Runtime Systems for Persistent Memories

by

Vaibhav Gogte

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2020

Doctoral Committee:

Professor Thomas F. Wenisch, Chair
Professor Peter M. Chen
Associate Professor Satish Narayanasamy
Associate Professor Zhengya Zhang

Vaibhav Gogte

vgogte@umich.edu

ORCID iD: 0000-0002-7382-636X

To my parents, Vinod Gogte and Sangita Gogte

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Prof. Tom Wenisch, my PhD advisor and mentor, who has been incredibly supportive during my PhD. Tom is a great educator and a devoted advisor. He is always eager to help; his open-door office policy and regular lab visits were incredibly helpful in getting things moving quickly. I can now admire his patience, having thoroughly exploited his open-door policy to my benefit for all these years. I have always been in awe of his technical expertise; he is unbelievably quick in identifying the problems and giving pointed feedback. He has always adapted to my working style, giving me sufficient space to independently work on my ideas when I desired, and closely working with me and helping me out with the problems when I was stuck. I am grateful to Tom for taking me in his lab and helping me evolve, both personally and professionally. I also worked closely with Prof. Peter Chen and Prof. Satish Narayanasamy on all the memory persistency projects. Pete and Satish taught me how to critically examine my work. Pete's systems outlook and Satish's PL perspective helped me evolve my ideas on architecture, systems and programmability fronts. I can now admit that the weekly meetings with them were overwhelming at first, but the discussions quickly became the highlight of my week. I would also like to thank Prof. Zhengya Zhang for serving on my thesis committee and for his valuable comments that have helped me improve this dissertation. I am also grateful to have worked with the amazing collaborators, Prof. Aasheesh Kolli, William Wang, Stephan Diestelhorst, Prof. Michael Cafarella, and Prof. Loris D'Antoni.

Its been my pleasure to have worked with some of the brightest and hard-working PhD researchers. I would like to thank the members of Wenisch lab – Aasheesh Kolli, Akshitha

Sriraman, Amir Mirhosseini, Amlan Nayak, Brendan West, Harini Muthukrishnan, Hossein Golestani, Kevin Loughin, Neha Agarwal, Ofir Weisse, and Steve Zekeny – not just for all the technical help, but also for organizing numerous unicorn-themed pranks on Tom. They made working in Wenisch lab lively and fun. I would specially like to thank Akshitha Sriraman for entertaining the long discussions on my half-baked ideas and helping me out with the low-level technical details in my projects. She is incredibly hardworking and devoted towards her goals, and she has always inspired me to be persistent and focused in difficult times. I would also like to acknowledge the CSE staff members, Ashley Andrea, Denise Duprie, Jamie Goldsmith, Karen Liska and Stephen Reger, for providing the logistical support in a timely manner.

My PhD experience over the last five years was full of ups and downs. When ideas failed to work after the umpteenth iteration, hopes were crushed after failure to meet paper deadlines, and reviewer B glorifyingly rejected the premise of my papers, I cannot imagine to have endured it all without the kind support of my friends. A big shout-out to the Juwaris group — Kumar Aanjaneya, Harshad Dharmatti, Kunal Garg, Animesh Jain, Sneha Joshi, Aditi Kulkarni, Saurabh Mahajan, Bharadwaj Mantha, Niket Prakash, Shriya Sethuraman, Poorwa Shekhar, Ripudaman Singh, Akshitha Sriraman, and Keval Ramani — for making Ann Arbor a home away from home. The board game nights, boisterous Avalon sessions, and impromptu road-trips with this jovial gang kept me sane these five years. I am grateful to Kumar Aanjaneya for being such an amazing roommate, for pushing me to go to work on dull winter mornings, and for the enlightening discussions over chai. I would specially like to thank Shriya Sethuraman, my closest ally and best friend. She always had my back, and showed me the fun side of life and put a smile on my face when times got tough.

Lastly, I am incredibly grateful to my parents, Sangita Gogte and Vinod Gogte, and my sister Aishwarya Gogte, for their unwavering love and support. I cannot thank them enough for the sacrifices they have made to give me all the independence I needed to follow my dreams. Their constant guidance and encouragement has made me what I am today.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Emerging persistent memory (PM) technologies promise the performance of DRAM with the durability of disk. However, several challenges remain in existing hardware, programming, and software systems that inhibit wide-scale PM adoption. This thesis focuses on building efficient mechanisms that span hardware and operating systems, and programming languages for integrating PMs in future systems.

First, this thesis proposes a mechanism to solve low-endurance problem in PMs. PMs suffer from limited write endurance—PM cells can be written only 10^7-10^9 times before they wear out. Without any wear management, PM lifetime might be as low as 1.1 months. This thesis presents *Kevlar*, an OS-based wear-management technique for PM, that requires no new hardware. Kevlar uses existing virtual memory mechanisms to remap pages, enabling it to perform both *wear leveling*—shuffling pages in PM to even wear; and *wear reduction*—transparently migrating heavily written pages to DRAM. Crucially, Kevlar avoids the need for hardware support to track wear at fine grain. It relies on a novel *wear-estimation* technique that builds upon Intel's Precise Event Based Sampling to approximately track processor cache contents via a software-maintained Bloom filter and estimate write-back rates at fine grain.

Second, this thesis proposes a persistency model for high-level languages to enable integration of PMs in to future programming systems. Prior works extend language memory models with a persistency model prescribing semantics for updates to PM. These approaches require high-overhead mechanisms, are restricted to certain synchronization constructs, provide incomplete semantics, and/or may recover to state that cannot arise in fault-free program execution. This thesis argues for persistency semantics that guarantee failure

atomicity of synchronization-free regions (SFRs) — program regions delimited by synchronization operations. The proposed approach provides clear semantics for the PM state that recovery code may observe and extends C++11's "sequential consistency for data-race-free" guarantee to post-failure recovery code. To this end, this thesis investigates two designs for failure-atomic SFRs that vary in performance and the degree to which commit of persistent state may lag execution.

Finally, this thesis proposes StrandWeaver, a hardware persistency model that minimally constrains ordering on PM operations. Several language-level persistency models have emerged recently to aid programming recoverable data structures in PM. The language-level persistency models are built upon hardware primitives that impose stricter ordering constraints on PM operations than the persistency models require. StrandWeaver manages PM order within a *strand*, a logically independent sequence of PM operations within a thread. PM operations that lie on separate strands are unordered and may drain concurrently to PM. StrandWeaver implements primitives under strand persistency to allow programmers to improve concurrency and relax ordering constraints on updates as they drain to PM. Furthermore, StrandWeaver proposes mechanisms that map persistency semantics in high-level language persistency models to the primitives implemented by StrandWeaver.

