Theme Article

Language Support for Memory Persistency

Aasheesh Kolli
Pennsylvania State University and VMware Research

Vaibhav Gogte
University of Michigan

Ali Saidi
Amazon Web Services

Stephan Diestelhorst
ARM Research

William Wang
University of Michigan

Peter M. Chen
ARM Research

Satish Narayanasamy
University of Michigan

Thomas F. Wenisch
University of Michigan

***Abstract*—Memory persistency models enable maintaining recoverable data structures in persistent memories and prior work has proposed ISA-level persistency models. In addition to these models, we argue for extending language-level memory models to provide persistence semantics. We present a taxonomy of guarantees a language-level persistency model could provide and characterize their programmability and performance.**

Persistent memories (PMs), such as Intel's upcoming 3D XPoint memory,[1] offer the durability of disk, better density than DRAM, and DRAM-like performance. These properties have spawned myriad efforts to adopt PM in computer systems. A particularly disruptive potential PM use case is to host in-memory recoverable data structures. PMs blur the traditional divide between a byte-addressable, volatile main memory, and a block-addressable, persistent storage. This memory allows programmers to directly manipulate recoverable data structures using processor loads and stores, rather than relying on performance-sapping software intermediaries like the operating system and file system.[2]

Ensuring the recoverability of data structures requires programmers to be able to control the order stores reach PM. With out-of-order processing and write-back caching, stores may reach PM out of order, compromising data structure recoverability. Existing systems do not provide efficient mechanisms to enforce the order in which stores are written back. Recent work has proposed *persistency models* to provide programmers an interface to control the order persistent stores write

Digital Object Identifier 10.1109/MM.2019.2910821
Date of publication 16 April 2019; date of current version 8 May 2019.

0272-1732 © 2019 IEEE
Published by the IEEE Computer Society

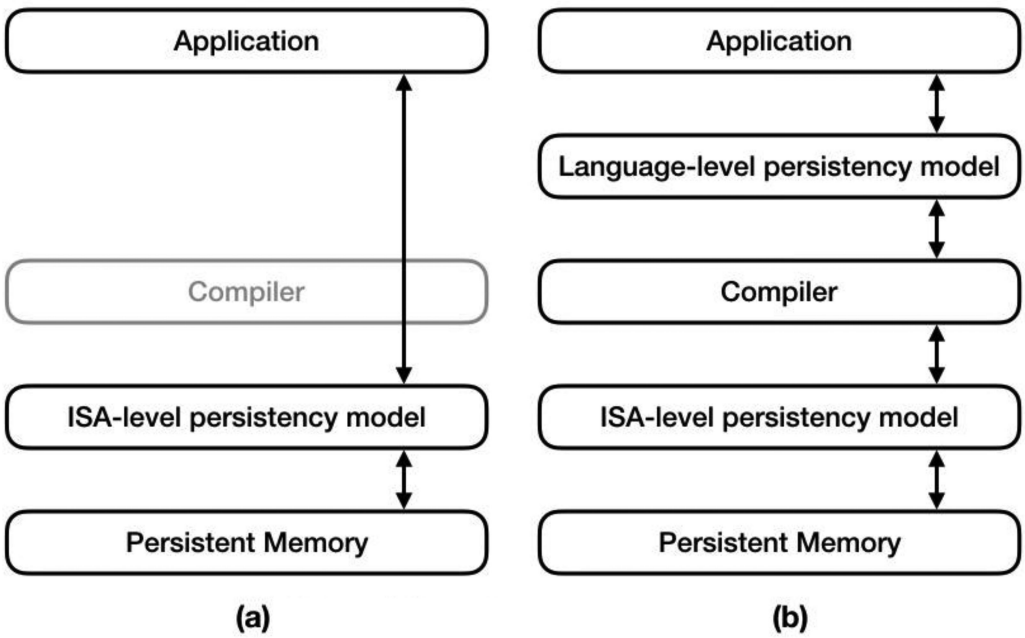

Figure 1. Prior works have proposed persistency models at the ISA-level, leading to programming complexity and portability issues. We instead argue for an additional language-level persistency model working in concert with the ISA-level persistency model to reduce programmer burden and improve portability. (a) State of the art. (b) Our goal.

to PM.[3] Like prior work, we refer to the act of writing a store durably in PM as a *persist*.

