

\textbf{libcrpm: Improving the Checkpoint Performance of NVM}

Feng Ren*  
Tsinghua University  
rfl7@mails.tsinghua.edu.cn

Kang Chen†  
Tsinghua University  
chenkang@tsinghua.edu.cn

Yongwei Wu*  
Tsinghua University  
wuyw@tsinghua.edu.cn

Abstract

\textit{libcrpm} is a new programming library to improve the checkpoint performance for applications running in NVM. It proposes the failure-atomic differential checkpointing protocol, which addresses two problems simultaneously that exist in the current NVM-based checkpoint-recovery libraries: (1) high write amplification when page-granularity incremental checkpointing is used, and (2) high persistence costs from excessive memory fence instructions when fine-grained undo-log or copy-on-write is used. Evaluation results show that libcrpm reduces the checkpoint overhead in realistic workloads. For MPI-based parallel applications such as LULESH, the checkpoint overhead of libcrpm is only 44.78\% of FTI, an application-level checkpoint-recovery library.

1 Introduction

Checkpoint-recovery [14] is a common programming paradigm for building recoverable applications. It typically follows an epoch-based model in which each epoch consists of an execution period and a checkpoint period. During the checkpoint period, the application suspends the execution, then saves the checkpoint state (i.e., necessary data for recovering the current execution) in persistent storage. After a system crash, the application can restore the latest checkpoint state from the persistent storage and continue its execution. Reducing the time of the checkpoint period is critical to minimize the disturbance to application execution. Recently, with the advent of non-volatile memory (NVM), such as Intel Optane DC Persistent Memory Module (DCPMM) [3], many checkpoint-recovery systems [7, 9, 10, 12, 15, 18] try to use the high performance of NVM to reduce the checkpoint overhead (i.e., additional execution time due to checkpoint). Such systems also benefit from the NVM’s direct memory access, i.e., without data serialization and deserialization that are usually needed for saving checkpoint states on disks.

However, existing works [10, 12, 15] cannot fully utilize the power of NVM because they often follow the traditional method that only treats NVM as a faster storage device. We find out that two problems limit the checkpoint performance using NVM.

(P1) \textbf{Page-granularity incremental checkpointing leads to high write amplification for NVM} (i.e., written data is larger than modified data). To reduce the amount of data that needs to be saved during checkpointing, many libraries implement incremental checkpointing, i.e., only to save changed program states. Thus, they have to trace changed program states during an epoch. The memory change tracing is often implemented by using the page fault mechanism, such as the mprotect() system call [15] and the soft-dirty bit technique [6]. The page fault mechanism detects the page-level (4KB page or larger) modification. However, NVM is byte-addressable, and the storage media of NVM has a smaller access granularity (e.g., 256B in DCPMM [3]). This mismatching will lead to copying the whole page during the checkpoint period even if only one cache line is actually modified.

(P2) \textbf{Finer-granularity checkpointing requires excess memory fences that increase persistent costs in NVM.} Since NVM provides a byte-addressable interface, we also investigated existing in-memory checkpoint-recovery libraries [17, 19] (often used for debugging). To reduce the checkpoint overhead, these libraries use static instrumentation to avoid page-level tracing while can still detect all modifications. They also use undo-log/copy-on-write mechanisms to keep the checkpoint state consistent. It is possible to transform these volatile checkpoint methods into non-volatile ones (i.e., making checkpoint data available after a crash) by using persistent instructions (e.g., clwb and sfence) — every time after appending a new undo-log/copy-on-write entry, clwb instructions will be used to flush data to NVM, followed by sfence instructions to guarantee the persistent order. Such transformation has to use more memory fences which incur non-trivial overhead [11].

This paper proposes \textit{libcrpm}, a programming library that provides the Checkpoint-Recovery interface using Persistent Memory. \textit{libcrpm} captures memory changes in finer granularity using static instrumentation. We propose a new checkpointing protocol that both \textbf{shrinks the amount of data to be checkpointed} (P1) and \textbf{reduces the fence instructions needed} (P2). To the best of our knowledge, \textit{libcrpm} is the first solution that addresses both problems simultaneously in software, i.e., without changing hardware.

