iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory

Qingrui Liu*, Joseph Izraelevitz†, Se Kwon Lee‡, Michael L. Scott†, Sam H. Noh‡, and Changhee Jung*

*Virginia Tech
Blacksburg, VA, USA
{lqingrui,chjung}@vt.edu
https://orcid.org/0000-0002-6422-6549

†University of Rochester
Rochester, NY, USA
{jhi1,scott}@cs.rochester.edu
https://orcid.org/0000-0001-8652-7644

‡UNIST
Ulsan, Korea
{sekwonlee,samhnoh}@unist.ac.kr

***Abstract*—This paper presents *iDO*, a compiler-directed approach to failure atomicity with nonvolatile memory. Unlike most prior work, which instruments each store of persistent data for redo or undo logging, the iDO compiler identifies *idempotent* instruction sequences, whose re-execution is guaranteed to be side-effect-free, thereby eliminating the need to log every persistent store. Using an extension of prior work on JUSTDO logging, the compiler then arranges, during recovery from failure, to back up each thread to the beginning of the current idempotent region and re-execute to the end of the current failure-atomic section. This extension transforms JUSTDO logging from a technique of value only on hypothetical future machines with nonvolatile caches into a technique that also significantly outperforms state-of-the art lock-based persistence mechanisms on current hardware during normal execution, while preserving very fast recovery times.**

I. Introduction

With the emergence of fast, byte-addressable nonvolatile memory such as commercial 3D XPoint, ReRAM, and STT-MRAM, we can now conceive of systems in which main memory, accessed with ordinary loads and stores, is simply "always available," and need not be flushed to the file system to survive a crash. The obvious use case of such a technology, and the one we focus on here, is to allow programmers to store heap objects persistently in memory, bypassing the expensive serialization of those objects onto traditional storage devices. This use case is widely applicable: we envision applications using persistent heap objects as an alternative to disk-resident local databases or as a way (e.g., on energy-harvesting devices with frequent crashes) to enable fast restarts.

Unfortunately, from the perspective of crash recovery, nonvolatile main memory is compromised by the fact that traditional caches can write data back to memory in arbitrary order, leading to inconsistent values in the wake of a crash [1], [2]. A failure in the middle of a linked-list insertion, for example, may lead to a post-crash dangling reference if the `next` pointer of the predecessor node is written back to memory before the inserted node itself. Moreover even in the absence of reordering, failure during an operation that is meant to be atomic can leave the contents of memory in an inconsistent intermediate state, rendering it unusable.

In order to avoid such errors and ensure post-crash consistency of persistent data, researchers have developed failure-atomicity systems that allow programmers to delineate failure-atomic operations on the persistent data—typically in the form of transactions [2]–[6] or *failure-atomic sections* (FASEs) protected by outermost locks [7]–[9]. Given knowledge of where operations start and end, the failure-atomicity system can ensure, via logging or some other approach, that all operations within the code region happen atomically with respect to failure and maintain the consistency of the persistent data. Transactions have potential advantages with respect to ease of programming and (potentially) performance during normal operation (at least in comparison to coarse-grain locking), but can be difficult to retrofit into existing code, due to idioms like hand-over-hand locking and limitations on the use of condition synchronization or irreversible operations. Transactions also tend to perform more poorly than well tuned fine-grain locking. Our own work is based on locking.

The principal challenge of FASE-based recovery, compared to transactional recovery, stems from the lack of isolation in critical sections. In a lock-based program, FASEs that involve more than one lock, even when data-race free, may be able to see each other's changes while both are still in progress; in fact, correct execution may *depend* on them seeing each other's changes (e.g., for condition synchronization). UNDO logging, which makes updates "in place," avoids hiding the FASE's updates, but must address the possibility that a completed FASE will depend on values written in some other FASE that was interrupted by a crash. Systems like Atlas [7], which incorporate UNDO logging, must therefore track cross-FASE dependences and be prepared to roll back even completed FASEs during post-crash recovery. A similar problem arises with REDO logging for FASEs, as in the NVThreads system [8]: if an incomplete FASE releases a lock, it must share its locally buffered changes with any thread that subsequently acquires the lock; it must also track the dependence. If the earlier FASE fails, the dependent must fail as well. This implies that when a thread reaches the end of a dependent FASE during normal execution, it must wait until the earlier FASE has completed before replaying its own log and proceeding.

To simplify the management of logs for FASE-based persistence, and, in particular, to avoid the need for dependence

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00029

tracking, Izraelevitz et al. introduced the notion of JUSTDO logging [9]. Rather than rolling back a FASE during recovery (as one would with UNDO logging) or replaying a FASE's writes (as one would with REDO logging), the JUSTDO system logs enough information to *resume* a FASE during recovery and execute it to completion ("recovery via resumption"). Immediately prior to each store instruction in a FASE, the JUSTDO system logs (in persistent memory) the program counter, the to-be-updated address, and the value to be written. During recovery, the system uses the code of the crashed program to complete the remainder of each interrupted FASE, beginning with the most recent log entry. Future program runs can then be assured that the recovered data is consistent, much as conventional programs can be ensured of the integrity of data in a journaled file system.

The problem with JUSTDO logging is its requirement that the log be written *and made persistent* before the related store—a requirement that is very expensive to fulfill on conventional machines with volatile caches. Current ISAs provide limited support for ordering write-back from cache to persistent memory, and these limitations seem likely to continue into the foreseeable future [10]. To ensure that writes reach memory in a particular order, the program must typically employ a sequence of instructions referred to as a *persist fence*. On an Intel x86, the sequence is ⟨`sfence, clwb, clwb, ..., sfence`⟩. This sequence initiates and waits for the write-back of a set of cache lines, ensuring that they will be persistent before any future writes. Unfortunately, the wait incurs the cost of round-trip communication with the memory controller.

Given the cost of persistence ordering, JUSTDO assumes—unlike Atlas and NVThreads—that it will run on a machine in which caches are persistent, due either to implementation in STT-MRAM or to capacitor-driven flushing in the event of power failure. On a more conventional machine with persist fences, JUSTDO is 2–3× slower than Atlas [8], [9].

JUSTDO logging also imposes a restricted programming model within FASEs, with no use of volatile data and no caching of values in registers [8], [9]. These restrictions would seem to preclude the use of SIMD instructions, widely regarded as essential to data-intensive applications [11], [12] and in-memory databases [13], [14].

The key contribution of our work is to demonstrate that recovery via resumption can in fact be made efficient on conventional machines, with volatile caches and expensive persist fences. The key is to arrange for each log operation (and in particular each persist fence) to cover multiple store instructions of the original application. We achieve this coverage via compiler-based identification of *idempotent* instruction sequences. Because an idempotent region of code can safely be re-executed an arbitrary number of times without changing its output, the recovery procedure in the wake of a crash can resume execution at the beginning of the current region, eliminating the need to log each individual store instruction of the original program.

This paper presents iDO, a practical compiler-directed failure-atomicity system. Like JUSTDO logging, iDO supports fine-grained concurrency through lock-based FASEs, and avoids the need to track dependences by executing forward to the end of each FASE during post-crash recovery. Unlike JUSTDO, iDO allows the use of registers in FASEs, and persists its stores at coarser granularity. While these advantages should allow iDO to outperform prior systems on hypothetical machines with nonvolatile caches, experiments confirm that it can also outperform them—by substantial margins—on conventional machines with volatile caches.

Instead of logging information at every store instruction, iDO logs (and persists) a slightly larger amount of program state (registers, live stack variables, and the program counter) at the beginning of every idempotent code region within the overall FASE. In practice, idempotent sequences tend to be significantly longer than the span between consecutive stores—tens of instructions in our benchmarks; hundreds or even thousands of instructions in larger applications [15]. As iDO is implemented in the LLVM tool chain [16], our implementation is also able to implement a variety of important optimizations, logging significantly less information—and packing it into fewer cache lines—than one might naively expect. We also introduce a new implementation for FASE-boundary locks that requires only a single memory fence, rather than the two employed in JUSTDO.

Our principal contributions can be summarized as follows:

- We introduce iDO logging, a lightweight strategy that leverages idempotence to ensure both the atomicity of FASEs and the consistency of persistent memory in the wake of a system crash. Rather than log individual memory stores, iDO logs a lightweight summary of live program state at the beginning of each idempotent region.
- We compare the performance of iDO to that of several existing systems, demonstrating up to an order of magnitude improvement over Atlas in run-time speed, and better scaling than transactional systems like Mnemosyne [2].
- We verify that recovery time in iDO is also very fast—one to two orders of magnitude faster than Atlas in long-running programs.

