Towards new memory paradigms : Integrating non-volatile main memory and remote direct memory access in modern systems

Rémi Dulong

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2023IPPAS027

Thèse de doctorat

Université de Neuchâtel
Institut d'informatique

TELECOM SudParis

IP PARIS

Towards New Memory Paradigms: Integrating Non-Volatile Main Memory and Remote Direct Memory Access in Modern Systems

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Université de Neuchâtel (Suisse)

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Neuchâtel, le 18 Décembre 2023, par

RÉMI DULONG

Composition du Jury :

Thomas Clausen Professeur, École Polytechnique, Institut Polytechnique de Paris	Président
Anne-Marie Kermarrec Directrice de recherche, École Polytechnique Fédérale de Lausanne	Rapportrice
Noël De Palma Professeur, Université Grenoble Alpes	Rapporteur
Patrick Eugster Professeur, Università della Svizzera italiana	Examinateur
Pascal Felber Professeur, Université de Neuchâtel	Co-directeur de thèse
Gaël Thomas Directeur de recherche, Télécom SudParis, Institut Polytechnique de Paris	Co-directeur de thèse

Abstract

Modern computers are built around two main parts: their Central Processing Unit (CPU), and their volatile main memory, or Random Access Memory (RAM). The basis of this architecture takes its roots in the 1970's first computers. Since, this principle has been constantly upgraded to provide more functionnality and performance.

In this thesis, we study two memory paradigms that drastically change the way we can interact with memory in modern systems: non-volatile memory and remote memory access. We implement software tools that leverage them in order to make them compatible and exploit their performance with concrete applications. We also analyze the impact of the technologies underlying these new memory medium, and the perspectives of their evolution in the coming years.

For non-volatile memory, as the main memory performance is key to unlock the full potential of a CPU, this feature has historically been abandoned on the race for performance. Even if the first computers were designed with non-volatile forms of memory, computer architects started to use volatile RAM for its incomparable performance compared to durable storage, and never questioned this decision for years. However, in 2019 Intel released a new component called Optane DC Persistent Memory (DCPMM), a device that made possible the use of Non-Volatile Main Memory (NVMM). That product, by its capabilities, provides a new way of thinking about data persistence. Yet, it also challenges the hardware architecture used in our current machines and the way we program them.

With this new form of memory we implemented NVCache, a cache designed for non-volatile memory that helps boosting the interactions with slower persistent storage medias, such as Solid State Drive (SSD). We find NVCache to be quite performant for workloads that require a high granularity of persistence guarantees, while being as easy to use as the traditional POSIX interface. Compared to file systems designed for NVMM, NVCache can reach similar or higher throughput when the non-volatile memory is used. In addition, NVCache allows the code to exploit NVMM performance while not being limited by the amount of NVMM installed in the machine.

Another major change of in the computer landscape has been the popularity of distributed systems. As individual machines tend to reach performance limitations,

using several machines and sharing workloads became the new way to build powerful computers. While this mode of computation allows the software to scale up the number of CPUs used simultaneously, it requires fast interconnection between the computing nodes. For that reason, several communication protocols implemented Remote Direct Memory Access (RDMA), a way to read or write directly into a distant machine's memory. RDMA provides low latencies and high throughput, bypassing many steps of the traditional network stack.

However, RDMA remains limited in its native features. For instance, there is no advanced multicast equivalent for the most efficient RDMA functions. Thanks to a programmable switch (the Intel Tofino), we implemented a special mode for RDMA that allows a client to read or write in multiple servers at the same time, with no performance penalty. Our system called Byp4ss makes the switch participate in transfers, duplicating RDMA packets. On top of Byp4ss, we implement a consensus protocol named DISMU, which shows the typical use of Byp4ss features and its impact on performance. By design, DISMU is optimal in terms of latency and throughput, as it can reduce to the minimum the number of packets exchanged through the network to reach a consensus.

Finally, by using these two technologies, we notice that future generations of hardware may require a new interface for memories of all kinds, in order to ease the interoperability in systems that tend to get more and more heterogeneous and complex.

Keywords: Non-Volatile Main Memory (NVMM), Remote Direct Memory Access (RDMA), Memory disaggregation, In-network computing.

Résumé

Les ordinateurs modernes sont construits autour de deux éléments : leur CPU et leur mémoire principale volatile, ou RAM. Depuis les années 1970, ce principe a été constamment amélioré pour offrir toujours plus de fonctionnalités et de performances.

Dans cette thèse, nous étudions deux paradigmes de mémoire qui proposent de nouvelles façons d'interagir avec la mémoire dans les systèmes modernes : la mémoire non-volatile et les accès mémoire distants. Nous mettons en œuvre des outils logiciels qui exploitent ces nouvelles approches afin de les rendre compatibles et d'exploiter leurs performances avec des applications concrètes. Nous analysons également l'impact des technologies utilisées, et les perspectives de leur évolution dans les années à venir.

Pour la mémoire non-volatile, comme les performances de la mémoire sont essentielles pour atteindre le potentiel d'un CPU, cette fonctionnalité a historiquement été abandonnée. Même si les premiers ordinateurs ont été conçus avec des formes de mémoire non volatiles, les architectes informatiques ont commencé à utiliser la RAM volatile pour ses performances inégalées, et n'ont jamais remis en question cette décision pendant des années. Cependant, en 2019, Intel a commercialisé un nouveau composant appelé Optane DCPMM qui rend possible l'utilisation de NVMM. Ce produit propose une nouvelle façon de penser la persistance des données. Mais il remet également en question l'architecture de nos machines et la manière dont nous les programmons.

Avec cette nouvelle forme de mémoire, nous avons implémenté NVCache, un cache en mémoire non-volatile qui permet d'accélérer les interactions avec des supports de stockage persistants plus lents, tels que les SSD. Nous montrons que NVCache est particulièrement performant pour les tâches qui nécessitent une granularité élevée des garanties de persistance, tout en étant aussi simple à utiliser que l'interface POSIX traditionnelle. Comparé aux systèmes de fichiers conçus pour NVMM, NVCache peut atteindre un débit similaire ou supérieur lorsque la mémoire non volatile est utilisée. De plus, NVCache permet aux programmes d'exploiter les performances de NVMM sans être limité par la quantité de NVMM installée sur la machine.

Un autre changement majeur dans le paysage informatique a été la popularité des

systèmes distribués. Alors que les machines ont individuellement tendance à atteindre des limites de performances, l'utilisation de plusieurs machines et le partage des tâches sont devenus la nouvelle façon de créer des ordinateurs puissants. Bien que ce mode de calcul permette d'augmenter le nombre de CPU utilisés simultanément, il nécessite une connexion rapide entre les nœuds de calcul. Pour cette raison, plusieurs protocoles de communication ont implémententé RDMA, un moyen de lire ou d'écrire directement dans la mémoire d'un serveur distant. RDMA offre de faibles latences et un débit élevé, contournant de nombreuses étapes de la pile réseau.

Cependant, RDMA reste limité dans ses fonctionnalités natives. Par exemple, il n'existe pas d'équivalent de multicast pour les fonctions RDMA les plus efficaces. Grâce à un switch programmable (le switch Intel Tofino), nous avons implémenté un mode spécial pour RDMA qui permet de lire ou d'écrire sur plusieurs serveurs en même temps, sans pénalité de performances. Notre système appelé Byp4ss fait participer le switch aux transferts, en dupliquant les paquets RDMA. Grâce à Byp4ss, nous avons implémenté un protocole de consensus nommé DisMu. De par sa conception, DisMu est optimal en termes de latence et de débit, car il peut réduire au minimum le nombre de paquets échangés sur le réseau pour parvenir à un consensus.

Enfin, en utilisant ces deux technologies, nous remarquons que les futures générations de matériel pourraient nécessiter une nouvelle interface pour les mémoires de toutes sortes, afin de faciliter l'interopérabilité dans des systèmes qui ont tendance à devenir de plus en plus hétérogènes et complexes.

Mots-clés : Mémoire principale non volatile (NVMM), Accès direct de mémoire distante (RDMA), Mémoire désagrégée, Réseaux programmables.

Remerciements

Merci aux Prof. Pascal Felber et Prof. Gaël Thomas, mes deux directeurs de thèse, qui m'ont fait découvrir le monde de la recherche, et qui ont choisi de me faire confiance du début à la fin de ce long parcours. Merci également au FNS (Fonds National Suisse) pour avoir financé cette thèse dans le cadre du projet PersiST (no. 178822). Merci à Valerio, pour son soutien dans tous nos travaux. Merci à Baptiste pour avoir amené un œil nouveau sur mon travail, ce qui a considérablement contribué à la seconde partie de cette thèse. Merci aux membres de mon jury, qui ont pris de leur précieux temps pour considérer et apprécier mon travail.

Merci à tous mes collègues, qu'ils aient été profs, doctorants ou stagiaires. En particulier, merci à Sébastien qui m'a fait découvrir le canton de Neuchâtel mais aussi la Suisse, sa culture, ses traditions, ses particularités linguistiques… Si bien que je n'en partirai pas tout de suite ! Merci à Jämes d'avoir toujours été un collègue sur lequel on peut compter, pour les pizze comme pour le travail, et d'avoir assuré quasiment seul la gestion de notre cluster pendant la période d'écriture de cette thèse. Merci à Peterson, l'ingénieur Camerounais dont le pays peut être fier ! Merci à Gal, pour son soutien et son humour, malgré la tristesse des évènements qui secouent son pays en ce moment. Merci à Rafael et Dorian, les experts sur lesquels j'ai pu m'appuyer au début de l'aventure. Merci à Isabelly et Catherine, pour leur sympathie et pour avoir réussi le tour de force de me motiver à améliorer mon anglais ! Merci à Pedro et Andreia, pour m'avoir guidé dans les arcanes de la NVMM. Merci à Quentin, Gilles et Nathan, pour avoir contribué de manière directe à ces travaux pendant leurs stages. Merci à Afrim, pour son travail mais surtout pour sa bonne humeur ! Merci à Lorenzo, pour m'avoir fourni le clavier de sa conception qui a servi à la rédaction de cette thèse (Adm42 pour les curieux). Merci à Pasquale, Victor, Andreas, Simon, Vladimir, et tous ceux que j'ai pu rencontrer au labo au fil de ces cinq années.

Merci à tous les professeurs et encadrants que j'ai pu avoir pendant ma scolarité, et qui ont tous contribué d'une manière ou d'une autre à la personne que je suis aujourd'hui. Merci à tous les étudiants pour lesquels j'ai eu le plaisir d'enseigner, ces quelques heures par semaine me manqueront probablement.

Merci à tous les anciens d'INTech, François, Florian, Victor, Lucas, et plus globalement à toute l'équipe du "Manoir". J'espère que la bande restera encore soudée

pendant de nombreuses années ! Merci à Matthieu, Pauline, Romain, et tous mes amis de longue date que je ne vois pas aussi souvent que je le voudrais.

Merci à mes grands-parents, Marie-France et Bernard, pour les innombrables heures à refaire le monde sur Skype le Dimanche soir. Merci à mes sœurs, Lucille et Candice, pour les pauses jeux vidéo et les retrouvailles chaque année à l'été, ou aux fêtes de fin d'année. Une pensée aussi pour mon grand-père Jean, avec qui j'ai appris bien plus que ce qu'on trouve dans les livres d'école, et pour mes arrière-grands-parents Lily et Joseph, chez qui j'avais eu mes premiers "cours" d'informatique ! S'ils avaient vu jusqu'où tout cela m'a mené... Enfin, merci à toute ma famille, mes parents, mes grands-parents, oncles, tantes, cousines et cousins, pour leur soutien, et pour avoir toujours créé cette ambiance si particulière qu'il est bon de retrouver à chaque réunion de famille. En particulier, merci à mes parents Dominique et Gilles qui m'ont soutenu et m'ont permis d'être guidé dans ce parcours scolaire si privilégié.

Contents

List of acronyms

API	Application Programming Interface
ADR	Asynchronous DRAM Refresh
ACK	Acknowledgment
ASIC	Application-Specific Integrated Circuit
ALU	Arithmetic Logic Unit
AI	Artificial Intelligence
BIOS	Basic Input Output System
BfRt	Barefoot runtime
CPU	Central Processing Unit
CA	Channel Adapter
CD-RW	Compact Disk-Rewritable
CXL	Compute Express Link
DBMS	Database Management System
DAX	Direct Access
DIMM	Dual In-line Memory Module
DCPMM	DC Persistent Memory
DDR4	Double Data Rate 4
DRAM	Dynamic Random-Access Memory
DMA	Direct Memory Access
DPDK	Data Plane Development Kit
eADR	External Asynchronous DRAM Refresh
FUSE	Filesystem in Userspace
FIO	Flexible I/O tester
FPGA	Field-Programmable Gate Array
GRH	Global Route Header
GPU	Graphics Processing Unit
HDD	Hard Disk Drive
HCI	Hyper-Converged Infrastructure
HPC	High-Performance Computing
IP	Internet Protocol
I/O	Input/Output
IBoE	Infiniband over Ethernet
LPC	Linux Page Cache
LRU	Least Recently Used
LAN	Local Area Network
LRH	Local Route Header
MTT	Memory Translation Table
MMU	Memory Management Unit
MTU	Maximum Transmission Unit
NVRAM	Non-Volatile RAM
NVMM	Non-Volatile Main Memory
NVDIMM	Non-Volatile Dual In-line Memory Module (DIMM)
NAK	Negative Acknowledgment
NIC	Network Interface Card
NUMA	Non-Uniform Memory Access
NVMe	NVM Express, or Non-Volatile Memory Host Controller Interface Specification
OS	operating system
OSI	Open Systems Inteconnection
PMEM	Intel Optane Persistent Memory
POSIX	Portable Operating System Interface
PSA	Protocol-independant Switch Architecture
PSN	Packet Sequence Number
PFENCE	Persistent Fence
PSYNC	Persistent Sync

PWB Persistent Write-Back
PCIe Peripheral Component Interconnect Express
PMDK Persistent Memory Development Kit
PCM Phase-Change Memory
QP Queue Pair
RAM Random Access Memory
RDMA Remote Direct Memory Access
RoCE RDMA over Converged Ethernet
RTR Ready to Receive
RTS Ready to Send
SSD Solid State Drive
SATA Serial Advance Technology Attachment
TNA Intel Tofino Native Architecture
UDP User Datagram Protocol
WPQ Write Pending Queue

List of figures

List of tables

Chapter 1.

Introduction

Modern computers are built around two main components: their Central Processing Unit (CPU), and their main memory, *i.e.*, Random Access Memory (RAM). This architecture became standard, and most computers around the world are built within the same pattern. However, we always try to make these machines more powerful, by increasing their computing capacity in various ways. The history of computer science, even rather short, is a constant quest for performance interspersed by physical or technical limitations.

There are numbers of ways to increase the computational capacity of a machine. The first approach is to increase the frequency of its CPU, so that more atomic operations are performed in the same amount of time. This idea is quickly limited, as the power required by a CPU is proportional to the frequency it is running at [134]. Moreover, increasing the frequency of computation does not make a lot of difference if the CPU is constantly waiting for data to process. For that reason, a second approach is to use faster memory. But memory performance has always been a trade-off with price. From this constraint emerged the concept of memory hierarchy. In a nutshell, we used fast volatile memory as main memory in the system, and we added a small amount of extremely fast memory inside CPUs, as close as possible to the core, and used it as a cache for all other memory accesses. This is the highest tier in the memory hierarchy, and also the most expensive. Other kinds of memory are considerably slower but more affordable, and can be used to store data on longer term. At this point, there is not much more that can be done to increase the computational power of a single CPU core, besides adding specific features for specific workloads that would save some CPU cycles on repetitive tasks. Yet, this is only the beginning of the performance quest.

A famous "law" of computer science, called Moore's law in the 1960's [97], made a simple observation: the number of transistors on a CPU chip doubles every two years. By the time it was edicted, Moore's law made a lot of sense, and was actually a good prediction for the next decades. Though, it is clear that this law is destined to dissapear, as CPU manufacturers are reaching the limits of miniaturization. Nowadays, CPU dies are processed with a precision of 3nm, which represents the width

of only 27 atoms of silicon. Additionally, building electrical circuitry at such a small scale even implies to take in consideration quantum tunelling effects. For these reasons, some industrial actors agree on the fact that Moore's law is already over. In this context, maintaining the evolution of computers requires other approaches.

A major revolution in modern computer science has been to use several computing units to exceed the limits of a single CPU core. When several CPU cores are on the same machine, we talk about parallel programming. This has made individual computers more efficient on plenty of workloads, as long as the work can be splitted into several independant parts. In this case, the main memory of the machine is shared between CPU cores. A second idea is to use several independant machines in a network and make them collaborate by splitting the tasks between them through a network. However, this presupose each machine has its own main memory, and does not share it with the others. A considerable part of the efforts to make more efficient computers and supercomputers is now aimed at building better collaboration between machines.

In this context, this thesis presents and uses two new ways of interacting with memory. The first one, non-volatile main memory, proposes to introduce a new stage in the memory hierarchy. By adding an intermediate kind of memory between the fast volatile main memory and the storage memory, we demonstrate how non-volatile memory could take an important place in the way we interact with local memory. The second one, remote direct memory access, makes an attempt to share main memory between machines. With the help of advanced networking components, we propose a way to use remote memory and ease the collaboration between machines within the same cluster.

Part I.

Non-Volatile Main Memory (NVMM)

Chapter 2.

A New Kind of Memory

In 2015, Intel revealed its 3DXPoint non-volatile memory technology would be available on the market in the coming years. This chapter explains why this announcement was a milestone, and how the resulting technology has evolved afterwards.

2.1. Definition and Terminology

Non-Volatile Main Memory (NVMM) is a kind of memory able to retain its content over power loss. Unlike an Solid State Drive (SSD) which must be accessed by blocks, NVMM is byte-addressable. It can be accessed using *load* and *store* instructions with a byte granularity, just as regular Random Access Memory (RAM). From an architectural point of view, NVMM is located on the RAM bus, generally combined with regular RAM modules.

Depending on the context, NVMM can also be referred as Non-Volatile RAM (NVRAM), or Persistent Memory (PMEM). Non-Volatile Dual In-line Memory Module (DIMM) (NVDIMM) is a more general term, as it can also refer to block-addressable persistent DIMM modules.

2.2. History of Persistent Memories

The idea of having a persistent main memory in computers originally comes from the first computers ever built in the 1950's. In that time, most computers were built with magnetic-core memory, also called core memory. Basically, a memory core stores bits in rings of ferro-magnetic metal. Copper wires were passing through each ring, so that flowing an electrical current by these wires would change the magnetic orientation of a targetted ring, making the user able to store one bit per ring. To read a ring content, one had to apply a current in the opposite direction. If a resistance was measured, then the ring was storing a `1`. Otherwise, it was storing `0`. By using this extremely simple and primitive form of memory, the programmers

were already using persistent memory. Indeed, no power supply was required to keep the state of each memory cell and thus, the content stored in that kind of memory. One of the main advantage of such a device is that, in case of a power outage, an ongoing computation could be resumed in the middle, instead of starting from the begining again. However, in the 1970's, that technology was rapidly replaced by faster and more compact forms of memory, due to the popularization of semiconductors. At this time, computer architects chose to give up on the persistence of their main memory, as volatile forms of memory were considerably faster.

During the following decades, a few kinds of persistent memories emerged, in particular in the world of embedded systems. Being exposed to more risks of power losses, these systems were a perfect use case for such memories, as long as computing performance was not the main concern. As the performance and the density of volatile memories was increasing exponentially, these persistent medias were not used in the context of high performance computing.

Suddenly, in 2015, Intel announced a partnership with Micron Technologies, aiming to create the new persistent 3DXPoint memory. This new memory cell would be Phase-Change Memory (PCM), and is based on the use of chalcogenide materials, the same kind of material that was used for the rewritable surface of Compact Disk-Rewritable (CD-RW). On the paper, 3DXPoint was reaching densities and performances that were unheard of, for a persistent device. Several products featuring this memory were about to be available in the public market in the coming years. One of these products is the Intel Optane DC Persistent Memory (DCPMM) module, a non-volatile module to install on the RAM bus, and dedicated to servers. In a device of the size of a RAM module, Intel managed to fit up to 512GB of non-volatile memory, when Double Data Rate 4 (DDR4) Dynamic Random-Access Memory (DRAM) modules hardly reached 128GB.

In early 2019, Intel Optane DCPMM modules became available on the public market. They provide an unprecedented set of features, in particular it is the first device with such high capacity that allow byte addressability. Generally, higher capacity devices are block devices. They communicate with the rest of the system with blocks of data, usually 4KiB blocks. But Intel Optane DCPMM uses a finer granularity of access, as each byte can be accessed individually, as if it was regular DRAM. In terms of performance, Intel also suggested this memory would reach latencies close to DRAM modules, which was the main breakthrough of their 3DXPoint memory cell.

2.3. Implementations

Non-volatile memory is a concept that was implemented in several ways and in different contexts, from embedded systems to high-end servers. The only common point between these technologies is the association of persistence and byte addressability.

2.3.1. For embedded systems

In the world of embedded systems, non-volatile memory is often reffered as NVRAM. The main use case of NVRAM is to remember parameters set by a user, even after rebooting the device. For instance, we can find NVRAM chips in some devices Basic Input Output System (BIOS), in order to keep critical boot settings. This is the case for most Apple Mac computers [119].

However, this thesis will not focus on this kind of NVRAM, as it is not meant to be used for proper computing. These chips are used for their practicality, but their design is not suited to become the main memory of the entire computer, both in terms of performance and compacity.

2.3.2. For servers

After decades of computing based on volatile main memory, new NVMM implementations were released with the hope of covering new use cases for servers.

Simulated persistent memory

As a first step, some simulators for NVMM have been created. Before 2018, with the motivation of having the actual hardware in a few years, many simulation tools were developped. Yet, as the details of Intel's implementation were not public, some of these tools had to base their approach on assumptions. These simulators are mostly developped in the architecture community, so they focus on simulating the hardware behavior of several persistent memory cells [131, 108].

Meanwhile in the systems community, the main concern was to find a way to emulate the behavior of persistent memory. For that purpose, since the version 4.2 of the Linux kernel, a compilation flag has been added to support NVMM. Once the kernel is compiled properly, a boot option allows to allocate a part of the DRAM memory and make it behave as if it was persistent memory. This trick is only meant

to test the Application Programming Interface (API) provided by the Linux kernel, while starting the development of applications without access to the hardware. Naturally, using a volatile support for that special memory space does not make it resilient to crashes. Also, the performance measured is considerably better than the one of the actual hardware. Nevertheless, this emulated persistent memory is transparently exposed by the kernel as if it was a module of Intel Optane DCPMM, which makes it a very useful development tool.

Battery backed modules

The main issue in order to use persistent memory as main memory, was the significant drop of performance of persistent cells (before 3DXPoint). To bypass this physical limitation, some attempts were made to build NVMM modules from other existing technologies. By merging the volatile DDR4 RAM with non-volatile Flash memory (found in SSD), some hybrid modules were created. On these modules, that look like regular RAM modules, one side has DDR4 chips while the other side has Flash memory. In case of a power loss, the module would use a capacitor to stay up and copy the content stored in DDR4 in its Flash memory.

This category of persistent memory is called NVDIMM-N. These modules have the advantage of presenting performances of DDR4 memory while running in normal conditions. However, their conception limits the memory compacity. For one module, the maximum capacity available was 32GB [95].

Intel Optane DCPMM 100 Series

The first generation of Intel Optane DCPMM, released in 2018, is the first broadly available implementation of the NVDIMM-P category. Optane modules only have 3DXPoint chips, and no battery or capacitor. By design, this new memory is persistent, so it does not require any special process in case of power loss. These characteristics gave the NVDIMM-P category the reputation of being the “real” persistent memory.

Additionally, 3DXPoint memory is denser than DDR4; DCPMM modules were commercialized as either 128, 256 or 512 GB per module. However, these improvements come at the cost of a higher latency, as Optane DCPMM was announced with a 10 times higher latency than typical DDR4.

These modules can be set in two modes from one machine’s BIOS:

- The *Memory mode*

- The *AppDirect mode*

The *Memory mode* uses the non-volatile memory as an extension of the machine's RAM. Thanks to its high compacity, NVMM can thus be used as a less expensive way to reach high amounts of RAM in a single machine. In this mode, the persistent memory is not exposed in the OS as a special device. It appears exactly as RAM, with not management of its persistence capability.

The *AppDirect mode* is the "manual" mode of Optane modules. On a Linux-based operating system (OS), it allows programmers to access the memory the same way they would access a disk (i.e through a file located in `dev`)

There are then submodes that can be set from the OS, among them:

- *fsdax mode*
- *devdax mode*

In *fsdax mode*, the device file of each module appears as `devpmemX` where `X` is the module number. It is designed to be used with a Direct Access (DAX) file system. One has to format the device in such a file system before using it. Then, files can be created and used as subdivisions of the NVMM module, as if they were DAX files themselves.

With *devdax mode*, the device appears as `devdaxX.0`. This submode requires to use the entire module as one DAX file. To use a DAX file, one has to memory-map the file into a program's virtual address space. The resulting memory area will thus be hosted in the persistent memory module.

Intel Optane DCPMM 200 Series

In 2020, Intel released a new version of the Optane DCPMM modules, called *200 Series*. In essence, these new modules contain the same 3DXPoint memory. They behave the same way from the OS perspective, with some minor improvements in terms of throughput [7].

Hence, the main difference between the two versions resides in the persistence guarantees. Intel Optane DCPMM 200 Series can use a mechanism called External Asynchronous DRAM Refresh (eADR), which is meant to solve the main problem with persistent memory integration. Indeed, there is a major design flaw that prevents integrating PMEM transparently in our common computer architectures: the volatility of Central Processing Unit (CPU) caches. eADR is a mechanism designed to expand the power supply of NVMM and CPU internal caches, with an external

battery located, for instance, on the motherboard [53]. If compatible CPUs were released at the same time (Xeon Scalable Platform v3), such motherboard has never been commercialized. Research papers evaluating the impact of eADR generally used simulation tools [50].

2.4. Integrating Persistent Memory in Modern Systems

Usual servers architecture can be a barrier in the acceptance of NVMM. This section presents the different ways persistent memory has been used since its commercialization, with pros and cons.

2.4.1. NVMM as RAM extension

In *Memory mode*, NVMM can become an extension of RAM in a machine. This mode makes sense from a budget point of view, as NVMM is considerably less expensive per gigabyte than DRAM.

However, as persistent memory is considerably slower than RAM, a caching mechanism is used in order to keep hot memory pages in RAM, while colder pages are stored in NVMM. This system is imperfect[82], but it allows to run large memory-bounded applications.

Thus, from a conceptual point of view, using NVMM this way does not provide anything new.

2.4.2. NVMM as a disk

Persistent memory can also be used as a regular SSD. One can setup an NVMM module in *fsdax mode*, format in a file system, and use that space to store files. On some disk-intensive applications, this can lead to a spectacular speedup, as Optane DCPMM performance is closer to DRAM than SSD.

Software adaptation

Using NVMM with a standard file system is not the usual way to go with persistent memory. Persistent memory is considered being a new tier of memory in the memory hierarchy, so it does not behave neither as RAM or SSD memories. Its performance stands in between, and considering PMEM as an SSD with no further

adaptation usually results in sub-optimal performance. The explaination resides in software. With time, optimizations have been made to get the full potential of the regular combo DRAM and SSD. We implemented caching mechanisms, such as the Linux Page Cache (LPC) in the Linux kernel. In an SSD file system, it is generally good to have RAM caching, as copying the data to RAM takes a negligible time compared to the actual copy on SSD (around 1,000 times slower). Also, the RAM copy of data can be used as an intermediate interface to modify an isolated byte in a memory mapped file, while the kernel still communicates with the SSD with blocks. Though, when NVMM is used as a disk, copying to RAM is not negligible anymore, as persistent memory is only around 10 times slower than DRAM. Moreover, it is not necessary to "hide" a block-granularity communication with the device, as NVMM can accept direct `load` and `store` instructions, with byte granularity. This is a typical case of software optimization that becomes harmful for performance when applied to the wrong hardware.

DAX File systems

The solution found to use NVMM efficiently in Linux was to add a new feature called DAX (Direct Access). In short, this feature is an option implemented by a file system. When this option is enabled (when the device is mounted), the files located in this file system are not cached by the LPC, avoiding a costly unnecessary copy in DRAM.

Some of the most commonly used file systems have DAX capabilities [32], for instance ext2 [33], ext4 [34] and xfs [35]. The DAX option does not change the inner organization of data inside the file system, as it only requires an additionnal flag when mounting the device.

In addition, when a file system is mounted with the DAX option, its files inherit the DAX flag. A file flagged as DAX can be memory mapped directly into a program's virtual address space, and accessed with the *load* and *store* instructions directly.

2.4.3. NVMM as a DAX areas

Persistent memory being a new tier in the memory hierarchy, it has a unique set of features that makes it usable neither as a disk nor as volatile RAM. This mode of utilization requires to interact with the device as a raw memory space. By setting a persistent memory module in *devdax mode*, it appears to the OS as a raw DAX device.

From a programmer point of view, this DAX device (or DAX file) can be memory mapped into a program's virtual address space, providing a large persistent space to store objects.

