POSTER: Quiescent and Versioned Shadow Copies for NVM

Zhenwei Wu*†, Kai Lu*, Wenzhe Zhang*, Andrew Nisbet†, Mikel Luján †
* National University of Defense Technology, † University of Manchester

***Abstract*—QuiescentNVM is a user-space runtime providing transparent failure-consistency guarantees for lock-based parallel programs executing on hybrid combinations of traditional DRAM and byte-addressable non-volatile memory (NVM) technologies. A dual-versioning mechanism performs in-place persistent writes over one copy, and consistent fallback guarantees are provided by the other copy. Thus, the two writes to NVM present in logging-based solutions (such as for durable memory transactions) are reduced to a single write. Further, we avoid the need to rewrite legacy applications to exploit durable transactions. Our system relies on its dual-copy framework operation that safely persists data during *global quiescent states*, where no thread must hold a lock on persistent data.**

For applications with low lock-contention, global lock-free quiescent states will occur sufficiently frequently, and we deliver better performance and lower wear to NVM than current systems. We do not cover high-lock contention scenarios while enforcing quiescent states to occur.

***Keywords*-non-volatile memory, persistent data, lock-based programs**

I. Introduction

Durable memory transactions are a programming abstraction for enforcing crash-consistency, where all updates in a transaction are atomically and persistently committed to Non-Volatile Memory (NVM) with respect to a system failure. Typically, durable transaction implementations are based on logging [1], [2], that results in additional write accesses to NVM to build log entry records of each update to persistent memory [3]. The twice-write overhead means that logging mechanisms become even more undesirable when considering the limited write endurance of NVM [4]. We present next an overview of the design of QuiescentNVM.

QuiescentNVM enforces crash-consistent persistent semantics for lock-based programs via the inference of program-defined *Failure-Atomic SEctions* (FASEs) from conventional critical sections. The FOUR main principal aspects of QuiescentNVM design concerning its dual-copy framework, memory access write monitoring, quiescent lock-free states, and failure-atomic version switching are respectively described in sections I-A to I-D.

A. A Dual-Copy Framework

A user program acquires persistent memory from the `nvmalloc` interface. Two copies of each persistent page are maintained in NVM (a 2x NVM storage allocation overhead), and one shadow copy in DRAM. User applications only interact directly with the DRAM-side copy, and our runtime persists DRAM shadow page copy updates to the corresponding durable NVM region whenever a *quiescent lock-free state* occurs. Each time we move a persistent page from DRAM-side to NVM-side, only one persistent copy, referred as the *working copy*, will be modified. The other persistent page will be regarded as the *consistent copy*.

We perform in-place updates directly to the *working copy* and leverage the *consistent copy* as a fallback to ensure failure-atomicity. We do not use undo/redo log entries in our system. Roles between the *working copy* and the *consistent copy* are dynamically switched by QuiescentNVM's version switching policy. NVM version switching is fully transparent to unmodified applications.

B. Transparent Memory Access Monitoring

The set of modified pages is determined using techniques based on i), compiler instrumentation of every write hashed to page boundaries, or ii), memory write-protection and page fault handling. As in DTHREADS [5], a page-diff of the DRAM-side and the *working copy* determines the accumulated updates to a page. This is expected to achieve better wear-levelling in comparison to logging-based solutions, as only accumulated changes need to be applied, rather than logging and applying every store instruction performed. Compiler instrumentation and page-level protection schemes are both provided because excessive `TLB misses` may be triggered by page protection faults in some applications leading to lower performance.

C. Quiescent Lock-Free States

Lock-based systems must decide when to persist updates to shared state in a consistent manner. Complex dependencies may exist between nested lock acquire/release critical sections, chained acquisition of locks, and wait/notify condition signalling.

NVthreads [6] and DTHREADS [5] manage such dependencies by performing sequential merges of updates in a synchronized order that matches program execution. Isolated thread execution is enforced where conventional threads are converted into child processes with separate address spaces, and key synchronization operations (thread creation, joining, lock acquire/release, etc.) are intercepted at runtime to globally merge and persist any modifications made by separate processes to shared storage.

