ADDING PERSISTENCE TO MAIN MEMORY PROGRAMMING

A Thesis Proposal
Presented to
The Academic Faculty

by

Pradeep Fernando

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
October 2019

ADDING PERSISTENCE TO MAIN MEMORY PROGRAMMING

Approved by:

Professor Ada Gavrilovska, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Tushar Krishna
Electrical and Computer Engineering
Georgia Institute of Technology

Professor UmaKishore Ramachandran
School of Computer Science
Georgia Institute of Technology

Dr. Amitabha Roy
Google

Professor Joy Arulraj
School of Computer Science
Georgia Institute of Technology

Date Approved: xx/xx/xxxx

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Unlocking the true potential of the new non-volatile memories (NVMs) requires eliminating traditional persistent I/O abstractions altogether, by introducing persistent semantics directly into main memory programming. Such a programming model elevates failure atomicity to a first-class application property in addition to in-memory data layout, concurrency-control, and fault tolerance, and therefore requires redesign of programming abstractions for both program correctness and maximum performance gains. To address these challenges, this thesis proposes a set of system software designs that integrate persistence with main memory programming, and makes the following contributions.

First, this thesis proposes an NVM-aware I/O runtime, NVStream, that supports fast durable streaming I/O. NVStream uses a memory-friendly I/O API that plugs into existing I/O data movement points of an application to accelerate persistent data writes. NVStream carefully designs its persistent data storage layout and crash-consistent semantics to match both application and NVM characteristics. Specifically we use a log-structured NVM storage engine with append only failure-atomic semantics to support streaming I/O produced during HPC simulations. Furthermore, the thesis acknowledges the NVM bandwidth bottlenecks during parallel HPC I/O writes and proposes a novel data movement design – PHX. PHX uses alternative network data movement paths available in data-centers to ease up the bandwidth pressure on the NVM memory interconnects, all while maintaining the correctness of the persistent data.

Next, the thesis explores the challenges and opportunities of using NVM for true main memory persistent programming – a single data domain for both runtime and persistent application state. Such a programming model includes maintaining ACID properties during each and every update to applications persistent structures. ACID-qualified persistent programming for multi-threaded applications is hard, as the programmer has to reason about both crash-consistency and synchronization – crash-sync – semantics for programming correctness. The thesis introduces NVMTSX, that extends the popular hardware transactional memory (HTM) primitive with durability semantics, and offers a hardware

accelerated crash-sync primitive that supports both low overhead synchronization and correct crash-consistency.

Finally, the application state stored on node-local persistent memory is still vulnerable to catastrophic node failures. The thesis proposes a replicated persistent memory runtime, Blizzard, that supports truly fault tolerant, concurrent and persistent data-structure programming. Blizzard carefully integrates userspace networking with byte addressable NVM for a fast, persistent memory replication runtime. Further, the design also supports a replication aware crash-sync protocol that supports consistent and concurrent updates on persistent data-structures.

CHAPTER I

INTRODUCTION

Emerging, byte-addressable non-volatile memory hardware technologies such as phase-change memory and STT-RAM have made significant progress in the last decade. Recently, the first large-capacity commercial products believed to be based on phase-change memory [**izraelevitz2019basic**, **izraelevitz2019guide**] were released, targeting the server market. These new memories support byte-addressability via CPU load/store instructions, much like main memory (DRAM), but differ from the latter, as they hold the data state across node-restarts, similar to a persistent disk device. While the exact numbers depend on the NVM hardware technology itself, the NVMs in general, support high capacity memory modules because of their high density, have high access latencies, e.g., writes are up to 4× slower than DRAM, and have limited device bandwidth, e.g., up to 8× lower than the DRAM. However, poor capacity scaling of DRAM main memory, combined with modern application's demand for fast, volatile and persistent storage, drive the integration of the NVMs in compute platforms in datacenters [**exadata:oracle**, **google:optane**, **pratap:pirl19**] and in HPC exascale systems [**aurora:optane**]. In these systems, NVMs can be integrated in two main ways. In the first one, NVM is placed "behind" DRAM, in a configuration where the system DRAM represents a hardware-managed cache for a much larger memory capacity provided by the NVM device; this configuration does not exploit the persistent properties of NVM. In the second, NVM is placed "side-by-side" with DRAM, as illustrated in Figure 1. In this, more interesting, configuration, both the additional capacity and the persistence provided by the NVM device are directly exposed to applications and software stacks. This work is primarily concerned with systems incorporating DRAM and NVM in this "side-by-side" configuration, referred to as App Direct mode in the context of the Intel Optane DC Persistent Memory [**izraelevitz2019basic**].

Augmenting the memory hierarchy with NVMs and leveraging the full benefits that

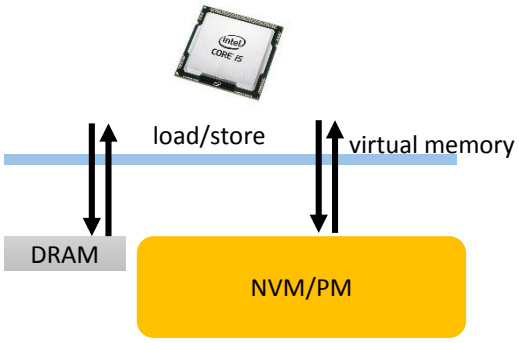

Figure 1: DRAM and NVM memory configuration

NVMs offer, presents both opportunities and challenges to system software designers. This is because the existing persistent system software stacks such as file systems and database management systems (DBMS) are built and optimized upon fundamentally different hardware abstraction – persistent block devices. On the other end of the spectrum, we have volatile memory programming abstractions – data-structures – such as hashmaps, B-trees, etc., that permit direct manipulation of the memory words using load/store instructions. Main memory programming semantics on NVM are shown to yield the maximum performance benefits [**Volos:Mneymosyne**, **kannan2016pvm**, **Coburn:NVheaps**] since it minimizes the imposed software instructions in the critical path of the NVM accesses, however at the cost of program correctness. This is because, unlike DRAM memory, NVM resident data survives across node restarts. An unplanned node restart is likely to terminate an ongoing sequence of NVM updates in the midway through, resulting in an inconsistent program state. Furthermore, most main memory programming platforms integrate transient CPU caches for performance, that in turn delay and/or reorder memory updates back to the memory device, making it impossible to guarantee the correctness of the persistent data in the event of a crash. Solving this involves integrating crash-consistent semantics in to application programs as a first class citizen.