Our paper is organized as follows. Section II gives additional background on failure-atomicity systems and idempotence. Section III discusses the high-level design of iDO logging; Section IV delves into system details. Performance results are presented in Section V. We discuss related work in Section VI and conclude in Section VII.

II. Background

A. System Model

iDO assumes a near-term hybrid architecture (Fig. 1), in which some of main memory has been replaced with nonvolatile memory, but the rest of main memory, the caches, and the processor registers remain volatile. Data in the core and caches are therefore transient and will be lost on system failure.[1] Portions of main memory are likely to continue to be

[1]In general, we refer to physical memory as *volatile* or *nonvolatile*, and to program memory (data) as *transient* or *persistent*.

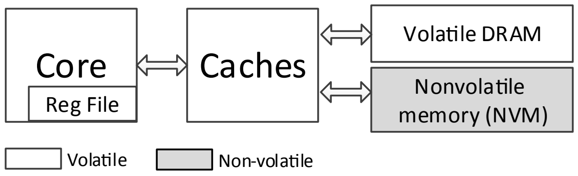
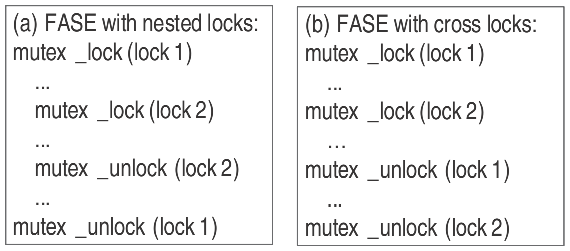


Fig. 1: Hybrid architecture model in which a portion of memory is nonvolatile, but the core, caches, and DRAM are volatile.



```
(a) FASE with nested locks:
mutex _lock (lock 1)
   ...
   mutex _lock (lock 2)
   ...
   mutex _unlock (lock 2)
   ...
mutex _unlock (lock 1)
```

```
(b) FASE with cross locks:
mutex _lock (lock 1)
   ...
mutex _lock (lock 2)
   ...
mutex _unlock (lock 1)
   ...
mutex _unlock (lock 2)
```

Fig. 2: FASEs with different interleaved lock patterns.

implemented with DRAM in the short term, due to density, cost, and/or endurance issues with some NVM technologies. As in other recent work, we assume that read and write latencies of NVM are similar to those of DRAM [7], [8] and that writes are atomic at 8-byte granularity [17]. Our failure model encompasses (only) fail-stop errors that arise outside the running application. These include kernel panics, power outages, and various kinds of hardware failure.

B. Programming Model

As noted in Section I, iDO employs a programming model based on lock-delineated failure-atomic sections (FASEs), primarily because of their ubiquity in existing code. A FASE is defined as a maximal-length region of code beginning with a lock (mutex) acquire operation and ending with a lock release, in which at least one lock is always held [7]–[9], [18]. Note that the outermost lock and unlock pairs do not necessarily need to be the same. Figure 2 shows examples of FASEs with two possible interleaved lock patterns. The left-hand side shows nested locks; the right has a cross-locking (hand-over-hand locking) pattern.

For each FASE, iDO provides a variant of the classic ACID transaction properties [19]:

Atomicity means that updates to persistent data performed in a FASE complete in an "all or nothing" manner. A FASE that is interrupted by a crash is completed as part of the recovery procedure.

Consistency is typically defined by program semantics. We assume that every FASE transitions memory from a state in which all program invariants hold to another in which they still hold. By completing an interrupted FASE during recovery, we preserve consistency even in the presence of failures by pushing persistent data to a state in which no locks are held.

Isolation requires that a transaction never see other threads' changes during its execution and, likewise, that its own changes be invisible until commit time. In a FASE-based programming model, isolation is a consequence of mutual exclusion, but only for properly-nested FASEs (Fig. 2(a)) with the same outermost locks.

Durability (persistence) means that the results of FASEs survive crashes. More specifically, if the results of one FASE are visible to a second, and the second survives a crash, the first survives as well.

For single-threaded programs or code that accesses privatized variables, iDO also supports programmer-delineated *durable code regions*. These code regions are defined by the programmer to be failure atomic but lack the isolation guarantees of lock-delineated FASEs. From here on, we use the term "FASE" to denote both lock- and programmer-delineated failure atomic code regions.

FASE-based failure-atomicity systems based on UNDO and REDO logging typically prohibit thread communication outside of critical sections [7], [8]. This prohibition prevents a happens-before dependence between critical sections from being created without the system's knowledge. An advantage we gain from recovery via resumption is that thread communication outside of critical sections can occur without compromising correctness.

Despite its strengths, recovery via resumption has some pitfalls. In order for recovery to succeed, the failure atomic code region must be allowed to be run to completion. For this reason, resumption is infeasible for speculative transactions, which must be able to abort and roll back all updates made so far when a conflict is detected late in their execution (possibly during recovery). Consequently, iDO logging is vulnerable to software bugs within FASEs—on recovery, reexecuting the buggy code will not restore consistency.

In this work, we assume a programming model that expects all writes to persistent locations to occur within lock- or programmer-delineated FASEs. This model ensures that program state after a crash corresponds to a cut across the store order aligned with either a lock release or the end of a durable code region in every thread. Like other FASE-based systems [7]–[9], we disallow not only conventional data races within FASEs but also races on atomic variables, to avoid the possibility that the order in which the race is resolved may be inverted at recovery time. Provided they do not cause a race, persistent reads are allowed outside FASEs.

C. Idempotence

An *idempotent region* is a single-entry, (possibly) multiple-exit subgraph of the control flow graph of the program. In keeping with standard terminology, we use the term *inputs* to refer to variables that are *live-in* to a region and used there. That is, an input has a definition that reaches the region entry and a use of that definition within the region. Similarly, we use the term *outputs* to refer to variables that are updated in a region and *live-out* at the end of the region. That is, an output of a region is a variable written in the region that serves as an input to some following region. We also use the term *antidependence* to refer to a write-after-read dependence, in which a variable is used and subsequently overwritten. A region is *idempotent* if and only if it would generate the same output if control were to jump back to the region entry from any execution point within the region (assuming isolation from other threads). To enable such a jump-back, the region inputs must not be overwritten—i.e., there must be no antidependence on the inputs—during the execution of the region.

Idempotent regions have been used for a variety of purposes in the literature, including recovery from exceptions, failed speculation, and various kinds of hardware faults [15], [20]–[22]. For any of these purposes—and for iDO—inputs must be preserved to enable re-execution.

III. iDO Failure Atomicity System

Unlike UNDO or REDO logging, iDO logging provides failure atomicity via *resumption* and requires no log for individual memory stores. Once a thread enters a FASE, iDO must ensure that it completes the FASE, even in the presence of failures. At the beginning of each idempotent code region in the body of a FASE, all inputs to the region are known to have been logged in persistent memory. Since the region is idempotent, the thread never overwrites the region's inputs before the next log event. Consequently, if a crash interrupts the execution of the idempotent region, iDO can re-execute the idempotent region from the beginning using the persistent inputs.

More precisely, at the end of each (compiler-delineated) idempotent region, iDO logs the *output* data of the region—the data that were modified by the region and that serve as input to some following region. In data flow terms, we define the output of region r as the live-out data defined in the region—the values that are written and *downward-exposed* [23]:

$$OutputSet_r = Def_r \bigcap LiveOut_r \tag{1}$$

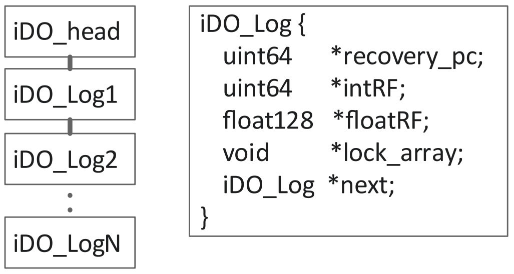
where Def_r is the set of values defined in r and $LiveOut_r$ is the set of live-out values of r. The iDO compiler persists the values in $OutputSet_r$ at the end of region r.

