

Vorpai: Vector Clock Ordering For Large Persistent Memory Systems

Kunal Korgaonkar
kkorgaon@eng.ucsd.edu
UC San Diego

Joseph Izraelevitz
jizraelevitz@eng.ucsd.edu
UC San Diego

Jishen Zhao
jzhao@eng.ucsd.edu
UC San Diego

Steven Swanson
swanson@eng.ucsd.edu
UC San Diego

ABSTRACT

In systems with non-volatile main memories (NVMMs), programmers must carefully control the order in which writes become persistent. Otherwise, what will remain in persistence after a crash may be unusable upon recovery. Prior art has already explored semantic models for specifying this persist order, but most enforcement algorithms for the order are not scalable to large server machines because they assume that the machine contains only one or two memory controllers. In this paper, we describe a collection of provably-correct algorithms for enforcing the persist-order across writes, generated at many different cores, and persisted across numerous different memory controllers. Relative to existing solutions, our algorithms improve performance by 48% by reducing both traffic and serialization overheads.

ACM Reference Format:

Kunal Korgaonkar, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. 2019. Vorpai: Vector Clock Ordering For Large Persistent Memory Systems. In *2019 ACM Symposium on Principles of Distributed Computing (PODC '19)*, July 29–August 2, 2019, Toronto, ON, Canada. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3293611.3331598>

1 INTRODUCTION

Non-volatile main memories (NVMMs), such as phase-change memory (PCM) [46], resistive RAM (ReRAM) [38], and 3D Xpoint [31], are likely to bring profound changes to many aspects of computer systems. Like DRAM, these devices are *byte-addressable*, and programs interact with them using the traditional load/store (read/write) interface. Unlike DRAM, but like disk, these devices are *non-volatile* — their contents are guaranteed to survive a power outage. As such, NVMMs are poised to provide a medium for fast, durable storage. Given their non-volatile property, NVMM are also referred to as *persistent* memories.

Unfortunately, from the perspective of durable storage, NVMMs are somewhat undermined by the fact that processor caches and

registers lose their contents on power failure. Consequently, barring additional effort, data in NVMMs is likely to be unusable after a power failure because processor caches and registers lose their contents on power failure — what will survive in NVMM is that data that was not held within the cache hierarchy at the time of failure. As a consequence, NVMMs require programs to carefully control the order in which their writes reach persistent memory. A failure in the middle of a linked-list insertion, for example, may lead to a post-crash dangling reference, if the “next” pointer of the predecessor node is written into persistent memory while the pointed (inserted) node itself remains in a cache.

To give programmers a tool to manage the order of writes to persistent memory, modern ISAs provide a *memory persistency model*, which is a set of instructions that control the order in which writes to NVMM become persistent, such that they will be visible after a power failure [35]. The memory persistency model is specified in conjunction with of the similar, yet, different, *memory consistency model*, which describes the order in which concurrent threads’ updates become visible to each other. These models are generally orthogonal (two ISAs might implement the same persistency model on top of different consistency models), but their interactions are generally tightly coupled — a persistency model might leverage the memory consistency model to induce orderings.

A growing body of research [20–23, 32, 35, 40, 41] explores ISAs and supporting micro-architectures to enforce these ordering constraints. Despite this previous work, scalability of persistence ordering support still remains an issue for NVMMs. Most previous work only considers the ordering of NVMM writes through one or two memory controllers [4, 20–23, 32, 40], but modern servers can include many memory controllers scattered across multiple sockets [15]. Enforcing ordering constraints between writes generated at many different cores and written into NVMM (that is, *persisted*) at many different memory controllers requires an efficient algorithm to capture and enforce these constraints.

In this paper, we propose a set of novel distributed algorithms for ordering writes to persistent memory on a large, shared-memory computer system. Our collection of algorithms, which we call Vector Ordered Persistence Algorithms (Vorpai), uses a vector clock [12, 30] representation to (1) construct a partial order over the writes to NVMM and (2) efficiently enforce their order at multiple memory controllers.

In particular, this paper makes the following contributions:

- We describe a basic vector clock algorithm for enforcing write order to persistent memories and demonstrate that it can implement the acquire-release memory persistency (ARP) model, defined in Section 2.2.3.
- We address the scalability problems in vector clocks as a write ordering mechanism in cache-coherent, shared-memory

This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. We thank our shepherd, Marc Shapiro, for his valuable guidance in writing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PODC '19, July 29–August 2, 2019, Toronto, ON, Canada

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6217-7/19/07...\$15.00

<https://doi.org/10.1145/3293611.3331598>

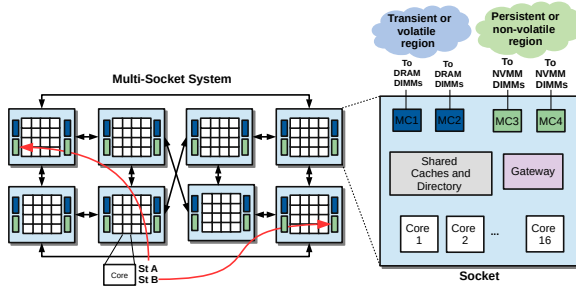


Figure 1: System overview Shows the expected memory system layout of a scale-up server with NVMM and DRAM. The left-hand side of the figure highlights the persistent write ordering problem where instructions ‘Store A’ and ‘Store B’ generate ordered writes. The right-hand side of the figure shows the participating entities of Vorpil inside a socket.

systems with a large number of memory controllers with other, more advanced, new vector clock algorithms.

- We provide safety and liveness proofs for each of the proposed algorithms.

2 BACKGROUND

This section describes the necessary architecture and theory background for situating the Vorpil algorithms, which use vector clocks to represent and enforce ordering across updates to persistent memory.

2.1 System Model

Future scale-up servers are expected to have tens of sockets, hundreds of cores, and tens of memory controllers controlling terabytes of memory [15]. In this work, we target a large, cache-coherent, shared-memory server that is divided into *sockets*, each of which has *cores* that execute programs and generate writes to persistent memory, *memory controllers* or MCs that persist the writes to NVMMs (see Figure 1), and a single *gateway* node used to manage asynchronous cross-socket communication (and used as a convenient point for processing outgoing messages from the socket). Without loss of generality, we assume single-threaded cores (e.g. hyperthreads can be considered two logically distinct cores). The components of this system are connected by an asynchronous, loss-less, FIFO, bounded-delay network (the memory bus) with no out-of-thin-air messages (every message received was sent and is not corrupted).

Given a write’s physical address, there is exactly one controller that can persist it [19]. Note that successive ordered writes to persistent memory from a single core may be persisted at different memory controllers (see Figure 1, left-hand side).

2.2 Programming Model

2.2.1 Memory Regions. Our programming model assumes that a program’s address space is divided into a *transient* and *persistent* region [36], both accessible via the traditional load/store (read/write) interface. For a given memory access, the hardware can determine whether it is persistent or not by its address. In the transient region, the program keeps state that it does not require to survive a crash, e.g., run-time variables that need not be persistent and can

be regenerated at recovery such as locks, certain indexes, and (most likely) the stack. Since the transient region does not need to be usable in the wake of a crash, it is generally expected to be faster and might even reside in DRAM. Traditional programs without the use of NVMM can be viewed as programs that only contain a transient region. Figure 1 shows how the transient and the persistent regions are clearly separated between the DRAM and NVMM DIMMs, respectively.