While various persistency models have been proposed, all of them have been specified at the instruction set architecture (ISA) level.[3,4] That is, programmers must reason about recovery correctness at the abstraction of assembly instructions, an approach that is error prone and imposes an unreasonable programmer burden. The programmer must invoke ISA-specific mechanisms to ensure persist order and often must reason carefully about compiler optimizations that may affect the relevant code. Since the ISA mechanisms differ in sometimes subtle ways, it is hard to write portable recoverable programs.

Our recent work[5, 6] has argued for a *language-level persistency model* that encapsulates the ISA-level persistency model to provide mechanisms to specify the semantics of PM accesses (including with respect to program failures) as an integral part of the programming language. Just as language-level memory, consistency models enable precise specification of memory access semantics from concurrent threads, a language-level persistency model provides a single, ISA-agnostic framework for reasoning about persistency and can enable portable recoverable software across language implementations (compiler, runtime, ISA, and hardware). Figure 1 contrasts prior approaches to memory persistency with our approach.

This work explores how a language-level persistency model might specify semantics for PM state after a failure, providing a taxonomy of guarantees that a language-level persistency model might provide. Stronger guarantees (e.g., failure-atomicity of critical sections) make writing recoverable software easier but impose substantial requirements on the implementation which entail performance penalties. Weaker guarantees complicate reasoning about recovery, but provide greater implementation freedom and performance. Weaker guarantees relax atomicity of critical sections and instead provide only ordering guarantees for individual persists. Note that ordering individual persists allows synthesis of coarser granularities of atomicity via logging.

Reasoning about recovery can be greatly simplified by providing failure atomicity over sets of PM updates. Failure atomicity assures that either all or none of a set's updates are visible after failure, reducing the state space recovery code might observe. Atomicity (beyond a PM access

granularity) can be achieved via numerous hardware or software mechanisms (e.g., logging[2]). Prior work, Atlas,[7] argues to simplify recovery design by guaranteeing failure-atomicity of entire outermost critical sections. However, we show the ATLAS approach incurs significant performance penalty and provides unclear semantics for PM updates outside critical sections.

Instead, we propose persistency semantics that provide precise failure-atomicity at the granularity of synchronization-free regions (SFRs)—thread regions delimited by synchronization operations or system calls. Under failure-atomic SFRs, the state observed by recovery always conforms to the program state at a frontier of past synchronization operations on each thread. In a well-formed program, SFRs must be data-race free. This property allows us to extend the sequential consistency guaranteed for data-race-free programs to recovery code.

We propose a concrete model, acquire–release persistency (ARP), to extend the C++11 memory model, and describe the compiler, ISA, and hardware features needed to ensure that only the ordering guarantees requested by the programmer at the language level are enforced at runtime.

Programmers would clearly prefer the coarsest granularity of failure-atomicity that a language can provide, as it simplifies PM programming. However, indications from hardware vendors (e.g., Intel[8]) are that future processors will only guarantee atomicity for individual persists. Because the compiler or runtime logging mechanisms required to ensure failure-atomicity must be general, they cannot take advantage of data-structure-specific optimizations (e.g., wait-free recoverable data structures,[9] static transactions[10]). Performance-centric approaches are designed with the rationale that the language should provide the most fundamental atomicity guarantee (individual persists); software solutions (e.g., in expert-crafted libraries) for larger atomic regions can be layered on top to reduce programmer burden.

Even when providing atomicity guarantees at the granularity of individual persists, different models may vary in the kinds of ordering guarantees they provide. Weaker models place fewer restrictions on persist order and can potentially deliver better performance. We propose a concrete model, acquire–release persistency (ARP), to extend the C++11 memory model, and describe the compiler, ISA, and hardware features needed to ensure that only the ordering guarantees requested by the programmer at the language level are enforced at runtime.

TAXONOMY OF LANGUAGE-LEVEL PERSISTENCY MODELS

We explore different factors that should be taken into account while designing a language-level persistency model to develop a taxonomy of guarantees that a language-level persistency model may provide.