Successful recovery requires additional care. In particular, if we re-execute a FASE using a *recovery thread*, this thread must hold the same locks as the original crashed thread. Tracking this information is the responsibility of the thread's local *lock_array* (Sec. III-A), which is updated at every lock acquisition and release.

The following subsections consider the structure of the iDO log, the implementation of FASE-boundary locks, and the recovery procedure. Additional compiler details—and in particular, the steps required to identify FASEs and transform the FASEs into a series of idempotent regions—are deferred to Section IV.

A. The iDO Log

For each thread, the iDO runtime creates a structure called the `iDO_Log`. We manage the per-thread iDO logs using a global linked list whose `iDO_head` is placed in a persistent memory location to be found by the recovery procedure (Sec. IV-C). Log structures are added to the list at thread creation. As shown in Figure 3, each `iDO_log` structure comprises four key fields. The `recovery_pc` field points to the initial instruction of the current idempotent region. The `intRF` and `floatRF` fields hold live-out register values; each register has a fixed location in its array. The `lock_array` field holds *indirect lock addresses* for the mutexes owned by the thread—more on this in Section III-B.



```
iDO_Log {
  uint64     *recovery_pc;
  uint64     *intRF;
  float128   *floatRF;
  void       *lock_array;
  iDO_Log    *next;
}
```

Fig. 3: iDO log structure and management: the number of iDO logs matches the number of threads created.

Here then is the series of steps required, within a FASE, to complete the execution of idempotent region r and begin the execution of region s:

1) Issue write-back instructions for all output registers of r (saving them to `intRF` and `floatRF`) and for all output values in the stack. Together, these comprise $OutputSet_r$. Note that live-out values that were not written in r are already sure to have persisted; no additional action is required for them.
2) Update `recovery_pc` to point to the beginning of s. Once this step is finished, s can be re-executed to recover from failures that occur during its execution.
3) Execute the code of s, generating the values in $OutputSet_s$. These values will be persisted at the end of s—i.e., at the boundary between s and its own successor t, as described in step 1. Note that by definition an idempotent region will never overwrite its own input.

iDO continues in this fashion until the end of the FASE. To enforce the order of these steps, the iDO compiler inserts a single *persist fence* between the first step and the second, and again between the second and the third. After completing the steps, a thread moves on to the next idempotent region. Output registers are written to `intRF` and `floatRF` immediately after their final modification in the current region. Writes-back of output values in the stack are likewise initiated immediately after the final write of the current region, though we do not wait for completion until the fence between steps 1 and 2. In the absence of precise pointer analysis, we cannot always identify the final writes to variables accessed via pointers; these are therefore tracked at run time and then written back at the end of each idempotent region.

Recovery in the wake of a crash is described in Section III-C.

B. Indirect Locking

Our discussion thus far has talked mostly about idempotent regions. To obtain failure atomicity for entire FASEs, we must introduce lock recovery. In particular, in the wake of a crash, we must reassign locks that were held at the time of the crash to the correct recovery threads, ensure that those locks are held before re-executing the interrupted FASEs, and guarantee that no other locks are accidentally left locked from the previous program execution (else deadlock might occur). Previous approaches [3], [9] persist each mutex. Then, during recovery, they unlock each held mutex to release it from a failed thread before assigning it to a recovery thread. In JUSTDO logging, this task requires updating a *lock intention log* and a *lock ownership log* before and after the lock operation. Each lock or unlock operation then entails two persist fence sequences—a significant expense.

iDO introduces a novel approach that avoids the need to make mutexes persistent. The key insight is that all mutexes must be unlocked after a system failure, so their values are not really needed. We can therefore minimize persistence overhead by introducing an *indirect lock holder* for each lock. The lock holder resides in persistent memory and holds the (immutable) address of the (transient) lock. During normal execution, immediately after acquiring a lock, a thread records the address of the lock holder in one of the `lock_array` entries of the `iDO_Log`. It also sets a bit in an initial index slot in the array to indicate which array slots are live. Immediately before releasing a lock, the thread clears both the `lock_array` entry and the bit. Finally, the iDO compiler inserts an idempotent region boundary immediately after each lock acquire and before each lock release.

Upon system failure, each transient mutex will be lost. The recovery procedure, however, will allocate a new transient lock for every indirect lock holder, and arrange for each recovery thread to acquire the (new) locks identified by lock holders in its `lock_array`. An interesting side effect of this scheme (also present in JUSTDO logging), is that if one thread acquires a lock and, before recording the indirect lock holder, the system crashes, another thread may steal the lock in recovery! This effect turns out to be harmless: the region boundaries after lock acquire ensure that the robbed thread failed to execute any instructions under the lock.

C. iDO Recovery

Building on the preceding subsections, we can now summarize the entire recovery procedure:

1) On process restart, iDO detects the crash and retrieves the `iDO_Log` linked list.
2) iDO initializes and creates a recovery thread for each entry in the log list.
3) Each recovery thread reacquires the locks in its `lock_array` and executes a barrier with respect to other threads.
4) Each recovery thread restores its registers (including the stack pointer) from its iDO log, and jumps to the beginning of its interrupted idempotent region.
5) Each thread executes to the end of its current FASE, at which point no thread holds a lock, recovery is complete, and the recovery process can terminate.

It should be emphasized that, as with all failure atomicity systems, iDO logging does not implement full checkpointing of an executing program, nor does it provide a means of restarting execution or of continuing beyond the end of interrupted FASEs. Once the crashed program's persistent data is consistent, further recovery (if any) is fully application specific.

IV. Implementation Details

A. Compiler Implementation

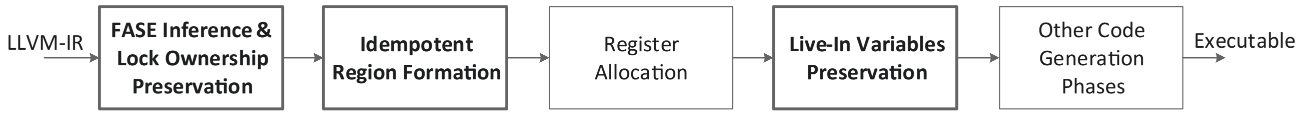
Figure 4 shows an overview of the iDO compiler, which is built on top of LLVM. It takes the generated LLVM-IR from the front end as input. It then performs three phases of instrumentation and generates the executable. We discuss the three phases in the paragraphs below.

a) FASE Inference and Lock Ownership Preservation: In its first instrumentation phase, the iDO compiler infers FASE boundaries in lock-based code, and then instruments lock and unlock operations with iDO library calls, on the assumption that each FASE is confined to a single function. As in the technical specification for transactions in C++ [24], one might prefer in a production-quality system to have language extensions with which to mark FASE boundaries in the program source, and to identify functions and function pointers that might be called from within a FASE.

b) Idempotent Region Formation: In its second instrumentation phase, the iDO compiler identifies idempotent regions. Previous idempotence-based recovery schemes have developed a simple region partition algorithm to guarantee the absence of memory antidependences, making the preservation of live-in variables the only run-time cost. We use the specific scheme developed by De Kruijf et al. [15]. The iDO compiler first computes a set of cutting points for antidependent pairs of memory accesses using LLVM's basicAA alias analysis, then applies a hitting set algorithm to select the best cutting strategy. We report region characteristics in Section V-C.

c) Preserving Inputs and Persisting Outputs: In its third and final instrumentation phase, the iDO compiler performs two key analyses. First, it guarantees that the inputs to each idempotent region are not overwritten during the region's execution. For registers, we artificially extend the live interval of each live-in register to the end of the region [25], thereby preventing the register allocator from assigning other live intervals in the region to the same register and reintroducing an antidependence. For stack variables, we perform a similar live interval extension, preventing them from being shared in LLVM's stack coloring phase [16].

In a second, related analysis, the iDO compiler ensures that outputs of the current idempotent region have persisted at the end of the region. As noted in Section III-A, registers that are live-out but were not written in the region (i.e., are being passed through from a previous region) are already known to have persisted. If a register is written multiple times, only



LLVM-IR
FASE Inference & Lock Ownership Preservation
Idempotent Region Formation
Register Allocation
Live-In Variables Preservation
Other Code Generation Phases
Executable

Fig. 4: iDO compiler overview. Starting with LLVM IR from dragonegg/clang, the compiler performs three iDO phases (indicated in bold) and then generates an executable.

the final value is logged. The log entries are then persisted (written back) at the end of the idempotent region. Similarly, writes-back of output values in the stack are initiated at the final write of the idempotent region. Writes-back of variables accessed via pointers (e.g., in the heap) are tracked at run time and then written back at the end of the region.