To achieve both goals, we redesign (1) the in-NVM compact memory layout for checkpoint-recovery, and (2) the checkpointing protocol that updates checkpoint states. The compact memory layout contains two regions, the main region is visible to applications, while the \textit{backup region} saves additional data to build the checkpoint state. Different from DICE [7], the checkpoint state keeps consistent even if the application is interrupted by system crashes during checkpointing. Both regions are partitioned into \textit{segments} (copy-on-write granularity, 2MB each) and further partitioned into \textit{blocks} (data copy granularity, 256B each). During checkpointing, the main region (containing the current program state) is atomically set as a new checkpoint state. \textit{libcrpm} performs segment-level copy-on-write to keep the checkpoint state consistent. Only two sfence instructions are needed per segment, so a relatively large segment reduces the persistence overhead from memory fences. We
also manage data changes at the block level using memory tracing based on static instrumentation. Thus, only dirty blocks are copied instead of the whole segment during copy-on-write. This reduces the memory copy costs for checkpointing.

We measure the performance of libcrpm using real-world workloads. The throughput of persistent unordered_map using libcrpm is up to 2.72x higher than Dalí [16]. For MPI-based parallel applications like LULESH [13], the checkpoint overhead of libcrpm is 44.78% of FTI [8], an application-level checkpoint-recovery library.

In summary, this paper makes the following contributions:

- Performing copy-on-write at segment granularity and copying data at block granularity solve the dilemma in the trade-off between extra NVM data writes and persistence overhead.
- We implement libcrpm, an NVM programming library that enables the checkpoint-recovery semantic to applications. Evaluations show that libcrpm can reduce the checkpoint overhead of realistic workloads.

2 Background and Motivation

2.1 Checkpoint-recovery using NVM

Checkpoint-recovery is a well-known technique for recoverable applications. It is widely used in high-performance computing [14] and data storage [16]. Most checkpoint-recovery libraries use the epoch-based model. Each epoch consists of an execution period and a checkpoint period. An application saves its current state in persistent storage during the checkpoint period. It loads the latest checkpoint state from the persistent storage to restore the execution after a crash and restart. Incremental checkpointing [10, 12] is a key technique to reduce the amount of data during checkpointing, which only stores the differences between the last checkpoint and the current state.

With the availability of NVM, we expect that the checkpoint overhead can be reduced using this faster storage device. Its direct memory access mode also avoids data serialization and deserialization. Many recent checkpoint-recovery systems [7, 9, 10, 12, 18] have switched from HDD/SSD to NVM. Some of them [10, 18] require hardware modifications to the memory architecture. Besides, some NVM data structures (e.g., Dalí [16]) keep data persistence at low costs by frequent checkpointing.

Persistence overhead is not negligible in NVM programs. The main reason for this overhead is from the explicit memory flushes. Platforms with volatile in-CPU caches need to explicitly use the clwb instruction to flush data from the cache to NVM. And then, sfence prevents the store instruction from reordering. Both instructions are costly compared to other instructions [11].

### 2.2 Empirical Analysis of Checkpoint Overhead

We measure the checkpoint overhead by the execution time of the unordered_map with different checkpoint-recovery implementations. The experimental setup is described in §5. Figure 1 shows the execution time breakdown for the balanced workload (50% update and 50% get). The checkpoint interval is 128ms.

2.2.1 Page-granularity Incremental Checkpointing. To implement incremental checkpointing, traditional checkpoint-recovery libraries use the page fault mechanism provided by the operating system such as mprotect [15] and the soft-dirty bit [6]. At the beginning of each epoch, all pages are marked as read-only. A page fault exception allows the library to detect the page modification event and then makes the page writable. Only the modified pages will be saved in NVM during the checkpoint period.

We found out that memory change tracing is expensive because of the high latency of page faults (about 2us per 4KB page). For example, mprotect takes about 48% of the total execution time for memory change tracing. Moreover, page-level incremental checkpointing also increases write amplification (P1). The page fault mechanism detects page-level (4KB or larger) modifications. However, the storage media of NVM has a smaller access granularity (e.g., 256B in DCPMM [3]). This mismatching leads to store the full page during the checkpoint period even if only one cache line is modified. As a result, checkpoint has significant overhead (42% and 66% of the total time for mprotect and soft-dirty bit respectively).

2.2.2 Fine-grained Checkpointing. We also measure the data persistence overhead by transforming in-memory checkpointing (Undo-log [19] and LMC [17]) to the persistent versions. These libraries resort to static instrumentation, avoiding the overhead from page faults. The instrumented code will create undo-entries (undo-logs or copy-on-write records) before any memory modification. The size of each undo-entry (excluding metadata) is 256B. These undo entries are deleted after completing a checkpoint. Such instrumentation-based in-memory checkpointing has a low overhead, and the checkpointing frequencies can be very high.

