X-SSD: A Storage System with Native Support for Database Logging and Replication

Sangjin Lee*
Hanyang University
Republic of Korea

Alberto Lerner
University of Fribourg
Switzerland

André Ryser
University of Fribourg
Switzerland

Kibin Park
Hanyang University
Republic of Korea

Chanyoung Jeon
Hanyang University
Republic of Korea

Jinsub Park
Hanyang University
Republic of Korea

Yong Ho Song
Hanyang University &
Samsung Electronics
Republic of Korea

Philippe Cudré-Mauroux
University of Fribourg
Switzerland

ABSTRACT

Transaction logging and log shipping are standard techniques to provide recoverability and high availability in data management systems. They entail an update to a local log file and a remote site at every transaction. Modern databases have leveraged technologies such as Persistent Memory (PM) and RDMA-enabled networking to perform these updates as fast as possible. This mix of technologies, however, presents several drawbacks: lack of portability, the complexity of the data path, and interoperability.

To address these issues, this paper introduces the X-SSD, a new SSD architecture that mixes NAND Flash and PM memory classes. A X-SSD device can take transaction log writes on a fast, PM-backed data path and be responsible for propagating the operation to remote sites and eventually to NAND Flash storage. We design and implement an actual reference X-SSD device called Villars to validate this new architecture. Our experiments show that the Villars device can offer a more straightforward and robust way to manage PM on behalf of the database and achieve equally fast results.

CCS CONCEPTS

• **Information systems** → **Database management system engines**; **Storage architectures**.

KEYWORDS

database-storage codesign, write-ahead log, database replication

ACM Reference Format:
Sangjin Lee, Alberto Lerner, André Ryser, Kibin Park, Chanyoung Jeon, Jinsub Park, Yong Ho Song, and Philippe Cudré-Mauroux. 2022. X-SSD: A Storage System with Native Support for Database Logging and Replication. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3526188

*The author performed most of the work while visiting the University of Fribourg.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526188

1 INTRODUCTION

Database replication is often performed by copying transactions' changes into a secondary site before committing these changes into local storage [41, 62]. Such a mechanism is called *(transaction) log shipping* and is present almost universally in databases that offer replication, (e.g., [3, 56, 63]). If the primary database site goes down, the secondary one can serve as a hot backup, as it caught up with all the primary database changes. Achieving this level of robustness, however, comes at a cost. Transaction logging and log shipping require writing to storage and exchanging data over the network, both relatively expensive operations.

Two technologies reached maturity recently that can be relevant in this scenario. The first one is Persistent Memory (PM), and more specifically, PM in a DIMM form factor that replaces server memory and can be accessed by an application via `load` and `store` instructions. PM comes in many flavors such as Intel Optane [31] or battery-backed DRAM [16][1]. Optane class PM has for instance proved to be useful in mixed memory indices [4, 50, 57], and can offer alternative ways to build a database system [5]. Battery-backet class PM behaves as regular DRAM but is not volatile. The second technology is RDMA-enabled networks [30]. These networks transport data with negligible overhead and have been useful, for instance, in query execution [22, 49, 64]. Just as with PM, RDMA-enabled networks have also fostered new database designs [11].

PM and RDMA-enabled networks can also help to record and propagate transaction log updates [75, 78]. In particular, we consider the case of *Main-Memory Databases* [19]. They can reach unprecedented performance levels because they maintain all their data in DRAM and persist only the transaction log, which therefore becomes their main bottleneck [51]. Figure 1 (left) depicts how a typical system can perform log writing and shipping with PM and RDMA. We can observe in the figure that the database system is responsible for coordinating several different steps, sometimes targeting local PM, sometimes remote PM or memory via an RDMA-enabled NIC, and lastly, fast SSD devices.

Each of these technologies offers a specific API and presents some restrictions. The combination of these restrictions creates a number of issues, including:

- The interaction of RDMA and PM is complex and poorly understood. For example, using RDMA to update a PM-backed address on a remote machine may make the update *visible*, but it does not

[1]The JEDEC standard, which supports DRAM interoperability, calls these NVDIMM-P and NVDIMM-F types of persistent memory, respectively [23].

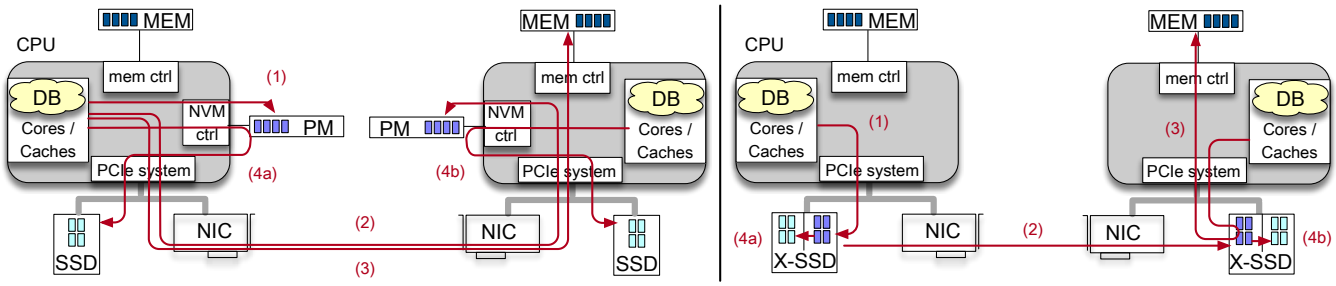

Figure 1: (Left) Logging and replication path using PM and RDMA. (1) The database writes log data into the PM. (2) It then ships the data to remote PM via RDMA. (3) It uses a second RDMA operation to make the change the log describes into the remote host's memory (e.g., using Active-Memory techniques [78]). Eventually, both hosts will need to make space on PM. (4a/b) They do so by copying some of its contents into an SSD. (Right) Logging and replication path using a X-SSD device. The sequence of steps is the same, but the X-SSD device takes responsibility for propagating data in steps (2) and (4a/b), while the update of the remote memory is done by the remote Database (3).

guarantee that that update is *persistent* [37]. If a machine crashes, a replication operation's correctness can be compromised.
- While PM can be accessed with simple `load`/`store` memory instructions, programming correct, persistent data structures is a daunting endeavor. A software crash can leave a structure in an arbitrary state, from which the database then needs to recover.
- Every DIMM slot used for PM is not used for DRAM. This forces the system designers to choose between DRAM or PM capacity.
- Optane and battery-backed DRAM require specific server support and cannot be ported across servers without certain characteristics. Optane, in particular, is not supported on AMD platforms.

To address these issues, this paper presents a new SSD design that allows database logging and replication to benefit from PM and fast networking, but without the above drawbacks. In summary, our design is based on a deceptively simple decision: it moves PM out of the CPU path and into the SSD, and it lets the latter manage the access to PM, locally or remotely, on behalf of the database. Specifically, we devise a new storage architecture that contains PM- and NAND Flash-based storage that is natively networked. The architecture provides a separate, fast data path and interface fully dedicated to transaction log writes and offers *Data Propagation Services*, including across servers, upon which database replication can be built. We call our storage architecture the X-SSD[2]. Figure 1 (right) shows how the logging and replication data paths can be simplified with a X-SSD device.

Moving PM into a X-SSD device frees DIMM slots for DRAM and restores the ability to deploy PM on vendor-independent server-class machines without special-purpose DIMM slots or battery-backing features. One can then use PM on an AMD server simply by plugging in this new NVMe device. The device also avoids interoperability problems between RDMA and remote PM. These problems occur because the RDMA writes may be routed to the CPU caches in a process called Data-Direct IO (DDIO) [21], before they reach the PM. Our system can resort to low-level mechanisms to request the NIC to deliver messages directly to storage, which would be impractical at the application level.

[2]The 'X' stands for "cross" for reasons that will become apparent shortly.

We design and implement a X-SSD device using an actual SSD prototyping platform [44]. We call this device Villars. The Villars device is fully compatible with the NVMe standard [55], the *de facto* standard for fast SSDs, even with our extensions. The Villars is the first in what we expect to be a series of X-SSD devices with increasing application functionality. We also provide a set of drop-in system call replacements, e.g, `pwrite()`, that make it easy to convert existing code to use our device. These syscall replacements can detect, with low overhead, when a previous write is persistent or is in-processing within a local device or a remote device, allowing the database to implement different replication flavours.

In summary, we make the following contributions:
- We propose a new SSD architecture that mixes PM and NAND Flash storage in a way that naturally matches transaction logging and replication behavior (§ 3).
- We describe a reference design of our architecture that presents precise durability semantics to the application (§ 4).
- We describe how to integrate data propagation services in an existing database as well as suggest how a new database can explore alternative designs (§ 5).
- We quantify the benefits of shifting data movement (across memory types and local and remote servers) to the storage, freeing application cycles in the process (§ 6).
- We present several use cases that can benefit from X-SSD devices' existing and potential features (§ 7).
- We position our solution relative to the state of the art (§ 8).

Before describing our architecture in detail, we start below by presenting the background necessary to follow our discussions.

2 BACKGROUND

The X-SSD device extends traditional SSDs with data propagation services exposed through a new interface. To understand the implications of such an extension, we revisit how data flows between a database and a storage device (§ 2.1) and, once it reached the latter, the path the data follows within a conventional SSD (§ 2.2). Lastly, we introduce two standard but little-known technologies called CMB and NTB upon which our architecture is based (§ 2.3).