B. Persist Coalescing

As a further optimization, the iDO compiler takes advantage of the fact that register values are small, and do not need to persist in any particular order. A system like Atlas, which logs 32 bytes of information for every store, can persist at most two contiguous log entries in a single 64-byte cache line write-back. In iDO, as many as eight register values can be persisted with a single write-back (`clflush`). This *persist coalescing* [1] is always safe in iDO, even though registers are grouped by name rather than by order of update at run time, because the registers logged in the current region are used only in later regions. If, for example, a running program updates registers A, C, and B, in that order, it is still safe to persist the logged values of A and B together, followed by C.

C. Persistent Region Support

iDO requires mechanisms to enable processes to allocate regions of persistent memory and make those regions visible to the program. We leverage Atlas's implementation for this purpose. Atlas's region manager represents persistent memory regions as files, which processes incorporate into their address space via `mmap`. The mapped regions then support memory allocation methods such as `nv_malloc` and `nv_calloc`.

V. EVALUATION

For our evaluation of iDO logging we compared against several alternative failure atomicity runtimes. We employ real-world applications to explore iDO logging's performance impact during normal (crash-free) execution. We also employ microbenchmarks to measure scalability. For all our benchmarks, we report statistics on the idempotent regions as a guide to understanding performance. Separately, we measure recovery time. Finally, we assess the sensitivity of our results to changes in NVM latency.

Where applicable, we compare against the following failure atomic runtimes, which guarantee crash consistency on a persistent memory machine.

Atlas [7] is an UNDO-logging system that uses locks for synchronization. Like iDO logging, Atlas equates failure-atomic regions with outermost critical sections. The use of UNDO logging allows Atlas to delay a FASE's writes-back (though not those of its UNDO log) until the end of the FASE. At the same time, the lack of isolation, combined with the rollback-based recovery model, forces Atlas to track dependences across critical sections and to be prepared to roll back even a completed FASE if it depends on some other FASE that failed to complete before a crash.

Mnemosyne [2] is a REDO-based transactional system integrated into the language-level transactions of C and C++. We used the updated version included in the recently published WHISPER benchmark suite [26], but fixed a scaling bug accidentally introduced in that version. Specifically, we removed the call to `__pm_trace_print` at line 139 of `pm_instr.h`.

JUSTDO [9] is a recovery-via-resumption system, originally designed for machines with persistent caches, that logs recovery information at every store. Unlike the version from the original paper, our JUSTDO implementation adopts the iDO strategy of placing the program stack in nonvolatile memory. This change leads to a significant performance improvement by avoiding the need to manually copy stack variables into the heap on FASE initialization.

NVML [4] is Intel's UNDO-logging system. It tracks information on persistent objects and separates persistence from synchronization using programmer delineated FASEs. A library-based system, NVML requires the programmer to annotate persistent accesses in each FASE.

NVThreads [8] is a REDO-logging, lock-based system that operates at the granularity of pages using OS page protections. Critical sections maintain copies of dirty pages and release them upon lock release.

Origin indicates the uninstrumented (and thus crash-vulnerable) code, used as a performance baseline.

Atlas, iDO, and JUSTDO use the LLVM [16] back end. Mnemosyne uses the gcc 4.8 back end due to its reliance on C++ transactions, a feature not yet implemented in LLVM. For all experiments, all runtimes use the same FASEs (but Mnemosyne, as a transactional system, treats them as critical sections on a single global lock, with a speculative implementation).

For testing, we used an otherwise-idle machine with four AMD Opteron 6276 processors, each of which has 16 single-threaded cores, for a total of 64 hardware threads. Each core has access to private L1 and shared L2 caches (totaling 1 MB per core); the L3 cache (12 MB) is shared across all cores of a single processor. The machine runs CentOS 7.4. In the absence of actual nonvolatile DIMMs, we placed our "persistent"

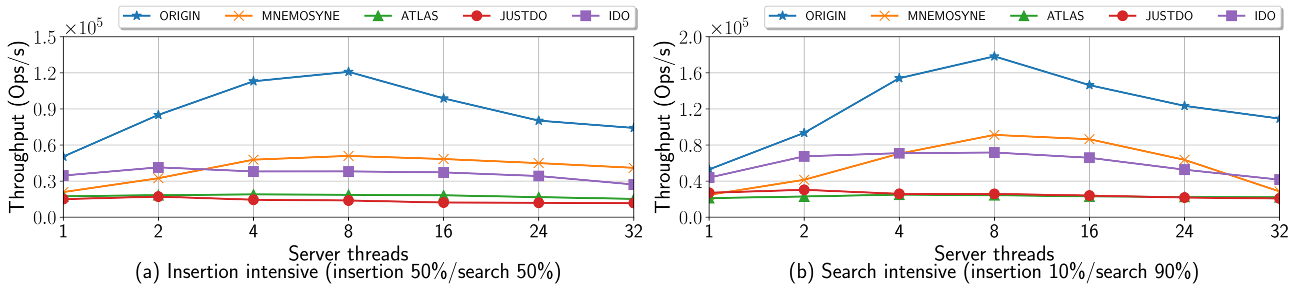


Fig. 5: Memcached throughput (millions of data structure operations per second) as a function of thread count.

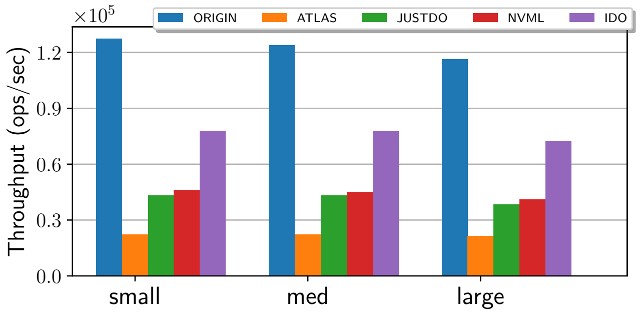


Fig. 6: Redis throughput for databases with 10K, 100K, and 1M-element key ranges.

data structures in ordinary memory (DRAM). We assume that `clflush` instructions followed by an `sfence` roughly approximate the overhead of persistence on machines (e.g., those with Intel's ADR [27]) in which the on-chip memory controller is part of the persistence domain. We explore the sensitivity of our results to this assumption in Section V-E.

A. *Performance Overhead*

To understand iDO's performance on real-world benchmarks, we integrated it, along with several other failure atomicity libraries, with **Memcached** [28] and **Redis** [29], two production-quality key-value stores.

Memcached [28] is used—typically to cache query results—by a wide variety of commercial enterprises, including Facebook, Wikipedia, and Flickr. It has been in active development since 2003. We took advantage of the fact that Mnemosyne was already integrated into an older version of the software (1.2.4) in the WHISPER benchmark [26] and further integrated iDO, Atlas, and JUSTDO into that same, lock-based code. For our experiments, we ran both a Memcached server and client on our AMD Opteron server and followed the methodology of Dice et al. [30] to maximize throughput. We used the tool `memaslap` [31] as the client to generate a stream of Memcached requests according to a desired distribution. We used 32 client threads, which generated requests with uniformly distributed 16-byte keys and 8-byte values.[2] We experimented with two types of workloads: insertion-intensive (50% insertion / 50% search) and search-intensive (10% insertion / 90% search).

[2]Memaslap accompanies Memcached in the WHISPER suite [26]. We also tried to use YCSB [32], but Mnemosyne crashed on this workload with 32 client threads.

Throughput appears in Figure 5. In general, iDO logging outperforms all other FASE-based competitors by a factor of two or more. At its peak, iDO throughput reaches 25–33% of that of the original code, imposing significant but arguably tolerable overhead in return for persistence and crash consistency. Notably, none of the systems manages to scale particularly well, and even the original version scales only to eight threads. Older versions of Memcached were notorious for exhibiting poor scaling due to coarse-grain locking [30] and the synchronization framework has been reworked since 1.2.4. Because of the coarse-grain locking in Memcached, Mnemosyne enjoys better performance than iDO. Given the scalability problems common to transactional systems (shown in Sec. V-B), we believe that iDO will outperform Mnemosyne in later versions of Memcached.