The system software research community has made significant progress in addressing both of these challenges, by redesigning NVM optimized file-systems [**pmfs**, **novafs**], DBMSs [**wbl**] and programming libraries [**mnemosyne**] and data structures [**LeeEtAl19-Recipe**]. Still a number of important questions remain open. In the context of NVM and modern

applications, Are the existing I/O runtimes and abstractions good enough for fast,durable data movements?. How to support fast, main memory like persistent programming abstractions without compromising the program correctness? How such new abstractions interplay with concurrency control of the applications? Further, how to support truly fault tolerant persistent memory programming? We next discuss these questions in detail.

1.1 Statement of problems

- **Memory speed persistent I/O:** Persistent disk storage hardware, accessed using the I/O subsystem is incredibly slow compared to volatile main memory. Furthermore, applications often use software I/O intermediaries like file-systems and DBMSs to manage their persistent state. These system software stacks are very useful with disk storage as they accelerate data movements using buffering, manage concurrent accesses to data, provide consistent data semantics, etc. Most of the existing software falls in to this class of applications and thus it is important to accelerate their persistent data usage model with new NVM hardware. The recent NVM aware persistent I/O proposals [**pmfs**, **novafs**] address the above by redesigning system software internals while keeping the abstraction unchanged. However, the generic system design and the abstractions themselves are likely to limit further potential performance gains. One reason is that these traditional file system abstractions imply functionality (e.g., in terms of metadata management) or semantics (in terms of visibility of file system updates) that form significant portions of the I/O costs when used with fast NVM devices; yet, they are not even necessary for important classes of I/O behaviors (such as for checkpoint generation and for streaming I/O in application workflows).

  This thesis shows that important types of streaming I/O, associated with long running applications and application workflows continuously producing and processing data outputs, can be easily adapted to memory-based I/O abstractions. Such abstractions provide applications with the desired persistence and crash-consistency properties, while affording the full benefits of the NVM performance benefits over traditional storage devices. Furthermore, they open up several optimization opportunities for

hardware- or application-specific optimization, that offer crucial benefits in terms of eliminating redundant operations or, more importantly, avoiding NVM-specific bottlenecks.

- **Reason about synchronization and crash-consistency in persistent programming:** Multi-threaded shared memory programming uses concurrency-control primitives such as locking and transactional-memory (software/hardware) for program correctness. They provide atomicity, consistency and isolation (ACI) guarantees for multi-threaded programs. Crash-consistent persistent memory updates in the form of storage transactions, independently, also provide the same guarantees, plus durability (ACID). One way to simplify the development of NVM applications is to use a single abstraction that provides guarantees on both crash consistency and correct synchronization. To this end, prior work has proposed numerous systems to simultaneously guarantee both these properties [**mnemosyne**, **nvheaps**, **phtm**, **phytm**, **Wang:cal:2015**, **Joshi:isca:2018**, **Doshi:hpca:2016**]. While these systems provide transactions with the desirable ACID properties that permit their use for both `crash-consistency` and `synchronization`, they do so in a myriad of ways, some implemented completely in software while some rely on hardware support, some use undo logging, while others use redo logging. So, developers are faced with a bewildering array of choices, with varied performance characteristics that change with applications and the system used. For a platform with given hardware features and consistency and synchronization requirements, is it possible to streamline the design space and quickly arrive at a correct and performant implementation of a transactional system?

- **Truly fault tolerant persistent memory:** Commodity servers which integrate NVM provide data reliability by means of hardware (erasure-codes) and software (RAID) mechanisms, but they do not tolerate full node failures. A truly fault tolerant persistent memory programming requires some form of data redundancy across nodes. However, it is not straight forward to add fault-tolerant semantics to persistent data structures while supporting concurrent, crash-consistent updates. This

is because, each of these semantics have their unique and often conflicting system software trade-offs between each other. For an example, replication stacks are often designed with serial log commits for correctness while persistent data-structures use concurrent updates for maximum throughput.

Summarizing, simply using memory-based interfaces and programming models to maximize the performance of NVM-based applications and software stacks in not trivial, and presents a number of challenges that must be addressed, and tradeoffs that must be fully evaluated in order to be leveraged.

1.2 Thesis statement

To unlock the true potential of the new non-volatile memories, and achieve both program correctness and maximum performance gains, requires eliminating persistent I/O abstractions altogether, by introducing persistent semantics directly into main memory programming, making it necessary to elevate failure atomicity to a first-class property in addition to data-structures, concurrency-control and fault tolerance during both application and programming abstraction design.

1.3 Contributions

- **Memory speed persistent I/O:** This thesis proposes a memory centric object-based abstraction for persistent I/O. The supporting system – NVStream – comprises an I/O library and runtime system, and its design is specialized for data movement in analytics workflows and checkpoint I/O, such as what is used in HPC. The design of NVStream combines a streaming, versioned object-store, with efficient log-based memory management, and hardware-accelerated persistence for consumer-producer patterns. The memory friendly persistent I/O API supports data movement and persist optimizations that uses both hardware and domain specific application characteristics.

- **NVM bandwidth aware persistent I/O:** Removing the software overheads from the NVM access paths, exposes new bottlenecks. For memory-based checkpoint I/O

in HPC applications, where many threads concurrently issue coordinated checkpoint operations, the limited NVM bandwidth poses a significant challenge in realizing the expected benefits from using NVMs.