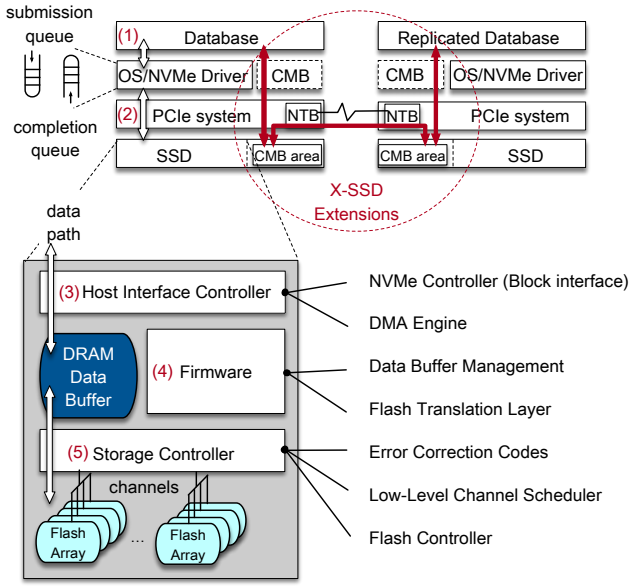

Figure 2: Top: Hardware and software stacks between a Database and an SSD. (1) The Database requests storage services through the NVMe driver. (2) The driver uses the PCIe system to send control and data requests to an SSD. CMB opens a second communication path—still using PCIe but now via memory mapping. Bottom: A traditional SSD Architecture (adapted from [44]). (3) The Host Interface Controller receives the host's commands through NVMe or the CMB feature. (4) The Firmware coordinates several aspects of the device, such as the scheduling of operations, the management of the Data Buffer, or the mapping of physical to logical pages. (5) The Storage controller interacts with the Flash Arrays and deals with aspects such as low-level operations scheduling and error correction. Note that the PCIe subsystems of different hosts can be interconnected via a technique called NTB.

2.1 Connecting the Database and SSDs

Today's main "conduit" between devices and the OS/applications is a standard protocol called NVMe [55]. Mainly, NVMe determines how commands to a device should be issued and how data is transferred. As these transfers are commonly performed in fixed-sized blocks, this kind of device is said to be a *block device*. A typical software stack that uses NVMe devices is depicted in Figure 2 (top).

When the OS wants to send work to the device, it encodes it as an NVMe (read or write) command and places it in a command submission queue shared with the device. The OS signals the device whenever it adds new commands through a mechanism called a *doorbell*. In turn, the device transfers data in and out of the OS cache (or application areas in case of Direct IO). When a transfer is finished, the device adds a *command completion* entry to a completion queue and notifies the NVMe driver via an interrupt.

Applications need not interact with NVMe devices directly—at least, that is the standard practice. They interact with the OS instead via calls such as `pread()` and `pwrite()`, which abstract away the details of the communication using a file-system interface.

The PCIe subsystem is the backbone interconnecting the host's memory and peripheral devices [33]. This subsystem has a long history. In its initial forms, called PCI and PCI-X, the subsystem was strictly a peripheral bus. As a consequence, only one device could be using it at a time. The protocol has dramatically changed over time, and PCIe became a family of protocol layers that communicate through packets. In other words, PCI may have been a bus, but PCIe is a full-fledged networking system. Each device receives an address based on the slot it is connected to, while the CPU receives a unique address.

The packets in this network are called Transaction Layer Packets (TLPs) and mainly carry read or write operation requests. These operations allow hosts and devices to access each other memory regions. For example, an NVMe device can read the submission queue in the host via TLP packets. It can also transfer the data to which the commands refer in the same way.

2.2 The Life of a Log Write

An SSD is usually divided into three main subsystems. The top subsystem is the Host Interface Controller (HIC), which implements the NVMe protocol and PCIe messaging. The bottom subsystem is the Storage Controller, which manipulates actual Flash arrays. We call the middle subsystem Firmware[3]; it performs several tasks necessary to convert NVMe commands into Flash operations. These subsystems are depicted in Figure 2 (bottom).

We explain how these components interact by following the path a log write takes inside an SSD. A log write starts as an NVMe command sitting on the OS driver submission queue. The HIC is capable of fetching these commands and recognizing the NVMe vocabulary. Given that the log command is a write, the HIC uses a *Direct Memory Access* (DMA) engine to bring the data into the device. The data is placed into a temporary Data Buffer area. It is very common for an SSD to cache data in this temporary area.

From this point on, the Firmware coordinates all aspects necessary to save the log record(s). Most notably, the Firmware runs the Flash Translation Layer (FTL), which is responsible for finding empty Flash page(s) in which to place the log data. By determining that page set, the Firmware chooses a physical address to save the data. However, the actual Flash write is yet to take place.

The Flash Storage Controller (FSC) monitors when the Flash Array units become available for work and keeps them busy with ongoing requests. In the case of our log write, the FSC checks if no other operation is using the destination Flash array, which contains many pages. Once the FSC finds the opportunity, it moves the data to the proper array and issues the `Flash program` (write) operation.

Once the FSC finished the write operation, it signals the Firmware. In turn, the latter asks the HIC to place a command completion in another NVMe queue called the completion queue and issues a completion notification. This is achieved via an interrupt generated after the completion command is stored in the completion queue. By processing the interrupt, the NVMe driver understands that our log write is complete.

We observe that there is a long data path a write command should follow before its data can be persisted. Fortunately, there are ways to shorten this data path.

[3]In our device, this module is indeed implemented as embedded software.

2.3 CMB and NTB

The NVMe standard allows a device to optionally *expose* an internal memory area to applications via memory mapping (MMIO). This feature is called Controller Memory Buffer (CMB) in the standard [10]. Reads or writes performed against the area are automatically translated to PCIe messages directed towards the device. A X-SSD device uses CMB to expose a second, byte-addressable data path, in addition to the conventional block-based one.

Moreover, another relevant NVMe feature exists that is called Persistent Memory Region (PMR). It can expose yet another memory area but assumes the device persists the operations against that area. The difference between CMB and PMR is that the latter has additional configuration possibilities. For our purposes, we consider CMB and PMR as functionally equivalent and refer to them henceforth only by CMB.

Another technology that X-SSD devices utilize is offered by the PCIe system. As mentioned above, PCIe is a networking system, and therefore it also supports interconnecting different hosts' systems. This technology is called Non-Transparent Bridging (NTB) [29]. In practice, the devices on a server equipped with NTB can be seen by any other NTB-connected servers. NTB is little known, but it has long been supported in mainstream OSs.

Our interest in NTB comes from its lower latency and faster bandwidth when compared to other networking technologies [43, 52]. Unlike Ethernet or Infiniband network cards, NTB cards do not have to convert PCIe traffic into other packet types. PCIe itself is used for networking; there is no conversion overhead.

Speed is a welcome feature but our interest in using NTB has more to do with its simplicity. As we develop the hardware logic that is involved in the implementation of our device, we naturally have to deal with low-level aspects of PCIe. In other words, we already deal in logic that manipulates PCIe's TLP packets. Using NTB to bridge two PCIe systems involves very little additional effort, mainly address translations and sometimes minor formatting—but nothing as complex as what would be required to implement an RDMA stack client inside the device.

3 THE X-SSD ARCHITECTURE

The transaction log workload presents a unique set of challenges for storage as it is central to database performance and robustness. We can summarize these challenges as follows:

(A) Log writes must incur low-latency, since flushing log records is on the critical path of transactional systems [34];
(B) These writes must have clear semantics in crash scenarios in order to facilitate crash-consistent behaviors;
(C) Log writes are likely to be replicated and, therefore, remote data paths—potentially leveraging modern networking [30]—must be considered.

Using current state-of-the-art storage leaves the database designer with at least two choices for implementing fast logging and replication. She can use PM directly from the database, as Figure 1 (left) shows. As discussed above, this option offers low-latency writes (A), but brings a complex data path with several issues regarding (B) and (C). This option also sacrifices the ability to port the database to other servers that do not support the particular kind of PM adopted.

Alternatively, she can avoid PM in favor of traditional NVMe SSDs, such as the one in Figure 2 (bottom), by using many SSDs. This alternative offers equivalent throughput but not the latency advantages of PM. The setup to take advantage of many SSDs at once is somewhat complex (as it involves using some sort of IO framework), and may require specifically designed crash-consistent measures. Granted, this would be portable in that it would not depend on PM support. In short, both alternatives introduce compromises. We believe a new SSD architecture can bridge this gap.

The first goal of this new architecture is to combine the alternatives above, offering the best of both worlds. In other words, we seek a data path for transaction logs that, first and foremost, meets the three log workload challenges listed above. We believe that solving these challenges alone justifies a new storage design. However, a new design creates the possibility of solving other issues beyond performance and robustness. Specifically, we wish to address two additional problems: to gain back both the portability and ease of programmability that were lost by the introduction of PM. Therefore, we consider two additional goals as follows.

The second goal of the new architecture is to conform to existing standards as a way to foster portability. In particular, we believe that the NVMe standard [55] has proven flexible enough to support current and future hardware evolution trends. We cite as evidence the Open Channel SSDs [13] and Zoned Namespaces [12] efforts, which were developed under the standard. These efforts can be seen as ways to give applications better control over storage, which is, in spirit, what we are trying to achieve for database logging.

The third goal of our architecture is to find a balance between application control and programming difficulty. Currently, getting good performance out of modern storage requires the database architect to use and combine several complex libraries such as SPDK [77], NVML/PMDK [73], ibverbs [30], or io_uring [7], to name but a few. We believe that there ought to be a simpler way to give control over storage to the database architect, without burying that control under layers of APIs and requiring long learning curves.

3.1 The X-SSD Features

To achieve these goals, we propose an architecture that combines two distinct *sides* within the same device: a *conventional* side and a *fast* side. Figure 3 depicts the two sides and how they interact. The conventional side is an independent SSD device whereas the fast side can be seen as a high-performance staging area for append-only workloads. Our design rationale is to give applications a choice between two IO profiles but using a single device and making sure the profiles are seamlessly integrated.

We introduce the X-SSD architecture in more detail by describing its key features below.

Byte-addressable Interface for Append-only Workloads. The X-SSD offers a *second-interface* that complements that of a traditional Flash-based SSD. Block-device operations are treated normally by the latter. Byte-level operations are handled by the second, CMB-based interface. The CMB interface consists of an MMIO region that an application can access via `load` and `store` instructions. Some form of PM backs the CMB area. We evaluate the feasibility of two types of memory, SRAM and DRAM, made persistent by assuming battery backing (§ 4.1).

