Research Statement

Anatole Lefort, *August 2023*

My research interests are in the field of computer systems, more specifically, about system support for modern distributed services and applications. Through my research, I have had hands-on experience in multiple system subfields, namely, *persistent memory* – from architectural concerns, to crash-resilient programming abstraction in applications, *programming language runtimes* and *distributed systems.*

Modern computing is unquestionably distributed. Applications offer personalized contents to user bases spanning the globe, while the cloud services cater for an unprecedented number of those applications. In this context, system software is challenged in a balancing act between general expectations from the two sides: (1) applications and services demand interfaces that combine *performance* and *ease-of-use* in accessing hardware resources, following their specific needs; and (2) cloud infrastructures can not compromise on *isolation* or *sharing flexibility* of those resources, as maximizing their utilization is key in scaling to large number of tenants.

I am generally interested in exploring opportunities in *systems design* across the stack, that can benefit either services or the platform in terms of *performance*, *productivity* or *functionality.* I also strive for research that builds real systems that demonstrate new use cases or tangible and non-trivial improvements over real-world workloads.

1 Recent Research Projects

I led a single research project during my PhD program at Télécom SudParis - Institut Polytechnique de Paris. The aim of my thesis was to design novel data types to enable the use of emerging persistent memory in big data workloads; and I built an all-around programming interface to efficiently access persistent memory in Java [3]. Before that, I also worked on a novel distributed protocol for genuine atomic multicast [2] during my masters' final internship, spent at IMDEA Software, Madrid, Spain.

1.1 Support for Persistent Memory in Java

General Context. Big data stores form the backbone of modern computing infrastructures. They support large data sets and enable processing frameworks to mine information from this data. They are designed for quick response and parallel computing at unprecedented scale. Recent examples of such systems include in-memory databases, NoSQL databases and key-value stores. Until recently, volatile media were orders of magnitude faster than persistent ones. This fundamental difference much impacted the way these systems are architectured.

In such systems, although processing occurs in main memory, the authoritative version of data is maintained on durable storage devices (SSD, disk) with significantly slower access times. Keeping those two versions of the data mutually consistent typically is a trade-off between: - performance bottlenecking on storage I/Os (*synchronous* persistence), or - degraded durability guarantees and increased software complexity (*asynchronous* persistence). Additionally, resuming operation after a system crash or software failure is bound to be slow, considering it needs to read from storage to reconstruct previous in-memory states or to repopulate a data cache.

The Advent of Persistent Memory. Recently released non-volatile main memory (NVMM), as fast and durable memory, promise to reshuffle the cards. First, they dramatically increase storage performance over traditional media (SSD, hard disk). Then, a substantial and unique property of NVMM is byte-addressability – complex memory data structures, maintained with regular load/store instructions, can now resist machine power-cycles, software faults or system crashes.

However, correctly managing persistence with the fine grain of memory instructions is laborious, with increased risk of compromising data integrity and recovery at any misstep. Programming abstractions from software libraries and support from language runtime and compilers are necessary to avoid memory bugs that are exacerbated with persistence. Considering the popularity of the Java language in developing modern data stores, data analytics or processing frameworks; bringing the full potential of NVMM to it could benefit a wide range of applications.

Overview. My thesis work addresses the challenges of supporting persistent memory in managed language environments by introducing J-NVM [3], a framework to efficiently access NVMM in Java. With J-NVM, we demonstrate how to design an efficient, simple and complete interface to weave NVMM-devised persistence into object-oriented programming, while remaining unobtrusive to the language runtime itself.

In detail, J-NVM offers a fully-fledged interface to persist plain Java objects using failure-atomic sections. This interface relies internally on proxy objects that intermediate direct off-heap access to NVMM. The framework also provides a library of highly-optimized persistent data types that resist reboots and power failures.

Key Insights. The novelty of this work lies with the *decoupling principle* I introduced between the data structure of a persistent object and its representation in the Java world. Following this principle, a persistent object now consists of two parts: (1) a data structure stored off-heap in NVMM that holds object fields, and (2) a proxy Java object that remains in volatile memory and provides the methods to manipulate those persistent fields.