We propose an NVM bandwidth aware persistent I/O library (PHX) for HPC checkpoint I/O. PHX deals with the limited NVM bandwidth through simultaneous use of NVM and local/ peer nodes' DRAM devices, thus increasing the effective data movement bandwidth. PHX's memory-centric object interface and NVM-bandwidth-aware design lead to reduction in the time length of I/O operations in the critical path, associated with the slow NVM device. To continue guaranteeing adequate reliability and persistence, DRAM-resident object state is replicated across peer nodes' memory, which is accessible through high-bandwidth interconnects. The use of a memory-based I/O both exposes the problem (since it would not be an issue if applications used "longer" block-device based I/O paths) and enables the solution (by making it possible to realize a low-cost replication so that the additional data movement introduced by PHX is outweight by substantial gains in application performance and system efficiency).

- **Building persistent memory systems with crash-sync-safety:** We characterize different transaction systems to identify design space of ACID transactions. The candidates are chosen based on *crash-sync-safety* property, that guarantee 1) proper synchronization, 2) crash-consistency semantics, and 3) correct composibility of (1) and (2). This new characterization of transaction systems provides a basis to compare different implementations and to identify the right set of crash-consistency and synchronization mechanisms for particular applications and hardware platforms.

- **Fault-tolerant persistent memory:** This thesis proposes a system software solution – Blizzard – that supports highly performant and fault-tolerant (replicated) persistent memory programming. The key idea in Blizzard is to exploit direct accesses to persistent memory from both CPU and the commodity NIC to provide efficient zero copy replication of RPC calls. We build on this by providing a library of persistent

data-structures with a recipe for concurrency that works well with the replication.

CHAPTER II

BACKGROUND AND RELATED WORK

2.1 Byte-addressable persistent memory

The well-known DRAM main memory realizes binary states using capacitor stored temporary electrical energy. DRAM main memory therefore is volatile and notorious for weak capacity scaling, as the device energy consumption is proportional to memory capacity. Byte-addressable persistent memory or simply persistent memory (PM), on the other hand realizes binary state using changing the material resistance using an electrical charge. The persistent memory updates are slow and consumes more energy due to this technical difference. However, they have excellent capacity scaling, as the device does not need energy refresh cycles. More importantly, for the same reason the written data is persistent and survive across machine restarts. Table 1 compares [**raoux2008phase**] device properties of PM against DRAM main memory and SSD/HDD storage.

	DRAM	**PCM**	**SSD**	**HDD**
Capacity per CPU	100s of GBs	Terabytes	Terabytes	Terabytes
latency	60 ns	300 ns	300 µs	10 ms
bandwidth	1×	$\frac{1}{4} \times to \frac{1}{8} \times$	-	-
Addressability	Byte	Byte	Block	Block

Table 1: Comparison of PM with DRAM and other storage devices

2.2 Persistent memory programming

PM is directly accessible by CPU instructions (e.g. load/store), and direct programming of persistent state without software intermediaries such as file-systems is one of the most desirable properties of new device. However, direct programming persistent memory involves maintaining consistent persistent data state at all times using careful data-update sequences – crash-consistent update protocols.

2.2.1 Storage Transactions

Storage transaction semantics allow programmers to mark/declare regions of programs with `transaction_begin` and `transaction_end` semantics. The semantics are intuitive and in the context of PM, can be implemented efficiently. PM storage transactions are often realized using compiler instrumentation, where compiler-pass fill-in the additional data-writes and data-orderings to achieve crash-consistency. Mnemosyne [**mnemosyne**] and NV-Heaps[**nvheaps**] were the first to support PM programming with storage transactions.

2.2.2 Transactional memory

Transactional memory (TM) uses the same programming semantics, but the target use-case is synchronization. TM enclosed code regions executed as atomic regions isolated from other executions. The semantics are analogous to a concurrency control using a one global lock, however the execution is significantly different as the TM libraries track critical region accesses at a more finer granularity. TM support has been implemented in both software – software transactional memory (STM) [**tl2**, **tinystm**] and hardware – hardware transactional memory (HTM) [**herlihy1993transactional**].

2.3 HPC I/O

Parts of this dissertation research explore the role of persistent memory for memory-based HPC I/O. Long-running scientific computations, such as material combustion, fusion and climate modeling, periodically produce program outputs of the simulation state. These periodic program outputs serve multiple purposes. First, they are used as application checkpoints, which are used for recovery in the event of simulation or system failure. Second, they are being used to provide online insights into the simulation state, and are directly consumed by co-running coupled analytics programs, performing output visualization, verification, uncertainty analysis, or other data analysis tasks [**insitu**, **insitucosched**] Traditionally these applications store their persistent I/O on parallel file-systems (PFS) that are accessed over a network fabric.

CHAPTER III

COMPLETED WORK

3.1 NVStream: Streaming persistent I/O

Recent research on storage stacks has explored various design points for implementing NVRAM aware system software stacks. On one end of the spectrum, we have stacks with file system APIs that treat the NVRAM as a block device [**aerie**, **moneta**, **scmfs**, **pmfs**, **bpfs**]. While these optimized file-system implementations remove redundant system components such as the page-cache from the I/O path and fit well into the existing persistent storage I/O model, they still introduce unnecessary systems software overheads. This is because the use of the file system stack leads to long I/O data movement paths due to multiple software abstraction layers (e.g., in the VFS layer), and requires switching back and forth from user-space to the kernel during data movements. Alternatively, one can treat NVRAM as fast memory and access the device using familiar 'malloc' like APIs. Memory APIs avoid kernel interactions for the most part and expose the device's load/store interfaces directly to the userspace. However current 'malloc' APIs (e.g., `malloc, free`), are not sufficient for NVRAM data interactions as they lack the awareness for device persistence.

Based on these observations, we design and implement `NVStream`, a user-level transport for workflow coupling and high-performance data streaming via NVRAM. `NVStream` provides benefits by leveraging the memory-based nature of NVRAM, the streaming semantics and temporal durability requirements of scientific workflow systems, and the new architectural capabilities in modern processor architectures. Its design combines a streaming, versioned object-store, with efficient log-based memory management, and hardware-accelerated persistence for consumer-producer patterns. Additional optimizations in the data movement path for applications exhibiting temporal locality in their data access behavior are realized through use of delta encoding [**treadmarks**].

