Failure-atomic Synchronization-free Regions

V. Gogte*, S. Diestelhorst†, W. Wang†, S. Narayanasamy*, P. M. Chen*, T. F. Wenisch*
*University of Michigan {vgogte,nsatish,pmchen,twenisch}@umich.edu
†ARM {stephan.diestelhorst,william.wang}@arm.com

I. Introduction

Emerging persistent memory (PM) technologies, such as Intel and Micron's 3D XPoint, aim to combine the byte-addressability of DRAM with the durability of storage. The promise of PM is to enable data structures that provide the convenience and performance of in-place load-store manipulation, and yet persist across failures, such as power interruptions and OS or program crashes. Following such a crash, volatile program state (DRAM, program counters, registers, etc.) are lost, but PM state is preserved. A *recovery* process can then examine the PM state, reconstruct required volatile state, and resume program execution.

Reasoning about the correctness of recovery code requires precise semantics for the allowable PM state after a failure. Specifying such semantics is complicated by the desire to support concurrent PM accesses from multiple threads and optimizations that reorder or coalesce accesses. The state observed at recovery can be greatly simplified by providing *failure atomicity* of sets of PM updates. Failure atomicity assures that either all or none of the updates in a set are visible after failure, reducing the state space recovery code might observe.

Recent work has proposed memory *persistency models* to provide programmers with such semantics [1], [2]. Like previous works, we refer to the act of writing the value of a store operation to PM as a *persist*. Similar to memory consistency models, which govern the visibility of writes to shared memory, persistency models govern the order of persists to PM. Most of these persistency models [1], [2] have been specified at the abstraction level of the hardware instruction set architecture (ISA). Such ISA-level persistency models do not specify semantics for higher-level languages, where compiler optimizations may also reorder or elide PM reads and writes.

We first discuss existing proposals that add persistency semantics to the language memory model. In particular, ATLAS [3] and acquire-release persistency (ARP) [4], [5] extend the C++ memory model with persistency semantics. The two proposals differ in the granularity of failure atomicity they guarantee and rely on different synchronization primitives to ensure correct persist ordering in PM.

ATLAS: ATLAS [3] provides persistency semantics for lock-based multi-threaded C++ programs. It guarantees failure atomicity at the granularity of an outermost critical section, as shown in Figure 1(a), where a critical section is the code bounded by *lock* and *unlock* synchronization primitives. Failure-atomicity of critical sections guarantees that recovery may only observe sequentially consistent PM state. However, we argue that this approach suffers from three key deficiencies: (1) its semantics are unclear for PM updates outside critical sections, (2) it does not generalize to other synchronization constructs (e.g., condition variables), and (3) it requires expensive cycle detection among critical sections on different threads to identify sets that must be jointly failure-atomic, which leads to high overhead.

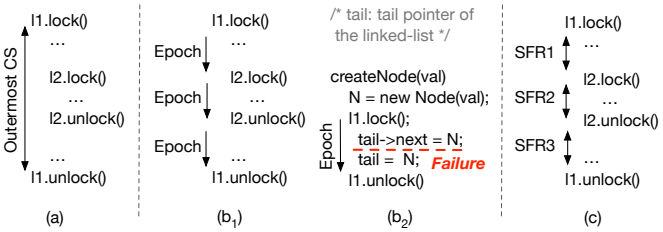

Fig. 1: **Failure-atomicity in ATLAS, ARP and our proposal.**

ARP: ARP [4] proposes extending the memory models of high-level languages, like C++11 and Java, with persistency semantics. ARP ascribes persists to ordered *epochs* using intra- and inter-thread ordering constraints prescribed via *acquire* and *release* synchronization operations. As shown in Figure 1(b_1), ARP may re-order persists within epochs but disallows reordering across epochs. Although ARP bounds the latest point at which a PM store may persist, it does not preclude PM stores from persisting *early*, ahead of preceding accesses in memory (visibility) order. Figure 1(b_2) shows example code to append a new node to a persistent linked-list. As ARP does not constrain the durability of the two updates before the completion of the epoch, the update to `tail` may become durable earlier than the update to `tail->next`. In case of a failure, an incomplete update to the tail pointer will result in an inconsistent linked-list. As such, the set of states a recovery program might observe includes many states that (1) do not correspond to SC program executions, and (2) could never arise in a fault-free execution, posing a daunting challenge for recovery design.

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 [6], as shown in Figure 1(c). In the absence of SFR atomicity, recovery may observe PM state that could never arise in fault-free execution (similar to ARP). 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. We argue that failure-atomic SFRs strike a compelling balance between programmability and performance. 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 and avoid the unclear semantics of ARP. Moreover, our approach avoids the limitations of ATLAS-like approaches.

II. Failure-atomicity of SFRs

We extend the C++ memory model with durability semantics for multi-threaded programs. The C++ memory model uses inter-thread and intra-thread *happens-before* ordering prescribed by acquire and release synchronization operations in multi-threaded 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.

Logging: We implement a compiler pass in LLVM v3.6.0, which instruments synchronization operations and PM accesses with undo-logging operations. For the synchronization operation that begins an SFR, and every PM store operation within the SFR, our compiler pass emits code to construct an undo log entry in PM. The log entry records the values PM locations had at the start of the SFR, 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 differ in when and how they perform these latter two steps.

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 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 dependency analysis and cycle-detection across all threads' logs to identify log entries that must commit atomically. Because accesses within SFRs must be data-race free, there can be no dependences between accesses in uncommitted SFRs; all inter-thread 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: Instead, we 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++11 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. Like garbage-collection, 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.

III. Evaluation

We implement a compiler pass that can emit code for both our logging approaches in LLVM v3.6.0. We study a suite of seven write-intensive multi-threaded micro-benchmarks and benchmarks, used in prior studies [4]. Due to space limitation, we list only the average improvement obtained in our designs.

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.

We next evaluate the cost of the background activity required by both ATLAS and Decoupled-SFR to commit their logs. Although the pruner/helper threads do not delay execution on the critical path, they nonetheless consume additional CPU resources. Overall, we find that Coupled-SFR and Decoupled-SFR have 72.1% lower and 33.2% lower CPU-cost per throughput than ATLAS.

References

[1] Intel, "Instruction set extensions programming reference." https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.

[2] ARM, "Armv8-a architecture evolution," 2016. https://community.arm.com/groups/processors/blog/2016/01/05/armv8-a-architecture-evolution.

[3] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *OOPSLA '14*.

[4] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *ISCA '17*.

[5] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Tarp: Translating acquire-release persistency," 2017. http://nvmw.eng.ucsd.edu/2017/assets/abstracts/1.

[6] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," PLDI, 2018.