`libcrpm`: Improving the Checkpoint Performance of NVM

Feng Ren*
Tsinghua University
rf17@mails.tsinghua.edu.cn

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

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

Abstract

`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), e.g., 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) *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) *Finer-granularity checkpointing requires excessive 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 `libcrpm`, a programming library that provides the Checkpoint-Recovery interface using Persistent Memory. `libcrpm` captures memory changes in finer granularity using static instrumentation. We propose a new checkpointing protocol that both **shrinks the amount of data to be checkpointed (P1)** and **reduces the fence instructions needed (P2)**. To the best of our knowledge, `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 *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 **segments** (copy-on-write granularity, 2MB each) and further partitioned into **blocks** (data copy granularity, 256B each). During checkpointing, the main region (containing the current program state) is atomically set as a new checkpoint state. `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

*Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China.
†Kang Chen is the corresponding author.

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530536

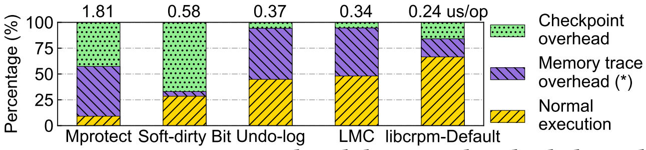

Figure 1: Execution time breakdown under the balanced workload using `unordered_map`[1].

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 2.72× 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].

[1] Appending undo-log or copy-on-write entries are considered as the *memory trace overhead* for undo-log and LMC respectively.

[2] eADR [3] makes in-CPU cache in the non-volatile domain, which can further eliminate the use of `clwb`. As eADR is not widely available, this paper focuses on the platforms with the volatile cache.

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 128*ms*.

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 find out that memory change tracing is expensive because of the high latency of page faults (about 2*us* 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 (includes 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_routine(addr, len)` instruction is inserted to mark memory area [`addr`, `addr` + `len`) dirty at runtime. The compiler also

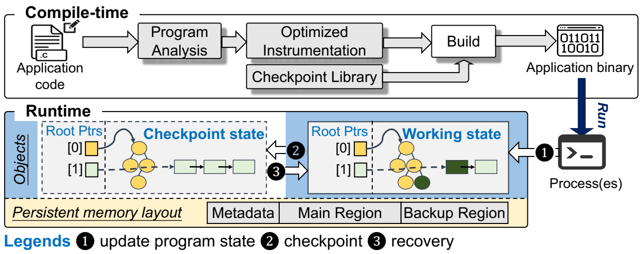

Figure 2: `libcrpm` overview.

```
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv); MPI_Comm_size(MPI_COMM_WORLD, &numRanks);
    crpm_option_t opt; init_option(&opt);
    ctr = crpm_mpi_open("/path/to/file", &opt, MPI_COMM_WORLD);
    locDom = (Domain *) crpm_get_root(ctr, 0);
    if (!locDom) {
        locDom = crpm_malloc(sizeof(Domain));
        new (locDom) Domain(numRanks, /* other arguments */);
        crpm_set_root(ctr, 0, locDom);
    }
    while ((locDom->time()<locDom->stoptime())&&(locDom->cycle()<opts.its)) {
        TimeIncrement(*locDom); LagrangeLeapFrog(*locDom);
        if (!(locDom->cycle()%(opts.numCkptIter)))
            crpm_mpi_checkpoint(ctr);
    }
    crpm_mpi_close(ctr); MPI_Finalize(); return 0;
}
```

Figure 3: Code snippets of the LULESH application.

removes unnecessary instrumentation and changes the position of some instrumented code to reduce memory tracing overhead.

In `libcrpm`, all objects have two states (§3.3): (S1) the *working state*, which is accessible to applications, and (S2) the *checkpoint state*, which is identical to the working state at the last checkpoint period. To save both states, two regions in NVM are used: (R1) the *main region* saves the working state, and (R2) the *backup region* saves differential data between the working and checkpoint states. Both regions are partitioned into segments (2MB each). A copy-on-write *virtually* replicates data of the entire segment using two `sfence` instructions (for persisting metadata). Large segment size reduces the metadata overhead and *persistence costs* (number of memory fences used). Each segment is further divided into blocks (256B each). Only *necessary blocks* are actually copied during a copy-on-write, while others are skipped. This reduces the *checkpoint size* (i.e., the amount of copied data during a checkpoint [10]).