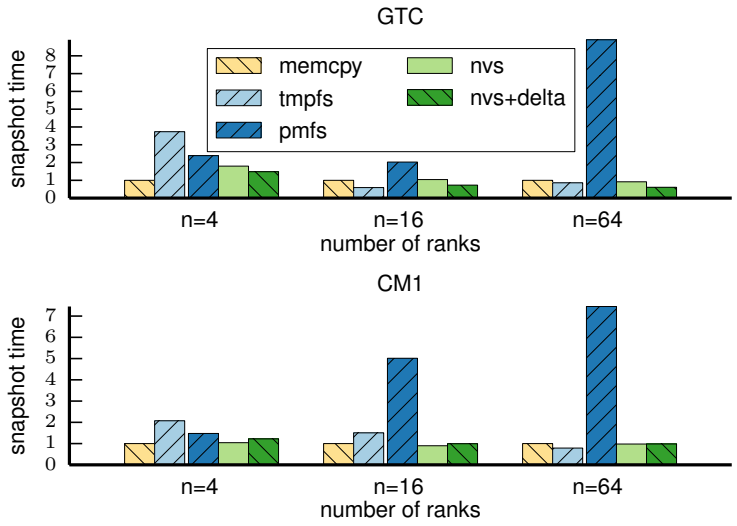

Figure 2: GTC and CM1 snapshot time for each of the I/O techniques. We normalize the times to best case data movement time – memcpy. We run each benchmark with increasing number of MPI ranks.

3.1.1 Evaluation highlights

We evaluate `NVStream` with two HPC applications namely, GTC and CM1. We configure the GTC test setup such that each rank outputs $\sim 210MB$ of I/O data per iteration. We run the benchmark for increasing number of MPI ranks(N value) while keeping the per rank output data size more or less the same (weak scaling). We report in figure 2 the average data I/O time for each I/O mechanism, normalized to `memcpy` I/O time. `NVStream` I/O is 24% faster than `pmfs` when 'N=4', whereas `pmfs` I/O performance drops drastically with the number of MPI ranks. As a result at 'N=64' `NVStream` I/O is 10× faster than `pmfs`. delta-encoded `NVStream` (nvs+delta) moves $\sim$ 50% less I/O data compared to other I/O techniques and is 33% faster compared to `NVStream` I/O and `memcpy`.

Similarly, we configure the CM1 application to output $\sim$ 45MB of I/O data per-iteration/per-rank. Unlike GTC, CM1 is a compute heavy HPC kernel, thus increasing the I/O data size leading to more time spent in the compute portions of the benchmark. We report in figure 2 the average data I/O time for each I/O mechanism normalized to `memcpy` I/O time. `NVStream` I/O is 7× faster than `pmfs` and delta-encoded `NVStream` I/O performs the same. It is important to note that the performance of `NVStream` I/O is as fast

as `memcpy` based I/O in most occasions, which confirms the lightweight crash-consistency semantics of `NVStream`.

3.2 PHX: NVM bandwidth aware persistent I/O

Non-volatile memory (NVRAM) provides persistent storage at close to memory speeds, with good capacity scaling, leading to opportunities to accelerate I/O in exascale machines. For use of NVRAM in checkpoint/restart (C/R), prior work [**Bent:PLFS**, **Dong:3DPCM**, **moneta**] has proposed methods to incorporate newer and faster storage technologies like Flash/SSD and NVRAMs as stable storage. These studies have shown that Flash-based storage is not sufficient to address the increase in per-node core count and the data-to-core ratio. Recent work on C/R uses byte-addressable NVRAM as stable storage that can significantly improve storage performance compared to block-based SSDs [**levy2014using**, **Dong:HybridCkpt**]. Although NVRAMs are expected to deliver 100x faster access compared to SSDs, current NVRAM technologies are limited by (i) relatively low per-node socket bandwidth (1-2 GB/sec die bandwidth compared to >10 GB/sec for DRAM), and (ii) high write latency (3-5x slower than DRAM). Concerning large volumes of checkpoint data transfer across DRAM and NVRAM (a norm in HPC applications), and high core count (64-128 cores/nodes in exascale) will severely limit the benefits of NVRAM. A common approach has been to exploit the byte-addressability of NVRAMs and treat them as a "slow memory", rather than "fast disks" to eliminate the overheads of data serialization cost mainly from converting the disk to memory format or vice versa, and avoid expensive kernel boundary crossing cost. Using a slow memory also permits fast tracking of incremental updates. Prior work [**Dong:HybridCkpt**] have discussed the benefits of using NVRAM as a memory or provided additional software-based optimizations for NVRAM bandwidth constraints [**kannan**]. However, these solutions are limited by the properties of the NVRAM technology because the checkpoint I/O time is governed by the device bandwidth.

To address this problem, we argue that the exascale software stacks require solutions that deal with the NVRAM bandwidth limitation by careful consideration of the available data movement paths across the system. The key idea explored in this paper is that, through the

use of additional interconnect bandwidth, it is possible to hide some of the NVRAM device limitations. The approach opens up new trade-offs in terms of performance, reliability, and energy use. We demonstrate that the design space creates opportunities to accelerate checkpoint/restart services and provide overall application speed-up, without sacrificing reliability or incurring energy overheads due to increased data movement operations.

3.3 Characterizing persistence and synchronization

One of the key appeals of NVM is that they allow applications to access storage directly using processor load and store instructions rather than relying on a software intermediary like the file system or a database [**pelleychen14**]. However, ensuring that data stored in NVM is always in a safe and recoverable state (i.e., the data is *crash-consistent*) is both hard and incurs performance overheads [**bpfs**, **mnemosyne**, **nvheaps**, **pelleychen14**]. To ensure crash consistency, application developers have to carefully orchestrate the movement of data from the volatile to non-volatile components in the memory hierarchy subject to recoverability constraints.

