Failure-Atomic Persistent Memory Updates via JUSTDO Logging

Joseph Izraelevitz
U. Rochester/Hewlett Packard Labs
jhi1@cs.rochester.edu

Terence Kelly
Hewlett Packard Labs
terence.p.kelly@hpe.com

Aasheesh Kolli
U. Michigan/Hewlett Packard Labs
akolli@umich.edu

Abstract

Persistent memory invites applications to manipulate persistent data via LOAD and STORE instructions. Because failures during updates may destroy transient data (e.g., in CPU registers), preserving data integrity in the presence of failures requires failure-atomic bundles of updates. Prior failure atomicity approaches for persistent memory entail overheads due to logging and CPU cache flushing. Persistent caches can eliminate the need for flushing, but conventional logging remains complex and memory intensive. We present the design and implementation of JUSTDO logging, a new failure atomicity mechanism that greatly reduces the memory footprint of logs, simplifies log management, and enables fast parallel recovery following failure. Crash-injection tests confirm that JUSTDO logging preserves application data integrity and performance evaluations show that it improves throughput 3× or more compared with a state-of-the-art alternative for a spectrum of data-intensive algorithms.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability—fault-tolerance

Keywords non-volatile memory, persistent memory, crash-resilience, failure-atomicity, transactions

1. Introduction

Emerging byte-addressable non-volatile memory (NVM) device technologies are widely expected to augment and perhaps eventually supplant conventional DRAM as density scaling limitations constrain the latter [37]. Meanwhile, persistent memory implemented with conventional device technologies (e.g., non-volatile DIMMS based on DRAM, flash storage, and supercapacitors [51]) offers similar attractions: Regardless of how it is implemented, persistent memory invites applications to manipulate persistent application data directly via ordinary LOAD and STORE instructions. By contrast, on today's conventional hardware with volatile byte-addressable memory and non-volatile block-addressed storage, complex multi-layered software stacks—operating systems, file systems, and database management systems—provide indirect and mediated access to persistent data.

Eliminating the memory/storage distinction and the associated layers of intermediary software promises to streamline software and improve performance, but direct in-place manipulation of persistent application data allows a failure during an update to corrupt data. Mechanisms supporting program-defined *failure-atomic sections* (FASEs) address this concern. FASEs can be implemented as transactional memory with additional durability guarantees [14, 52] or by leveraging applications' use of mutual exclusion primitives to infer consistent states of persistent memory and guarantee consistent recovery [9]. These prior systems offer generality and convenience by automatically maintaining UNDO [9, 14] or REDO [52] logs that allow recovery to roll back FASEs that were interrupted by failure. Maintaining such logs inevitably entails space and time overheads; flushing logs from transient CPU caches to persistent memory incurs additional time and memory-bandwidth overheads.

Persistent CPU caches eliminate the need to flush caches to persistent memory and can be implemented in several ways, e.g., by using inherently non-volatile bit-storage devices in caches [61] or by maintaining sufficient standby power to flush caches to persistent memory in case of power failure. The amount of power required to perform such a flush is so small that it may be obtained from a supercapacitor [53] or even from the system power supply [32]. Preserving CPU cache contents in the face of detectable non-corrupting application software failures requires no special hardware: STOREs to file-backed memory mappings persist beyond process crashes [33].

However, even if persistent caches eliminate the cache flushing overheads of FASE mechanisms, the overhead of conventional UNDO or REDO log management remains. A simple example illustrates the magnitude of the problem: Consider a multi-threaded program in which each thread uses a FASE to atomically update the entire contents of a long linked list. Persistent memory transaction systems [14, 52] would serialize the FASEs—in effect, each thread acquires a global lock on the list—and would furthermore

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA
Copyright © 2016 ACM 978-1-4503-4091-5/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2872362.2872410

maintain a log whose size is proportional to the list modifications to support rolling back changes. A mutex-based FASE mechanism for persistent memory [9] avoids serializing FASEs by allowing concurrent updates via hand-over-hand locking but must still maintain per-thread logs, proportional in size to the amount of modified list data.

The key insight behind our approach is that mutex-based critical sections are intended to execute to completion; unlike optimistic transactions, they do not abort due to conflict. While it is possible to implement rollback for lock-based FASEs [9], we might instead simply *resume* FASEs following failure and execute them to completion. This insight suggests a design that employs minimalist logging in the service of FASE resumption rather than rollback.

Our contribution, JUSTDO logging, unlike traditional UNDO and REDO logging, does not discard changes made during FASEs cut short by failure. Instead, our approach resumes execution of each interrupted FASE at its last STORE instruction then executes the FASE to completion. Each thread maintains a small log that records its most recent STORE within a FASE; the log contains the destination address of the STORE, the value to be placed at the destination, and the program counter. FASEs that employ JUSTDO logging access only persistent memory, which ensures that all data necessary for resuming an interrupted FASE will be available during recovery. As in the Atlas system [9], we define a FASE to be an outermost critical section protected by one or more mutexes; the first mutex acquired at the start of a FASE need not be the same as the last mutex released at the end of the FASE (see Figure 1). Auxiliary logs record threads' mutex ownership for recovery.

Our approach has several benefits: By leveraging persistent CPU caches where available, we can eliminate cache flushing overheads. Furthermore the small size of JUSTDO logs can dramatically reduce the space overheads and complexity of log management. By relying on mutexes rather than transactions for multi-threaded isolation, our approach supports high concurrency in scenarios such as the aforementioned list update example. Furthermore we enable fast *parallel* recovery of all FASEs that were interrupted by failure. JUSTDO logging can provide resilience against both power outages and non-corrupting software failures, with one important exception: Because we sacrifice the ability to roll back FASEs that were interrupted by failure, bugs *within* FASEs are not tolerated. Hardware and software technologies for fine-grained intra-process memory protection [12, 54] and for software quality assurance [7, 8, 19, 42, 60] complement our approach respectively by preventing arbitrary corruption and by eliminating bugs in FASEs.

In this paper, we describe the design and implementation of JUSTDO logging and evaluate its correctness and performance. Our results show that JUSTDO logging provides a useful new way to implement persistent memory FASEs with improved performance compared with a state-of-the-art system: On five very different mutex-based concurrent data structures, JUSTDO logging increases operation throughput over 3× compared with crash resilience by the state-of-the-art Atlas FASE mechanism [9].

The remainder of this paper is organized as follows: Section 2 presents key concepts and terminology used in our paper. Section 3 reviews persistent memory technologies and their implications for software. Section 4 presents our assumptions regarding the system on which JUSTDO logging runs and the programming model that our approach supports. Section 5 describes the design of JUSTDO logging, and Section 6 presents the details of our current implementation. Section 7 evaluates the correctness and performance of our approach, and Section 8 concludes with a discussion.

2. Concepts & Terminology

Application data typically must satisfy application-level invariants or other correctness criteria. We say that data are *consistent* if the relevant application-level correctness criteria hold, otherwise the data are *corrupt*. *Failures* are events that may corrupt application data; familiar examples include application process crashes, operating system kernel panics, and abrupt power outages. We say that a failure is *tolerated* if application data consistency either is unaffected by the failure or is restored by post-failure recovery procedures. We distinguish between *corrupting* and *non-corrupting* failures; the former preclude successful recovery by corrupting application data directly or by corrupting data necessary for recovery (e.g., logs). A corrupting failure may be caused, for example, by a STORE through a pointer variable containing an invalid address.

We say that data are *persistent* if they survive tolerated failures intact and are accessible by recovery code, otherwise the data are *transient*. Similarly we say that memory locations, memory address ranges, processor cache lines, and other places where data may reside are persistent or transient depending on whether or not the data they contain will be available to recovery code following any tolerated failure. For example, a *persistent memory region* is a contiguous range of virtual addresses whose contents will survive tolerated failures. Note that persistence does not imply consistency: Failure may render persistent data irreparably corrupt, making recovery impossible.

We reserve the term *non-volatile* for characterizing device technologies that retain data even in the absence of supplied power; examples include memristor, STT-RAM, and PCM. Similarly the term *volatile* characterizes device technologies such as DRAM that require continuously supplied power to retain data. We emphasize that our persistent/transient distinction is orthogonal to volatility. For example, while non-volatile memory (NVM) facilitates the implementation of memory that is persistent with respect to certain kinds of failure, persistent memory also admits alternative implementations that do not involve NVM. Moreover, NVM

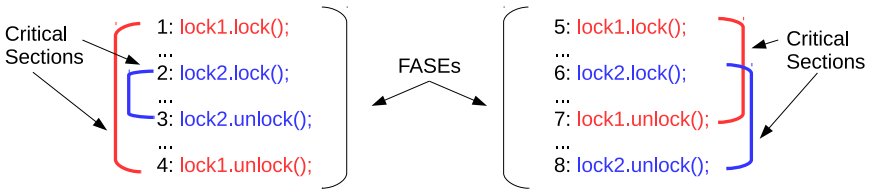

Figure 1. Two examples of lock-delimited FASEs. Left (lines 1–4): Nested. Right (lines 5–8): Hand-over-hand.