Redis [29] is an object-based key-value store that supports a wide variety of data structures as values. Unlike Memcached, Redis is single threaded, so we relied on programmer-annotated FASEs (rather than outermost locks) to delineate failure-atomic regions. As in our Memcached experiments, we took advantage of the fact that Redis has already been adapted for persistent memory [26]—in this case using NVML. Building on this prior work, we integrated iDO, JUSTDO, and Atlas into the code base. We ran both server and client on our AMD Opteron machine, using Redis's included `lru` test as the client. This client queries the server with a mix of 80% `gets` and 20% `puts`, with a power-law key distribution over a fixed key range (10K, 100K, or 1M) for one minute.

As shown in Figure 6, iDO outperforms existing persistence systems on Redis by significant margins for all key ranges, with overhead of 30–50% relative to the crash-vulnerable code. As Redis has rather long FASEs with relatively few persistent writes, iDO can take significant advantage of idempotent regions. Notably, as the database grows, the performance difference between iDO and the uninstrumented code shrinks. This effect occurs because the benchmark spends more time searching for keys in the larger database, and iDO logging imposes minimal costs on read paths, since they are idempotent. Also of note is the performance of the two UNDO logging systems—Atlas and NVML. While both provide UNDO logging, NVML has neither compiler integration nor synchronization; programmers must manually annotate every persistent store in a FASE and insert necessary synchronization. Atlas, on the other hand, achieves substantially greater ease of use (for multithreaded code) through compiler-based detection of persistent accesses and automatic tracking of cross-FASE dependences. These additional features in Atlas become performance overheads in a single-threaded benchmark like Redis.

B. Scalability

For scalability experiments, we used the same data structure microbenchmarks used in the evaluation of JUSTDO logging [9]. These microbenchmarks perform repeated accesses to a shared data structure across a varying number of threads. The data structures we implemented were:

Stack A locking variation on the Treiber Stack [33].

Queue The two-lock version of the M&S queue [34].

Ordered List A sorted list traversed using hand-over-hand locking. This implementation allows for concurrent accesses within the list, but threads cannot pass one another.

Map A fixed-size hash map that uses the ordered list implementation for each bucket, obviating the need for per-bucket locks.

These data structures allow varying degrees of parallelism. The stack, for example, serializes accesses in a very small critical section. At the other extreme, the hash map allows concurrent accesses both across and within buckets. We expect low-parallelism data structures to scale poorly with worker thread count whereas high-parallelism data structures should exhibit nearly linear scaling. Our performance results are conservative in that they present the maximum possible stress-test throughput of the structure. In real code, these data structures may not be the overall bottleneck.

At each thread count, tests are run for a fixed time interval using a low overhead hardware timer, and total operations are aggregated at the end. For the duration of microbenchmark execution, each thread repeatedly chooses a random operation to execute on the structure. For our evaluations of the queues and stacks, threads choose randomly between `insert` or `remove`. For the ordered list and hash maps, threads choose randomly between `get` or `put` on a random key within a fixed range. Threads were pinned to cores in a consistent order for all experiments: we entirely fill a single 16-core processor before moving to the next.

During each test, threads synchronize only through the tested data structure. Variables within the data structures are appropriately padded to avoid false sharing. To generate random numbers, threads use thread-local generators to avoid contention. To smooth performance curves, pages are prefaulted to prevent soft page faults. Performance of the microbenchmarks is up to 10× better without persistence; we elided this result for clarity.

Scalability results appear in Figure 7. As in Memcached and Redis, iDO logging matches or outperforms other FASE-based schemes in all configurations, especially at higher thread counts. In general, iDO logging also scales better than

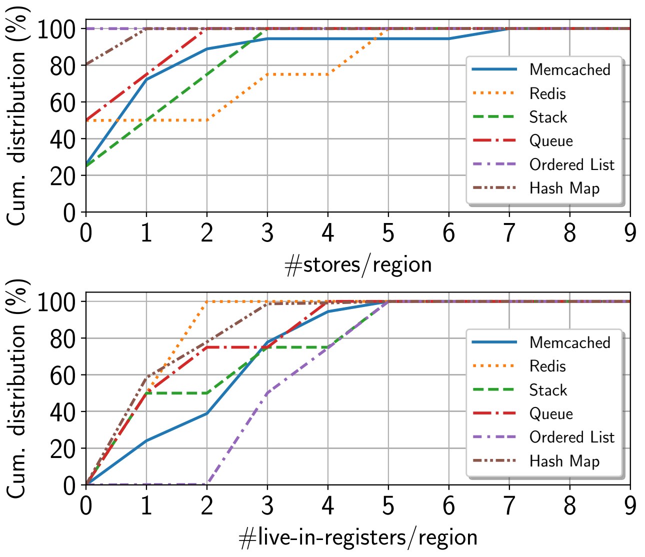


Fig. 8: Benchmark region characteristics: cumulative distribution of stores (top) and live-in registers (bottom) per dynamic region.

Mnemosyne, showing near perfect speedup on the hash map. This scaling demonstrates the absolute lack of synchronization between threads in the iDO runtime—all thread synchronization is handled through the locks of the original program. In contrast, both Atlas and Mnemosyne quickly saturate their runtime's synchronization and throttle performance.

Mnemosyne performs better when the applications have little inherent concurrency or when the number of worker threads is low. Since both iDO and Atlas require ordered writes to persistent memory at every lock acquisition and release in order to track lock ownership, their per-thread execution is slowed relative to Mnemosyne, which employs a speculative implementation. Conversely, both iDO and Atlas support hand-over-hand locking, as used in the ordered list. Mnemosyne, with its transactional API, does not support this idiom, so the entire traversal is done in a single transaction and data is written to persistent memory only once. iDO and Atlas extract more concurrency from the benchmark, but per-thread execution is slower than Mnemosyne. Consequently, at very high thread counts, iDO outperforms Mnemosyne due to extracted parallelism, despite its single thread performance being about 4× slower.

C. Region Characteristics

To better understand performance differences, we used Intel's Pin tool [35] to collect statistics on idempotent regions. For each of our applications and microbenchmarks, the upper half of Figure 8 displays the cumulative dynamic distribution of stores per idempotent region. Any number larger than one indicates a savings in logging operations relative to REDO, UNDO, or JUSTDO logging.

In the microbenchmarks, most regions contain zero or one stores. The Ordered List, in particular, spends much of its time

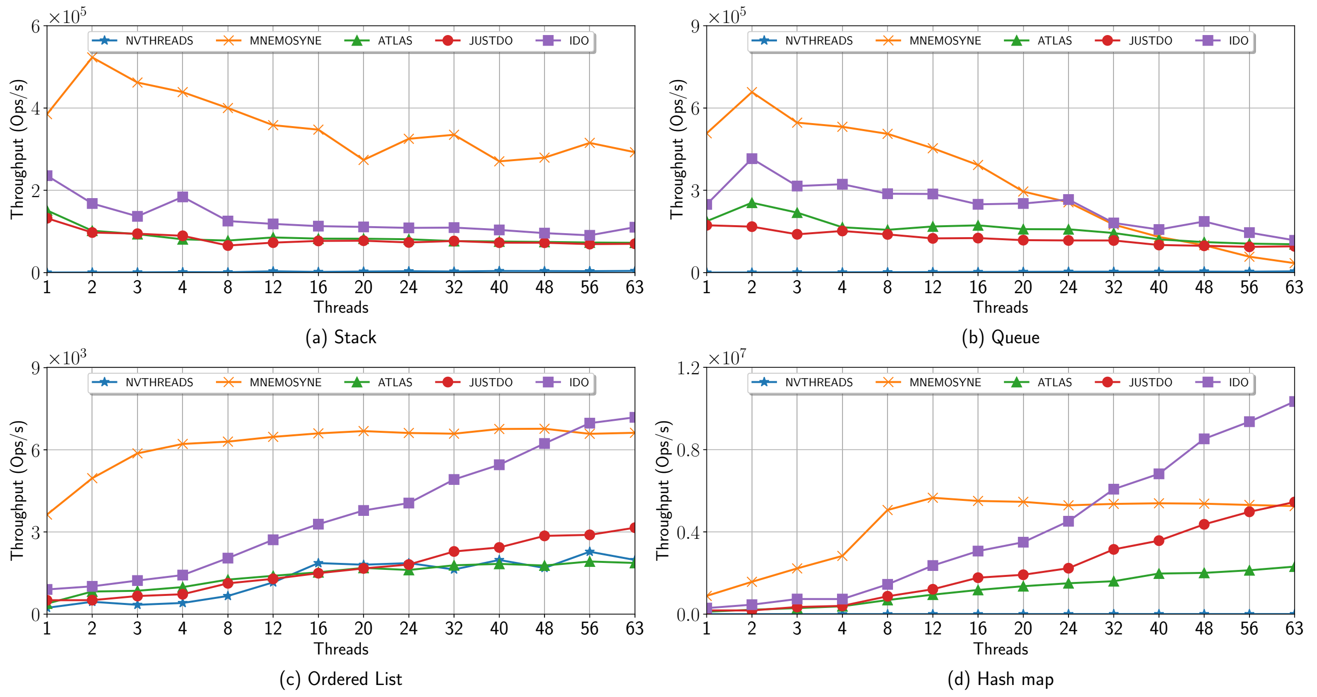


Fig. 7: Throughput (millions of data structure operations per second) as a function of thread count.