In the persistent region, the program keeps state that it requires to survive a crash. This region resides in NVMM. The programmer must ensure that its state is updated in an order that allows it to be usable after a crash (For instance, in a linked-list insertion, the inserted node must be made persistent before the pointer to it, in order to avoid a dangling pointer in the event of a crash).

2.2.2 Memory Consistency. As with any parallel programming model on a shared-memory system (including those without persistent memory), our programming model uses a *memory consistency model*. A memory consistency model is a semantic model that describes the order in which an individual thread’s memory updates become visible to other, concurrently running, threads. We chose the *release consistency* memory model [14] for this purpose. The release consistency model corresponds closely to that of the ARM v8 instruction set [3].

For single-threaded programs, release consistency is simple. A *store* instruction writes to a memory address, and a *load* instruction reads from a memory address. A load returns the value most recently written by a store to the address, where “recently” is with respect to the thread’s executed program-order ($<_{PO}$).

In multi-threaded programs, release consistency also defines the order in which one thread’s stores become visible to other threads (called the *release-acquire order* [1] and written as $<_{RA}$). A load returns the value of the most recent store within the release-acquire order, defined as follows. A program’s (shared) memory is logically divided into two types of variables. *Data variables* are written-to, and read-from using normal load and store instructions. *Synchronization variables*, in contrast, are written-to and read-from exclusively using the special *load-acquire* and *store-release* instructions (abbreviated respectively as load-acq and store-rel), which define how one thread’s writes propagate to another.

In more detail, the interaction between store-release and load-acquire on synchronization variables defines the release-acquire order (Figure 2), which in turn defines the order in which one thread’s stores become visible to another’s loads. When a thread uses a load-acquire to read a synchronization variable, it *synchronizes-with* the last thread that used a store-release to write to that location, and the load-acquire returns the store-release’s written value. Therefore, when threads synchronize-with each other (e.g., by passing a lock or flag), the first thread’s store-release *precedes* the second thread’s load-acquire in release-acquire order (see Figure 2, where Thread 2 load-acquires x). Furthermore, any of the first thread’s writes (to data or synchronization variables) that preceded the store-release in program order must now be visible to the second thread’s reads that follow the load-acquire (see Figure 2, where Thread 2 loads a). The release-acquire order is also transitive — if the second thread

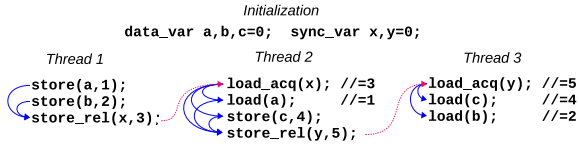


Figure 2: Memory Consistency Threads' updates become visible to other threads using the release-acquire order under the release consistency model. Arrows show the release-acquire relationships between instructions, and red arrows show those relationships that are also synchronize-with pairs.

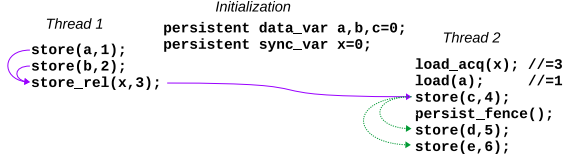


Figure 3: Memory Persistency Threads' updates are ordered into persist memory under the ARP memory persistency model. Arrows show the persist-before relationships between instructions. Purple, solid, arrows show those persist-before orderings derived from the release-acquire order. Green, dotted arrows show those orderings that are not release-acquire orderings but are persist-before orderings (ordered by the persist fence).

issues a subsequent store-release and a third thread synchronizes with it using a load-acquire, the third thread will also see the first thread's writes (see Figure 2, where Thread 3 loads b).

For example, in a lock-based programming model under release consistency, every (data) memory location is associated with a specific lock. While a process holds a lock (a synchronization variable), i.e., between acquiring and releasing the lock, it has sole access to the associated memory locations (data variables), which it can read (load) or update (store). Before the first access to a memory location, the program must acquire the associated lock (load-acquire) and pull the current value of the location from the last writing process that has released the lock (via a store-release).

Note that the release-acquire order is a partial (not total) order across instructions, thereby allowing for aggressive optimization at the compiler and processor level. For instance, sequential stores to data variables by a thread can be reordered with no ill effects (they are not ordered by release-acquire and their effects are not guaranteed to be visible to other threads until the next store-release). It is important to also note that our memory consistency model relies on the *coherence* of synchronization variables — that is, all threads agree on the order of store-release and load-acquire's to a given synchronization variable and a load-acquire to that variable will give the most recent value stored by a store-release in this same order.

2.2.3 Memory Persistency. In persistent memory programming, not only must define a memory consistency model, but we also must define the *persists-before order*, that is, the order in which writes (i.e. stores) become persistent by leaving the cache hierarchy and entering NVMM¹. The persist-before order (which we

¹Hereafter we write "store" or "load" to refer to a store or load instruction respectively. A store first enters the cache, then later becomes persistent once that cache-line is written back to NVMM.

write as $\leq_{PB}$) is controlled using an ISA level *memory persistency model*, which is defined in addition to the memory consistency model. Like the release-acquire order, the persists-before order is a partial order, though, unlike release-acquire, persists-before only includes stores (and store-release's) to persistent memory addresses (as opposed to all memory accesses). In the event of a power failure, the programmer can expect that what will survive in NVMM is some consistent-cut across the persists-before order: given a store that was written to NVMM and survived a power failure, any prior store in persists-before order should also have been written and survived. Assuming the programmer used the persistency model correctly, this cut should be a state from which the program can recover its data. We chose a relatively relaxed persistency model called *acquire-release persistency* (ARP) [23] in order to demonstrate the flexibility of the Vorpall algorithms — with some work, Vorpall should support stricter persistency models as well.

As the name suggests, ARP is a persistency model that leverages the release consistency memory model to define the persists-before order. In ARP, all release-acquire orderings in the memory consistency model are *also* persists-before orderings in the memory persistency model; given two stores S_1 and S_2 to persistent addresses, $S_1 <_{RA} S_2 \implies S_1 <_{PB} S_2$. This rule means that if one thread updates persistent state, then synchronizes with another thread, any of the second thread's later persistent writes will be ordered after the first thread's (in Figure 3, Thread 1's stores to a and b will become persistent before its store to x , and its store to x will become persistent before Thread 2's store to c). This rule simplifies reasoning for programs that are both concurrent and use persistence. However, this rule not sufficient. Writes to persistent memory might also need to be ordered within a single thread independent of synchronization. For instance, in a linked-list insertion, the inserted node must become persistent *before* the pointer to it, yet these two writes are unordered by the release-acquire order (as they are contained within a single thread). In order to solve this problem, ARP adds a *persist fence* instruction that ensures that stores that precede the fence (in program order) persist-before those that follow the fence: given two stores S_1 and S_2 to persistent addresses and persist fence F , $S_1 <_{PO} F <_{PO} S_2 \implies S_1 <_{PB} S_2$. (in Figure 3, Thread 2 ensures the otherwise unordered stores to c and d become persistent in program order by putting a persist fence between them, otherwise, as is the case with stores a and b , the writes to persistent memory will be unordered). Like the release-acquire order, the persists-before order is transitive, that is, given three stores S_1 , S_2 , and S_3 to persistent addresses, $(S_1 <_{PB} S_2 \text{ and } S_2 <_{PB} S_3) \implies S_1 <_{PB} S_3$. Thus, in Figure 3, Thread 1's store to a will become persistent before Thread 2's store to c).