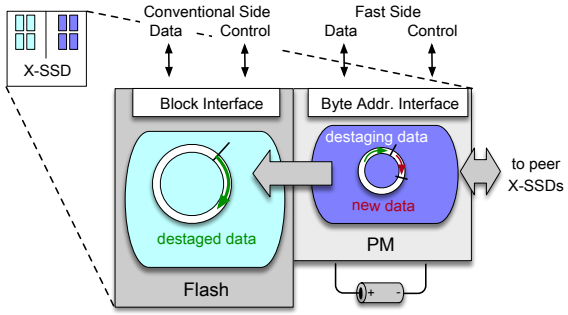

Figure 3: Logical X-SSD architecture. The device is divided into two *sides*, a Flash-based side and a PM-based side. The Flash device is primarily a conventional SSD. The PM device persists data coming from a byte-addressable interface on a circular buffer. It eventually destages that data to a designated circular buffer on the conventional side (and optionally to peer X-SSDs). The fast side is backed by capacitors that allow destage data even in a sudden power interruption. Therefore, fast-side writes can be acknowledged to the application before reaching the conventional side.

Data Propagation Services. The semantics of a write on the fast side is different from that on the conventional side. A fast write against CMB will be eventually *destaged* into the conventional side of the device in the same order it was issued (§ 4.3). The destaging occurs without any intervention from the application that issued the writes, as follows. The fast side is conceptually a ring to which the application writes. In the background, the device is constantly moving data from this ring onto the conventional side. The conventional side of a X-SSD device also maintains a destaging area in the shape of a ring, but much larger than the one on the fast side (cf. Figure 3). Optionally, that fast write can also be sent to Villars devices configured as secondary (replicas). We evaluate remote connections using PCIe NTB as an interconnect, but other networking technologies such as RDMA are possible (§ 4.2).

Crash Consistency. The battery-backing allows the device to offer a predictable behavior w.r.t. the data that was in transit should a sudden power interruption occur. The extra power supply allows the device to finish destaging any data present on the fast side. In practice, an application will see the data in transit during a crash on the conventional side of the device after a reboot (§ 4.1).

Logging Status Monitor. A X-SSD device provides a control interface used by applications to inquire about the progress of the data movement above. For example, an application can check whether a given write was already persisted on a remote X-SSD device (§ 4.2). Applications can use the provided conventional-style system calls replacements to obtain the same guarantees (§ 5.1).

Device Setup Interface. An additional control interface allows an application to configure the device (e.g., size of the PM area or the area on the conventional side used for long-term storage) and identify and connect to peer X-SSD devices. We also discuss how advanced setups allowing multi-threaded or multi-client use of X-SSD devices can be supported (§ 7).

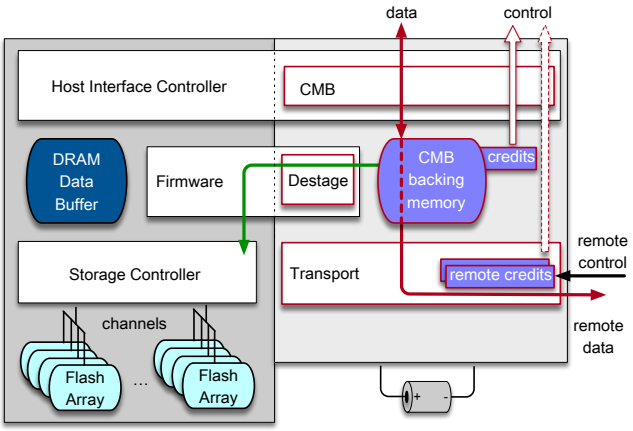

Figure 4: Physical X-SSD Architecture. The fast side consists of three modules (in red). The CMB module exposes the CMB backing memory to the database. The Destage module moves the latter's contents to the conventional side. The Transport module (on a primary device) propagates the fast writes to remote devices and monitors their progress.

4 VILLARS: A X-SSD REFERENCE DESIGN

This section introduces a reference design for the X-SSD architecture, which we call the Villars device. As mentioned above, we expect this to be only one in a series of devices that implement our architecture in a particular way. The design of the Villars device is geared towards meeting the goals described in Section 3. The design roughly consists of three modules, as Figure 4 shows: the CMB, Transport, and Destage modules. The CMB module is the top-level module, and it handles the byte-addressable interface that the application uses (§ 4.1). The Transport module is responsible for connecting to remote peer Villars devices and replicating the CMB writes stream across them (§ 4.2). The Destage module connects the two sides of a Villars device and, as its name implies, destages data from CMB's PM backing memory into the conventional side's NAND Flash (§ 4.3).

4.1 The CMB Module

The CMB module gets its name from a little-known NVMe standard feature [55] called Controller Memory Buffer (§ 2.3). The standard is liberal in how the area can be used. In the Villars device, the CMB module is backed with Persistent Memory, and the module allows the database to read and write to the backing area via an MMIO interface. Figure 5 depicts the CMB's internal components and its data and control paths.

Persistent Write Semantics. The CMB module receives application data as PCIe TLP packets. The module tries to make the path to persistence as short as possible, but it takes as many steps as necessary to achieve this safely. The arriving data is placed on an SRAM-based queue with a pre-negotiated size with the database. The importance of this queue's size will become clear shortly. The CMB module proactively dequeues elements and places them in the backing memory.

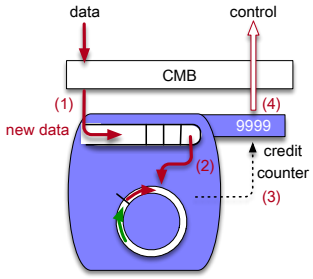

Figure 5: Data and control paths in the CMB module. (1) Data arriving from the PCIe system (CMB) is placed on a queue, (2) from which it is written to backing memory. (3) After reaching backing memory—but never before—a counter is incremented with the size of the data written. (4) This counter is available for the database to read (4).

The Villars device has two alternative implementations for the backing memory, one using SRAM and the other using DRAM. We assume that the device is guarded against power losses by super-capacitors [9]. Other types of memory could be easily supported, such as small-block Flash (e.g., Samsung's Z-NAND) and Storage Class Memory, such as Intel's Optane DC Memory.

Logically, the device treats the backing area as a ring and expects the database to write to the tail addresses of the area *mostly sequentially*. By mostly sequential, we mean that the Villars device can tolerate data arriving at the tail of the queue out of order within established bounds. Our decision follows some experiments' results in which the database sends one or very few bytes at a time through the interface. These bytes could arrive out of order. The real problem with this approach is the low performance obtained.

We, therefore, evaluated ways to have the application send chunks of bytes at once. An application can benefit from an existing chunking mechanism inside CPUs if the device classifies the CMB region to the system as a *Write Combining* (as opposed to an *Uncached*) region [32]. We discuss experiments in Section 6.1 in which writing in a few bytes' chunks using this mechanism improves the CMB performance considerably.

We offer the following semantics for writes: writes to a Villars device are considered persistent once they reach the backing memory's ring. The CMB module maintains and increments a *credit counter* with the number of bytes written to the ring. The counter and the queue's size are, as mentioned above, the cornerstone of a flow-control mechanism between the application and the device.

Credit-based Flow Control. The device may wish to exert *back-pressure* on the database on certain occasions. In other words, the database may be asked to pause writing until the previously accumulated data on the queue has become persistent, either locally or on a secondary host's device (for replication). To this end, the Villars device allows the database to monitor when its pending writes become persistent without necessarily resorting to calls such as `fsync()`. (In fact, the mechanisms we describe here can be used to implement an `fsync()` replacement (§ 5.1). The monitoring is done by exposing the credit counter via a control interface (also utilizing MMIO).

The mechanism works as follows. If there are enough outstanding writes to fill up the queue, whose size the database was notified about initially, the database should assume it needs to pause its writing momentarily until a portion of the queue gets persisted. For example, if the queue is 4096 bytes and the application has not yet written anything, it may write 4096 bytes without checking the credit counter. Once it finished writing, the counter may have advanced but not by the entire 4096 bytes; some writes may still be making their way through the data path. For the sake of this example, let us assume that the application reads the credit counter, and it comes back at 4000 bytes. It means that 96 (tail) bytes are still in flight. Therefore, the application can write at most 4000 bytes before rechecking the counter. The back-pressure mechanism is advisory and requires the application to adhere to the protocol if it wishes to benefit from the Villars device's guarantees.

Crash Consistency Behavior. The CMB module is responsible for implementing a *crash protocol*. When it detects that the device suffered a sudden loss of power, it uses the Destage module (§ 4.3) to destage the CMB ring in full. In practice, when the device (or database process) reboots, a certain additional amount of writes from the tail of the ring will appear on the conventional side. The device will stop Destaging if it encounters a gap in the data. This is consistent with the increment of the credit counter. The counter can only be incremented when contiguous chunks of data are formed.

4.2 The Transport Module

The Transport module is optional. When inactive, the Villars device works in stand-alone mode, which means only the CMB and Destaging modules are fully operational. When active on a primary Villars, the Transport module inserts itself into the data and control paths of the device. It does so to receive a mirror of the stream of writes directed at the fast side, and to ship these writes across to other X-SSD devices it considers its peers. Note that the data transfer across devices may be slower than the CMB intake, which is why the Transport module may also insert itself into the *back-pressure* mechanism we discussed above. Figure 6 shows both the data and back-pressure flows.

We assume the transport module has access to a network adapter card and is designed to shield the rest of the device from the networking details. For simplicity, we designed the Villars device's Transport module to use the PCIe system itself as an interconnect, through NTB. As we explain in Section 2.3, the PCIe system is, in fact, a comprehensive networking system and NTB is its native internetworking mechanism.

Networking. To use NTB means that we interconnect our servers via Non-Transparent Bridging adapter cards. The Transport module receives its input by mirroring the TLP packets that arrive on the CMB interface. From there, the module repackages the traffic and sends the resulting TLP packets to the address of the secondary hosts. NTB adapters have hardware support for multicasting, which alleviates the load on the primary device should many secondaries be deployed. However, for simplicity we chose not to use it. The primary Villars device's Transport module creates one mirror flow per secondary. While this may cause scalability issues when using many secondaries, it allows each secondary to receive traffic at an independent pace.