We do not use isolated execution, and keep all threads within a single shared process address space, thus benefiting from lower context switch overheads. Note that in our

```
function PTHREAD_MUTEX_LOCK_HOOK(mutex)
    while True do
        PTHREAD_MUTEX_LOCK(global_lock)
        ret ← PTHREAD_MUTEX_TRYLOCK(mutex)
        if ret == 0 then
            $N_{locks} \leftarrow N_{locks}$ + 1
            PTHREAD_MUTEX_UNLOCK(global_lock)
            break
        end if
        PTHREAD_MUTEX_UNLOCK(global_lock)
    end while
end function

function PTHREAD_MUTEX_UNLOCK_HOOK(mutex)
    PTHREAD_MUTEX_LOCK(global_lock)
    $N_{locks} \leftarrow N_{locks}$ - 1
    if $N_{locks}$ == 0 then
        PERSIST()
    end if
    PTHREAD_MUTEX_UNLOCK(mutex)
    PTHREAD_MUTEX_UNLOCK(global_lock)
end function
```

Figure 1. Quiescent lock-free state detection

system we only persist updates when a *globally quiescent lock-free state* exists where no application threads hold a lock. Thus, no threads are then eligible to issue writes to persistent memory in such a state. Our runtime intercepts lock acquisition/release where a global lock is used to safely observe, and to maintain a quiescent state, whilst a `PERSIST` operation is used to perform failure-atomic version switching. Once the captured consistent states have been made durable by the `PERSIST` operations, then the global lock is released and application threads can acquire locks to further modify persistent data. Figure 1 outlines the operation of our runtime in this regard.

D. Failure-Atomic Version Switching

Figure 2 describes the algorithm for failure-atomic version switching using `clflush` and `sfence` x86 instructions. To achieve version switching, we maintain i), a *global version number* G_{seq} that is incremented at the end of a procedure that we use to `PERSIST` data to NVM, and ii), each persistent NVM copy of a page is associated with a local version number (*seq*) that is used to distinguish the *working copy* and the *consistent copy* of a page in NVM. Specifically, during the `PERSIST` procedure, the persistent NVM page copy with smaller *seq* in a dual-version pair becomes the *working copy*.

In summary, updating the *seq* of a *working copy* is represented as a version-switch local to the dual-version shadow pair, which cannot be visible to the post-crash stage unless the `PERSIST` procedure completes normally.

```
input: $S$ (The set of modified pages.)
function PERSIST($S$)
    for each $p \in S$ do
        $w \leftarrow$ persistent copy with smaller seq
        $w_{seq} \leftarrow G_{seq}$ + 1
        clflush $w_{seq}$
        sfence
        clflush DIFF_BYTES($p$, $w$)
    end for
    sfence
    $G_{seq} \leftarrow G_{seq}$ + 1
    clflush $G_{seq}$
    sfence
end function
```

Figure 2. Failure-atomic version switching.

Acknowledgments

Partially supported by the National Science Foundation China 61902405, National High-level Personnel for Defense Technology Program 2017-JCJQ-ZQ-013, the HUNAN Province Science Foundation 2017RS3045, and EU Horizon 2020 ACTiCLOUD 732366. Mikel is supported by an Arm/RAEng Research Chair award.

References

[1] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proc. of ASPLOS 16*, 2011, pp. 91–104. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950379

[2] N. Cohen, M. Friedman, and J. R. Larus, "Efficient logging in non-volatile memory by exploiting coherency protocols," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 67:1–67:24, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133891

[3] W. Zhang, K. Lu, M. Luján, X. Wang, and X. Zhou, "Write-combined logging: An optimized logging for consistency in NVRAM," *Sci. Program.*, vol. 2015, pp. 25:25–25:25, Jan. 2015. [Online]. Available: https://doi.org/10.1155/2015/398369

[4] A. Awad, S. Blagodurov, and Y. Solihin, "Write-aware management of NVM-based memory extensions," in *Proc. of 2016 Intl. Conf. Supercomputing*, 2016, pp. 9:1–9:12. [Online]. Available: http://doi.acm.org/10.1145/2925426.2926284

[5] T. Liu, C. Curtsinger, and E. D. Berger, "DTHREADS: efficient deterministic multithreading," in *Proc. of 23rd ACM Symp. on Operating Systems Principles*, 2011, pp. 327–336.

[6] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "NVthreads: Practical persistence for multi-threaded applications," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17, 2017, pp. 468–482. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064204