2.3 Vector Clocks in Vorpall

We expect the hardware of our system to enforce the persists-before order ($\leq_{PB}$) across the many memory controllers in different sockets. For correctness, at any instant in (asynchronous) time, those stores persisted across all memory controllers must form a consistent-cut across the persists-before order. Our Vorpall algorithms represent this persists-before order using *vector clocks*.

Vector clocks [12, 30] are a well-known construct for representing logical time in a distributed system. If n is the number of nodes,

a vector clock is an n -tuple of positive integers. Typically, each node maintains its own copy of a vector clock and every message between nodes is tagged with a copy of the sender's vector clock (we term this copy a *timestamp*). For a vector clock V , we note V_i the i th entry in the vector. Vector clocks are useful in that they represent time as a partial order: every node's vector clock acts as a local clock occasionally synchronized with remote clocks.

With vector clocks, a node can progress local time by *incrementing* the vector entry that corresponds to the local node by one. We express this operation as: $V_i + +$, that is, an increment by 1 of the i th entry of V . When the local node receives a message from a remote node, it updates its local clock to include all remote clock times prior to the message's send time, that is, it *merges* its local clock with that of the message's send time. We express the merge of vector clocks U and V as $U \oplus V = Z$ where $\forall_i Z_i = \max(V_i, U_i)$. As vector clocks describe a partial order, we can compare clocks easily. Given clocks U and V , $V \leq U \iff \forall_i V_i \leq U_i$.

In Vorpai, we use vector clocks to represent and enforce the persist-before orderings. Every core has its own local clock stored in a vector clock register. The size of the vector clock register at any core is equal to the number of cores in the whole system. Cores tag the persistent store (and store-release) instructions they generate with a timestamp. When a core synchronizes with another other (i.e., when one core load-acquires another's store-release), the acquiring core will merge its clock with the timestamp it receives (i.e., that timestamp attached to the store-release). In addition, each memory controller maintains its own vector clock register, of size again the same as the number of cores in the system, to track what stores (and their associated timestamps) have already been persisted. For our more advanced hierarchical Vorpai algorithm, clock registers at each socket gateway summarize clocks within the socket, which allows us to use smaller vector clocks (details in Section 6).

In general, we will discuss the metadata storage cost of vector clock registers and other costs as we proceed through describing each of the algorithms below. Detailed analysis of the costs follows with results in Section 7.

3 OVERVIEW

Vorpai is a scalable, distributed mechanism for enforcing memory persistency models.² It uses vector clocks [12, 30] for encoding the persist-before orderings across stores and provides a scalable distributed algorithm for persisting stores at the memory controllers in accordance with this ordering.

Vorpai includes several variants, each of which is a refinement of previous versions. The simplest, Vorpai₀, is simple to implement but incurs extraneous orderings between persistent stores. Its refinement, Vorpai_{chunk}, operates on a group of mutually-unordered stores rather than single stores, thereby supporting a more relaxed persistent order (necessary for higher performance). However, neither Vorpai₀ nor Vorpai_{chunk} exploits the hierarchical nature of systems — and for systems with many processors, these algorithms will require very large vector clocks. To shrink the vector clocks, we introduce Vorpai_H, a variant that maintains smaller clocks by

leveraging the hierarchical nature of modern systems [15]. For a brief summary of the metadata cost of each algorithm, see Table 1.

3.1 Vorpai Definitions and Preliminaries

Vorpai algorithms comprise two parts: *order construction* and *order enforcement*, each with its own requirements. We first explain both parts, then define correctness (both safety and liveness) for the entire class of Vorpai algorithms. Finally, we prove a theorem that shows if Vorpai algorithms satisfy the necessary requirements for both order construction and order enforcement, then they are safe. We leverage this theorem in proving each Vorpai variant.

Order Construction. In the order construction part, the Vorpai algorithm assigns vector timestamps to persistent stores (i.e., store instructions). The order built by these timestamps, termed the *clock order*, is designed to subsume the persists-before order.

In order to build the clock order, every single core maintains a vector clock register that will represent its local and monotonically-increasing logical time. We call this register the *core vector clock*. For simpler Vorpai algorithms, this vector clock contains as many entries as there are cores in the system (i.e., if there are N cores in the system, each core vector clock contains N entries). Our more advanced algorithm shrinks the register size for performance. Assuming the Vorpai algorithm works correctly, this clock represents the local core's view of the persist-before order, that is:

Definition 3.1 (Order Construction Requirement). A Vorpai algorithm satisfies the *order construction requirement* if and only if, given stores S and T with timestamps V^S and V^T , $S \leq_{PB} T \implies V^S < V^T$.

Order Enforcement. In the order enforcement part, memory controllers persist stores according to their clock order and, consequently, their persists-before order (given that the store's timestamps meet the Order Construction requirement). The controller persists new store if and only if *all* preceding stores in clock-order have been persisted, that is:

Definition 3.2 (Order Enforcement Requirement). Consider the set of persisted stores (with their timestamps) *Persisted* and some store S with vector timestamp V^S . A Vorpai memory controller that satisfies the *order enforcement requirement* will increase the set of persistent stores to *Persisted* := *Persisted* $\cup$ S if and only if $\forall_T V^T < V^S \implies T \in \text{Persisted}$.

To do order enforcement, each memory controller maintains a *progress vector clock* register to track the completion of stores. At any given time, the progress register P is equal to the merge of all persisted timestamps that the controller is aware of (i.e. it has either persisted them itself or it has received notification of their persistence from other controllers).

3.2 Correctness

As all Vorpai algorithms have a similar structure, we define our correctness criteria (safety and liveness) here. We also prove that if a Vorpai algorithm meets both the Order Construction and Order Enforcement requirements, it is safe. Subsequent proofs of correctness (following each variant description) will leverage this property to demonstrate our various Vorpai algorithms are correct.

²We chose ARP for this work, though the Vorpai mechanism should extend to any stricter persistency model.

```

1 // Corei State and Events (Order Construction)
2 clock_register C;
3 On: Issue persistent store or persistent store-rel  $S$  to MCs Pending set
4    $C_i + +; V^S := C;$ 
5 On: Issue store-rel  $S$  (non-persistent)
6    $V^S := C;$ 
7 On: Issue load-acq  $A$  that synchronizes-with store-rel  $S$ 
8    $C := C \oplus V^S;$ 
9 // MC State and Events (Order Enforcement)
10 clock_register P;
11 On: Occasionally
12   if ( $\exists S \in \text{Pending} \wedge S \notin \text{Persisted} \wedge \text{immediately\_succeeds}(P, V^S)$ )
13     {  $\text{Persisted} := \text{Persisted} \cup S; P := P \oplus V^S;$  }
14 On: Received broadcast clock  $B$ 
15    $P := P \oplus B;$ 
16 On: Occasionally
17   Broadcast  $P;$ 

```

Figure 4: Vorpalo₀: The pseudo-code for Vorpalo₀, the baseline version of Vorpal. Actions of the cores and the memory controllers (MC) for order construction and enforcement, respectively, are shown.

Definition 3.3 (Safety). A Vorpal algorithm is safe if and only if, at any given point in time, the set of persisted stores is a consistent-cut across the persists-before order; that is, if a store is persisted, all of its predecessors in persists-before order are also persisted.

THEOREM 3.4 (SAFETY). Any Vorpal algorithm that meets both the Order Construction and Order Enforcement requirements is safe.