To exacerbate the challenge of ensuring crash consistency, different systems provide different guarantees on when data may be considered persistent. For example, Intel and Micron guarantee that data becomes persistent only when it reaches the memory controller of the NVM device, i.e., the persistent domain of the system includes the memory controller and the NVM devices [**intel2016pcommit**]. We refer to such systems as having *transient caches*. However, HPE's NVM [**hpnvdimm**] guarantees that the entire cache hierarchy is persistent, i.e., the persistent domain of the system includes the entire memory hierarchy. We refer to such systems as having *persistent caches*. Based on the persistent domain of the systems, developers have to tailor their applications to achieve crash consistency. The diversity of NVM applications [**whisper**] further complicates achieving crash consistency. Multi-threaded applications require developers to ensure correct synchronization on top of crash consistency.

One way to simplify the development of NVM applications is to use a single abstraction that provides guarantees on both crash consistency and correct synchronization. The

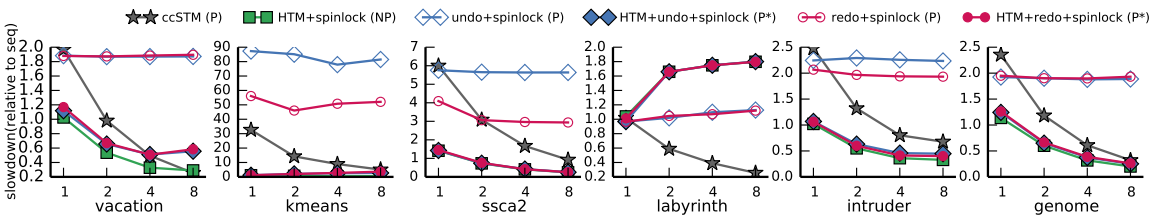

Figure 3: TSX-enabled hardware with real NVM and transient caches by number of threads (X axis). (P) crash-consistent; (NP) not crash-consistent; (P*) approximates crash-consistent solution.

transactional programming interface is particularly well suited for this approach as we have been using transactions to independently achieve crash consistency (e.g., database transactions) or proper synchronization (e.g., transactional memory systems). To this end, prior work has proposed numerous systems to simultaneously guarantee both these properties [**mnemosyne**, **nvheaps**, **phtm**, **phytm**, **Wang:cal:2015**, **Joshi:isca:2018**, **Doshi:hpca:2016**]. While these systems provide transactions with the desirable ACID properties that permit their use for both `crash-consistency` and `synchronization`, they do so in a they do so in a myriad of ways, some implemented completely in software while some rely on hardware support, some use undo logging, while others use redo logging. So, developers are faced with a bewildering array of choices, with varied performance characteristics that change with applications and the system used. For a platform with given hardware features and consistency and synchronization requirements, is it possible to streamline the design space and quickly arrive at a correct and performant implementation of a transactional system?

We answer the question by careful selection and evaluation of number of synchronization and crash-consistency primitives with each other including crash-consistent hardware and software transactional memory primitives.

3.3.1 Evaluation highlights

Figure 3 shows the scalability of our selected crash-consistent and synchronization approaches against varying number of threads, on real hardware, using Intel TSX as the hardware transactional memory. Note, how the HTM accelerated `crash-consistency` implementations outperform others by quite margin for wide array of workloads from the

stamp [**stamp**] benchmark suite. The only exception is labyrinth workload, where crash-consistent software transactional memory implementation performs better. For labyrinth workload HTM accelerated logging does not work, as the relatively large transaction working set triggers HTM capacity aborts. Please refer the attached paper for more details.

CHAPTER IV

CURRENT WORK

4.1 Need for fault-tolerant, persistent memory

Persistent memory with correct system software primitives (§3.3) support fast and durable application state maintenance using direct CPU load/stores. However, NVMs integrated in commodity servers only provide limited reliability guarantees for persistent data in the form of software RAID among NVM modules, erasure codes, etc, but fail to protect data against catastrophic node or persistent media failures (hard-failures). Fault-tolerant persistent application state that survives hard-failures is a must for highly available enterprise applications.

The design of system software for truly fault tolerant persistent memory programming is not trivial. We identify two main components in such a design: **Persistent memory programming layer** – that stores application specific persistent data, and **Replication layer** – that supports fault-tolerance by synchronizing persistent data between multiple server replicas. Integrating these two components together to form a high-throughput, fault-tolerant persistent memory programming system software stack is challenging, as the system software techniques commonly used to optimize each of these two components individually, often conflict with each other. For an example, state machine replication protocols such as Paxos and RAFT [**paxos**, **raft**] rely on *serial log-commits* to preserve consistent replica state among distributed nodes, while high throughput persistent programming involves concurrent data-updates to offset some of the crash-consistent write-ahead-logging (WAL) costs (§3.3). Naive coupling of these two components will compromise performance (e.g. turn-off concurrent updates), or worse yet, will lead to incorrect persistent data states.

This thesis presents Blizzard, a system service that supports truly fault-tolerant persistent memory programming. Blizzard takes care of availability and provides client with access to persistent state via an RPC interface. The approach is consistent with modern

microservice-based datacenter stacks. To demonstrate and evaluate Blizzard, we will provide a small (but growing) set of persistent implementations of popular data structures that will allow performant implementations of a wide array of real world applications.

The underlying innovation that makes Blizzard possible is exploiting direct access to persistent memory from both the CPU and the (commodity) NIC to provide efficient zero copy replication of RPC calls. We build on this by providing a library of persistent data structures with a recipe for concurrency that works well with replication. Concretely, Blizzard provides and arbitrary number of logical channels to named data structures held in persistent memory. Operations that do not commute go to the same channel while operations that commute go to different channels. Blizzard enforces sequential execution on the same channel.

We next describe each of our design choices and their rationale in-detail.

