With the end of DRAM scaling in sight, new memory types are competing to replace or augment DRAM as the main memory technology of choice. However, unlike DRAM, these new memory technologies all share an interesting property: they are nonvolatile (their contents survive power outages). These nonvolatile memory (NVM) technologies provide us the opportunity to re-think the memory–storage divide, and to entertain the possibility of maintaining traditional in-memory data structures across program runs and crashes.

My research has addressed the entire system stack in preparation for nonvolatile memory. The introduction of NVM creates both opportunities and problems. Opportunities, because the technology promises fast durable storage, low-power machine modes, and low overhead checkpointing. Problems, because traditional machine architectures, operating systems, compilers, and system libraries are not built to manage or leverage nonvolatile main memory. Serious work remains to be done in adapting the system stack to NVM if we are to realize the promise of this technology. There is a vast gap between the promise of NVM and the capabilities of current machines—it is a fruitful area for research.

If we can redesign the system stack, we can expect NVM to be a transformative technology that enables scientific advances well beyond the current state of the art. Using NVM to its full potential means a drastic shift in the way computing is done and devices are built. Accessible NVM means faster databases for transaction processing and much lower latency to access stored data. It also grants the capability to perform real-time analytics on memory-resident storage, the ability to replicate data directly into persistence using fast RDMA network protocols, significantly faster restart times for increased reliability and up-time, and additional redundancy in the event of a power failure.

1 Research Summary

My research investigates both theoretical and practical infrastructure to integrate nonvolatile memory into traditional parallel programming. My work lies at the intersection of systems software and distributed systems theory — I build practical systems with formal guarantees. My work is driven by the question, “What infrastructure is required to give application programmers safe, fast, and reliable access to NVM storage?” With the commercial release of the first NVM hardware, this question is increasingly urgent. In order for programmers to realize the promise of fast, durable storage on NVM, they need a comprehensive set of tools to manage its contents and reason about its guarantees. I have worked to answer this need, publishing in top-tier venues including full papers at MICRO, ASPLOS, DISC×2, PACT, PPoPP, TOPC, and ICPP, and short or workshop papers at SPAA×2, PODC, TRANSACT×2, and NVMW.

My thesis work has focused on crash consistency for data structures in nonvolatile memory. Unfortunately, processor caches and registers are likely to remain volatile, at least in the near term, so what remains in nonvolatile memory after a crash is only that data that has been written back. Even more unfortunately, hardware capacity and associativity constraints require that caches be permitted to perform their writes-back in essentially arbitrary order. When this order differs from the happens-before order of the running program, the values that have “leaked back” to NVM at any particular time may be mutually inconsistent.

One line of my research has investigated systems to ensure failure-atomic updates to NVM, thereby guaranteeing that the persistent state is always consistent in the wake of a crash. These failure atomicity systems, resembling traditional software transactional memory, provide the programmer with an easy and fast method for avoiding inconsistency [4, 6, 11]. The first, JUSTDO logging [4], leverages locking to define
quiescent and consistent states: when all locks are released the program is consistent. JUSTDO logging responds to failure by resuming the execution of any thread within a critical section and running it until it releases all locks. This strategy avoids complex run-time checking of dependencies between updates. Follow-on work, iDO logging [11], integrates this idea into a compiler pass, and leverages idempotence to reduce the amount of logging done by the run time. I have also investigated “breaking” transactions to allow concurrent data structure operations within the failure atomic update, thereby avoiding transactional overheads within the data structure [6]. Another line of my research has investigated stand-alone concurrent data structures for NVM that ensure their contents are consistent [12], and proposes a new, generic, design philosophy for building data structures in nonvolatile memory. I have further developed a theoretical infrastructure proving these data structures correct and demonstrated that all linearizable nonblocking algorithms can be trivially transformed into correct, nonblocking algorithms for NVM, even with a very relaxed instruction set [7, 8].

My interest in nonvolatile memory is derived from my studies in traditional shared memory synchronization. One thread of my work in shared memory programming has developed new nonblocking data structures: dual containers [9], double-ended queues [1], and trees [2]. Each not only demonstrates performance gains over state-of-the-art comparable algorithms, but also include a proof of correctness via linearizability. I have also investigated nonblocking techniques for memory reclamation in data structures [13]. Other work has investigated the impact of lock elision on real-world programs [5] and proposed the use of hardware transactional memory as a prefetching tool [10].

Since completing my doctorate, I have had two postdocs. The first, with Prof. Alexey Gotsman at IMDEA Software Institute in Madrid, filled the gap between my December Ph.D. defense and the start of the next school year. In this period, I expanded my systems expertise by learning about both RDMA network protocols and consensus algorithms. The resulting work, Acuerdo, is a novel RDMA-optimized protocol similar to Zookeeper’s ZAB but with stronger guarantees on leader election [3]. The work is being prepared for submission to USENIX ATC in January with a follow-on theoretical paper exploring some impossibility results surrounding leader election to PODC in February. During my time at IMDEA I also wrote a grant proposal in collaboration with colleagues at Tel Aviv University. This grant has been conditionally approved to fund approximately $30,000 worth of equipment from Mellanox Technologies. On completion of my position in Madrid, I relocated to my current position as a postdoc in Prof. Steve Swanson’s Nonvolatile Systems lab at UC San Diego. Since arriving, I have collaborated and advised on several burgeoning projects within the lab, including a submission to FAST’19 on distributed file systems for nonvolatile memory [14]. Other projects I am involved in include benchmarks, compilers, and failure atomicity libraries for nonvolatile memory.