PROOF. By induction. **Base case.** $\text{Persisted} = \emptyset$. The empty set is a consistent-cut across any partial order. **Induction.** We induct across store persists. Assume Persisted is a consistent-cut across the persist-before order and some controller adds a store S such that $\text{Persisted}' = \text{Persisted} \cup S$. By order enforcement, S is only persisted $\forall_T V^T < V^S \implies T \in \text{Persisted}$, and by order construction, $S <_{PB} T \implies V^S < V^T$. Therefore, store S is added to Persisted only if $\forall_T T < S, T \in \text{Persisted}$. Since, for any partial order, the union of two consistent-cuts (both Persisted and S and all its predecessors) is itself a consistent-cut, $\text{Persisted}'$ is a consistent-cut across the persists-before order. $\square$

Definition 3.5 (Liveness). A Vorpal algorithm is live if and only if, given a long enough failure-free execution, a given persistent store will eventually become persisted.

4 VORPAL₀: BASELINE VORPAL

Having laid out the requirements for a Vorpal algorithm, we introduce Vorpalo₀, a simple (though conservative) mechanism for meeting the two Vorpal requirements. Subsequent sections will refine the algorithm. Its code is shown in Figure 4 using an event-driven programming model. For simplicity, we assume any code block is executed atomically.

Order Construction. In Vorpalo₀, as with all Vorpal algorithms, whenever a core issues a store, it tags the stores with a timestamp, setting it to the core's local vector clock. In Vorpalo₀, the algorithm builds an over-constrained clock order using a timestamp that contains an entry for every core in the system. At every store, the issuing core increments its own clock for its own entry (line 3), and

tags the store with the timestamp. At every load-acquire, the core merges the local clock with the timestamp value attached to the corresponding store-release (line 8).

Vorpalo₀ meets the order construction requirement (the clock order across stores subsumes the persist-before order). To show this, we consider all possible ways two stores S_1 and S_2 to persistent addresses can be persist-before ordered under ARP. If S_1 and S_2 are issued by the same thread and persist-before ordered ($S_1 <_{PB} S_2$), they can only be persist-before ordered under ARP by some instruction X where X is a persist fence, store-release, or load-acquire, and $S_1 <_{PO} X <_{PO} S_2$. Since every store is assigned a timestamp issued in program order, this persist-before ordering is subsumed by the clock order. If S_1 and S_2 are issued by different threads and persist-before ordered, then $S_1 <_{RA} S_2$, consequently, there is a synchronizes-with event between them. Since, when issuing a load-acquire that synchronizes-with a store-release, a core merges its clock with the incoming store-release's timestamp, any subsequent stores it issues will be clock ordered after the incoming store-release — so the persist-before ordering is subsumed by the clock order.

Order Enforcement. Stores issued by the cores and those already assigned a timestamp (by Order Construction) arrive at their respective memory controllers and are placed in that controller's *Pending* set. Vorpalo₀'s memory controllers enforce the ordering created by the processors using a simple rule that leverages its local progress vector clock (P), which, like the core vector clocks, contains an entry for every core. Whenever a controller persists a store (line 13), or discovers the store has been persisted at another controller (line 15), it merges the store's timestamp to its clock. In order to ensure all stores are eventually persisted, controllers occasionally broadcast their clocks to all other controllers (line 17). A controller who receives a broadcasted timestamp value will merge the timestamp into its own local progress clock.

To decide whether to persist a store, the memory controller uses a notion we term *immediate succession*. A timestamp B is the *immediate successor* of timestamp A if and only if $B_i \leq A_i$ for all vector entries i , except, for some vector entry j , $B_j = A_j + 1$. This is a useful concept because, if B is the immediate successor of A , then there cannot exist a timestamp X where $A < X < B$. Put another way, if X is a predecessor of B , it must be less than or equal to A . In code, we write the test of immediate succession as `immediately_succeeds(A,B)`, checking if B immediately succeeds A .

In Vorpalo₀, a memory controller persists a pending store S if it has received with timestamp V^S if and only if the store's timestamp is the immediate successor to the current progress clock P . This ensures that, for any persisting store, all of its predecessors in clock order have already become persistent. The controller can persist several stores at once in a batch, assuming they are all immediate successors of its current progress clock. The ability to do so comes from the asynchronous nature of the system (see system description in Section 2.1), as well as, the usage of vector clocks by cores and memory controllers in concert (as described in 2.3).

We can show that Vorpalo₀ meets the order enforcement requirement by induction on P , where our hypothesis is that any store with a clock $V \leq P$ is persisted. When a controller persists some store S , any preceding store $T < S$ must have a timestamp smaller than V^S

(by order construction), and, consequently, V^T must be less than or equal to P (by the immediate succession property). Therefore T is persistent (by induction).

Liveness. Vorp₀ is not only safe (by order construction and enforcement), but also live. We reason by contradiction: suppose some store S cannot be persisted and has the smallest timestamp of any non-persisted store. Given a long enough failure-free execution, any persistent store (including S) will be issued to their controllers. Since S cannot be persisted, eventually all progress clocks will reach (or exceed) a timestamp value P such that $\forall T \in \text{Persisted } V^T \leq P$, where, by safety, $P \leq S$. By our contradiction assumption, S cannot be the immediate successor to P since it cannot be persisted. Therefore, there exists some store $S' \notin \text{Persisted}$ where $P < S' < S$; yet the existence of S' violates our contradiction assumption that S has the smallest timestamp of any store that cannot be persisted.

5 VORPAL_{chunk}: CHUNK-GRANULARITY VECTOR CLOCK ORDERING

Vorp₀ is correct but introduces many extraneous clock orderings. In particular, it assumes that all stores issued by a given thread must persist in program order, but this constraint is in fact not required by ARP. For instance, in Figure 3, stores to a and b would be persisted in program order by Vorp₀, but this enforcement is unnecessary as they are not persist-before ordered — ARP assumes that a thread's stores are unordered into persistence unless constrained by a release-acquire ordering or persist fence.

5.1 Base Vorp_{chunk}

In this subsection, we introduce a relaxed version of Vorp₀, called Vorp_{chunk}, which groups together stores that are unordered with respect to one another. These groups of unordered stores, or *chunks*, are those stores program-ordered between persist-before ordering instructions (store-release, load-acquire, or persist fences), issued from a single thread, and destined for a single memory controller (we relax this last restriction in Section 5.2). For instance, assuming all stores are destined for the same controller, Figure 3 contains four chunks ($a, b; x; c$; and d, e). All stores within a chunk are all assigned the same timestamp. Like all Vorp₀ algorithms, the clock order captured by Vorp_{chunk} ($\leq_C$) subsumes the persist-before order, that is: $\leq \subseteq \leq_C$, and, furthermore, the controller, in turn, completes the chunks following the generated vector clock order (corresponding to $\leq_C$). We show its code in Figure 5.

Order Construction. In Vorp_{chunk}, a chunk is created by incrementing the local clock register — all stores in the chunk will receive the same timestamp (as with Vorp₀, the timestamp has as many entries as there are cores in the system). A new chunk of stores is created on seeing new store following an earlier persist fence or load-acquire (lines 22 and 21 respectively). In addition to these instruction-triggered chunk boundaries, to simplify enforcement across memory controllers, Vorp_{chunk} creates a new chunk when the next store is issued to a different memory controller than the previous one (line 21). Cores also occasionally start new chunks to maintain liveness (line 32).