searching with hand-over-hand locking and no data updates. This allows Mnemosyne, which avoids logging lock operations, to outperform all other schemes in this microbenchmark for low and moderate thread counts. Even with very small regions, iDO still outperforms JUSTDO by substantial margins, largely because of its indirect locking mechanism.

For more realistic applications, we observe that roughly 30% (Memcached) to 50% (Redis) of all regions have multiple stores, allowing iDO to consolidate log operations, leading to higher throughput even at low thread counts. We believe the average region size could be improved with better alias analysis in the compiler. We currently rely on LLVM's basic-AA algorithm, which is quite conservative.

The lower half of Figure 8 displays the cumulative dynamic distribution of live-in registers per idempotent region. Significantly, more than 99% of the dynamic regions in all the benchmarks have fewer than five live-in registers, indicating that the typical log operation requires only a single cache line flush for the register inputs.

D. Recovery Overheads

We evaluate the speed and correctness of recovery by running the microbenchmarks of Section V-B and killing the process. We interrupt the applications by sending an external SIGKILL signal after the applications have run for 1, 10, 20, 30, 40 and 50 seconds. For the recovery, iDO follows the recovery procedure in Section III-C. As summarized before, iDO needs to first initialize the recovery threads. Then iDO recovers the live-in variables for the interrupted region, jumps back to the entry of the interrupted region, and continues execution until the end of the FASE. Interestingly, the recovery time for iDO with 64 threads is always about one second. Since most of the FASEs in the benchmarks are short (generally on the order of a microsecond), the main overhead for iDO recovery comes from mapping the persistent region into the process's virtual address space and creating the recovery threads—all of which is essentially constant overhead. In contrast, for Atlas, recovery needs to first traverse the logs and compute a global consistent state following the happens-before order recorded in the logs, then undo any stores in the interrupted FASEs.

Kill Time	1 s	10 s	20 s	30 s	40 s	50 s
Stack	0.7	6.6	14.0	20.7	28.7	34.9
Queue	0.8	9.0	20.1	31.6	43.3	56.1
OrderedList	4.1	72.1	162.2	260.9	301.8	424.8
HashMap	0.3	1.5	2.7	4.2	5.2	6.2

TABLE I: Recovery time ratio (ATLAS/iDO) at different kill times.

Table I shows the ratio of recovery times for ATLAS and iDO. When the applications run for only a short time (1 second) before "crashing," ATLAS can quickly traverse the small number of logs and compute a consistent state, while iDO still has to pay the overhead of creating and initializing recovery threads. However, when the applications run for a longer time (> 10 seconds), ATLAS must traverse a much larger number of logs and compute a consistent state. We can observe up to 400$\times$ faster recovery for iDO in this case.

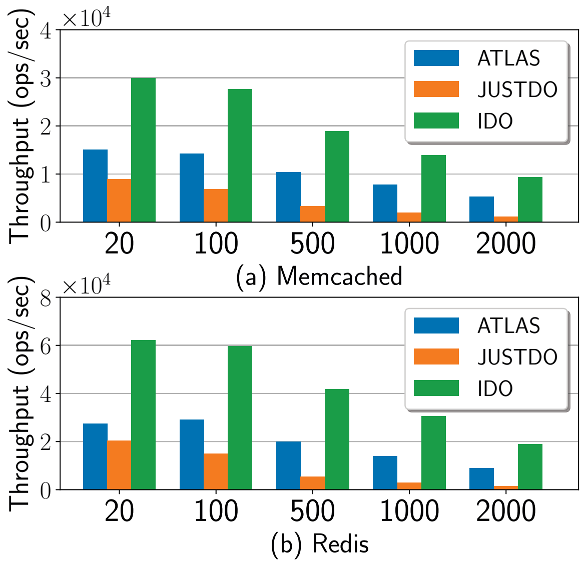


Fig. 9: Sensitivity to NVM latency (ns).

E. Sensitivity to NVM Latency

In the experiments of Sections V-A through V-D, we relied on `sfence` instructions to capture the cost of waiting for previous writes-back (`clflush`es) to reach the on-chip memory controller. Given that some machines may implement this controller with, say, STT-MRAM instead of capacitor-backed SRAM, while others may require a handshake across the memory bus, we re-ran our Memcached and Redis experiments with additional delays to emulate the cost of nonvolatile writes (which are typically more expensive than reads) or of traversing a long data path. As in Mnemosyne [2], we inserted a configurable delay (looping with `nop`s) after each non-cacheable store to nonvolatile memory, and after each `clflush` that follows cacheable stores to such memory. We also leveraged a similar capability in Atlas. Results, for delays ranging from 20–2000 ns, appear in Figure 9. The Memcached result reprises the 32-server, insertion-intensive data point of Figure 5(a); the Redis result reprises the "large" data point of Figure 6.

Both iDO logging and Atlas maintain their performance up to a delay of around 100 ns; beyond this point, significant slowdown occurs. JUSTDO logging, by contrast, sees significant (1.5–2×) slowdown relative to Figures 5 and 6 with even 20 ns of additional delay. We attribute this difference to the relatively frequent logging of JUSTDO relative to Atlas and iDO. While very preliminary, we take these results both as a strong endorsement of Intel's ADR [27] (asynchronous DRAM refresh on power fail) and as a suggestion that it may be reasonable, *with an appropriate runtime*, to replace capacitor-backed SRAM with physically nonvolatile memory in an ADR memory controller.

VI. Related Work

A. Nonvolatile Memory

With the impending end of DRAM scaling, several device technologies are competing to provide inexpensive, dense, and nonvolatile storage in the hopes of becoming the next dominant main memory technology. Candidates include PCM [36], [37], Memristors [38], and spin-transfer torque magnetic memory (STT-MRAM) [39]. In building these technologies, researchers are attempting to maximize density, endurance, economy, and speed, resulting in various compromises across these variables.

At the level of the microarchitecture, architects are trying to give programmers fast and fine-grained control over the ordering and timing of writes-back from volatile caches into nonvolatile main memory; the semantics surrounding this ordering comprise the *memory persistency model* [1] analogous to traditional memory consistency [40]. While existing processors provide rough control over write back (e.g., Intel's `clflush` and nontemporal stores), future designs may track thread-local orderings and buffering to reduce the penalty of ordering writes into persistence. Various schemes and their hardware include epoch persistence [36], buffered epoch persistence [1], [41], explicit epoch persistence [42], DPO [43], and HOPS [26]. Intel has also released ISA extensions for its persistency model [10] including the new `clwb` and `clflushopt` instructions.

On top of these persistency models, several research groups have built high performance software for persistent applications. Example projects include concurrent data structures [44]–[47] transactional key-value stores [6], [48]–[50], file systems [17], and databases [51]–[53].