For practical reasons, one can also use the *fsdax mode*, format with a DAX-capable file system, mount with the DAX option, create an empty file of an arbitrary size, and use that file as a persistent memory pool. The DAX flag being inherited from the file system, the file can be memory mapped with direct access enabled.

This is the most interesting mode of persistent memory, as it gives the programmer access to all of the features at the same time: persistence and byte addressability. With these two capabilities used simultaneously, one can write programs that store intermediate data in a persistent media, reaching a very high level of crash tolerance. If the program crashes, important data can be retrieved from persistent memory, and the program can continue to run with no data loss at all. However, to use this mode and unlock the full potential of NVMM, existing programs have to be modified.

2.4.4. Working with CPU caches

One of the main challenge regarding NVMM programming is to properly deal with the CPU caches. As these caches have been designed to work with volatile memory, their behavior does not comply with persistent memory requirements.

First, the volatile nature of intermediate caches breaks the persistence guarantee. For example: If you write a value into a variable located in NVMM, you would expect your new value to be immediately stored in the persistent memory. In case of a crash, you would expect to retrieve that new value after restarting the program. In reality, to avoid costly accesses to distant DRAM, the CPU cached your change into its embedded caches. This default behavior is perfectly acceptable with volatile DRAM, as it makes the system faster, and in case of crash or power loss, all of the data stored in DRAM and CPU caches would be lost together. With persistent memory, one could expect that data to be safely saved in the persistent media, while it is in reality still in the volatile cache. If a crash occured, the volatile caches content would be lost, and the initial variable would still appear with its outdated value after a restart.

Then, the way CPUs manage their caches and the communication with DRAM is tainted by this volatility idea. In particular, as CPU caches and DRAM are both volatile, the CPU is free to re-order exchanges between its caches and distant

DRAM. Thus, any crash can let persistent memory in a partially outdated state, likely an incoherent one.

2.5. Programming With Persistent Memory

In order to compensate for this design incompatibility, programmers have to use specific CPU instructions.

2.5.1. Flushing cache lines

The only way to avoid that CPU volatile caches break the persistence guarantee is to manually evict modified cache lines. To that purpose, two commands already existed on Intel x86 instruction set, before the existence of NVMM. These instructions, `clflush` [18] (Cache Line Flush) and `clflushopt` [19] were initially designed to prevent cache pollution. When called, they invalidate the pointed cache line in every level of cache. If the said cache line is marked as containing modifications, the CPU must propagate these changes to the DRAM module before the eviction. By using these instructions with NVMM, one can guarantee the data has indeed been written to the persistent memory.

However, even if these instructions can help with the persistence guarantee, they can severely decrease performance as they completely evict a potentially useful cache line from all of the caches. For instance, imagine a loop of three instructions on the same cache line: `store`, `clflush`, `load`. By evicting the cache line, the clflush operation ensures the `load` instruction will result in a cache miss, thus in a costly interaction with the NVMM module.

For that reason, Intel introduced a new instruction dedicated for NVMM: `clwb` (Cache Line Write Back) in their Skylake SP Series CPUs [20]. This instruction would make no sense with DRAM, as it only ensures the modified cache line is propagated to the underlying media, without eviction of the said cache line. But with NVMM, it gives a persistence guarantee with no perturbation to the cache management, thus no performance penalty for the following instructions.

In some special cases, one could also use the ntstore (non-temporal store) instruction, which is designed to write directly into a DRAM module without going through the caches at all. Its primary use is to write data that is known to be useless for some time, and avoid polluting the caches with such data. It can also be used with NVMM, but its behavior requires some special attention when using multithreaded code, as

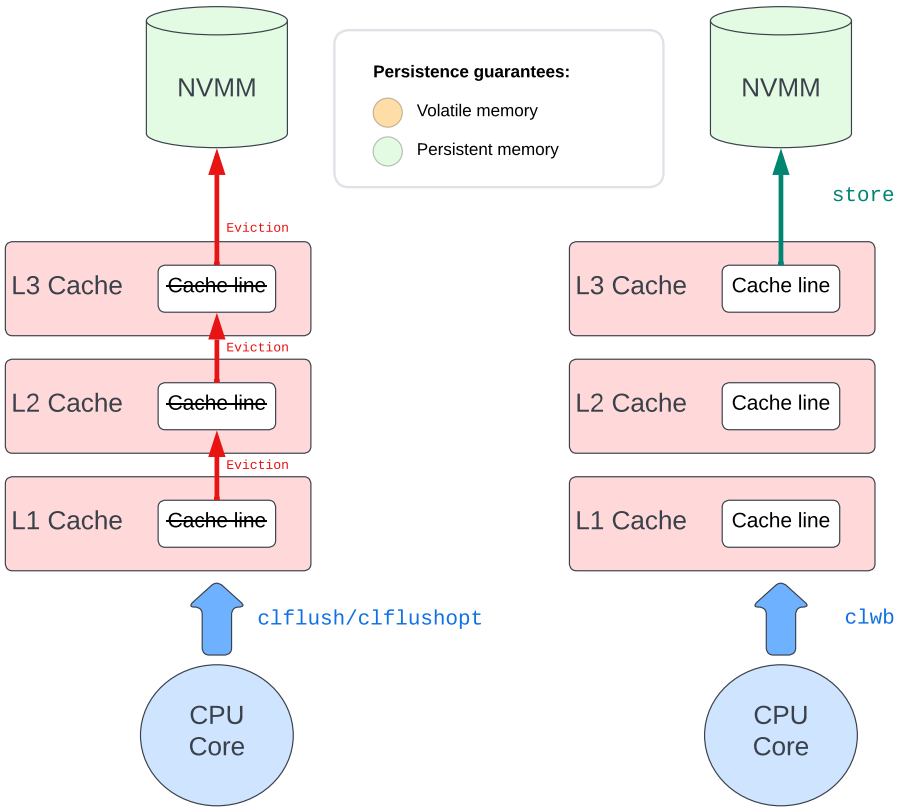

Figure 2.1.: Comparison between the `clflush` and `clwb` instructions.

the visibility of `ntstore` among threads must be ensured manually. However, in the right conditions, it can result in some performance improvements [139].

2.5.2. Ordering and durability guarantees

By design, `clwb` instructions are not ordered between themselves. To ensure the CPU does not break the coherence of data in persistent memory, on can use the `sfence` operation. After using `clwb` on some cache lines, using `sfence` ensures later calls to `clwb` will not be applied before the fence.

The safest way to use persistent memory with these operations is to always combine a call to `clwb` with a `sfence`, enforcing the total ordering of all `clwb` operations. Yet, if some persistent modifications are known to be of the same importance in terms of ordering, a `pfence` can also be issued after all of these modifications are marked with `clwb`.

Additionally to sfence, a developer may need a second fence instruction named `pfence`. This second fence blocks, and waits for all previous written cache lines to be propagated to NVMM. On a modern Intel CPU, `pfence` is also implemented with a `sfence`, because these CPUs support Asynchronous DRAM Refresh (ADR) [54]. With ADR, a CPU uses its residual energy to ensure that a cache-line in the memory controller's Write Pending Queue (WPQ) will actually be persisted before power outage [115].

2.5.3. Persistence model

In our implementations, we used a set of primitives inherited from the persistence model used in the Romulus PTM [27]. Basically, the authors used three functions to generalize the guarantees expected from Intel x86 CPUs. These three functions are:

- Persistent Write-Back (PWB): Asks to write-back a specific cache line in NVMM. This is a non-blocking function.
- Persistent Fence (PFENCE): Emits a fence that prevents next PWB calls to be reordered with previous ones. Still a non-blocking function.
- Persistent Sync (PSYNC): Waits for previous PWB calls to be performed. This is a blocking function.

For machines that only support `clflush`, PWB is set to send a `clflush` while PFENCE and PSYNC are `nop`. Indeed, `clflush` calls are already blocking functions, ordered between themselves.

For more recent hardware, calling PWB emits either `clflushopt` or `clwb` (if `clwb` is available, it is the one to be used). PFENCE and PSYNC are then set to emit a `sfence`.

Chapter 3.

NVCache: A NVMM-based I/O Booster for Legacy Systems

3.1. Introduction

Having Intel Optane DCPMM [59] available in the beginning of 2019 was a milestone for persistent memory researchers. The entire community started to look for meaningfull approaches to use this technology in a concrete context. In comparison with other advanced hardware innovations, Optane had no groundbreaking single feature. Yet, by combining those individual features, and in particular byte addressability with persistence, we obtained a brand new and unique device in the memory hierarchy. This chapter explains how we used Intel Optane DCPMM to build a non-volatile cache, named NVCache.

3.1.1. NVCache in a nutshell

NVCache is a memory cache in userspace. It uses NVMM as a write cache able to receive bursts of data, while the cache content is asynchronously propagated to the main, slower non-volatile media, *i.e.*, Hard Disk Drive (HDD) or SSD. The key idea is to exploit the latency of NVMM, which takes way less time than a regular SSD to acknowledge for the persistence of data. To summarize, NVCache does not need to call `fsync` on the critical path, as the persistence guarantee can be easily obtained in userland by the NVMM device.

3.1.2. NVCache features

In order to build a functionnal and practical tool, we decided to fix some ground rules on the final set of features.

First, NVCache has to be transparent. We want to avoid, as much as possible, modifications in the original source code of the target application. Ideally, we can even

run compiled programs with NVCache without having access to the source code at all.

Second, NVCache must use NVMM as the new device it is, and not as a better SSD or a slower RAM. Persistent memory brings a new feature set, we have to use it fully in order to exploit NVMM at its full potential.

Third, NVCache must be efficient both on throughput and latency. The quest of lower latencies with NVMM should not be persued on the detriment of the throughput. In particular, using NVCache, applications should reach the throughput they could get using batched operations on a regular disk.

The goal of NVCache is to propose this new unique set of features, that cannot be achieved with other NVMM solutions. An overview and comparison with other NVMM software tools is compiled in Table 3.1.

Large storage space

NVCache can handle big datasets, as its storage capacity is not limited to the amount of NVMM available. By using NVMM as a simple cache, the limitation depends on the underlying storage media, which can for instance be an SSD or a HDD. These storage medias are usually considerably cheaper per gigabyte than NVMM.

Syncronous durability

When using NVCache, one can expect the maximum persistence guarantee after each call to the `write` function. All calls to `fsync` are unnecessary, and thus redirected to an empty function. The synchronous durability is the default mode. Technically, a more relaxed logic could be implemented, by placing the right memory fences in the `fsync` function, but this mode has been considered out of scope for the initial development of NVCache.

Durable linearizability

Crash consistency is the major feature we expect from the use of NVMM. However, it requires to add more complex guarantee mechanisms, so that the program could crash at any time and not expect any rollback when restarting. In practice, every `write` function that returns has to be persisted in NVMM, but every incomplete operation must be canceled. Also, there cannot be any reordering among different `store` operations, or the consistency of the cache content could be compromised.

Table 3.1.: Properties of several NVMM systems, all fully compatible with the POSIX API.

	Ext4-DAX [26, 136]	NOVA [138]	Strata [76]	SplitFS [66]	DM-WriteCache [120]	NVCache
Offer a large storage space	−	−	+	−	+	+
Efficient for synchronous durability	+	++	++	++	−	+
Durable linearizability [61]	+	+	+	+	−	+
Reuse legacy file systems	+ (Ext4)	−	−	+ (Ext4)	+ (Any)	+ (Any)
Stock kernel	+	−	−	−	+	+
Legacy kernel API	+	+	−	−	+	+

That type of guarantee is often reffered as durable linearizability [27, 109].

Software compatibility

We wanted NVCache to be as portable and easy to use as possible. As a result, using NVCache does not require deep changes to work on a regular machine. The cache runs on any modern stock Linux kernel, and does not require any kernel-side modification, nor kernel module. Thus, it uses the regular kernel API.

Moreover, the media cached by NVCache can be formatted in any file system. We mostly used the very standard Ext4, but also tested it with more specific file systems such as NOVA [138].

3.1.3. Target applications

By design, NVCache is efficient on applications that require a high granularity of persistence guarantees. A software developer has no choice regarding local persistence: to ensure some level of crash resilience, one has to call `fsync` after each critical `write`. With NVCache enabled, such an application would not suffer from a major Input/Output (I/O) bottleneck even with high persistence requirements.

Current applications tend to leverage this problem by batching modifications, and wait for the batch to be complete before flushing it to disk. That approach dilutes the frequency of `fsync` calls. However, it does require to find a trade-off between performance and persistence granularity. Indeed, increasing the batch size will tend do increase performance, but it can also result in bigger rollbacks when a crash occurs. Similarly, reducing the batch size makes the system slower, but reduces the likeliness and the importance of potential rollbacks.

Thus, NVCache is a perfect fit for applications that are sensitive to the persistence latency, *i.e.*, the time required to ensure a data has been persisted. For instance, a Database Management System (DBMS) with regular updates on a massive database. In an ideal context, the database would be updated on disk for each `write` or `update` request. However, under heavy workloads, the time required to wait for the disk to acknowledge after each operation would create a severe bottleneck. If the database fits in Intel Optane DCPMM, one could imagine using this device as the only storage media. But this solution requires to have a lot of NVMM available, which is expensive [45] or even impossible over a certain amount. On the other hand, with NVCache we proposes a way to get the latency benefits from NVMM while not being limited by the amount of persistent memory in the server.

3.2. NVCache: Implementation

3.2.1. Overview

NVCache is available either as a library or as a modified `libc` [36]. The library that can be included in a target application, providing I/O primitives such as `nvopen`, `nvwrite`, `nvread`, and `nvclose`. The `libc` can be dynamically linked to a compiled program. This way, it transparently intercepts calls to basic IO functions, like `open`, `read`, `write`, and `close`.

In case of a crash, some data might remain in the non-volatile cache. On reboot, after loading NVCache again, a synchronisation phase starts, setting the disk in a state that would be considered valid by the application. The program then has to recover its data from disk and restart exactly as it would on a regular machine.

3.2.2. NVLog

The main idea in NVCache is to use the NVMM space as a log of pending operations. Each log entry represents a write operation that will eventually be applied on disk.

On startup, a NVMM module in Device DAX is memory mapped, and this virtual space is casted into a data structured that we called NVLog.

Index	fd	off	size	data	written	commit
1	5	0	12	`'Hello World!'`	0	1

Figure 3.1.: NVLOG entry example

Log entry format

In order to delay the actual write operation, the NVLog has to keep the entire set of metadata provided by the application when the `write` function was called. A log entry must contain:

- A file descriptor (`fd`)
- An offset (`off`)
- A size (`size`)
- The data to be written (`data`)

As log entries are statically allocated to maximize performance, they also come with a fixed size. When a `write` operation is shorter than the maximal `data` size (*i.e.*, 4 KiB), the `size` field guarantees the end of the `data` buffer will be ignored. When the operation is longer than 4 KiB, it has to be splitted in several parts. For instance, a 9 KiB `write` will result in two 4 KiB entries, plus a smaller 1 KiB one.

These four fields already ensure that we will be able to execute the `write` operation later. Though, in order to make NVCACHE crash resilient, we had to add two more fields:

- A commit boolean (`commit`)
- An reference index for long writes (`waiting`)
- An ignore boolean for entries already written to disk (`written`)

The `commit` boolean is the last field to modify. If a crash occurs, a non-committed entry is considered incomplete, and thus will not be synchronized on disk when the system restarts. The reference index is the index of the first log entry of a long write. With this field, we know what commit flag to look at when a write is splitted in several chunks. Last, the `written` flag allows to ignore entries that have already been written to disk, for instance if the file has been closed. As writing to an outdated file descriptor could cause errors, there is a specific procedure when the hosted program closes a file. That procedure requires to flag some entries as `written`.

	Index	fd	off	size	data	written	commit	
	1	5	0	6	`'Hello '`	1	1	⟵`tail`
`head`⟶	2	5	6	5	`'World'`	0	1	
	3	5	11	1	`'!'`	0	0	

Figure 3.2.: NVLOG `head` and `tail` counters

NVLOG data structure

On NVCACHE startup, a DAX space (our NVMM) is memory mapped. This space is then hosting the NVLOG data structure. It stores :

- log entries (the amount is defined at compile time, we call it `log_size`)
- A `head` counter
- A correspondence table between file paths and file descriptors
- A `exit_status` boolean

The log itself is circular: when it reaches the end of the N allocated log entries, it starts back at the first one. The `head` and `tail` counters respectively keep in NVMM the index of the last entry written, and the index of the last entry already flushed to disk. As `head` is not mandatory for recovery, this value is kept in volatile memory. A correspondance table gives a path to each file descriptor used in the log, so that the program can open them again even after a crash. Last, the `exit_status` flag is set to 1 when NVCACHE starts, and set to 0 after exiting in a normal way. This is basically a crash detector for the recovery procedure.

Essentially, this data structure is designed to store the bare minimum required to recover in case of crash. Every variable that is not absolutely necessary to the recovery procedure is stored in regular volatile memory.

Adding a new entry

In order to add a new entry in the non-volatile log, the user program calls regular `libc` functions to write into a file, *i.e.*, `pwrite` or `write`. Instead of submitting a `write` system call immediately to the kernel, the following events are triggered :

- Allocate a new (or recycled) log entry in the log
- Fill the entry content as described previously, except for the `commit` field
- Emit a PWB per cache line and a PFENCE

- Set the `commit` boolean to `True`
- Emit a PWB on the commit boolean cache line, and a PSYNC

This sequence ensures the commit boolean will never be set to `True` before the log entry content is written in the non-volatile log. Without this security, after a crash, a `commit` boolean could be set to `True` while its content was not (or only partially) written in NVMM. However, as we took this precaution, `commit` can be used as the answer of the question: Is this log entry ready to be written on disk?

Log entry allocation

Before storing the content of an entry, NVCache has to allocate it. In this context, the objective is to increase the index stored in the `head` counter.

In order to anticipate multithreaded workloads, this mechanism is implemented in a lock-free fashion. In practice, NVCache starts by attempting to atomically decrease the `available_entries` counter (this counter is kept in volatile memory). This requires the use of a `compare_and_swap` atomic function, checking that no other thread changed the value of `available_entries` in the meantime.

If this greater than zero, meaning there is space left in our log, the thread manages to decrease its value. Otherwise, it spins until the decrement is effective. NVCache then calls `fetch_and_add(1)` on the `head` counter, and returns the new value as the index of the allocated entry.

However, as the `fetch_and_add` function does not take into account the circularity of our log, the `head` and `tail` counters are monotonic. Before using them, NVCache has to apply a modulo by the `log_size`.

One could argue that having these counters monotonic can lead to a problem when they reach overflow. Though, `head` and `tail` being implemented as 64 B variables, NVCache would have to store $4KiB * 2^64 = 64EiB$ to reach this overflow. Thus, we considered this problem as out of scope for our implementation.

Overlapping entries

As soon as a write request is longer than the maximum size of a log entry payload (in practice, 4 KiB), NVCache has to split it in several log entries. Nonetheless, to prevent partial writes to pollute the disk, an additional mechanism is implemented for overlapping entries.

Algorithm 3.1.: NVCache `write` function.

```
1  struct nvram {                                                    // Non-volatile memory
2    struct { char path[PATH_MAX]; } fds[FD_MAX];
3    struct entry entries[NB_ENTRIES];
4    uint64_t persistent_tail;
5  }* nvram;

7  uint64_t head, volatile_tail;                                     // Volatile memory

9  void write(int fd, const char* buf, size_t n) {
10   struct open_file* o = open_files[fd];
11   struct file* f = o->file;
12   struct page_desc* p = get(f->radix, o->offset);

14   uint64_t index = next_entry();
15   struct entry* e = &nvram->entries[index % NB_ENTRIES];

17   acquire(&p->atomic_lock);

19   memcpy(e->data, buf, n);                                        // Write cache
20   e->fd = fd;
21   e->off = o->off;
22   pwb_range(e, sizeof(*e));                          // Send the uncommited entry to NVMM
23   pfence();                                                  // Ensure commit is executed after

25   e->commit = 1;
26   pwb_range(e, CACHE_LINE_SIZE);                                  // Send the commit to NVMM
27   psync();                                                      // Ensure durable linearizability

29   atomic_fetch_add(&p->dirty_counter, 1);                         // Read cache
30   if(p->content)                                  // Update page if present in the read cache
31   memcpy(p->content->data + o->off % PAGE_SIZE, buf, n);
32   release(&p->atomic_lock);
33 }

35 int next_entry() {
36   int index = atomic_load(&head);
37   while((index + 1) % NB_ENTRIES == atomic_load(&volatile_tail)) ||
38   !atomic_compare_and_swap(&head, index, index + 1))
39   index = atomic_load(&head);
40   return index;                                  // Commit flag at index is 0 (see cleanup thread)
41 }
```

	Index	fd	off	size	data	waiting	written	commit	
	n	5	0	4096	`First 4096 bytes`	n	0	0	⟵`tail`
	$n+1$	5	4096	4096	`Next 4096 bytes`	n	0	1	
`head`⟶	$n+2$	5	8192	8	`Last 8 bytes`	n	0	1	

Figure 3.3.: State of NVLog while adding 8200 bytes.

We leverage this problem by adding the `waiting` field in each log entry. On single entries, this field is set to -1. On multiple entries, it stores the index of the first one. The first entry of an overlapping entry is the one used as a reference for all of its followers. The sequence to follow to add an overlapping entry is the following:

- Compute the number of log entries to allocate (k)
- Allocate these k entries
- Write the first entry, with a `waiting` index pointing to itself, and a `commit` boolean to `False`
- Write the $k-1$ following entries, with a `waiting` index to the first one, and `commit` boolean set to `True`
- When all entries are written, perform a `sfence`
- Change the first entry `commit` flag to `True`
- Call `sfence` and `clflush`

This sequence allows to keep overlapping writes atomic, as they cannot be partially propagated to disk.

3.2.3. Cleanup thread

While NVCache is writing new entries in the log, a background thread is responsible for flushing these log entries in the actual non volatile storage media.

This thread interacts with three elements of the NVLog:

- The `tail` counter, index of the last flushed entry
- The `written` field of log entries
- The `available_entries` main counter, keeping track of the number of entries that can be allocated.

As entries are added to the log at the `head` index, the cleanup thread is responsible for synchronizing these entries with the disk. It uses the `tail` index, stored in volatile memory, to call the standard `libc` I/O functions, and submit each entry as a write system call to the backend file system.

Interaction with the LPC

When an entry is processed by the cleanup thread, it is sent to kernel space in order to be written on the physical disk. Though, for performance reasons, calling the `write` system call alone does not give any persistence guarantee. Instead, data is sent to the volatile LPC, opportunistically waiting for a moment to write in non volatile storage. From userland, the usual way to make sure every pending write operation is propagated to disk is to call the `sync` system call. As this system call returns, the user is guaranteed that its data has been persisted on disk.

Ideally, data in our cache should never transit back into a volatile storage before reaching its final persistent storage of destination. However, there are two reasons why this is not the technical choice made for NVCache. First, there is no way from user space to write into a device file system without going through the LPC. Except using a Filesystem in Userspace (FUSE) file system, which is not as efficient because of multiple context switches for each operation. Second, we can use the LPC design at our advantage to optimize some of our disk throughput, and in particular by using a batching strategy.

Batch strategy

From user space, NVCache first has to call the `sync` system call, ensuring a log entry has been written to disk. Then, it can free the log entry (by setting the `written` flag).

Yet, flushing NVLog entries one by one requires to call `sync` for each of them. While this was the initial implementation of NVCache, it suffered from a massive throughput bottleneck due to the user space constantly waiting for the kernel to flush pages.

Instead, NVCache makes batches of writes and only send one `sync` command after each batch. This strategy considerably increased the throughput, and this can be explained by several factors.

First, the LPC can opportunistically write to disk when it is not used, which means it can start writing on disk while the batch is still being submitted by the user space.

In comparison, calling `sync` between each log entry implies an additionnal context switch per entry, and adds the latency required to ensure the LPC is synchronized with the disk on the kernel side before returning.

Second, as the LPC organizes its space in pages, if two modifications of the same page are in the same batch of entries, they may be applied to the same page in DRAM. In this case, it may result in only one actual transaction to the disk, optimizing the available throughput.

Flush policy

As described previously, our NVCache cleanup thread is responsible for flushing entries from persistent memory to the disk by batches. Using NVMM as an intermediate between the application and the disk also gives NVCache the ability to limit problems when the machine crashes. In particular, it avoids partial writes on disk by implementing a redo-log behavior.

While the application is writing data into the NVLog, this mechanism is implemented using the `waiting` flag. The cleanup thread has to wait for the first entry of a long write to be commited, which is only set when all the entries have been appened to the log.

Once the first write entry is committed, the cleanup thread can embed all of the entries for the next batch to be flushed. As soon as the kernel guarantees the last entry of the entire write has been flushed, all of the entries are marked as `written`.

This strategy ensures that a crash would never result in a partial write on disk. Technically, a partial write could occur when the cleanup thread is submitting entries to the kernel. However, as they are not marked `written` before the last one is, a partial write on disk would inevitably end in a re-submission of all of this write entries.

Recovery procedure

The particularity of NVCache is that it splits the persistence domain in two different storage units. Only the two persistent areas together can ensure a coherent state.

In case of a crash, the `exit_status` boolean would stay at 1. If the user restarts the program and loads NVCache, on initialization, NVCache would detect a malfunction happened in the previous run and would ask the user if they want to start

a recovery. The NVLOG is always either empty or ready to be synchronized with the underlying disk through the recovery procedure.

As the NVLOG data structure in NVMM contains the `tail` pointer, the translation table between file descriptors and their respective paths to each file, and the log entries yet to be flushed, we implemented a recovery procedure that flushes all pending modifications to the actual disk. This synchronization step is only responsible for emptying the cache, and goes back to a regular state on disk. One could decide to restart the application without NVCACHE after a recovery, and there would be no difference with a regular restart of the said application on a regular system.

3.2.4. Write-only performance

By combining the previous elements of NVCACHE, we obtain a crash resilient write cache. For the moment, the cache is only unidirectional, as we can only use it in the write direction.

3.2.5. Ensuring consistency

In this state, NVCACHE can only handle `write` operations. If an application tries to read data, the request would be submitted to the kernel, which would fetch the content from disk. However, there are two major failures that could happen in this situation. First, if the memory area has been modified and the modification is still pending in the NVMM log, reading from disk would result in getting outdated data. Second, as the cleanup thread is constantly writing to the disk, there is a risk the page to read could be modified simultaneously. As the LPC does not guarantee the atomicity of I/O operations, reading data while the cleanup thread is active remains a synchronization challenge. For these reasons, there is no easy way to get such an asymetric cache.

The paging dilemma

As described in Section 3.2.2, the NVLOG in NVMM only keeps track of pending modifications. These can apply on unaligned offsets inside our files, which means offsets that are not a multiple of the LPC standard page size (4 KiB). Transforming NVCACHE in a more read-friendly system could be achieved by keeping full pages of 4 KiB in the cache, ready to be read as is. As a matter of fact, this is exactly how the LPC manages files, by splitting them in 4 KiB aligned pages. However, this choice comes with a some disadvantages. First of all, accepting modifications

inside pages requires to frequently read inside the disk. Indeed, if the modified page is not cached yet, it requires to retrieve the content the page from the disk first. Second, the amount of data to transfer to disk, even for a small modification, is always a multiple of 4 kB, making the cache less efficient when used for small `write` operations.

This comparison between logging and paging in NVMM will be made in Chapter 4.

Regarding NVCache, the decision has been made to maximize the write cache performance. That is why we kept a log-based use of the NVMM.

Complementary DRAM cache

The ideal solution to ensure coherence when reading is to add a complementary page cache in DRAM. By manually keeping up to date pages of data, NVCache is able to answer to read requests, while staying un user space. By design, a cache is only covering a subset of the real data area. Thus, this additionnal mechanism is not able to avoid cache misses in all scenarios. However, it is able to probabilistically reduce the amount of cache misses, and therefore, the time required to read in NVCache.

DRAM cache interaction

This section covers the behaviour of the custom DRAM cache build in NVCache.

Read. The integrated DRAM cache is accessed whenever a read request is sent by the hosted program. If the requested data is already in the cache, it is a cache hit, and NVCache answers directly. But if the said data is not in cache yet, the cache miss procedure is triggered.

Cache miss. When a page is not found in the cache, the cache miss procedure reads it from disk. But in the context of NVCache, reading from disk is not sufficient to guarantee that the read data is coherent. Here is a very likely scenario that could lead to incoherence: a page recently modified by a write request has a modification pending in the NVLog. That page is not in the DRAM cache. When the program tries to read that page, it triggers a cache miss procedure. Given these circumstances, reading from the disk would fetch data that is now outdated, as the pending modification has not been applied on disk yet. In our implementation, this precise scenario would call another more complex cache miss procedure, that we called `dirty miss`. This very specific procedure will be detailed in section 3.2.6.