CHAPTER I

Introduction

Persistent memory (PM) technologies, such as Intel and Micron's 3D XPoint, are here — cloud vendors have already started public offerings with support for Intel's Optane DC persistent memory [7, 13, 2, 20]. PMs aim to revolutionize storage by integrating the byte-addressability of DRAM with the durability of disks. These technologies exhibit many useful characteristics that make them appealing to system designers. PMs can be accessed using a load-store interface eliminating the need for expensive block-based software interface required for traditional storage devices. The load-store interface to storage allows fine-grained manipulation of data and lowers access latency.

As PMs are durable, they retain data across failures such as power failures or OS or program crashes. PMs can be adopted in existing systems to store data structures that can be manipulated using fine-grained load-store interface for high performance, and yet persist across failures. In case of a failure, the volatile state of the program (*e.g.* registers, caches, and DRAM) are lost, but PMs retain the state. The applications can rely on PM durability to inspect PM state, reconstruct required volatile state, and resume program execution. However, despite many of these appealing properties, several challenges remain in existing hardware, programming and software systems that inhibit PM adoption in future systems. Specifically, this thesis aims to address following challenges.

Low PM endurance. PMs have a low device endurance. Each PM cell can be written up to a limited number of times (*e.g.* 10^7-10^9) [130, 173, 175] before it wears out and fails.

The low device endurance reduces system lifetime and incurs high hardware cost. This requires system designers to manage PM writes and restrict device wear out to improve overall system lifetime.

Lack of language-level persistency models. Recovery of data structures stored in PM requires precise program state after failure. Several persistency models [22, 98, 56, 171, 112] have been proposed to define semantics of PM state post failure. Such persistency models define that the data *persists* when the effects of a store are guaranteed to be observed in PM in case of a failure. Similar to memory consistency models that govern the visibility of memory accesses, these persistency models govern the order in which updates persist in PM. Both industry [22, 98] and academia [171, 56, 62] have proposed candidate persistency models that rely on lower-level hardware ISA primitives to prescribe order over PM updates. Unfortunately, these persistency models do not specify semantics for high-level programming languages. Advancing these approaches will return us to the "wild west" days of hardware memory consistency, where every vendor offered a different model and programmers often resorted to ISA-specific inline assembly not just for performance, but to ensure the correctness of concurrent code. Such models place an unreasonable burden on programmers, make writing portable programs exceedingly difficult, and hinder both hardware and compiler optimizations that may reorder or elide PM reads and writes.

Inefficient hardware persistency models. Modern hardware systems build ISA primitives to order updates as they persist to PM. Unfortunately, these primitives apply stricter than required ordering constraints on PM operations. The additional constraints serialize concurrent PM operations and restrict any potential reordering opportunities between them. Prior research proposals relax persist ordering constraints by performing hardware logging mechanisms [111, 62, 166, 113] to perform failure-atomic PM updates or implementing relaxed persistency models [171, 125, 112, 160] in hardware to order PM operations. Hardware logging mechanisms order programmer-annotated set of PM operations to ensure efficient PM logging and ensure relaxed failure-atomic implementation for PM

updates. However, they rely on fixed and inflexible hardware implementations that fail to extend to a wide range of evolving language-level persistency models. In contrast, other works propose hardware mechanisms for relaxed persistency models such as epoch persistency model. Under epoch persistency model, *persist barriers* divide program regions into *epochs*; they allow persist reordering within epochs and disallow persist reordering across epochs. Unfortunately, these works allow only the consecutive persists that lie within the same epoch as concurrent. They fail to relax ordering constraints on persists that do not lie in the same epoch, but are concurrent.

This thesis proposes three mechanisms, that span across hardware, programming and operating systems, to solve the challenges in deploying PMs in future systems. First, it builds software-managed wear-management mechanisms to improve PM device lifetime. Further, it defines a persistency model at a language-level to provide persistency semantics for high-level programming languages such as C++. Finally, it builds a hardware *strand persistency model*, a relaxed persistency model in hardware, that allows precise ordering constraints on PM accesses as required by high-level language persistency models. The proposed mechanisms are briefly described below.

1.1 Software Wear Management

PMs exhibit limited write endurance – a PM cell can be written a limited number of times before it wears out. For example, a phase-change memory is expected to have a write endurance of $10^7 - 10^9$ writes [130, 175, 173] while resistive RAM is expected to sustain over 10^{10} writes before wearing out [209]. The system designers need to be cognizant of low PM endurance, to ensure that memory does not wear out and fail. This thesis considers systems with heterogeneous memory – with both DRAM and PM connected to the memory bus. Such systems may use PM for persistent data storage or as a means to replace some or all of DRAM with a cheaper/higher-capacity technology.

PM *wear-management* mechanisms employ *wear leveling* [173, 175] to spread writes

to all memory locations uniformly. The wear-leveling schemes periodically remap memory locations so that memory writes are uniformly distributed in PM. Other mechanisms employ *wear reduction* by reducing the number of writes to PM with additional caching layers [130, 173, 225, 176, 183]. The wear-leveling and wear-reducing mechanisms introduce addition design complexities in hardware required to remap memory locations. Moreover, these mechanisms rely on additional caching layers and/or volatile DRAM. Thus, these mechanisms do not readily extend to the applications that use PMs as a persistent storage and rely on its durability to recover program state in case of failure.

This thesis proposes *Kevlar*, a wear-management mechanism for PMs in software. Kevlar leverages the observation that the operating system already maintains a mapping of virtual to physical memory locations. Kevlar reuses existing virtual to physical mappings to periodically remap memory locations in the OS – this eliminates overheads of additional translation layer for wear management. It builds a random page *shuffle* mechanism that periodically remaps memory pages to randomly chosen physical memory frames. Kevlar's simple page shuffle mechanism is sufficient for PM technologies with higher write endurance (*e.g.* , resistive RAM) to achieve target system lifetime of four years. Note that, Kevlar's wear-leveling mechanism is wear-oblivious – it remaps the entire application footprint. Each shuffle operation remaps all of the PM – frequent shuffles can have a prohibitive performance overhead. Interestingly, our analysis shows that the number of required shuffles is quite small; shuffling all PM pages roughly every four hours is sufficient to achieve uniform wear out and incurs less than 0.15% performance overhead.

Additionally, this thesis shows that Kevlar's wear-leveling mechanism alone is not sufficient to achieve target system lifetime for PMs with lower write endurance (e.g., Intel and Micron's 3D XPoint memory [102]). To this end, Kevlar exploits memory heterogeneity to reduce wear to PM. It performs carefully targeted page migrations to a neighbouring high-endurance DRAM for applications that use a PM device for memory expansion alone. We show that migrating as few as 3% of pages from PM to DRAM is sufficient to achieve

our target lifetime. Kevlar relies on reserve footprint in PM to perform page shuffles across the nominal and reserve capacity.