When appending a undo entry, the transformed versions make it persistent immediately using clwb and sfence instructions. At the end of each epoch, the current program state is flushed into NVM before truncating any undo-entries. However, excessive memory fences have high persistence costs (P2). The memory tracing (including appending undo entries) becomes the performance bottleneck in our test (49% and 46% of the total execution time for undo-log and LMC, respectively). Profiling shows that excessive memory fence instructions incur high persistence overhead. Two memory fence instructions are issued every time appending an undo-entry, one for the undo-entry and the other for updating the metadata.

### 3 Design

3.1 Overview

libcrpm is a pure software solution. Figure 2 shows its architecture, consisting of a customized compiler and a runtime library.

The customized compiler allows libcrpm to identify the dirty data in a finer granularity other than page-level granularity. Before each instruction that may modify program state objects, a call hook_rout_line(addr, len) instruction is inserted to mark memory area [addr, addr + len) dirty at runtime. The compiler also
After the checkpoint period, main segments save the working state, and (R2) the backup segment saves the checkpoint state. A segment-level copy-on-write (CoW) example.

3.3 Compacted Memory Layout

3.4.1 Segment state array. (seg_state, with nr_main_segments elements) is a list of segments that save the checkpoint state. The i-th element (segment state of Mi) can be either – (1) SS_Initial: M_i does not store program state; (2) SS_Main: M_i saves the checkpoint state; or (3) SS_Backup: B_j saves the checkpoint state, where B_j is the paired backup segment of M_i. For crash consistency, the metadata contains two seg_state arrays. If the committed epoch is e, seg_state[e%2] is active and used for the checkpoint state.

3.4 Failure-atomic Differential Checkpointing

3.4.1 Segment-level Copy-on-Write. To reduce the use of s_fence instructions, libcrpm implements copy-on-write at the segment level (Figure 5): After the checkpoint period, main segments save the checkpoint state, and they are virtually read-only. Before modifying data in the main segment M_i, a copy-on-write is triggered. We make the whole data of paired backup segment B_j identical to M_i. The segment state of M_i switches from SS_Main to SS_Backup. M_i is writable after copy-on-write completes, and modifications to M_i in the current epoch do not corrupt the checkpoint state.