When a core starts a new chunk, the core sends the previous chunk, along with the count of stores inside the chunk, to the corresponding controller and resets the count of the number of

```

18 //Corei State and Events (Order Construction)
19 clock_register C; Integer ChunkCount; Set<Store> Chunk;
20 On: Issue persistent store or store-rel S
21   if (S switches controller  $\vee$  S is first store after load-acq
22        $\vee$  S is first store after persist fence)
23     Ci ++; Issue Chunk w/ ChunkCount to MCs Pending Set;
24     ChunkCount:=0; Chunk:= $\emptyset$ ;
25   }
26   Add S to Chunk; ChunkCount++; VS := C;
27 On: Issue store-rel S (non-persistent)
28   VS := C;
29 On: Issue load-acq A that synchronizes-with store-rel S
30   C := C  $\oplus$  VS;
31 On: Occasionally when ChunkCount $\neq$ 0
32   Ci ++; Issue Chunk w/ ChunkCount to MCs Pending Set;
33   ChunkCount:=0; Chunk:= $\emptyset$ ;
34 //MC State and Events (Order Enforcement)
35 clock_register P;
36 On: Occasionally
37   if( $\exists$  chunk Chunk  $\in$  Pending  $\wedge$  Chunk  $\notin$  Persisted
38        $\wedge$  immediately_succeeds(P, VChunk))
39     {Persisted := Persisted  $\cup$  Chunk; P := P  $\oplus$  VChunk;}
40 On: Received broadcast clock B
41   P := P  $\oplus$  B;
42 On: Occasionally
43   Broadcast P;

```

Figure 5: Vorp_{chunk}: The pseudo-code for Vorp_{chunk}, a vector clock Vorp₀ algorithm that constructs and enforces ordering at a relaxed granularity of chunks. Actions of the cores and the memory controllers (MC) are shown.

stores (*ChunkCount*) inside a chunk (lines 23 and 32). Controllers use the *ChunkCount* for order enforcement: they must be sure all stores in any earlier chunk are persistent (by checking the number of stores tagged with the earlier chunk's timestamp) before it can persist any store in a subsequent chunk.

Vorp_{chunk}'s clock-order subsumes the persist-before order (and therefore meets order construction). Instructions that induce a persist-before ordering (either a persist fence, store-release, or load-acquire) result in a chunk boundary the increment of the local clock register. The use of chunks is safe since stores in a chunk not persist-before ordered with respect to each other.

Order Enforcement. Vorp_{chunk} inherits its order enforcement code at the memory controller from the simpler Vorp₀, but its code operates on chunks instead of stores. That is, if some pending chunk has yet to be persisted and is the immediate successor to the progress register P , then all component stores in the chunk can be persisted by the controller (line 36). Note that the count of stores inside the chunk is made known to its controller by the core at chunk dispatch time, hence it can safely complete the chunk's component stores in any order and then mark the chunk as complete (chunks can be sent piecemeal if necessary). As with Vorp₀, the memory controller occasionally broadcasts its clock to allow for cross-controller progress (line 40).

For the correctness of order enforcement, the proof previously laid out for Vorp₀'s enforcement can be reapplied verbatim, substituting “chunks” for “stores”. This substitution is possible because an individual chunk is assigned a single timestamp, and stores that share a timestamp are unordered. Similarly, the Vorp₀ proof of liveness applies, substituting in chunks for stores.

5.2 Vorpals_{chunk+}

Although Vorpals_{chunk} has fewer extraneous orderings relative to Vorpals, the demarcation of chunks based on controller memory regions creates false orderings. Below we propose Vorpals_{chunk+} which provides a way to remove these extraneous orderings through the use of *sub-chunks*.

In Vorpals_{chunk+}, cores do not create chunk boundaries between persistent stores issued to different memory controllers — a chunk spans all stores between two persist ordering instructions. Instead, in Vorpals_{chunk+}, within a chunk, cores create *sub-chunks* when they switch controllers. Sub-chunks of a given chunk all share a timestamp but are persisted at different memory controllers; thus a chunk is broken into a sequence of sub-chunks.

With regard to order creation, this modification is safe, as stores in different sub-chunks are still unordered with respect to each other in both clock-order and persists-before. However, enforcing the order becomes a problem as every component sub-chunk of some chunk C across all memory controllers needs to persist before any store of a subsequent chunk C' is persisted.

To solve this issue, we use a type of “chain notification”: for a given chunk, every controller that persists a sub-chunk notifies the controller of the next sub-chunk when it finishes persisting but *does not* increment its local progress clock. The controller that persists the final sub-chunk in a chunk it increments its progress clock (eventually broadcasting this new timestamp to the other controllers). The effect of this chained notification is that the final memory controller effectively persisted all the entire chunk.

To implement this optimization, each sub-chunk contains a unique id, along with the ids of its successor and predecessor sub-chunks within the original chunk. Since chunks are issued by a single thread, sub-chunks can be ordered by the thread’s executed program order. A single integer register, along with a unique core id, suffices for the id.

Vorpals_{chunk+} also correctly achieves order enforcement: if two stores are clock-ordered, the first’s chunk will entirely complete across all memory controllers before any progress clock is incremented, and, by the immediate successor property, the second’s chunk must wait until the first’s chunk completes; otherwise the proof of order enforcement is the same. Similarly, the liveness arguments of Vorpals_{chunk} directly apply.

6 VORPAL_H: HIERARCHY-BASED VECTOR CLOCK ORDERING

The above Vorpals variants capture and enforce the persist-before order with few false orderings, however, they are unlikely to scale to a many-socket machine because clock and timestamp size grows in proportion to the number of cores in the system. Since the algorithms require a clock entry per core, the vector clocks can get quite large and, more problematically, they ignore the organizational structure of the machine (see Figure 1).

In this section, we introduce a hierarchical variant of Vorpals, Vorpals_H. Vorpals_H uses a two-level clock hierarchy — a *global clock* across sockets and many socket-specific *local clocks* across the local cores. A global clock contains an entry for each socket in the system, while the clock size for the socket-specific local clock is the number of cores inside the socket. The global clock for the

```

44 // Socketp Gateway: State and Events (Order Construction)
45 clock_register L[N]; clock_register G;
46 typedef enum type = {LOCAL, GLOBAL}; type chunk_type = GLOBAL;
47 On: Received a first store event S after load-acq, persistent fence, or controller-switch
48 from local corei
49 if(no remote stores or updates to G since this event fired last){L[i] + +;}
50 else{Gp + +;}
51 type new_chunk_type := (destination_socket(S)==p)?LOCAL:GLOBAL;
52 if(chunk_type==LOCAL ^ new_chunk_type==GLOBAL)
53   Notify core to dispatch chunk and mark as last local chunk
54 else
55   Notify core to dispatch chunk
56   chunk_type:=new_chunk_type
57 On: Received a store-rel S from local corei
58   ViS := (L[i], G, p);
59 On: Received a load-acq A from local corei that synchronizes-with local store-rel S
60   L[i] := L[i] ⊕ ViS;
61 On: Received a load acq A from local corei that synchronizes-with remote store-rel S
62   G := G ⊕ ViS;
63 On: Received a clock request from local corei to be assigned to a chunk
64   Reply with clock (L[i], G, p);
65 // Core state and Events (Order Construction)
66 Integer ChunkCount; Set<Store> Chunk;
67 On: Any load-acquire, store-rel, persistent fence
68 or when a persistent store switches controller or on a first store after load-acq
69   Forward the event to the gateway
70 On: Received gateway notification to dispatch chunk (bool mark)
71   (Vi, Vg) := RequestClocksFromGateway();
72   if(mark){mark_as_last_local(Chunk);}
73   {Issue Chunk w/ ChunkCount to MCs Pending Set;}
74 On: Issue persistent store S
75   Add S to Chunk; ChunkCount++;
76 On: Occasionally
77   Issue persist fence;
78 // MC State and Events (Order Enforcement)
79 clock_register Pg; clock_register Pi;
80 On: Occasionally complete chunks
81   if(∃Chunk Chunk ∈ Pending
82     ∧ Chunk ∉ Persisted ∧ immediately_succeeds(Pi, ViChunk)
83     ∧ immediately_succeeds(Pg, VgChunk) ∧ is_from_local_socket(Chunk))
84     Persisted := Persisted ∪ Chunk; Pi := Pi ⊕ ViChunk;
85     if(is_last_local(Chunk))
86       {Persisted := Persisted ∪ Chunk; Pg := Pg ⊕ VgChunk;}
87   }
88   if(∃Chunk Chunk ∈ Pending ∧ Chunk ∉ Persisted
89     ∧ immediately_succeeds(Pg, VgChunk) ∧ is_from_local_socket(Chunk))
90     Persisted := Persisted ∪ Chunk; Pg := Pg ⊕ VgChunk;
91 On: Received broadcast clock Bi
92   Pi := Pi ⊕ Bi;
93 On: Received broadcast clock Bg
94   Pg := Pg ⊕ Bg;
95 On: Occasionally
96   Broadcast both Pi and Pg;