A key challenge solved by Kevlar is to identify which pages are frequently written back to main memory (as opposed to those that are frequently written to, due to the presence of hardware caches) without any new hardware extensions. Kevlar designs a heuristic based on Intel's Precise Event Based Sampling (PEBS) [97] to approximate cache contents and estimate writebacks to PM. Kevlar's wear-reduction mechanism achieves target system lifetime for low-endurance PMs by migrating less than 3% of the application working set to neighboring DRAM (when durability is not needed) and incurring a performance overhead of less than 1.5%.

1.2 Persistency for Synchronization-Free Regions

The promise of PM is to enable data structures that provide the convenience and performance of in-place load-store manipulation, and yet persist across failures, such as power interruptions and program crashes. Future programming systems can employ PMs to store data durably, reconstruct required volatile state, and resume program execution. Unfortunately, no high-level language yet provides any durability semantics, which are required to enable programming recoverable data-structures in PM. This thesis makes a strong case for building persistency semantics upon the strong foundations of the data-race-free (DRF) memory model of C++, using existing C++ synchronization operations to prescribe ordering for persists.

This work proposes persistency semantics that provide failure atomicity at the granularity of *synchronization free regions* (SFRs)—thread regions delimited by synchronization operations or system calls. Failure-atomic SFRs guarantee that either all or none of the updates within an SFR are visible to recovery post failure. In the absence of SFR atomicity, recovery may observe PM state that could never arise in fault-free execution. Under failure-atomic SFRs, the state observed by recovery always conforms to the program state at a

frontier of past synchronization operations on each thread. This work argues that failure-atomic SFRs strike a compelling balance between programmability and performance. In a well-formed program, SFRs must be data-race free. This property allows us to extend the "sequential-consistency for data-race-free programs (SC for DRF)" guarantee to recovery code.

This thesis builds the persistency model that relies on synchronization operations in C++ implementation (built on LLVM [129] v3.6.0). The compiler implementation introduces undo logging to ensure that SFRs are failure-atomic. This work considers two implementations that vary in simplicity and performance.

SFR-atomicity with coupled visibility: In this design, the persistent state lags the frontier of execution by at most a single (incomplete) SFR. Recovery rolls back to the start of the ongoing SFR upon failure. This approach admits simple logging, but exposes the latency of PM flushing and commit.

SFR-atomicity with decoupled visibility: In this design, execution is allowed to run ahead of the persistent state. We defer flushing and commit to background threads using a garbage-collection-like mechanism. Further, the work proposes efficient mechanisms to ensure that the SFR commit order matches their execution order. This implementation enables high performance as we decouple and perform flush and commit operations in background.

Owing to the simple logging, SFR-atomicity with coupled visibility results in an average performance improvement of 63.2% over state-of-the-art ATLAS design [47]. SFR-atomicity with decoupled visibility further enables light-weight recording of SFR order and performs flush and commit operations off the critical execution path. As a result, this design leads to a further performance improvement of 50.1% over SFR-atomicity with coupled visibility.

1.3 Relaxed Hardware Persistency Model

The language-level persistency models [123, 78, 47, 77, 124] define semantics for PM access in high-level programming languages. These persistency models provide two key persistency semantics. First, they define the order in which updates persist in PM. Second, they ensure failure-atomicity for a set of updates to PM. These persistency models rely on low-level instruction primitives to provide ordering and failure-atomicity for PM accesses. For instance, Intel x86 systems employ `CLWB` (or `CLFLUSHOPT` in older systems) instruction to explicitly flush dirty cache lines to the memory controller and a subsequent `SFENCE` instruction to order subsequent stores with prior `CLWB`s and stores. Compiler implementations rely on these instructions to: (1) order and flush updates to PM, and (2) build logging mechanisms to ensure failure-atomicity for set of updates to PM. Unfortunately, existing hardware approaches enforce additional ordering constraints on persists that are not required for ensuring correct recovery. These ordering constraints limit persist concurrency.

This work proposes StrandWeaver, which formally defines and implements the *strand persistency* model to minimally constrain ordering on persists to PM. The strand persistency model defines the order in which persists may drain to the PM. It decouples persist order from the write visibility order (defined by the memory consistency model)—memory operations can be made visible in shared memory without stalling for prior persists to drain to PM. To implement strand persistency, we introduce three new hardware ISA primitives to manage persist order. A *NewStrand* primitive initiates a new *strand*, that defines a logically independent stream of PM operations within a logical thread that may persist concurrently to PM. A *persist barrier* orders PM operations within a strand — persists separated by a persist barrier drain to PM in order. A *JoinStrand* primitive orders persists initiated earlier on previous strands before the subsequent persists are issued.

StrandWeaver proposes hardware mechanisms to build the strand persistency model upon these primitives. StrandWeaver implements a *strand buffer unit* alongside the L1 cache that issues persists on different strands concurrently to PM, and orders persists within

a strand separated by a persist barrier. It also implements a *persist buffer* alongside the load-store queue, that records the ongoing strand persistency primitives, and ensures that the persists separated by *JoinStrand* primitive complete in order.

StrandWeaver integrates the strand persistency primitives defined in hardware ISA with the programmer friendly language-level persistency models. Thus, programmers no longer need to reason about the persist order at the abstraction of the hardware ISA. StrandWeaver builds logging mechanisms that employ strand persistency primitives to minimally constrain persists required to guarantee correct failure recovery.

We showcase the wide applicability of StrandWeaver primitives by integrating our logging with three prior language-level persistency models that provide failure-atomic transactions [8, 201, 54], synchronization-free regions [78], and outermost critical sections [47], respectively. These persistency models provide simpler primitives to program recoverable data structures in PM—programmer-transparent logging mechanisms layered on top of our StrandWeaver hardware hide low-level hardware ISA primitives and reduce the programmability burden.

1.4 Summary

The upcoming PM technologies can potentially revolutionize the future storage systems. However, several challenges need to be addressed to integrate PMs efficiently in future programming and hardware systems: 1) PMs have a low device endurance, and wear out after 10^7-10^9 writes. Untimely device failures lower data reliability and increase hardware provisioning and replacement costs. 2) Existing programming systems lack support for programming persistent data-structures in PMs. Without the support in high-level languages, programmers need to rely on custom assembly-level program implementations that are difficult to program and error prone. 3) Existing hardware systems build ISA extensions for PMs that restrict concurrency of PM operations and introduce high performance overhead. This thesis aims to address these challenges.

The remainder of this thesis is structured as follows. Chapter II discusses the relevant background required for understanding the proposals in this thesis. Chapter III illustrates the runtime mechanisms for performing wear management and improving device lifetime for PMs. Chapter IV defines the language-level persistency model that extends general synchronization primitives in high-level languages such as C++. Chapter V discusses hardware strand persistency model to relax ordering of PM operations. Finally, Chapter VII details future work to efficiently integrate PMs in future systems and concludes this thesis.

While this thesis focuses on runtime systems for PMs, I have also worked on the hardware accelerator for matching regular expressions in an unstructured textual data. Appendix A describes *HARE*, a stall-free hardware accelerator design. HARE scans input data at a fixed rate, examining multiple characters from a single input stream in parallel in a single accelerator clock cycle. HARE implements a 1GHz 32-character-wide design targeting ASIC implementation that processes data at 32 GB/s—matching modern memory bandwidths. This ASIC design outperforms software solutions by as much as two orders of magnitude.