To reduce the amount of data to be checkpointed, block-based data copy is used. Initially, data in B_j is equal to M_i. During the next execution period, some memory blocks in M_i are modified by applications. Therefore, only these blocks in M_i that are different from B_j, i.e., by copying these blocks, data in B_j equal to M_i again. To record a block being modified during an epoch, libcrpm uses the dirty block bitmap (dirty_blocks) in DRAM. Both dirty block/segment recording and segment-level copy-on-write are triggered by the instrumented code. Figure 6 shows the pseudo-code.
```
def copy_on_write(ctx, main):  # ctx: abbreviation of container
    lock(main.lock)
    e = ctx.committed_epoch & 2
    if e == 2:  
        if ctx segmented_state[ctx][main] == SS_Main:
            return False
        backup = find_paired_backup_segment(ctx, main)
        ctx segmented_state[ctx][main] = ctx segmented_state[ctx][backup]
        persist_copy(backup, main, SegmentSize)
        return True
    else:
        delta = backup.base_addr - main.base_addr
        if not diff_ckpt:  # a new backup segment allocated
            # data already in DRAM
            persist_copy(ctx segmented_state[ctx][main], SS_Backup); sfence()
        else:
            sfence(); barrier()
            ctx segmented_state[ctx][main] = SS_Backup
            persist_fetch_and_add(ctx segmented_state[ctx][main], 1); sfence()
            ctx segmented_state[ctx][main] = ctx segmented_state[ctx][backup]
            persist_copy(backup, main, SegmentSize)
    sfence(); barrier()
    persist(ctx segmented_state[ctx][main], SS_Backup); sfence()
    ctx segmented_state[ctx][main] = SS_Main
    for s in ctx segmented_state[ctx][main].dirty_segments:
        ctx segmented_state[ctx][main] = ctx segmented_state[ctx][main - 1]
    if is_leader:
        sfence(); barrier()
        wbinvd()
    else:
        clwb(b)
        for b in ctx segmented_state[ctx][main].dirty_blocks:
            # distribute to each thread
            if len(ctx segmented_state[ctx][main].dirty_blocks) < Threshold:
                is_leader = assign_leader()
                barrier()
                def crpm_checkpoint():
                    ctx, block, valid = locate_ctx_and_block(addr, length)
                    copy_on_write(ctx, block.segment)
                    ctx segmented_state[ctx][main].dirty_segments.add(s)
                    persist_store(ctx segmented_state[ctx][main], SS_Backup); sfence()
                    sfence()
                    return True
                def hook_routine(addr, length):  # instrumented code
                    copy_on_write(ctx, block.segment)
                    ctx segmented_state[ctx][main].dirty_segments.add(s)
                    persist_store(ctx segmented_state[ctx][main], SS_Backup); sfence()
                    sfence()
            # trigger when opening a container
            e = ctx.committed_epoch % 2
            if e == 2:
                if ctx segmented_state[ctx][main] == SS_Main:
                    e = ctx.committed_epoch % 2
                    for s in ctx segmented_state[ctx][main].dirty_segments:
                        ctx segmented_state[ctx][main] = ctx segmented_state[ctx][main - 1]
                    if is_leader:
                        sfence(); barrier()
                        wbinvd()
                    else:
                        clwb(b)
                        for b in ctx segmented_state[ctx][main].dirty_blocks:
                            # distribute to each thread
                            if len(ctx segmented_state[ctx][main].dirty_blocks) < Threshold:
                                is_leader = assign_leader()
                                barrier()
                                def crpm_checkpoint():
                                    ctx, block, valid = locate_ctx_and_block(addr, length)
                                    copy_on_write(ctx, block.segment)
                                    ctx segmented_state[ctx][main].dirty_segments.add(s)
                                    persist_store(ctx segmented_state[ctx][main], SS_Backup); sfence()
                                    sfence()
5 Evaluation

5.1 Setup

All experiments are conducted on a dual-socket Intel Xeon Gold 6240R 2.4GHz server. Each socket has 48 logical cores, 192 GB DRAM, and 768 GB DCMM. All benchmarks run on a single processor to avoid overheads from inter-socket NVM access [3]. Our machine runs Ubuntu 20.04 (Linux kernel 5.4.0).

We compare the following systems: (1) Mprotect and soft-dirty bit — incremental checkpointing using mprotect and soft-dirty bit respectively. (2) Undo-log — generating and persisting undo logs before memory modifications. (3) LMC — a lightweight memory checkpointing library [17] that tolerances power failures. (4) Dali [16] — a periodically persistent hash map. (5) NVM-NP — data structures running in NVM but with No Persistence instruction used. (6) FTI [8] — generating full checkpoints using FTI with multi-level checkpointing disabled. (7) libcrpm-Default and libcrpm-Buffered — libcrpm with default protocol (§3.4) and buffered mode (§3.5) respectively. Both undo-log and LMC require static instrumentation. The instrumented code creates undo-logs/copy-on-write records before any modification (§2). The size of each record is 256B.

5.2 End-to-end Performance

5.2.1 Data Structures. Many applications keep their states using data structures. We build two periodically persistent data structures based on the C++ Standard Template Library (STL): (1) map, a red-black tree; and (2) unordered_map, an unordered hash table. A wrapper class CrpmAllocator is used to replace the default allocator. Passing it as one of the template parameters, elements are allocated from a container. The compiler will instantiate the template and then instrument the instantiated code. As a result, a single line of code change will enable recoverable data structures.