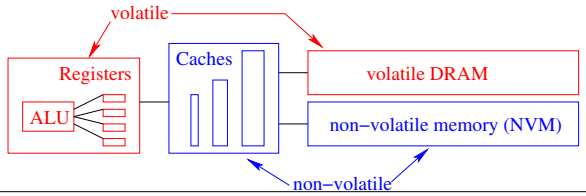

Figure 2. Hybrid architecture incorporating both conventional volatile CPU registers and DRAM in addition to non-volatile CPU caches and NVM.

need not be persistent according to our definition: For example, if thread stacks on a particular computer are reclaimed by the operating system following abnormal process termination, then stack data are not available to recovery code and are therefore transient, even if every byte of memory on the machine is non-volatile.

We distinguish between *partial* and *whole-system* persistence. The latter applies when the entire state of a machine survives tolerated failures, whereas the former describes situations in which some data is persistent and some is transient. Partial persistence results when applications designate only some data as persistent (e.g., a persistent memory region containing long-term application data) and allow the remainder to be transient (e.g., per-thread function call stacks). Partial persistence is a natural match for future *hybrid architectures* that incorporate both volatile and non-volatile components, as depicted in Figure 2.

We conclude this section by using our definitions to briefly define our hardware and software system model, characterize the failures that JUSTDO logging can tolerate, and describe situations where our approach is likely to offer good performance; all of these topics are covered in greater detail in subsequent sections. JUSTDO logging is designed with future hybrid architectures in mind (Figure 2). More specifically, our system model (Section 4) and our design (Section 5) assume that CPU registers are transient but that both CPU caches and (part of) main memory are persistent, and our programming model assumes partial persistence. JUSTDO logging tolerates non-corrupting failures that were not caused by software bugs within a failure-atomic section. We expect JUSTDO logging to achieve good performance if it is inexpensive to impose ordering constraints on modifications to persistent data—as would be the case with persistent caches and/or persistent STORE buffers integrated into the CPU in addition to persistent memory.

3. Related Work

Hardware for Persistence Memory footprints of modern workloads are increasing at a rate higher than Moore's law [56]. With the end of DRAM scaling (the memory technology of choice for decades) on the horizon, much research has focused on incorporating alternative, denser memory technologies into future architectures [22, 23, 40, 56]. Potential DRAM replacement technologies such as phase change memory (PCM) [23], memristor [48] and spin-torque transfer memory (SST-RAM) [3] offer higher densities and are furthermore non-volatile. These non-volatile memory (NVM) device technologies facilitate the implementation of memory that is persistent with respect to power outages. Systems with persistent memory allow persistent data manipulation via processor LOADs and STOREs [37], a stark deviation from traditional disk/flash based durability. The prospect of persistent memory has sparked new research on file systems [10, 15, 18, 24], database systems [11, 13, 20, 27, 53], and persistent programming and data structures [9, 14, 52, 58].

Ensuring recovery requires constraining the order in which STOREs attain persistence. If memory is persistent but CPU caches are not, carefully synchronized cache flushing must be used to ensure that STOREs reach memory in the desired order [5]. Intel has recently announced ISA extensions to optimize cache flushes for systems with persistent memory [41]. Pelley et al. recently introduced memory persistency models that build on memory consistency models [1] to provide guarantees on the order in which STOREs become persistent [37].

The different NVM device technologies offer different read/write/endurance characteristics and are expected to be deployed accordingly in future systems. For example, while PCM and Memristor are mainly considered as candidates for main memory, STT-RAM is expected to be used in caches [61]. Non-volatile caches imply that STOREs become persistent upon leaving the CPU's STORE buffers. Persistent caches can also be implemented by relying on stand-by power [32–34] or employing supercapacitor-backed volatile caches to flush data from caches to persistent memory in the case of a failure [53]. Recent trends show that non-volatile caches are possible in the near future [53].

Software for Persistence Disk-based database systems have traditionally used write-ahead logging to ensure consistent recoverability [31]. Changing workloads, data models,

and storage media have inspired numerous refinements and specializations represented by systems such as eNVy [55], Berkeley DB [35], Stasis [44], MARS [13], and Rio Vista [27]. Proper transactional updates to files in file systems can simplify complex and error-prone procedures such as software upgrades. Transactional file updates have been explored in research prototypes [36, 47]; commercial implementations include Microsoft Windows TxF [30] and Hewlett Packard Enterprise AdvFS [50]. Transactional file update is readily implementable atop more general operating systems transactions, which offer additional security advantages and support scenarios including on-the-fly software upgrades [39]. At the opposite end of the spectrum, user-space implementations of persistent heaps supporting failure-atomic updates have been explored in research [59] and incorporated into commercial products [6]. Logging-based mechanisms in general ensure consistency by discarding changes from an update interrupted by failure. In contrast, for idempotent updates, the update cut short by failure can simply be re-executed rather than discarding changes, reducing required logging (similar to [4, 21]).

Persistent memory has inspired research on programming interfaces with durability semantics. Kiln [61] and Lu et al. [28] describe mechanisms that provide atomicity and durability guarantees for systems with persistent caches and persistent memory, however, the programmer remains responsible for ensuring isolation. Whole-system persistence [32], a combined hardware and software technique, gives consistency guarantees for power failures, but does not tolerate software errors. Mnemosyne [52] and NV-Heaps [14] extend transactional memory to provide durability guarantees for ACID transactions on persistent memory. Mnemosyne emphasizes performance; the use of REDO logs, for example, postpones the need to flush data to persistence to a single end-of-transaction batch. NV-heaps, an UNDO log system, emphasizes programmer convenience, providing garbage collection and strong type checking to help avoid pitfalls unique to persistent memory, e.g., pointers to transient data inadvertently stored in persistent memory. Given the ubiquity of lock-based concurrency control, Atlas [9] provides durability guarantees for mutex-based FASEs. Like JUSTDO logging, Atlas guarantees failure atomicity for FASEs defined as outermost critical sections (see Figure 1). Mnemosyne, NV-Heaps, and Atlas can all tolerate abrupt power failures, non-corrupting kernel panics, and non-corrupting process crashes; they do not tolerate corrupting software errors because the latter may corrupt persistent application data directly or may corrupt logs needed for recovery.

The Atlas system, which we compare against in our evaluation (Section 7), illustrates the tradeoffs among convenience, compatibility, generality, and performance that confront any implementation of FASEs. Atlas employs per-thread UNDO logging to ensure the atomicity of FASEs. An UNDO log entry is created for every STORE to persistent memory that is executed by a thread in a program. The log entry must be made persistent before the corresponding STORE can occur. Unlike the isolated transactions of NV-heaps and Mnemosyne, the outermost critical sections that constitute Atlas FASEs may be linked by dependencies: Sometimes an outermost critical section that has completed must nonetheless be rolled back during recovery. Reclaiming UNDO log entries no longer needed for recovery is therefore a non-trivial task in Atlas and is performed in parallel by a separate helper thread. Because dependencies between FASEs must be explicitly tracked, Atlas requires persistent memory updates to be synchronized via locks, which precludes the use of `atomics` (in the sense of C++11) in the current version of Atlas. Atlas emphasizes generality, programmer convenience, and compatibility with conventional lock-based concurrency control; a sophisticated infrastructure is required to support these advantages, and the requisite UNDO logging and log pruning carry performance overheads. Specifically, the size of UNDO logs is proportional to the amount of data modified in the corresponding FASE, and the complexity of tracking dependencies for log reclamation can grow with the number of FASEs.

4. System Model & Programming Model

System Model Figure 2 illustrates our system model. As in prior work [28, 61], we consider a system in which both main memory and processor caches are persistent, i.e., their contents survive tolerated failures intact. We place no restrictions on how persistent memory and persistent caches are implemented. A tolerated failure on such a system causes all processor state to be lost but the contents of the caches and memory survive and are available upon recovery. We assume that power failures and non-corrupting fail-stop software failures have these consequences.