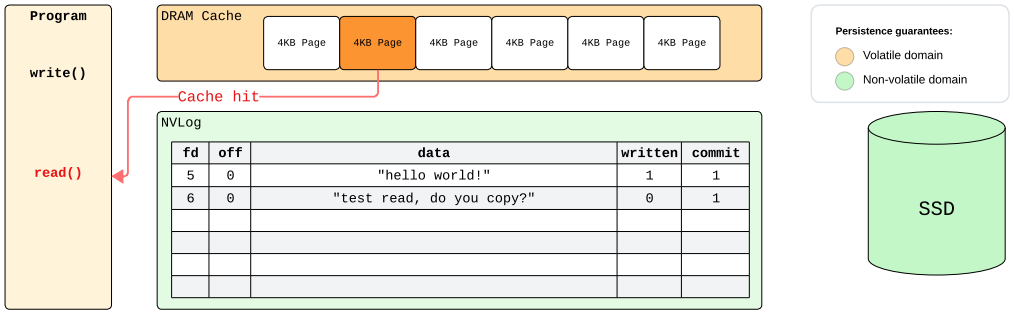

Figure 3.4.: NVCACHE behavior in case of a cache hit.

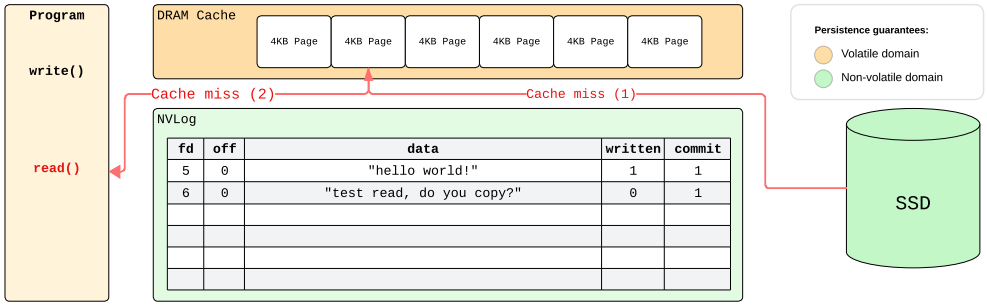

Figure 3.5.: NVCACHE behavior in case of a cache miss.

Independantly of the kind of cache miss triggered, the data page is eventually retrieved. In order to make sure this page stays available for potential future read requests, it is also added to the page cache of NVCACHE.

Write. The key idea of having the DRAM cache is to make sure we can read from it at any time. As any caching system, the cost of a cache miss is considerably higher than the cost of a cache hit. This is particularly true for NVCACHE, as it can also trigger an even more costly `dirty miss`. However, by ensuring the DRAM cache stays up to date, we can avoid many calls to the `dirty miss` procedure. For that precise reason, every write submitted to NVCACHE also checks if a page in the DRAM cache has to be updated. If the page is not already in the cache, this mechanism is skipped, as there is no reason to believe the program would read the page it just wrote if the said page has not been read recently. However, if the page is in the

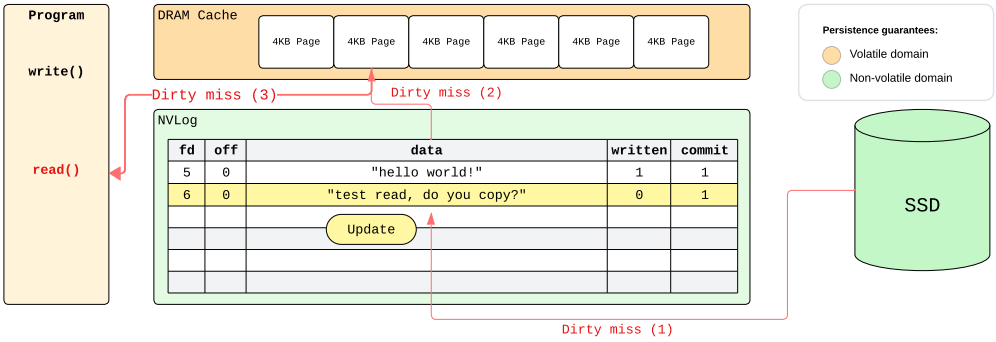

Figure 3.6.: NVCache behavior in case of a “dirty miss”.

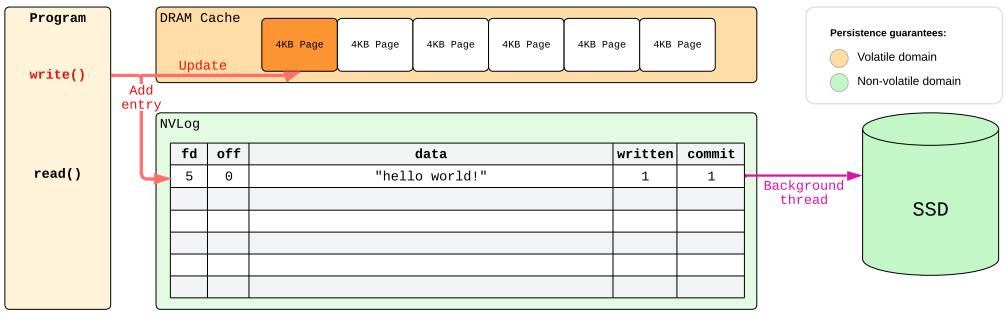

Figure 3.7.: NVCache behavior in case of a write.

cache, meaning that it has been read recently, the page is updated and ready to be read again.

Page management

In order to keep the most used pages in our DRAM page cache, NVCache uses the Least Recently Used (LRU) eviction policy. The page cache itself is a chained list of `pages` that keeps in first position the latest page accessed, while the last one is always the least recently used one. Having this organisation allows to easily identify and recycle the least useful page when the cache is full and we need to insert a newer page.

Effect of the DRAM cache

The goal of maintaining a complementary page cache in DRAM is to reduce the probability of complex cache misses. As NVCache keeps pending modifications in its userland non-volatile log, there is a time frame during which the system is not aware that change will have to be applied. In this scenario, if the program tries to directly read from a cached file, it would obtain outdated data coming from the disk or the LPC. That is why we added a custom page cache with NVCache, that keeps its pages updated all the time and thus can answer read requests without compromising data coherence. Nonetheless, there are still some extreme cases where the up to date page will not be in the DRAM cache anymore, and the modification is still pending in the non-volatile log. In these cases, NVCache has to proceed to a `dirty miss` procedure. As this event is quite slow, increasing the size of the DRAM cache can reduce its probability of happening.

3.2.6. Control structures

Some of NVCache components are key data structures that help achieving the aforementioned features. Such elements deserve a proper description in this section.

Radix Tree

Inspired by the LPC implementation in the Linux kernel, our page cache is accessed using a Radix tree. Radix trees are known to be efficent for page management systems, as they guarantee a small memory footprint while ensuring a low latency of answer. More specifically, a radix tree has a low latency on cache misses, as a failed lookup will return in a time shorter or equal to the time for a valid lookup.

In our implementation, a radix tree is created for each file opened. Then, the key is the number of the 4 KiB page inside this file, which is the aligned offset divided by 4 KiB. The key (an offset, 64 bits unsigned integer) is subdivided in 8 bits parts that represent the granularity of our radix tree stages.

When performing a lookup, the answer can either be a null pointer or a pointer to a data page.

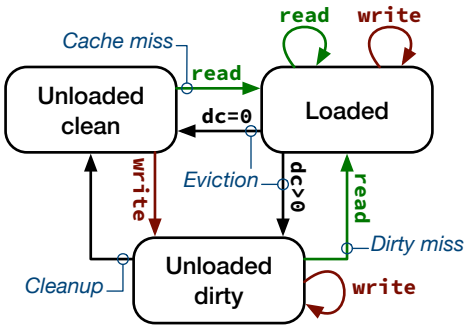

Figure 3.8.: State machine of pages (*dc:* dirty counter).

Page descriptors

Technically, the leaves of our radix tree are not directly data pages, but instead, they are page descriptors. In a page descriptor, several control fields are defined:

1. A lock
2. A `dirty` counter
3. A pointer to the actual page

This page descriptor allows to keep track of a page state regarding its modifications pending in the NVLOG. The `dirty` counter is an integer, and it provides an easy access to the number of modifications in the log one would have to apply in order to get this page up to date. When a cache miss occurs, and this counter is not zero, then a "dirty miss" procedure is triggered.

The lock ensures the synchronization between the interface exposed to the application and the background cleanup thread.

Pages state machine

The behavior of each page can be summarized in a simple state machine.

Synchronization

By design, NVCACHE must contain some internal synchronization mechanisms. Seen from the application, using NVCACHE API must ensure the same level of guarantees mentioned by the Portable Operating System Interface (POSIX) standard.

However, we definitely do not want to ensure more guarantees (except the I/O persistence one). For instance, our interface must not provide any supplemental guarantee regarding thread safety and data consistency, as such mechanisms would lower our performance in comparison with a regular system. If we keep the spirit described by the POSIX standard, these securities are the concern of the programmer. Thus, there is no protection regarding writing at the same moment in the same page, which would undoubtably end with an inconsistent state in memory.

While the user interface remains unchanged, the interaction with the kernel has to be adapted to the inner functions of NVCache. In a regular machine, writing to the disk requires to call `write`, and a `fsync` that guarantees the data is safe in case of a crash. But with NVCache standing in between the program and the actual disk, the logic is different. First, the `write` call itself is sufficient to get the persistence guarantee[62]. Second, the `fsync` call is ignored, as it is redundant with the guarantee provided by `write`. Then, as soon as the program has received the guarantee, it is NVCache responsibility to get the data to the disk safely, while being crash resilient. In this context, the cleanup thread is responsible for synchronizing the NVLog of pending `write` requests and the disk. We did not provide any additionnal guarantee of thread safety between the program threads, however, we cannot introduce perturbations to the main threads because of the cleanup thread, and this problem lead to some more complexity in NVCache.

Let us imagine the following scenario: a main thread, from the application, writes 4 KiB of data in a file with NVCache's API. A new log entry is added to the NVLog. At the same time, another thread asks to read that precise page.

- **Option 1: Clean miss.** The cleanup thread already applied the log entry to the disk. Reading will result in a cache miss, and as the `dirty counter` of that page is at zero (meaning no log entry has to be applied) the page is read from disk, added to the LRU cache and returned to the user.
- **Option 2: Dirty miss.** The cleanup thread is late, and has not applied the log entry to disk yet. As the `dirty counter` of that page is at 1, the cache miss becomes a dirty miss, and the page has to be read from disk and updated before it is loaded in the LRU cache and returned to the user.
- **Option 3: Inconsistency.** The cleanup thread is currently writing the page to disk, but has not finished yet. The main thread tries to read, but the page in not in the LRU cache, triggering a cache miss. It is read from disk and... We just read data while it was being modified by the cleanup thread: the result may be a mix of the old and the new page.

Options 1 and 2 are perfectly acceptable, as they virtually react as we would expect from a POSIX compliant interface. But option 3 creates a very unpredictible behavior, and breaks the guarantee NVCACHE gave to the program when returning from the `write` call.

In order to avoid this tricky scenario, each page is equipped with a lock.

When the cleanup thread is about to apply modification on a batch of pages, it first tries to lock as many affected pages as possible. If a locked page is found before the maximum batch size, the thread stops acquiring locks and flushes the batch of pages already locked. Optimistically, during the time the partial batch is flushed, the locked page would be unlocked and another batch could be started. In the mean time, if the main application thread triggers a cache miss procedure (dirty or not), the lock has to be taken on the given page before reading from the disk. When the page is already loaded in the DRAM cache, there is no NVCACHE specific synchronization issue, as this cache behaves the same way as the LPC. The only synchronization conflicts that could happen with the DRAM cache are, as designed in POSIX, the responsibility of the application programmer.

3.3. Evaluation

We evaluate NVCACHE on two applications. The first one, Flexible I/O tester (FIO), is a performance measurement tool that can evaluate the reachable performance of several components of the I/O chain. The second one, RocksDB, is a key-value store based on Google's LevelDB and currently used by companies such as Facebook or LinkedIn.

3.3.1. Experimental setup

Hardware

Our benchmark machine is a Supermicro dual socket server with two NUMA domains. Each socket is equipped with an Intel Xeon Gold 5215 CPU. These CPUs run 10 physical cores (20 hardware threads) at 2.50 GHz. Each of these CPUs memory channels are populated with 6×32 GiB of DDR4 DRAM and 4×128 GiB of Optane NVDIMM v100. Overall, this machine has 40 hardware threads, 384 GiB of DRAM

Table 3.2.: Evaluated file systems.

Name	Write cache	Storage space	FS	Synchronous durability	Durable linearizability
NVCache +SSD	NVCache	SSD	Ext4	by default	by default
DM-WriteCache	kernel page cache	SSD	Ext4	O_DIRECT | O_SYNC	no
Ext4-DAX	kernel page cache	NVMM	Ext4	O_DIRECT | O_SYNC	no
NOVA[1]	none	NVMM	NOVA	O_DIRECT | O_SYNC	by default
SSD	kernel page cache	SSD	Ext4	O_DIRECT | O_SYNC	no
tmpfs	kernel page cache	DDR4	none	no	no
NVCache +NOVA	NVCache	NVMM	NOVA	by default	by default

and 512 GiB of NVMM. The machine has two Serial Advance Technology Attachment (SATA) Intel SSD DC S4600 of 512 GiB. The main SSD contains the system, while the second is used for the experiments.

Software

The machine runs Ubuntu 20.04 with Linux version 5.1.0 (NOVA[138] repository version) and musl[40] v1.1.24, revision 9b2921be. For simplicity, benchmarks are run inside docker containers, with the Alpine Linux[25] distribution. As this distribution is designed around the use of the musl libc, all of the packages pre-compiled in its repositories are natively compatible with our modified musl C library. Both SSDs are formatted in Ext4.

NVCache parameters

Unless stated otherwise, we configure NVCache as follow. Each entry in our NVM log is 4 KiB large. The log itself is constituted of 16 million entries (around 64 GiB). The RAM cache uses 250 thousand pages of 4 KiB each (around 1 GiB). The minimum number of entries before attempting to batch data to the disk is 1 thousand. The maximum number of entries in a batch is 10 thousand. When log entries are flushed, depending on the configuration, they are either propagated to the test SSD formatted in Ext4, or to another PMEM module formatted with NOVA.

Comparison with other systems

One of the most important obstacle in evaluating NVCache performance, is to make fair comparisons with other systems. We selected 2 configurations with NVCache, and 5 configurations without, as baselines. The purpose of adding a NVCache +NOVA (on PMEM) configuration is to check if NVCache performance

is, as expected, limited by the throughput of the underlying device. By comparing this setup with the regular NVCache +Ext4 (on a SSD), we expect to measure better performances when the log is emptied in a faster device.

SSD (ext4). This configuration is from far the one expected to be the slowest. It consists in a regular ext4 filesystem on the machine SSD. As it would not make a fair comparison without guarantees, the benchmark has to call `fsync` after every write operation, which is considerably slow. Even if this is not really the expected way to use such a configuration, this is a good baseline to show what having NVCache-tier guarantees would cost on a regular machine.

Tmpfs. This is the kind of file system used to store temporary files in Unix-like operating systems. It is stored in DRAM, and thus, does not provide any persistence guarantee.

Ext4-DAX. Ext4 system is one of the most popular journaling file system for modern Unix-like operating systems. It is initially meant to be used on a HDD or SSD, which explains why is uses the LPC. However, when deployed on NVMM, the LPC tends to slow down operations instead of making them more efficient. That is why Ext4 can be used in DAX mode with NVMM, skipping the useless copy of data in the LPC.

Dm-Writecache. A popular implementation of a write cache in the linux kernel. It does not cache reads, only writes.

NOVA. This is one of the most popular file systems released for NVMM. It is DAX-capable by design. In order to get the best performance out of Intel Optane DCPMM, it has been designed as a log-structured file system[133].

NVCache on SSD. In this configuration, NVCache is used on a standard SSD, formatted in a standard Ext4 file system. The main advantage is that the storage space available is not limited by the amount of NVMM available.

NVCache on NOVA. This setup is not meant to be used in real conditions, but it is a good way to evaluate what part of NVCache measured performance is a consequence of the underlying device raw performance. It does not offer a large storage space like NVCache with an SSD, but shows the theoretical performance NVCache could reach with a more efficient secondary storage.

All of these systems and their properties are summarized in Table 3.2.

3.3.2. Benchmarks

NVCACHE performance has been evaluated with several benchmarking tools.

FIO: Flexible I/O tester

FIO[4] is a benchmark application, designed to measure various I/O performance indicators of bandwidth and latency, while providing an easy interface to generate many kinds of workloads. We used FIO to simulate I/O intensive workloads with and without NVCACHE, while changing either the parameters of the workload generator or the configuration of NVCACHE.

RocksDB benchmark

RocksDB is a DBMS based on Facebook's LevelDB. More precisely, it is an embedded database coded in C++, meaning it has to be integrated into an application to be used.

Is is used by large scale companies such as Facebook, Yahoo! and LinkedIn[13]. As a demonstration, the RocksDB source code can be compiled as a benchmarking tool named `db_bench`. We evaluated the performance of RocksDB's `db_bench` with the different configurations we wanted to compare NVCACHE with.

SQLite

SQLite [22] is an embedded relational database engine coded in C. It is a very ligh-weight and yet fully fledged library, used from very popular software (Firefox[98], Thunderbird[99], etc.[64, 74]) to embedded systems. We evaluated SQLite v3.25, with a port of the `db_bench` benchmarking tool[125].

3.3.3. Read-oriented workloads

In all our experiments, read performance on SQLite and RocksDB is roughly equivalent for all systems3.9. This behavior is not surprising, as both DBMS use an internal DRAM cache. Actually, many DBMS use similar mechanisms to avoid dealing with the LPC. They use the `O_DIRECT` flag[91] when they open the database file, and manage their own local DRAM cache. However, this behavior is not a problem for our writing tests, as the cache has to be regularly (or manually) synchronized with the persistent media.

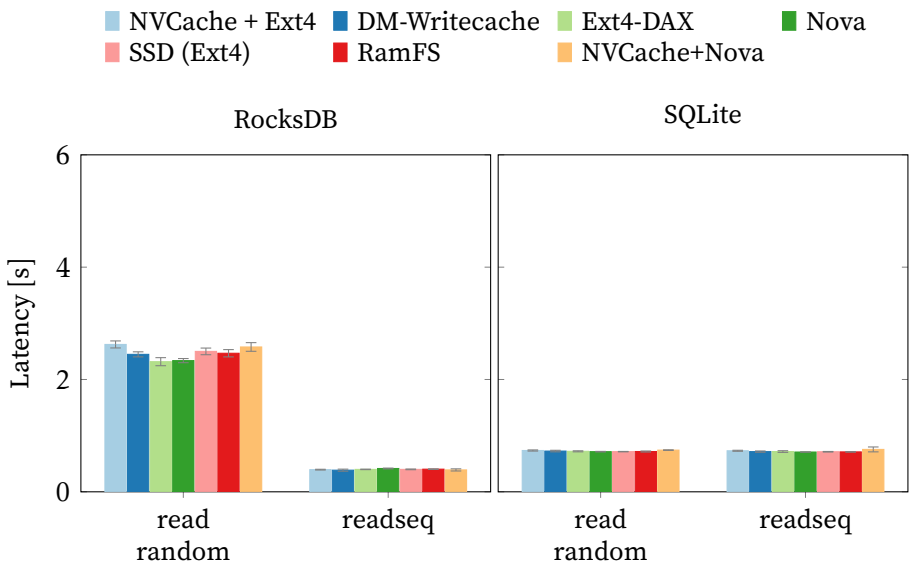

Figure 3.9.: Performance of each system under SQLite and RocksDB read intensive workloads

3.3.4. Write-oriented workloads

Tuning NVCache with FIO

In the following experiments, we evaluated the impact of changes in NVCache's configuration. First, in figure 3.10, we study the parameters related to writing in the NVLog. By changing the length of the NVLog, we expect NVCache to reach higher performance in write bandwidth, as the amount of data that can be cached to NVMM increases. Increasing the log size delays the moment the NVLog becomes full, and thus, relies on the SSD performance. In this test, FIO is used in write-only mode.

As expected, when NVCache can absorb more data in its log, it reaches better performance. The two first plots of figure 3.10 show the bandwidth and latency reached by the benchmark over time. The third one is a like a progress bar, showing the advancement of each benchmark to write the 20 GiB of data. Thanks to these graphs, we can identify the different phases of the benchmark, depending on the physical support receiving the data. In particular, the 8 GiB log shows a clear rupture between the first phase, during which the benchmark can write in the NVMM cache,

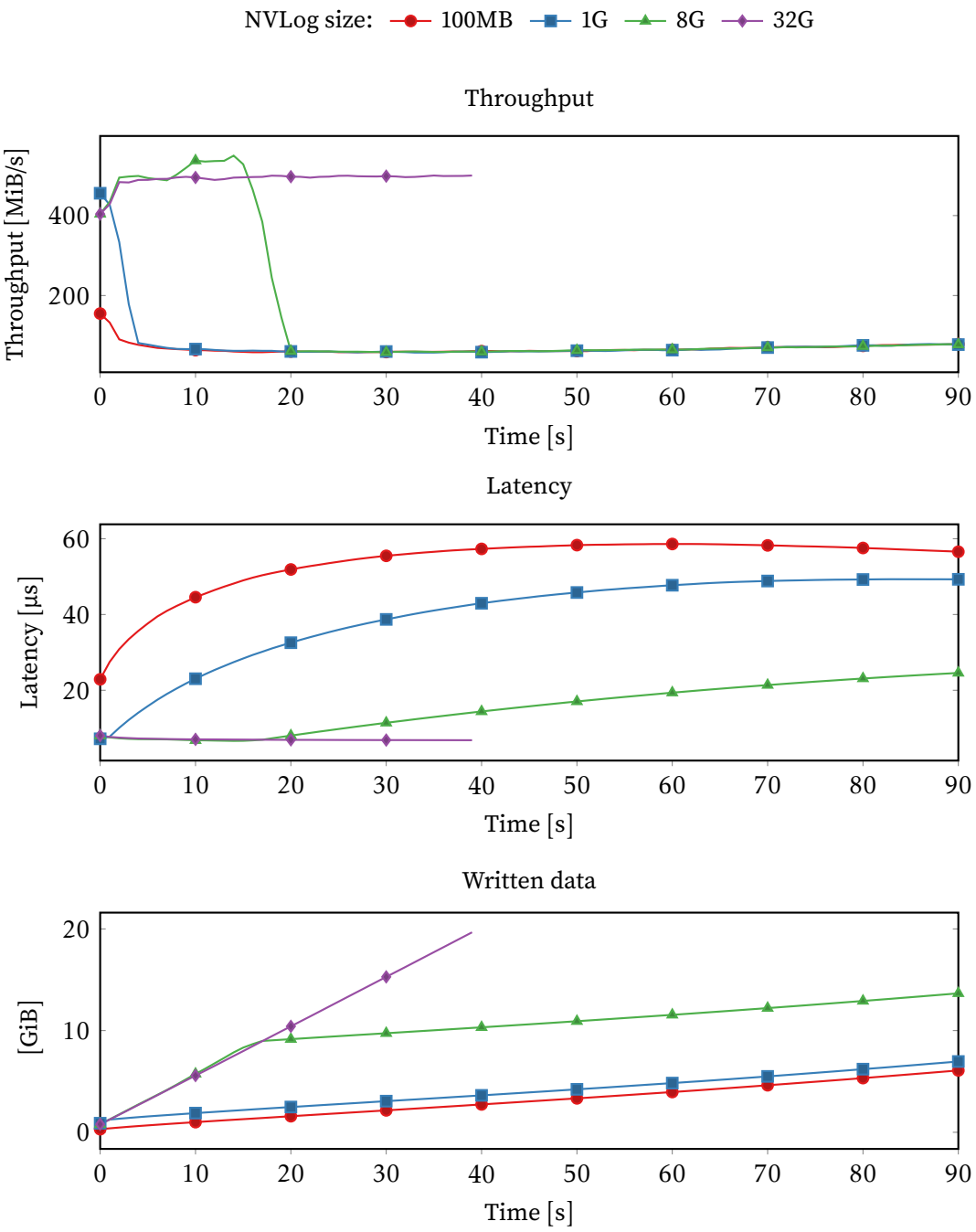

Figure 3.10.: Performance of NVCache under random write intensive loads for 20 GiB, with variable NVMM log size.

and the second phase, when the cache is full, the disk becoming the performance bottleneck.

In real conditions, one could use NVCache with a smaller log size to preserve the amount of NVMM used. However, in our case, we focus on analyzing NVCache performance in the caching phase. As soon as the cache is full, the performance of the cached SSD is to be expected, and cannot be exceeded.

On the same write-only benchmark, we evaluated the effect of our batching mechanism. This system is expected to be more efficient when the batch size (the number of entries written to disk before a call to fsync) increases. Indeed, as we are submitting write requests to the LPC, asking for guarantees less often reduces the time spent waiting on the disk to send a confirmation our data has been written. Moreover, if modifications to apply on the same page are in the batch, they will be applied in RAM and thus merged before the final page is sent only once to the disk.

During the first phase (when the log is not full), the batch size has no effect on the performance. In the second phase, there is a clear improvement of the bandwidth and latency when the batching mechanism is enabled. However, this improvement quickly reduces while the size of the batches increases. That experiment explains why there is no need to select a very high batch size, as in a mixed I/O situation, the amount of locks to acquire could actually hurt performance if this parameter is too high. As long as the batch size is big enough, we almost reach the maximum performance of the SSD.

Comparing NVCache with other systems

In these experiments, we compared the behavior of NVCache against other comparable systems offering the same guarantees. NVCache is configured with a 32GiB log in NVMM, so that the 20GiB workload does not saturate the log. Thanks to its simplicity, NVCache reaches good performance while providing a very high level of persistence guarantees. It provides better latencies than DM-WriteCache, Ext4-DAX and the Ext4 SSD in all of the tested workloads.

Fig. 3.12 gives a measurement of throughput, latency and written data over time, while running a random write workload. During this experiment, NVCache consistently keeps the lowest latency and the highest throughput. NOVA is the closest, as it also benefits from a log-based design suited for NVMM devices. Yet, as NOVA has to issue a system call on each write, NVCache ends up slightly more efficient in this ideal case. On average, NVCache maintains 493MiB/s vs. 403MiB/s for NOVA. DM-WriteCache also reaches a high throughput, but in an unstable way that makes

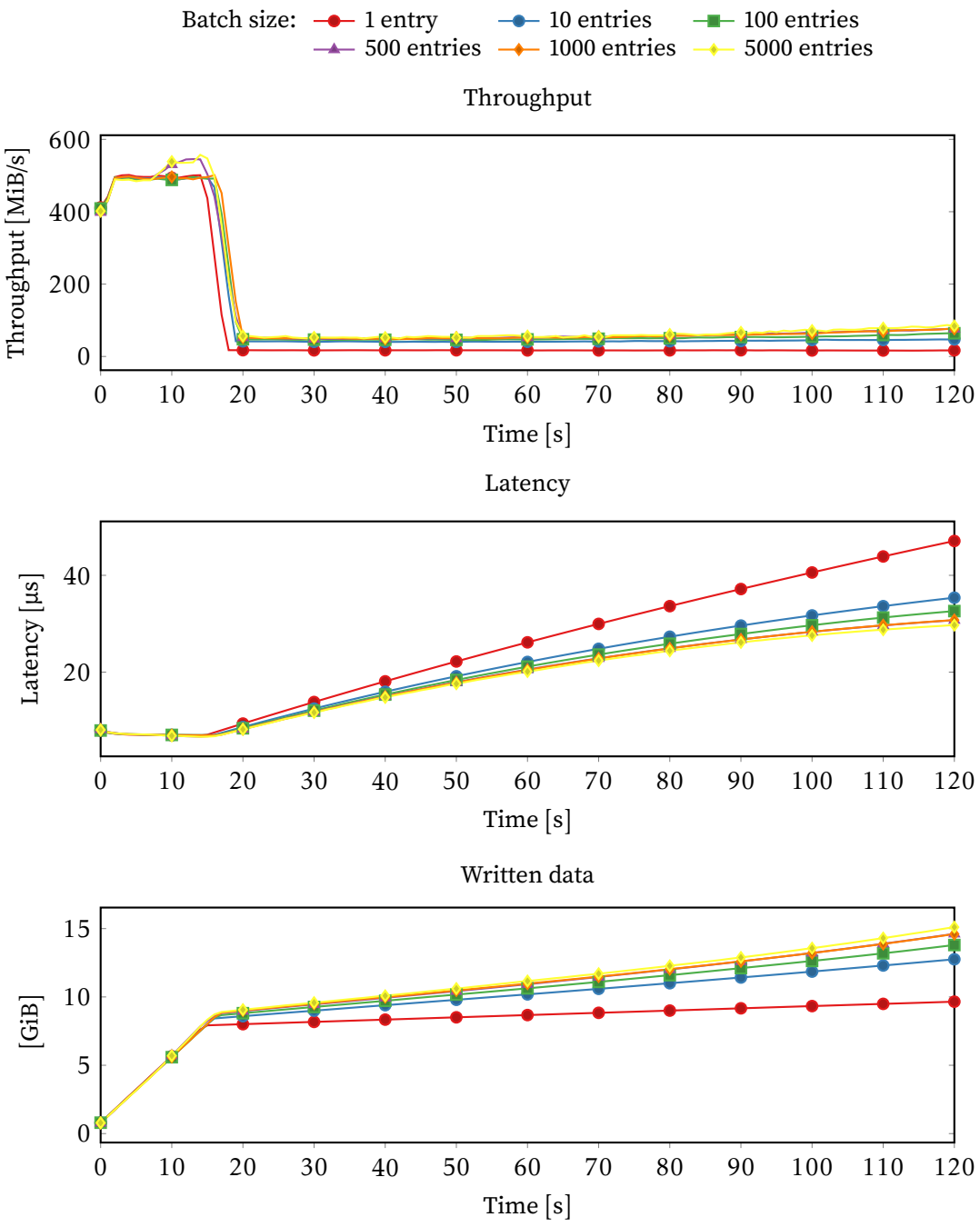

Figure 3.11.: Influence of batching and batch size parameter.