After a successful checkpoint (§3.4.2), the current working state becomes a *new* checkpoint state. A segment-level copy-on-write (CoW) (§3.4.1) copies necessary data in the main region to a backup one. Data in the backup region plus the unmodified data in the main region make up the *consistent* checkpoint state. After a crash and restart, the working state can be recovered from the checkpoint state (§3.4.3). In addition, `libcrpm` can buffer state objects to DRAM (§3.5) and it also supports MPI-based parallel programs (§3.6).

3.2 Programming Interface

`libcrpm`'s APIs help developers define program states to be saved during the checkpoint period. Most APIs are self-explaining. When opening a container, the latest checkpoint state is mapped into the virtual memory address space of the current process. The *root pointer array* is used for retrieving objects after a restart. The checkpoint function call works in *collective* mode: each thread executes `crpm_checkpoint()` and blocks until other threads have entered this function (i.e., nobody modifies the container's data). `libcrpm` also supports MPI programs, and `crpm_mpi_checkpoint()` establishes globally consistent checkpoints for multiple containers. Figure 3 illustrates fault-tolerant LULESH [13] using `libcrpm` APIs.

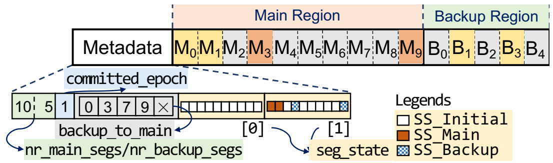

Figure 4: Persistent memory layout of a container.

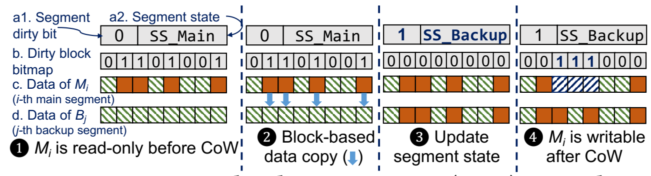

Figure 5: Segment-level copy-on-write (CoW) example.

3.3 Compacted Memory Layout

Figure 4 shows the persistent memory layout. Both the main and backup *regions* are divided into `nr_main_segs` and `nr_backup_segs` *segments* respectively. Each segment is numbered according to the address offset. The maximal modified segments per epoch is limited by the number of backup regions. To keep the checkpoint state consistent, `libcrpm` includes two data structures in the metadata:

(1) Backup-to-main-segment mapping array (`backup_to_main`, with `nr_backup_segs` elements) records the *paired backup segment* B_j (j-th segment in the backup region) of a *main segment* M_i (i-th segment in the main region). For example, in Figure 4, B_1 is the paired backup segment of M_3, because `backup_to_main[1]==3`. Either M_i or B_j saves the checkpoint state. For main segments with no paired backup segments (e.g., M_1), a copy-on-write allocates one in the backup region if needed. A backup segment B_j can be allocated if it is not used for saving the checkpoint state.