```

Figure 6: Vorpals_H: The pseudo-code for Vorpals_H, a vector-clock hierarchy-based Vorpals algorithm designed for large multi-socket systems. Actions of the cores, gateways and the memory controllers are shown.

socket subsumes all local clocks in that same socket. Using hierarchy, Vorpals_H achieves a reduction in vector clock sizes, a task necessary to scale persistence ordering to large-scale systems. As the algorithm needs to capture messages heading into and out of the socket, Vorpals_H leverages the socket’s single gateway for processing messages across the server’s FIFO inter-socket network.

Order Construction. In Vorpals_H, each store is labeled using a *hierarchical vector clock* with a *hierarchical timestamp*. This more detailed timestamp consists of two vector clock times (the global

and local clock), along with the store's socket of origin. We write the hierarchical timestamp U originating at socket p as the tuple (U_l, U_g, p) . Hierarchical timestamps generate a *hierarchical clock order*, written $\leq_H$. Given two timestamps $U = (U_l, U_g, p)$ and $V = (V_l, V_g, q)$, the order $(\leq_H)$ between the timestamps can be stated as follows: $(U_l, U_g) \leq_H (V_l, V_g) \iff (U_g < V_g) \text{ or } (U_g = V_g \text{ and } U_l < V_l \text{ and } p = q)$. Vorpals_H uses the per-socket gateway to assign global and local timestamps to store instructions of that socket. The gateway monitors the traffic into and out of its socket and is, therefore, a useful location for managing the clocks: the gateway contains a clock register (G) for the global clock and local clock registers ($L[N]$) for every one of the N cores of the socket.

The overall structure of Vorpals_H is similar to the non-hierarchical variants. However, in Vorpals_H, cores do not keep their own clocks locally, but instead have a clock on the gateway – the gateway serves as a serializing point for all chunks within the socket. A Vorpals_H core queries its gateway for copies of the global and its local clock. Upon receiving the clocks, a core sends the chunks it generates to the corresponding controllers along with their timestamps. The controllers then persist the chunks respecting the $\leq_H$ order, as specified by the hierarchical vector order.

Figure 6 shows the Vorpals_H algorithm. Like Vorpals_{chunk}, Vorpals_H groups together unordered stores into chunks, that all have the same timestamp. Like Vorpals_{chunk}, the boundaries between chunks are created at memory controller switches, persist fences, or load-acquire synchronization events. Vorpals_H further differentiates between *local chunks*, whose stores are persisted by a memory controller in the local socket, and *global chunks*, that are persisted by remote memory controllers.

As the socket's gateway participates in Vorpals_H on behalf of the cores of that socket – cores must query the gateway for their clocks (line 71). The constructed order as captured in the local and global clock registers is used to make the local and global clock assignments to the chunks at the cores (line 64 and line 73) and to construct a per-socket total order across chunks. Vorpals_H uses this total order to issue timestamps based on the adjacent chunk type. When two successive chunks (in the total order serialized by the socket's gateway) are local, the algorithm uses the local clock registers (lines 49, 58 and 60) to order them, thereby avoiding publishing the ordering beyond the socket. In contrast, for any two successive chunks (again, within the socket's total order) in which one chunk is global, the algorithm orders them by the global clock (lines 49, 58 and 62). These timestamp rules on successive chunks mean the gateway oscillates between issuing global chunks, each with increasing global timestamps, and local chunks, each with an increasing local timestamp but common global timestamp (line 46).

By reasoning about local and global chunks, we can demonstrate that Vorpals_H's order construction is correct. Consider any two stores that are adjacent in persist-before. Their ordering is induced by either a persist fence or persist-before ordering induced through the release-acquire order. Their corresponding chunks are also either local or global. If both are local, the ordering instruction (the persist fence or load-acquire) will result in ordering by the local vector clocks in the gateway (line 49) before the second store. If either one is or both are global, the intervening ordering instruction will result in ordering by the global vector clock (line 50) before

Algorithms	At Core	At Memory Controller	At Gateway	Per store/store-rel or chunk
Vorpals ₀	(C)	(C)	None	(C)
Vorpals _{chunk}	(C)	(C)	None	(C + CT)
Vorpals _{chunk+}	(C)	(C + CI)	None	(C + CT + 3 * CI)
Vorpals _H	None	(C _{socket} + S)	(C _{socket} × C _{socket} + S)	(S + MB) - inter-socket (C _{socket} + S) - intra-socket

Table 1: Metadata sizes (clocks and timestamps). C is the number of cores in the system, S is the number of sockets, C_{socket} is the number of cores per socket, M is the number of memory controllers in the system, CT is the size of the “chunk count” variable required in Vorpals_{chunk} and Vorpals_{chunk+}, CI is the size needed for the “chunk id” in Vorpals_{chunk+}, and MB is the mark bit whose size is 1 used by Vorpals_H.

the second store. In either case, the stores are ordered by the clock order $\leq_H$.

Order Enforcement. Order enforcement in Vorpals_H takes advantage of the fact that its clock-order is *nested*, that is, two or more local chunks may carry the same global clock but are ordered by their local clock.

For enforcement, Vorpals_H maintains two progress registers at a controller, one for the local clock (P_l) and another for global clocks (P_g). The controller completes the global chunks in global clock order (line 89) and the local chunks nested within the global chunk in their local clock order (line 83), updating the global and local progress registers respectively. When persisting local chunks, we delay updating the global progress clock until all nested local chunks that share a global timestamp are persisted. We accomplish this tracking using a single bit to *mark* the final local chunk before a global one (lines 53, 72, and 85).

Reasoning about global and local chunks also demonstrates that Vorpals_H's order enforcement is correct. Using the safety proof of Vorpals_{chunk}, global chunks (and their stores) are correctly persisted with respect to other global chunks. Similarly, local chunks (and their stores) are persisted in the correct order with respect to other local chunks. Furthermore, local chunks are ordered with respect to subsequent global chunks through the marking mechanism, which guarantees an earlier local chunk persists before any subsequent global chunk, and persists after earlier global chunks by the immediate succession check on the global progress clock.