Because NVMM is managed as off-heap memory, J-NVM avoids running garbage collection (GC) over its content at runtime. This design decision was motivated from our observation that the cost of GC becomes prohibitive when working on large data sets at NVMM scales. Our design also removes the cost of converting objects across media, because it is able to retain direct-access to NVMM and avoid duplicating data in DRAM; all by leveraging a JVM interface that inlines the low-level instructions that access NVMM directly in the Java methods.

Main Artifacts. J-NVM implements these key ideas as a lightweight pure-Java library that runs on Hotspot JVM with the minimal addition of three NVMM-specific instructions. J-NVM precisely is the low-level interface that focuses on efficient and crash-consistent heap management, and that provides the bare logic to instantiate/destroy persistent objects or efficiently access their fields. I build up from J-NVM three simple programming abstractions that ensure consistency of durable data and aid recoverability, namely, *J-PFA*, *J-PDT* and *J-Transformer*.

- *J-PFA* provides failure-atomic blocks of code, i.e., a generic way of making any section of code crash-consistent.
- *J-PDT* is a collection of hand-crafted crash-consistent data structures for NVMM (e.g., arrays, maps, trees).
- *J-Transformer* is a code generator that automates the translation of any plain old Java object class into a persistent one based on a single code annotation. It is implemented as an off-line Java bytecode to bytecode post-compilation transformation plugin, fully integrated in the application build system.

Experimental Results. I evaluated J-NVM by implementing several persistent backends for Infinispan [4] – an industrial-grade NoSQL data store – and investigated the impact of J-NVM design relative to: *data caching*, *marshalling*, or *GC*. The overall performance results, obtained with a TPC-B like workload [8] and the YCSB benchmark [1], show that: (1) J-NVM is consistently faster than other existing approaches at accessing NVMM in Java, by at least 10x across all YCSB workloads, and (2) J-NVM was between 4.7x and 8.6x faster to recover on the TPC-B like workload.

1.2 Atomic Multicast

Modern cloud services are built for large scale and geo-distribution. In such setting, faults become more frequent as the number of machines involved increases. Building resilient distributed systems is done more easily with stronger communication primitives. A classical approach is to rely on an *atomic broadcast* protocol to deliver application messages in some total order shared across replicas, to ensure they evolve in the same way. Unfortunately, the global service state is often too large to exist on each replicas, and has to be partitioned.

In this instance, *atomic multicast* can be used, as it solves the generalized problem of delivering messages following some total order, when each is addressed to an arbitrary set of processes. Further, atomic multicast protocols are said to be *genuine*, if only the processes in the destination set of a message participate in ordering it. Being genuine is key to enabling scalability of atomic multicast, as messages with disjoint destination sets can be ordered in parallel. For instance, genuine atomic multicast can be used to scale distributed logs or transaction processing systems that can be found in geo-distributed databases.

We proposed WB-amcast [2], a novel protocol for genuine atomic multicast that can deliver messages in only 3 network delays (5 under contention), when previous solutions needed 4 delays (8 under contention). We achieve this lower network complexity by parting from the traditional way of layering Skeen's reliable multicast with Paxos replication. Instead, we weave the two protocols together into a single cohesive one, and exploit opportunities for white-box optimization. The backlash is that the resulting protocol is more complex to reason about, especially in the recovery case for leader failures.

I implemented a WB-amcast prototype in C and evaluated it in both local-area and geo-distributed settings against the state of the art protocol and a naive baseline. In our experiments, we gradually increase the load on the cluster by adding closed-loop clients until it reaches saturation. We observe that WB-amcast outperforms other protocols by 2x on average, for both latency and throughput in the two settings alike. It also achieves better scalability as observed when raising the size of the destination set of messages. I also benchmarked the leader recovery procedure in the geo-distributed setting, it took around 6sec for the affected group to fully recover. In all, the superior theoretical characteristics of our protocol are well reflected in practical performance pay-offs.

2 Future Directions

Persistent Memory Perspectives. After working with persistent memory for the past few years, I can say it is truly a game changing piece of tech. Beyond being just faster storage, it is an opportunity to rethink how programs interact with persistent data at much finer grain. Of course, it could find natural use cases with *in-memory databases* and other storage services; but it may also be employed to create new and more exotic scenarios. For instance, to devise applicative read/write query caches in network devices [7], GPU-accelerated key-value stores [5], direct-access byte-addressable file systems [6], or perhaps even to build on its fast recovery capabilities to power data center servers through energy harvesting.