(2) Segment state array (`seg_state`, with `nr_main_segs` elements) is a list of segments that save the checkpoint state. The i-th element (segment state of M_i) 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 `sfence` instructions, `libcrpm` implements copy-on-write at the *segment* level (Figure 5): ❶ 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. ❷ We make the whole data of paired backup segment B_j identical to M_i. ❸ The segment state of M_i switches from `SS_Main` to `SS_Backup`. ❹ 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, ❷ *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 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.

```
1  def copy_on_write(ctr, main):              # ctr: abbreviation of container
2      lock(main.lock)
3      e = ctr.committed_epoch % 2
4      if ctr.seg_state[e][main] == SS_Main:
5          backup, diff_ckpt = find_paired_backup_segment(ctr, main)
6          if not diff_ckpt:                   # a new backup segment allocated
7              persist_copy(backup, main, SegmentSize)
8          else:                                   # use block-based data copy
9              delta = backup.base_addr - main.base_addr
10             for b in main.blocks:
11                 if ctr.dirty_blocks.contains(b):
12                     persist_copy(b + delta, b, BlockSize)
13         sfence()
14         persist_store(ctr.seg_state[e][main],SS_Backup);sfence()
15         ctr.dirty_blocks.clear(main.blocks)
16     ctr.dirty_segments.add(s)
17     unlock(main.lock)
18
19 def hook_routine(addr, length):                           # instrumented code
20     ctr, block, valid = locate_ctr_and_block(addr, length)
21     if not valid: return               # do not proceed if address is invalid
22     if not ctr.dirty_segments.contains(block.segment):
23         copy_on_write(ctr, block.segment)
24     ctr.dirty_blocks.add(block)
25
26 def crpm_checkpoint(ctr):
27     is_leader = assign_leader()
28     barrier()                                # synchronize multiple threads
29     if len(ctr.dirty_blocks) < Threshold:
30         for b in ctr.dirty_blocks:               # distribute to each thread
31             clwb(b)
32     else:
33         wbinvd()
34     sfence(); barrier()
35     if is_leader:
36         e = (ctr.committed_epoch + 1) % 2
37         ctr.seg_state[e] = ctr.seg_state[1-e]
38         for s in ctr.dirty_segments:
39             ctr.seg_state[e][s] = SS_Main
40         persist(ctr.seg_state[e]); sfence()
41         persist_fetch_and_add(ctr.committed_epoch, 1); sfence()
42         ctr.dirty_segments.clear_all()
43     barrier()
44
45 def crpm_recovery(ctr):                    # trigger when opening a container
46     e = ctr.committed_epoch % 2
47     for backup in ctr.back_segments:
48         main = ctr.backup_to_main[backup]
49         state = ctr.seg_state[e][main]
50         if state == SS_Main: persist_copy(backup, main, SegmentSize)
51         if state == SS_Backup: persist_copy(main, backup, SegmentSize)
```

Figure 6: The checkpoint-recovery protocol.

Dirty block/segment recording: The instrumented code marks memory blocks dirty before they are modified (Line 24). At the end of each epoch, we *do not* clear the dirty block bitmap. Instead, we find different blocks between the main segment and its paired backup segment during the copy-on-write of the next epoch (Line 11). Dirty bits of these blocks are cleared after a successful copy-on-write (Line 15). We also use the *dirty segment bitmap* to identify whether the main segment has been modified during the *current* epoch (Line 16). It will be cleared at the end of the epoch (Line 42).

Copy-on-write: Copy-on-write is triggered before the first time to modify data in a main segment, i.e., the segment is clean in the current epoch (Line 22). It normally performs block-based data copy (Lines 9–12), updates the segment state (Line 14), and marks the segment dirty (Line 16). To support multi-threading, the per-segment locks serialize concurrent copy-on-writes on the same segment M_i (Lines 2 and 17). They also guarantee that the copy-on-write for M_i has been completed after unlocking (§3.4.4).

3.4.2 Checkpoint Protocol. Figure 6 also shows the checkpoint protocol. (1) Find dirty blocks from the main region and persist them in NVM (Lines 29–34). To reduce the persistence overhead, we choose either `clwb` or `wbinvd` instructions by comparing the total size of dirty blocks and a threshold (the last level cache size, 32MB in our platform). (2) Segment states of all dirty segments are changed to `SS_Main` atomically (Lines 36–41): `libcrpm` firstly updates the inactive segment state array and makes it durable (Lines 37–40); `committed_epoch` is then atomically updated, which swaps inactive and active segment state arrays (Line 41). If the number of dirty segments is less than a threshold, copy-on-write of *all dirty segments* will be immediately executed during the checkpoint period. This further reduces the use of `sfence` instructions.

3.4.3 Recovery Protocol. Before restarting, segments from the backup region are copied to the corresponding main segments, making the working state identical to the checkpoint state (Lines 51 of Figure 6). If data in the main segment is also the checkpoint state, its paired backup segment (if existed) needs to be updated for ensuring the correctness of block-based data copy (Lines 50).