For each test, 24M keys are populated initially (except the insert-only workload). Both keys and values are 8 bytes. Then we perform the following workloads: (1) Insert-only; (2) Balanced: 50% update, 50% get; (3) Read-heavy: 5% update, 95% get; and (4) Read-only: 100% get. For the insert-only workload, we measure the time of inserting 5M entries, where keys are uniformly distributed. We properly set the load factor to avoid hash table resizing. For other workloads, keys are generated in a Zipfian distribution with parameter $\alpha = 0.09$. The execution period of each epoch is 128ms.

Result: Figure 7 shows the throughput under different workloads. Compared with NVM-NP, 1libcrpm-Default supports checkpoint-recovery at the cost of increasing its execution time by 13.7% under the balanced workload. The throughput of unordered_map using 1libcrpm-Default is up to 7.23x and 7.08x higher than mprotect and soft-dirty bit respectively, because 1libcrpm reduces both memory tracing overhead and the checkpoint size. 1libcrpm-Default also has up to 1.47x and 1.38x higher throughput than undo-log and LMC respectively, because segment-level copy-on-write requires fewer memory fences. We will further discuss the reasons in §5.3. The throughput of unordered_map using 1libcrpm-Default is 1.80x/2.72x higher than Dali in the insert-only/balanced workloads respectively. For the read-only workload, as there is nothing to be checkpointed, 1libcrpm-Default can run as fast as NVM-NP.

5.2.2 Parallel Computing Applications. We transform three well-known applications for checkpoint-recovery support: LULESH [13] (90/5546), HPCCG [2] (38/1652), and CoMD [1] (22/3054). This is done by replacing memory allocation functions and adding checkpoint logic. We measure the execution time with different input datasets and checkpoint-recovery systems. Program states of each application are buffered in DRAM, so only FTI and 1libcrpm-Buffered are evaluated. We run each application with eight processes in a single machine, and checkpoints are generated every five iterations.

Result: Figure 8 reports the relative execution time of FTI and 1libcrpm-Buffered (i.e., execution time without checkpointing is normalized to 1.0 for each workload). For LULESH, 1libcrpm-Buffered supports fault tolerance at the cost of 5.16% of extra execution time (10.25x) if the input dataset size is 903, while FTI is 11.53% (22.89x). The checkpoint overhead of 1libcrpm-Buffered is only 44.78% of FTI. For both HPCCG and CoMD, 1libcrpm-Buffered reduces the checkpoint overhead by 49.83% – 81.85%, compared to FTI.

5.3 Effectiveness of Design Choices

This section shows how 1libcrpm can mitigate performance problems that existed in previous works (§2). We report the result of persistent unordered_map under the balanced workload (§5.2.1).

Checkpoint size. As shown in Table 1a, the average checkpoint size per operation is significantly reduced by 91.56%, 94.30%, and 93.86% compared to FTI.

Table 1: Detailed analysis for persistent unordered_map.

<table>
<thead>
<tr>
<th></th>
<th>Insert-only</th>
<th>Balanced</th>
<th>Read-heavy</th>
</tr>
</thead>
<tbody>
<tr>
<td>mprotect</td>
<td>3,190</td>
<td>987</td>
<td>117</td>
</tr>
<tr>
<td>Soft-dirty bit</td>
<td>1,303</td>
<td>872</td>
<td>846</td>
</tr>
<tr>
<td>1libcrpm-Default</td>
<td>269</td>
<td>56</td>
<td>7</td>
</tr>
<tr>
<td>LMC</td>
<td>222,702</td>
<td>184,574</td>
<td>40,584</td>
</tr>
<tr>
<td>1libcrpm-Buffered</td>
<td>333</td>
<td>281</td>
<td>8</td>
</tr>
</tbody>
</table>

(a) Average checkpoint size in bytes per operation.

(b) Number of fence instructions issued per epoch.
Throughput (MOP/s)

With different segment sizes (512B to 32MB), while the block size work is executed during the execution period. The throughput of 128.

Checkpoint interval is soft-dirty. The result is shown in Figure 9. At higher frequencies, For balanced & read-heavy workloads, the maximal throughput is 1. For LULESH with libcrpm-Buffered, the size of checkpoint states is 258MB per process if the input dataset size is 90. It is 1.35× larger than FTI, because checkpoint states of 1.libcrpm are not serialized. The checkpoint size of 1.libcrpm is 187MB per epoch. 1.libcrpm requires 258MB DRAM as the in-memory buffer, and 452MB NVM as main/backup regions. In-NVM metadata size of the container is less than 3KB, while the dirty block bitmap takes 129KB DRAM.

6 Conclusion

In this paper, we describe 1.libcrpm, a general-purpose checkpoint-recovery programming library using NVM. The failure-atomic differential checkpointing technique reduces both write amplification and persistence costs. Our evaluation result shows that 1.libcrpm reduces both coding efforts and the execution overhead.

Acknowledgments

This work is supported by National Key Research & Development Program of China (2020YFC1522702), and Natural Science Foundation of China (62141216, 61877035). This research was supported partly by Tsinghua University – Meituan Joint Institute for Digital Life, and Beijing HaiZhi XingTu Technology Co., Ltd.

References

[1] [n.d.]. CoMD. https://github.com/EC2-copac/CoMD.
[2] [n.d.]. HPCCG. https://github.com/Mantevo/HPCCG.
[10] Swapnil Haria et al. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In ASPLOS.

5.5 Recovery Time

We measure the recovery time by killing and restarting LULESH ($\S 5.2.2$) processes. The recovery time is proportional to the size of the program state: 288ms if the input dataset size is 90, and 517ms if the input dataset size is 110. During recovery, 1.libcrpm-Buffered firstly makes the working state consistent with the checkpoint state (43% – 56% of the total recovery time), and then copies data in the main region to DRAM (this is not used in libcrpm-Default).

6.6 Storage Cost

For LULESH with libcrpm-Buffered ($\S 5.2.2$), the size of checkpoint states is 258MB per process if the input dataset size is 90. It is 1.35× larger than FTI, because checkpoint states of 1.libcrpm are not serialized. The checkpoint size of 1.libcrpm is 187MB per epoch. 1.libcrpm requires 258MB DRAM as the in-memory buffer, and 452MB NVM as main/backup regions. In-NVM metadata size of the container is less than 3KB, while the dirty block bitmap takes 129KB DRAM.