If caches are persistent, a STORE will become persistent once it reaches the coherence layer; release fences force the STORE into persistence. By comparison, on an x86 system with persistent memory but without persistent caches, STOREs can be pushed toward persistence using a `CLFLUSH`. On future Intel systems, new flushing instructions such as `CLFLUSHOPT` and `CLWB` will provide fine-grained control over persistence with lower overhead [16]. Flushing instructions will be used with `SFENCE` and `PCOMMIT` to constrain persistence ordering.

Programming Model JUSTDO logging leverages mutex-based concurrency control to infer FASEs. We assume that failure-atomic modifications to shared data in persistent memory are performed in critical sections delimited by lock acquisitions and releases: A thread that holds one or more locks may temporarily violate application-level consistency invariants, but all such invariants are restored before the thread releases its last lock (Atlas makes the same assumption [9]). Therefore, data in persistent memory are con-

sistent in the quiescent state in which no thread holds any locks, and we accordingly equate outermost critical sections with FASEs: JUSTDO logging guarantees that a quiescent (and thus consistent) state is restored following a failure. At FASE exit, all modifications are guaranteed to be persistent, therefore JUSTDO applications may safely emit output dependent on a FASE—e.g., acknowledging to a remote client that a transaction has completed—immediately after exiting the FASE. Aside from standard lock-mediated data accesses, JUSTDO logging supports unsynchronized read-write (but not write-write) races, which are condoned for C++ `atomics`. Although JUSTDO's approach to failure atomicity is designed for concurrency, it can be adapted to serial code by simply delimiting FASEs as in parallel software.

As in prior approaches [9, 14, 52] our technique allows the programmer to specify explicitly which data is to be preserved across failure by placing it in persistent memory; such control is useful for, e.g., hybrid architectures that incorporate both DRAM and NVM. In such partially persistent systems that expose both persistent and transient memory to applications, JUSTDO logging requires that FASEs access only persistent memory.

Our current implementation of JUSTDO logging is a C++ library with bindings for C and C++. The library requires FASEs to reside in functions that consolidate boilerplate required for recovery and requires that STOREs occur via special JUSTDO library calls. Future compiler support could eliminate nearly all of the verbosity that our current prototype requires and could eliminate opportunities for incorrect usage.

As in prior implementations of persistent memory FASEs, JUSTDO logging maintains logs crucial to recovery in the address space of a running application process. Application software bugs or OS bugs that corrupt the logs or the application's data in persistent memory cannot be tolerated by any of these approaches, but detectable non-corrupting software failures can be tolerated. The main difference between JUSTDO logging and the earlier approaches to persistent memory FASEs is that JUSTDO logging does not tolerate software failures *within* FASEs: Our approach of resuming execution at the point of interruption is inappropriate for such failures, and our approach does not have the ability to roll back a FASE interrupted by failure.

Use Cases Two widespread and important use cases, which we call *library-managed persistence* and *mandatory mediated access*, are well suited to the strengths and limitations of JUSTDO logging and its synergies with complementary technologies for fine-grained fault isolation and software quality assurance techniques.

JUSTDO logging can provide the foundation for high-performance thread-safe libraries that manage persistent data structures on behalf of application logic. In such scenarios, exemplified today by SQLite [45] and similar software, the library assumes responsibility both for orderly concurrent access to shared data in persistent memory and for recovering persistent memory to a consistent state following failures. JUSTDO logging enables expert library developers to write lock-based FASEs in library routines and employ JUSTDO logging to ensure consistent recoverability with low overhead during failure-free operation. A well-designed JUSTDO-based library will consolidate persistent data updates in small FASEs that lend themselves readily to powerful software quality assurance techniques [7, 8, 19]. One advantage of this approach is that JUSTDO logging is employed only in the expert-written library and is not visible to application developers; one limitation of this use case is that FASEs cannot span library calls.

A related use case involves application logic of questionable quality or security that must be constrained to manipulate valuable persistent data only indirectly, via a trusted high-quality intermediary. A widespread example of this pattern is commercial relational database management systems, which mediate application access to database tables while upholding data integrity constraints and preventing arbitrary modifications of the database by buggy, misguided, or malicious application logic. JUSTDO logging provides a new high-performance logging strategy for the intermediary in mandatory mediated access scenarios. OS process boundaries coupled with user permissions can isolate untrusted application code from trusted intermediary software, allowing only the latter direct access to persistent data. However this isolation strategy, widely used today in high-integrity database configurations, requires application logic to communicate with trusted code via heavyweight inter-process communication (IPC) mechanisms. Research on fine-grained intra-process isolation [12], together with JUSTDO logging, suggests a lightweight alternative: Application logic accesses persistent data via a *library* linked into the same address space as the application, precisely as in the library-managed persistence scenario, but with a crucial difference: The intra-process isolation mechanism protects both the data and the trusted library from untrusted application code. Such a strategy eliminates the overhead of IPC between application code and the trusted intermediary without weakening protection.

5. Design

JUSTDO logging implements lock-delimited failure-atomic sections (FASEs) by recording sufficient information during the execution of a FASE such that, if a crash occurs, each FASE can resume at the last STORE it attempted prior to the failure.

The key data structure for our technique is the JUSTDO log, a small per-thread log. This thread-local log contains only a *single active entry* at any one time, and is written before every STORE within a FASE. The single active log entry contains only the *address* to be written, the *new value* to be written there, the *size* of the write, and the *pro-*

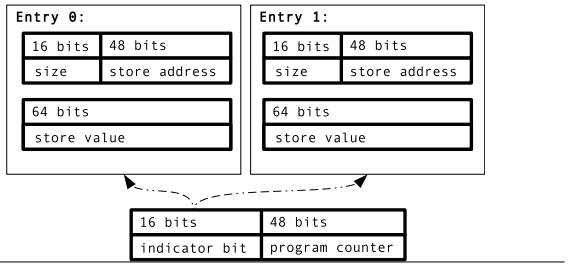

Figure 3. JUSTDO log format.

gram counter. Immediately after the log entry is completed, the corresponding STORE is performed. Conceptually, the JUSTDO log is a "done to here" note; recovery code leverages the idempotence of STOREs to transform at-least-once execution of the most recent STORE before a failure into correct resumption.

To recover using a crashed program's set of per-thread JUSTDO logs, recovery threads re-enter each interrupted FASE at the program counter indicated in the FASE's JUSTDO log, re-acquire the appropriate locks, re-execute the idempotent STORE, and continue execution until the end of each FASE.

Successful recovery requires additional steps when writing a JUSTDO FASE. In particular, we must ensure that the instructions in a FASE do not access data that was stored in transient memory, which will not have survived the failure. We satisfy this requirement by mandating that all LOADs and STOREs within a FASE access only persistent memory. Furthermore, we must ensure that instructions in a FASE do not depend on data held only in volatile CPU registers. We satisfy this requirement by preventing register promotion [26, 43] of memory values within FASEs. Finally, the recovery-time completion of each FASE must respect all mutual exclusion constraints present in the program. We ensure this by recording the locks acquired and released in each FASE in thread-local lock logs.

This section describes the design of the JUSTDO log and our auxiliary data structures. For brevity we largely omit release fences from our discussion. We employ release fences as necessary to constrain the order in which STOREs attain persistence.

5.1 JUSTDO Log

The JUSTDO log is updated for every STORE within a FASE. Our approach transforms every STORE in the original crash-vulnerable FASE to both a log update and then the STORE in the JUSTDO-fortified FASE.

Figure 3 illustrates the format of the entire thread-local JUSTDO log. The log is implemented as a tightly packed struct where each field holds critical recovery information. To ensure atomic updates to the log, it actually holds two entries, although only one is active at a time. In each entry, we store the destination address, size, and new value. The program counter value is shared between the entries, and we use the high order bits of the program counter to indicate which entry is active. On Intel x86, virtual addresses are 48 bits, facilitating this tight packing [16]. Additional bits in the size field and indicator bit are reserved for future use (e.g., flags indicating an atomic store).

To update the log, both the new value and destination address are STOREd (with the size packed into the high order bits of the address pointer) in the inactive entry, followed by a release fence to ensure that the writes have reached the persistent cache. Subsequently, we STORE the new program counter (with the indicator bit set for the recently updated entry).

After the log has been successfully updated, we execute a release fence (again to ensure that the updates are persistent), then complete the persistent STORE by writing the new value to the destination address.

5.2 Persistent-Only Accesses

We require that all memory LOADs and STOREs within FASEs access only persistent data. This requirement extends to thread-local locations that would ordinarily be transient, such as variables on the stack. By mandating that FASEs can access only persistent data we ensure that no updates in a FASE are dependent on state destroyed by failure.

