NVM has high performance.