2 Near-Term Research Goals

I believe that NVM infrastructure is an important and under-developed area. The engineering effort required to give the application programmer safe, fast, and reliable access to NVM storage is only beginning. Critical open topics in NVM include memory safety, OS abstractions, crash consistency, and integration into both languages and compilers. I believe that these near-term topics are addressable within first few years as an assistant professor, though in-depth exploration will require more effort, time, and collaboration.

Memory Safety  The most immediate concern in achieving usable byte-addressable NVM is memory safety. Failure atomicity systems protect durable data from power outages and other fail-stop errors using transaction style semantics, but leave this same data vulnerable to memory corruption from software errors. If we expect the world to use NVM for durable storage, we must be able to protect persistent data from stray writes issued by buggy client applications, while still allowing safe access to this same data by (presumably) a trusted user-level library. Since NVM necessitates hardware changes to ensure consistency, what additional hardware primitives should we use to protect persistent memory regions? Or can we creatively leverage existing ISAs to provide high-performance and safe access to these regions?

Crash Consistency  Ensuring consistent NVM state in the wake of a crash important for all persistent applications. What is the correct semantic model for expressing consistent updates, and is it applicable to
all languages? Given that fast persistence will be expected, how can we reduce the latency and increase bandwidth into persistence?

**OS Abstractions** Exposing NVM memory regions as an OS abstraction requires the operating system to explicitly manage the region and provide some support to the user. How do we allocate within the region, and should the operating system manage garbage collection after a crash? How do we map the region into the process address space, and what do we do about address clashes? How can processes share a region and must they map it to the same address? How can we share persistent data between machines and languages, and how do we ensure that persistent data is interpreted by the correct source program? If NVM is widely available, it makes sense to move the file system from disk to NVM. What optimizations exist when designing a file system directly for NVM? Can we avoid syscall overhead in this case? How do we protect metadata sufficiently, and what consistency guarantees can we provide for data?

**Language and Compiler Integration** Compiler and language awareness of the benefits and pitfalls of NVM is also in its infancy. Some semantic models exist for the ordering and timing of writes-back from caches to NVM, but no in-depth theoretical study exists. What characterizes these “persistency models,” and are some insufficiently strong? Are some persistency models incompatible with certain consistency models? On a more practical level, languages currently interact with NVM via libraries, but very little has been done to explore language extensions. What language-level constructs can be used to distinguish between persistent data stored in NVM and transient data stored in DRAM? How do we represent persistent and dynamically-typed objects? How can we safely access the persistent object across source code updates? Compilers should also provide persistence support. Can compilers reduce the cost of persistent updates by eliminating redundancy or by using compression? Given that NVM writes are expected to be slower than reads, what compiler optimizations are worth re-investigating for persistent memory?

### 3 Long-Term Research Goals

The arrival of nonvolatile memory puts computing in an interesting place. For decades, programmers have been thinking about and designing for volatility, while persistence has been a separate, distinct concern hidden behind a clear interface. With the arrival of nonvolatile memory, volatility is no longer the default. Such a drastic shift has deep implications for how we design and program computers, despite the fact that we have conditioned ourselves to consider volatile data to be the normal state of the world. In the future, I expect that the deep integration of nonvolatile memory will change the way we think not only about persistence, but also, volatility.

**Volatile by Design** In a world where all memory is persistent, special care must be taken to ensure volatility. When all memory is persistent, how do we ensure that data is truly erased, and erased rapidly? How does nonvolatility impact privacy and security, and how are we sure that our precautions are sufficient? What is a proof of privacy when all data is persistent? Since volatility is no longer the default, we might expect the programmer to explicitly request it for sensitive data. How can we design secure and private interfaces to volatility, what guarantees can they give, and how performant must they be?

**Immortal Programs** Because computational state is volatile and can be lost, applications are necessarily written to be restarted. However, when nonvolatile memory is commonplace, programs (or at least parts of them) are effectively immortal; even on reboot, they may retain their state in persistent memory. These “immortal programs” will continue computation where they left off, even when suffering from an unexpected fail-stop error. I am interested in exploring both practical and theoretical problems related to immortal programs that have lost the ability to be restarted. For instance, how do immortal programs handle errors or corruption of their data? What is a “correct” immortal program? How can we upgrade immortal programs online without loss of service? Can immortal programs even be “offline”? How can we design, build, test, and verify such programs?
I am also interested in reenvisioning how data is organized in nonvolatile memory. Currently, without byte-addressable persistence, data is serialized, that is, translated, from its in-memory representation into a format amenable to storage, be it in a file system or database. This translation was originally a byproduct of the performance properties of disk and now, with the advent of NVM, it is not always needed. I am interested in exploring how to store unstructured and semi-structured data in-memory without serialization or marshalling. If we no longer need to serialize data in order to store it persistently, how should we change how we organize data? How universal can data storage become without an explicit, synchronous, translation layer? Can we mmap the same file into our virtual address space across languages? Across machines? Across versions of the same program? Across different programs? And if our data is, to some extent, universal, what can we share between programs? Can we share a heap across languages if we represent data in a universal manner? And, since memory-resident data is now persistent, how should we change our execution model, our stack, or our heap, in order to leverage universal data?

References