CHAPTER II

Background and Motivation

This chapter describes relevant background for the works proposed in this thesis.

2.1 Persistent Memories (PMs)

Persistent memory technologies, such as Phase Change Memory [130, 175], Memristor [209], and Spin Torque Transfer RAM [217] are byte-addressable, achieve near-DRAM performance, and are denser than DRAM (hence cheaper), consume less power than DRAM, and are also non-volatile. These characteristics allow systems to leverage PMs in exciting new ways. We focus on two well-studied use cases: (1) capacity expansion and (2) memory persistency.

Capacity expansion: Owing to their higher density and lower power consumption, PMs are projected to be cheaper than DRAM [130, 175, 225, 16, 116, 65] on a dollar per GB basis. System designers (for example, cloud vendors) may pass these cost advantages on to end users in two ways. First, for applications bound by memory capacity, users may obtain systems (or instances) with larger main memory for the same cost. For instance, Intel expects to soon offer servers with up to *6TB* of PM [86, 10]. Second, users may also get access to systems with the same amount of main memory as before for a cheaper price. In both cases, end users may observe improvements in their application performance per dollar.

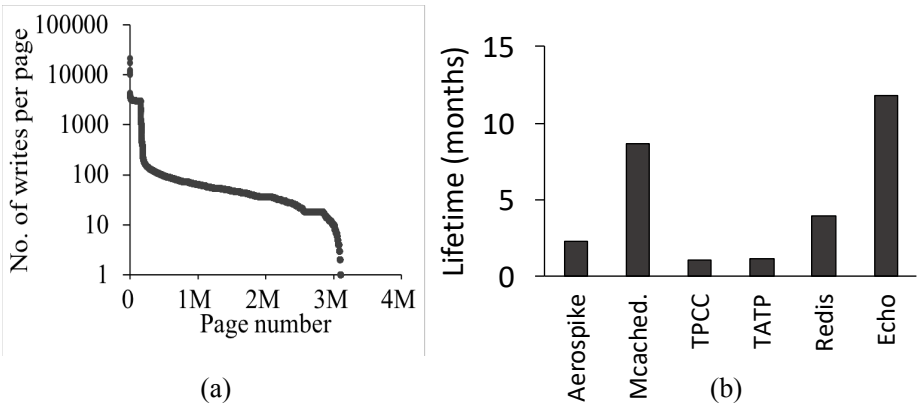


Figure 2.1: **Disparity in PM page writes.** (a) Pages sorted by number of writes (program entirety) in Aerospike: There is a large disparity between most and least written pages. (b) PM lifetime with no wear leveling: The lifetime until 1% of pages sustain 10^7 writes can be as short as 1.1 months.

Memory persistency: Since PMs are non-volatile, they blur the traditional distinctions between memory and storage. Recent research leverages PM non-volatility by accessing persistent data directly in memory via loads and stores [56, 54, 201, 171, 126, 104, 122, 112, 62, 160, 123, 78]. The byte-addressable load-store PM interface enables fined-grained accesses to persistent data and avoids the expensive serialization and de-serialization layer of conventional storage [115].

2.2 Endurance of Persistent Memories

Whereas PMs exhibit many useful properties, they suffer from a limited write endurance. For example, PCM endures only 10^7 - 10^9 writes [173]. In contrast, DRAM endurance is essentially unbounded (> 10^{15} writes) [175]. Limited PM endurance may lead to a rapid capacity loss for write-intensive applications. Figure 2.1(a) shows the disparity between writes seen by the hottest and coldest pages for Aerospike (see Section 3.4 for our methodology). Absent wear management, frequently written-back addresses wear out sooner, compromising device lifetime. Figure 2.1(b) shows the lifetime until 1% of memory locations wear out in a device with a write endurance of 10^7 writes (such as PCM) under the write patterns of various applications assuming no efforts to manage wear. For

example, we observe that TPCC can wear out a PCM memory device within 1.1 months.

2.2.1 Wear Management for PMs

Prior *wear management* mechanisms improve memory device lifetimes despite the poor write endurance of the underlying technology at a minimal performance loss. Wear management techniques can be broadly classified into two orthogonal categories: (1) wear leveling and (2) wear reducing. Some applications tend to write back frequently to only a small region of their memory footprint. The physical memory cells containing the frequently written-back addresses get worn out, leading to poor lifetimes even while other memory cells observe no wear. Wear leveling [175, 225, 173] aims to solve this problem by uniformly distributing writebacks to all memory cells. Such techniques generally use a programmer-transparent address translation mechanism in the device controller to spread writebacks over different memory locations.

Even with perfect wear leveling, device lifetimes may be unacceptably low for devices with poor write endurance. At best, wear leveling ensures that all locations incur the application's mean writeback rate. If this rate is too high, all locations may wear out sooner than the desired lifetime. Moreover, wear leveling necessarily requires swapping data between memory locations, generating additional writes. Wear-reducing techniques [175, 183, 220] improve device lifetimes by reducing the number of writes that reach the memory device. These techniques usually add a large cache in front of the memory device to buffer and coalesce writes so as to reduce the overall number of writes reaching memory. Since wear-leveling and wear-reducing techniques are orthogonal, they can be employed together to achieve even better device lifetimes.

2.2.2 Wear-aware Virtual Memory System

Prior PM wear-management mechanisms [175, 173, 225, 183, 174] require an additional indirection layer in hardware to uniformly wear PM cells. However, these mecha-

nisms suffer from several drawbacks. First, these mechanisms [175, 173, 174, 183] use volatile DRAM caches to reduce wear to PM. These mechanisms do not readily support applications [160] that rely on PM durability, since the volatile DRAM caches lose data upon power failure. Second, these mechanisms perform additional DRAM cache lookups and address translation for each memory access, delaying PM loads/stores. Third, wear leveling alone sometimes achieves PM lifetime of only 2.3 years (as shown later in Section 3.5.2)—lower than the desired system lifetimes. These device-level mechanisms are unable to exploit memory system heterogeneity for applications that employ PMs for capacity expansion.

This thesis explores low-overhead OS wear-management mechanisms that can extend PM device lifetime to a desired target without any additional indirection layers. Indeed, our approach is analogous to similar ongoing efforts [94, 93, 29, 221, 169, 39, 132, 110] in Flash-based systems to identify and eliminate performance bottlenecks in the Flash translation layer (FTL). These works avoid FTL complexities and overheads by folding its features either into the virtual memory system [94, 93, 29, 221, 39], or into file system applications [169, 39, 132, 110]. Like these works, this thesis aim to build PM wear management into the virtual memory system.

