Statement of Research Interests

Aasheesh Kolli - akolli@umich.edu

I am a computer systems researcher with particular interests in multi-core systems and memory systems. My past research primarily investigates designing computing systems for the new storage paradigm presented by *Persistent Memory* technologies (also known as Non-Volatile RAM, e.g., Intel's 3D XPoint). These technologies offer the exciting possibility of disk-like durability with the performance of main-memory, aiding the development of high-performance, recoverable software. For example, imagine storage software that provides the safety of PostgreSQL *and* the performance of Redis. However, designing persistent memory systems requires novel computer architectures and programming interfaces, and is the focus of my dissertation.

Moving forward, I will continue to design computer architectures and programming interfaces to fully exploit the benefits of emerging technology trends. I am particularly interested in investigating computing systems in light of ultra-low latency networking technologies and the increasing reliance on hardware specialization to deliver future performance and energy efficiency expectations. These trends will radically transform how we design and program future computing systems. I look forward to tackling the challenge of developing efficient hardware implementations for such systems while still providing programmers with simple interfaces to hardware. In this statement, I briefly summarize the contributions of my prior research and then detail my plans for future research.

1 Dissertation Research - Persistent Memory Systems

Advances in memory technologies will soon result in computing systems with byte-addressable, persistent main-memories, a significant departure from traditional systems with byte-addressable, volatile main-memory and separate block-addressable, persistent storage. These persistent memory systems promise drastic performance improvements by enabling programmers to maintain recoverable data structures (e.g., key-value stores) in main-memory and directly access them via regular processor loads and stores without having to rely on performance-sapping software intermediaries like the operating and file systems.

However, ensuring correct recovery requires programmers to *control the order in which stores reach persistent memory*, a task complicated by the presence of programmer-transparent, *volatile* hardware structures (e.g., caches, memory controller) that actively reorder and coalesce requests to main-memory for performance. Moreover, precluding such optimizations hinders system performance. Today's systems do not provide efficient mechanisms (if any) that programmers can rely on to maintain recoverable data structures in main-memory. Hardware vendors are soon expected to provide *memory persistency models* [PCW14] (precisely defined here [KPS+15]) that will allow programmers to express the desired order of stores to persistent memory. The hardware is responsible for ensuring that stores become durable in persistent memory (an act I refer to as a *persist*) in the specified order.

Just as memory consistency models are critical for writing shared-memory parallel programs, memory persistency models are critical for ensuring the recoverability of data structures in persistent memory. There are many parallels between the problems solved by memory consistency research and the problems currently being faced by memory persistency. Designing high-performance hardware implementations for various persistency models, tailoring software for these models, and simplifying persistent memory programming are all important challenges that have to be addressed before persistent memory systems can be widely adopted. So, with the benefit of being able to draw upon years of memory consistency research, I tackle the following question in my dissertation research: **how must hardware implementations, programming interfaces, and software be re-designed to maximize the benefits of persistent memory systems?**

The contributions of my dissertation research can be categorized as follows:

Persistency model implementations: While memory persistency models are required to ensure recovery correctness, their semantics limit allowable hardware optimizations and hence system performance. For example, the persistency model proposed by Intel via their instruction set architecture (ISA) extensions for persistent memory programming [Int14] conflates ordering persists with their completion. That is, to order

two persists, a processor must issue the first persist and then stall execution until the persist operation completes before issuing the second persist. This synchronous approach to ordering persists results in frequent processor stalls and poor performance, exacerbated by the high write latencies of persistent memory technologies. I identify one of the proposed instructions, PCOMMIT, to be particularly troublesome [KRD+16]. In fact, in September 2016, Intel announced the *deprecation* of the PCOMMIT instruction [Int16].

Instead of a synchronous approach, I argue for persistency models with semantics that allow ordering of persists to be decoupled from execution at the processor. Furthermore, I propose an implementation strategy for such persistency models, *delegated persist ordering*, wherein persist dependencies are communicated directly to the persistent memory controller, decoupling execution at the processor from ordering of persists [KRD+16]. Delegated persist ordering effectively trades-off "freshness" of state in persistent memory for higher performance without sacrificing any guarantees on persist order. This work was a *best paper award nominee* at MICRO'16.