4.2 Blizzard: fault-tolerant, persistent data-structures

4.2.1 Interface

The core operational model in Blizzard is a client-server one: users write services that receive remote procedure calls from clients, lookup and manipulate persistent state and then return a response. In order to provide maximum flexibility to the user, we treat the actual RPC call as a binary blob.

```
1        // Client side API
2        Status MakeUpdateRPC(const string& request, string* response);
3        Status MakeReadRPC(const string& request, string* response);
4
5        // Server side API
6        class Lock() {
7         public:
8          virtual ReleaseLock() = 0;  // Override to release your lock
9        }
10       void HandleRPC(const string& request, vector<Lock>* delayed_locks,
     string* response);
11       bool Commutes(const string& requestA, const string& requestB);
```

Listing 4.1: Blizzard API.

The Listing 4.1 shows the Blizzard API available to programmers - we elide some setup details for reasons of brevity and focus on the core APIs. On the client side, the programmer can send an RPC call as a binary blob. Blizzard takes care of service discovery (primarily locating the replica leader) and sends the server the RPC call. It returns the response as

another binary blob, and returns a Status object detailing whether the RPC call could be successfully completed, or otherwise provides error codes. We distinguish between read and update RPCs as separate calls. This is because reads are not replicated but updates need to be.

The server side API encapsulated in `HandleRPC` is implemented by the programmer and executed as a callback by Blizzard. It contains the received request blob and expects a response blob to be returned after execution. For an update RPC call, Blizzard guarantees that the `HandleRPC` call is executed *after* the call has been successfully replicated on all replicas (Section 4.2.2). The `HandleRPC` call is executed in the context of a persistent memory transaction (Section 4.2.3). Any updates to persistent memory are only committed *after* `HandleRPC` returns to the Blizzard runtime.

We expect concurrent invocations of `HandleRPC` to access shared (persistent) memory data structures, and that they do so in a thread-safe manner. The Blizzard library will provide several thread-safe data structures; this set can further be extended using the PMDK [**pmdk**] library and the recipes contributed with Blizzard. In all cases, data race safety enforcement will be responsibility of the programmer, using their favorite lock implementation, which they are free to place in volatile memory. The only requirement will be that they wrap these locks in an object derived from `Lock` in the Blizzard API (see Listing 4.1) and *do not* release them when executing the RPC callback. Instead, these must be returned in the vector object provided in the API. Blizzard releases all the locks *after* updates to persistent memory have been committed (we discuss the reasons for this in Section 4.2.3).

Finally, we expect the programmer to implement a callback to help determine the commutativity of various RPC calls. A programmer declares two RPC calls as commutative if the application correctness is not affected if Blizzard interchanges their order of execution. Commutative RPC calls can be executed concurrently by Blizzard allowing better performance than simply following the sequential execution specified by the replication order.

4.2.2 Replication

Replication in Blizzard is critical to ensure in-memory data structures are truly available even when the underlying persistent memory fails or the machine goes down. We use RAFT [**raft**] to replicate a durable log of updates to all in-memory data-structures. However, log replication is a synchronous operation. The latency components of accessing durable storage and the network hops required to update replicas add to the latency of operation completion. We note that although various pieces of work have attempted to improve network overheads for replication protocols [**consensus˙box**, **dare**, **fast˙dc˙rpc**], *they have examined replication without persisting any state.* This is because, conventional Flash storage comes at a significant latency and throughput cost compared to network performance. In contrast, Blizzard is designed ground up for persistent memory that is at least an order of magnitude faster than Flash. This therefore puts us in a position to focus on the network component of a fully functional replication stack that *includes* persistence.

A key building block for Blizzard is userspace network access. RAFT (as well as most other replication protocols [**paxos**]) are already designed to work on unreliable networks. Therefore, it makes sense to jettison stream oriented reliable delivery by the kernel TCP/IP stack in exchange for significant gains in latency. We use the Data Plane Development Kit (DPDK [**dpdk**]) for fast access to the network from userspace. Although this restricts Blizzard to operate in a single Data Center environment, where the Ethernet header is sufficient for routing from source to destination, and where networks are largely reliable, we believe that this is an acceptable tradeoff given the intended use of Blizzard. We also exploit the direct addressability of persistent memory to build a high performance replication stack using two simple principles: zero copy and batching.

Copying log entries in various parts of the replication state machine – from receiving the client request to sending out copies to replicas – is expensive. This is even more so since persistent memory is still slower than volatile random access memory. The fact that persistent memory allows one data structure to simply point to another (rather than indirecting through a block address on Flash) leads to an efficient solution to this problem in Blizzard. Figure 4 shows how RAFT log entries are organized in relation to DPDK's

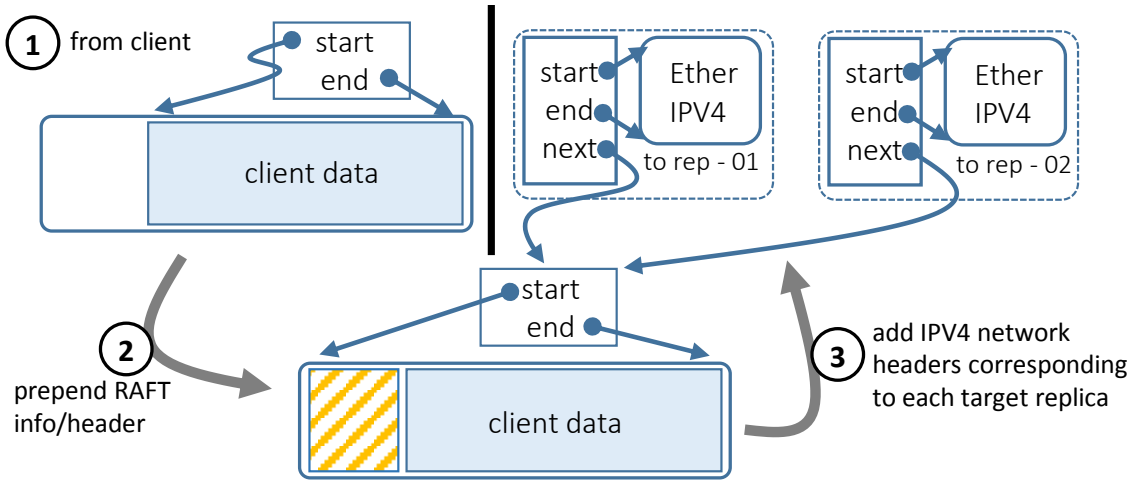

Figure 4: RAFT log entry in Blizzard

memory buffers. A DPDK memory buffer holding an incoming request (i.e., a complete Ethernet frame) is placed in an aligned block of memory together with external metadata pointing to the start and finish of the block – we remind the reader that this is all persistent memory. We keep the client request in its DPDK memory buffer for its lifetime, spanning the replication and execution phase.