In contrast with these high-performance and specialized applications, a growing body of work, of which iDO logging is a member, addresses run-time libraries and compiler support to enable programmers to more easily write crash-resistant code. Table II summarizes the differences among several of these systems. Mnemosyne [2], NV-Heaps [3], SoftWrAP [55], and NVML [54] extend transactional memory to provide durability guarantees on nonvolatile memory. Mnemosyne emphasizes performance; its use of REDO logs postpones the need to flush data to persistence until a transaction commits. SoftWrAP, also a REDO system, uses shadow paging and Intel's now deprecated `pcommit` instruction [10] to batch updates from DRAM to NVM. NV-heaps, an UNDO log system, emphasizes programmer convenience, providing garbage collection and strong type checking to help avoid pitfalls unique to persistence—e.g., pointers to transient data inadvertently stored in persistent memory. NVML, Intel's persistent memory transaction system, uses UNDO logging on persistent objects and implements several highly optimized procedures that bypass transactional tracking for common functions.

Other failure atomic run-time systems [7]–[9], like iDO logging, use locks for synchronization and delineate failure atomic regions as outermost critical sections, as discussed in Section I.

TABLE II: Failure Atomic Systems and their Properties

System	Failure-atomic region semantics	Recovery Method	Logging Granularity	Dependency tracking needed?	Designed for transient caches?
iDO Logging	Lock-inferred FASE	Resumption	Idempotent Region	No	Yes
Atlas [7]	Lock-inferred FASE	UNDO	Store	Yes	Yes
Mnemosyne [2]	C++ Transactions	REDO	Store	No	Yes
NVThreads [8]	Lock-inferred FASE	REDO	Page	Yes	Yes
JUSTDO [9]	Lock-inferred FASE	Resumption	Store	No	No
NVHeaps [3]	Transactions	UNDO	Object	No	Yes
NVML [54]	Programmer Delineated	UNDO	Object	No	Yes
SoftWrAP [55]	Programmer Delineated	REDO	Contiguous data blocks	No	Yes

Extensions to several of these systems explore how to compose operations on concurrent persistent data structures into larger failure atomic sections, thereby eliminating fine-grained write tracking within the persistent data structure. For data structures that meet *detectable execution* [56], query-based logging [57] allows UNDO and JUSTDO based systems to support this optimized composition (analogous to "boosting" in software transactional memory [58]). It seems clear that similar optimizations could work in iDO logging.

All of these specialized persistent applications and runtimes can be seen as nonvolatile memory analogues of traditional failure atomic systems for disk/flash, and they borrow many techniques from the literature. Disk-based database systems have traditionally used write-ahead logging to ensure consistent recoverability [59]. Transactional file updates have been explored in research prototypes [60], [61] and commercial implementations [62]. User-space implementations of persistent heaps supporting failure-atomic updates have been explored in research [63]. Logging-based systems have historically ensured consistency by discarding changes from any update interrupted by failure (even in the REDO case, an update will not be completed on recovery unless it recorded everything it wanted to do before the failure occurred). In contrast, for idempotent updates, an update cut short by failure can simply be re-executed rather than discarding changes, reducing required logging [64], [65].

B. Idempotence

Over the years, many researchers have leveraged *idempotence* for various purposes. Mahlke et al. were the first to exploit the idea, which they used to recover from exceptions during speculative execution in a VLIW processor [66]. Around the same time, Bershad et al. proposed *restartable atomic sequences* for a uniprocessor based on idempotence [65].

Kim et al. leveraged idempotence to reduce the hardware storage required to buffer data in their compiler-assisted speculative execution model [67]. Hampton et al. used idempotence to support fast and precise exceptions in a vector processor with virtual memory [68]. Tseng et al. used idempotent regions for data-triggered thread execution [69].

Recently, researchers have leveraged idempotence for recovery from soft errors—e.g., ECC faults [15], [70]. Also, Liu et al. [20] advanced the state of the art with *checkpoint pruning*, which serves to remove logging operations that can be reconstructed from other logs in the event of a soft run-time error. Liu et al. [21], [22], [71], [72] also extended idempotent processing in the context of sensor-based soft error detectors to ensure complete recovery.

More recently, the energy-harvesting system community has started using idempotent processing to recover from the frequent power failures that occur in systems without batteries. Xie et al. [73] use idempotence-based recovery and heuristics to approximate minimal checkpoints (logs) to survive power failures. Their design revolves around the idea of severing anti-dependences by placing a checkpoint between a load-store pair, in a manner reminiscent of Feng et al. [70] and de Kruijf et al. [15]. Lately, their techniques were used by Woude et al. [74] to highlight both the promise and the limitations of using idempotence to ensure forward progress when multiple power failures occur within a span of microseconds. In a similar vein, Liu et al. [75] highlight the limitations of anti-dependence based idempotence analysis in terms of additional power consumption due to unnecessary checkpoints. Significantly, all of these projects target embedded processors in which out-of-order execution and caches do not exist.

Despite this wealth of related work, iDO is, to the best of our knowledge, the first system to use idempotence to achieve lightweight, fault-tolerant execution of failure-atomic sections in general-purpose programs.

VII. Conclusion

Fault tolerance is one of the most exciting applications of emerging nonvolatile memory technologies. Existing approaches to persistence, however, suffer from problems with both performance and usability. Transactional approaches are generally incompatible with existing lock-based code, and tend not to scale to high levels of concurrency. Failure-atomic regions (FASEs), by contrast, are compatible with most common locking idioms and introduce no new barriers to scalability. Unfortunately, prior FASE-based approaches to persistence incur significant run-time overhead.

To address these limitations, we have introduced iDO logging, a compiler-directed approach to failure atomicity. The iDO compiler divides each FASE into idempotent regions, arranging on failure recovery to restart any interrupted idempotent region and execute forward to the end of the FASE. Unlike systems based on UNDO or REDO logging, iDO avoids the need to log individual program stores, thereby achieving a significant reduction in instrumentation overhead. Across a variety of benchmark applications, iDO significantly outperforms the fastest existing lock-based persistent systems during normal execution, even on machines with conventional volatile caches, while preserving very fast recovery times.

Acknowledgments

At Virginia Tech, this work was supported by NSF grants 1750503 (CAREER), 1527463, and 1814430, and by Google/AMD Faculty Research awards. At the University of Rochester, this work was supported by NSF grants 1319417, 1337224, 1422649, and 1717712, and by a Google Faculty Research award. At UNIST, this work was supported by the Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1402-52. Our thanks to Wentao Cai for identifying the scaling bug in the WHISPER version of Mnemosyne.

References

[1] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *41st Intl. Symp. on Computer Architecuture (ISCA)*, Minneapolis, MN, 2014.

[2] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, 2011.

[3] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, 2011.

[4] Intel, "Intel NVM Library," https://github.com/pmem/nvml/.

[5] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "DudeTM: Building durable transactions with decoupling for persistent memory," in *22nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, 2017.

[6] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, 2016.

[7] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Intl. Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, Portland, OR, 2014.

[8] T. C.-H. Hsu, H. Bruegner, I. Roy, K. Keeton, and P. Eugster, "NVthreads: Practical persistence for multi-threaded applications," in *12th ACM European Systems Conf. (EuroSys)*, Belgrade, Republic of Serbia, 2017.

[9] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via JUSTDO logging," in *21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, 2016.

[10] Intel Corporation, "Intel architecture instruction set extensions programming reference," Intel Corporation, Tech. Rep. 3319433-029, Apr. 2017.

[11] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking SIMD vectorization for in-memory databases," in *Intl. Conf. on Management of Data (SIGMOD)*, Melbourne, Victoria, Australia, 2015.

[12] T. Willhalm, I. Oukid, I. Müller, and F. Faerber, "Vectorizing database column scans with complex predicates." in *Intl. Wkshp. on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB)*, Riva del Garda, Trento, Italy, 2013.

[13] K. Keeton, "The machine: An architecture for memory-centric computing," in *5th Intl. Wkshp. on Runtime and Operating Systems for Supercomputers (ROSS)*, Portland, OR, 2015.

[14] W. Wei, D. Jiang, J. Xiong, and M. Chen, "Exploring opportunities for non-volatile memories in big data applications," in *Wkshp. on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*. Springer, 2014.

[15] M. A. de Kruijf, K. Sankaralingam, and S. Jha, "Static analysis and compiler design for idempotent processing," in *33rd ACM Conf. on Programming Language Design and Implementation (PLDI)*, Beijing, China, 2012.

[16] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Intl. Symp. on Code Generation and Optimization (CGO)*, San Jose, GA, 2004.

[17] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *22nd Symp. on Operating Systems Principles (SOSP)*, Big Sky, MT, 2009.

[18] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," in *Intl. Conf. on Object-Oriented Programming, Systems, Languages, & Applications (OOPSLA)*, Amsterdam, The Netherlands, 2016.