Our taxonomy considers two dimensions of persistency model guarantees: 1) the granularity of failure atomicity—the stores within a failure atomic unit that appear to persist atomically (outermost critical sections, SFRs, or individual stores) and 2) the persist order of these failure-atomic units (sequential consistency or epoch ordering). Stronger guarantees (e.g., failure-atomicity of entire critical sections and sequential consistency) make writing recoverable software easier but impose substantial requirements on the implementation, entailing performance penalties. Weaker guarantees complicate reasoning about recovery, but provide greater implementation freedom and performance. Next, we describe interesting points in the design space laid out by our taxonomy.

Sequentially Consistent Failure-Atomic Outer Critical Sections

Under this approach, all the persists from an outer critical section (from first lock acquire until no locks are held) are guaranteed by the language implementation to be failure atomic [see Figure 2(a)]. Furthermore, different outer critical sections must persist in sequentially consistent order.

The idea of sequentially consistent failure-atomic outer critical sections was first explored by Chakrabarti *et al.*[7] The central appeal of this guarantee is that, by ensuring failure-atomicity of

entire critical sections, the PM state post-recovery always reflects a state that would have arisen in fault-free execution and when no thread holds a lock. Thus, no recovery code is needed, the programmer is assured that her data structures are always in a consistent state postrecovery.

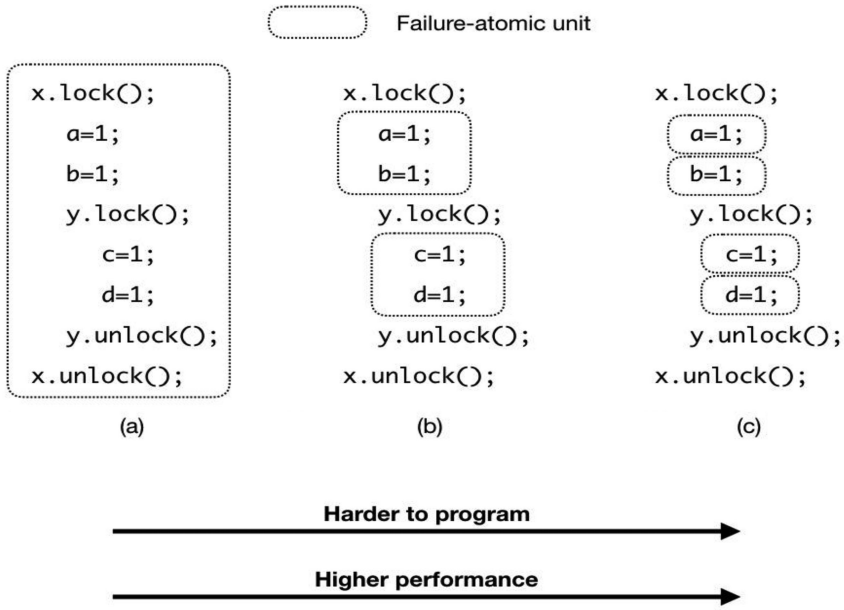

Figure 2. This figure shows the different granularities of failure atomicity that a language could provide: (a) outer critical sections; (b) Synchronization-free-region and (c) individual persists. Coarser granularities of failure-atomicity make it easier to program, however, they impose higher performance overheads.

Sequentially Consistent Failure-Atomic SFRs

Whereas atomicity of outer critical sections is appealing, it requires the implementation to support atomicity guarantees spanning multiple threads, since critical sections can become coupled through shared memory communication. Instead, failure atomicity for SFRs guarantees atomicity only for code regions between synchronization accesses (or system calls) on a single thread [see Figure 2(b)], ensuring such regions persist in a sequentially consistent order. Because C++ requires SFRs to be data race free, they may not become coupled across threads, greatly simplifying implementation.

For transaction-based programs or programs without overlapping critical sections, SFRs and critical sections are the same. However, for programs which have overlapping critical sections [see Figure 2(b)], a critical section may span SFRs. For such programs, partially completed critical sections may be visible postrecovery. While developing recovery software, the programmer must consider this possibility. If failure-atomicity of outer critical sections is desired, the programmer must add roll-back mechanisms for partially completed critical sections.