Language interfaces for persistent memory programming: While various persistency models have been proposed, all of them have been specified at the ISA level. That is, programmers must reason about recovery correctness at the abstraction of assembly instructions, an approach that is error prone, places an unreasonable burden on the programmer, and restricts software portability. I argue for *language-level persistency models*, as an extension to language-level memory models, that provide mechanisms to specify the semantics of accesses to persistent memory as an integral part of the programming language [KGD+]. A language-level persistency model provides a single, ISA-agnostic framework for reasoning about persistency and can enable portability of recoverable software across language implementations (compiler, runtime, ISA, and hardware).

Based on a survey of proposed ISA persistency models and usage patterns exhibited by recovery software, I propose a concrete language-level persistency model, *acquire-release persistency* [KGD+], to extend the C++11 memory model. Ideally, the language and ISA persistency models work in concert to enforce only the minimal guarantees required for correct recovery. However, I find that state-of-the-art ISA persistency models are not able to exploit the persist concurrency exposed by acquire-release persistency. I propose minor modifications to the C++11 language, compiler, ISA, and hardware to be able to exploit all of the available persist concurrency, improving application performance.

Rethinking software for persistent memory systems: While efficient persistency models and implementations are necessary to deliver high-performance, recoverable software must be optimized for the underlying persistent memory system hardware to extract all of the available performance. In this context, I investigated the design of transaction libraries for different kinds of persistent memory systems. Similar to how software primitives like mutexes simplify reasoning about concurrency control, transactions that abstract away the details of memory persistency models will be widely used by persistent memory programmers. My work [KPS+16] takes a fundamental approach to developing high-performance transactions under different persistency models as opposed to focusing on a particular model.

I show that the straightforward way to implement transactions for persistent memories, which I call *synchronous commit transactions*, enforces many unnecessary constraints on the order of persists. Such constraints degrade performance as they preclude optimizations that batch and reorder persists. I propose a novel transaction design, *deferred commit transactions*, that leverages the insight that stores from different transactions may be persisted in an order more relaxed than in which they were executed without sacrificing any recovery guarantees. Deferred commit transactions deliver higher performance by exposing additional persist concurrency under four different persistency models.

Apart from the persistency model, software must also be optimized for specific characteristics of the persistent memory system so that available performance is not squandered. For example, persistent memory systems with *persistent caches* (vs. volatile caches) have been proposed in recent literature that drastically reduce the time taken to complete a persist from hundreds of nanoseconds to less than ten nanoseconds. Such systems call for novel transaction designs. As an intern at HPE Labs, I collaborated with Izraelevitz [IKK16] to show that conventional logging mechanisms (UNDO/REDO logging), commonly employed by transaction libraries to ensure failure-atomicity, are unnecessarily complex and memory intensive for systems with persistent caches. We present the design and implementation of *JUSTDO-logging*, a new failure-atomicity mechanism that greatly reduces the memory footprint of logs, simplifies log management, and enables fast

parallel recovery following failure. The key insight behind JUSTDO-logging is that for systems with persistent caches, it is more efficient to effectively "checkpoint" the state of thread execution and resume execution from the checkpoint during recovery than it is to maintain logs to rollback partially completed updates.

From my dissertation research, it is clear to me that designing efficient persistent memory systems requires changes to computer architectures, programming interfaces, and storage software while being cognizant of the interactions between these different layers of system design. While such holistic design approaches are good in general, they are especially important when designing systems around emerging technologies, as a well-designed system could be the difference between wide-spread adoption or extinction of the technology. As a professor, I will apply these lessons to incorporate exciting new technologies into computing systems.

2 Future Research Directions

Computing systems will soon undergo radical transformations due to two emerging technology trends: (1) specialized network interfaces offering extremely fast remote memory access and, (2) architects relying on hardware specialization to meet both performance and energy efficiency expectations. To address the challenges, and exploit the opportunities presented by these technology trends, I plan to pursue two directions of research:

Accelerating distributed applications: Improving optical network technologies and tighter integration of network interfaces and memory hierarchies have resulted in drastic improvements in remote memory access latencies. Trends indicate that accessing close-by remote memory is expected to be a *sub* μs operation. Similar to how the emergence of persistent memories drastically reduced the access latency to durable media and made recoverable software a performance bottleneck, these extremely fast networks will make application software, rather than data communication, the performance bottleneck for distributed applications. In light of this trend, novel architectures to accelerate distributed application software will be essential to meet future performance targets. Identifying common programming patterns within distributed applications will allow the development of architectural solutions to accelerate them. Furthermore, of these common patterns, complex ones like asynchronous network programming, can be abstracted away from programmers via domain-specific languages or libraries, improving performance as well as reducing programming burden. Given the maturation of these networking technologies, I expect this direction of my research to be a short-term focus. As a concrete first step in this line of research, considering my prior work on persistent memory systems, I will target applications that use remote memory as persistent storage.