[19] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys*, vol. 15, Dec. 1983.

[20] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery," in *Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, Salt Lake City, UT, 2016.

[21] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed soft error detection and recovery to avoid DUE and SDC via Tail-DMR," *ACM Trans. on Embedded Computing Systems (TECS)*, vol. 16, Dec. 2016.

[22] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection," in *49th Intl. Symp. on Microarchitecture (MICRO)*, Taipei, Taiwan, 2016.

[23] S. Muchnick, *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[24] M. Wong, V. Luchangco *et al.*, "SG5 transactional memory support for C++," Oct. 2014, document number N4180, Programming Language C++, Evolution Working Group, Intl. Organization for Standardization.

[25] M. de Kruijf and K. Sankaralingam, "Idempotent code generation: Implementation, analysis, and evaluation," in *Intl. Symp. on Code Generation and Optimization (CGO)*, Shenzhen, China, 2013.

[26] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with WHISPER," in *22nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, 2017.

[27] A. M. Rudoff, "Deprecating the pcommit instruction," software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction, Intel Corp., Sep. 2016.

[28] memcached.org, "memcached – a distributed memory object caching system," http://memcached.org/, accessed: 2017.

[29] RedisLabs, "Redis," 2015, http://redis.io.

[30] D. Dice, V. J. Marathe, and N. Shavit, "Lock cohorting: A general technique for designing NUMA locks," *ACM Trans. on Parallel Computing (TOPC)*, vol. 1, 2015.

[31] libMemcached.org, "libMemcached," http://www.libMemcached.org, 2011.

[32] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *1st ACM Symp. on Cloud Computing (SoCC)*, Indianapolis, IN, 2010.

[33] R. K. Treiber, "Systems programming: Coping with parallelism," IBM Almaden Research Center, Tech. Rep. RJ 5118, Apr. 1986.

[34] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *1996 ACM Symp. on Principles of Distributed Computing (PODC)*, Philadelphia, PA, 1996.

[35] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *26th ACM Conf. on Programming Language Design and Implementation (PLDI)*, Chicago, IL, 2005.

[36] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *36th Intl. Symp. on Computer Architecture (ISCA)*, Austin, TX, 2009.

[37] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla,

B. Rajendran, S. Raoux, and R. S. Shenoy, "Phase change memory technology," *Journal of Vacuum Science and Technology*, vol. 28, 2010.

[38] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, May 2008.

[39] D. Apalkov, A. Khvalkovskiy, S. Watta, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi, "Spin-transfer torque magnetic random access memory (STT-MRAM)," in *ACM Journal on Emerging Technologies in Computing Systems (JETC)—Special issue on memory technologies*, 2013.

[40] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, Dec. 1996.

[41] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *48th Intl. Symp. on Microarchitecture (MICRO)*, Waikiki, HI, 2015.

[42] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model," in *30th Intl. Conf. on Distributed Computing (DISC)*, Paris, France, 2016.

[43] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *49th Intl. Symp. on Microarchitecture (MICRO)*, Taipei, Taiwan, 2016.

[44] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner, "NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories," in *3rd VLDB Wkshp. on In-Memory Data Mangement and Analytics (IMDM)*, Kohala Coast, HI, 2015.

[45] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory," in *Intl. Conf. on Management of Data (SIGMOD)*, San Francisco, CA, 2016.

[46] S. Chen and Q. Jin, "Persistent B+-trees in non-volatile main memory," *Proc. of the VLDB Endowment*, vol. 8, Feb. 2015.

[47] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey, D. Chakrabarti, and M. L. Scott, "Dalí: A periodically persistent hash map," in *31st Intl. Symp. on Distributed Computing (DISC)*, Vienna, Austria, 2017.

[48] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *9th USENIX Conf. on File and Stroage Technologies (FAST)*, San Jose, CA, 2011.

[49] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-tree: Reducing consistency cost for NVM-based single level systems," in *13th USENIX Conf. on File and Storage Technologies (FAST)*, Santa Clara, CA, 2015.

[50] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "Wort: Write optimal radix tree for persistent memory storage systems," in *15th USENIX Conf. on File and Storage Technologies (FAST)*, Santa Clara, CA, 2017.

[51] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage management in the NVRAM era," *Proc. of the VLDB Endowment*, vol. 7, Oct. 2013.

[52] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proc. of the VLDB Endowment*, vol. 7, Jun. 2014.

[53] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. Dulloor, "A prolegomenon on OLTP database systems for non-volatile memory," in *Intl. Wkshp. on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB)*, Hangzhou, China, 2014.

[54] A. Rudoff, "Persistent memory programming," http://pmem.io/, accessed: 2017-04-21.

[55] E. R. Giles, K. Doshi, and P. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *31st Symp. on Massive Storage Systems and Technology (MSST)*, Santa Clara, CA, 2015.

[56] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank, "A persistent lock-free queue for non-volatile memory," in *23rd ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Vienna, Austria, 2018.

[57] J. Izraelevitz, "Concurrency implications of nonvolatile byte-addressable memory," Dept. of Computer Science, U. of Rochester, 2018, Ph.D. thesis.

[58] M. Herlihy and E. Koskinen, "Transactional boosting: A methodology for highly-concurrent transactional objects," in *13th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Salt Lake City, UT, 2008.

[59] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. on Database Systems (TODS)*, vol. 17, Mar. 1992.

[60] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright, "Enabling transactional file access via lightweight kernel extensions," in *7th USENIX Conf. on File and Storage Technologies (FAST)*, San Francisco, CA, 2009.

[61] S. Park, T. Kelly, and K. Shen, "Failure-atomic `msync()`: A simple and efficient mechanism for preserving the integrity of durable data," in *8th ACM European Systems Conf. (EuroSys)*, Prague, Czech Republic, 2013.

[62] R. Verma, A. A. Mendez, S. Park, S. Mannarswamy, T. Kelly, and C. B. M. III, "Failure-atomic updates of application data in a Linux file system," in *13th USENIX Conf. on File and Storage Technologies (FAST)*, Santa Clara, CA, 2015.

[63] S. Yoo, C. Killian, T. Kelly, H. K. Cho, and S. Plite, "Composable reliability for asynchronous systems," in *USENIX Annual Technical Conf. (ATC)*, Boston, MA, 2012.

[64] M. de Kruijf and K. Sankaralingam, "Idempotent processor architecture," in *44th Intl. Symp. on Microarchitecture (MICRO)*, Porto Alegre, Brazil, 2011.

[65] B. N. Bershad, D. D. Redell, and J. R. Ellis, "Fast mutual exclusion for uniprocessors," in *5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, 1992.

[66] S. A. Mahlke, W. Y. Chen, W.-m. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for vliw and superscalar processors," in *5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, 1992.

[67] S. W. Kim, C.-L. Ooi, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar, "Exploiting reference idempotency to reduce speculative storage overflow," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 28, Sep. 2006.

[68] M. Hampton and K. Asanović, "Implementing virtual memory in a vector processor with software restart markers," in *20th Intl. Conf. on Supercomputing (ICS)*, Cairns, Queensland, Australia, 2006.

[69] H.-W. Tseng and D. M. Tullsen, "CDTT: Compiler-generated data-triggered threads," in *20th Intl. Symp. on High Performance Computer Architecture (HPCA)*, Orlando, FL, 2014.

[70] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August, "Encore: low-cost, fine-grained transient fault recovery," in *44th Intl. Symp. on Microarchitecture (MICRO)*, Porto Alegre, Brazil, 2011.

[71] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler directed lightweight soft error resilience," in *16th ACM Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES)*, Portland, OR, 2015.

[72] Q. Liu, "Compiler-directed error resilience for reliable computing," Ph.D. dissertation, Virginia Tech, 2018.

[73] M. Xie, M. Zhao, C. Pan, J. Hu, Y. Liu, and C. Xue, "Fixing the broken time machine: Consistency-aware checkpointing for energy harvesting powered non-volatile processor," in *52nd IEEE/ACM Design Automation Conf. (DAC)*, San Francisco, CA, 2015.

[74] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, 2016.

[75] Q. Liu and C. Jung, "Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems," in *6th IEEE Non-Volatile Memory Systems and Applications Symp. (NVMSA)*, Daegu, South Korea, 2016.