The persistent-only access requirement means that any thread-local memory locations that might be accessed in the FASE (including those normally stored on the stack) must be moved to persistent memory prior to entering the first critical section of a FASE, and, if desired, moved out of persistent memory at the end of a FASE.

While this "persistent-only access" restriction may appear limiting, we find that it is compatible with familiar design patterns. Consider, for example, the ubiquitous "container" pattern as applied to persistent data: (nearly) all of the container metadata maintained by the library code is persistent; similarly, the data stored in a persistent container is also persistent. User code will ensure that its (possibly large) data values are persistent before passing pointers to them into the library; the library can verify that the data are in persistent memory via range checking. It is straightforward to migrate into persistent memory the relatively small amount of transient data passed on the stack between client and library (e.g., the pointer to persistent data). Unlike physical logging-type systems, our technique only requires the data to be written to persistence once, and is consequently insensitive to data size (see Section 7.4). The "small transient state property" is typical of the exported methods of shared persistent data structures and their maintenance operations (e.g., rebalancing).

5.3 Register Promotion in FASEs

Register promotion is a compiler optimization that eliminates redundant LOADs from memory by caching memory locations in CPU registers [26, 43]. Register promotion in FASEs is problematic for JUSTDO logging. Consider a value

in persistent memory that within a FASE is LOADed into a register upon which two subsequent STOREs depend. If, due to register promotion, the value is not re-LOADed from persistent memory prior to influencing the second STORE, recovery from a crash immediately after the first STORE is impossible: The crash erases the register containing the value upon which the second STORE depends.

Our current implementation prevents such anomalies by selective use of the C/C++ "`volatile`" keyword. We employ a templated type wrapper within FASEs to ensure that LOADs within FASEs occur via `volatile` pointers and are therefore not elided by compiler optimization. Note that STOREs are not affected by this LOAD-specific mechanism. Manual inspection of the assembly code generated for our FASEs confirms that our current approach prevents problematic register promotions without affecting STOREs.

The penalty of disabling register promotion within FASEs in our current prototype is that FASE execution time is roughly doubled, i.e., crash-resilient JUSTDO-fortified critical sections are twice as slow as crash-vulnerable transient critical sections. Compared to the logging overheads of alternative failure atomicity mechanisms, our overhead is lower (Section 7). In the future, a JUSTDO-aware compiler could more selectively disable register promotion in FASEs, allowing it where it does not preclude recovery, thus improving performance.

5.4 Lock Logs

Recovering from failure by resumption requires that every recovery thread know which locks it holds, and furthermore that no locks are held forever; otherwise safety violations or deadlock can occur. Our design supports arbitrary lock implementations; our current prototype employs standard pthread mutexes.

To preserve lock ownership information across crashes, we require that locks reside in a persistent memory region. Threads maintain two per-thread persistent logs to facilitate proper restoration of lock ownership during recovery: a *lock intention log* and a *lock ownership log*. The purpose of the former is to speed recovery by obviating the need to inspect all locks in persistent memory, whereas the latter is used to reassign locks to recovery threads. The size of a thread's lock ownership and lock intention logs at a particular point in time is proportional to the number of locks held by the thread.

Immediately prior to attempting a lock acquisition, a thread declares its intent by recording the lock address in the lock intention log. Immediately after acquiring the lock, the thread records the acquisition in the lock ownership log using a JUSTDO store. To unlock a mutex, a thread performs the same operations in reverse order: It first uses a JUSTDO store to remove the lock from its lock ownership log, then unlocks the mutex, and finally removes the lock from the lock intention log. This protocol ensures that following a crash the per-thread lock intention logs collectively record all locks that *might* be locked, and the lock ownership logs record which thread has locked each lock that is *certainly* locked.

5.5 Recovery

Recovery begins by using the per-thread lock intention logs to *unlock* all mutexes that might have been locked at the moment of failure. Without lock intention logs, unlocking all mutexes would require inspecting them all or using generational locks in the manner of NV-heaps [14]. The lock intention log enables both arbitrary mutex implementations and fast recovery.

After unlocking all mutexes, recovery spawns one thread per non-empty JUSTDO log; a recovery thread's duty is to execute to completion a corresponding FASE that had been cut short by failure. Each recovery thread inherits a JUSTDO log and the pair of lock logs left behind by its deceased predecessor.

Recovery threads begin by acquiring all locks in their respective lock ownership logs, then waiting at a barrier for all other threads to do likewise. Once all locks have been acquired by all recovery threads, each thread re-executes the STORE instruction contained in its JUSTDO log. Finally, each recovery thread jumps to the program counter value contained in the JUSTDO log and continues execution of the interrupted FASE. Recovery threads track the number of mutexes they hold, and when this count drops to zero the FASE has been completed and the thread exits.

Interestingly, recovery *must* be executed with an interleaving of instructions (either in parallel or by context switching across recovery threads): Some FASEs may be blocked waiting for other FASEs to release mutexes. This interleaving requirement is actually an advantage, because our approach naturally supports parallel recovery. Furthermore, once our recovery threads have re-acquired all of their locks and passed the barrier, access to shared persistent state is properly synchronized by the appropriate mutexes. Consequently, the resurrected application may spawn ordinary (non-recovery) threads that operate, with appropriate synchronization, upon persistent memory *even before our recovery threads have completed the execution of interrupted FASEs.* In other words, the restoration of consistency to persistent memory can proceed in parallel with resumed application execution. Section 7.3 presents recovery time measurements of crashed processes that manipulated large volumes of persistent data via JUSTDO logging.

Reasoning about the barrier employed by recovery makes it easy to show that our approach tolerates failures during recovery. No persistent memory state is altered before our recovery threads reach the barrier, so a crash before this point has no effect and recovery may simply be attempted again. After our recovery threads pass the barrier, they execute FASEs under the protection of JUSTDO logging, precisely as in an ordinary execution of the program.

6. Implementation

Our current JUSTDO logging prototype is a C++ library with bindings for both C++ and C. Annotations for JUSTDO-enabled FASEs are a straightforward, if tedious, transformation of transient (non-crash-resilient) code, somewhat analogous to the annotations employed in software transactional memory systems. We hope that future work, integrating compiler support, can automate nearly all of the chores surrounding annotations while also providing additional type safety guarantees to ensure that the "persistent-only accesses" rule is followed within FASEs. In the meantime, however, JUSTDO integration requires every possible code path and access within a FASE to be identified and annotated at compile time, making JUSTDO integration significantly more complex than other failure atomicity systems. Other systems, such as Atlas, do not need to know all possible FASE code paths at compile time. Compared with prior FASE implementations, our current prototype deliberately trades programmer convenience and generality for performance.

Our library contains three major elements: the `jd_root`, the `jd_obj`, and the JUSTDO routine. The first two are C++ classes and are entry points into our library. The JUSTDO routine consolidates the boilerplate required to execute an application-defined FASE under JUSTDO logging. The remainder of this section illustrates the use of these elements in a detailed example shown in Figures 4, 5, 6, and 7. Our example code failure-atomically transfers money from `acnt1` to `acnt2`; for clarity we omit type casts and the use of the `volatile` keyword. Our example code shows the usage of JUSTDO annotations and how to set up a JUSTDO FASE.

By definition, persistent memory outlives the processes that access it. Therefore JUSTDO logging requires mechanisms to enable newly created processes to locate persistent memory containing data of interest and to make the data accessible to application software. At a high level, we follow the same straightforward approach taken by prior research implementations of FASEs and by emerging industry standards for persistent memory [38, 46]: A file system (or the moral equivalent thereof) maps short, memorable, human-readable strings (names) to long persistent byte sequences, and processes use an `mmap`-like interface to incorporate into their address spaces the persistent data thus located. More specifically, our JUSTDO logging prototype uses the Atlas [9] implementation of persistent memory regions, which supports memory allocation methods `nv_malloc` and `nv_calloc` and contains a header for its root pointer (accessed via `Get/SetRegionRoot` methods), as shown in our example code.

6.1 `jd_root`

The `jd_root` object is the main entry point to the JUSTDO library. This object is placed in a well-known location in the persistent region that is accessible by recovery code via `GetRegionRoot`.

The `jd_root` is a global object and is the factory object for `jd_objs`, which are thread-local. The `jd_root` maintains a list of the `jd_objs` that have been allocated to threads.

During recovery, the `jd_root` object is responsible for unlocking all mutexes and coordinating lock re-acquisitions across recovery threads. Finally, it initiates thread-local recovery, in which recovery threads jump back into their respective FASEs.