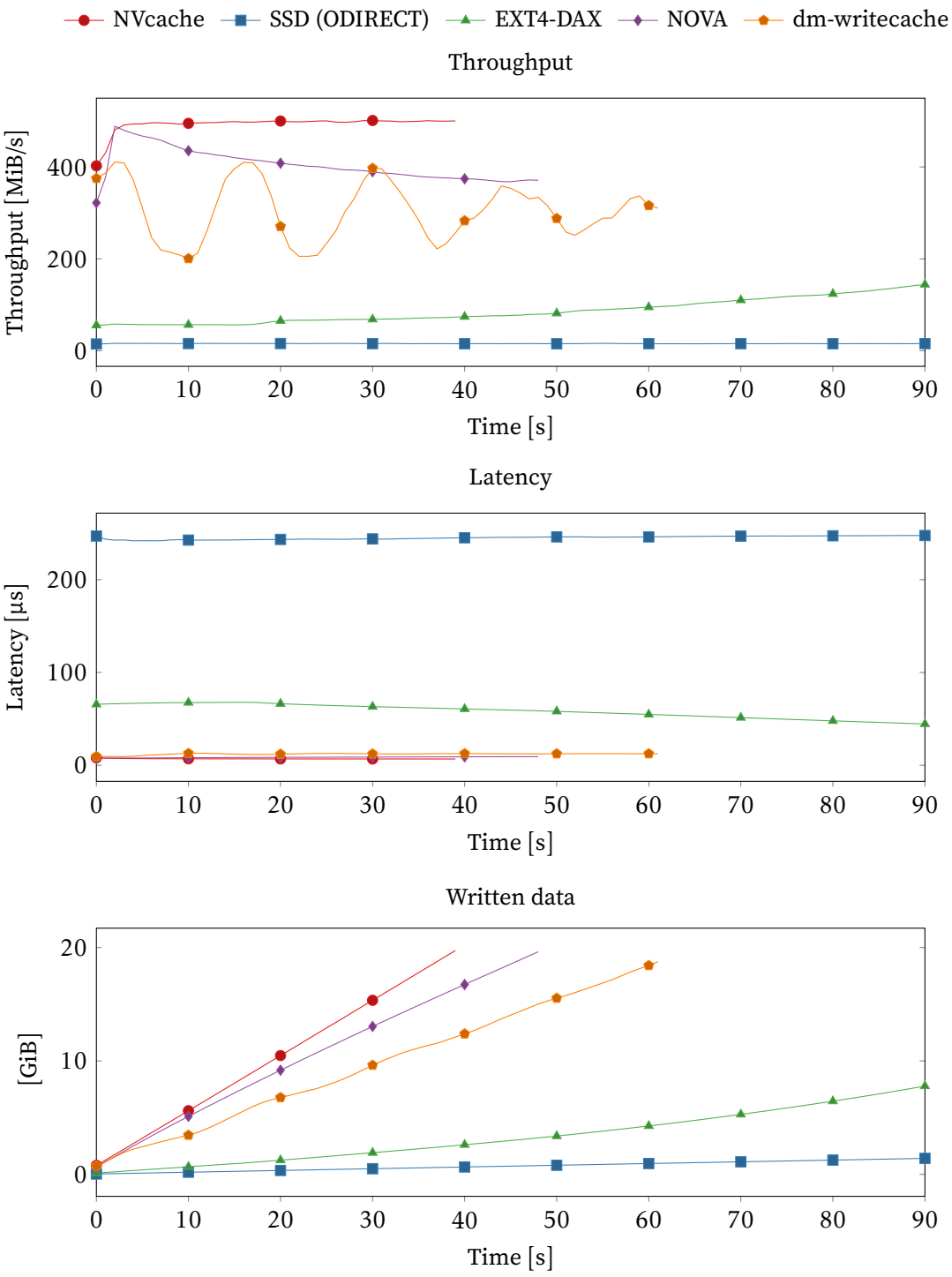

Figure 3.12.: Behavior of NVCACHE compared to other systems during a 20GiB FIO random write workload.

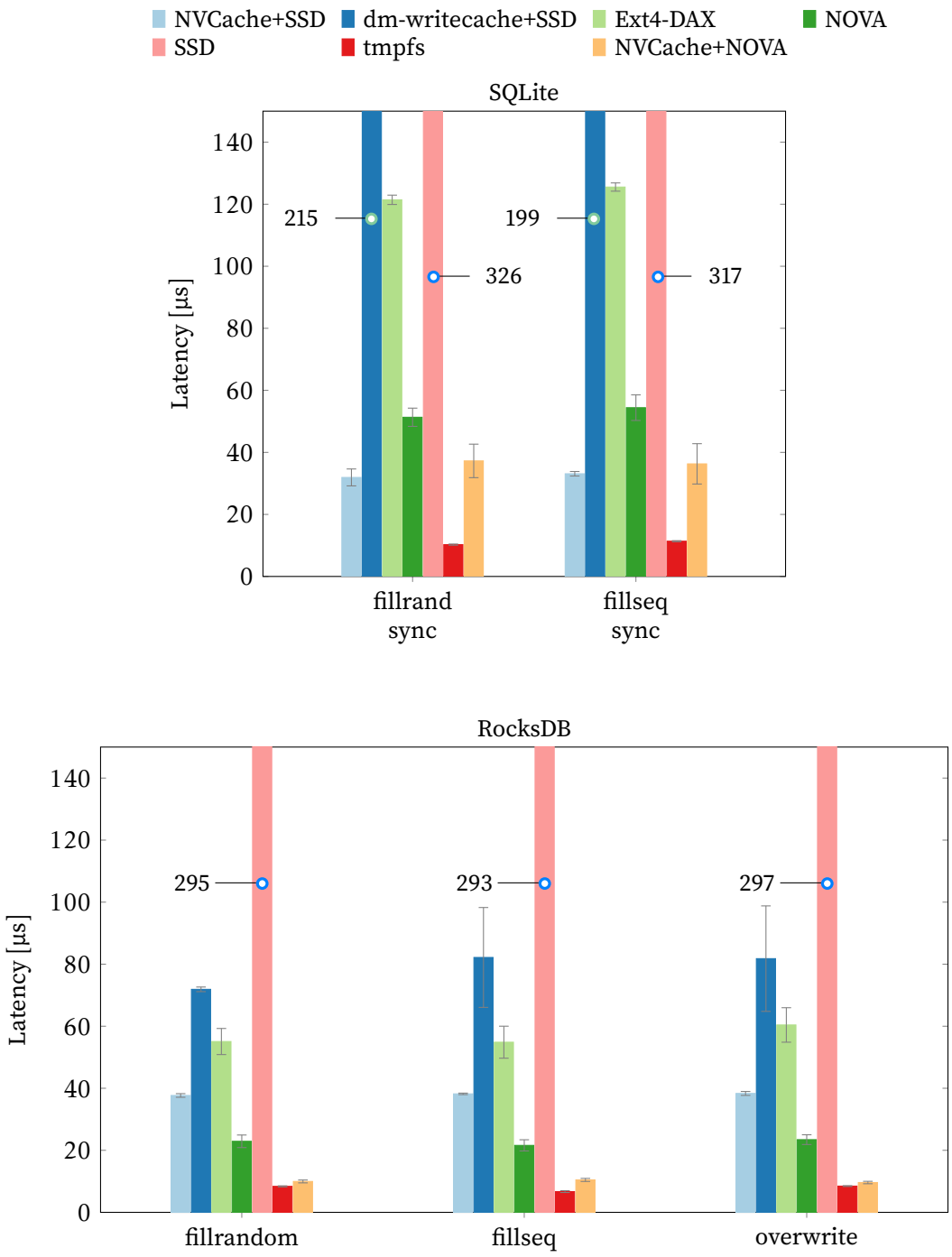

Figure 3.13.: Behavior of NVCache compared to other systems with SQLite and RocksDB write intensive workloads.

it oscillate through the benchmark. In comparison, Ext4-DAX does not reach very high performance measurements, and that is probably the consequence of its internal design that has not been optimized to very frequently call `fsync`. Then, the bare SSD gets the slowest results, being so slow that the experiment was stopped before reaching the 20GiB transferred. This is expected, as it represents the baseline of what durable linearizability would imply on a regular system without NVMM.

Fig. 3.13 reports the average latency of several write-based operations in RocksDB and SQLite. This experiment also presents NVCache as reaching higher throughput and lower latencies than the other systems during its caching phase. First, the Ext4 SSD setup takes around 300µs per operation. It is by far the slowest setup tested, as each write results in a slow `fsync` operation. In this case, DM-WriteCache is the second slowest configuration, which is different from the previous experiment. It is the closest comparison point with NVCache, as it also uses NVMM to cache an SSD. However, its performance with SQLite is particularly bad, being almost twice slower than Ext4-DAX. Then, Ext4-DAX and NOVA results show how NOVA has been optimized for best latency on NVMM, in comparison with the DAX-enabled Ext4 journaling file system. There is a clear advantage to use NOVA for this kind of workload, NOVA being almost two times faster than Ext4-DAX in every circumstance. To evaluate the impact of the secondary storage device in NVCache, we rin the same workload on NVCache +SSD. On RocksDB, the effect of using NVCache as an I/O booster on NOVA makes this configuration almost as fast as tmpfs, which is a baseline in a RAM file system with no persistence guarantee. On SQLite, NVCache gets the same level of performance on SSD as on NOVA. Yet, it gives NOVA a boost of performance.

3.4. Conclusion

NVCache is a log-based write cache in NVMM that boosts the I/O performance of a slower secondary storage. By using NVCache with an SSD, one can reach performances comparable with NVMM-based file systems, without being limitated by the available NVMM storage space. Also, NVCache has been designed to adapt any legacy application to get the benefits of persistent memory without modifications to the original code. One of the main barrier to the adoption of persistent memory being the impact in terms of code modification, NVCache is the kind of tools that could help legacy applications to get the benefits of this new tier of memory, without having to perform deep modifications in their code base. It also opens the

door to a new perspective of I/O, where persistence guarantees could be given by default, as it is not the massive performance penalty it used to be.

Chapter 4.

NVMM Cache Design: Logging vs. Paging

4.1. Motivation

Modern NVMM is closing the gap between DRAM and persistent storage, both in terms of performance and features. Having both byte addressability and persistence on the same device gives NVMM an unprecedented set of features, leading to the following question: How should we design an NVMM-based caching system to fully exploit its potential? After implementing NVCache, a follow-up question remained to be answered: Is NVCache a strictly better design than the more usual page-based approach of caches?

To answer this question, we compared two different implementations of NVMM-based caches, NVPages and NVCache, with two radically different design approaches. NVPages stores memory pages in NVMM, similar to the LPC. NVCache uses NVMM to store a log of pending write operations to be submitted to the LPC, while it ensures reads with a small DRAM cache. Our study shows and quantifies advantages and flaws for both designs.

The emergence of modern NVMM is a great opportunity to implement known designs and adapt them, or invent new ones. We tried these two approaches with caching mechanisms for a file system stored in secondary storage. Indeed, caching data for slower tier storage devices (SSD or HDD) is a great use case for NVMM. It provides high persistence guarantees, higher read and write bandwidth and lower latencies than most persistent block devices [63]. In this study, we target applications that require a high level of data consistency, which would highly solicit a regular disk with frequent calls to `fsync`. For such applications, we propose a persistent cache able to give fast persistence guarantees without having to wait for a slow secondary storage.

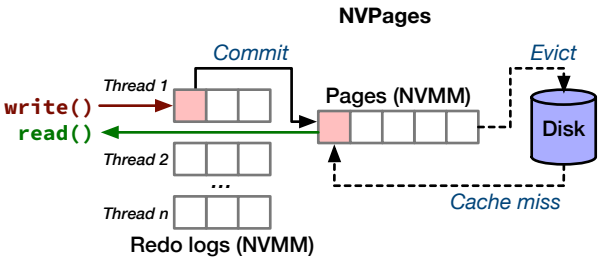

Figure 4.1.: Core design of NVPAGES

4.2. NVMM-based Caching

NVPAGES and NVCACHE are POSIX-like IO shared libraries. They provide standard IO functions, such as `open`, `pread`, `pwrite`, `close`, *etc*. When the shared library is loaded, NVMM is mapped, and data structures are initialized. A flag in NVMM is set to 1 when the program is loaded, and set to 0 when it is unloaded properly. This flag allows both caches to start a recovery procedure in case of a previous crash, flushing to disk every modification still pending in cache when the crash occurred. So far, they do not support multiple threads. However, they differ in their core implementation, depicted in Fig. 4.1 and Fig. 4.2.

NVPAGES. NVPAGES is designed as a regular page cache, with a few adaptations to make it compliant with NVMM and its persistence guarantees. 4 KiB pages are stored in NVMM. When a page is accessed, a radix tree in volatile memory looks for a volatile metadata structure that contains a pointer to the non-volatile page. In order to ensure consistency after a crash, calls to `pwrite` first write data in a redo log stored in persistent memory. Then, the redo log content is flushed to the actual non-volatile page cache. The page cache eviction is done with a least recently used (LRU) policy. NVPAGES can be used in `O_DIRECT` mode, bypassing the LPC to interact directly with the disk with aligned 4 KiB blocks. We do not report performance with this mode since we measured that bypassing the LPC reduces performance in read. As described in Fig.4.1, NVPAGES is designed to be adapted for multithreaded workloads.

NVCACHE. For these experiments, NVCACHE is used in a shared library mode, dynamically linked with our benchmarks. It embeds two main components: a NVMM log, and a small DRAM page cache. When the `pwrite` function is called, data is written to the NVMM log. A background thread continuously waits for log entries

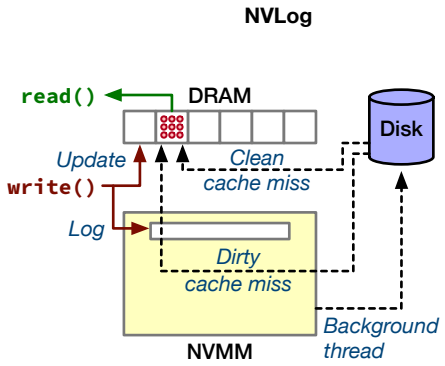

Figure 4.2.: Core design of NVCache

and writes them to disk as soon as possible. To ensure consistency in this configuration, every call to `pread` should get the page from disk and check if patches (log entries) have to be applied before returning the data. As this would make reads very slow, NVCache keeps a small DRAM page cache (2 GiB) with up-to-date data. It also keeps track of pages that would need to be patched before returning, so it only searches in the NVMM log when necessary. For reads, NVCache uses the LPC as an extension of NVCache's DRAM cache, from which it can fetch data instead of waiting for the disk. For writes, NVCache submits changes to the LPC in batches, before calling fsync to ensure the data is persisted on disk. This way, it benefits from LPC optimizations, such as merging consecutive writes on the same offset before writing the page on disk. Its design is complex because of the internal synchronization between the application and the background thread. Adapting it for multithread remains challenging.

Discussion. NVCache is designed to absorb bursts of writes in its log, but may not be suited for mixed or parallel I/Os. It only keeps a small amount of pages updated in DRAM. Increasing the amount of NVMM in NVCache does not change the probability of cache hit. Instead, NVPages is designed to maximize the probability of cache hit by keeping a lot of memory pages available in NVMM, as almost all of its allocated NVMM is dedicated to pages. We expected the latter approach to be more efficient for mixed IOs, reducing the amount of interactions with the disk.

4.3. Evaluation

Our benchmark machine is a Supermicro mono-socket machine with an Intel Xeon Gold 6326 CPU, 2 modules of 128 GiB of Intel Optane v200 DCPMM [55], and a 512 GB NVMe SSD, running Ubuntu 20.04 LTS.

We evaluated our 2 systems with FIO [4]. These tests are performing 20 GiB of random accesses through a 20 GiB-wide file. In Fig. 4.3 and Fig. 4.4, we submit pure reads (`randr`), 50% reads and 50% writes (`randrw`), 90% reads and 10% writes (`randrw90`), and pure writes (`randw`). Then, to show the efficiency of the caching policies, we measure the same tests with a Zipfian distribution[135] that ensures 95% of random offsets will be in 5% of the file. Each bar is the average completion time of 5 runs. For each plot, we compare NVPages and NVCache with a given amount of NVMM allocated. Our reference is the regular `psync` I/O engine of FIO which uses regular POSIX functions, measuring the performance of the LPC in DRAM. With this baseline, there is no guarantee of persistence, while NVPages and NVCache both guarantee persistence as soon as a `pwrite` call returns. Having similar persistence guarantees with `psync` is possible, by enabling a `fsync` call after each pwrite. However, completion times were so long that we did not include them in these plots (more than an hour for 20 GiB of pure writes).

We expected NVPages to be less efficient in pure write workloads, because the use of redo logs leads to write every data to NVMM twice. On the other hand, we also expected it to be more efficient than NVCache on mixed I/O workloads, because it can store much more data in its page cache, increasing the cache hit probability and reducing interactions with the SSD to the minimum.

However, these results show NVCache performs significantly better in almost every workload. The pure read performance of NVPages reveals a fundamental flaw that prevents it to perform better with mixed IOs. By design, cache misses have a cost in NVPages, because they imply to copy the missing page to NVMM. But the main flaw in this design is the bandwidth limitation of current NVMM compared to DRAM. NVPages can take pages from the LPC in DRAM, but will then require to read in NVMM to retrieve them for reads or writes. On the contrary, NVCache keeps fresh pages in DRAM, which allows us to get the full potential of DRAM read bandwidth, as we measured in Fig. 4.3 and Fig. 4.4 with `randr` and `randr-zipf` benchmarks.

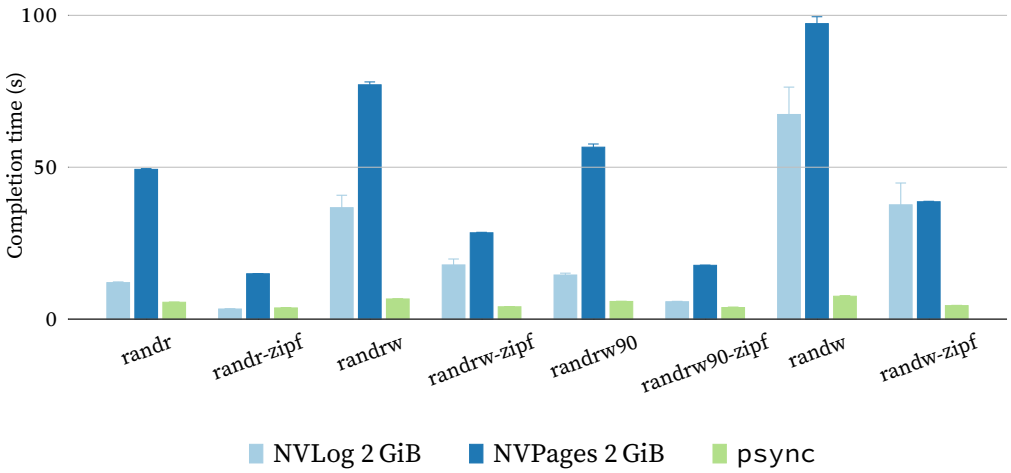

Figure 4.3.: FIO benchmarks with 2 GiB of NVMM cache

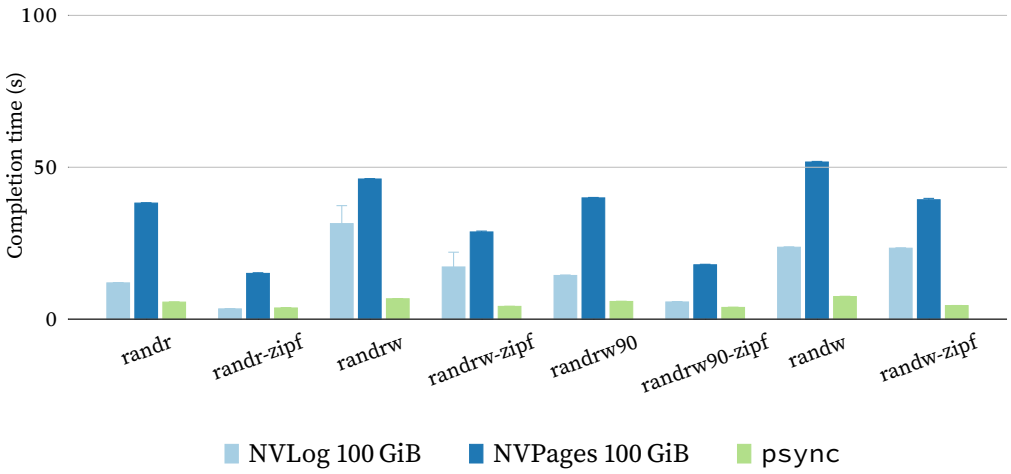

Figure 4.4.: FIO benchmarks with 100 GiB of NVMM cache

4.4. Conclusion

By running these experiments, we notice NVCache seems to have a clear edge over NVPages. It performed better on all workloads, even on those we could have expected NVPages to be more efficient. From a design point of view, NVPages has several advantages and may outperform NVCache on parallel I/O thanks to its independent redo logs (while NVCache must share the same log with all threads). However, the main bottleneck we found in NVPages relies on the difference of performance between DRAM and NVMM. It would also alter performance in a highly parallel mixed I/O scenario. This performance limitation is likely going to remain a problem with future generations of NVMM, as we could imagine DRAM throughputs an latencies to increase as well. In conclusion, the current design of NVPages does not seem to be efficient for NVMM caching, while NVCache circumvents the main problem of NVPages thanks to its log-based caching mechanism.

Chapter 5.

Conclusion on Persistent Memory

Persistent memory has brought a new set a features no piece of hardware had before. This peculiarity made it an ideal research topic. However, Intel announced in 2022 that they would not produce NVMM anymore, stopping at the same time the production and the developpment of all of the 3DXPoint line of products. That thesis was made during the very short time while Optane NVMM was commercialized. As a result, we witnessed the begining and the end of that piece of hardware.

5.1. Lessons Learned

Optane PMEM opened the way for several new applications of main memory. Among these applications, some fields seem to have particularly used this device and its capabilities.

- File systems (Ext4 [34], NOVA [138], SplitFS [66])
- Key-value stores (KVell [80], KVell+ [81], Chameleon [142])
- Data structures (BPTree [49], uTree [17])
- Caches (Johnny Cache [82], NVCache)
- Persistent Transactional Memory (Romulus [27], OneFile [109], TL4x [3])
- Fault tolerance programming (NVthreads [47], ResPCT [71], J-NVM [78], Plinius [140])

Through all of these examples, some general remarks can be done about the use of Intel's implementation of persistent memory.

5.1.1. Software integration

From a software perspective, integrating NVMM is not transparent. When coding for persistent memory, there are some caveats to keep in mind.

Programming with persistence

With great power comes great responsibility. Using PMEM with its persistence guarantees requires to code with extra care. Even with dedicated libraries such as Persistent Memory Development Kit (PMDK) [130], originally designed by Intel to simplify the use of PMEM, the programmer must have a clear idea of the level of guarantees they want to implement. Each modification in persistent memory can result in an inconsistent state, if the operations are not perfectly ordered and protected by the appropriate instructions. Thus, the main purpose of this new device can already be the source of major disfunctions, that can ruin the quest for crash consistency.

Fighting against optimizations

Among applications designed for persistent memory, many highlight the need to disable or avoid some hidden optimizations. This may seem counter-intuitive, but many low level optimizations are designed with the standard memory hierarchy in mind, where persistence is extremely slow while volatility is almost instantaneous.

The perfect example is the caching layer of the LPC, that end up reducing the overall preformance of PMEM by adding write amplification. The implementation of DAX in the Linux kernel is the solution found to bypass that precise optimization, but it is not seamless. Sometimes, those optimizations are so deeply linked to the core of the program that it may be complicated to disable them completely.

Another example would be the implementation of embedded DRAM caches in DBMS such as RocksDB or SQLite. These caches are written to avoid the LPC, and get full control over the caching mechanisms for the database. Yet, these have been designed to maximize performance with a slow disk or SSD as only persistent device. With persistent memory in mind, developers would probably take a different approach.

Compatibility with legacy software

The only way to use persistent memory transparently, is to give up on part of its features and use it either as RAM extension or as a regular block storage device. In order to get the full potential of this new device, using both persistence and byte-addressability, one inevitably has to modify the software. This change can go from a few lines to a deep refactor of the program.

5.1.2. Hardware integration

If software optimizations can hurt performance when using PMEM, it goes the same way for hardware optimizations. Though, the hardware is more complicated or even impossible to bypass.

Architectural limitations

The main problem of PMEM is the interaction with CPU caches. There is almost no way to bypass this hardware optimization which, once again, was designed for volatile memory. Technically, the `ntstore` instruction can bypass all of the caches when writing to persistent memory. Yet, this is not a recommended solution for many scenarios as in this case, accessing the written data will result in a slower cache miss.

Intel had to introduce the `clwb` instruction to overcome this situation, and allow the programmers to propagate a cache line to the non-volatile memory without eviction from the cache. However, that is not a trivial change. Intel even implemented the instruction before implementing its actual behavior, which lead to incomprehension from the PMEM developers [70].

Even with appropriate instructions, the overall feeling when programming for persistent memory is that it was not meant to exist. Using NVMM requires knowing the details of the architecture, which is not very developer-friendly.

eADR platforms

An interesting attempt to change the behavior of CPU caches was announced by Intel with the release of their Optane NVDIMM v200. eADR was supposed to address the complexity of NVMM programming by making the entire system more compliant with the need for persistence. By adding a battery alongside the CPU, eADR was supposed to give a more transparent persistence guarantee. In case of a power loss, the CPU caches would be powered long enough to flush every cache line located in PMEM to the persistent device.

The advantage of such a system would be to simplify programming, as there is no need to flush cache lines manually to get the persistence guarantee. However, even after the release of Intel's compliant CPUs and Optane NVDIMM v200, no platform supporting eADR has been commercialized.

5.2. On the Future of Persistent Memory

For almost a decade before the commercialization of Intel Optane, researchers imagined systems that would gratly benefit from persistent memory. Now, only three years after the commercialization, Intel announced they would not develop and sell this technology anymore. However, there is some hope that a new generation of devices could appear in the coming years.

5.2.1. The rise and fall of Intel Optane

In summer 2022, Intel announced the end of the 3DXPoint line of products. There were some hints of this event when Micron decided to end their partnership with Intel, the production of 3DXPoint in summer 2021, and selling their factory to Texas Instruments. Also, the "Silicon crisis" that hit the world around that time must have played a role [96, 132]. Yet, Intel ensured they would continue the production of 3DXPoint in another factory [118]. A year later, Intel officialized the end of investments in this technology.

There are some reasons that tend to explain why Intel Optane DCPMM were not a commercial success.

- The computer world is not ready. Using NVMM in current computers is not trivial, and requires efforts both in terms of architecture and software development. Tech companies did not want to invest all of their work force to change the design of their clusters and softwares. Moreover, persistent memory suffered from the image of a technology not mature enough to invest in.
- The idea of extending DRAM with cheaper and denser NVMM was great, but it is a niche market. It only addresses to people who want to run huge RAM-consuming software, while not having enough budget to buy a machine or a cluster with more DRAM. This mode also reduces performance in comparison with a DRAM-only setup. It could address to startups or research laboratories, but it does not match with the main market of Intel.
- Between the idea of 3DXPoint and its commercialization, NVM Express, or Non-Volatile Memory Host Controller Interface Specification (NVMe) SSDs became extremely performant, and almost matched Optane's performance on some workloads, while being based on a more mature technology (Flash memory) and for a cheaper price per gigabyte.

- Using Optane requires an Intel platform. There is no compatibility with other CPUs.
- Programming with Optane DCPMM requires an extensive knowledge of the platform you are coding for, which makes the software development more complicated, while reducing portability.

If these elements explain the lack of popularity of Intel Optane DCPMM (PMEM modules), other devices made out of 3DXPoint memory were quite popular, and in particular the Intel Optane SSD line of products. With quite low latencies for a SSD, Optane SSDs became popular among developers, video editors, and many other disk intensive use cases. It was a high-end SSD dedicated to professionals with a higher budget. Yet, this product disappeared with the other 3DXPoint equipment.

5.2.2. Compute Express Link

Both Micron and Intel agree that the next step for persistent memory will have to do with the new Compute Express Link (CXL) standard [21]. CXL is an interconnect protocol, initially designed to communicate between several CPUs and their peripherals. But CXL also extends interconnect communications over the Peripheral Component Interconnect Express (PCIe) bus. It aims to bring new features between extension cards (Graphics Processing Unit (GPU)s, Network Interface Card (NIC)s, Field-Programmable Gate Array (FPGA)s, other accelerators...) and the main memory. In particular, a part of the specification called `CXL.cache` is supposed to bring cache coherence between the different components over the PCIe bus and the main memory.