2.3 Memory Persistency Models

Modern hardware systems implement hardware structures to reorder, coalesce, elide, or buffer updates, which complicate ordering persists to PM [171, 34, 56]. For instance, write-back caches lazily drain dirty cache lines to memory—ordering visibility of stores to the write-back caches does not imply that the PM writes are ordered. Such reordering complicates using PMs for recovery because the correctness of recovery mechanisms rely on the order in which updates persist to the PM [171, 52, 201, 56, 54]. Several persistency models have been proposed, both in industry [98, 22] and in academia [171, 56, 62], to guarantee persist order.

Similar to memory consistency models, which reason about visibility of memory operations, memory persistency models specify the order in which updates persist to PM. Pelley et al. [171] propose strict and relaxed memory persistency models to specify persist ordering. Strict persistency couples visibility of stores to the order in which they persist to PM—persist order follows the visibility order of PM operations. Unfortunately, strict persistency has a high performance overhead as it restricts persist concurrency especially under conservative consistency models, such as TSO, which strictly order visibility of stores.

Relaxed persistency models decouple persist order from the order in which memory operations become visible. Epoch persistency introduces *persist barriers* that divide program execution into *epochs*. Persists within epochs can be issued concurrently, while persists separated by a persist barrier are ordered to the PM. Several implementations [112, 98, 160, 56, 188], including Intel's x86 ISA, build epoch persistency models.

Intel's persistency model. Intel x86 systems employ `CLWB` (or `CLFLUSHOPT` in older systems) instruction [98] to explicitly flush dirty cache lines to an asynchronous data refresh (ADR)-supported PM controller [100]. In case of power failure, an ADR-supported PM controller flushes pending operations to PM. `SFENCE` acts as a persist barrier that orders any subsequent `CLWB`s with the preceding `CLWB`s. Additionally, `SFENCE` also orders visibility of subsequent stores after the preceding `CLWB`s complete to ensure that the stores do not drain from the write-back caches to PM before prior `CLWB`s finish. Thus, `CLWB-SFENCE` ensures that the data persists to PM before any subsequent stores are visible.

Note, however, that PM stores may (unexpectedly) persist well before the `CLWB` if they are replaced from the cache hierarchy, and failure atomicity is assured only at the granularity of individual persist operations. Logging mechanisms must be built if larger failure-atomicity granularity is desired [123] and recovery code must explicitly account for the possibility that PM stores are replaced from the cache well before they are explicitly flushed.

2.3.1 Failure Atomicity

Logging mechanisms such as shadowing [56] and write-ahead-logging (WAL) [54, 201, 111, 91, 99] provide failure atomicity for a group of persists. In shadowing, updates are made to a shadow copy of the original data. The shadow copy is then committed by atomically switching a pointer in a metadata structure (e.g., page table). WAL provides failure atomicity by either logging the updates in *redo* or *undo logs*. In redo-logging [201, 91, 54], updates are first recorded in persistent logs and then applied in-place in the original location. Thus, a store implies (at least) two PM writes, one to log the update and one to mutate the original location. In case of failure, the recovery process inspects the redo logs and reapplies the updates. In contrast, undo logs record the old value of a location before it is written. On failure, the recovery process rolls back partial PM updates from undo logs. Redo-logging requires isolation [54] or redirection [201, 91] of subsequent loads to the log area, which typically incurs high overhead (in a fault-free execution). In contrast, undo logs allow in-place updates to data structures, so subsequent loads can read these locations directly.

2.3.2 Logging Mechanisms

Prior hardware and software mechanisms use logging to provide failure atomicity, but most work has focused on transaction-based programs [141, 126, 201, 54, 99]. Hardware mechanisms create undo logs [111] or redo logs [62] transparently for transaction-based code, thus enabling failure atomicity for the transaction, but require complex hardware structures for log management. Software solutions, such as Mnemosyne [201] and NV-Heaps [54], implement libraries that enable failure atomicity for transaction-based programs. However, in addition to semantic differences, there are significant challenges in porting existing lock-based programs to a transactional execution model [47, 43]. We seek to look beyond transaction-based programs and define durability semantics for the more general synchronization primitives offered by modern programming languages.

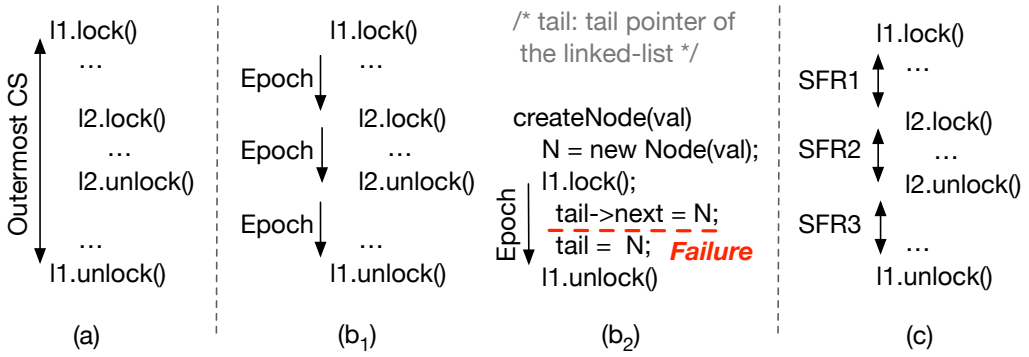


Figure 2.2: **Failure atomicity guarantees by language-level persistency models.** (a) Failure atomicity for outermost critical sections in ATLAS, (b_1) Epoch ordering in ARP, (b_2) Unclear failure atomicity semantics in ARP resulting in partial updates in PM, and (c) Our proposal: failure atomicity for SFRs.

2.4 Persistency Semantics for Languages

This chapter discusses existing proposals that add persistency semantics to the language memory model. In particular, ATLAS [47] and acquire-release persistency (ARP) [123] extend the C++ memory model with persistency semantics. The two proposals differ in the granularity of failure atomicity they guarantee and rely on different synchronization primitives to ensure correct persist ordering in PM. We discuss each proposal below.

ATLAS: ATLAS provides persistency semantics for lock-based multi-threaded C++ programs. It guarantees failure atomicity at the granularity of an outermost critical section, as shown in Figure 2.2(a), where a critical section is the code bounded by *lock* and *unlock* synchronization primitives. The failure atomicity of outer-most critical sections ensures that recovery observes PM state as it existed when no locks were held in the program. This guarantee precludes recovery from observing state that is partially updated by an interrupted critical section. Failure-atomicity of critical sections is appealing from a programmability perspective, as it guarantees that recovery may only observe sequentially consistent PM state.

However, ATLAS's persistency semantics have significant shortcomings. ATLAS fails to provide any durability semantics for synchronization operations other than mutexes. It

does not support widely used synchronization primitives, such as condition variables, and does not offer any semantics for lock-free programs. Moreover, it does not provide clear semantics for persistent updates outside of critical sections. Such updates may be partially visible after failure. In addition, ATLAS requires recording the total order of lock acquires and releases during execution and a complex cycle-detection mechanism to ensure that mutually-dependent critical sections are made failure atomic together. As we will show, the performance overhead of the required logging and cycle-detection mechanisms are high.