6.2 `jd_obj`

The `jd_obj` is a thread local object for executing a FASE under JUSTDO logging. It contains both the JUSTDO log structure and its associated lock logs. `jd_obj` exports methods `jd_lock`, `jd_store`, and `jd_unlock`; consequently most lines within a JUSTDO FASE will touch the `jd_obj`.

The `jd_obj` also provides a handle to thread-local persistent memory that is used to persist variables normally on the stack; this handle facilitates compliance with the "persistent-only access" rule of Section 5.2. In an *exception* to the "persistent-only access" rule, each thread maintains a *reference* to its `jd_obj` on the stack. Following a crash, this reference is correctly re-set in each recovery thread. This exception allows a recovery thread to share a reference to its `jd_obj` with its failed predecessor.

6.3 JUSTDO routine

A JUSTDO routine is a function containing a JUSTDO FASE. Such functions have a defined prototype and are annotated to enable recovery. During recovery, the JUSTDO routine's stack frame provides thread-local scratch space that would be inconvenient to obtain otherwise. The annotations are illustrated in our example code at line 10 in `transfer_justdo` of Figure 5.

A JUSTDO routine complies with several annotation requirements. It takes three arguments: a `jd_obj` and two `void` pointers for the arguments and return values. We also require that the first line of the JUSTDO routine be a special macro: `JD_ROUTINE_ON_ENTRY` (line 12).

There are two ways to execute a JUSTDO routine, corresponding to normal (failure-free) execution and recovery. During failure-free operation, invocation of a JUSTDO routine simply executes the function (and FASE) as written.

During recovery, however, the execution of a JUSTDO routine is different. A recovery thread that has acquired mutexes as described in Section 5.5 invokes the JUSTDO routine, passing as an argument a reference to the the `jd_obj` that it inherits from its failed predecessor thread and NULL for the remaining two arguments, `args` and `rets`. The `JD_ROUTINE_ON_ENTRY` macro in the JUSTDO routine determines from the `jd_obj` that it is running in recovery mode and uses the JUSTDO log within the `jd_obj` to cause control to jump to the last STORE within the FASE executed prior to failure. When a recovery thread unlocks its last mutex, it

```
1  struct Root { int* accounts;
2                lock* locks;
3                jd_root* jdr; };
4  Root* rt;
5  struct Args { int acnt1,acnt2,amount; };
6  struct Returns { bool success; };
7  struct Locals { int acnt1, acnt2;
8                bool success; };
```

Figure 4. JUSTDO logging example (Globals)

knows that its assigned FASE has completed and therefore it exits.

6.4 Recovery Implementation

Having introduced all library constructs and their design, we can now summarize the entire recovery procedure:

1. The application detects a crash and invokes JUSTDO recovery via `jd_root`.
2. The `jd_root` resets all locks using the lock intention logs.
3. The `jd_root` spawns recovery threads for every active `jd_obj`.
4. Each recovery thread re-acquires locks using its lock ownership logs in its `jd_obj`, then barriers.
5. Following the barrier, the recovery threads invoke interrupted JUSTDO routines with their inherited `jd_obj`.
6. Each recovery thread uses the `JD_ROUTINE_ON_ENTRY` macro to jump to the program counter as indicated by its JUSTDO log.
7. When a recovery thread's lock count reaches zero, it exits.

```
9  // jd_routine for account transfer
10 void transfer_justdo(jd_obj* jdo,
11  void* args, void* rets){
12   JD_ROUTINE_ON_ENTRY(jdo);
13   // copy locals off the stack
14   jdo->set_locs<Locals>();
15   jdo->locs->acnt1 = args->acnt1;
16   jdo->locs->acnt2 = args->acnt2;
17   jdo->locs->amount = args->amount;
18   // begin FASE
19   jdo->jd_lock(
20    rt->locks[jdo->locs->acnt1]);
21   jdo->jd_lock(
22    rt->locks[jdo->locs->acnt2]);
23   // increment first account
24   jdo->jd_store(
25    &rt->accounts[jdo->locs->acnt1],
26    rt->accounts[jdo->locs->acnt1] +
27    jdo->locs->amount);
28   // decrement second account
29   jdo->jd_store(
30    &rt->accounts[jdo->locs->acnt2],
31    rt->accounts[jdo->locs->acnt2] -
32    jdo->locs->amount);
33   // end FASE
34   jdo->jd_unlock(
35    rt->locks[jdo->locs->acnt1]);
36   jdo->jd_unlock(
37    rt->locks[jdo->locs->acnt2]);
38   // outside FASE, can access transient
39   rets->success = true;
40 }
```

Figure 5. JUSTDO logging example (JUSTDO Routine)

7. Experiments

In the current state of our prototype it is difficult to retrofit JUSTDO logging onto large and complex real-world legacy applications, so our evaluations employ small benchmarks. We implemented five high-throughput concurrent data structures to evaluate the performance and recoverability of JUSTDO logging. Each data structure is implemented in three variants: a *Transient* (crash vulnerable) version, a JUSTDO crash-resilient version, and a version fortified with the Atlas crash-resilience system [9]. The five algorithms are the following:

Queue The two-lock queue implementation of Michael and Scott [29].

Stack A locking variation on the Treiber Stack [49].

Priority Queue 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 sorted-list priority queue implementation for each bucket, obviating the need for per-bucket locks.

Vector An array-based resizable vector in the style of the contiguous storage solution proposed by Dechev et al. [17]. This algorithm supports lookups and updates during resizing.

The queue and stack are lock-based implementations of algorithms in the `java.util.concurrent` library. The vector's design allows it to exploit atomic STORE instructions, and our transient and JUSTDO versions of the vector take advantage of this feature. Atlas supports only mutex-based synchronization and consequently our Atlas version of the vector uses a reader-writer lock instead, which incurs a non-negligible performance overhead. In all other respects, the

```
41  int main(){
42    int rid =
43     LoadPersistentRegion("my_region");
44    rt = GetRegionRoot(rid);
45    // initialize our root if needed
46    if(rt == NULL) {
47      rt = nv_malloc(sizeof(Root),rid);
48      rt->accounts =
49       nv_calloc(sizeof(int),N_ACCTS,rid);
50      rt->locks =
51       nv_calloc(sizeof(lock),N_ACCTS,rid);
52      rt->jdr =
53       nv_malloc(sizeof(jd_root),rid);
54      new(rt->jdr) justdo_root(jdr);
55      SetRegionRoot(rt,rid);
56    }
57    // otherwise recover if needed
58    else{rt->jdr->recover();}
59    // get a thread local jd_obj
60    jd_obj* jdo = rt->jdr->new_jd_obj();
61    // conduct transfer
62    Args args;
63    args.acnt1 = 3;    // arguments passed
64    args.acnt2 = 5;    // into FASE
65    args.amount = 50; // via jd_routine
66    Returns rets;
67    transfer_justdo(jdo,args,rets);
68    // delete jd_obj
69    rt->jdr->delete_jd_obj(jdo);
70  }
```

Figure 6. JUSTDO logging example (main)

```
71  // The equivalent transient routine
72  bool transfer_transient(int acnt1,
73   int acnt2, int amount){
74    lock(rt->locks[acnt1]);
75    lock(rt->locks[acnt2]);
76    accounts[acnt1] += amount;
77    accounts[acnt2] -= amount;
78    unlock(rt->locks[acnt1]);
79    unlock(rt->locks[acnt2]);
80    return true;
81  }
```

Figure 7. JUSTDO logging example (equivalent transient routine)

three versions of each of our five data structures differ only in the implementation—or non-implementation—of crash resilience.

Note that our implementations of these data structures admit parallelism to varying degrees. Our stack, for example, serializes accesses in a very small critical section. At the other extreme, our hash map admits parallel accesses both across and within buckets. We therefore expect low-parallelism data structures to scale poorly with worker thread count whereas high-parallelism data structures should exhibit nearly linear performance scaling.

7.1 Correctness Verification

Conventional hardware suffices for the purposes of verifying the crash-resilience guarantees of JUSTDO logging because both conventional CPU caches and conventional DRAM main memory can be *persistent with respect to process crashes*: Specifically, STOREs to a file-backed memory mapping are required by POSIX to be "kernel persistent," meaning that such STOREs are guaranteed to outlive the process that issued them; neither `msync` nor any other measures are required after a STORE to obtain this guarantee [33].

To test JUSTDO recovery we installed a 128 GB JUSTDO-fortified hash map in a file-backed memory mapping on a server-class machine (described in more detail in Section 7.2). After building the hash table, we used all sixty of the server's hardware threads to perform inserts and removes in equal proportion on random keys in the hash table. Our hash buckets are implemented as sorted linked lists, so corruption (if present) will manifest as dangling pointers within a bucket, resulting in a segmentation fault or assertion failure. At intervals of two seconds, we killed the threads using an external `SIGKILL`. On restarting the process, we performed JUSTDO recovery before continuing execution. This test was conducted for approximately four hours. We constructed similar tests for each of our other four concurrent data structures; these additional tests also injected crashes every two seconds and ran for over twelve hours. No inconsistencies or corruption occurred.