At first sight, CXL has nothing to do with NVMM. It is not specifically designed for persistent memory. However, from the already published specifications, all of the required concepts for PMEM are supposed to be integrated in future CXL devices. Thus, future persistent memory devices could probably be CXL extension cards.

There are some advantages for persistent memory extension cards. First, they would not rely on a single manufacturer. Any manufacturer that want to provide some kind of persistent memory only has to comply with the CXL specification. Then, extension cards are compatible with every kind of architecture and manufacturer, as long as the CPU is CXL-capable. That would let a lot more choices to customers regarding the CPU they want to use with their persistent memory, even in the probable event of `x86` architectures being replaces by other ones in the future. Last, the extension card format is suited for a niche market. There is already

a market of specific accelerators for very precise workloads. As NVMM is probably not going to be mainstream in computers anytime soon, keeping the format of an accelerator dedicated to specific workloads makes a lot of sense.

So far, only a few CXL compliant devices have been commercialized, but tech companies already prepare for that new generation of accelerators. For instance, Samsung already released a DRAM extension card featuring CXL [112].

5.3. Conclusion

The era of Optane DCPMM started in 2019 and ended in 2022. Over these three years, a lot of research has been done about that device, and it is likely that there will be future implementations of such persistent memory devices. Only, the format we used to have DCPMM is probably going to dissapear, as integrating different devices over the DRAM bus is quite complicated. The new CXL standard is opening persistent memory manufacturing to other companies. However, we do not know yet if another company will try to create persistent memory anytime soon, after the business failure Intel faced, trying to drastically change the way we use memory in modern systems.

Part II.

Remote Direct Memory Access (RDMA) & Programmable Networks

Chapter 6.

Introduction to RDMA

Remote Direct Memory Access (RDMA) defines how one machine can submit memory operations to a distant machine through a RDMA-capable network. This chapter is a summary of the notions of RDMA we had to use for our implementations. The complete explanation for these features is documented in the Infiniband specification.

6.1. RDMA-Capable Protocols

Properly speaking, there is no "RDMA protocol". There are only protocols that have an implementation of RDMA operations. The most popular RDMA-capable protocols are IWarp, Infiniband, and RoCE (Infiniband over Ethernet). However, as RoCE has a lot of advantages over IWarp [122], we chose to focus on Infiniband and RoCE.

6.1.1. Infiniband

Originally, the Infiniband protocol was released in the early 2000's. Its main objective was to provide higher throughputs and lower latencies than other protocols such as Ethernet, in particular in the context of local networking among servers of the same cluster. At this time, it was capable of 10Gbit/s, which gave it a clear advantage over Ethernet. Between 2014 and 2016, a majority of the supercomputers in the Top500 ranking [124] were using Infiniband [123]. Nowadays, Infiniband hardware can reach up to 400Gbit/s, while providing latencies around a microsecond.

One of the main reasons pure Infiniband networks are only found in massive clusters and supercomputers is that they require buying all of the network components specifically for this protocol. In the mean time, while almost matching the maximum link capacity of Infiniband hardware, Ethernet became the most popular amongst user-grade Local Area Network (LAN) equipments. This popularity lead to the creation of a hybrid protocol, RDMA over Converged Ethernet (RoCE).

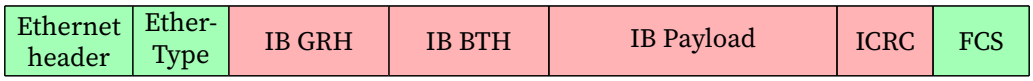

Figure 6.1.: A RoCE v1 frame. In red, the embedded Infiniband frame.

6.1.2. RoCE

Also called Infiniband over Ethernet (IBoE), RoCE is an implementation of Infiniband encapsulated in standard Ethernet frames. It allows using RDMA on more standard network hardware components than pure-Infiniband ones. For instance, one could use RDMA between two servers that are equipped with a RoCE-capable Ethernet Network Interface Card (NIC), while the links and switches between these two machines are simply standard Ethernet components.

From a software perspective, the RoCE protocol is used transparently as if it was regular Infiniband. All of the concepts defined in the Infiniband specification [52] are valid in RoCE, except from a few details in the connection procedure.

The RoCE protocol comes in two different versions.

RoCE v1

RoCE v1 only uses the Ethernet link layer. This is the simplest way to encapsulate Infiniband frames in an Ethernet one. However, these packets cannot be routed, which means they are limited to machines in the broadcast group. Thus, this protocol could not be used through the Internet.

Amongst these fields:

- Ethernet header: Contains source and destination MAC addresses
- EtherType: The last field of the Ethernet Header. In our case, this contains a special value to signify this is a RoCE frame.
- IB GRH: Infiniband Global Route Header. This is the equivalent of the Ethernet header for Infiniband frames.
- ICRC: Invariable Cyclic Redundancy Check. In Infiniband, this is the error-detecting code computed from the Infiniband frame.
- FCS: Frame Check Sequence. In Ethernet, this is the error-detecting mechanism.

Figure 6.2.: A RoCE v2 frame. In red, the embedded Infiniband frame.

RoCE v2

RoCE v2 still uses the Ethernet link layer, but it also uses a User Datagram Protocol (UDP) encapsulation. By adding this new level of encapsulation, RoCE v2 becomes routable. It can reach other subnetworks or even go through the Internet.

In RoCE v2, the IB GRH field has been replaced by:

- IP Header: This field can either be IPv4 or IPv6 and ensures the routability.
- UDP Header: The UDP field has a fixed destination port (4791) reserved for RoCE frames.

These two fields ensure regular routers can process and route RoCE packets. Even if we did not use that routability feature, we chose RoCE v2 as a support for our implementations.

6.2. RDMA-Capable Hardware

In order to use a RDMA-capable protocol, one has to be equipped with specific hardware.

6.2.1. Switches

As explained in section 6.1.1, using pure Infiniband requires to buy dedicated network equipment. The central element of any network of this kind would be a switch. In Infiniband, a switch has to read the Local Route Header (LRH) and Global Route Header (GRH) fields in order to decide on which port that packet has to be sent.

However, as mentioned earlier, the RoCE protocol allows to use regular Ethernet switches. This makes it easier to integrate in larger existing networks that would not already be Infiniband.

6.2.2. Network Interface Cards

To use RDMA features, a machine has to be equipped with a network card that provides two key functionnalities.

First, the network card must be able to access the main memory (Dynamic Random-Access Memory (DRAM)) of its host machine. In other words, the card has to use Direct Memory Access (DMA). DMA is a technology that makes a device able to read or write directly into the main memory, without the involvement of the Central Processing Unit (CPU).

Then, the NIC has to know the protocol in use, and act consequently. Either for Infiniband, RoCE v1 or RoCE v2, the card is responsible for the execution of the RDMA operations. It has to understand each packet, verify that the protocol is respected, and apply the operation requested. In a way, the RDMA NIC must be able to work alone and apply modifications in DRAM, while the CPU is doing something totally unrelated. To exploit Infiniband performances to their potential, Infiniband and RoCE network cards implement the 4 first Open Systems Inteconnection (OSI) layers in hardware (physical, link, network, and transport layers).

However, from a programmability point of view, the communication between a NIC and the CPU can be asynchronously re-established, by using event-driven programming. In practice, some RDMA operations can trigger an event to notify the CPU. For instance, a machine can send some raw data to another one by RDMA, and ask the receiving network interface to notify the CPU, meaning that data processing can start. That implies the network card can rise a hardware interrupt on the CPU when it is required.

RDMA-specific network interfaces are sometimes refered as Channel Adapter (CA) in the Infiniband literature.

6.3. RDMA Concepts

By design, RDMA connections are made of several specified data structures. Before a connection is opened, each machine has to initialize a certain amount of components in order to prepare its network card for future communications through the network. Moreover, the NIC must be aware of the actions it is allowed to perform in the host machine memory, which is part of this initialization process. Here are the main structures required to attempt a RDMA connection.

6.3.1. Memory region

The memory region (*mr*) is the most central element to configure on each machine. Basically, it points to a memory space that must have been "registered" beforehand. Registering a memory area means we are asking the NIC to prepare everything that will be required later to access and modify that space without any CPU intervention.

When a program registers a buffer of memory, it submits a contiguous space of virtual memory. However, when the NIC receives a write request for that space, it cannot ask the CPU Memory Management Unit (MMU) to translate virtual addresses to physical ones. That is why the network card itself has to keep a part of the translation table for the given memory region. Once a memory region has been registered, its physical addresses are stored in the NIC Memory Translation Table (MTT).

Several memory regions can be used simultaneously, even if they are not contiguous. Each one also has access rights, given by the developer when the region is registered. For instance, a memory region can accept distant writes, but refuse distant reads and local writes.

6.3.2. Work request

A work request is a structure that contains all of the required information to proceed with one RDMA operation. Amongst other fields, it contains the `opcode` (a number that represents the type of operation) and a pointer to the area where this operation must be executed.

6.3.3. Queue pairs

For each connection, a *send queue* and a *receive queue* must be initialized. These structures are responsible for storing work requests, either incoming (in the *receive queue*) or outgoing (in the *send queue*).

Each queue pair status follows a state machine, and each state can allow transmissions or not. For instance, the receiving queue must be in Ready to Receive (RTR) mode, while the sending queue must be in the Ready to Send (RTS) mode so that the work requests can actually be processed by the sending side.

6.3.4. Completion queue

The completion queue associated with a connection contains completion events that are generated when some work requests have been processed. For instance, on the receiver side, if a `RECV` request has been consumed, it means a `SEND` request has been received and processed. In this case, a completion event is generated, so that the server can know some data has been transferred.

However, when using the one-sided RDMA operations (`READ` and `WRITE`) the receiver does not have any notification. Some hybrid work requests can have the effect of one-sided operations but still raise a completion event, for example the `WRITE_WITH_IMM` operation.

6.4. RDMA Verbs

RDMA operations are accessible through an Application Programming Interface (API) called Verbs. The Verbs library contains low-level functions to set up RDMA connections, as well as all of the RDMA operations.

6.4.1. Two-sided verbs

Two-sided verbs require a particular setup on both ends. We call the two operations `SEND` and `RECEIVE`.

First, one side has to issue a `RECEIVE` work request, which contains among other fields, the address where the received data has to be written.. This request is stored in the `receive queue`.

Then, the other side has to post a `SEND` work request, which contains the actual data, but no location information. When this request reaches the destination, the `SEND` request is processed and the `RECEIVE` request provides the missing metadata. With all this information, the receiving networ card is able to execute the DMA operation in Random Access Memory (RAM). Then, a completion event is added to the completion queue, so that the program that issue the `RECEIVE` work request is notified its request has been fulfilled.

This mode of communication is very versatile, and can be used for many applications when the machine issuing the RDMA requests is not aware of how the other machine manages its RAM space.

6.4.2. One-sided verbs

One-sided verbs only require the intervention of one machine to access another machine's RAM. One machine can provide all of the information required to issue a DMA call to another machine's network card.

For instance, if a machine wants to write some data in another one, it can post a `WRITE` request that contains both the data and the remote pointer where this data must be written in the target's RAM. The network card will receive the work request, check whether the security concerns are fulfilled, *i.e.*, the address is in a registered memory region, the remote key is correct... and execute the corresponding DMA operation. It also manages by itself the sending of an Acknowledgment (ACK) when it was asked by the emitter, but the receiver is never notified an operation has been executed on its side.

However, in order to be allowed to write into another machine, one must know the remote key (or `R_key`) of the destination memory region. Exchanging this key must be done by an other way, either a `SEND/RECEIVE` operation or through a totally different network protocol.

From a performance point of view, one-sided operations are more efficient, as they do not require any CPU wake up on the receiving side. It is often considered good practice in RDMA to use one-sided operations as much as possible.

6.4.3. Special verbs

Some verbs can have intermediate behaviors, such as `WRITE_WITH_IMMEDIATE`. This verb almost behaves as a regular one-sided `WRITE` request, but offers to embed a small field (4 Bytes) to give the receiver an additionnal information.

6.5. Intel Tofino

The Intel Tofino platform is a programmable switch able to manage high-bandwidth traffic (up to 6.5Tbit/s in total) while applying modifications on network packets on the This switch architecture has been released in two different versions. We used the first generation of the product for our implementations, which features up to 65 Ethernet ports at 100Gbit/s.

6.5.1. Presentation

A programmable switch is a device supposed to be used for switching-like purposes, but can also be modified in order to create in-network features. By changing the behavior of a switch, one can offload some tasks in the network, by making some transformations on the network packets while they are being exchanged between machines.

With an Intel Tofino platform, the switch is built in two parts: the control plane and the data plane.

Data plane

The data plane is the core of programmable network devices. For the Intel Tofino, it is a pipeline made of successive stages that each have their own role. By going through this pipeline, packets can receive modifications between the moment they enter the switch and the moment they exit it to their destination.

The data plane is programmed using a dedicated language called $P4_{16}$. It describes the different actions to apply to a packet thanks to match-action tables. Some stages can also be set up for more advance features such as replicating packets, or calculating hashes. But programming a Tofino remains very challenging, as not a lot of information can be retrieved from the data plane for debugging purposes. Sometimes, the most efficient way to debug the data plane is to run the $P4_{16}$ code in a simulator and observe the state of each register while packets are flowing through the switch. Otherwise, an analysis of packet captures (PCAP format [46]) with Wireshark [137] is often a good way to find unexpected behaviors and deduce the origin of these bugs inside the $P4_{16}$ code.

Control plane

The control plane is responsible for higher level tasks, and in particular, it can set up the data plane. In practice, the control plane is a Linux machine embedded in the switch. It has a connection bus to communicate with the data plane, either for submitting a new $P4_{16}$ program or to modify some parameters of that program while it is running.

The control plane is reserved for slow interactions. In order to benefit from the switch performance, one has to avoid too many interactions between the control plane and the data plane.

6.5.2. Performance guarantees

As the Tofino works by pipeline, the same path has to be taken by each packet. Consequently, as long as packets are not going through the pipeline multiple times (one of the many features of the Tofino), they all spend the same amount of time to pass the switch. This time depends on the program that has been submitted to the data plane.

Chapter 7.

Byp4ss: Latency- and Throughput-Optimal Consensus Over RDMA

7.1. Introduction

Consensus is at the basis of any crash-tolerant distributed system. Ideally, reaching consensus should impose minimal overhead, but all existing protocols make trade-offs between minimizing latency and maximizing network capacity usage.

Modern consensus protocols have shown the importance of using RDMA to minimize latency [2, 127, 44, 67]. For instance, in Mu [2], a leader replicates data by doing a single RDMA `write` to the log of each replica. The network cards of the replicas acknowledge the `writes`, and once enough acknowledgements have been received, consensus is reached. Key to the low latency of Mu is the use of the RDMA `write` operation that allows the leader's data to be written and acknowledged without involving the replica's CPU, drastically reducing the latency of acknowledgements. While close to optimal in terms of latency, these RDMA-based protocols only use a fraction of the replicas' available bandwidth because the leader divides its own network bandwidth by the number of replicas. Some protocols have also been designed to maximize network capacity usage [43, 14], for instance by having replicas forward messages to each other in a ring. Such protocols are suboptimal in terms of latency because a message has to be forwarded multiple times in order to be accepted by a majority.

The trade-offs presented above arise because, in order to minimize latency, the coordination between replicas has to be minimized, which means that a single machine *decides* which values to replicate. This machine then has to *communicate* these values to the other replicas and becomes a bottleneck.

In order to both minimize latency and maximize throughput, we propose to decouple decision and communication. We accomplish this by handling the RDMA communication part of the consensus entirely in a data plane-programmable switch.

Having the communication handled by the switch is advantageous because it can broadcast and aggregate packets to/from all replicas at link speed.

Starting off on this idea, we implement **Byp4ss**, a shared memory interface exposed through an RDMA protocol. With Byp4ss, instead of individually accessing the memory of each replica, the leader accesses the shared memory interface exposed by the switch, which transparently broadcasts and aggregates RDMA commands to the replicas. To write in a system that tolerates f failures, Byp4ss broadcasts the request to all the replicas and waits for $f + 1$ acknowledgments before sending an acknowledgment to the leader. To read from the same system, Byp4ss first broadcasts the request to the replicas, and then only forwards the answer to the leader if the first $f + 1$ answers are identical. In the presence of a discrepancy, an error is returned instead.

Based on Byp4ss, we design a leader-based consensus protocol named **DISMU** (*"Disaggregated Mu"*). In short, DISMU implements the *decision* process of the Mu protocol. Unlike Mu, DISMU only sends a single message to Byp4ss instead of n messages to n replicas. As a result, DISMU is the first RDMA consensus protocol able to reach consensus in a single round-trip (minimal latency) while optimally using the network links of both the leader and the replicas (maximal throughput by having a single request/response per network link and per consensus).

Despite an apparent simplicity in its principle, implementing Byp4ss yields some intricate issues because the RDMA protocol only supports point-to-point `read` and `write` operations. A switch can easily multicast a carbon-copy of a packet but, to support RDMA, the various copies of the packet have to be modified to maintain the illusion that each server is talking to a single other server. To that end, Byp4ss keeps track of various stateful metadata, such as the authentication keys that authorize a particular server to read or write a particular region of memory of another server (these keys are randomly generated and different on each server).

The complexity of transparently replicating RDMA connection is further exacerbated by the fact that each RDMA requests is acknowledged. So, after multicasting a request, Byp4ss needs to compare and aggregate the acknowledgements from the replicas before forwarding a single acknowledgement to the leader. The acknowledgements are used by the RDMA network cards to inform each others of their relative congestion status. Byp4ss also aggregates congestion information to avoid a replica from becoming overloaded with pending RDMA requests.

We have implemented Byp4ss on a commercially-available Intel Tofino switch, and

DISMU, which relies on Byp4ss to decouple decision and communication. We compare DISMU against Mu. DISMU outperforms Mu by up to 10× in terms of throughput (on 4 replicas) while exhibiting 2.7× lower latency.

To summarize, we make the following contributions in this chapter:

- We propose to decouple decision and communications, and have the RDMA communications done entirely in the network to attain both optimal latency and throughput.
- We introduce the Byp4ss library, which exposes an RDMA shared memory interface through a programmable switch, which transparently forwards the operations to the replicas.
- We implement the DISMU consensus protocol, which relies on Byp4ss to achieve both optimal latency and throughput.
- Finally, we evaluate DISMU and Byp4ss by comparing DISMU with Mu.

The rest of the chapter is organized as follows. We first introduce background information on the protocols and programmable switches in Section 7.2. We then present the design of Byp4ss in Section 7.3 and describe its implementation in depth in Section 7.4. Evaluation results are shown and analyzed in Section 7.5. We finally discuss related work in Section 7.6 before concluding in Section 7.7.

7.2. Background

In this section, we briefly introduce the underlying technologies upon which Byp4ss builds, *i.e.*, RDMA and programmable switches.

7.2.1. Remote Direct Memory Access

RDMA reduces networking overheads by allowing a client to read to and write from a remote server without involving any other component than the network card on the remote side. In particular, the InfiniBand [52] network protocol provides a comprehensive implementation of RDMA, which is supported by network cards from several vendors [11, 56, 106]. Unless explicitly stated, mentions of RDMA refer to the InfiniBand implementation of RDMA. In this section, we explain the terminology relevant to our work, *i.e.*, the main metadata that Byp4ss modifies in order to replicate and aggregate RDMA packets. Table 7.1 summarizes these terms.

Table 7.1.: Metadata contained in an RDMA packet.

RDMA field	Usage
Operation code (OpCode)	Type of the packet: *ConnectRequest*, *ConnectReply*, RDMA *read* request, RDMA *write* request, RDMA *read* reply, *etc*.
Queue pair identifier	Which queue pair the packet is destined for (conceptually similar to TCP destination ports)
R_key	Randomly-generated shared key between the client and server, which authorizes a particular client to perform RDMA operations against a server's memory region
Packet sequence number (PSN)	Identifies the position of a packet within a sequence of packets
Credit count	How many requests the client may send to the server at this time

Queue pair. RDMA operations are transmitted using a point-to-point communication channel between a client and a server. Internally, the client and the server create a *queue pair*, a structure located in the RDMA card, that contains a send and a receive queue. The receiving end of the queue pair is uniquely identified by a number: the *queue pair identifier*. This identifier disambiguates which connection a packet is meant for.

Connection handshake. An RDMA server waits for incoming connections. A client establishes a connection by sending a `ConnectRequest` message to the server.The message contains several fields disclosing information about the client, one of which is the identifier of its queue pair. This identifier will be used by the server when replying to the client, for instance to acknowledge its requests. Once the server receives the `ConnectRequest`, it creates the other half of the queue pair, and sends a `ConnectReply` message to the client. The `ConnectReply` contains the identifier of the server's queue pair (*i.e.*, used by the client to send requests to the server). The connection becomes ready for RDMA transfer after a final `ReadyToUse` message gets transmitted from the client to the server.

Permissions. To ensure that RDMA operations against a server can only be performed by authorized clients, a shared authentication key (the `R_key`) is associated to every memory region exposed via RDMA. RDMA `read` or `write` commands need to include the right `R_key` for them to succeed.

Read/write requests. RDMA requests start with the client posting a `read` or `write` request in its send queue. It contains the authentication `R_key` of the memory region the request targets, as well as the virtual address where to read to or write from. In the case of a `write` request, the data to write is attached to the request.

The request is asynchronously dequeued by the network card of the client, which forges a corresponding `read` or `write` request network packet. The packet is addressed to the server, and more specifically to the right *queue pair* by including the *queue pair identifier* that was negotiated during the initial connection handshake. On the receiving side, the network card of the server checks the authenticity of the `R_key`, executes the given operation (provided the key is correct), and finally sends a response to the client all on its own, without ever involving the CPU.

Each RDMA packet contains a field called the *packet sequence number* (PSN) that uniquely identifies that packet within the sequence of packets that are communicated on a queue pair. Each new request increments the PSN. An important aspect of PSNs is that the reply associated to a request with PSN X will have the same PSN X.

Congestion. The *credit count* is a counter, maintained in hardware by every RDMA-capable network card, that is used as an estimate of how many extra requests can be buffered by the card. The idea is that most server-to-client RDMA responses announce the server's current *credit count* back to the client. Those packets can be standalone `ACK` packets, but RDMA `read` response packets for instance also contain the value. When a network card receives the credit count of another card, it uses that information to throttle its connections if need be, so that the other card does not get overloaded with queries.

7.2.2. Programmable switches

Programmable switches allow fine-grain control on packet flows, as well as executing custom operations on those packets, effectively leading to *in-network* processing. Conceptually, switches are split on two layers: the *control plane* above and the *data plane* below. The data plane consists of a specialized Application-Specific

Integrated Circuit (ASIC) which can process packets at line-rate across all ports of the switch. The control plane, on the other hand, executes on a standard processor and can be programmed using any programming language. Both layers can be programmed arbitrarily. We can summarize the relationship between those two layers by saying that the *control* plane *controls* how the *data* plane processes *data*. Nonetheless, as the control plane executes on a traditional processor, it is possible to use it for packet processing, with a lower barrier-of-entry in terms of development complexity, but with much lower performance than the data plane.

The predominant language used to program the *data plane* of a switch is P4 [10]. Protocol-independant Switch Architecture (PSA)—the $P4_{16}$ architecture that the Intel Tofino programmable switch ASIC is loosely based on—defines a *pipelined* architecture composed of an *ingress* and an *egress* (Figure 7.1). Both *gresses* are composed of three parts: (i) a programmable parser that extracts and organizes data from packet headers; (ii) a series of stages that can modify the packets and decide where to forward them, and (iii) a deparser that rearranges internal metadata to a stream of bytes. The stages are a composition of "match-action" steps: if (part of) the header of a packet matches a set of criteria, then an action is performed, such as choosing the next step of the processing, modifying the packet, deciding where to forward the packet, *etc*. These "match-actions" are stored in *tables* (the equivalent of a C switch/case, but implemented at the hardware level).

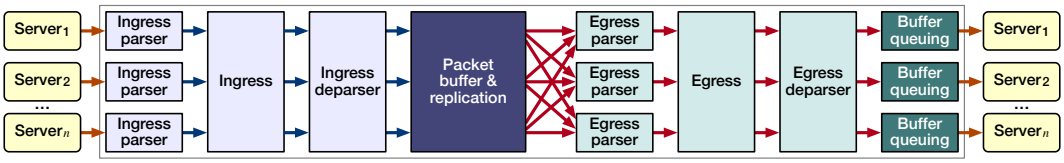

Figure 7.1.: Protocol-independant Switch Architecture pipeline. Blocks with a light background (black font) are fully-programmable whereas those with a dark background (white font) are configurable. Each server link has its own ingress and egress parser, while the gresses themselves have enough capacity to process simultaneous line-rate traffic across all links.

In between the ingress and egress sits a buffer and the replication engine. The latter enables flexible duplication of packets across multiple physical output ports. This design forces routing and replication decisions to be taken in the ingress. Conversely, operating on packet replicas must be done in the egress.

Intel Tofino is a fully-programmable P4 switch ASIC [58]. Tofino switches extend

the P4 programming language with stateful operations. In the ingress and egress pipelines, it is possible to store a value in a *register* when a packet matches a set of conditions, and read back the value when a later packet matches another set of conditions. Registers are very flexible as they embed a Arithmetic Logic Unit (ALU) that can perform computations when storing and when reading back its elements.

Computations in the data plane of a Tofino can be done at link speed (100 Gbit/s per link, or 6.5 Tbit/s in total, for the first version of the switch, and 400 Gbit/s, or 12.8 Tbit/s in total, for the second version). These numbers highlight that a programmable switch can process data at a much higher rate than a normal server machine.

7.3. From Mu to DisMu

In Mu [2], consensus is done in two phases. First, a leader *decides* the values to agree on, and then it *communicates* these values to the replicas. The key idea behind our work is to decouple *decision* and *communication*, and to optimize the communication using in-network processing. This section details the Mu protocol, shows how we adapted it to use Byp4ss, and gives an overview of Byp4ss.

7.3.1. The original Mu protocol

In Mu, each replica has a log, and a leader appends data to these logs. The replicas then consume the content of their logs asynchronously.

The constraint that ensures that logs stay consistent is that, at any given time, a single leader is allowed to write to the logs. Replicas rely on RDMA permissions to control which machines are allowed to write to their log.

More precisely, every machine participating in the protocol is given an ID. The leader is always the live machine with the lowest ID. To prove its liveness, each machine maintains a *heartbeat* value that it periodically increases. Machines frequently read each others' heartbeats, and the liveness of other machines is assessed by checking if their heartbeats increase over time.

Once a replica has chosen another server as a leader, it reconfigures its RDMA permissions to allow the leader to write to its log. A replica that elects itself as a leader waits until it gets the permission to write to a majority of the other machines before

writing any message. This ensures that the actions of a leader can always be seen by a majority of the participating machines.

The leader is responsible for sending the data to the replicas. It does so using RDMA `write` commands to append data to the replicas' logs. A data item is considered replicated once the network cards of $f + 1$ replicas have acknowledged the write.

7.3.2. DisMu overview

In the original Mu protocol, the leader sends the data individually to all the replicas. As a consequence, its network represents a bottleneck that impairs scalability. The key idea behind Byp4ss is to push the *communication* logic in the switch. DisMu uses thus the same leader-based *decision* process as Mu, but DisMu uses Byp4ss to write in the logs of the replicas.

Figure 7.2 presents the dataflow without Byp4ss and with Byp4ss decoupling. The communication pattern is illustrated in Figure 7.3. With DisMu, when the leader writes in the log of the replicas, instead of individually sending a write request to each replicas, the leader sends a single write request to Byp4ss. Byp4ss then replicates the request to the replicas on behalf of the leader, waits for $f + 1$ acknowledgments as for Mu, and sends a single acknowledgment to the leader.

As a result, while Mu divides the bandwidth of the leader by the number of replicas, this is not the case with Byp4ss. With Byp4ss, the available bandwidth of the leader is not impacted by the number of replicas.

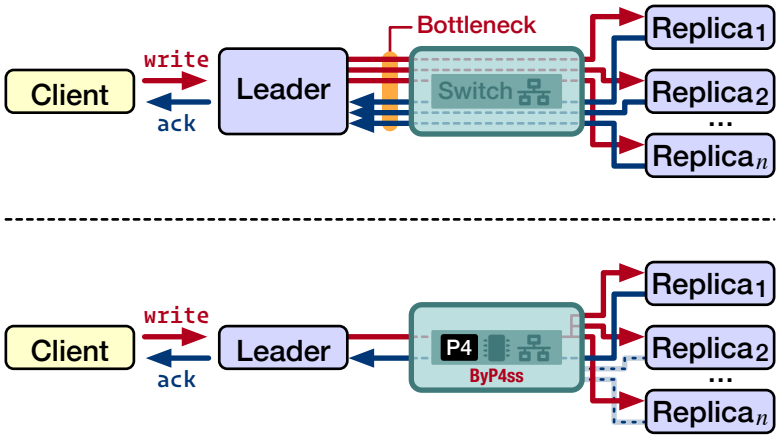

Figure 7.2.: Communication without and with Byp4ss.

