Research Statement

Joseph Izraelevitz
University of Rochester
Rochester, NY, USA
jhi1@cs.rochester.edu

With the end of DRAM scaling in sight, new memory types are competing to replace 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 over-head 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 we can snapshot long-running high performance computing applications nearly continuously—failures could cost mere milliseconds of compute time, instead of minutes. Accessible NVM also means we can build an Internet of Things constructed out of intermittently powered devices. Less transformative, but no less impressive, accessible NVM means faster databases for transaction processing, and an additional layer of redundancy in the face of a power failure.

1 Research Summary

My doctoral research investigates both theoretical and practical infrastructure to integrate nonvolatile memory into traditional parallel programming. My work is driven by the question, “What infrastructure is required to give application programmers safe, fine-grained, 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. During my dissertation, I have worked to answer this need, publishing in top-tier venues including full papers at 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 which 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 [3, 5, 10]. The first, JUSTDO logging [3], 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. Ongoing follow-on work, iDO logging \cite{10}, 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 \cite{5}. Another line of my research has investigated stand-alone concurrent data structures for NVM that ensure their contents are consistent \cite{11}, 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 \cite{6,7}.

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 \cite{8}, double-ended queues \cite{1}, and trees \cite{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 \cite{12}. Other work has investigated the impact of lock elision on real-world programs \cite{4} and proposed the use of hardware transactional memory as a prefetching tool \cite{9}.

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, fine-grained, fast, and reliable access to NVM storage is only beginning. Critical open topics in NVM include memory safety, language and compiler integration, OS abstractions, and, of course, crash consistency. I am excited to bring my knowledge of the problems in NVM to a new department where I can also investigate how the technology might impact those systems best known to my colleagues.

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? In my opinion this problem remains a critical gap in the literature, and is an essential problem to be solved if NVM is to become an acceptable alternative to file I/O.

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 and compiler-optimized NVM updates. What language-level constructs can be used to distinguish between persistent data stored in NVM and transient data stored in DRAM? Can compilers reduce the cost of persistent updates by eliminating redundancy or by using compression? Can compilers automatically generate code to restart the process after a crash? Given that NVM writes are expected to be somewhat slower than reads, what compiler optimizations are worth reinvestigating? Or, since some varieties of NVM tend to have lower write endurance than DRAM, can we use compilers to spread writes across the heap to minimize wear-out? Answering even some of these questions would significantly lower the programming effort needed to begin using NVM, and would allow the technology to be used by all classes of programmers.
**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 send persistent regions from one machine to another and ensure compatibility? However an operating system decides to answer these questions, the solutions will have major ramifications on the design and capabilities of user-level software.

**Crash Consistency** Ensuring consistent NVM state in the wake of a crash is still important, and the development of failure atomicity systems will continue. It is likely worth drawing inspiration from other fields. In particular, techniques from traditional transactional memory and databases seem relevant as they support similar operations. Leveraging newer hardware primitives, such as hardware transactional memory, might also provide performance improvements. I am particularly interested in applying nonblocking techniques to these problems, since nonblocking algorithms are already fail-stop resilient.

**Internet of Things** Looking farther afield, I expect to leverage my expertise in failure atomicity to explore intermittently powered devices either in the mobile space or as part of the Internet of Things. Devices that harvest energy from their surroundings must be prepared to lose power at any moment, but should be able to make progress regardless. Optimizing energy-aware and failure atomicity systems for these devices is likely to be a critical step in the development of the Internet of Things.

**Shared Memory Synchronization** I further expect to continue researching traditional parallel programming constructs. Despite two decades of research, concurrent nonblocking data structures get faster every year. Hardware transactional memory has yet to reach its full potential, and software transactional memory is still improving. Some problems in this area include improving the performance of concurrent tree and heap data structures, faster memory reclamation for concurrent programs, and the application of hardware transactional memory to these problems.

3 Long-Term Research Goals

In the long term, I intend to continue to contribute to the shared memory community and to sustain my interest in the run-time performance of parallel programs, while also maintaining my contacts in industry. I expect that with the anticipated demise of Moore’s Law, parallel programming will grow in importance as programmers seek other methods of performance improvement. As such, I intend to grow my expertise into the wider parallel computing world, including database systems, distributed computing, and high performance computing. Developing in-house parallel programming expertise in the computer science department—and publicizing it—maximizes the opportunities for inter-disciplinary collaborations with the hard sciences where custom software may be the key enabler in achieving results.

**References**