Unfortunately, these prospects were cut short when Intel – the only vendor of commercial NVMM hardware – discontinued their Optane Persistent Memory product line last year. Nonetheless, the encounter with persistent memory underlined shortcomings in current system software, essentially: (1) I/O stacks in the OS impede the latency of microsecond-scale devices, and (2) moving these functionality to applications for performance, also offloads all of the system's complexity to them. Going forward, we will need to rethink the design of the systems stack (esp. I/O data paths) for modern workloads to take advantage of emerging hardware, and overall, properly support upcoming changes in computing platforms.

Upcoming Computing Platforms. In particular, several hardware trends started to reshape commodity hardware available in data centers, towards more diversity and complexity. Addressing the opportunities presented by those trends will mostly be an exercise in *system design*, done in concordance with specific requirements of target workloads and their execution environment. That is to say, understanding both applications and hosting infrastructures is crucial in providing them the specialized interfaces they need.

- ***Active Devices.*** Vendors started or plan on embedding computational units in a variety of devices: in network (e.g., SmartNICs), storage (computational NVMe) and memory (Processing-In-Memory chips). Harnessing this extra processing power most likely will save precious CPU cycles by offloading strenuous tasks, as *packet processing/filtering*, *transport protocols*, *data marshalling*, or other basic overly used memory functions (e.g., *memcpy*, *strcmp*). Ultimately, applications will want to experiment and decide on which functional units are worth offloading to devices and be put closer to the data. The role of system software will then be to manage these new computational units, export their services to applications and schedule assigned tasks onto them.
- ***Heterogeneous Memory.*** Diverse memory technologies (e.g., non-volatile memory, HBM, or far/remote memory) with their own specific characteristics and access semantics will be available. Though, efficiently or correctly addressing them often requires additional hints from the application, due to potential discrepancies across their memory consistency models and that of the local DRAM. As with persistent memory, solutions likely lies with language compilers and runtime to break down the complexity from comprehensive programming abstractions.
- ***Rack-scale architectures.*** High-speed network fabrics (e.g., RDMA, upcoming CXL) are expected to become the norm, and institute diverse resource and device sharing capabilities between servers of a rack. Ultimately, the rack will replace servers as the basic building block of modern data centers. Although the exact shape remains unclear, it already raises an avalanche of questions. For instance, *how the whole software stack will cooperate to orchestrate safe and efficient multiplexing of resources*, or *how the added flexibility can be exploited to maximize occupancy in data centers, or provision arbitrary resource aggregates to accommodate specific workloads.*

References

[1] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, ACM Press, Indianapolis, Indiana, USA, 143. DOI:https://doi.org/10.1145/1807128.1807152

[2] Alexey Gotsman, Anatole Lefort, and Gregory Chockler. 2019. White-Box Atomic Multicast. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, Portland, OR, USA, 176–187. DOI:https://doi.org/10.1109/DSN.2019.00030

[3] Anatole Lefort, Yohan Pipereau, Kwabena Amponsem, Pierre Sutra, and Gaël Thomas. 2021. J-NVM: Off-heap Persistent Objects in Java. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, ACM, Virtual Event Germany, 408–423. DOI:https://doi.org/10.1145/3477132.3483579

[4] Francesco Marchioni and Manik Surtani. 2012. *Infinispan Data Grid Platform*. Packt Publishing Ltd.

[5] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. 2022. GPM: Leveraging persistent memory from a GPU. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, Lausanne Switzerland, 142–156. DOI:https://doi.org/10.1145/3503222.3507758

[6] Yujie Ren, Changwoo Min, and Sudarsun Kannan. 2020. CrossFS: A Cross-layered Direct-Access File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, 137–154. Retrieved from https://www.usenix.org/conference/osdi20/presentation/ren

[7] Korakit Seemakhupt, Sihang Liu, Yasas Senevirathne, Muhammad Shahbaz, and Samira Khan. 2021. PMNet: In-Network Data Persistence. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, Valencia, Spain, 804–817. DOI:https://doi.org/10.1109/ISCA52012.2021.00068

[8] 1990. TPC-B. Retrieved from http://www.tpc.org/tpcb