ARP: ARP specifies persistency semantics that provide failure atomicity of individual stores. ARP ascribes persists to ordered *epochs* using intra- and inter-thread ordering constraints prescribed via synchronization operations. As shown in Figure 2.2(b_1), ARP may re-order persists within epochs but disallows reordering across epochs. However, ARP constrains only the latest point at which a PM write may become durable—ARP allows for volatile write-back caches that reorder PM writes. A PM write may become durable as soon as it is globally visible. As such, a potentially unbounded set of writes may be reordered and visible even though preceding writes (in program order) are lost upon failure.

Figure 2.2(b_2) shows example code to append a new node to a persistent linked-list. Under fault-free execution in ARP, this code first acquires an exclusive lock on the linked-list, updates the `Next` pointer of the tail to the newly created node, and then the tail pointer is updated to the new node. As ARP does not constrain the durability of the two updates before the completion of the epoch, the update to `tail` may become durable earlier than the update to `tail->next`. In case of a failure, an incomplete update to the tail pointer will result in an inconsistent linked-list. The two updates within the critical section must be failure-atomic to ensure consistency of the linked-list. Additional logging mechanisms are required to provide failure atomicity at larger granularity.

We find ARP semantics unsatisfying. Although it may be possible to construct logging mechanisms that can tolerate writes that become persistent far earlier than expected (e.g., well before preceding store and release operations), reasoning in such a framework is

difficult—a logging mechanism might have to resort to checksums or other complex, probabilistic mechanisms to detect partial log records. Importantly, a programmer must reason about non-serializable states when writing recovery code.

We argue instead for persistency semantics that provide failure-atomic synchronization-free regions — regions of code delimited by synchronization operations or system calls. This thesis proposes a persistency model that can support arbitrary C++ synchronization operations with clear semantics and simple runtime mechanisms, avoiding the performance overheads of ATLAS.

2.4.1 C++ Memory Model

The C++ memory model provides synchronization operations, namely atomic loads, stores, and read-modify-writes, to order shared memory accesses. These accesses may directly manipulate synchronization variables, enabling implementation of a wide variety of synchronization primitives. In this thesis, we refer to accesses to atomic variables that have load semantics as *acquire* operations, and those with store semantics as *release* operations. The C++ memory model prescribes a *happens-before* ordering relation between release and acquire operation, to enable programmers to order shared memory accesses (formalized later in Section 4.4.1). The happens-before relation orders the visibility of data accesses in (volatile) memory. However, C++ currently provides no durability semantics for accesses to PM. This thesis extends the semantics of synchronization operations to also prescribe the order in which PM updates become durable.

2.5 ISA-Level Persistency Mechanisms

Modern hardware systems, such as Intel x86 system, extend instruction set architectures to order persists to PM. In Intel x86 systems, `CLWB` (or `CLFLUSHOPT` in older systems) instruction [98] flush dirty cache lines to an ADR-supported PM controller. Additionally, `SFENCE` operation orders any subsequent `CLWB`s with the preceding `CLWB`s. `SFENCE` divides

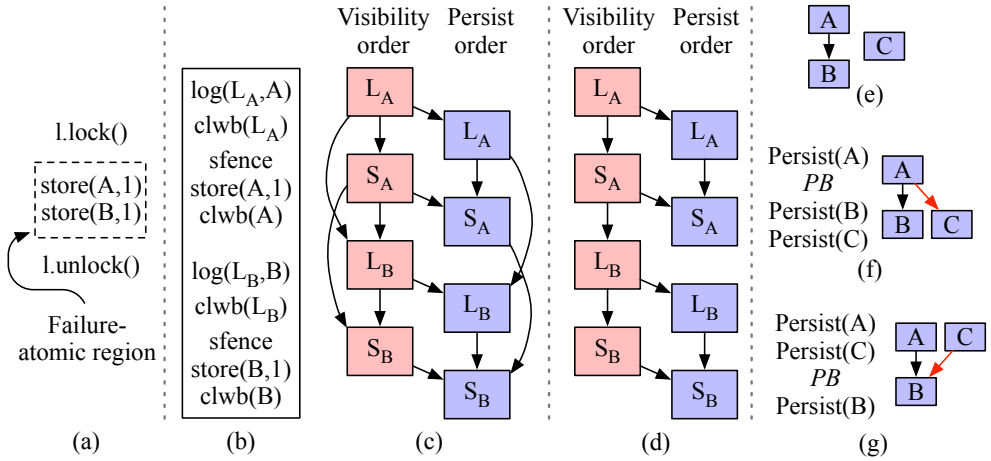


Figure 2.3: **Mapping language-level persistency models to Intel's x86 ISA.** (a) Example failure-atomic region in ATLAS bounded by lock and unlock operations, (b) ATLAS logging using Intel's ISA extensions for PM, (c) Visibility and persist ordering of logs and in-place updates on a TSO system, (d) Ideal persist ordering constraints that are sufficient for correct recovery, (e) Desired order on persists `A`, `B`, and `C`, (f) Persist barrier (PB) additionally orders persists `A` and `C`, (g) PB additionally orders persists `C` and `B`.

program execution into epochs — `CLWB`s within the epoch are allowed to reorder to the PM, while `CLWB`s in different epochs are ordered. Intel x86 system builds an epoch persistency model to ensure persists in different epochs are ordered.

Limitations. Persist concurrency is limited by the size of epochs [112]. Although `CLWB`s within an epoch can flush data concurrently to PM, `SFENCE` enforces stricter ordering constraints on persists, which are not required for ensuring correct recovery. Any subsequent stores and `CLWB`s that are independent and can be issued concurrently to the PM are serialized by `SFENCE`. `SFENCE` causes long-latency stalls as it delays visibility of subsequent stores until prior `CLWB`s flush data to the PM controller.

Example. The language-level persistency models build compiler frameworks or software libraries to map high-level semantics in languages to low-level hardware ISA primitives. Figure 2.3(a) shows example code for a failure-atomic region in ATLAS enclosed by lock and unlock operations. ATLAS instruments each store with undo logging to assure failure atomicity. On failure, recovery inspects undo logs and rolls back partially executed failure-atomic regions. For correct recovery, logs need to persist before in-place updates

are made to PM—ATLAS relies on low-level hardware ISA primitives to assure this ordering. Unfortunately, hardware imposes stricter ordering constraints than required by these persistency models for correct recovery.