7.2 Performance Evaluation

The goal of our performance evaluation is to estimate the overhead of JUSTDO crash resilience compared with high-performance transient (crash-vulnerable) versions of the same concurrent algorithms. We took care to ensure that the transient versions of our five algorithms exhibit quite good performance; these versions provide reasonable performance baselines. For example, running on newer hardware, our transient hash map achieves per-core throughput approaching published results on the state-of-the-art MICA concurrent hash table [25].

Our results are conservative/pessimistic in the sense that our experiments involve small data-intensive microbenchmarks that magnify the overheads of crash resilience to the greatest possible extent. In real applications, concurrent ac-

cesses to shared persistent data structures might *not* be a performance bottleneck, and therefore by Amdahl's law [2] the overheads of any crash resilience mechanism would likely be smaller. This effect is shown in Section 7.4, where the overhead of initializing large data values eliminates the overhead of persistence.

Our tests consist of microbenchmarks with a varying number of worker threads. 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. Unless stated otherwise (see Section 7.4), keys and values are eight-byte integers, are generated in transient memory, and are copied into persistent memory upon entry into a FASE. For our evaluations of queues, stacks, and priority queues, threads choose randomly between `insert` or `remove`; these three data structures were sized such that most accesses were served from the CPU caches. Therefore performance for our stack and queues is limited by synchronization.

Our vector and map evaluations drew inspiration from the standard YCSB benchmark [57]. For vectors and maps, the containers are filled to 80% of the key range, then we perform `overwrite` operations for random keys in the range. The `overwrite` operation replaces the value only if it exists, but otherwise does not modify the data structure. We sized our vectors and maps so that the vast majority of these two structures did *not* fit in the CPU caches; keys for accesses were drawn randomly from a uniform distribution. Most accesses miss in the CPU caches, therefore our vector and map are limited by memory performance.

During each test, threads synchronize only through the tested data structure. To smooth performance curves, pages are prefaulted to prevent soft page faults. For data structures with high memory allocator usage (all except the vector), we implemented a simple thread-local bump pointer block pool to prevent bottlenecking on `malloc` and to minimize the impact of Atlas's custom memory allocator, which tends to underperform at high thread counts. Variables within the data structures are appropriately padded to prevent false sharing. To generate random numbers, threads use thread-local generators to avoid contention.

Software threads for all experiments are pinned to specific hardware threads. Our thread pinning progression fills all cores of a socket first, then fills the corresponding hyperthreads. Once all cores and hyperthreads are filled, we add additional sockets, filling them in the same order. For all machines, we ran every experimental configuration five times and took the average.

Compilation for the transient and JUSTDO data structures was done using gcc 4.8.4 with the `-O3` flag. Atlas structures were compiled using the clang and llvm-based Atlas compiler, again with the `-O3` flag set.

"Persistent Cache" Machines We conducted performance tests on three machines. The first is a single-socket workstation with an Intel Core i7-4770 CPU running at 3.40 GHz. The CPU has four two-way hyperthreaded cores (eight hardware threads). It has a shared 8 MB L3 cache, with per-core private L2 and L1 caches, 256 KB and 32 KB respectively. The workstation runs Ubuntu 12.04.5 LTS.

Our second machine is a server with four Intel Xeon E7-4890 v2 sockets, each of which has 15 cores (60 hardware threads total). The machine has 3 TB of main memory, with 37.5 MB per-socket L3 caches. L2 and L1 caches are private per core, 256 KB and 32 KB respectively. The server and the workstation are used to mimic machines that implement persistent memory using supercapacitor-backed DRAM (e.g., Viking NVDIMMs [51]) and supercapacitor-backed SRAM.

Figures 8 and 9 show aggregate operation throughput as a function of worker thread count for all three versions of our data structures—transient, JUSTDO-fortified, and Atlas-fortified. Our results show that on both the workstation and the server, JUSTDO logging outperforms Atlas for every data structure and nearly all thread counts. JUSTDO performance ranges from three to one hundred times faster than Atlas. JUSTDO logging furthermore achieves between 33% and 75% of the throughput of the transient (crash-vulnerable) versions of each data structure. For data structures that are naturally parallel (vector and hash map), the transient and JUSTDO implementations scale with the number of threads. In contrast, Atlas does not scale well for our vectors and maps. This inefficiency is a product of Atlas's dependency tracking between FASEs, which creates a synchronization bottleneck in the presence of large numbers of locks.

Future NVM-based main memories that employ PCM or resistive RAM are expected to be slower than DRAM, and thus the ratio of memory speed to CPU speed is likely to be lower on such systems. We therefore investigate whether changes to memory speed degrade the performance of JUSTDO logging. Since commodity PCM and resistive RAM chips are not currently available, we investigate the implications of changing memory speed by under-clocking and over-clocking DRAM. For these experiments we use a third machine, a single-socket workstation with a four-core (two-way hyperthreaded) Intel i7-4770K system running at 3.5 GHz with 32 KB, 256 KB private L1 and L2 caches per core and one shared 8 MB L3 cache. We use 32 GBs of G.SKILLs TridentX DDR3 DRAM operating at frequencies of 800, 1333 (default), 2000, and 2400 MHz.

For our tests involving small data structures (queue, stack, and priority queue), the performance impact of changing memory speed was negligible (under 1% change)—which is not surprising because by design these entire data structures fit in the L3 cache. For our tests involving larger data structures deliberately sized to be far larger than our CPU caches and accessed randomly (map and vector), we find that the ratio of transient (crash-vulnerable) throughput to

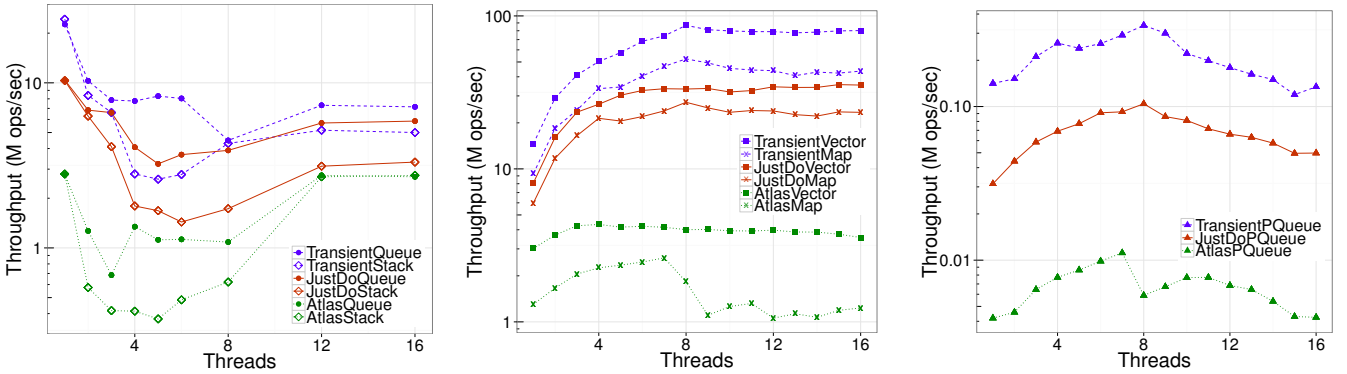

Figure 8. Throughput on workstation (log scale)

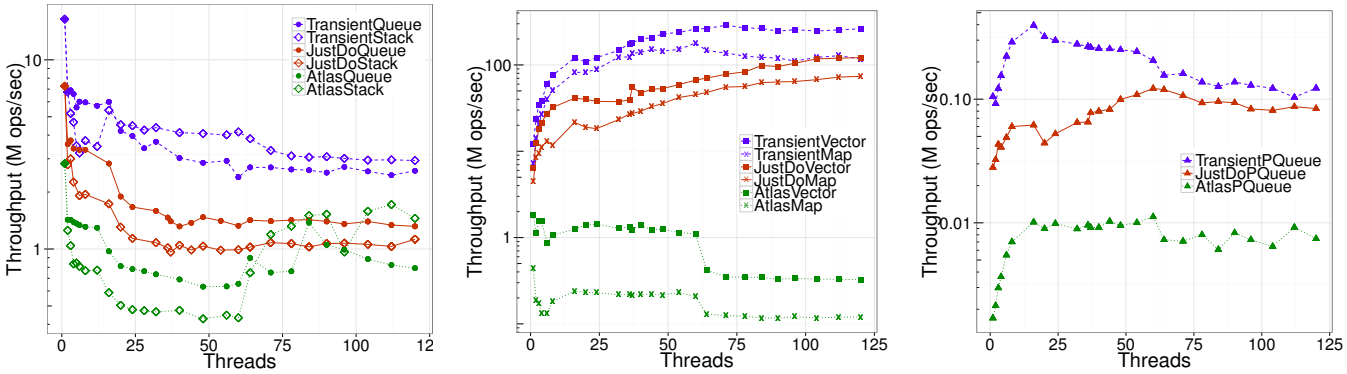

Figure 9. Throughput on server (log scale)

JUSTDO logging throughput remains constant (between 1.93 and 2.00) as memory speed varies over a 3× range. Varying memory speed does not change the overhead of of JUSTDO logging.