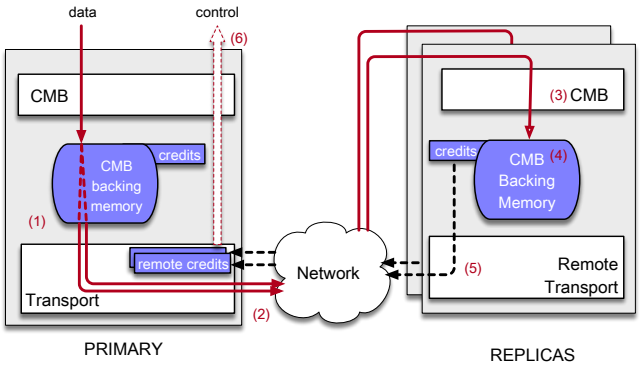

Figure 6: Data and control paths between the primary and secondary VILLARS devices. (1) The primary receives mirror streams of the writes against CMB, one for each secondary. (2) It redirects each stream to a secondary X-SSD device. (3) The latter receives the writes through its CMB interface and (4) persists them to the backing memory, updating the local credit counter. (5) The secondary's transport occasionally updates a shadow copy of its counter in the primary. (6) The device uses a combination of shadow counters to respond depending on the replication protocol implemented.

Each secondary VILLARS's Transport module receives the TLP packets from the primary in the same way it would receive byte-addressed traffic from the local database: through the CMB module. The difference with a secondary device is that the Transport module periodically reports its credit counter to the primary. We will discuss the behavior of credit counters in a secondary setting shortly.

Turning a VILLARS device's Transport module on, either in primary or secondary mode, can be done with an NVMe command. To put it differently, changing the networking mode for a VILLARS device or its peers is done via software and does not require any change to its hardware. The NVMe protocol supports a type of command extension called *vendor-specific*, in which a device manufacturer can include pre-defined commands that are shipped with a device, and the results can be returned to the caller to verify a successful execution. As we mentioned before, the VILLARS device is a fully conformant NVMe device, and the commands we added are sent using vendor-specific features of the regular NVMe drivers that come in a "vanilla" Linux distribution.

Shadow Counters and Replication. As mentioned above, a secondary will send updates about its credit counter to the primary. The frequency with which it does so is adjustable, and we will discuss the implications of these adjustments experimentally (§ 6.5). The primary keeps a copy of each secondary counter, called *shadow counters*, where it applies the credit updates received. The counters can be used to implement different replication protocols.

The VILLARS device implements a primary-secondary eager replication scheme via log-shipping [38]. A transaction's log entries get recorded locally and remotely on the secondary server(s). However, when the database reads the credit counter, the value that the VILLARS device returns is the counter with the most significant delay among the secondaries. The database, therefore, only considers a log entry persisted if it is persisted in all secondaries.

Other replication schemes can be implemented simply by changing which counter or combination thereof the database sees, for example:

- Lazy replication [58] can return the primary counter, allowing the database to proceed independently of the secondaries' speed.
- Chain replication [72] can return the counter of the last secondary in the chain. In that case, all but the last server would have a single shadow counter from the server in the chain.

More sophisticated commit protocols such 2PC [24], Quorum [2], Replicated-State Machine [59], or even deterministic commit schemes (e.g., [69]) can also be supported. However, they require additional coordination and *promise-rollback* mechanisms. These mechanisms are exciting additions to our current design, and we plan on exploring them in future X-SSD devices.

4.3 The Destage Module

The Destage module is a Firmware extension that moves data from the fast to the conventional side. It monitors the amount of data on the CMB backing memory and schedules Flash write operations in small batches. From that point on, the data path for CMB data is almost the same as the data path for conventional data. The only difference is that the Destage module uses a predefined range of logical blocks on the SSD and, as described above, implements a ring buffer over that area. Figure 7 shows how the VILLARS device implements the Destage path.

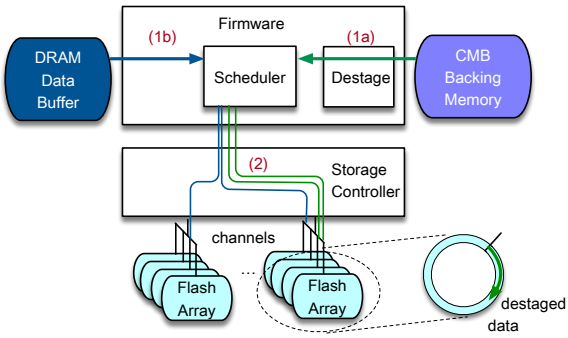

Figure 7: (1a) The Destage module notifies the scheduler when a new chunk of log data is available. (1b) The scheduler considers the data on both the CMB and Data Buffer areas, and (2) determines which one to save in Flash next.

Detailed Path. The Destage module maintains a range of Logical Block Addresses (LBAs) to store CMB data to be destaged. It treats that area as a ring also. For every new page, the module assigns the next LBA on the ring to that page and, if the ring wrapped around, it adjusts its head. The ring's head (and tail, if the ring is not full) can be accessed via the log control interface.

To fill in a new Flash page, the unit of Flash write on the VILLARS device, the Destage module monitors CMB's backing memory area regularly via that ring's head and tail. Occasionally, it bundles a portion of the ring's head data as a Flash data page. The module may also decide to destage less data than a page in order to meet a given latency threshold. It uses *filler* data to complete a page's worth of data, in case the ring's head area is not large enough.

The actual destaging, i.e., moving a data page from the fast to the conventional sides, can be done in different ways. The reason is that the device may also have pages from the conventional side sitting in the Data Buffer that need to be made persistent. The Villars device can be configured to operate with different policies via a *Destage Priority*, *Conventional Priority*, or Neutral scheduling modes.

In each mode, the device's scheduler will prioritize a different source of write requests. The neutral scheduling is that of a traditional device, i.e., it tries to divide the writing opportunities equally. The Destage (Conventional) priority gives precedence to pages coming from the fast (conventional) side. In the Destage (Conventional) mode, the scheduler will only place low-priority requests in the "gaps" of the high priority ones, i.e., the brief moments where a Channel has nothing scheduled for a given array. We call this type of scheduling *Opportunistic Destaging*. Note that, other than in the scheduler, practically no additional change is necessary to the Storage Controller (§ 2.2) to implement a Villars device.

Destaging Efficiency. Performing destaging inside the storage device saves some host memory's bandwidth. The reason is that if an application were to perform destaging (cf. Figure 1 left), it would move data unnecessarily, spending precious bandwidth as follows. The application would first write data to NVM memory. Assuming the NVM is connected to the CPU's memory controller, this represents one data movement. A second and third data movement would be to read the data from NVM and sending it to a storage device. As discussed in Section 2, the data would be copied into the device's data buffer. A fourth data movement would occur when the device reads the data buffer and writes it to Flash. An application performing logging in NVM and its destaging has no option but to trigger these data movements.

In contrast, a X-SSD device, and the Villars in particular, allows the application to write to the backing memory in one data movement and allows the device's Storage Controller to read that area directly. To put it differently, the Villars device can perform in two data movements what the application does using four. The memory bandwidth it saves in the process can then contribute to the database's performance [14].

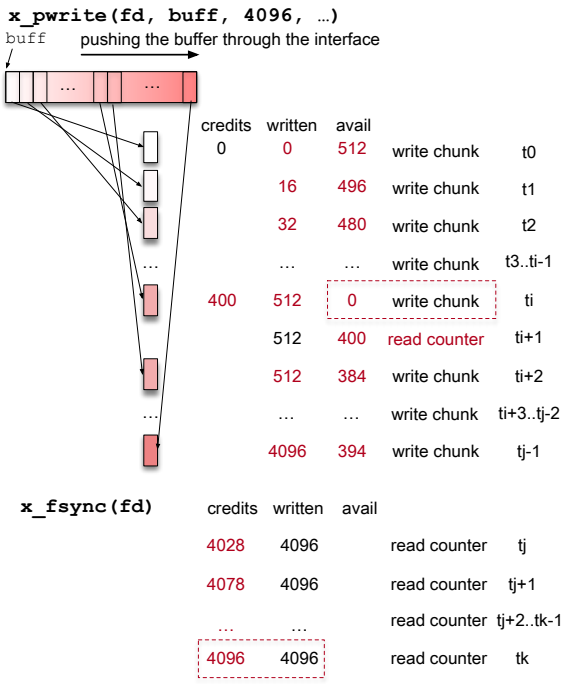

Figure 8: Villars's API implementation. Top: the `x_pwrite()` call starts at time t_0. It knows the CMB queue size, 512 bytes in this example (cf. Section 4.1). The implementation writes chunks of `buff` to CMB until there is no more queue space left, at time t_i. It then issues a read against the control interface, and discovers that, in the meantime, 400 bytes were retired into CMB, leaving 112 bytes still in flight. The `x_pwrite()` implementation alternates between checking credits and writing until it processed `buff`, at time t_{j-1}. Bottom: the application issues an `x_fsync()` to verify that all the prior writes are persistent. The Villars API implementation reads the counter until it sees that it was incremented by the same amount of `buff`'s size, at time t_k, and then returns.

5 THE DATABASE-VILLARS INTERFACE

The database and the Villars device interact on at least two data paths. The first one is when a primary or stand-alone server requests the Villars device to write log data (cf. step (1) in Figure 1 right). The second data path is when a secondary server requests the Villars device to read log data coming from the primary server (cf. step (3) of the same figure). In these cases, we offer drop-in APIs that interact with the Villars device and replace the familiar read and write system calls (§ 5.1). We also offer an alternative, advanced API with no equivalent system call that exports a view of the Villars's fast side as memory, rather than a file (§ 5.2).

5.1 Drop-In Replacement API

We describe how alternative calls to `pwrite()` and `fsync()` for writing and `pread()` for reading can be implemented in the Villars device. These calls are naturally blocking and, therefore, can transmit back pressure when necessary (cf. Section 4.1).