Sequentially Consistent Persists

Further relaxing atomicity guarantees leads to a model where only individual stores persist atomically and in a sequentially consistent order [see Figure 2(c)]. In such a model, the programmer must implement failure-atomicity mechanisms in software if larger atomicity granularities are required. The programmer can rely on the sequentially consistent order of persists while implementing the logging mechanisms.

Epoch-Ordered Persists

One can construct even more relaxed persistency models, which admit even greater concurrency and implementation freedom, by relaxing the program ordering requirement of persists. In such models, individual stores persist atomically, but *need not* persist in a sequentially consistent order. Special sequence point annotations are used by a programmer to break a thread into epochs; persists across epochs are ordered, but may be reordered within epochs. Persists on different threads are still governed by synchronization order.

Models that relax sequential consistency for persists may astonish programmers: the programmer must issue explicit sequence points when ordering guarantees are required, complicating the implementation of recoverable data structures. Yet, relaxed atomics have long been a part of the C++ programming language; epoch-ordered persists provide a similar promise of

performance at the cost of much higher programmer burden.

Having outlined a taxonomy of guarantees that a language-level persistency model may provide, we next summarize our two works[5,6] that compare and contrast various ways to build programmability centric and performance-centric language-level persistency models.

DESIGNING PROGRAMMABILITY CENTRIC LANGUAGE-LEVEL PERSISTENCY

In our most recent work,[6] we consider how to develop programmability centric language-level persistency models. The main factor that determines ease of programming is the granularity of failure-atomicity guaranteed by the language. Generally, coarser granularity leads to easier programming as it reduces the number of possible PM states a programmer must reason about. When the persistency model guarantees atomicity only for individual persists, recovery may observe PM state that could never arise in fault-free execution. Prior work[7] proposes failure atomicity at the granularity of outermost critical sections. However, such an approach provides unclear semantics for PM updates outside critical sections, does not generalize to other synchronization constructs (e.g., condition variables), and requires high-overhead cycle detection among critical sections on different threads to identify sets that must be jointly failure-atomic.

Persistency for Synchronization Free Regions

We argue that failure-atomic SFRs strike a compelling balance between programmability and performance. Under failure-atomic SFRs, the state observed by recovery will always conform to the program state at a frontier of past synchronization operations on each thread. In a well-formed program, SFRs must be data-race free. This property allows us to extend the SC-for-DRF guarantee to recovery code, while avoiding the disadvantages of critical-section-grain atomicity.

We extend the C++ memory model with persistency semantics for multithreaded programs. The C++ memory model uses interthread and intrathread happens-before ordering prescribed by acquire and release synchronization operations in multithreaded applications to order memory accesses. We extend these guarantees to ensure that the memory accesses within SFRs become persistent in an order consistent with the constraints on when they may become visible. We investigate two designs based on undo-logging that provide failure-atomicity of SFRs and vary in simplicity and performance.

Coupled-SFR design: In this design, the visibility of the program state in volatile caches is coupled with its persistent state in PM. The in-place PM mutations are flushed at the end of each SFR and the undo log is immediately committed. Before the SFR's terminal synchronization, a memory barrier is emitted to ensure that all PM mutations persist before any writes in the next SFR. Thus, the committed state lags the frontier of execution by at most a single SFR; recovery rolls back to its start, minimizing the state loss upon failure.

The central advantage of Coupled-SFR is that each thread must track only log entries for stores within its still-incomplete SFR, and does not interact with any other thread. The thread-private nature of our commit stands in stark contrast to ATLAS, which must perform a dependence analysis and cycle-detection across all threads' logs to identify log entries that must commit atomically. Because accesses within an SFR are data-race free, there can be no dependencies among accesses in uncommitted SFRs; all interthread dependencies must be ordered by the synchronization commencing the SFR, and, hence, may depend only on committed state. The PM state after recovery is easy to interpret, as it conforms to the state at the latest synchronization on each thread.

However, the downside of Coupled-SFR is that the execution stalls at the end of the SFR until all PM writes are flushed and the log is committed, potentially exposing much of PM persist latency on the critical path.