"Transient Cache" Machines To investigate how JUSTDO logging will likely perform on machines without persistent caches, but with persistent main memory, we modified our JUSTDO library to use the synchronous `CLFLUSH` instruction to push STOREs within FASEs toward persistent memory. This x86 instruction invalidates and writes back a cache line, blocking the thread until it completes. While Intel has announced faster flushing mechanisms in future ISAs [41], this instruction is the only method on existing hardware. Our `CLFLUSH`-ing version uses the `CLFLUSH` instruction where before it used only a release fence, forcing dirty data back to persistent storage in a consistent order.

We performed `CLFLUSH` experiments on our i7-4770 workstation and compared with Atlas's "flush-as-you-go" mode, which also makes liberal use of `CLFLUSH` in the same way (see Figure 10). As expected, JUSTDO logging takes a serious performance hit when it uses `CLFLUSH` after every STORE in a FASE, since the reduced computational overhead of our technique is overshadowed by the more expensive flushing cost. Furthermore, the advantage of a JUSTDO log that fits in a single cache line is negated because the log is repeatedly invalidated and forced out of the cache. The cache line invalidation causes a massive performance hit. For the JUSTDO map using four worker threads, the L3 cache miss ratio increases from 5.5% to 80% when we switch from release fences to `CLFLUSH`es. We expect that the new Intel instruction `CLWB`, which drains the cache line back to memory but does not invalidate it, will significantly improve our performance in this scenario when it becomes available.

In contrast to JUSTDO logging, Atlas's additional sophistication pays off here, since it can buffer writes back to memory and consolidate flushes to the same cache line. Atlas outperforms the JUSTDO variants by 2–3× across our tested parameters on "transient cache" machines.

7.3 Recovery Speed

In our correctness verification test (Section 7.1), which churned sixty threads on a 128 GB hash table, we also recorded recovery time. After recovery process start-up, we spend on average 2000 microseconds to `mmap` the large hash table back into the virtual address space of the recovery process. Reading the root pointer takes an additional microsecond. To check if recovery is necessary takes 64 microseconds. In our tests, an average of 24 FASEs were interrupted by failure, so 24 threads needed to be recovered. It took

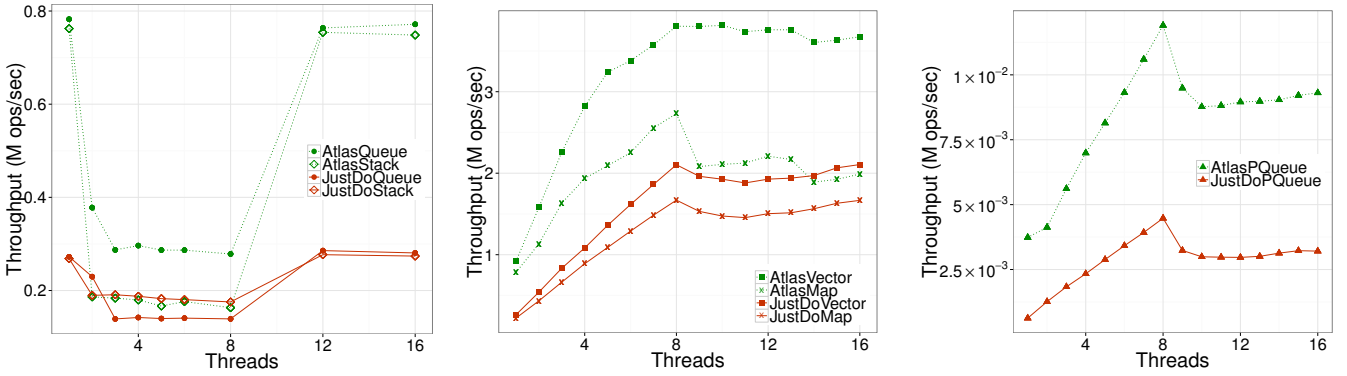

Figure 10. Throughput on workstation using `CLFLUSH` (linear scale)

on average 2700 microseconds for all recovery threads to launch, complete their FASEs in parallel, and terminate. (To put these numbers in perspective, during failure-free execution a FASE in these tests took on the order of one microsecond.) From start to finish, recovering a 128 GB hash table takes under 5 ms.

7.4 Data Size

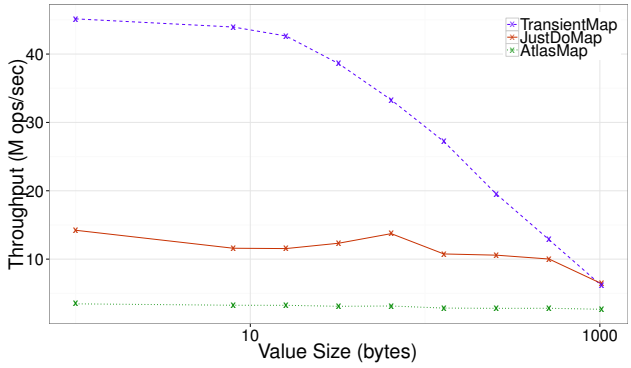

Figure 11. Throughput on server as a function of value size (linear scale)

Figure 11 shows throughput as a function of data size on the various key-value (hash map) implementations. Tests were run on the server machine with eight threads, assume a persistent cache, and vary value sizes from a single byte to one kilobyte. For each operation, values were created and initialized with random contents by the operating thread; the Atlas and JUSTDO variants allocate and initialize data in persistent memory. Allocation and initialization quickly become bottlenecks for the transient implementation. The JUSTDO implementation is less sensitive to data size, since it operates at a slower speed, and value initialization does not begin to affect throughput until around half a kilobyte. At one kilobyte, the allocation and initialization of the data values becomes the bottleneck for both implementations, meaning the overhead for persistence is effectively zero beyond this data size. In contrast to the transient and JUSTDO implementations, the Atlas implementation is nearly unaffected by data size changes: Atlas's bottleneck remains dependency tracking between FASEs.

Note that only Atlas copies the entire data value into a log; in the case of a crash between initialization of a data value and its insertion, Atlas may need to roll back the data's initial values. In contrast, JUSTDO logging relies on the fact that the data value resides in persistent memory. After verifying that the data is indeed persistent, the JUSTDO map inserts a pointer to the data. The "precopy" of JUSTDO copies only the value's pointer off the stack into persistent memory. Consequently, it is affected by data size only as allocation and initialization become a larger part of overall execution. Obviously, the transient version never copies the data value as it is not failure-resilient.

8. Conclusions

We have shown that JUSTDO logging provides a useful new way to implement failure-atomic sections. Compared with persistent memory transaction systems and other existing mechanisms for implementing FASEs, JUSTDO logging greatly simplifies log maintenance, thereby reducing performance overheads significantly. Our crash-injection tests confirm that JUSTDO logging preserves the consistency of application data in the face of sudden failures. Our performance results show that JUSTDO logging effectively leverages persistent caches to improve performance substantially compared with a state-of-the-art FASE implementation.

Acknowledgments

This work was partially supported by the U.S. Department of Energy under Award Number DE-SC-0012199. We thank Dhruva Chakrabarti, Adam Izraelevitz, Harumi Kuno, Mark Lillibridge, Brad Morrey, Faisal Nawab, Michael Scott, Joe Tucek, the ASPLOS reviewers, and our shepherd Mike Swift for suggestions that greatly improved our paper.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. "Shared Memory Consistency Models: A Tutorial." In *IEEE Computer*, Vol. 29 No. 12, December 1996.

[2] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." In *Spring Joint Computer Conference*, 1967.

[3] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Valdimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. "Spin-transfer Torque Magnetic Random Access Memory (STT-MRAM)." In *Journal on Emerging Technologies in Computing Systems (JETC)—Special issue on memory technologies*, 2013.