Substituting `pwrite()` and `fsync()`. A `pwrite()` takes as parameters a descriptor, buffer, size, and offset. It writes the size's worth of data from the buffer into the descriptor. Our `x_pwrite()` alternative does not require a descriptor as it implicitly writes only to the fast side. The call copies the buffer into CMB in small iterations, occasionally checking if it needs to back off via the credit counters. Figure 8 depicts this scenario. We tried alternative ways and frequencies to check the credit counter during the copy. The best performance was obtained when using all the credits available without intermediate checks then pausing to read the credit anew.

An `fsync()` takes a file descriptor and blocks until all the bytes written against that descriptor were flushed. Our `x_fsync()` implementation performs a similar wait until the credit counter signals that all bytes written by `x_pwrite()` were retired to PM. The `x_pwrite()` and `x_fsync()` share an internal counter that keeps track of the bytes written. We note that our implementation of `x_pwrite()` and `x_fsync()` are not system calls and therefore do not incur the penalty of context switching into the OS.

Substituting `pread()`. A `pread()` takes as parameters a descriptor, buffer, size, and offset. It reads the size's worth of data into the buffer, starting at offset (relative to the start of the file). This exact semantics are challenging to implement on a circular file because its start keeps moving. We implement, instead, an operation with *tail read* semantics. Our `x_pread()` takes a special offset flag to indicate it wants to read the tail of the log as it grows. Again, we do not require a descriptor as the call implicitly accesses the destage area on the conventional side. Internally, the implementation keeps track of the last read area on the destaged ring; a subsequent call returns the next adjacent area. The implementation obtains information about the destage's progress (§ 4.3) and blocks if not enough data reaches the conventional side to fill the read buffer.

5.2 Advanced API

The CMB implements the file interface above for backward compatibility purposes. However, that area can be exposed as a memory abstraction just as well. The important condition is to establish a destaging criterion in the device. In other words, in a ring abstraction, the head of the ring can be destaged. In another abstraction, the Villars device needs to be able to decide when to move entries from the CMB region into Flash.

To illustrate one of many possible advanced APIs, we show how to use an allocator's pair of calls, `x_alloc()` and `x_free()`, to export CMB functionality. The allocation call determines an area in which the application can write randomly. The area would be active, i.e., not destage-able, until it is freed. The ring abstraction is still valid in this scenario; the next allocated area can be adjacent to the previous one on the ring. There are examples of database systems that use PM for logging purposes in this way [39, 75]. Different database *worker* threads request transaction log buffer this way but fill the areas in parallel. Such a scheme is known as one of the fastest ways to write to a transaction log [34].

6 EXPERIMENTS

To validate the X-SSD architecture and the design decisions we built into the Villars device in particular, we carried out five sets of experiments. The first set contrasts logging locally to the Villars device's conventional side versus logging to its fast side (§ 6.1). The second set evaluates to which extent the size of the writes against the fast side influences the throughput (§ 6.2). The third set looks into how different options to implement the fast side contribute to performance (§ 6.3). The fourth set evaluates the efficiency of different opportunistic destaging policies (§ 6.4). Lastly, we discuss the delays caused by the underlying replication mechanisms (§ 6.5). We start by describing our implementation of the X-SSD along with details of our experimental environment.

Implementation and Environment Details. We implemented the Villars device using the Cosmos+ OpenSSD [44]. The Cosmos+ is a full-fledged SSD prototyping platform built over a Xilinx Zynq Z-7045 SoC FPGA. It hosts 2 TB of Hynix NAND-Flash memory. Our Villars device's conventional side reuses the HCI, Firmware, and Storage Controller components (cf. Figure 2) of the Cosmos+ almost without modifications. The Villars device's fast side components were all implemented from scratch. Of particular relevance are the details on how CMB support was added to the Cosmos+.

First, we decided to constraint the original Cosmos+'s PCIe interface from ×8 Gen2 to a ×4, i.e., 2 GB/s for the CMB experiments. The rationale behind this decision is that the CMB traffic shares bandwidth with the traditional NVMe interface, and our imposed lane restriction better reflects the fact that the full PCIe bandwidth may seldom be available for CMB to consume.

The second relevant detail is how we experiment with different kinds of CMB backing memory. We selected two types of memory to evaluate, SRAM and DRAM, and created a scenario in which we may consider them to be persistent, as we describe shortly. The SRAM memory is provided by BlockRAM in the FPGA in which the Cosmos+ is implemented. This kind of memory runs at the same 250 MHz clock rate as the FPGA and, given that we use a 128-bit wide bus to access it, its bandwidth is 4 GB/s. The DRAM memory comes from the Cosmos+ Data Buffer (cf. Figure 2). We allocated part of that memory pool exclusively for CMB use during the experiments. The DRAM's type is DDR3 and its controller provides a maximum bandwidth of 4 GB/s. However, given that in our setup we access it through a 64-bit wide bus at 250 MHz, the fast side only uses 2 GB/s. The DRAM access is shared with the device's regular data buffering activity. In terms of sizes, the capacity used for CMB in our setup is 128 KB and 128 MB, for SRAM and DRAM, respectively. We could increase this capacity by making certain compromises in terms of how to allocate FPGA resources, but we thought these quantities were appropriate for our experiments.

We note that the Cosmos+ device is powered independently from the server. In other words, the Cosmos+ board does not need power coming from the PCIe slot; it uses its own power source through a small transformer connected to a separate 120/240V outlet. In case of server crashes, the power can be maintained while the Villars implementation destages the contents of CMB and achieves a consistent state even if the database is no longer online. In case of sudden power failures, some groups have experimented with adding supercapacitors that can provide additional power beyond the power source and keep the device online for a while longer [36, 67]. In our experiments, we assume the supercapacitors to be present but have not implemented them. Since this allows CMB's SRAM and DRAM backing to be destaged, they behave effectively as persistent memory.

We use three Xeon-SP 4110 servers, each with 8-cores, 128 GB of RAM, and one X-SSD device. The servers were connected via a RoCE RDMA network using Mellanox ConnectX-5 cards and an Ethernet switch, and via an NTB network using Dolphin PXH830 adapters in a daisy-chained setup.

The servers run ERMIA [39], an open-source in-memory database, to produce WAL transaction logs. ERMIA can generate transactional workloads upwards of 300 ktxn/s using the TPC-C benchmark [70] while performing replication via log-shipping over RDMA and PM [75]. In particular, ERMIA emulates PM in the same way we do: it assumes that the servers' DRAM DIMMS are battery backed, thus supporting NVDIMMs. This assumption allows ERMIA (and us) to conduct all experiments as if DRAM were persistent, even if in practice the servers have regular DIMMs.

ERMIA's logging system is considered to be state-of-the-art. We integrated the X-SSD device's logger into that system by having ERMIA call our drop-in version of `pwrite()` and `fsync()` (cf. Section 5.1). ERMIA pins each of its log writers to a core, therefore the

experiments can scale to up to 8 threads in our servers. Unless mentioned otherwise, we run the TPC-C workload with 16 warehouses. Our goal is to generate the log as fast as the original ERMIA log shipping work [75], which we do.

6.1 Logging to Local Storage

In this experiment, we measure the impact of mapping CMB to different types of backing memory—i.e., as FPGA Block RAM (Villars-SRAM) or DRAM (Villars-DRAM)—and measure the performance of a transaction log over this area. We also compare the cost of writing log records directly to NVDIMM (Memory), against the X-SSD conventional side (NVMe), or not writing log records altogether (No Log).

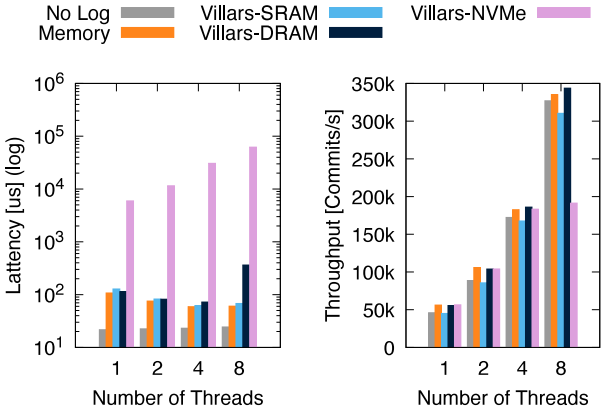

Figure 9: Comparison of latency (left) and throughput (right) with an increasing number of log writes and under different local logging setups. The log workload in this experiment was generated with ERMIA [39, 75].

Figure 9 shows the latency and throughput of the different methods. The x-axis shows the number of workers (threads) the database uses to process transactions, and thus generate log records. The y-axis shows the average transaction latency in log-scale, and the average transaction throughput in terms of committed transactions per second.

For latency, the experiment shows that logging against NVMDIMM or logging against SRAM via CMB yield similar results. The CMB path overhead is compensated by the faster SRAM speed. We note that the transaction latency *decreases* as the number of workers increase. The reason is that the system waits until it has 16 KB worth of log records before it commits. With additional threads, that threshold is reached earlier.

Logging against DRAM backed CMB is also advantageous if compared with logging to the conventional side. The latency remains relatively constant up to 4 threads. With 8 threads, we start seeing back-pressure build-up with DRAM-backed CMB, which reflects on the time transactions need to wait until they can get written.

In terms of throughput, the four methods perform almost equally up until 4 worker threads. With 8 threads, something curious takes place. The logging workload has a queue depth of 1. Given the long latency, the conventional side of the device cannot achieve anything better than approximately 200k transactions per second.

6.2 Effects of Write Combining

The fast side is a byte-addressable interface, i.e., it does not require data to be put in a block before writing (§ 4.1). Technically, even 1-byte writes could be performed in some implementations. However, writes through the fast side become PCIe TLP packets (§ 2.1), and we expect that the overhead of these packets can be significant.

We quantify the overheard through this experiment by sending increasingly large write operations through the fast side and measuring the obtained throughput. To control the amount of bytes written at once to the PCIe subsystem, we use Write Combining (WC) memory regions [32]. When WC is on, the MMIO subsystem block writes before issuing them. When the same memory region is in Uncached (UC) mode, no blocking occurs.