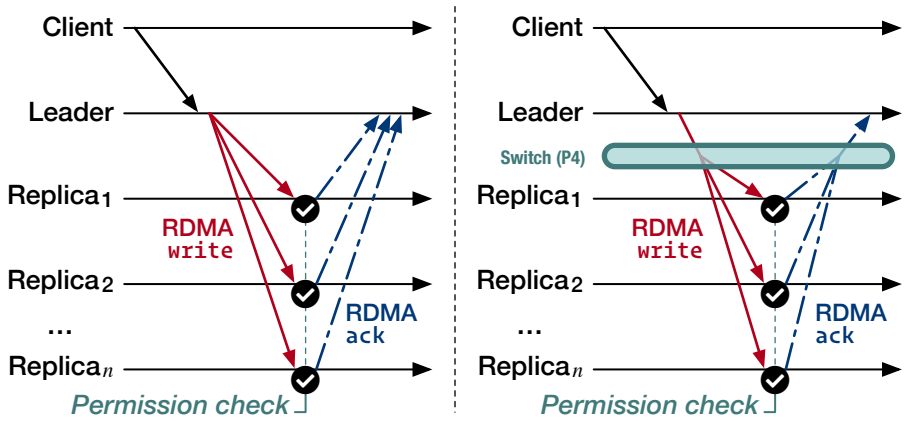

Figure 7.3.: Communication pattern used for consensus: without the programmable switch (left), messages are sent and response are processed by the leader, which represents a bottleneck; with Byp4ss, these tasks are preformed directly within the switch.

7.3.3. Byp4ss overview

In order to implement the communication logic of DISMU in a switch, we designed Byp4ss. Byp4ss is a general-purpose library that a consensus protocol can use to decouple decision and communication. It implements RDMA group communication, and handle both RDMA read and write requests.

In order to handle a request, Byp4ss ensures two key features: broadcast and gather.

Broadcast. When Byp4ss receives a request, it broadcasts the request to a set of replicas. For that, Byp4ss rewrites the request for each replica. At a high level, it rewrites the destination queue pair, the authentication key, the virtual address of the buffer accessed by the request, the packet sequence number and the IP address of the destination.

Gather. With RDMA, replicas have to answer to a leader, with an acknowledgment for a write request, and with the read data for a read request.

After sending a write request, in Byp4ss the switch waits for $f+1$ acknowledgments before transmitting an acknowledgment to the leader. As for a broadcast, Byp4ss has to rewrite several fields of the message: the destination queue pair, the packet

sequence number and the IP address of the destination. It also has to compute the fields related to congestion control. If one of the server sends a negative acknowledgment (NAK), the switch forwards it immediately to the leader. This allows the leader to become aware that one of the replicas is misbehaving and to handle the error – either by excluding the replica from future broadcasts, or by doing more in depth diagnosis.

Additionally, Byp4ss can also aggregate the answers of read requests. Since replicas may send different values, Byp4ss is also responsible for checking discrepancies in the data received from the replicas. In order to detect discrepancies, Byp4ss compares the hashes of the read data items. Based on the hash code, Byp4ss can identify if all the responses to a read request are identical.

The behavior of Byp4ss to handle read requests is not classical. Byp4ss compares the hash of the responses to the hash of the first packet it received. If the hashes of the first $f + 1$ answers are identical, the $(f + 1)^{th}$ answer is forwarded to the leader. If one of the responses does not match the hash code received in the first answer, Byp4ss sends an error message and the leader has to solve discrepancies by querying servers individually instead of relying on Byp4ss. Generally, in a consensus protocol, the leader waits for $f + 1$ identical responses among the n replicas and then fixes possible discrepancies when they arise. We made the choice of sending an error message to make the *decision* part of the protocol aware of discrepancies between replicas—otherwise the replica would have no way of knowing that some of its replicas have a different state from the majority because the aggregation is done in-network.

7.4. Implementation

Byp4ss implements group communication for RDMA. Since RDMA, as specified by the InfiniBand protocol, only supports point-to-point `read` and `write` commands, Byp4ss acts as a middle man that transparently multicasts and gathers RDMA packets between a source and multiple destinations. Byp4ss thus needs to duplicate, redirect, and transform individual packets so that every participant in the network has the illusion of communicating with a single machine.

Internally, Byp4ss maintains a set of *communication groups*. A group is composed of a single *source* server and a set of *destination* servers. When Byp4ss receives a request from the source, it broadcasts the request to all the destinations, aggregates

their response, and forward a single response to the source. In DISMU, the leader is the source, and the replicas are the destinations (see §7.3).

We implemented Byp4ss in $P4_{16}$ on an Intel Tofino-based programmable switch [58]. The section presents the main data structures used by Byp4ss, how a machine creates a communication group and how Byp4ss implements multicasting.

7.4.1. Communication groups and connections

Without in-network replication, the leader initiates the consensus protocol by establishing an individual RDMA connection with each replica. This operation is now handled by Byp4ss. With Byp4ss, the leader establishes a single RDMA connection *to the switch* and the switch broadcasts the connection request to the replicas.

Capturing incoming connections. Byp4ss configures the data plane of the switch to have all `ConnectRequests` intended for the switch, *i.e.*, that contain its IP address as destination, to be redirected to the control plane (see §7.2.1, a `ConnectRequest` is the first message of the handshake necessary to establish a RDMA connection). New connections are not a frequent operation, so handling them in the control plane simplifies the development effort and has no overhead in practice.

Broadcasting connections. The RDMA protocol allows `ConnectRequests` to be piggybacked with custom data. In Byp4ss we use the custom data to store the IPs of the replicas participating in the communication group. Byp4ss establishes a connection with all the destination IPs. Each replica receives a `ConnectRequest`, seemingly coming from the leader. If the replica agrees that the source IP of the `ConnectRequest` is the leader, then it replies with a `ConnectReply` as it would while communicating directly with the leader. Byp4ss aggregates the answers and sends a single `ConnectReply` to the leader.

In case of an error (*e.g.*, if a replica thinks that the source IP is not the new leader), the replica refuses to establish the connection and sends an error in the form of a `ConnectReject` packet. In that case, we follow the logic of the original Mu [2] protocol. If a connection has been successfully established with more than $f + 1$ replicas, then the leader receives a successful `ConnectReply` answer. Otherwise, the leader receives an error and knows that less than a majority of servers believe it is the leader.

Table 7.2.: Multicast metadata.

Name	Meaning
BCastQP	Virtual queue pair identifier for the source; all packets received on this QP will be broadcasted
MulticastGroup	Unique identifier used by the replication engine to know the destination ports of a multicast command
AggrQP	Virtual queue pair identifier for the replicas; all packets received on this QP will be aggregated and sent to the source
NumRecv[PSN]	An array used to store the number of replies for a given PSN
Hash[PSN]	The hash of the first reply for a given PSN
MinCredit	Minimum credit of the replicas of the communication group

The `ConnectReply` sent by the servers can also be piggybacked with custom data. We use the custom data to send the virtual address of the log of each replica, and the authentication key used to check read/write permissions (the R_key described in Table 7.1).

Getting ready for future RDMA commands. After establishing connections, all requests are handled by the *data plane* to guarantee that further processing is done at line speed. To allow the data plane to broadcast the packets to the right set of replicas, Byp4ss creates a multicast group in the replication engine of the switch. The multicast group has a unique identifier that will be used by the data plane when scattering packets.

Note that, on top of handling future RDMA commands for the established connections, the control plane still listens for new `ConnectRequest` packets to create new parallel connections, as Byp4ss supports multiple connections in parallel.

Metadata per group. For each communication group (Table 7.2), Byp4ss stores a `BCast` queue pair. The `BCast` QP is the queue pair identifier sent to the leader in the `ConnectReply`. All messages received on that QP are broadcast to the replicas. The replicas use the `Aggr` queue pair to send their replies (all replicas use the same QP number).

RDMA packets are uniquely identified by a packet sequence number (PSN) (as described in Table 7.1). For each answer received in the `Aggr` queue pair, the switch counts how many answers with the same PSN it has received. The $(f+1)^{\text{th}}$ answer

Table 7.3.: Connection structure. Byp4ss keeps one connection structure for each connection it made or received from a server (leader or replica). The connection structures can be found using an *endpoint identifier*.

Name	Meaning
Remote IP	IP address of the remote server
Remote QP	Queue pair identifier of the remote server
Remote Port	Port of the remote server
VA	Virtual address of the remote buffer
Size	Size of the remote buffer
R_key	Remote authentication key

is forwarded to the leader. In order to ensure that all replicas agree on the answer, Byp4ss also stores the hash of the first received answer for a given PSN. If subsequent answers have the same hash, the answer is forwarded. Otherwise, an error is returned.

Byp4ss also maintains congestion control-related metadata in order to avoid packets being dropped. The `MinCredit` represents the number of packets that the most saturated RDMA card of the group can handle.

Metadata per connection. For each established connection, Byp4ss stores metadata in the tables of the data plane of the switch. This metadata will be used to transparently forward future read and write requests. Table 7.3 presents the per-connection metadata.

For each connection to a server (leader and replicas), Byp4ss maintains a structure named the *connection structure* (Table 7.3). The structure fully identifies a connection: it contains the IP address of the server, its queue pair identifier and its port. When the server is a replica, the structure additionally contains the virtual address of the buffer, the size of the buffer and the authentication key. Byp4ss internally identifies a connection with an 8-bit integer that we refer to as *endpoint identifier*.

7.4.2. Scatter.

Multicasting read and write operations is critical for performance. Byp4ss implements them in the data plane.

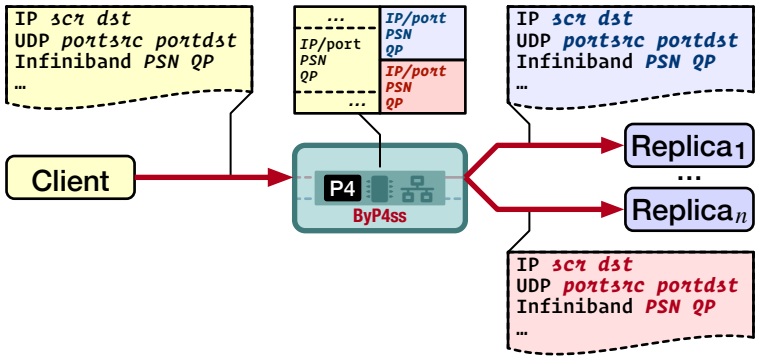

Figure 7.4.: Principle of packet duplication: Byp4ss transforms the different protocol addresses and identifiers in the headers to provide the illusion that the replicas receive the requests directly from the leader.

High-level implementation. Figure 7.4 presents the high level principle of packet duplication. The data plane mainly adapts the IP, UDP and RDMA connection fields to maintain the illusion that replicas are receiving packets as if they were coming directly from the leader.

Most importantly, after duplicating a packet, Byp4ss changes the PSN and queue pair identifier to match those expected by the replica. Byp4ss also changes the virtual address to/from which data is read and written. Indeed, when communicating directly with replicas, the leader reads and writes at the virtual address of their logs. With Byp4ss, the leader sends a single write request to the `BCast` queue pair of Byp4ss that is then replicated. Because the replicas may store their log at different virtual addresses, the leader just sends an *offset* to the `BCast` queue pair, and Byp4ss adds this offset to the virtual address location of each replica's log.

In the switch. Specifically, when a `read` or `write` request enters the switch *ingress* pipeline, Byp4ss starts by matching the destination IP to check if the packet is addressed to the switch. If not, it means that the packet is not meant to be duplicated, and it is transmitted directly to its destination. Otherwise, Byp4ss matches the destination Queue Pair (QP) number contained in the packet against the ones that were stored in a P4 *table* during the initialization phase. Matching against the contents of the P4 *table* returns a `MulticastGroup`.

The data plane uses the `MulticastGroup` as the key to instruct the hardware multicasting engine of the programmable switch to duplicate the packet (automatically,

according to the configuration made by the control plane when setting up the communication group). To ensure that the future answers to the request are properly aggregated, the dataplane also resets the `NumRecv` and `Hash` corresponding to the PSN of the packet it is multicasting (see Table 7.2 for the definition of `NumRecv` and `Hash`).

After the multicast engine, n packets are generated, one per *egress* pipeline of the replica they are destined to. At that stage, the n packets are carbon copies of the original packet. The only way to differentiate them is using an identifier that the multicast engine sets in the metadata associated with each packet. To speed up computations, we configured the multicast engine so that the identifier consists in the *endpoint identifier* of the destination replica.

We match the endpoint identifier against a P4 *table*, which gives us the connection structure of the replica (see Table 7.3). Updating addresses, ports and similar identifiers in Ethernet, Internet Protocol (IP), UDP and InfiniBand headers of the packet requires replacing the original values with the values saved in the connection structure during the initialization phase. Updating RDMA-related fields requires to compute their value. For instance, if the leader writes at offset o of the log of its replica, the data plane updates o to write at $VA + o$, VA being the virtual address of the replica's log.

Note that RDMA commands may spawn multiple packets when the amount of data to transfer exceeds the Maximum Transmission Unit (MTU), *i.e.*, when the payload cannot fit in one Ethernet packet. For instance, when writing a large amount of data on a connection configured with the Ethernet-standard MTU of 1500 B, a `write` request may get split into multiple packets, each with a payload of 1 KiB. In that case, Byp4ss multicasts each individual packet to all replicas.

7.4.3. Gather

At a high-level, Byp4ss waits for $f + 1$ answers from the replicas before sending an answer to the leader. However, aggregating the answers is not trivial because the switch needs to compare the answers, which makes the operation *stateful*.

In the switch. Byp4ss aggregates answers using the `NumRecv` and `Hash` *registers* (see Table 7.2). In our current implementation, we can aggregate 256 different PSNs per connection at a given time, which means that Byp4ss can handle up to 256 unacknowledged packets on the fly per connection—as a comparison, a given RDMA connection can only have up to 16 pending write requests and, while the number of

read requests is theoretically unbounded, we never observed more than 10 on-the fly in our network. Our current sizing thus works on current networks and is likely to remain viable on future faster networks.

Upon receiving the first response from one of the replicas, Byp4ss updates its *registers* accordingly: (i) it sets NumRecv[packet.PSN], the number of received replies, to 1; (ii) it stores the hash that the replica computed of the packet in Hash[PSN], and (iii) it tells the *deparser* of the ingress to drop the packet.

When receiving later packets, Byp4ss: (i) increments the number of replies, and (ii) compares the hash of the packet to the previously stored hash. In case of a hash value discrepancy, the packet is transformed to a Negative Acknowledgment (NAK). If less than $f + 1$ answers have been received, the packet is dropped. When $f + 1$ replies with the same hash have been received, the packet is finally forwarded to the leader.

Additionally, acknowledgment messages (which can be standalone or piggybacked on some types of packets, notably `read` response packets) contain important information about current InfiniBand-related resource usage; most importantly, the "credit count" needs to be correctly sent back to the leader to prevent overloading. As replicas may handle query at a different speed, Byp4ss takes the worst case into account. Byp4ss stores the most recent credit count announced by each replica in *registers*, and sends the minimum count across replicas to the leader (more details on how we achieve this can be found in section 7.4.4). The credit counts are stored at the communication group level, and not per PSN, to be able to send the latest credit count received per replica—otherwise, because the $(f + 1)^{th}$ ACK is forwarded, the credit count of the slowest replicas would always be ignored.

7.4.4. Under the hood

We now present some more intricate details about how Byp4ss works at a finer granularity. The *data plane* code of Byp4ss consists of 949 lines of $P4_{16}$ for Tofino Native Architecture (TNA) [57], while its *control plane* amounts to 1237 lines of Python. In the control plane, we use Scapy [9] as a framework to decode and craft RoCE initialization packets. The control plane uses Barefoot runtime (BfRt) APIs to interface with the data plane.

Performance concerns. A crucial aspect of our in-network processing is to be able to process packets at link speed. With our P4 code running, each ingress and each

egress parser can process 121 million packets per second. While large, this number actually becomes a bottleneck if a single parser ends up parsing queries coming from multiple replicas. For instance, in our first implementation, all the ACKs coming from the replicas were first processed in the replicas ingresses and then sent to the leader's *egress* where they were dropped. As a consequence, the leader's *egress parser* was a bottleneck and Byp4ss could only aggregate a total number of 121 millions packets per second. Changing the processing of ACKs to drop the packet directly in the *ingress* of the replicas, before they reach the *egress* of the leader, allows us to handle 121 million answers per second and *per replica* (so a total of 726 millions ACKs per second with 6 replicas for instance).

Doing in-network computations. Doing in-network computations is harder than one may think at first glance. As an example, we explain how we compute the minimum credit count of the replicas. We use a *register* per replica that stores the last credit count seen from that replica. On a standard ARM/x86 CPU, it would be easy to compute the minimum of these values, but doing the computation on a programmable switch is tricky.

For instance, on a Tofino switch, it is not possible to write the following code:

```
if(a < b) min = a else min = b
```

...because it is not possible, in hardware, to compare two variables (the ASIC can only compare a variable with a constant). Such limitations force us to work our way around the limitations of the ASIC using indirect ways to perform computations. In that case, the computation of the minimum ends up being:

```
if(identity_hash( (a - b) underflows? ))
    min = a else min = b
```

Checking if a subtraction underflows is a standard way of emulating comparisons, but the result of the underflow cannot be used in a conditional clause (because not cabling exists between the underflow information of the ALU and any conditionally programmable hardware). We thus forward the underflow information to an `identity_hash` (a module that simply returns the input value), which can finally be used in a conditional clause.

While this example may seem a bit trivial, one of the major source of complexity when performing in-network computing stems from the stringent hardware constrains of the Tofino ASIC. Every computation that a developer wants to program in P4 could be implemented in dozens of possible ways, but most of them cannot be deployed on actual hardware.

7.4.5. DisMu

DisMu uses the same decision process as Mu [2], and so has the same safety guarantees. Adapting the communication part of Mu to use Byp4ss is straightforward. Instead of sending n queries to the replicas and receiving for n acknowledgments, DisMu sends a single query and receives a single acknowledgment.

Faulty replica. When receiving an error or a NAK, DisMu switches back to the standard communication mechanism of Mu, dialoguing with individual replicas. If a replica is diagnosed as failed, DisMu reconfigures the communication group of Byp4ss by first disconnecting from the switch and then re-establishing a connection, excluding the faulty replica from the communication group.

Faulty leader. In case of a view change, one of the replica becomes the new leader, and initiates a new multicast group with a different list of participants. It is possible that, for a while, Byp4ss maintains the communication group of the old leader and of the new leader. However, any attempt to broadcast using the communication group of the old leader will result in an error because the replicas have revoked the authentication key of the old leader.

Faulty switch. Similarly to previous work, we assume that switches either work correctly or crash (we do not handle Byzantine behavior). If a single link of the switch fails, the replica attached to that link is considered faulty by the other replicas. In case of a more global switch failure, the leader will fail to connect to the switch and will try to talk directly to the replicas. Provided that the replicas can be reached via another network route, the leader will be able to connect to a majority of replicas and will run the protocol using manual replication. In that case, the leader periodically tries to re-establish a connection with the switch to go back to in-network replication.

7.5. Evaluation

In this section, we present our detailed evaluation of the latency and throughput of Byp4ss, comparing it against Mu.

7.5.1. Experimental setup

We tested Byp4ss and DISMU on 6 servers Supermicro SYS-520P-WTR, each with an Intel Xeon Gold 6326 CPU (16 cores). Four of these machines have 64 GiB of DDR4 RAM, and two have 128 GiB. Our client consist of a Dell PowerEdge R7515 machine with an AMD EPYC 7302P CPU and 32 GiB of RAM. These are all connected through an Edgecore Wedge 100BF-32X programmable switch that contains a 2-pipe first generation Intel Tofino ASIC. All machines are equipped with NVIDIA ConnectX-5 network cards interfaced through PCI Express 3.0 x16 links. Each card is directly connected to the programmable switch using 100 Gbit/s Ethernet.

7.5.2. Methodology

Each point of measure is an average value of 1 million operations. We noticed very small variations (less than 1%). We compare DISMU with our implementation of Mu on various setups, and in particular varying the number of replicas.

7.5.3. Bandwidth and Throughput

Maximum bandwidth. We first evaluate the maximum goodput achievable by DISMU, varying the size of the values sent by the leader (Figure 7.5). The goodput is the number of useful bytes sent per second, which excludes protocol overhead bits and retransmitted data packets. With 2 or 4 replicas, DISMU is as fast as Mu with a single replica. In details, DISMU is 2× faster than Mu when the consensus involves 2 replicas, and 4× faster when the consensus involves 4 replicas. DISMU is able to achieve consensus at link speed with value sizes above 500 B (11 GB/s of goodput shown in the figure, 12.5 GB/s total bandwidth). These results are expected: in DISMU the leader sends a single message to the switch and so can use its link at maximum capacity, while in Mu the leader shares its bandwidth between replicas.

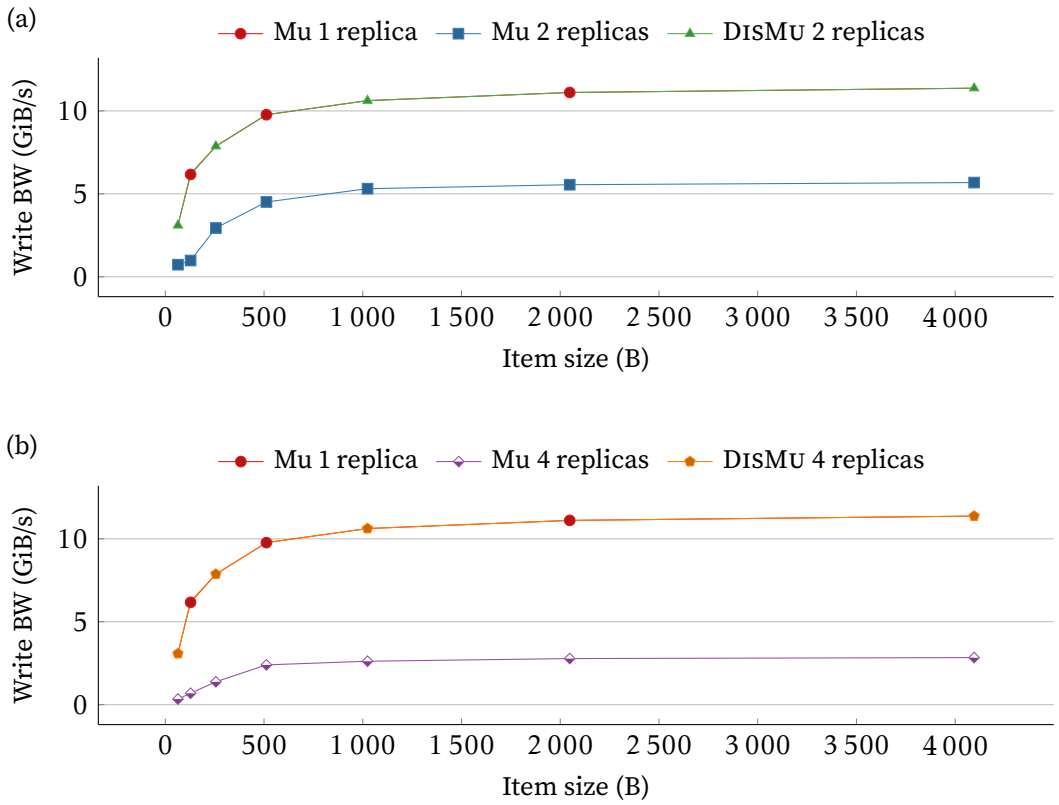

Figure 7.5.: Write goodput with different item sizes. DisMu maximizes the available bandwidth while Mu is limited by the leader's ability to duplicate packets. (a) With 2 replicas; (b) with 4 replicas.

Maximum number of consensus per second. Sending a single message is obviously advantageous in terms of network bandwidth, but it is also advantageous in terms of CPU usage. Figure 7.6 presents the throughput we measured doing consensus on small values (64 B), and varying the number of threads. In that configuration, the network is not a bottleneck; the consensus is limited by the speed at which the leader can generate RDMA packets and the bottleneck is the CPU of the leader.

With a low number of threads, DisMu exhibits a twofold speed increase over Mu with 2 replicas. The difference raises to up to 6× with 16 threads. With 4 replicas, the improvement goes from around 4× to over 10×. Running 16 threads, DisMu can achieve up to 52 million consensus per second.

When using a low number of threads, DISMU achieves higher throughput because it generates fewer packets (half with 2 replicas, quarter with 4 replicas, *etc.*) and because it does not need to aggregate the ACKs of the replicas. When using more than 12 threads, even though each thread has its own connection to the replicas and only accesses per-thread data structures, Mu suffers from contention issues. Profiling shows that the contention comes from the large number of atomic operations issued within the RDMA network card driver, which end up saturating the CPU.

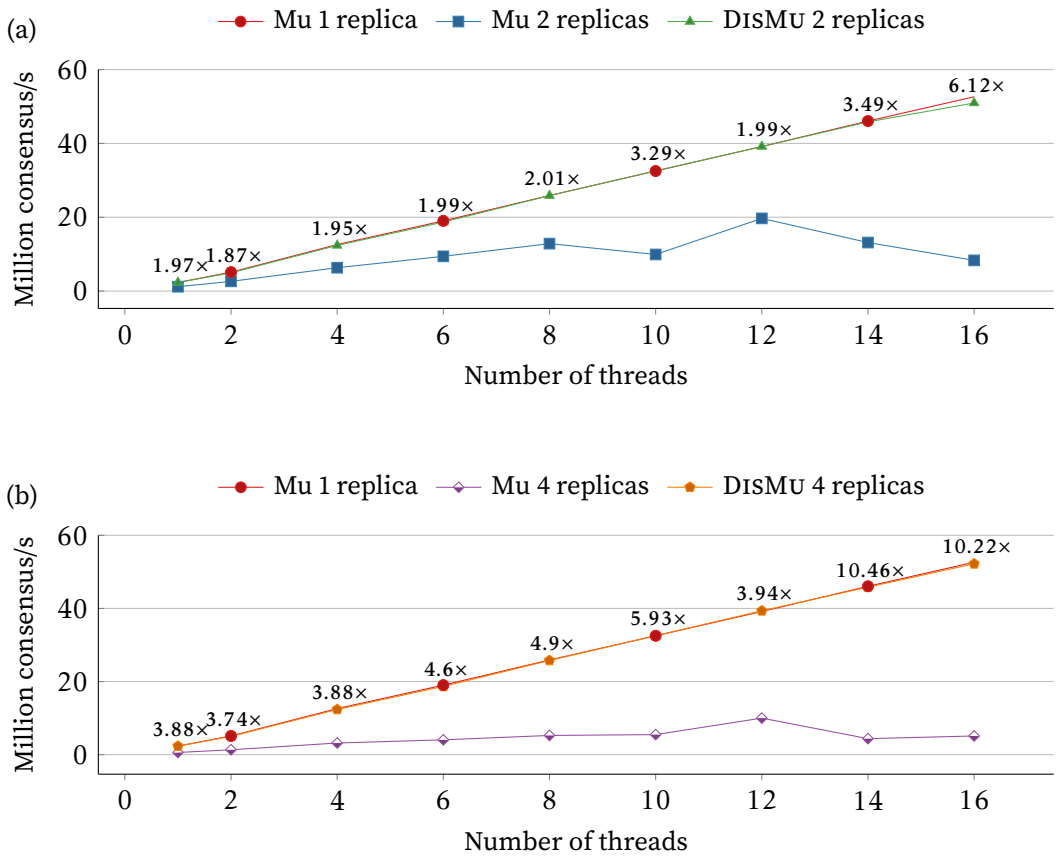

Figure 7.6.: Write throughput with 64 B requests, varying the number of threads, with 2 or 4 replicas. DISMU is up to 10× faster than Mu. Labels indicate the performance improvement of DISMU over Mu. (a) With 2 replicas; (b) with 4 replicas.

Summary. When doing consensus on large item sizes, DISMU outperforms Mu because the leader does not have to share its bandwidth between replicas. When doing consensus on small item sizes, DISMU outperforms Mu because it has a lower CPU overhead. Decoupling the decision and the communication is thus beneficial regardless of the size of the values to be replicated.

7.5.4. Latency.

In this section, we evaluate the impact of DISMU on latency. Figure 7.7 presents the relationship between the per-thread throughput and latency, for DISMU and Mu. Below 700 k consensus per second with 2 replicas and 400 k consensus per second with 4 replicas, DISMU's latency is 10% lower than that of Mu. The small difference is due to DISMU doing a bit less work on the critical path of queries (fewer RDMA requests, and no aggregation of ACKs), but neither DISMU nor Mu are CPU bound.

However, above 700 k consensus per second (400 k with 4 replicas), Mu becomes CPU-bound and its latency starts to grow. Mu cannot handle more than 1.2 million consensus per second per thread (600 k with 4 replicas) and queries start accumulating when generated at a higher rate. Thanks to its lower overhead, DISMU can handle up to 2.3 million consensus per second, regardless of the number of replicas.