Decoupled-SFR design: Alternatively, we can decouple the visibility of updates (as governed by cache coherence and the C++ memory model) from the frontier of persistent state; that is, we can allow persistent state to lag execution—an approach we call Decoupled-SFR. To ensure that persistent state does not fall too far behind (which risks losing forward progress in the event of failure), we periodically invoke a flush-and-commit mechanism, much like garbage

collection in managed languages. This mechanism flushes in-place updates and commits logs. Nevertheless, Decoupled-SFR must still assure that recovery will roll PM state back to the prior state that conforms to a frontier of synchronization operations on each thread.

Recoverability requires that logs are pruned—committing the updates in the corresponding SFR—in the same order as the SFRs execute, else the state after recovery will not correspond to a state consistent with fault-free execution. As such, our logging mechanism must log the happens-before ordering relations between SFRs (as governed by the C++ memory model) and commit according to this order. We record happens-before by: 1) adding acquire/release annotations to the per-thread logs, 2) maintaining per-thread logs in program order (thereby capturing intra-thread ordering), and 3) tracking order across threads by maintaining a monotonic sequence number across release/acquire pairs.

Each program thread has an accompanying pruner thread that flushes mutations and commits the log on its behalf. The pruner threads are invoked periodically to commit and recycle log space. In case of failure, undo logs are processed in reverse order to recover program state to the start of committed SFRs.

Logging: We implement a compiler pass in LLVM v3.6.0, which instruments synchronization operations and PM accesses with undologging operations. Our compiler pass emits code to construct an undo log entry in PM for synchronization operations and PM store operations within the SFR. The log entry records the old value of PM locations, before any mutation. The log entry is then persisted by explicitly flushing it from volatile caches to the PM. Next, our compiler pass emits an ISA-level memory ordering barrier (to order the flush with subsequent writes) and the store operation that updates the persistent data structure in place. These updates are then explicitly flushed and persisted, and the corresponding undo log entries are committed. Our two atomicity schemes described above enable persistency semantics for SFRs, but differ in when and how they perform these latter two steps.

Evaluation: We study a suite of seven write-intensive multithreaded microbenchmarks and benchmarks, used in prior studies.[4] Owing to the simple logging, Coupled-SFR results in an average performance improvement of 63.2% over the ATLAS design. Decoupled-SFR enables light-weight recording of SFR order and performs flush and commit operations off the critical execution path. As a result, Decoupled-SFR leads to a further performance improvement of 50.1% over Coupled-SFR.

DESIGNING PERFORMANCE-CENTRIC LANGUAGE-LEVEL PERSISTENCY

In earlier work,[5] we delved deeper into building performance-centric language-level persistency. Whereas guaranteeing failure-atomicity only for individual persists is a first step in building high-performance persistency implementations, we also show how relaxing ordering requirements between individual persists also plays a major role in improving performance. To this end, we proposed ARP, a language-level persistency model that extends the C++11 memory model and introduced hardware extensions to minimize unnecessary ordering restrictions that arise from translating the language-level persistency model to the ISA-level persist ordering mechanisms.

We describe two major sources of unnecessary persist ordering restrictions and describe how ARP mitigates them.

Fence directionality: Whereas ARP (and the C++11 memory model) is based on release consistency, the ISA-level persistency model on the ARM systems we target is based on the more conservative ARMv7 consistency model.[4] Hence, its ISA-level model is oblivious to unidirectional acquire and release operations that are available in C++11 and ISAs based on release consistency (e.g., ARMv8).

ARP allows programmers to use unidirectional synchronization operations (*acq* and *rel*) to order memory accesses. Both *acq* and *rel* operations are usually used to ensure memory accesses within a critical section do not "leak out," however, they allow memory accesses from outside the critical section to "leak into" the critical section. But as ARMv7 does not distinguish between an *acq* and a *rel*, compilers are forced to use a full fence for both, which precludes memory access reordering in both directions, leading to unnecessary constraints as shown in Figure 3(a). A thread performs stores to three persistent addresses, A, B, and C. The stores to A and B are separated by an *acq*,