Liveness. Like all other Vorpals algorithms, Vorpals_H is live. The liveness proof of Vorpals₀ applies verbatim here, substituting “chunk” for “store”, and running the proof twice: first to show that global chunks are live, then to show that local chunks are live.

7 RESULTS

Metadata cost analysis. Vorpals incurs a metadata cost both in the clock registers at cores memory controllers, and gateways, and also in the timestamp sizes for different messages. Table 1 summarizes the overall metadata costs for different Vorpals algorithms.

With regard to the metadata cost for stores and chunks, Vorpals_H not only reduced the effective size of messages by a factor C (number of cores) to a factor S (number of sockets) compared to Vorpals₀, but also enables many of the Vorpals MC broadcasts to be restricted inside the socket. We observe a 56% average reduction in the number

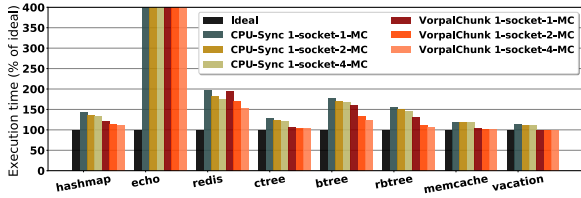


Figure 7: Scalability of Vorpalsync.

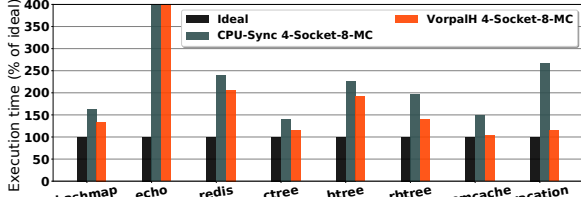


Figure 8: Scalability of Vorpalsync.

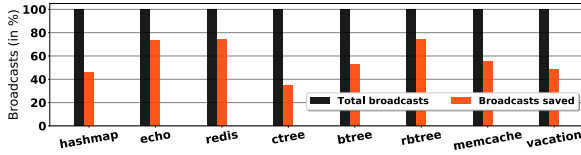


Figure 9: Broadcasts saved by Vorpalsync versus Vorpalsync.

of broadcasts. Figure 9 shows the percentage broadcasts saved by the individual benchmarks (relative to the broadcasts in Vorpalsync).

Performance and scalability analysis. We implemented our algorithm as a shared memory system on top of the MARSSx86 [34] simulator to model our performance. The simulator models an out-of-order core running at 4GHz. To test our solution, we use applications from the persistent memory Whisper suite for our evaluation [32]. To model a non-volatile memory with slower write speed, we set the NVMM write latency to 600 cycles [11, 32], and set the NVMM read and DRAM read/write latency to 150 cycles. Like ARM [4], our model does not assume that the memory controllers have access to residual power and will not be flushed to persistence on failure (in contrast to Intel’s ADR [39] domain for Optane DC). For comparison, we modeled a non-distributed ordering implementation, where each core coordinates its own persistent ordering via synchronous messages with one memory controller at a time (similar to Kolli et al. [22]). We call it **CPU-Sync**, and we compare it against the Vorpalsync implementations. CPU-Sync requires synchronous messages between CPU and MC to enforce every ARP generated order. In contrast, we also implemented **Ideal**. Ideal witnesses no persistence-induced CPU stalls, i.e. the persistent stores are assumed to complete instantaneously. The Ideal baseline can be considered the best case scenario for persistence wherein the performance penalty of enforcing the order is zero. To evaluate the effect of controllers we vary the number of total number memory controllers as well as the number of sockets (e.g. **Vorpalsync 4-Socket-8-MC** has four sockets with two MCs each). We assume a 200 ns delay per inter-socket hop and topology with a single socket connected to at most two other sockets. Writes to the persistent memory generated by the stores of a given core are sent to the memory controllers at the granularity of a cache line. Each MC has a bounded length queue of 64 entries, each the size of a cache line. Each entry also stores the vector clock metadata

for the line. We assume a 4 KB page size and spread consecutive pages across the controllers in a round robin fashion, as expected on interleaved NVMM [19]. All the figures show the average over eight runs — we did not notice any large variation.

Figure 7 Vorpalsync, compared to CPU-Sync and Ideal, varying the number of memory controllers. Vorpalsync’s performance approaches that of Ideal with increasing controller count, by being able to exploit the parallelism offered by multiple memory controllers and deferred completion enabled by vector clocks, as opposed to the synchronous scheme. Most workloads benefit significantly from the proposed scheme (except for echo due to high store intensity).

In Figure 8 we show the results of Vorpalsync for a multi-socket setting with additional hop latencies. Large slowdowns relative to Ideal are witnessed here for CPU-Sync. Compared to the slowdowns on a single socket system (Figure 7), the slowdowns witnessed are much higher, with slowdowns over 2X. Vorpalsync due to its latency tolerance provides significant benefits in this setting.

8 RELATED WORK

Memory Persistency. In anticipation of the arrival of persistent memory devices, the research community has undertaken a significant effort in preparing various components of the system stack (from cores to caches to memory controllers) for their arrival [20, 22, 24]. Vorpalsync sits alongside a number of architectural solutions for enforcing the persists-before order. Most existing systems enforce persist orderings using synchronous communication [4, 20–23, 32, 35, 40, 41]. None of these systems scales to tens of MCs. Since ordering is enforced with some form of enqueue and acknowledgement messages, messages introduce slow serializing communication. Our design specifically deals with the challenge of scaling with multiple MCs through a distributed algorithm.

Alongside the architectural community, the theory community has explored what persistent memory means at the language level, investigating novel persistency models [35] for specifying the persists-before order, e.g. [17, 18, 37]. Other work has explored systems software, such as transactional memory-type abstractions [7, 16, 27, 45], file systems [47], and data structures [8, 13, 33, 44].

Vector Clocks. Vector clocks were introduced independently in 1988 by Mattern [30] and Fidge [12] as a method of ordering events in a distributed setting. In this manner, the clocks improved on the original scalar clocks of Lamport [26], which could not represent the partially ordered relationship between distributed events. Other methods of tracking logical time, including matrix time, have been proposed, for a summary of methods see [43]. The size of vector clocks has repeatedly been a concern, resulting in various methods of shrinking them [25, 42].

Causal Consistency. In a causal consistency model, observers cannot see a data item before they can see the data items that influenced its creation, a notion that shows some similarities with ARP. Causal consistency has drawn attention recently as it is the strongest consistency model available in the presence of partitions [29]. Causal memory [2] evolved from applying the notion of the causal ordering of events in a distributed system [26] to reads and writes. Enforcing causal consistency (for a summary, see [5]) in a distributed system has some similarities to our Vorpalsync algorithms

enforcement of persist-before. Often, causal consistency is tracked using logical clocks (such as vector clocks) (e.g. [9, 10, 28]), and may even be hierarchical [6].

9 CONCLUSION

This paper addresses the problem of ensuring the ordering of stores on large-scale persistent memory systems. Previous solutions do not scale with core and memory controller count. In this paper, we proposed a distributed solution to the problem of persistent ordering using vector clocks. Our algorithms provide both scalability and performance (with an average speedup of 48%), while overcoming the limitations of applying vector clocks in this setting. We plan to explore even better algorithms and reflect on how the proposed implementations may impact persistent memory semantics as part of future work, for instance, leveraging other work in compressing vector clocks or combining the vector registers between cores.