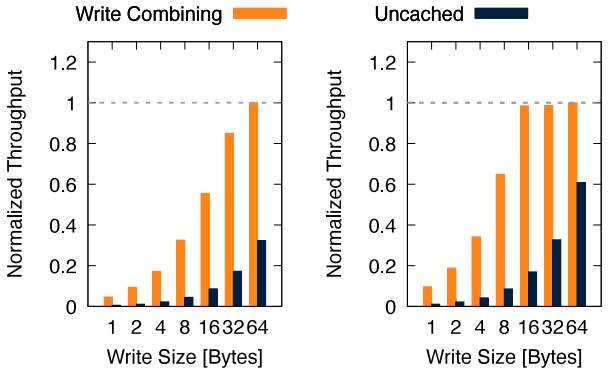

Figure 10: Comparison of different write sizes under write-combine and uncached when writing to device SRAM (left) and DRAM (right).

We show the throughput variations we obtained when using an SRAM backed CMB in Figure 10 (left) and when using a DRAM-backed CMB (right). We normalize the results according to the best throughput, which occurs when we write 64 bytes at once. WC is faster than UC mode in all sizes we tested. For SRAM, the maximum throughput can only be achieved when sending 64 bytes at once. For DRAM-backed CMB, the maximum throughput is reached with 16 bytes or more.

6.3 Effects of CMB Queue Size

The CMB module uses a queue to receive writes before staging them to PM (§ 4.1). The queue's size is important because it determines how much data the database can write over the fast side before checking if the device requires time to catch up with the writes. This experiment sends a controlled workload through the fast side while varying the CMB queue size. The workload's write sizes are also varying to simulate, for instance, larger log records or some form of group commit [27].

We show the latency variations we obtained in Figure 11 (top). As we expected, the latency is primarily dominated by the size of the writes, when the queue size is at least as big as the write size. We expect the majority of log records from OTLP workloads such as TPC-C to be less than 20KB [15]. A queue of 32KB can allow for these records to be sent through the fast side without needing to check the credit counter during those writes.

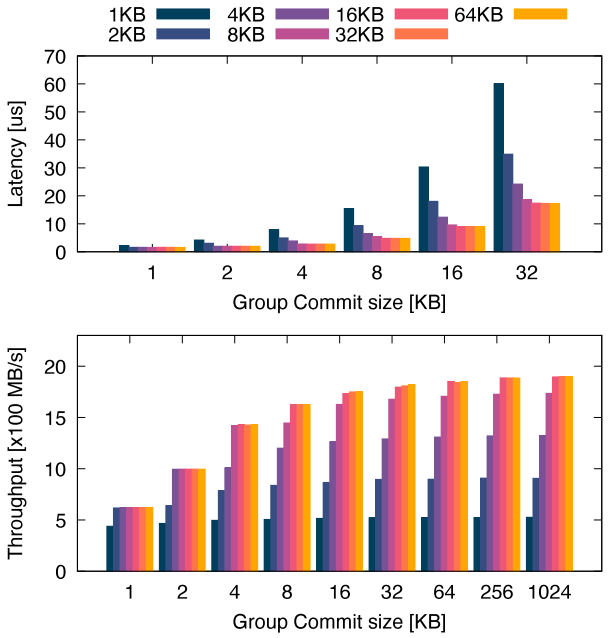

Figure 11: Latency (top) and throughput (bottom) of different group commit sizes (x-axis) with varying device queue sizes (colors) when writing to device SRAM.

We show the throughput variations we obtained in Figure 11 (bottom). A queue with 32KB achieves the best throughput across all the group commit sizes we tested. We note that the platform upon which we built the Villars device is sized to accommodate a maximum of 2GB/s of data transfers [44].

6.4 Effects of Opportunistic Destaging

To test the *Destage Priority* feature of the Villars device (§ 4.3), we send two workloads to it at once. The first workload uses the conventional side, and we sized its throughput at approximately 50% of the device's bandwidth. We sent the second workload through the fast side, and we varied it from 30 to 60%. The experiment's idea is to observe whether the workloads get the correct priority if that feature is turned on.

We show the results in Figure 12, with *neutral priority* on the right and *conventional priority* on the left. We observe that with neutral priority, the device services both workloads until its capacity. At which point, the workloads start interfering with one another, and both suffer reduced bandwidth. With conventional priority, the conventional throughput is preserved independently of the fast workload. We obtained a similar result when using *destage priority* and omit the results for brevity.

6.5 Replication Delay

We implemented a micro-benchmark to evaluate the delay introduced by the shadow counter mechanism (cf. Section 4.2). As described in that section, if a pair of Villars devices are set up as primary and secondary, the former will forward all CMB updates to the latter. In turn, the secondary will periodically update the shadow counter the primary keeps.

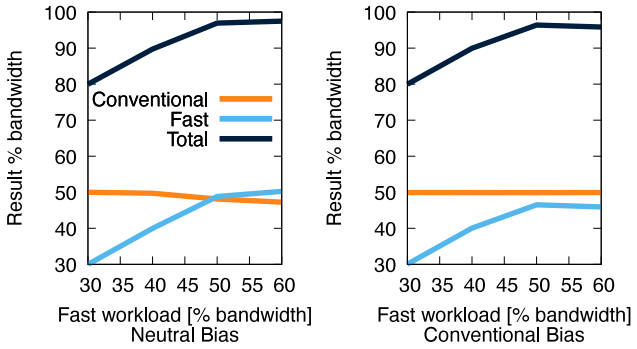

Figure 12: Opportunistic Destaging. Left: The device takes a conventional workload corresponding to 50% of its bandwidth and a varying fast workload. Interference occurs when the device has less bandwidth than requested. Right: The same scenario using *conventional priority*, in which the device is instructed to protect the conventional side's workload.

In this experiment, we vary how frequently the secondary issues these counter updates. We measure the delay between (a) a write against the CMB area of the primary Villars device and (b) receiving the corresponding shadow-counter updates. In other words, we measure the time it takes to the primary to confirm that a write has reached the secondary's CMB area, i.e., the write was safely replicated. Figure 13 presents the results of the experiment.

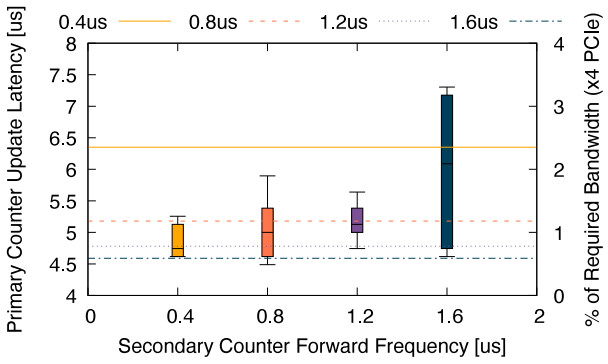

Figure 13: Effect of increasing the frequency in which the secondary X-SSD forwards its credit counter to primary. (Left y axis / candlesticks) Latency taken by the primary X-SSD to update its shadow counter. (Right y axis / horizontal lines) Percentage of bandwidth that credit counter updates require at a given update frequency.

We can observe that updating the shadow counter at higher frequencies, e.g., every 0.4 μs, allows the primary to refresh its shadow counter with low latency variance, between 4.5 and 5.2 μs. Decreasing the frequency, e.g., every 1.6 μs, causes a much higher latency variance, between 4.6 and 7.3 μs. The difference is due to the amount of time between the secondary getting new data and the time until the next counter update cycle[4].

[4]We found the 1.2 μs frequency to be an anomaly. The average latency is higher than with higher latencies, which is expected, but the variance is lower.

The choice of update frequency is a tradeoff between the freshness of the shadow counter and PCIe bandwidth usage. On the one hand, too frequent updates may consume bandwidth that would otherwise be used for CMB writes. For instance, the chart shows that having the secondary send updates every 0.4 μs consumes 2.35% of the available PCIe bandwidth. For a single replica, this may be a reasonable tradeoff. However, as the number of replicas increases, the shadow counter update traffic may start competing with the (user data) CMB traffic. On the other hand, too infrequent updates may compromise the freshness of the shadow counter, introducing replication delays from the database point of view. Ultimately, an increase in bandwidth usage is the price for a tighter latency variance on the primary.

7 USABILITY

We have explained how a single database system interacts with a Villars device during normal operation. This section presents different utilization scenarios, divided into two themes. The first one discusses advanced interactions such as error handling and potential multi-threaded access, still considering a single database instance (§ 7.1). On the second theme, we discuss multiple application utilizations, such as in virtualized scenarios (§ 7.2).

7.1 Single Database-Device Interactions

Errors on the conventional side of a Villars device are handled the same way they would in regular devices. The device marks a page/block as bad and reports the error to the OS/application, e.g., through an error return code. The fast side, however, brings three additional error scenarios: a destage may fail, an entry may fail to propagate from a primary to a secondary, and a credit counter may fail to be forwarded from a secondary into a primary.

In the case of the destage, a failure indicates that the device was trying to write log entries to a bad block. This case is handled internally by picking a new block to write.

In the replicated scenarios cases, the database perceives the error as an indeterminate delay in updating the credit counter. Rather than forcing the database on a counter read loop, the Villars device keeps a status register for the transport module's state. An implementation of `pwrite()` and `fsync()` (cf. Section 5.1) may check that register if it suspects the credit counter to be stale.

Upon encountering a replication error, a database system would want to (re-)configure a device's transport module. That module is responsible for the replication status. As mentioned before, the transport module can be controlled via extended NVMe admin commands (§ 4.2). We added commands to turn an existing or new device from a stand-alone into a primary state, from a primary into a secondary, and from a secondary into a primary. We note that, currently, it is the database system's responsibility to decide when to perform such promotions/demotions, including the transfer of data into or out of the device, if necessary. In a future device, we intend to automate these tasks via hardware/firmware, but this ability was outside the scope of the Villars device.