The sustained throughput of both DISMU and Mu may seem high, but a similar observation can be made on the latency of short bursts of consensus operations. Both Mu and DISMU handle multiple consensus queries concurrently: when the leader receives a burst of queries, it sends a burst of RDMA `write` requests to the replicas' logs instead of sending one request and waiting for its ACK before sending the next one. As a consequence, Mu and DISMU can have multiple consensus "on the fly". We measure the latency of bursts of requests, varying the size of the burst. Results are shown in Figure 7.8. The latency difference between DISMU and Mu increases with the number of consensus on the fly. Mu starts to become CPU-limited when handling more than 10 queries simultaneously. DISMU's latency is half that of Mu when handling bursts of 100 requests. So doing less work on the critical path of queries also improves the performance of short bursty workloads, regardless of the total throughput.

Summary. Surprisingly, decoupling decision and communication is also beneficial in terms of latency because it reduces the amount of work to be done in the critical path of the leader.

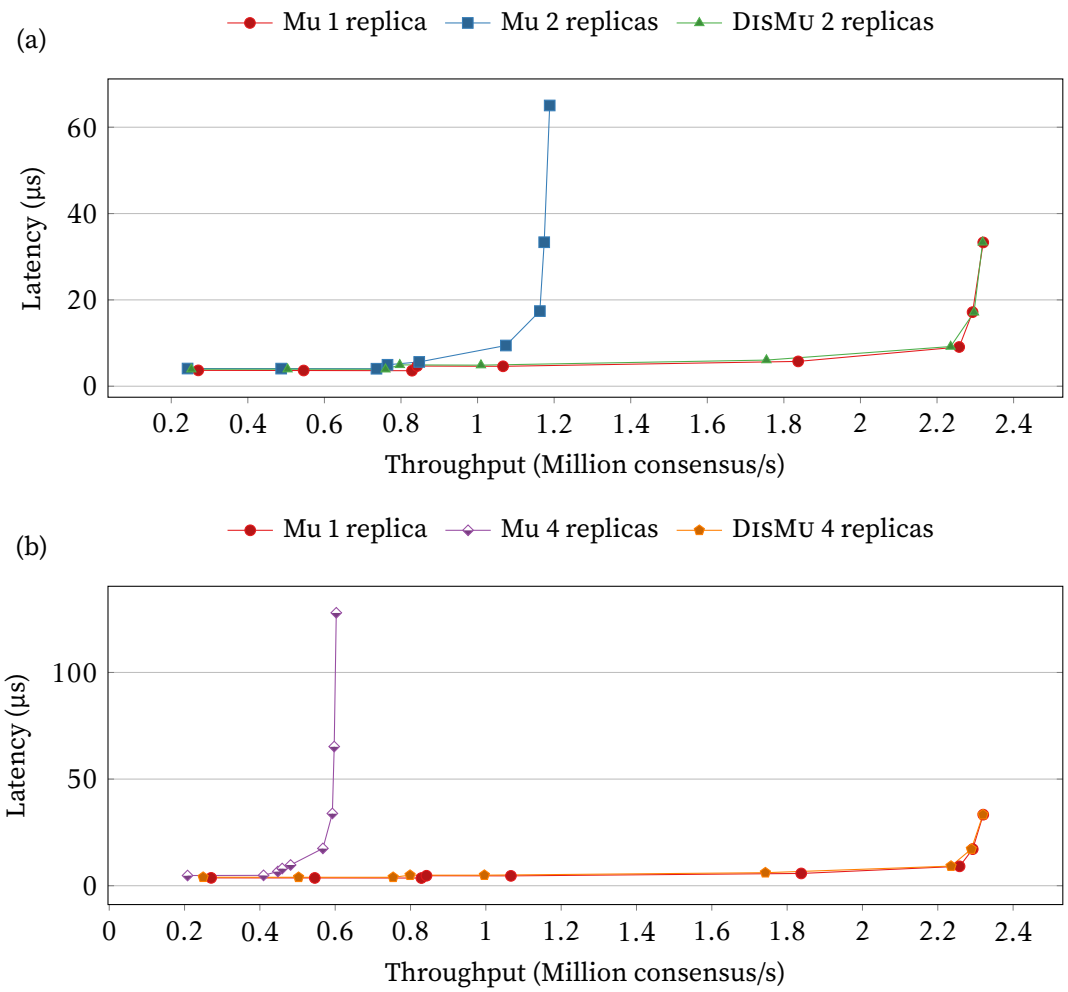

Figure 7.7.: Evolution of latency with 64 B requests vs. per-thread throughput. (a) With 2 replicas; (b) with 4 replicas.

7.5.5. Read workloads

Traditionally, consensus was performed in the same way when sending values to the replicas (*e.g.*, doing a PUT query in a key-value store) and when reading data from the replicas (*e.g.*, doing a GET query in a key-value store): in both cases the replicas would agree on a query to be executed. This approach made sense because the data stored in the replicas was not directly accessible by the leader. Recent

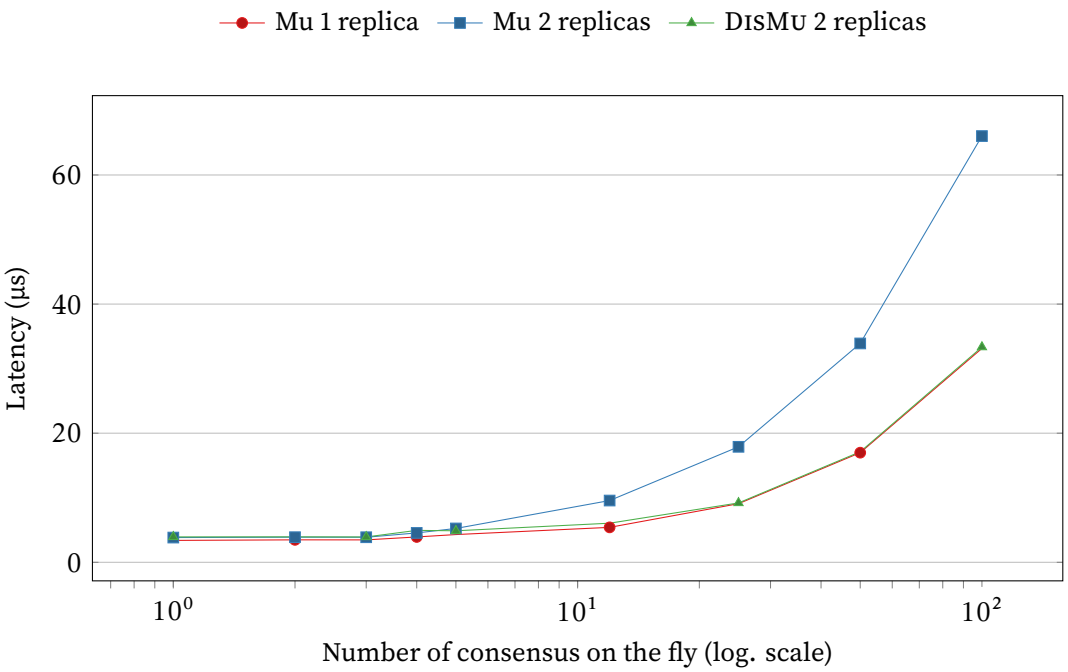

Figure 7.8.: Latency with 64 B requests, 1 thread

advances in key-value stores accessible over RDMA allow remote servers to directly read the stored values using RDMA `read` requests [12]. So, for completeness, we also test the performance of agreement on reads. In that experiment, the leader sends a `read` request to the replicas and considers the value valid if the same value is read from all replicas.

In DisMu, the leader sends a single RDMA `read` request to Byp4ss, which aggregates the answers. In Mu, the leader sends individual requests to all replicas and compares their answers. Table 7.4 presents the throughput of DisMu and Mu, reading 64 B values, varying the number of threads. Unsurprisingly, our observations match that of the traditional consensus: DisMu is at least 2× faster than Mu with 2 replicas and 4× faster than Mu with 4 replicas. Profiling shows that the performance difference is mainly explained by two factors: (i) DisMu crafts fewer RDMA requests, and (ii) DisMu does not have to compare the content of the packets to verify that the same data has been read from the replicas (this operation is done in the switch). So DisMu is less limited by the CPU than Mu.

Table 7.4.: Read throughput with 64 B requests, varying the number of threads, with 2 or 4 replicas. DISMU is up to 9× faster than Mu.

	Mu			DISMU
# threads	1 replica	2 replicas	4 replicas	1, 2 or 4 replicas
1	2.3 M reads/s	1.1 M reads/s	0.59 M reads/s	2.3 M reads/s
16	52.1 M reads/s	12.5 M reads/s	5.7 M reads/s	52.2 M reads/s

Summary. Decoupling decision and communication is also beneficial when reading data from the replicas.

7.6. Related Work.

The availability of programmable switches on the market and their increasing availability continues to attract the attention of academia and industry, opening the pathway to novel and more efficient network protocol designs [113].

Accelerating via programmable switches. Network hardware acceleration has been used to accelerate Paxos, and other consensus protocols [60]. NetPaxos [30] motivated the use of switches to accelerate the roles of the original Paxos protocol. P4xos [28] (and its ancestor [29]) implemented these ideas, using multiple Tofino switches, each playing a specific role (one switch is a proposer, another the acceptor, *etc.*). P4xos implements the decision of these roles inside the switch, and packets are forwarded between switches as they would be forwarded between standard servers playing the Paxos roles. P4xos requires a specific network topology with multiple programmable switches, which makes it hard to deploy outside specially-designed clusters. The large amount of messages exchanged between switches (and actual servers sitting behind the switches) means that P4xos neither minimizes latency, nor maximizes bandwidth usage. For instance, the latency of P4xos exceeds 100 µs at 100 k consensus/s (compared to 33 µs when executing 2 million consensus/s with a single thread in DISMU).

FastCast [8] proposes to use a switch to perform efficient multicast of IPv4 messages. NOPaxos [84] simulated the use of programmable switches to further tag the multicasted messages with a sequence number, in order to replace consensus with network ordering. HovercRaft [75] uses programmable switches to speed up

the multicasting of R2P2 messages. Byp4ss builds on these ideas and, crucially, enhances the multicasting with the in-network aggregation of acknowledgments to allow the transparent replication of RDMA connections. Allowing the transparent replication of RDMA connections is needed to have the whole *communication* part of the consensus run in the network, and key to the high throughput and low latency of DISMU.

Belocchi *et al.* [6] have proposed to use SmartNICs to accelerate Paxos. Hyperloop [72] use SmartNICs to offload the handling persisting transactions on multiple replicas to the network card. These approaches are useful to reduce CPU overheads on the leader, but do not eliminate the bandwidth bottleneck. Harmonia [143] took a different take on in-network processing and uses programmable switches to detect conflict read-write conflicts between leader(s) and replica(s). The goal is to ensure that a leader always reads values from an up-to-date replica. The approach used by Harmonia [143] is complementary to that of Byp4ss and could be used to speed up read-intensive workloads.

Programmable switches have also been used to accelerate various workloads: from machine learning [41, 114, 88, 65, 89], map reduce tasks [15], distributed datastores [73], software-defined network functions [141], future Internet architecture (SCION [31]), *etc.* P4SC [16] is a framework to implement service function chains into P4-enabled devices and execute common network functions (*i.e.*, NAT, firewalling, L2/L3 forwarding, load-balancing) at line speed in the data plane. Wang *et al.* [129] propose an IoT framework to aggregate small network packets into large ones and to disaggregate them later. Rather than such routing-level services, Byp4ss is optimized to support efficient replication protocols.

Reducing bandwidth requirements of the leader. Multiple protocols have been proposed to reduce the bandwidth requirement of leaders. Most of these papers rely on replicas forwarding packets between each other, usually following a ring topology [43, 14]. These protocols offer suboptimal latency, and an active replication action from each replica (which adds CPU overhead on the replicas, or requires the use of SmartNICs).

Accelerating workloads using RDMA. Using RDMA is key to the low latency and high throughput of DISMU. DISMU follows the logic of Mu [2], which uses RDMA permissions to ensure that, at a given time, a single leader can write to the replica's logs. Other RDMA-accelerated consensus protocols have been proposed. Similarly to Mu, APUS [127] pushes messages in replicas' logs using RDMA `write` commands. Replicas acknowledge having received the messages by sending other

RDMA `write` commands. Velos [44] leverages RDMA compare-and-swap operations to allow consensus to be achieved even when multiple competing leaders try to write to the logs of the replicas. Sift [67] separates replicas in CPU nodes that store non-persistent state and storage nodes, modified using RDMA. In this work, we focused on improving Mu [2] because it outperforms the other protocols, but Byp4ss can be used to improve the performance of all these protocols by transparently multicasting and aggregating RDMA operations and their results in the network.

Changing and extending RDMA. RDMA multicast has been implemented in software libraries [5, 77]. These libraries extend the standard RDMA libraries to make it possible to send data to multiple replicas. In RDMC [5], multicasting is done in multiple hops and using multiple point-to-point requests: the leader manually replicates data on a set of replicas that then re-send the data to other replicas, allowing dissemination over a large group of replicas. In RamCast [77], the leader coordinates replicas to give the illusion of an atomic broadcast. None of these software implementation offers latency-optimal or throughput-optimal replication because they are limited by the bandwidth and CPU of the servers doing manual replication of the data.

Researchers also explored the possibility of adding support for multicast directly in RDMA, most notably by allowing replicas to accept and process the exact same RDMA packet [79], which would allow a passive optical cross-connect fabric to do the duplication of packets. Such extensions would simplify the design of the scattering performed by Byp4ss, but not of the aggregation of ACKs.

7.7. Conclusion

We have proposed the first RDMA-based consensus protocol able to achieve consensus in a single round trip at link speed. We have shown that decoupling decision and communication brings significant throughput increase, while reducing the latency of reaching a consensus.

The idea of Byp4ss is deceptively simple, but yields important performance gains in practice. Our results demonstrate that Byp4ss allows consensus to scale across multiple replicas, regardless of the size of the exchanged values. We demonstrate $2 - 10\times$ better throughput and up to $2.7\times$ better latency than the state-of-the-art consensus protocols.

Chapter 8.

Conclusion on RDMA and Programmable Networks

Developping Byp4ss was an opportunity to experience both programmable networks and RDMA programming, going deep into the specification of the Infiniband protocol. If these two topics may seem unrelated outside of the Byp4ss project, they are in reality quite intricated, as they could become a more common combination in the future.

8.1. Programmable Networks

Programmable networks have become a very wide topic in recent years. By design, they are an ideal platform for research, both for networking and indirectly related research topics. Similar to Persistent Memory (PMEM), the first implementations of ideas using programmable networking devices were using simulation or virtualization. As a result, a community of developers started to form around the Data Plane Development Kit (DPDK) project [83] and $P4_{14}$ [23], a language designed to control packet forwarding planes. A few years later, $P4_{16}$ [24] was released with major changes in its design, and some hardware supporting $P4_{16}$ were commercialized. Among them, some smart-NICs and the Barefoot Tofino programmable switch [58]. Since, a lot of research papers have been using P4 programmable devices to offload some tasks to the network [126, 110, 68, 69].

In 2019, Intel bought the startup that designed the Tofino switch, Barefoot Networks. In 2022 a Tofino 2 version was commercialized. It is basically an upgrade of all of the capabilities of the first Tofino switch, with increased throughput and more programmable stages. At this time, a Tofino 3 was announced, but this platform will likely never be commercialized. Indeed, in January 2023, Intel decided to halt the development and production of its networking chips.

Other platforms still continue to develop to provide programmability to the network. In particular, NVidia is becoming a major actor of the field, since they bought

the networking company Mellanox in 2020. They mostly provide programmable NICs, the Bluefield line of products [102, 94]. Also, Broadcom's Tomahawk switches are a direct concurrent of Intel's Tofino, except they come with less flexibility in their programming. They also use a language called NPL [100].

The field does not seem to face a major issue with the end of the Intel Tofino line of products. More programmable devices are likely to be commercialized in the coming years, but the trend does not seem to be the quest of ideal programmability. The high level of programmability of the Tofino is, like persistent memory, a very interesting research platform. It is ideal to evaluate the impact of logic changes in a network, and does make some complex protocols implementable. However, this level or programmability (being able to modify the data plane on the fly) is probably a niche, and might not be the main concern of industrial customers. In comparison, reaching higher throughputs and lower latencies, while offloading some repetitive tasks to the network, are probably decisive in the choice of a programmable network solution.

Programmable switches are probably not going to be the only programmable components in future networks, as they would likely be used alongside smart-NICs. Given the current communication of companies like NVidia, the current vision of in-network computation is very application-specific. For instance, some products provide some acceleration for Artificial Intelligence (AI) training [103, 101, 104]. In this case, the network may be performing some of the computation, but there is no particular user programmability. Yet, for research and development purposes, the Intel Tofino platform is an ideal component, as it allows to manipulate the lowest stages of the OSI model.

8.2. RDMA in the Computing Landscape

RDMA has already been in the computing field for two decades. Its main implementations, RoCE v2 and iWarp, have gained popularity since they do not require too specific equipments anymore. Using RDMA has never been more accessible and easy to deploy. Yet, it is used only in very specific softwares, and mostly High-Performance Computing (HPC) applications. A few reasons come to mind to explain why the high performance of these protocols is not a sufficient reason for a wider adoption.

8.2.1. A complex programming interface

As described in chapter 6, the programming interface to use RDMA, Verbs [105], is quite complex. Setting up a basic RDMA connection and performing a RDMA "hello world" can already require hundreds of lines of code [38]. It necessitates to initiate several data structures and submit them to the network card, following a relatively strict order [39]. Indeed, some values depend on previous answers from the network card, such as the initial Packet Sequence Number (PSN) when the queue pair is created.

From a developer point of view, debugging RDMA code can also be quite painful. In case of a problem, RDMA Network cards are only able to raise events with an error code, which can give hints on the origin, but rarely more. High network speed can also prevent from good debugging. Indeed, error events can sometimes be raised only after several thousands of packets are exchanged, making network captures very hard to analyse. Similarly, even simple debugging operations can become an obstacle when the network can exchange more than 10GiB of data per second. For instance, capturing network exchanges at this throughput cannot be done easily. First, because RDMA operations are by design done in hardware by the network card. In order to capture packets, one has to use a dedicated tool, provided by Mellanox as a docker container [93]. Second, because the capture itself starts to miss packets above a certain throughput, probably to avoid congestion if the capturing machine tends to slow down the transfer. A network capture with missing packets is less likely to help at finding an unexpected behavior.

However, while being complex and hard to master, the Verbs API is probably the reason why RDMA transfers can be so efficient. In regular computer architectures, writing directly inside DRAM from a distant machine in a zero-copy fashion cannot come for free, it requires to adapt the software with the actual hardware requirements to achieve such peculiar memory transfers. The Verbs API is complex, but that complexity comes from the need to circumvent several software (*i.e.*, the kernel) or even hardware (*i.e.*, the MMU) limitations.

8.2.2. A challenge for performance

RDMA networking shines when using one-sided operations. That being said, it is not trivial to adapt a software behavior to these one-sided operations, in particular on the receiving side. A server receiving RDMA one-sided operations is not even notified when such operation has been performed in its memory region.

Generally, integrating RDMA in an application requires to use both one-sided and two-sided operations, playing with the notifications of the latter to notify the receiver side when it requires an action.

Another downside of one-sided operations is the management of memory pointers. In this scenario, the sender has to know the exact location of a data structure to be able to make zero-copy in-place transfers. That adds a heavy constraint on the development style choosen for the application.

In practice, applications that are not explicitely designed for RDMA compliance are not going to fully benefit from the performance boost. Since the availability of RDMA, many research papers showed the potential of this technology when it is applied to concrete applications [144, 90, 51, 121, 48]. However, these generally require quite deep modifications in the source code, and tweaks that can deserve a dedicated research paper.

8.2.3. A unique API

There is no transparent way to use RDMA in a program. The behavior of the protocol does not easily adapt to the more traditional interfaces used to program network exchanges. For instance, the Portable Operating System Interface (POSIX) socket API [86] has been designed in a way that cannot easily be reproduced within RDMA. One reason is that the RDMA programming interface is essentially asynchronous, while POSIX sockets are by default synchronous. Realistically, that obstacle could be circumvented, but deeper details of the RDMA API could not. First, in order to use RDMA in a one-sided zero-copy fashion, one needs to register the memory areas used as source (on the sender side) and destination (on the receiver side). Memory areas that have not been registered are not accessible by the network card, as it cannot ask the CPU MMU to translate the virtual address asked into a physical one. Then, interacting with another machine's memory requires to know the right virtual addresses inside that other machine, either for reading or writing data. In practice, buffer addresses are exchanged when the connection is established, via another communication protocol. Last but not least, using the one-sided programming model efficiently can impact a lot on the inner logic of a program. In particular, the receiver side behavior may cause problems, as for most applications, a notification triggers an action. Working without that trigger is often impossible, thus programmers must use RDMA operations that embed a notification request. In this case, the receiver can be woken up by an event pushed by the network card into the event queue.

The closest implementation of a POSIX-like API is the `rsocket` library [87]. While being easier to use, the limited features of this approach and the constraints to use it highlight how complicated adapting RDMA to another programming model can be. With a more conventional and portable programming model, it seems likely that RDMA could be used in more diverse applications. Instead, RDMA remains mostly used in some storage or HPC applications that, in all likelihood, influenced its API design.

8.3. Perspectives

While RoCE-capable networks gained some popularity in datacenters, only a subset of applications currently make use the remote memory access feature. Yet, efficient access to memory through the network and memory disaggregation are a necessary building block in the context of Hyper-Converged Infrastructure (HCI). This statement opens a debate on the future of memory disaggregation, and in particular in the role of RDMA in this future new memory paradigm.

8.3.1. CXL

In summer 2022, the Compute Express Link (CXL) consortium published their version 3.0 of the CXL specification, in which they describe how memory could be accessed remotely with CXL. Even if CXL 3.0 is currently only available on paper (no compatible hardware exists yet), it is a promising contender to become a popular memory disaggregation support, for several reasons. First, because CXL is an open industry standard, backed by an impressive list of industrial actors. In the adoption of a new standard interface, a high number of companies actually interested in its standardization is a good sign. Second, it is designed to reach low latencies even for remote memory extensions (around 600ns), making it a new intermediate between accesses to local memory and RDMA. Some early work on CXL compared it with RDMA accesses and reached significant improvements[42] on the latency. Third, even if it can support RAM extension cards, CXL is also meant to be compatible with a large range of extension devices, from memory-equipped accelerators, *i.e.*, Graphics Processing Unit (GPU)s or Field-Programmable Gate Array (FPGA)s, to persistent memory. And last but not least, CXL makes all of these memories accessible with the standard `load/store` interface, while giving guarantees on the cache coherence of the overall system.

However, CXL also comes with some limitations that makes it unlikely to strictly replace RDMA. A first barrier for wide scale adoption, is the need of fully compatible hardware. Inside one machine, even if CXL devices simply plug on a Peripheral Component Interconnect Express (PCIe) port, using CXL requires to have a compatible CPU, extension module, and motherboard. In order to use the rack-scalability feature, each machine will likely require a specific CXL port, and the rack would contain a dedicated CXL switch. Apart from the physical limitations, CXL being based on the PCIe bus, it is not designed for long distance communications and will likely be limited to the rack scale. To break this limitations, some projects use another transportation protocol to embed CXL packets [128]. Inevitably, this solution would increase the latencies of pure CXL transactions. Yet, it could allow to extend CXL and make it a real competitor for RDMA replacement. Nonetheless, while local CXL calls are expected to take around 200 to 400ns, RDMA requests generally take 2 to 4µs [92]. Thus, there is a high chance CXL and RDMA will coexist, as each of them represent a distinct layer in the memory hierarchy.

8.3.2. The end of the computer-centric era

Among the different technologies that are trendy enough to be considered as a probable perspective, there is a common point. They all take the direction of complete hardware disaggregation. RDMA, CXL, In-network computing, programmable networks, In-memory computing... all of these popular research topics are backed by major industrial actors. Each of them questions the idea of monolithic machines, in a context where scaling is the main challenge. Moreover, they all question the central role of the CPU and its main memory.

It is probable that in the coming years, full hardware disaggregation becomes standard for clusters and data centers. If so, cloud providers could have a rack of CPUs, a rack of main memory, a rack of storage, and maybe a rack of more exotic components such as GPUs, Non-Volatile Main Memory (NVMM)[111]... But the main barrier remains the main memory disaggregation [1], as it is a very sensitive parameter of the entire system's latency. In order to keep acceptable performances, the interconnect between all of these components has to reach extremely low latencies and very high throughput. Even if RDMA is impressive by its ability to reach lower latencies through a network, it still requires at least around one order of magnitude more than a typical local DRAM access. In comparison, CXL can reach latencies similar to a Non-Uniform Memory Access (NUMA) memory access, which is still slower but definitely closer than RDMA.

On the software side, some projects already address the problems a fully disaggregated architecture would cause [116, 37, 117]. In practice, such an architecture would already be a particularly good match for distributed, serverless and virtualization systems [85, 107] as it allows using any part of the computer as an on-demand resource.

Chapter 9.

Conclusion

Memory is a central element in modern computers, and there have been recent innovations that demonstrate the potential of exploring new memory paradigms. In this thesis, we experimented with two emerging memory-related technologies, NVMM and RDMA. Even if these two approaches seem at first, completely unrelated, they are in fact part of the same story. Since the core architecture of our computers has become standard, memory has always been designed only to assist a CPU in its computation. This strategy made sense for a long time, but the current needs for computing may question this choice, as memory could become a more versatile device.

Nonetheless, exploring these new ideas is a complex journey. Indeed, challenging the hegemonic way we use memory in all of our systems may not always be a successful adventure. As illustrated by the rise and fall of Intel's NVMM technology, Intel Optane, altering our interpretation of memory can look like a dead end, even to a giant industrial actor. Developing such deep changes in the core of our computer architecture requires a global acceptation and integration. In order to convince industrial actors worldwide, the technological change must be a significant breakthrough, but this is not a sufficient condition. It must also be an innovation safe to invest in, and this is probably where Intel Optane did not convince.

However, some changes are to be expected in the upcoming years, and this is due to the diversification of workloads. With the rise of AI training, streaming services, and various kinds of cloud computing, the ways we build and use our computers has never been so varied. Thus, some of these workloads are particularly suited with specialized hardware, such as GPUs, FPGAs, or even ASICs. In the same way, some workloads are likely to perform better with different kinds of memory in the future.

Building scalable platforms with dedicated hardware is another challenge we are about to face. The dynamics of research and industry on this topic clearly tend towards a complete disaggregation of computers as we know them. By increasing the

modularity of our systems, we may create machines able to maintain high performance while being able to dynamically scale depending on the computing needs. But this project also requires the industry to take some form of risk, by being early adopters of new technologies.

When searching about these topics, a particular technology seems to stand out: CXL. At this point, CXL may become the next long-term standard for all of our computer components, or it may disappear in a few years. However, there are some hints that CXL should become popular. First, it is backed by a consortium of industrial actors, in which we find manufacturers but also future potential adopters of the compatible devices. Also, the standard is created to ease compatibility between various devices, and will not be the closed system of a single manufacturer. Finally, the first CXL 1.0 devices have already been commercialized. Even if they do not have the most interesting features CXL 2.0 and 3.0 standards are supposed to provide, the production has started, and it is likely we will hear about CXL a lot in the coming years.

Bibliography

[1] Marcos K. Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam and Gerd Zellweger. 2023. Memory disaggregation: why now and what are the challenges. *ACM SIGOPS Oper. Syst. Rev.*, 57, 1, 38–46.

[2] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis and Igor Zablotchi. 2020. Microsecond consensus for microsecond applications. In *OSDI*. USENIX Association, 599–616.

[3] Gal Assa, Andreia Correia, Pedro Ramalhete, Valerio Schiavoni and Pascal Felber. 2023. Tl4x: buffered durable transactions on disk as fast as in memory. In *PPoPP*. ACM, 245–259.

[4] Jens Axboe. [n. d.] Fio-flexible I/O tester synthetic benchmark. https://github.com/axboe/fio. Accessed: 2020-05-25. ().

[5] Jonathan Behrens, Sagar Jha, Ken Birman and Edward Tremel. 2018. RDMC: A reliable RDMA multicast for large objects. In *DSN*. IEEE Computer Society, 71–82.

[6] Giacomo Belocchi, Valeria Cardellini, Aniello Cammarano and Giuseppe Bianchi. 2020. Paxos in the NIC: hardware acceleration of distributed consensus protocols. In *DRCN*. IEEE, 1–6.

[7] Lawrence Benson, Leon Papke and Tilmann Rabl. 2022. Perma-bench: benchmarking persistent memory access. *Proc. VLDB Endow.*, 15, 11, 2463–2476.

[8] Gautier Berthou and Vivien Quéma. 2013. Fastcast: a throughput-and latency-efficient total order broadcast protocol. In *Middleware 2013: ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings 14*. Springer, 1–20.

[9] Philippe Biondi et al. 2023. Scapy. Packet crafting for Python2 and Python3. https://scapy.net/.

[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese and David Walker. 2014. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.*, 44, 3, 87–95.

[11] Broadcom Inc. 2022. RDMA over Converged Ethernet feature in Ethernet NIC controllers. Retrieved 5 Sept. 2022 from https://techdocs.broadcom.com/us/en/storage-and-ethernet-connectivity/ethernet-nic-controllers/bcm957xxx/1-0/introduction/features_27/rdma-over-converged-ethernet--roce.html.