[4] Brian N. Bershad. "Fast Mutual Exclusion for Uniprocessors." In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992.

[5] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. "Implications of CPU Caching on Byte-addressable Non-Volatile Memory Programming." *Technical report HPL-2012-236, Hewlett-Packard*, 2012.

[6] Aviv Blattner, Ram Dagan, and Terence Kelly. "Generic Crash-Resilient Storage for Indigo and Beyond." *Technical report HPL-2013-75, Hewlett-Packard*, 2013. `http://www.labs.hp.com/techreports/2013/HPL-2013-75.pdf`

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[8] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: Automatically Generating Inputs of Death." In *Conference on Computer and Communications Security (CCS)*, 2006.

[9] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. "Atlas: Leveraging Locks for Non-Volatile Memory Consistency." In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2014.

[10] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David E. Lowell. "The Rio File Cache: Surviving Operating System Crashes." In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[11] Shimin Chen, Phillip B. Gibbons, and Suman Nath. "Rethinking Database Algorithms for Phase Change Memory." In *Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011.

[12] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. "Beyond the PDP-11: Processor support for a memory-safe C abstract machine." In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2015.

[13] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. "From ARIES to MARS: Transaction Support for Next-Generation, Solid-State Drives." In *Symposium on Operating Systems Principles (SOSP)*, 2013.

[14] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-volatile Memories." In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[15] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. "Better I/O Through Byte-addressable, Persistent Memory." In *Symposium on Operating Systems Principles (SOSP)*, 2009.

[16] Intel Corporation. "Intel Architecture Instruction Set Extensions Programming Reference." No. 319433-023, October 2014.

[17] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. "Lock-free Dynamically Resizable Arrays." In *International Conference on Principles of Distributed Systems (OPODIS)*, 2006.

[18] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. "System Software for Persistent Memory." In *European Conference on Computer Systems (EuroSys)*, 2014.

[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing." In *Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[20] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. "NVRAM-aware Logging in Transaction Systems." In *Proceedings of the VLDB Endowment*, 2014.

[21] Marc de Krujf and Karthikeyan Sankaralingam. "Idempotent Processor Architecture." In *International Symposium on Microarchitecture (MICRO)*, 2011.

[22] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative." In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[23] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. "Architecting Phase Change Memory As a Scalable DRAM Alternative." In *International Symposium on Computer Architecture (ISCA)*, 2009.

[24] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. "Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory." In *Conference on File and Storage Technologies (FAST)*, 2013.

[25] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. "MICA: A Holistic Approach to Fast In-Memory Key-value Storage." In *Conference on Networked Systems Design and Implementation (NSDI)*, 2014.

[26] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. "Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores." In *Conference*

on Programming Language Design and Implementation (PLDI), 1998.

[27] David E. Lowell and Peter M. Chen. "Free transactions with Rio Vista." In *Symposium on Operating Systems Principles (SOSP)*, 1997.

[28] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. "Loose-Ordering Consistency for Persistent Memory." In *International Conference on Computer Design (ICCD)*, 2014.

[29] Maged M. Michael and Michael L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms." In *Symposium on Principles of Distributed Computing (PODC)*, 1996.

[30] Microsoft Developer Network. "Alternative to using Transactional NTFS." `http://msdn.microsoft.com/en-us/library/hh802690.aspx`, Accessed 17 September 2014.

[31] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. "ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging." In *ACM Transactions on Database Systems*, Vol. 17 No. 1, March 1992.

[32] Dushyanth Narayan and Orion Hodson. "Whole-System Persistence." In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[33] Faisal Nawab, Dhruva R. Chakrabarti, Terence Kelly, and Charles B. Morrey III. "Procrastination Beats Prevention: Timely Sufficient Persistence for Efficient Crash Resilience." In *International Conference on Extending Database Technology (EDBT)*, 2015. `http://openproceedings.org/2015/conf/edbt/paper-336.pdf`

[34] Faisal Nawab, Dhruva R. Chakrabarti. Terence Kelly, and Charles B. Morrey III "Zero-Overhead NVM Crash Resilience." In *Non-Volatile Memories Workshop (NVMW)*, 2015. `http://nvmw.ucsd.edu/2015/assets/abstracts/41`

[35] Micheal A. Olson, Keith Bostic, and Margo Seltzer. "Berkeley DB." In *USENIX Annual Technical Conference (FREENIX track)*, 1999.

[36] Stan Park, Terence Kelly, and Kai Shen. "Failure-Atomic `msync()`: A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data." In *European Conference on Computer Systems (EuroSys)*, 2013. `http://doi.acm.org/10.1145/2465351.2465374`

[37] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. "Memory Persistency." In *International Symposium on Computer Architecture (ISCA)*, 2014. `http://dl.acm.org/citation.cfm?id=2665671.2665712`

[38] "Persistent Memory Programming." `http://pmem.io/`, Accessed 12 August 2015.

[39] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. "Operating System Transactions." In *Symposium on Operating Systems Principles (SOSP)*, 2009.

[40] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. "Scalable High Performance Main Memory System Using Phase-change Memory Technology." In *International Symposium on Computer Architecture (ISCA)*, 2009.

[41] Andy Rudoff. "In a World with Persistent Memory." In *Non-Volatile Memories Workshop (NVMW)*, 2015.

[42] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. "Automatic Device Driver Synthesis with Termite." In *Symposium on Operating Systems Principles (SOSP)*, 2009.

[43] A.V.S. Sastry and Roy D.C. Ju. "A New Algorithm for Scalar Register Promotion Based on SSA Form." In *Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[44] Russell Sears and Eric Brewer. "Stasis: Flexible Transactional Storage." In *Symposium on Operating Systems Design and Implementation (SOSP)*, 2006.

[45] SQLite `http://www.sqlite.org/`, Accessed 15 January 2016.

[46] "Storage Networking Industry Association (SNIA) Non-Volatile Memory Programming Model." `http://www.snia.org/tech_activities/standards/curr_standards/npm`, Accessed 10 January 2016.

[47] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. "Enabling Transactional File Access via Lightweight Kernel Extensions." In *Conference on File and Storage Technologies (FAST)*, 2009.

[48] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. "The missing memristor found." In *Nature*, Vol. 453 No. 7191, May 2008.

[49] R. Kent Treiber. "Systems Programming: Coping with Parallelism." *Technical Report RJ 5118, IBM Almaden Research Center*, 1986.

[50] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Mannarswamy, Terence Kelly, and Charles B. Morrey III. "Failure-Atomic Updates of Application Data in a Linux File System." In *Conference on File and Storage Technologies (FAST)*, 2015. `https://www.usenix.org/system/files/conference/fast15/fast15-paper-verma.pdf`

[51] Viking Technology. "NVDIMM Technology: ArxCis-NV." `http://www.vikingtechnology.com/nvdimm-technology`, Accessed 10 August 2015.

[52] Haris Volos, Andres Jaan Tack, and Micheal M. Swift. "Mnemosyne: Lightweight Persistent Memory." In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[53] Tianzheng Wang and Ryan Johnson. "Scalable Logging through Emerging Non-Volatile Memory." In *Proceedings of the VLDB Endowment*, 2014.

[54] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. "The CHERI capability model: Revisiting RISC in an age of risk." In *International Symposium on Computer Architecture (ISCA)*, 2014.

[55] Micheal Wu and Willy Zwaenepoel. "eNVy: A Non-Volatile, Main Memory Storage System." In *International Conference*

on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1994.

[56] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. "Overcoming the challenges of crossbar resistive memory architectures." In *Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[57] "Yahoo! Cloud Serving Benchmark (YCSB)." `https://github.com/brianfrankcooper/YCSB/wiki`, Accessed 4 August 2015.

[58] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. "NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems." In *Conference on File and Storage Technologies (FAST)*, 2015.

[59] Sunghwan Yoo, Charles Killian, Terence Kelly, Hyoun Kyu Cho, and Steven Plite. "Composable Reliability for Asynchronous Systems." In *USENIX Annual Technical Conference (ATC)*, 2012. `https://www.usenix.org/system/files/conference/atc12/atc12-final206-7-20-12.pdf`

[60] Anna Zaks and Rajeev Joshi. "Verifying Multi-threaded C Programs with SPIN." Model Checking Software. Springer Berlin Heidelberg, 2008.

[61] Jishen Zhao, Sheng Li, Doe Hyun Lee, Yuan Xie, and Norman P. Jouppi. "Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support." In *International Symposium on Microarchitecture (MICRO)*, 2013.