Figure 2.3(b) shows example undo logging code for the ATLAS persistency model to ensure failure atomicity of updates to PM locations `A` and `B` on an Intel x86 system. Undo logging requires pairwise ordering of logs and a subsequent store—logs must persist before corresponding updates for correct recovery. Note that logs L_A and L_B (and similarly, updates to locations `A` and `B`) can persist to PM concurrently (as shown in Figure 2.3(d)). Unfortunately, `SFENCE` orders log creation and flush to L_A with log creation and flush to L_B. Under Intel's TSO consistency model [170], visibility of stores is ordered (visibility of L_A and S_A is ordered in Figure 2.3(c)). `SFENCE` additionally restricts visibility of subsequent stores until prior `CLWB`s complete—S_A is not issued until L_A persists. Thus, Intel's persistency model imposes stricter constraints on visibility and persist order that are not required for recovery by language-level models—epoch size limits persist concurrency.

Managing epoch size. Persist concurrency is limited by small epoch size, as language-level model implementations instrument each store within a failure-atomic region with a log operation followed by `SFENCE`. The presence of ambiguous memory dependencies make it challenging for compilers to perform static analysis at compile time [71, 74, 185] to coalesce logging operations within failure-atomic regions. Even with ideal compiler mechanisms to group persists, epoch persistency fails to specify precise ordering constraints. Figure 2.3(e) shows the desired order on persists `A`, `B`, and `C`—persist `C` can be issued concurrent to persists `A` and `B`. The persist barrier introduces additional ordering constraints on `C` when it is issued in either of the epochs as shown in Figure 2.3(f,g).

CHAPTER III

Software Wear Management for Persistent Memories

3.1 Introduction

Forthcoming Persistent Memory (PM) technologies, such as 3D XPoint [10, 102], promise to revolutionize storage hierarchies. These technologies are appealing in many ways. For example, they are being considered as cheaper, higher capacity and/or energy-efficient replacements for DRAM [130, 175, 225, 16], low-latency and byte-addressable persistent storage [56, 54, 201, 171], and even as hardware accelerators for neural networks [186, 177]. We focus on systems with heterogeneous memory—with both DRAM and PM connected to the memory bus. Such systems may use PM for persistent data storage or to replace some or all of DRAM with a cheaper/higher-capacity technology.

Nevertheless, PM's limited write endurance [130, 175, 225, 220, 53] may hinder adoption. Just like erase operations wear out Flash cells, PM devices may also wear out after a certain number of writes. The expected PM cell write endurance varies significantly across technologies. For example, a phase-change memory is expected to endure $10^7 - 10^9$ writes [130, 175, 173] while resistive RAM may sustain over 10^{10} writes [209]. So, system developers must consider PM cell write frequency and manage wear to ensure memory endures for the expected system lifetime.

PM wear-management techniques employ *wear leveling*, spreading writes uniformly over all memory locations, and/or *wear reduction*, reducing the number of writes with

additional caching layers [130, 173, 225, 176, 183, 59]. Unfortunately, prior techniques rely on various kinds of hardware support. Some proposals [173, 225] add an additional programmer-transparent address translation mechanism in the PM memory controller. These mechanisms periodically remap memory locations to uniformly distribute writes across the PM. Other techniques [176, 220, 59] perform wear reduction by remapping contents of frequently-written PM page frames to higher-endurance DRAM. Such techniques depend on hardware support to estimate wear, for example, via per-page counters or specialized priority queues/monitoring in the memory controller. Unfortunately, PM-based mechanisms [176, 220, 59] that rely on higher-endurance but volatile DRAM to reduce wear do not support applications [160] that require crash consistency when using PM as storage.

The indirection mechanisms proposed for PMs are analogous to the translation layer in Flash firmware [70, 131, 119], which perform functionalities such as garbage collection [119, 70, 213] and out-of-place updates [133, 119, 131, 70] in addition to wear leveling, and incur high erasure latency [133, 114, 70]. Additional translation layers increase design complexity and incur higher access latency and power/energy consumption. Indeed, recent work [94, 93, 29, 221, 39, 169, 132, 110] aims to eliminate complexity and overhead associated with a Flash translation layer by combining its features in either the virtual memory system in the OS [94, 93, 29, 221, 39], or in file-system applications [169, 39, 132, 110]. We would prefer to avoid additional indirection mechanisms for byte-addressable PMs, which have lower access latency and offer a direct load/store interface.

We note that the OS already maintains a mapping of virtual to physical memory locations and that these mappings can be periodically updated to implement wear management without an additional translation layer. We build upon virtual memory to implement *Kevlar*, a software wear-management system for fast, byte-addressable persistent memories. Kevlar performs both wear leveling, by reshuffling pages among physical PM frames, and wear reduction, by judicious migration of wear-heavy pages to DRAM, to achieve a configurable lifetime target.

A critical aspect of wear management is to estimate the wear to each memory location. Existing hardware tracks PM writes only at the granularity of memory channels—too coarse to be useful for wear management. Tracking PM writes at finer granularity is complicated by write-back hardware caches; an update to a memory location leads to a PM write only when a dirty cache block is evicted from the processor's caches.

Kevlar relies upon a novel, low-overhead *wear-estimation* mechanism by using Intel's Precise Events Based Sampling (PEBS) [97], which allows us to intercept a sample of store operations. Kevlar maintains an approximate representation of hardware cache contents using Bloom filters [40], and uses it to estimate relative fine-grain writeback rates. We demonstrate that our estimation strategy incurs less than 1% performance overhead.

Kevlar enables wear management for applications that employ PMs for capacity expansion [16, 176, 116] and/or durability [160]. When a PM device is used for capacity expansion, Kevlar exploits memory device heterogeneity and migrates frequently updated PM pages to the neighboring DRAM—a system-level option that cannot be exploited by device-level wear-management schemes [173, 225, 183]. We show that migrating as few as 1% of pages from PM to DRAM is sufficient to achieve our target PM lifetime. For pages that require durability, Kevlar relies on reserve PM capacity and performs directed migrations of frequently written pages across the nominal and reserve capacity.

We implement Kevlar in Linux version 4.5.0 and evaluate its impact on performance and PM lifetime. To summarize, the contributions of Kevlar are:

- *Wear leveling:* We first develop an analytical framework to show that even a simple, wear-oblivious random page shuffling is sufficient to achieve near-ideal (uniform) wear over the memory device lifetime at negligible ($< 0.1\%$) performance overhead. Unfortunately, even ideal wear leveling provides insufficient lifetime for lower-endurance PMs.

- *Wear estimation:* We demonstrate how to estimate wear at fine grain by using Intel's PEBS to approximate cache contents via a Bloom filter, thereby estimating the cache write-backs to each page. We show that this mechanism is $21.7\times$ more accurate than

naive write sampling.

- *Wear reduction:* We demonstrate Kevlar, which uses our wear-estimation technique to apply both wear leveling and wear reduction, reducing wear by migrating less than 1% of the application working set to neighboring DRAM (when durability is not needed) incurring 1.2% (avg.) performance overhead.

3.2 Kevlar

We detail wear-management approaches in Kevlar.