3.4.4 Correctness. `libcrpm` ensures checkpoint state (including the segment state array in the metadata and the used segments) equivalent to the working state of the latest completed epoch, even if the execution is interrupted during checkpointing. (1) The checkpoint protocol atomically updates the checkpoint state by increasing the `committed_epoch` (Line 41). Before this operation, the previous checkpoint state exists, and it will be used for recovery if the system fails. Contents of the previous checkpoint state do not change at this time. By successfully updating the `committed_epoch`, the new checkpoint state is available for recovery. (2) Segment-level copy-on-write separates the working and checkpoint versions of a segment. Both copy-on-write and subsequent changes to the working version do not corrupt the checkpoint state. Concurrent copy-on-writes on the same segment are also handled correctly. For example, assume threads A and B write the same segment. Thread A locks first and performs copy-on-write, while thread B will block until thread A completes copy-on-write and releases the lock.

3.5 Buffered Mode

`libcrpm` can also run in a *buffered mode* that applications manipulate program state in DRAM for better performance. During checkpointing, `libcrpm` updates either main or backup segments depending on the current epoch number e. If e is even, dirty blocks from the i-th in-DRAM segment (modified during epochs $e-1$ or e) are replicated to the main segment M_i with the same index. Otherwise, these blocks are copied to the paired backup segment of M_i. The checkpoint protocol also needs to modify the segment array and the committed epoch atomically (Lines 36–41 of Figure 6).

3.6 Support for MPI Applications

`libcrpm` supports *coordinated checkpoints* in MPI programs so that multiple containers can update their checkpoints atomically. During the checkpoint period of epoch e, each process commits its new checkpoint state individually, then all processes are synchronized by `MPI_Barrier()` calls. This ensures that before `MPI_Barrier()` returns, each container keeps *both* checkpoint states of epochs e and $e-1$. During recovery, each process obtains its committed epoch number e_i from metadata, and determines the minimum one $e_{\min}$ among all processes. The checkpoint state of epoch $e_{\min}$ will be used for further recovery (c.f. §3.4.3).

4 Implementation

`libcrpm` consists of a Clang/LLVM v10.0.0-compatible [5] pass (1481 lines of code) as a compiler plugin, and a runtime library written by C++ (7506 lines of code). The AVX-512 instruction set [4] accelerates non-temporal memory copying. We implement a memory allocator for managing program state objects (§3.2). The allocator metadata is also considered as program states to be saved, so we instrument related code when building `libcrpm`.

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 DCPMM. 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) **Dalí** [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 then 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 $\alpha = 0.99$. The execution period of each epoch is $128ms$.

Result: Figure 7 shows the throughput under different workloads. Compared with NVM-NP, `libcrpm`-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 `libcrpm`-Default is up to 7.23× and 7.08× higher than `mprotect` and soft-dirty bit respectively, because `libcrpm` reduces both memory tracing overhead and the checkpoint size. `libcrpm`-Default also has up to 1.47× and 1.38× 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 `libcrpm`-Default is 1.80×/2.72× higher than Dalí in the insert-only/balanced workloads respectively. For the read-only workload, as there is nothing to be checkpointed, `libcrpm`-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[3]), 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 `libcrpm`-Buffered are evaluated. We run each application with eight processes in a single machine, and checkpoints are generated every five iterations.

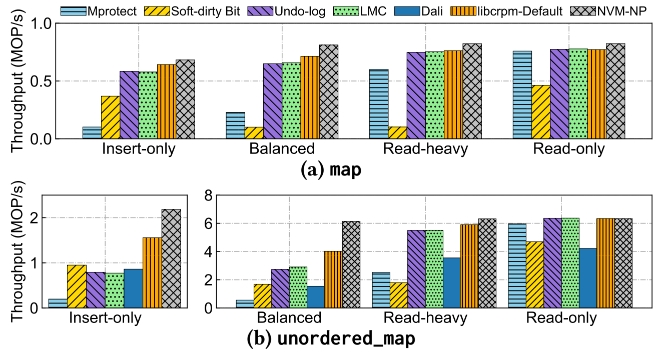

Figure 7: Throughput of the persistent `map` and `unordered_map` with a single thread. The checkpoint interval is $128ms$.