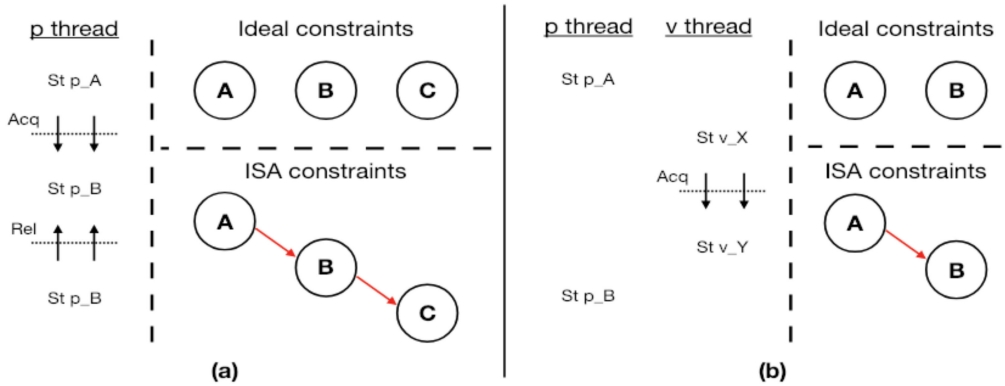

Figure 3. (a) Unnecessary constraints enforced due to fence directionality obliviousness. (b) Unnecessary constraints enforced due to lack of support to identify volatile fences.

while stores to B and C are separated by a *rel*. As per the semantics of ARP, all three are considered concurrent and may execute and persist in any order. However, replacing the *acq* and *rel* with a full fence requires that persists to A, B, and C are serialized. Such overconstraints on persist order arise whenever the ISA-level persistency model is stricter than the language-level model.

Conflating synchronization with recoverability: The second set of unnecessary constraints are caused by the lack of mechanisms to allow programmers to distinguish constraints required for concurrency control, but not for recoverability. Consider the case in Figure 3(b), where two unrelated threads (*p_thread* and *v_thread*) issue memory accesses. ISA-level persistency serializes persists and fences from all cores into the write queue at the PM controller.[4] So, if the *acq* from v_thread happens to arrive at the PM controller between the two persists requests from p_thread, then the PM controller will place them in different epochs—an overconstraint.

Ideally, we would like the hardware to enforce only constraints required for recovery. We observe that programmers can identify *acq* and *rel* memory operations that have no persist semantics (i.e., they are required only for concurrency control). For example, some threads may never issue any persist operations and communicate only among themselves.[11] With minor extensions to the C++11 memory model, programmers can annotate *acq* and *rel* that do not have persist semantics as nonpersistent or "volatile" and the hardware will not enforce associated persist constraints.

Discussion: Mitigating the two sources of unnecessary persist constraints allows more persists to join each epoch at the PM controller. Larger epochs in turn provide greater flexibility to schedule and batch persist operations, improving persist concurrency, leading to substantial performance gains.

Evaluation: We study a suite of seven write-intensive multithreaded microbenchmarks and benchmarks, used in prior studies.[10] Overall, ARP improves microbenchmark execution time by 32.4% as compared to SCP and 21.2% as compared to the baseline ISA-level persistency model. For the macrobenchmarks, ARP improves execution time of the three benchmarks by 24.3% and 15.5% over SCP and ISA-level persistency model, respectively.

IMPACT

Advent of Byte-Addressable Persistent Memory is Now

We anticipate that systems incorporating high-performance byte-addressable PMs will become widely available within a few years. Intel and Micron have already made announcements regarding their 3D XPoint memory technology and competing offerings will likely soon follow. Indeed, as cost and yield improve, such memories may become ubiquitous across the computing spectrum: they may become the preferred storage for small devices in the Internet of Things and similarly may become critical to performance and recoverability in cloud systems. Programs written for such systems will have to manage the transfer of data between volatile and persistent domains (for example, a volatile

cache hierarchy and persistent main memory). The correctness of recoverable data structures relies on guaranteeing the order of durable writes, the primary focus of our work.

Memory Persistency Must Be Guaranteed End-to-End, From the Language to the Hardware

Both industry[8] and academia[3,4] have proposed candidate persistency models that rely on lower level hardware ISA primitives to prescribe order over PM updates. These approaches require that programmers must invoke ISA-specific mechanisms (via library calls or inline assembly) to ensure the desired order of PM updates and that they reason carefully about compiler optimizations that may affect the relevant code. Our community's experience with ISA-level memory consistency models makes us well aware of the severe portability and programmability challenges that arise with this approach.