Data-sensitive applications typically mirror their data sets on remote nodes for both failure-tolerant data persistence and load balancing. I have seen the prevalence of such applications first hand during my internship at Google last summer. In today's systems, programmers have to reason separately about the management of a local copy of data and the serialization and data transfer required to ensure a consistent remote copy. However, given the properties of remote direct memory access protocols (e.g., losslessness), it is feasible to provide a single interface that programmers can use to manage both local and remote data copies while offloading the frequent serialization and data transfer management tasks to runtime/hardware. I will develop such simple programming interfaces and efficient architectural implementations for these tasks.

Architecting extreme heterogenous systems: In an era where both performance and energy efficiency are first-class design constraints, system designers are increasingly turning to hardware specializations like domain-specific acceleration and near-data processing. For example, I was involved in two projects that showcased the advantages of specialized hardware: (1) I helped design a text processing hardware accelerator [GKC+16] that achieved orders of magnitude better performance than traditional software solutions and (2) I designed an instruction prefetcher optimized for server applications [KSW13] that achieved near state-of-the-art performance with significant reductions in hardware and energy overheads.

While a large number of research efforts focus on specializations catered for individual applications, little attention has been paid to the problem of designing future systems that could potentially have hundreds of heterogenous compute units, running hundreds of applications simultaneously. I envision future applications using many different highly specialized compute units and programmers having to marshall application data

across different compute units during different phases of the applications. Ensuring such systems deliver on their promises requires us to address two challenges: (1) ensure efficient data communication between different compute units and (2) provide useful programming interfaces. My prior work on persistent memory systems required a deep understanding of application requirements, programming interfaces, hardware implementations, and their interactions and has prepared me to pursue a holistic approach to designing these extreme heterogenous systems.

Below are several questions I'd like to investigate.

- *Programming interfaces:* How will programmers write applications for these systems? Due to the complexity of these systems, a generic programming language is unlikely to be very useful for programmers. Domain-specific languages or libraries will have to be developed to reduce programmer burden while still allowing efficient hardware implementations.
- *Data communication:* How will different compute units communicate with each other? It is likely that compute units will be grouped together in coherence domains, compute units within the same domain can rely on shared coherent memory for communication, while compute units across domains will have to rely on the programmer (with compiler/run time support) to handle explicit communication. Such memory system organization will require new memory consistency models and efficient hardware implementations to minimize data communication overheads.
- *Resource contention:* With many applications (or threads) likely sharing the system at the same time, they will likely contend for compute units. How will the runtime handle resource conflicts? For example, virtualization of compute units will be necessary along with robust architectural support to reduce performance overheads.

Streamlining software, programming interfaces, and hardware implementations will be necessary to deliver future performance and energy efficiency targets. I am looking forward to pursuing the inter-disciplinary collaborative efforts that will be required to achieve these goals.

References

[GKC+16] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. Hare: Hardware accelerator for regular expressions. In *Proceedings of the 49th International Symposium on Microarchitecture*, 2016.

[IKK16] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[Int14] Intel. Intel architecture instruction set extensions programming reference (319433-022), 2014. `https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf`.

[Int16] Intel. Deprecating the pcommit instruction, 2016. `https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction`.

[KGD+] Aasheesh Kolli, Vaibhav Gogte, Stephan Diestelhorst, Ali Saidi, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level persistency. *Currently under submission.*

[KPS+15] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. Persistency programming 101, 2015. `http://nvmw.ucsd.edu/2015/assets/abstracts/33`.

[KPS+16] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[KRD+16] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *Proceedings of the 49th International Symposium on Microarchitecture*, 2016.

[KSW13] Aasheesh Kolli, Ali Saidi, and Thomas F. Wenisch. Rdip: Return-address-stack directed instruction prefetching. In *Proceedings of the 46th International Symposium on Microarchitecture*, 2013.

[PCW14] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceedings of the 41st International Symposium on Computer Architecture*, 2014.