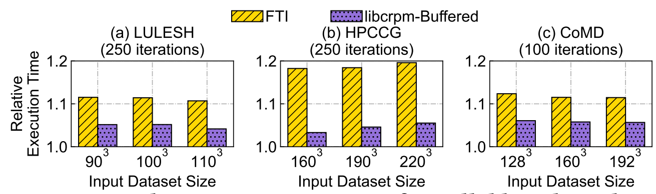

Figure 8: Relative execution time of parallel benchmarks.

Table 1: Detailed analysis for persistent `unordered_map`.

(a) Average checkpoint size in bytes per operation.

	Insert-only	Balanced	Read-heavy
`mprotect`	3,190	987	117
Soft-dirty bit	1,303	872	846
`libcrpm`-Default	269	56	7

(b) Number of `sfence` instructions issued per epoch.

	Insert-only	Balanced	Read-heavy
Undo-log	218,409	173,916	40,591
LMC	222,702	184,574	40,584
`libcrpm`-Default	333	281	8

Result: Figure 8 reports the relative execution time of FTI and `libcrpm`-Buffered (i.e., execution time without checkpointing is normalized to 1.0 for each workload). For LULESH, `libcrpm`-Buffered supports fault tolerance at the cost of 5.16% of extra execution time ($10.25s$) if the input dataset size is 90^3, while FTI is 11.53% ($22.89s$)[4]. The checkpoint overhead of `libcrpm`-Buffered is only 44.78% of FTI. For both HPCCG and CoMD, `libcrpm`-Buffered reduces the checkpoint overhead by 49.83% ~ 81.85%, compared to FTI.

5.3 Effectiveness of Design Choices

This section shows how `libcrpm` 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%

[3] X/Y means X lines of code (LOC) to be added/changed to support the checkpoint-recovery semantic, while the original version has Y LOC.

[4] FTI with hash-based incremental checkpointing takes $28.46s$ extra execution time, because hash computation dominates the checkpoint overhead.

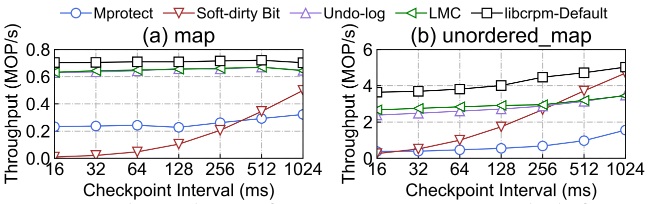

Figure 9: Throughput of `map/unordered_map` with different checkpoint intervals (under the balanced workload).

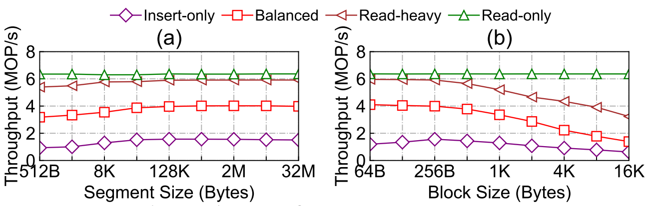

Figure 10: Throughput of `unordered_map` using `libcrpm`-Default with various segment and block sizes.

compared with `mprotect`-based system under the insert-only, balanced, and read-heavy workloads respectively. This is a key factor to mitigate the checkpoint overhead. For soft-dirty bit, modifications to a single page may trigger multiple pages to be marked dirty, which incurs high checkpoint size under the read-heavy workload.

Memory fences. `libcrpm` also reduces the `sfence` [11] instructions by using segment-level copy-on-write (Table 1b). Compared to LMC, `libcrpm`-Default reduces 99.85% and 99.84% of `sfence` instructions under the insert-only and balanced workloads respectively.

5.4 Analysis of Parameters

Checkpoint frequency. We tweak the execution period of an epoch and measure the throughput of both `map` and `unordered_map` (§5.2.1). The result is shown in Figure 9. At higher frequencies, soft-dirty bit's performance is worse than `mprotect`, as the checkpoint time is longer than normal execution. The throughput of undo-log and LMC is insensitive to the checkpoint frequency because most of the work is executed during the execution period. The throughput of `libcrpm`-Default is not severely hurt when the checkpoint interval is shorter than 128*ms*. It also outperforms other systems at various checkpoint frequencies. This indicates that `libcrpm` can reduce the checkpoint interval without harming the performance.