To start with, the leader prepends a RAFT control block (with information such as term and index) to the buffer adjusting the external metadata, as necessary. We then append a pointer to the metadata of the memory buffer in a circular log (also in persistent memory) thereby encoding its ordering in the RAFT log.

The leader is now ready to replicate the request. We exploit the fact that DPDK allows multiple memory buffers to be chained together. To do so it simply creates an Ethernet header for *each replica* and chains the same log entry packet to each of them. It hands off all the headers to DPDK. The NIC then does the heavy lifting of assembling the Ethernet frames and sending them out. We underline that this is only possible because the logs are not in block storage and persistent memory is accessible from all connected agents, including I/O agents, in the system. In contrast, directly accessing block storage from the NIC involves serious complications [**reflex**], and this design illustrates how persistent memory can simplify the design of distributed system primitives that need persistence.

Although this design lifts most of the load off the CPU, we note that the RAFT consensus

protocol still represents an overhead for each log entry. More pertinently, this overhead is sequential, since each log entry needs to be processed before moving on to the next one. A simple way to further improve performance under load is therefore to batch process log entries. DPDK already provides an efficient vector interface to receive multiple packets waiting in the NIC queue. We chain these packet buffers together in userspace and treat them as a single RAFT log entry. This effectively amortizes the CPU cost of running the RAFT protocol state machine over multiple log entries.

4.2.3 Execution

Blizzard's execution layer aims to concurrently execute *committed* operations in RAFT's execution logs. We depend on the application programmer to specify commutativity between operations (Section 4.2.1). The execution layer couples with the replication layer via a set of queues to receive operations on, along with shared flags in the RAFT log entries, needed to agree on the state of the operation – replicated, executing and complete. The most complex part of Blizzard's execution layer is the scheduler that aims to schedule ready operations as soon as possible, while respecting commutativity. The actual execution leverages PMDK's persistent memory transaction library to enforce failure atomicity. We discuss each of these components below. Finally we also discuss the implications of declaring operations as commutative and how the programmer can control departure from serializability for better performance.

4.2.3.1 Coupling

Figure 5 illustrates the overall design and interfacing between Blizzard's replication and execution layers. Every read and write operation received by the replication layer is added to a queue (`Q`) of operations to be considered by the scheduler. The entry in the queue also includes a pointer back to the RAFT log entry for the operation. The RAFT log entry includes a set of flags read from and written to by both the execution and replication layers to allow them to communicate the state of the operations. Most importantly, these flags are persistent and survive restarts, forming the foundation for recovery. All currently executing operations are in a set (`E`) that is asynchronously consulted by execution threads

to determine which operations can be selected. After the execution completes, the execution thread marks the operation as complete in the status flags.

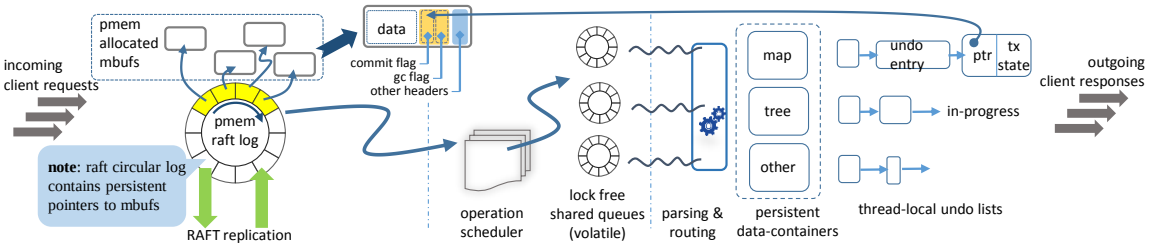

Figure 5: Blizzard - Coupling replication to execution

4.2.3.2 Scheduling

The scheduler runs as part of the continuous event loop in Blizzard, executing the scheduling algorithm shown below.

Algorithm 1: `Blizzard` operations scheduler algorithm.

input: 1. queue Q of updates and reads.
2. Set E of operations that are ready-to-execute/executing

```
1 repeat
2     if Q.head().state == FAILED_REPLICATION then
3         Q.dequeue()
4     else if Q.head().state != REPLICATING and Q.head() commutes with all ops in
      E then
5         op = Q.dequeue()
6         E.insert(op)
7 until server-shudown;
```

The scheduler considers scheduling the operations that the replication has been successful, or immediately for a read operation, the operation at the head of `Q` is considered for execution. The scheduler checks the operation against all currently executing operations in the set `E`. If it commutes with all operations in `E`, the operation is added to `E` for execution. Operations at the head of `Q` that have failed replication (perhaps due to a RAFT leadership change after the operation was received) are removed from consideration by the scheduler.

We note that failure recovery proceeds exactly as with normal operation – replicated operations are moved to execution regardless of their progress in previous attempts.

4.2.3.3 Execution

Execution of operation in Blizzard is done by dedicated executor threads that pick up (with appropriate synchronization) operations added to the set `E`, execute it, and then remove it from `E` after execution is completed and its effects persisted to memory. Execution follows the steps detailed below:

Algorithm 2: `Blizzard` Execution algorithm.

```
   input: o = operation to execute, E = set of executing/ready operations
 1 if o.state != COMPLETED then
 2     BEGIN FAILURE_ATOMIC_TX
 3     lockset = []
 4     HandleRPC(o.request, lockset, o.response)
 5     o.state = COMPLETED
 6     END FAILURE_ATOMIC_TX
 7     foreach l in lockset do
 8         l.ReleaseLock()
 9 remove o from E
10 Send o.response to client
```

The replication and execution layers cooperate to ensure that the states for an operation move as per the state machine diagram shown in Figure 6.