Another scenario worth mentioning is that of an application that wishes to have several CMB writer threads. The issue with this scenario is that it cannot be supported adequately with a single credit counter. There would be no way to tell which specific writer caused the counter to increase. While the Villars device has not addressed this case, a simple approach is to keep several counters, potentially one per core, and pin writers to the cores. This is akin to maintaining several NVMe work submission queues. Once again, this is a simple extension to the X-SSD architecture.

7.2 Multiple Databases Scenarios

Arguably, a specialized device such as the Villars may be attractive to hyperscalers. These vendors offer database cloud services and could reap substantial savings at scale, thanks to *logging acceleration*, by deploying X-SSD devices at these services. In this context, one may wish to have many virtual databases share a single device.

While not currently implemented by the Villars device, nothing on the X-SSD architecture prevents a device from supporting a virtualization mechanism such as SR-IOV [66]. SR-IOV allows several virtual devices to be exported by a single peripheral, which can then be used separately by different virtual machines. For instance, this mechanism is pervasive in network cards virtualization [17]. In our case, an SR-IOV implementation could simply segment the CMB across smaller, independent regions. The device would create and maintain individual replication configurations for each region, which would then be assigned to different virtual machines.

We note that X-SSD devices may also be beneficial for different workloads than database logging. We consider different scenarios, depending on the use of replication. If the replication mechanism is turned off, the CMB area acts as a low-latency append feature with precise crash semantics. Some usages may include, for instance, transactional services such as Google's Percolator [61]. This system deploys append-only logs but is not exactly a complete database. For another instance, the workload of journaled file systems such as ext4's JBD2 also behaves as a log and can arguably take advantage of CMB for fast writes to that area [54]. If the replication mechanism is turned on, different systems may benefit from a X-SSD device if it could support other replication schemes (§ 4.2). As mentioned then, combining the shadow counters in specific ways may support additional schemes such as chain-replication [72].

8 RELATED WORK

SSDs have long been considered beneficial to database workloads [46], and the literature discussing their convergence is extensive. To the best of our knowledge, however, the X-SSD is the first architecture and Villars the first device to seamlessly integrate data propagation services, both locally and across devices, tailored for transaction logging workloads.

Many similar techniques to those we applied in our device were studied before. We discuss three areas in particular: approaches that streamline the transaction log's (or journaling's) write paths (§ 8.1), approaches that facilitate replication of that workload (§ 8.2), and techniques that can move application logic into SSDs, adapting them to specific workloads, including databases workloads (§ 8.3).

8.1 Fast Logging

Several fast transaction logging primitives were proposed that allow the database to make direct use of battery-backed DRAM [26], Optane (commercial PM) [71], or some form of simulated PM [6, 20, 28, 40, 60, 74]. They all achieve better speeds than logging to

conventional storage. Several of those schemes are compatible with a X-SSD device's fast side and could be implemented atop such a device leveraging that side's API (§ 5.2). By doing so, they would get several benefits automatically, such as destaging and replication.

Other approaches propose to manage PM within the storage instead of exposing it directly to the database. Kang et al. [36] use battery-backed DRAM inside an SSD, specifically in the device's Data Buffer (cf. Figure 2). The data is persistent upon reaching the data buffer. The device provides a *flusher* module, the equivalent of our destaging (§ 4.3). A X-SSD device adopts a similar technique, co-locating a battery-backed DRAM area beside the data buffer (cf. Figure 4). This co-location opens up new opportunities, e.g., we can separate workloads that need or need not immediate persistence.

Bae et al. [8] propose the 2B-SSD, a device that, like ours, presents a block- and a byte-addressable side to an application. In the 2B-SSD, the application is responsible for issuing a new command in the software layer to "move" a given block from the conventional to the fast side, or vice-versa. Once again, we adopt some of the same ideas but use fixed semantics for destaging, obviating the need to move data explicitly. This allows us to offer faster coordination such as opportunistic destaging and log shipping since there is no software (delay) involved in those paths.

When performing transaction logging on traditional SSDs, other workloads such as checkpoints or sorted batches can be the source of interference, causing performance loss [48]. Several techniques investigate how to provide *workload isolation* in this context via scheduling [68], resource allocation [45] or via NVMe directives such as streams [35, 76] and Predictable Latency Mode [1]. Even if the log workload is naturally isolated in a X-SSD device, these techniques remain helpful, as they apply when data is destaging.

8.2 Replication Support

Modern log shipping techniques such as Query Fresh [75] and Active Memory [78] have been proposed that leverage direct access to PM via RDMA. They support hot-standby replicas via a mix of updating remote auxiliary structures on the replicas and by changing the remote query execution engine to consider data in those structures when answering queries. The remote engine occasionally merges the data from the auxiliary structures into the base tables. However, updating the remote PM regions via RDMA can be problematic in crash scenarios. In certain situations, updates can be visible but not necessarily persistent [37], especially without proper hardware support [18].

The Villars device utilizes an equivalent sequence of operations to support log shipping but stops short of updating remote memory. The advantage of this approach is twofold. First, storage-managed PM, i.e., using PM inside an SSD, can offer a more precise crash behavior. Second, the approach is portable in that it only requires simple NVMe support rather than a CPU of a particular brand or a motherboard with support for battery-backed DRAM. The compromise is that, in this particular version, the Villars device supports eager replication rather than a hot-standby solution.

NVMe devices can be visible and accessible to a remote server via numerous ways. The NVMe-over-Fabrics standard [55] considers different types of interconnects, include RDMA-capable networks. There are approaches that provide added functionality, such as ReFlex [42] which is similar in spirit to NVMe-oF but adds isolation between workloads. There are also ways to utilize interconnects not covered by the standards such as NTB and offer device sharing capabilities [53]. These protocols are useful to the conventional side and are orthogonal to the automatic replication that X-SSD devices implement across the fast sides of several devices.

8.3 SSD Extensions

Several proposals exist to turn SSDs into user programmable platforms, such as Willow [65] and Biscuit [25]. While these platforms allow moving several application computations into SSD devices, they do not fundamentally support altering its data and control paths. In contrast, the X-SSD architecture may be seen as building block—a domain-specific language of sorts—that gives databases the ability to express logging and log shipping schemes atop of it.

Lerner and Bonnet [47] offer a taxonomy of techniques to extend SSDs. Devices under the X-SSD architecture are considered *co-designed devices* in that they present themselves as standard devices but offer special semantics when they detect the workload they are built to optimize–the workload sent to the fast side, in the case of the Villars device.

9 CONCLUSION

In this paper, we introduced the X-SSD architecture, a class of SSDs designed with transaction log workloads in mind. A X-SSD device presents one *fast side* that accepts byte-addressable requests at low latency, used for transaction logging, and a *conventional side* offering a traditional SSD block interface to accept regular workloads. The two sides are tightly integrated, allowing data to move seamlessly from the fast to the conventional side. Moreover, the fast side of different device instances can also communicate, allowing the log workload to be shipped remotely with low latency *across* X-SSD devices. Databases can use these data propagation features to build different transaction logging and log shipping-based replication schemes.

This paper also presented the Villars device, a reference design for the X-SSD architecture. The Villars is a full-fledged NVMe device, the standard for direct-attached PCIe (fast) SSDs. We showed that the Villars device can absorb transaction log workloads from a modern database with several advantages—simpler interface, comparable latency, and clearer crash behavior semantics—over having the database directly manipulate PM.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments, Jaewook Kwak for answering many questions about the device's Firmware, and Hyeongyou Jeong for his support with Cosmos+ boards. This work was partly supported by the European Research Council (ERC) under the Horizon 2020 Program (grant agreements 683253/Graphint), by the Samsung Research Funding & Incubation Center of Samsung Electronics (Project Number: SRFC-IT1802-07), and by the University of Fribourg visiting Ph.D. student and Centenaire grants.

REFERENCES

[1] Nick Adams and David Woolf. 2019. NVMe 1.4 Features and Compliance: Everything You Need to Know.

[2] D. Agrawal and A. El Abbadi. 1992. The Generalized Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data. *ACM Trans. Database Syst.* 17, 4 (Dec. 1992), 689–717. https://doi.org/10.1145/146931.146935

[3] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 1743–1756.

[4] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565.

[5] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 707–722. https://doi.org/10.1145/2723372.2749441

[6] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind Logging. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 337–348. https://doi.org/10.14778/3025111.3025116

[7] Jens Axboe. 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[8] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2018. 2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 425–438. https://doi.org/10.1109/ISCA.2018.00043

[9] Kwanhu Bang, Kyung-Il Im, Dong-Gun Kim, Sang-Hoon Park, and Eui-Young Chung. 2013. Power failure protection scheme for reliable high-performance solid state disks. *IEICE TRANSACTIONS on Information and Systems* 96, 5 (2013), 1078–1085.

[10] Stephen Bates and Oren Duer. 2018. Enabling the NVMe CMB and PMR Ecosystem. https://nvmexpress.org/wp-content/uploads/Session-2-Enabling-the-NVMe-CMB-and-PMR-Ecosystem-Eideticom-and-Mell.

[11] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.* 9, 7 (March 2016), 528–539. https://doi.org/10.14778/2904483.2904485

[12] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 689–703. https://www.usenix.org/conference/atc21/presentation/bjorling

[13] Matias Bjørling, Javier González, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies* (Santa clara, CA, USA) *(FAST'17)*. USENIX Association, USA, 359–373.

[14] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 54–65.

[15] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. 2011. TPC-E vs. TPC-C: Characterizing the New TPC-E Benchmark via an I/O Comparison Study. *SIGMOD Rec.* 39, 3 (Feb. 2011), 5–10. https://doi.org/10.1145/1942776.1942778

[16] Dell. [n. d.]. Dell EMC NVDIMM-N Persistent Memory User Guide. https://www.dell.com/support/manuals/en-us/poweredge-r940/nvdimm-n_ug_pub/backup-to-flash.

[17] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. 2012. High performance network virtualization with SR-IOV. *J. Parallel and Distrib. Comput.* 72, 11 (2012), 1471–1480.