REFERENCES

- [1] Release-acquire ordering, 2017. https://en.cppreference.com/w/cpp/atomic/memory_order#Release-Acquire_ordering.
- [2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1): 37–49, Mar 1995.
- [3] ARM. Armv8 reference manual, . <https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile>.
- [4] ARM. ARMv8-A architecture evolution, persistent memory instructions, . <https://community.arm.com/processors/b/blog/posts/armv8-a-architecture-evolution>.
- [5] M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. Rodrigues. On the use of clocks to enforce consistency in the cloud. *Data Engineering*, p. 18, 2015.
- [6] M. Bravo, L. Rodrigues, and P. Van Roy. Towards a scalable, distributed metadata service for causal consistency under partial geo-replication. In *Doctoral Symp. of the 16th Intl. Middleware Conf.*, Middleware Doct Symp. '15, pp. 5:1–5:4, Vancouver, BC, Canada, Dec. 2015.
- [7] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pp. 105–118, Newport Beach, CA, Mar. 2011.
- [8] N. Cohen, D. T. Aksun, and J. R. Larus. Object-oriented recovery for non-volatile memory. *ACM Program. Lang.*, 2(OOPSLA):153:1–153:22, Oct. 2018.
- [9] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *4th Ann. Symp. on Cloud Computing*, SOCC '13, pp. 11:1–11:14, Santa Clara, CA, Oct. 2013.
- [10] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *ACM Symp. on Cloud Computing*, SOCC '14, pp. 4:1–4:13, Seattle, WA, Nov. 2014.
- [11] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data tiering in heterogeneous memory systems. In *11th European Conf. on Computer Systems*, EuroSys '16, pp. 15:1–15:16, London, UK, Apr. 2016.
- [12] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conf.*, pp. 55–66, 1988.
- [13] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. In *23rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '18, pp. 28–40, Vienna, Austria, Feb. 2018.
- [14] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th Ann. Intl. Symp. on Computer Architecture*, ISCA '90, pp. 15–26, Seattle, WA, June 1990.
- [15] Intel. Xeon servers, Intel, 2019. <https://www.intel.com/content/www/us/en/processors/xeon/scalable/xeon-scalable-platform-brief.html>.
- [16] J. Izraelevitz, T. Kelly, and A. Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pp. 427–442, Atlanta, GA, 2016.
- [17] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *30th Intl. Conf. on Distributed Computing*, DISC '16, pp. 313–327, Paris, France, Sept. 2016.
- [18] J. Izraelevitz, H. Mendes, and M. L. Scott. Brief announcement: Preserving happens-before in persistent memory. In *28th ACM Symp. on Parallelism in Algorithms and Architectures*, SPAA'16, pp. 157–159, Asilomar Beach, CA, 2016.
- [19] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the Intel Optane DC persistent memory module, Apr. 2019.
- [20] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Efficient persist barriers for multicores. In *48th Intl. Symp. on Microarchitecture*, MICRO-48, pp. 660–671, Waikiki, HI, Dec. 2015.
- [21] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-performance transactions for persistent memories. In *21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pp. 399–411, Atlanta, GA, 2016.
- [22] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. Delegated persist ordering. In *49th Ann. IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, pp. 1–13, Taipei, Taiwan, Oct 2016.
- [23] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch. Language-level persistency. In *44th Ann. Intl. Symp. on Computer Architecture*, ISCA '17, pp. 481–493, Toronto, ON, Canada, June 2017.
- [24] K. Korgaonkar, I. Bhati, H. Liu, J. Gaur, S. Manipatruni, S. Subramoney, T. Karnik, S. Swanson, I. Young, and H. Wang. Density tradeoffs of non-volatile memory as a replacement for sram based last level cache. In *45th ACM/IEEE Ann. Intl. Symp. on Computer Architecture (ISCA)*, pp. 315–327, Los Angeles, CA, USA, June 2018.
- [25] A. D. Kshemkalyani, A. Khokhar, and M. Shen. Encoded vector clock: Using primes to characterize causality in distributed systems. In *19th Intl. Conf. on Distributed Computing and Networking*, ICDCN '18, pp. 12:1–12:8, Varanasi, India, Jan. 2018.
- [26] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
- [27] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *51st Ann. IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, pp. 258–270, Fukuoka City, Japan, Oct 2018.
- [28] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *23rd ACM Symp. on Operating Systems Principles*, SOSP '11, pp. 401–416, Cascais, Portugal, Oct. 2011.
- [29] P. Mahajan, L. Alvisi, M. Dahlin, et al. Consistency, availability, and convergence.
- [30] F. Marten. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pp. 215–226, 1988.
- [31] Micron. 3D XPoint Technology, 2017. <http://www.micron.com/products/advanced-solutions/3d-xpoint-technology>.
- [32] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. An analysis of persistent memory use with WHISPER. In *22nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pp. 135–148, Xi'an, China, Apr. 2017.
- [33] F. Nawab, J. Izraelevitz, T. Kelly, C. B. M. III, D. R. Chakrabarti, and M. L. Scott. Dali: A periodically persistent hash map. In *31st Intl. Symp. on Distributed Computing (DISC 2017)*, volume 91, pp. 37:1–37:16, Vienna, Austria, Oct. 2017.
- [34] A. Patel, F. Afram, S. Chen, and K. Ghose. Marss: A full system simulator for multicore x86 CPUs. In *48th Design Automation Conf.*, DAC '11, pp. 1050–1055, San Diego, CA, 2011.
- [35] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *41st Ann. Intl. Symp. on Computer Architecture*, ISCA '14, pp. 265–276, Minneapolis, MN, 2014.
- [36] pmem.io. Persistent Memory Development Kit, 2017. <http://pmem.io/pmdk>.
- [37] A. Raad and V. Vafeiadis. Persistence semantics for weak memory: Integrating epoch persistency with the TSO memory model. *ACM Program. Lang.*, 2(OOPSLA): 137:1–137:27, Oct. 2018.
- [38] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev.*, 52(4):465–479, July 2008.
- [39] A. Rudoff. Deprecating the pcommit instruction, 2016. <https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction>. Accessed 04/18/2019.
- [40] A. Rudoff. Processor support for NVM programming. http://www.snia.org/sites/default/files/AndyRudoff_Processor_Support_NVM.pdf.
- [41] S. Shin, J. Tuck, and Y. Solihin. Hiding the long latency of persist barriers using speculative execution. In *44th Ann. Intl. Symp. on Computer Architecture*, ISCA '17, pp. 175–186, Toronto, ON, Canada, June 2017.
- [42] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43(1):47 – 52, 1992.
- [43] M. Singhal and M. Raynal. Logical time: Capturing causality in distributed systems. *Computer*, 29:49–56, 02 1996.
- [44] S. Venkataraman, N. Tolia, P. Ranganathan, and R. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *9th USENIX Conf. on File and Storage Technologies*, FAST '11, San Jose, CA, February 2011.
- [45] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pp. 91–104, Newport Beach, CA, 2011.
- [46] H. S. P. Wong, H. Y. Lee, S. Yu, Y. S. Chen, Y. Wu, P. S. Chen, B. Lee, F. T. Chen, and M. J. Tsai. Metal oxide RRAM. *IEEE*, 100(6):1951–1970, June 2012.
- [47] J. Xu and S. Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conf. on File and Storage Technologies (FAST 16)*, pp. 323–338, Santa Clara, CA, Feb. 2016.