Segment and block sizes. To test the effects of different segment and block sizes of `libcrpm`-Default, we evaluate `unordered_map`. The checkpoint interval is 128*ms*. Figure 10a shows the throughput with different segment sizes (512B to 32MB), while the block size is 256B. When the segment size is small ($\leq$ 32KB), the throughput of the balanced workload drops. Such a penalty is caused by the increased size of the segment state array (§3.3). It takes a longer time to update the segment states atomically during the checkpoint period. More memory fence instructions are also desired.

Figure 10b shows the throughput with block sizes ranging from 64B to 16KB, while the segment size is 2MB. A smaller block usually improves the performance by reducing the checkpoint size. E.g., the checkpoint size using 256-byte blocks is 13.79% of 4096-byte blocks and 1.38× of 64-byte blocks under the balanced workload. However, small blocks incur overhead from manipulating dirty bitmaps. For balanced & read-heavy workloads, the maximal throughput is reached when using 256-byte blocks (1.81× higher than 4KB).

5.5 Recovery Time

We measure the recovery time by killing and restarting LULESH (§5.2.2) processes. The recovery time is proportional to the size of the program state: 288*ms* if the input dataset size is 90^3, and 515*ms* if the input dataset size is 110^3. During recovery, `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).

5.6 Storage Cost

For LULESH with `libcrpm`-Buffered (§5.2.2), the size of checkpoint states is 258MB per process if the input dataset size is 90^3. It is 1.35× larger than FTI, because checkpoint states of `libcrpm` are not serialized. The checkpoint size of `libcrpm` is 187MB per epoch. `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 `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 `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/ECP-copa/CoMD.
[2] [n.d.]. HPCCG. https://github.com/Mantevo/HPCCG.
[3] [n.d.]. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html.
[4] [n.d.]. Intel 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.
[5] [n.d.]. The LLVM Compiler Infrastructure. https://llvm.org/.
[6] [n.d.]. Soft-dirty PTEs. https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt.
[7] Saad Ahmed et al. 2020. Fast and Energy-efficient State Checkpointing for Intermittent Computing. *TECS* 19, 6 (2020), 1–27.
[8] Leonardo Bautista-Gomez et al. 2011. FTI: High Performance Fault Tolerance Interface for Hybrid Systems. In *SC*.
[9] Jungsik Choi et al. 2020. Libnvmmio: Reconstructing Software IO Path with Failure-Atomic Memory-Mapped Interface. In *2020 USENIX ATC*.
[10] Xiangyu Dong et al. 2011. Hybrid Checkpointing Using Emerging Nonvolatile Memories for Future Exascale Systems. *TACO* 8, 2, Article 6 (June 2011), 29 pages.
[11] Swapnil Haria et al. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *ASPLOS*.
[12] Sudarsun Kannan et al. 2013. Optimizing checkpoints using nvm as virtual memory. In *IPDPS*. IEEE.
[13] Ian Karlin et al. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973. Lawrence Livermore National Laboratory. 1–9 pages.
[14] Michael R Lyu. 1995. *Software fault tolerance*. John Wiley & Sons, Inc.
[15] Leonardo Marmol et al. 2018. LibPM: Simplifying Application Usage of Persistent Memory. *TOS* 14, 4, Article 34 (Dec. 2018), 18 pages.
[16] Faisal Nawab et al. 2017. Dalí: A periodically persistent hash map. In *DISC*.
[17] Dirk Vogt et al. 2015. Lightweight memory checkpointing. In *DSN*. IEEE, 474–484.
[18] Song Wu et al. 2019. Dual-Page Checkpointing: An Architectural Approach to Efficient Data Persistence for In-Memory Applications. *TACO* 15, 4, Article 57 (Jan. 2019), 27 pages.
[19] Chuck Chengyan Zhao et al. 2012. Compiler support for fine-grain software-only checkpointing. In *CC*. Springer, 200–219.