> We anticipate that systems incorporating high-performance byte-addressable PMs will become widely available within a few years. Intel and Micron have already made announcements regarding their 3D XPoint memory technology and competing offerings will likely soon follow.

This work argues for a language-level persistency model that provides mechanisms to specify the semantics of PM accesses (including with respect to program failures) as an integral part of the programming language, just as language-level memory consistency models enable precise specification of the semantics of memory accesses from concurrent threads. A language-level persistency model enables portable recoverable software across different ISAs, hardware, runtimes, and compilers and also provides a single, ISA-agnostic framework for reasoning about recoverability. Furthermore, a language-level model allows software and hardware implementations to be developed and tested independently, significantly simplifying design-test-debug cycles.

Need to Drive Industry to Provide Better Programming Models

Whereas the first commercial PM chips are slated for release within a year, surprisingly little consensus can be found between industry and academia on the best programming interfaces for PMs. To further muddy the waters, Intel recently announced that it will deprecate the newly proposed pcommit instruction before it is even released in any commercial product.[12] Exploratory studies, such as our work, that seek to understand the implications of the semantics of various memory persistency models on system architecture, programmability, and performance have the potential to guide the direction taken by industry. For example, the RISC-V memory model is only now being formalized—there is an urgent opportunity for academics to ensure that we get memory persistency for RISC-V right.

Follow-on Research

Our language-level persistency model can influence a wide spectrum of future research. First, it can enable portable recoverable software. Programmers can build the software systems by relying on persistency guarantees as part of the high-level language. Second, our persistency model can be used to design recovery mechanisms for general programming systems. Our compiler implementation emits logging for PM updates and synchronization operations to enable SFR failure-atomicity transparently. Finally, failure-atomic SFRs do not expose non-SC state to recovery. Novel compiler or hardware solutions can be built to leverage reordering or coalescing opportunities for PM accesses within SFRs.

REFERENCES

1. Intel and Micron, "Intel and Micron Produce Breakthrough Memory Technology," 2015. [Online]. Available: http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology
2. J. Coburn *et al.*, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proc. 16th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2011, pp. 105–118.
3. S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proc. 41st Annu. Int. Symp. Comput. Archit.*, 2014, pp. 265–276.
4. A. Kolli *et al.*, "Delegated persist ordering," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, Art. No. 58 .

5. A. Kolli *et al.*, "Language-level persistency," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 481–493.
6. V. Gogte *et al.*, "Persistency for synchronization-free regions," in *Proc. 39th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2018, pp. 46–61.
7. D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2014, pp. 433–452.
8. Intel 2019, "Intel 64 and IA-32 Architectures Software Developer Manuals." 2019. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm
9. F. Nawab, D. Chakrabarti, T. Kelly, and C. B. Morey III, "Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience," Hewlett-Packard Tech. Rep. HPL-2014-70, 2014.
10. A. Kolli, S. Pelley, A. Saidi, M. C. Peter, and F. W. Thomas, "High-performance transactions for persistent memories," in *Proc. 21st Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2016, pp. 399–411.
11. C. Blundell, M. M. K. Martin, and T. F. Wenisch, "InvisiFence: Performance-transparent memory ordering in conventional multiprocessors," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 233–244.
12. Intel, "Deprecating the PCOMMIT instruction." 2016. [Online]. Available: https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction

Aasheesh Kolli is an assistant professor of computer science and engineering at the Pennsylvania State University, State College, and an affiliated researcher with VMware Research.

Vaibhav Gogte is currently a PhD in computer science and engineering at the University of Michigan.

Ali Saidi is a principal engineer at Amazon Web Services, Seattle. This work was done while he was ARM Research, Cambridge, U.K.

Stephan Diestelhorst is a principal research engineer and Skill Group Lead at Arm Research, Cambridge, U.K.

William Wang is the Arthur F. Thurnau Professor of computer science and engineering at the University of Michigan.

Peter M. Chen is a Staff Research Engineer at Arm Research, Cambridge, U.K.

Satish Narayanasamy is an associate professor of computer science and engineering at the University of Michigan.

Thomas F. Wenisch is an associate professor of computer science and engineering at the University of Michigan.