[18] Chet Douglas. 2019. RPMEM 2.0: Intel Next General Platform Supprot for RDMA with PMEM. https://downloads.openfabrics.org/WorkGroups/ofiwg/remotepersistentmemory/RPMEM2.0Public.pdf.

[19] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin Levandoski, Thomas Neumann, Andrew Pavlo, et al. 2017. Main Memory Database Systems. *Foundations and Trends® in Databases* 8, 1-2 (2017), 1–130.

[20] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. 2011. High Performance Database Logging Using Storage Class Memory. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, USA, 1221–1231. https://doi.org/10.1109/ICDE.2011.5767918

[21] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2020. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 673–689. https://www.usenix.org/conference/atc20/presentation/farshin

[22] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Low-latency communication for fast DBMS using RDMA and shared memory. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1477–1488.

[23] Bill Gervasi and Jonathan Hinkle. 2017. Overcoming System Memory Challenges with Persistent Memory and NVDIMM-P. https://www.jedec.org/sites/default/files/Bill_Gervasi.pdf

[24] Jim Gray and Andreas Reuter. 1992. *Transaction processing: concepts and techniques*. Elsevier.

[25] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A Framework for near-Data Processing of Big Data Workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 153–165. https://doi.org/10.1109/ISCA.2016.23

[26] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 877–892. https://doi.org/10.1145/3318464.3389716

[27] Pat Helland, Harald Sammer, Jim Lyon, Richard Carr, Phil Garrett, and Andreas Reuter. 1987. Group Commit Timers and High Volume Transaction Systems. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*. Springer-Verlag, Berlin, Heidelberg, 301–329.

[28] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-Aware Logging in Transaction Systems. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 389–400. https://doi.org/10.14778/2735496.2735502

[29] Allen Hubbe and Dave Jiang. 2016. Linux NTB. https://events.static.linuxfound.org/sites/events/files/slides/LinuxNTB_0.pdf.

[30] Infiniband Architecture Specifications [n. d.]. Infiniband Architecture Specification. https://www.infinibandta.org/ibta-specifications-download/.

[31] Intel. [n. d.]. Intel Optane/Micron 3d-XPoint Memory. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html.

[32] Intel. [n. d.]. Memory Cache Control. In *Intel 64 and IA-32 Architectures Sofware Developer's Manual*. Chapter 11.

[33] Mike Jackson, Ravi Budruk, Joseph Winkles, and Don Anderson. 2012. *PCI Express Technology 3.0*. Mindshare press.

[34] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: A Scalable Approach to Logging. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 681–692. https://doi.org/10.14778/1920841.1920928

[35] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The Multi-streamed Solid-State Drive. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*.

[36] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. 2014. Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 529–540. https://doi.org/10.1145/2588555.2595632

[37] Sanidhya Kashyap, Dai Qin, Steve Byan, Virendra J Marathe, and Sanketh Nalli. 2019. Correct, fast remote persistence. *arXiv preprint arXiv:1909.02092* (2019).

[38] Bettina Kemme, Ricardo Jiménez-Peris, and Marta Patiño-Martínez. 2010. Database replication. *Synthesis Lectures on Data Management* 5, 1 (2010), 1–153.

[39] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1675–1687. https://doi.org/10.1145/2882903.2882905

[40] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 385–398. https://doi.org/10.1145/2872362.2872392

[41] Richard P. King, Nagui Halim, Hector Garcia-Molina, and Christos A. Polyzois. 1991. Management of a Remote Backup Copy for Disaster Recovery. *ACM Trans. Database Syst.* 16, 2 (May 1991), 338–368. https://doi.org/10.1145/114325.103715

[42] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash = Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 345–359. https://doi.org/10.1145/3037697.3037732

[43] Lars Bjørlykke Kristiansen, Jonas Markussen, Håkon Kvale Stensland, Michael Riegler, Hugo Kohmann, Friedrich Seifert, Roy Nordstrøm, Carsten Griwodz, and Pål Halvorsen. 2016. Device lending in PCI express networks. In *Proceedings of the 26th International Workshop on Network and Operating Systems Support for*

Digital Audio and Video. 1–6.

[44] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. 2020. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. *ACM Trans. Storage* 16, 3, Article 15 (July 2020), 35 pages. https://doi.org/10.1145/3385073

[45] Miryeong Kwon, Donghyun Gouk, Changrim Lee, Byounggeun Kim, Jooyoung Hwang, and Myoungsoo Jung. 2020. DC-Store: Eliminating Noisy Neighbor Containers using Deterministic I/O Performance and Resource Isolation. In *USENIX Conference on File and Storage Technologies*.

[46] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. 2008. A Case for Flash Memory Ssd in Enterprise Database Applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 1075–1086. https://doi.org/10.1145/1376616.1376723

[47] Alberto Lerner and Philippe Bonnet. 2021. *Not Your Grandpa's SSD: The Era of Co-Designed Storage Devices*. Association for Computing Machinery, New York, NY, USA, 2852–2858. https://doi.org/10.1145/3448016.3457540

[48] Alberto Lerner, Jaewook Kwak, Sangjin Lee, Kibin Park, Yong Ho Song, and Philippe Cudré-Mauroux. 2020. It Takes Two: Instrumenting the Interaction between In-Memory Databases and Solid-State Drives. In *CIDR 2020, 10th Conference on Innovative Data Systems Research*. http://http://cidrdb.org/cidr2020/papers/p19-lerner-cidr20.pdf

[49] Feilong Liu, Claude Barthels, Spyros Blanas, Hideaki Kimura, and Garret Swart. 2021. Beyond mpi: New communication interfaces for database systems and data-intensive applications. *ACM SIGMOD Record* 49, 4 (2021), 12–17.

[50] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: scalable hashing on persistent memory. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1147–1161.

[51] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking main memory OLTP recovery. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 604–615.

[52] Jonas Markussen, Lars Bjørlykke Kristiansen, and Hugo Kohmann. 2019. NVMe over PCIe Fabrics using Device Lending. http://dolphinics.no/download/WHITEPAPERS/nvme_over_pcie_fabrics_device_lending.pdf.

[53] Jonas Markussen, Lars Bjørlykke Kristiansen, Håkon Kvale Stensland, Friedrich Seifert, Carsten Griwodz, and Pål Halvorsen. 2018. Flexible Device Sharing in PCIe Clusters Using Device Lending. In *Proceedings of the 47th International Conference on Parallel Processing Companion* (Eugene, OR, USA) *(ICPP '18)*. Association for Computing Machinery, New York, NY, USA, Article 48, 10 pages. https://doi.org/10.1145/3229710.3229759

[54] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, Vol. 2. 21–33.

[55] NVMe Standard. 2020. *NVM Express Base Specification Revision 1.4b*. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4b-2020.09.21-Ratified.pdf

[56] Oracle. [n. d.]. Creating a Standby Database with Recovery Manager. https://docs.oracle.com/en/database/oracle/oracle-database/19/sbydb/creating-data-guard-standby-database-using-RMAN.html.

[57] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 371–386.

[58] Esther Pacitti, Pascale Minet, and Eric Simon. 1999. Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 126–137.

[59] Fernando Pedone, Rachid Guerraoui, and André Schiper. 2003. The database state machine approach. *Distributed and Parallel Databases* 14, 1 (2003), 71–98.

[60] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. 2013. Storage Management in the NVRAM Era. *Proc. VLDB Endow.* 7, 2 (Oct. 2013), 121–132. https://doi.org/10.14778/2732228.2732231

[61] Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. 251–264.

[62] Christos A. Polyzois and Héctor García-Molina. 1994. Evaluation of Remote Backup Algorithms for Transaction-Processing Systems. *ACM Trans. Database Syst.* 19, 3 (Sept. 1994), 423–449. https://doi.org/10.1145/185827.185836

[63] PostgreSQL. [n. d.]. Log-Shipping Standby Servers. https://www.postgresql.org/docs/current/warm-standby.html.

[64] Abdallah Salama, Carsten Binnig, Tim Kraska, Ansgar Scherp, and Tobias Ziegler. 2017. Rethinking Distributed Query Execution on High-Speed Networks. *IEEE Data Eng. Bull.* 40, 1 (2017), 27–37.

[65] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-Programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 67–80.

[66] PCI SIG. 2007. PCI-SIG Single Root I/O Virtualization. http://www.pcisig.com/specifications/iov/single_root/.

[67] Yongseok Son, Jaeyoon Choi, Jekyeom Jeon, Cheolgi Min, Sunggon Kim, Heon Young Yeom, and Hyuck Han. 2017. SSD-Assisted Backup and Recovery for Database Systems. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 285–296. https://doi.org/10.1109/ICDE.2017.88

[68] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie S. Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. 2018. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 397–410. https://doi.org/10.1109/ISCA.2018.00041

[69] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[70] Transaction Processing Performance Council. 2010. TPC-C Benchmark Revision 5.11.0. http://www.tpc.org/tpcc/.

[71] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent Memory I/O Primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware* (Amsterdam, Netherlands) *(DaMoN'19)*. Association for Computing Machinery, New York, NY, USA, Article 12, 7 pages. https://doi.org/10.1145/3329785.3329930

[72] Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (San Francisco, CA) *(OSDI'04)*. USENIX Association, USA, 7.

[73] Paul Von Behren. 2015. NVML: implementing persistent memory applications. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*.

[74] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging through Emerging Non-Volatile Memory. *Proc. VLDB Endow.* 7, 10 (June 2014), 865–876. https://doi.org/10.14778/2732951.2732960

[75] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2017. Query Fresh: Log Shipping on Steroids. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 406–419. https://doi.org/10.1145/3164135.3164137

[76] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. 2017. AutoStream: Automatic Stream Management for Multi-Streamed SSDs. In *ACM International Systems and Storage Conference (Systor)*.

[77] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *International Conference on Cloud Computing Technology and Science (CloudCom)*.

[78] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking Database High Availability with RDMA Networks. *Proc. VLDB Endow.* 12, 11 (July 2019), 1637–1650. https://doi.org/10.14778/3342263.3342639