[12] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang and Dan R. K. Ports. 2021. PRISM: rethinking the RDMA interface for distributed systems. In *SOSP*. ACM, 228–242.

[13] Zhichao Cao, Siying Dong, Sagar Vemuri and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, (Feb. 2020), 209–223. ISBN: 978-1-939133-12-0. https://www.usenix.org/conference/fast20/presentation/cao-zhichao.

[14] Aleksey Charapko, Ailidani Ailijiang and Murat Demirbas. 2021. PigPaxos: Devouring the communication bottlenecks in distributed consensus. In *Proceedings of the 2021 International Conference on Management of Data*, 235–247.

[15] Ge Chen, Gaoxiong Zeng and Li Chen. 2021. P4COM: in-network computation with programmable switches. *CoRR*, abs/2107.13694. https://arxiv.org/abs/2107.13694 arXiv: 2107.13694.

[16] Xiang Chen, Dong Zhang, Xiaojun Wang, Kai Zhu and Haifeng Zhou. 2019. P4SC: towards high-performance service function chain implementation on the p4-capable device. In *IM*. IFIP, 1–9.

[17] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang and Jiwu Shu. 2020. Utree: a persistent b+-tree with low tail latency. *Proceedings of the VLDB Endowment*, 13, 12, 2634–2648.

[18] Felix Cloutier. 2023. Online x86 assembly reference: clflush operation. Retrieved 25 Nov. 2023 from https://www.felixcloutier.com/x86/clflush.

[19] Felix Cloutier. 2023. Online x86 assembly reference: clflushopt operation. Retrieved 25 Nov. 2023 from https://www.felixcloutier.com/x86/clflushopt.

[20] Felix Cloutier. 2023. Online x86 assembly reference: clwb operation. Retrieved 25 Nov. 2023 from https://www.felixcloutier.com/x86/clwb.

[21] CXL Consortium. 2023. Cxl consortium main page. Retrieved 25 Nov. 2023 from https://www.computeexpresslink.org/.

[22] SQLite Consortium. 2023. SQLite. Retrieved 25 Nov. 2023 from.

[23] The P4 Language Consortium. 2018. The p4 14 language specification. Retrieved 10 Nov. 2023 from https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.

[24] The P4 Language Consortium. 2023. The p4 16 language specification. Retrieved 10 Nov. 2023 from https://staging.p4.org/p4-spec/docs/P4-16-v1.2.4.pdf.

[25] Natanael Copa et al. 2023. Alpine Linux. Retrieved 25 Nov. 2023 from https://alpinelinux.org/.

[26] Jonathan Corbet. 2014. Supporting filesystems in persistent memory. *Linux Weekly News*.

[27] Andreia Correia, Pascal Felber and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (SPAA '18). Association for Computing Machinery, Vienna, Austria, 271–282. ISBN: 9781450357999. DOI: 10.1145/3210377.3210392.

[28] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone and Robert Soulé. 2020. P4xos: consensus as a network service. *IEEE/ACM Trans. Netw.*, 28, 4, 1726–1738.

[29] Huynh Tu Dang, Marco Canini, Fernando Pedone and Robert Soulé. 2016. Paxos made switch-y. *Comput. Commun. Rev.*, 46, 2, 18–24.

[30] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone and Robert Soulé. 2015. Netpaxos: consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, 1–7.

[31] Joeri de Ruiter and Caspar Schutijser. 2021. Next-generation internet at terabit speed: SCION in P4. In *CoNEXT*. ACM, 119–125.

[32] Linux Documentation. 2023. Linux kernel documentation of the dax feature. Retrieved 25 Nov. 2023 from https://www.kernel.org/doc/html/latest/filesystems/dax.html.

[33] Linux Documentation. 2023. Linux kernel documentation of the ext2 file system. Retrieved 25 Nov. 2023 from https://www.kernel.org/doc/html/latest/filesystems/ext2.html.

[34] Linux Documentation. 2023. Linux kernel documentation of the ext4 file system. Retrieved 25 Nov. 2023 from https://www.kernel.org/doc/html/v4.19/filesystems/ext4/ext4.html.

[35] Linux Documentation. 2023. Linux kernel documentation of the xfs file system. Retrieved 25 Nov. 2023 from https://docs.kernel.org/admin-guide/xfs.html.

[36] Rémi Dulong. 2023. Github repository of nvcache. Retrieved 25 Nov. 2023 from https://github.com/Xarboule/nvcache.

[37] Brice Ekane, Alain Tchana, Daniel Hagimont, Boris Teabe and Noel De Palma. 2023. Networking in next generation disaggregated datacenters. *Concurr. Comput. Pract. Exp.*, 35, 21.

[38] [n. d.] Example of a simple hello world code with rdma. Retrieved 10 Nov. 2023 from https://github.com/Arlu/RDMA-Hello-World/tree/36a00ffa8a8ba0f9debaee7810972ba536072fbf.

[39] [n. d.] Exchange identifier information to establish connection and change the queue pair state. Retrieved 10 Nov. 2023 from https://insujang.github.io/2020-02-09/introduction-to-programming-infiniband/#5-exchange-identifier-information-to-establish-connection-and-6-change-the-queue-pair-state.

[40] Rich Felker et al. 2023. Musl libc. Retrieved 25 Nov. 2023 from https://musl.libc.org/.

[41] Aoxiang Feng, Dezun Dong, Fei Lei, Junchao Ma, Enda Yu and Ruiqi Wang. 2023. In-network aggregation for data center networks: A survey. *Comput. Commun.*, 198, 63–76.

[42] Donghyun Gouk, Sangwon Lee, Miryeong Kwon and Myoungsoo Jung. 2022. Direct access, high-performance memory disaggregation with directcxl. In *USENIX Annual Technical Conference*. USENIX Association, 287–294.

[43] Rachid Guerraoui, Ron R. Levy, Bastian Pochon and Vivien Quéma. 2010. Throughput optimal total order broadcast for cluster environments. *ACM Trans. Comput. Syst.*, 28, 2, 5:1–5:32. DOI: 10.1145/1813654.1813656.

[44] Rachid Guerraoui, Antoine Murat and Athanasios Xygkis. 2021. Velos: one-sided paxos for RDMA applications. *CoRR*, abs/2106.08676. https://arxiv.org/abs/2106.08676 arXiv: 2106.08676.

[45] Paul Alcorn (Tom's Hardware). 2019. Intel optane dimm pricing: $695 for 128gb, $2595 for 256gb, $7816 for 512gb. Retrieved 25 Nov. 2023 from https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html.

[46] Guy Harris and Michael Richardson. 2022. PCAP Capture File Format. Internet-Draft draft-ietf-opsawg-pcap-01. Work in Progress. Internet Engineering Task Force, (29 July 2022). https://datatracker.ietf.org/doc/draft-ietf-opsawg-pcap/01/.

[47] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton and Patrick Eugster. 2017. Nvthreads: practical persistence for multi-threaded applications. In *EuroSys*. ACM, 468–482.

[48] Bobo Huang, Li Jin, Zhihui Lu, Ming Yan, Jie Wu, Patrick C. K. Hung and Qifeng Tang. 2019. Rdma-driven mongodb: an approach of RDMA enhanced nosql paradigm for large-scale data processing. *Inf. Sci.*, 502, 376–393.

[49] Chenchen Huang, Huiqi Hu and Aoying Zhou. 2021. Bptree: an optimized index with batch persistence on optane DC PM. In *DASFAA (3)* (Lecture Notes in Computer Science). Vol. 12683. Springer, 478–486.

[50] Jianming Huang and Yu Hua. 2023. From ideal to practice: data encryption in eadr-based secure non-volatile memory systems. *CoRR*, abs/2307.02050.

[51] Wei Huang, Qi Gao, Jiuxing Liu and Dhabaleswar K. Panda. 2007. High performance virtual machine migration with RDMA over modern interconnects. In *CLUSTER*. IEEE Computer Society, 11–20.

[52] InfiniBand Trade Association. 2020. *InfiniBand Architecture Specification. Volume 1*. Version 1.4. (7 Apr. 2020).

[53] Intel. 2023. Eadr technology presentation. Retrieved 25 Nov. 2023 from https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[54] Intel. 2023. Eadr, new opportunities for persistent memory applications. Retrieved 25 Nov. 2023 from https://www.intel.cn/content/www/cn/zh/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[55] 2023. Intel 3D XPoint™ Technology. https://www.intel.fr/content/www/fr/fr/products/details/memory-storage/optane-dc-persistent-memory.html. (2023).

[56] Intel Corporation. 2021. Which Intel Ethernet network adapters support iWARP and RoCE v2? Retrieved 5 Sept. 2022 from https://www.intel.com/content/www/us/en/support/articles/000031905/ethernet-products/700-series-controllers-up-to-40gbe.html.

[57] [SW] Intel Corporation, Open Tofino 2023. URL: https://github.com/barefootnetworks/open-tofinoRetrieved 1 Sept. 2022 from.

[58] Intel Corporation. [n. d.] Intel Tofino intelligent fabric processors. Retrieved 1 Nov. 2023 from https://www.intel.fr/content/www/fr/fr/products/details/network-io/intelligent-fabric-processors.html.

[59] 2020. Intel® Optane™ DC Persistent Memory. https://intel.ly/2WFisT8. (2020).

[60] Zsolt István, David Sidler, Gustavo Alonso and Marko Vukolic. 2016. Consensus in a box: inexpensive coordination in hardware. In *NSDI*. USENIX Association, 425–438.

[61] Joseph Izraelevitz, Terence Kelly and Aasheesh Kolli. 2016. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '16). Association for Computing Machinery, Atlanta, Georgia, USA, 427–442. ISBN: 9781450340915. DOI: 10.1145/2872362.2872410.

[62] Joseph Izraelevitz, Hammurabi Mendes and Michael L Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*. Springer, 313–327.

[63] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao and Steven Swanson. 2019. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714.

[64] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son and Youjip Won. 2013. I/o stack optimization for smartphones. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (USENIX ATC'13). USENIX Association, San Jose, CA, 309–320.

[65] Chengfan Jia, Junnan Liu, Xu Jin, Han Lin, Hong An, Wenting Han, Zheng Wu and Mengxian Chi. 2018. Improving the performance of distributed tensorflow with RDMA. *Int. J. Parallel Program.*, 46, 4, 674–685.

[66] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli and Vijay Chidambaram. 2019. SplitFS: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (SOSP '19). ACM, Huntsville, Ontario, Canada, 494–508. ISBN: 9781450368735. DOI: 10.1145/3341301.3359631.

[67] Mikhail Kazhamiaka, Babar Naveed Memon, Chathura Kankanamge, Siddhartha Sahu, Sajjad Rizvi, Bernard Wong and Khuzaima Daudjee. 2019. Sift: resource-efficient consensus with RDMA. In *CoNEXT*. ACM, 260–271.

[68] Elie F. Kfoury, Jorge Crichigno and Elias Bou-Harb. 2020. Offloading media traffic to programmable data plane switches. In *ICC*. IEEE, 1–7.

[69] Elie F. Kfoury, Jorge Crichigno and Elias Bou-Harb. 2021. An exhaustive survey on P4 programmable data plane switches: taxonomy, applications, challenges, and future trends. *IEEE Access*, 9, 87094–87155.

[70] Ana Khorguani. 2023. Intel's clwb invalidating cache lines. Retrieved 25 Nov. 2023 from https://stackoverflow.com/questions/60266778/intels-clwb-instruction-invalidating-cache-lines.

[71] Ana Khorguani, Thomas Ropars and Noel De Palma. 2022. Respct: fast checkpointing in non-volatile memory for multi-threaded applications. In *EuroSys*. ACM, 525–540.

[72] Daehyeok Kim, Amir Saman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar and Srinivasan Seshan. 2018. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *SIGCOMM*. ACM, 297–312.

[73] Gyuyeong Kim and Wonjun Lee. 2022. In-network leaderless replication for distributed data stores. *Proc. VLDB Endow.*, 15, 7, 1337–1349.

[74] Hyojun Kim, Nitin Agrawal and Cristian Ungureanu. 2012. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)*, 8, 4, 1–25.

[75] Marios Kogias and Edouard Bugnion. 2020. Hovercraft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *EuroSys*. ACM, 25:1–25:17.

[76] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (SOSP '17). ACM, Shanghai, China, 460–477. ISBN: 9781450350853. DOI: 10.1145/3132747.3132770.

[77] Long Hoang Le, Mojtaba Eslahi-Kelorazi, Paulo R. Coelho and Fernando Pedone. 2021. Ramcast: rdma-based atomic multicast. In *Middleware*. ACM, 172–184.

[78] Anatole Lefort, Yohan Pipereau, Kwabena Amponsem, Pierre Sutra and Gaël Thomas. 2021. J-NVM: off-heap persistent objects in java. In *SOSP*. ACM, 408–423.

[79] Kin-Wai Leong, Zhilong Li and Yunqu Leon Liu. 2019. Reliable multicast using remote direct memory access (RDMA) over a passive optical cross-connect fabric enhanced with wavelength division multiplexing (WDM). *APSIPA Transactions on Signal and Information Processing*, 8, e25.

[80] Baptiste Lepers, Oana Balmau, Karan Gupta and Willy Zwaenepoel. 2019. Kvell: the design and implementation of a fast persistent key-value store. In *SOSP*. ACM, 447–461.

[81] Baptiste Lepers, Oana Balmau, Karan Gupta and Willy Zwaenepoel. 2020. Kvell+: snapshot isolation without snapshots. In *OSDI*. USENIX Association, 425–441.

[82] Baptiste Lepers and Willy Zwaenepoel. 2023. Johnny cache: the end of DRAM cache conflicts (in tiered main memory systems). In *OSDI*. USENIX Association, 519–534.

[83] LF Projects, LLC. 2022. Data plane development kit. Retrieved 31 Aug. 2022 from https://www.dpdk.org/.

[84] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres and Dan RK Ports. 2016. Just say no to paxos overhead: replacing consensus with network ordering. In *OSDI*, 467–483.

[85] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen and Jiajie Sheng. 2023. ROLEX: A scalable rdma-oriented learned key-value store for disaggregated memory systems. In *FAST*. USENIX Association, 99–114.

[86] [n. d.] Linux manual page of the posix socket api. Retrieved 10 Nov. 2023 from https://www.man7.org/linux/man-pages/man2/socket.2.html.

[87] [n. d.] Linux manual page of the rsocket api. Retrieved 10 Nov. 2023 from https://linux.die.net/man/7/rsocket.

[88] Shuo Liu, Qiaoling Wang, Junyi Zhang, Qinliang Lin, Yao Liu, Meng Xu, Ray C. C. Cheung and Jianfei He. 2020. Netreduce: rdma-compatible in-network reduction for distributed DNN training acceleration. *CoRR*, abs/2009.09736. https://arxiv.org/abs/2009.09736 arXiv: 2009.09736.

[89] Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray CC Cheung and Jianfei He. 2023. In-network aggregation with transport transparency for distributed training. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 376–391.

[90] Xiaoyi Lu, Md. Wasi-ur-Rahman, Nusrat S. Islam, Dipti Shankar and Dhabaleswar K. Panda. 2014. Accelerating spark with RDMA for big data processing: early experiences. In *Hot Interconnects*. IEEE Computer Society, 9–16.

[91] Linux manual. 2023. Linux manual page of the open system call. Retrieved 25 Nov. 2023 from https://www.man7.org/linux/man-pages/man2/open.2.html.

[92] Hasan Al Maruf and Mosharaf Chowdhury. 2023. Memory disaggregation: advances and open challenges. *ACM SIGOPS Oper. Syst. Rev.*, 57, 1, 29–37.

[93] Mellanox. [n. d.] Tcpdump rdma docker container. Retrieved 10 Nov. 2023 from https://hub.docker.com/r/mellanox/tcpdump-rdma/.

[94] Benjamin Michalowicz, Kaushik Kandadi Suresh, Hari Subramoni, Dhabaleswar K. D. K. Panda and Stephen W. Poole. 2023. Battle of the bluefields: an in-depth comparison of the bluefield-2 and bluefield-3 smartnics. In *HOTI*. IEEE, 41–48.

[95] Inc Micron Technology. 2023. Ddr4 sdram nvrdimm data-sheet. Retrieved 25 Nov. 2023 from https://www.micron.com/-/media/client/global/documents/products/data-sheet/modules/nvdimm/ddr4/asf18c2gx72pf1z_rv.pdf.

[96] *Global Disruption of Semiconductor Supply Chains During COVID-19: An Evaluation of Leading Causal Factors*, vol. Volume 2: Manufacturing Processes; Manufacturing Systems of *International Manufacturing Science and Engineering Conference*, (June 2022), V002T06A011. eprint: https://asmedigitalcollection.asme.org/MSEC/proceedings-pdf/MSEC2022/85819/V002T06A011/6922569/v002t06a011-msec2022-85306.pdf. DOI: 10.1115/MSEC2022-85306.

[97] Gordon E. Moore. 1965. Moore's law. Retrieved 25 Nov. 2023 from http://large.stanford.edu/courses/2012/ph250/kumar1/docs/Gordon_Moore_1965_Article.pdf.

[98] Mozilla. 2023. Firefox browser. Retrieved 25 Nov. 2023 from https://www.mozilla.org/en-US/firefox/new/.

[99] Mozilla. 2023. Thunderbird email client. Retrieved 25 Nov. 2023 from https://www.thunderbird.net/en-US/.

[100] 2023. Npl: open, high-level language for developing feature-rich solutions for programmable networking platforms. Retrieved 10 Nov. 2023 from https://nplang.org.

[101] NVIDIA. 2023. In-network computing and next generation hdr 200g infiniband. Retrieved 10 Nov. 2023 from https://network.nvidia.com/pdf/whitepapers/WP_In-Network_Computing_Next_Generation_HDR_200G_IB.pdf.

[102] NVIDIA. 2023. Nvidia bluefield networking platform. Retrieved 10 Nov. 2023 from https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[103] NVIDIA. 2023. Nvidia gpudirect over rdma. Retrieved 10 Nov. 2023 from https://network.nvidia.com/products/GPUDirect-RDMA/.

[104] NVIDIA. 2023. Nvidia sharp. Retrieved 10 Nov. 2023 from https://docs.nvidia.com/networking/display/sharpv300.

[105] NVIDIA. [n. d.] Programming examples using ibv verbs. Retrieved 10 Nov. 2023 from https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/programming+examples+using+ibv+verbs.

[106] NVIDIA Corporation. 2022. ConnectX SmartNICs. Retrieved 5 Sept. 2022 from https://www.nvidia.com/en-us/networking/ethernet-adapters/.

[107] Adarsh Patil, Vijay Nagarajan, Nikos Nikoleris and Nicolai Oswald. 2023. Āpta: fault-tolerant object-granular CXL disaggregated memory for accelerating faas. In *DSN*. IEEE, 201–215.

[108] Matthew Poremba, Tao Zhang and Yuan Xie. 2015. Nvmain 2.0: a user-friendly memory simulator to model (non-)volatile memory systems. *IEEE Computer Architecture Letters*, 14, 2, 140–143. DOI: 10.1109/LCA.2015.2402435.

[109] P. Ramalhete, A. Correia, P. Felber and N. Cohen. 2019. OneFile: A Wait-Free Persistent Transactional Memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. (June 2019), 151–163. DOI: 10.1109/DSN.2019.00028.

[110] Carmine Rizzi, Zhiyuan Yao, Yoann Desmouceaux, Mark Townsley and Thomas H. Clausen. 2021. Charon: load-aware load-balancing in P4. In *CNSM*. IEEE, 91–97.

[111] Chaoyi Ruan, Yingqiang Zhang, Chao Bi, Xiaosong Ma, Hao Chen, Feifei Li, Xinjun Yang, Cheng Li, Ashraf Aboulnaga and Yinlong Xu. 2023. Persistent memory disaggregation for cloud-native relational databases. In *ASPLOS (3)*. ACM, 498–512.

[112] Samsung. 2023. Samsung unveils industry first memory module incorporating new cxl interconnect standard. Retrieved 25 Nov. 2023 from https://news.samsung.com/global/samsung-unveils-industry-first-memory-module-incorporating-new-cxl-interconnect-standard.

[113] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini and Panos Kalnis. 2017. In-network computation is a dumb idea whose time has come. In *HotNets*. ACM, 150–156.

[114] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports and Peter Richtárik. 2021. Scaling distributed machine learning with in-network aggregation. In *NSDI*. USENIX Association, 785–808.

[115] Steve Scargall. 2020. *Programming Persistent Memory, A Comprehensive Guide for Developers*. Open access (November 2023): https://link.springer.com/content/pdf/10.1007/978-1-4842-4932-1.pdf. Springer Nature.

[116] Yizhou Shan, Yutong Huang, Yilun Chen and Yiying Zhang. 2018. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*. USENIX Association, 69–87.

[117] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou and Michael R. Lyu. 2023. Ditto: an elastic and adaptive memory-disaggregated caching system. In *SOSP*. ACM, 675–691.

[118] Carol Sliwa. 2023. Intel breaks silence on effects of micron's 3d-xpoint exit. Accessed on November, 2023. (2023). Retrieved 25 Nov. 2023 from https://www.techtarget.com/searchstorage/news/252499716/Intel-breaks-silence-on-effects-of-Microns-3D-XPoint-exit.

[119] SS64.com. 2023. Osx nvram command line documentation. Retrieved 25 Nov. 2023 from https://ss64.com/osx/nvram.html.

[120] Rajesh Tadakamadla, Mikulas Patocka, Toshi Kani and Scott J. Norton. 2019. Accelerating Database Workloads with DM-WriteCache and Persistent Memory. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering* (ICPE '19). Association for Computing Machinery, Mumbai, India, 255–263. ISBN: 9781450362399. DOI: 10.1145/3297663.3309669.

[121] Wenhui Tang, Yutong Lu, Nong Xiao, Fang Liu and Zhiguang Chen. 2017. Accelerating redis with RDMA over infiniband. In *DMBD* (Lecture Notes in Computer Science). Vol. 10387. Springer, 472–483.

[122] Mellanox Technologies. [n. d.] Roce vs. iwarp competitive analysis. Accessed on November, 2023. (). https://network.nvidia.com/related-docs/whitepapers/WP_RoCE_vs_iWARP.pdf.

[123] [n. d.] Top500 highlights of june 2016. Accessed on November, 2023. (). https://www.top500.org/lists/top500/2016/06/highlights/.

[124] [n. d.] Top500 ranking of the most powerful supercomputers. Accessed on November, 2023. (). https://www.top500.org/.

[125] ukontainer. 2023. Sqlite3 dbbench benchmark port. Retrieved 25 Nov. 2023 from https://github.com/ukontainer/sqlite-bench.

[126] Sébastien Vaucher, Niloofar Yazdani, Pascal Felber, Daniel E. Lucani and Valerio Schiavoni. 2020. Zipline: in-network compression at line speed. In *CoNEXT*. ACM, 399–405.

[127] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi and Heming Cui. 2017. APUS: fast and scalable paxos on RDMA. In *SoCC*. ACM, 94–107.

[128] Chenjiu Wang, Ke He, Ruiqi Fan, Xiaonan Wang, Wei Wang and Qinfen Hao. 2023. CXL over ethernet: A novel fpga-based memory disaggregation design in data centers. In *FCCM*. IEEE, 75–82.

[129] Shie-Yuan Wang, Chia-Ming Wu, Yi-Bing Lin and Ching-Chun Huang. 2019. High-speed data-plane packet aggregation and disaggregation by P4 switches. *J. Netw. Comput. Appl.*, 142, 98–110.

[130] William Wang and Stephan Diestelhorst. 2018. Quantify the performance overheads of PMDK. In *MEMSYS*. ACM, 50–52.

[131] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson and Jishen Zhao. 2020. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 496–508. DOI: 10.1109/MICRO50266.2020.00049.

[132] Wikipedia. 2023. Global chip shortage. (2023). Retrieved 25 Nov. 2023 from https://en.wikipedia.org/wiki/2020%E2%80%932023_global_chip_shortage.

[133] Wikipedia. 2023. Log-structured file system. Retrieved 25 Nov. 2023 from https://en.wikipedia.org/wiki/Log-structured_file_system.

[134] Wikipedia. 2023. Processor power dissipation (wikipedia). Retrieved 25 Nov. 2023 from https://en.wikipedia.org/wiki/Processor_power_dissipation.

[135] Wikipedia. 2023. Zipfian law. Retrieved 25 Nov. 2023 from https://en.wikipedia.org/wiki/Zipf%5C%27s%5C_law.

[136] Matthew Wilcox. 2014. Add support for NV-DIMMs to ext4. *https://lwn.net/Articles/613384*.

[137] [n. d.] Wireshark project home page. Accessed on November, 2023. (). https://www.wireshark.org/.

[138] Jian Xu and Steven Swanson. 2016. NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (FAST'16). USENIX Association, Santa Clara, CA, 323–338. ISBN: 9781931971287.

[139] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz and Steven Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *FAST*. USENIX Association, 169–182.

[140] Peterson Yuhala, Pascal Felber, Valerio Schiavoni and Alain Tchana. 2021. Plinius: secure and persistent machine learning model training. In *DSN*. IEEE, 52–62.

[141] Kaiyuan Zhang, Danyang Zhuo and Arvind Krishnamurthy. 2020. Gallium: Automated Software Middlebox Offloading to Programmable Switches. In *SIGCOMM*. ACM, 283–295.

[142] Wenhui Zhang, Xingsheng Zhao, Song Jiang and Hong Jiang. 2021. Chameleondb: a key-value store for optane persistent memory. In *EuroSys*. ACM, 194–209.

[143] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica and Xin Jin. 2019. Harmonia: near-linear scalability for replicated storage with in-network conflict detection. *Proc. VLDB Endow.*, 13, 3, 376–389.

[144] Tobias Ziegler, Carsten Binnig and Viktor Leis. 2022. Scalestore: A fast and cost-efficient storage engine using dram, nvme, and RDMA. In *SIGMOD Conference*. ACM, 685–699.

ECOLE
DOCTORALE

Titre : Vers de nouveaux paradigmes mémoire: Intégration de mémoire principale non-volatile et d'accès direct de mémoire distante dans les systèmes modernes

Mots clés : Mémoire principale non-volatile, Accès direct de mémoire distante, Mémoire désagrégée

Résumé : Les ordinateurs modernes sont construits autour de deux éléments : leur CPU et leur mémoire principale volatile, ou RAM. Depuis les années 1970, ce principe a été constamment amélioré pour offrir toujours plus de fonctionnalités et de performances.
Dans cette thèse, nous étudions deux paradigmes de mémoire qui proposent de nouvelles façons d'interagir avec la mémoire dans les systèmes modernes : la mémoire non-volatile et les accès mémoire distants. Nous mettons en œuvre des outils logiciels qui exploitent ces nouvelles approches afin de les rendre compatibles et d'exploiter leurs performances avec des applications concrètes. Nous analysons également l'impact des technologies utilisées, et les perspectives de leur évolution dans les années à venir.
En 2019, Intel a commercialisé un nouveau composant appelé Optane DCPMM qui rend possible l'utilisation de NVMM. Ce produit propose une nouvelle façon de penser la persistance des données. Mais il remet également en question l'architecture de nos machines et la manière dont nous les programmons.
Alors que les machines ont individuellement tendance à atteindre des limites de performances, l'utilisation de plusieurs machines et le partage des tâches sont devenus la nouvelle façon de créer des ordinateurs puissants. Pour cette raison, plusieurs protocoles de communication ont implémententé RDMA, un moyen de lire ou d'écrire directement dans la mémoire d'un serveur distant. RDMA offre de faibles latences et un débit élevé, contournant de nombreuses étapes de la pile réseau.
En utilisant ces deux technologies, nous remarquons que les futures générations de matériel pourraient nécessiter une nouvelle interface pour les mémoires de toutes sortes, afin de faciliter l'interopérabilité dans des systèmes qui ont tendance à devenir de plus en plus hétérogènes et complexes.

Title : Towards new memory paradigms: Integrating Non-Volatile Main Memory and Remote Direct Memory Access in modern systems

Keywords : Non-volatile main memory, Remote direct memory access, Memory disaggregation

Abstract : Modern computers are built around two main parts: their CPU, and their volatile main memory, or RAM. The basis of this architecture takes its roots in the 1970's first computers. Since, this principle has been constantly upgraded to provide more functionnality and performance.
In this thesis, we study two memory paradigms that drastically change the way we can interact with memory in modern systems: non-volatile memory and remote memory access. We implement software tools that leverage them in order to make them compatible and exploit their performance with concrete applications. We also analyze the impact of the technologies underlying these new memory medium, and the perspectives of their evolution in the coming years.
In 2019, Intel released a new component called Optane DCPMM, a device that made possible the use of NVMM. That product, by its capabilities, provides a new way of thinking about data persistence. Yet, it also challenges the hardware architecture used in our current machines and the way we program them.
As individual machines tend to reach performance limitations, using several machines and sharing workloads became the new way to build powerful computers. For that reason, several communication protocols implemented RDMA, a way to read or write directly into a distant machine's memory. RDMA provides low latencies and high throughput, bypassing many steps of the traditional network stack.
By using these two technologies, we notice that future generations of hardware may require a new interface for memories of all kinds, in order to ease the interoperability in systems that tend to get more and more heterogeneous and complex.

Institut Polytechnique de Paris
91120 Palaiseau, France