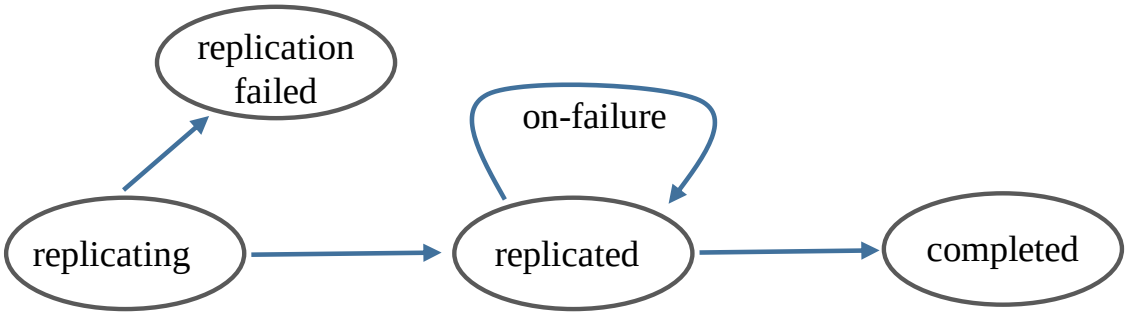

Figure 6: State machine diagram of an operation in Blizzard

We draw particular attention to recovery. If an operation begins execution but fails before executing, its persistent state flag continues to be set at `REPLICATED` when the system restarts. Any persistent changes made by the previous execution are automatically undone by PMDK. It then proceeds as usual through the current attempt till completion.

On the other hand if an operation finishes execution and manages to move its persistent state flag to `COMPLETED`, we do not execute it again by checking for this condition, thereby ensuring that operations are executed exactly once with respect to changes to persistent memory.

4.2.3.4 Commutativity

We now consider how commutativity mediates the interaction between replication and concurrency. The base case we start with is when no two operations are declared as commutative, in other words `Commutes` always returns false. In this case operations are executed exactly in the order specified by their committed order in the `RAFT` log. Since all replicas execute operations in the same committed order this means update operations follow the same serialized order on all replicas. Further, we execute reads in the order they are received, *after* any current operations in the log have committed - this is enforced by only scheduling from the head of the queue `Q` in the scheduler. This means a client sending a read request after it had a committed update request, is guaranteed to see the results of its own updates. Therefore, this baseline implementation of commute leads to serializability with read-your-own-writes consistency.

As an example of allowing some commutativity, consider a single container in persistent memory with a dictionary (implemented as a persistent hash table or tree) interface. Most such APIs (e.g., in C++ STL containers) disallow operations to multiple keys. Therefore a natural setting for commutativity is to allow operations (reads or writes) to different keys to commute, since reads by clients cannot reveal out-of-order application of operations to different keys at different replicas. In such as situation, the programmer can set `Commutes` to return true if and only if the operations are made to different keys. The result is also serializability with read-your-own-writes consistency but *restricted* to a single key.

CHAPTER V

PROPOSED WORK, AND TIMELINE

5.1 Proposed Work

For the remainder of my dissertation research I plan to focus on verifying Blizzard's proposed solution (hypothesis) and gaining deeper insights of the overall system behavior. The proposed work includes the following.

5.1.1 Implementing Blizzard's core software

The task includes implementing and evaluating Blizzard's interface, replication and execution components.

- **Interface:** Implement remote, persistent data-container interface libraries: a client-proxy library for remote data-structure operations and a server-side library supporting incoming message parsing and routing.
- **Replication:** The implementation of Blizzard reuses the open sourced replication runtime Cyclone [**cyclone**]. We plan on extending it to work with real NVM hardware, including via porting the DPDK [**dpdk**] network stack to use NVM memory, augmenting the runtime with persistent smart-pointers, and other necessary steps.
- **Execution:** Implement and evaluate the tradeoffs afforded by Blizzard's commute scheduler and crash-consistency protocol.

5.1.2 Implementing Blizzard applications

To prove the flexibility offered by Blizzard to support diverse applications requiring persistence, fault-tolerance, and performance, we plan to implement three real-world applications to use Blizzard's fault-tolerant persistent memory structures.

- **Persistent key-value store:** Key-value stores are analogous to hashmap and tree main memory data-structures. We plan to implement concurrent, persistent B-tree and a hashmap that can work with Facebook's [**fbmemcache**] key-value store workload.

- **Web application backend:** Lobsters [**lobsters**] is an Hackerank-like news aggregation site. User submitted posts get visibility in-proportion to their community vote count. We plan to implement a Blizzard backend for Lobsters using persistent memory queue structure that supports top-k operation. We evaluate our implementation with the Noria benchmark vote [**noria**].

- **Streaming graph analytics:** We plan to implement a Blizzard persistent memory backend for streaming graph analytics using an adjacency-list in-memory data structure. We will evaluate our implementation with twitter [**twitter**] graph benchmark.

5.1.3 Comprehensive evaluation of Blizzard

The evaluation of Blizzard will consider both the performance (i.e., its implementation) and the functionality (i.e., its features), using the above workloads. We have selected RocksDB [**rocksdb**], memcached [**memcached**], and Llama [**llama**] as tentative baseline backends as of now. Most of the evaluation will be conducted on a 3 node server cluster with real NVM based on Intel's Optane DC Persistent Memory, that is available to us through Intel's academic early access program.

5.2 Timeline

- **Fall 2019:** Complete the implementation of Blizzard's core software pieces and use cases. We plan on targeting Eurosys for a potential paper submission based on the initial evaluation numbers.

- **Jan 2020:** Complete evaluation of Blizzard with all three use cases on the Optane-based cluster. We also plan to explore the opportunities for evaluations at larger scale.

- **March 2020:** Submitting a complete paper on Blizzard to OSDI.
- **March 2020:** Thesis defence.
- **Jan-May 2020:** Job search.
- **May 2020:** Graduation.