3.2.1 Wear Leveling

Modern OSes, such as Linux, manage memory via a paging mechanism to translate virtual to physical memory addresses. Linux manages the page tables used by the hardware translation mechanism, and already reassigns virtual-to-physical mappings for a variety of reasons (e.g., to improve NUMA locality).

Kevlar's *Wear-Leveling* (WL) mechanism uses existing OS support to periodically remap virtual pages to spread writes uniformly. Kevlar makes a conservative assumption that a write to a physical PM page modifies all locations within that page. Thus, Kevlar does not need an additional intra-page wear-leveling mechanism. We observe that periodic random shuffling of virtual-to-physical mappings—migrating each virtual page to a randomly selected physical page frame—is sufficient to uniformly distribute writes to PM provided shuffles are frequent enough. A key advantage of this approach is that it is wear oblivious—it requires no information about the wear to each location; it only requires the aggregate write-back rate to memory, which is easily measurable on modern hardware. Surprisingly, we find that this simple approach may be acceptable for PM devices with a sufficiently high endurance (e.g., 10^9 writes).

We consider a scheme that periodically performs a random *shuffle* of all virtual pages,

reassigning each virtual page to a randomly selected physical page. Whereas our analysis assumes all pages are shuffled at once for simplicity, in practice, pages are shuffled continuously and incrementally over the course of the shuffle period. Our analysis poses the question: How many times must the address space be shuffled for the expected number of writes to each page to approach uniformity? Furthermore, at what point does the wear incurred by shuffling exceed the wear from the application? To simplify discussion, we use "write" to mean write-back from the last-level cache to the PM throughout this section.

Analysis. Let W represent the write distribution to physical pages and W_i be the write rate to i^{th} physical page in the memory. We define an equality function E as:

$$E(x,y) = \begin{cases} 1 & x == y \\ 0 & x! = y \end{cases} \tag{3.1}$$

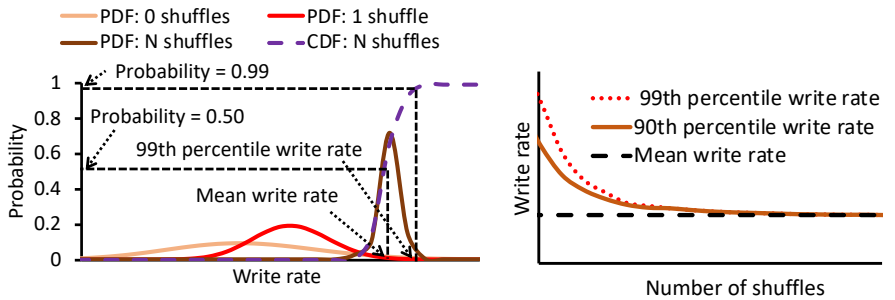
Given a write distribution W over n physical pages, P_n^k represents the probability density function (PDF) for W after k shuffles. Using the distribution W, we can compute the probability $P_n^0(x)$ of physical page with the write rate x with 0 shuffles (initial state) as:

$$P_n^0(x) = \frac{1}{n} \times \sum_{i=1}^{n} E(W_i, x) \tag{3.2}$$

With no shuffles, one can easily compute the expected life of each physical page by dividing the expected endurance (in number of writes) by the write rate x, yielding an expected lifetime distribution over pages. When we consider a shuffle's effect, each page will experience an average write rate x' of two write rates x_1 and x_2 chosen uniformly at random from W. Since the PDF of the sum of two random variables is the convolution of their respective PDFs, we can calculate the expected distribution of write rates after S shuffles, P_n^S, as:

$$P_n^S(X = x'/2) = \sum_{k=-\infty}^{\infty} P_n^{S-1}(X = k) P_n^{S-1}(X = x' - k) \tag{3.3}$$

Note the normalization by one half, since we want the average (rather than the sum) of the random variables.



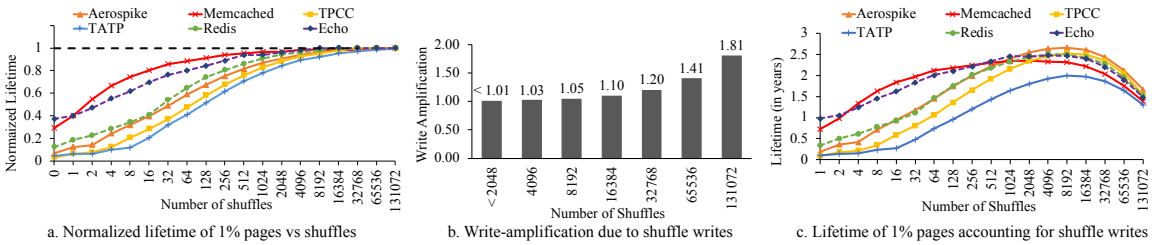
a. PDF and CDF for N shuffles

b. Write rate of nth percentile hot pages

Figure 3.1: **Write-back rate distribution with page shuffles.** (a) We use an application's write distribution to derive 99^{th} percentile write rate after N shuffles. (b) The disparity in page write rates shrinks with the increase in shuffles.

We illustrate the PDF P_n^0 (expected write rate without shuffles) of the page write distribution as expressed by Eq. 3.2 in Fig. 3.1 (a). The PDF P_n^0 has a heavy right-tailed distribution with high variance (*i.e.* the write-rate of few pages is high as compared to the mean write rate), a characteristic typical of the applications we have studied. Moreover, due to high variance, there is a wide write-rate range that might occur for any given page. Next, we compute the PDF P_n^S using Eq. 3.3 for shuffles ranging from one to N. With each shuffle, the PDF variance shrinks, while the probability of a near-mean write rate increases. Note that the PDF mean P_n^1 *appears* to be higher than the PDF P_n^0 due to the heavy right-tail of P_n^0. The mean in fact stays constant after each shuffle.

Fig. 3.1 (a) illustrates how the PDF after N shuffles converges to the mean write rate (equivalently, writes become uniformly distributed over the physical pages). In Figure 3.1 (a), we also show the cumulative distribution function (CDF) for N shuffles where the CDF C_n^N is used to compute the top n^{th} percentile of pages with the highest write rate after N shuffles (i.e., the "hottest" pages). $C_n^N(p)$ provides the minimum expected write rate of the most heavily written $(1-p)*100\%$ of the pages. For example, in Fig. 3.1 (a), we mark with a dotted line the 99^{th} percentile. The $C_n^N(p=0.99)$ gives the minimum expected write rate of the most heavily written 1% of pages after N shuffles. From this rate, we can estimate when we expect this 1% of pages to have worn out. As the number of shuffles grows, the variance shrinks and $C_n^N(p=0.99)$ approaches the mean write rate.



a. Normalized lifetime of 1% pages vs shuffles

b. Write-amplification due to shuffle writes

c. Lifetime of 1% pages accounting for shuffle writes

Figure 3.2: **PM lifetime with Kevlar's wear-leveling mechanism.** (a) Lifetime of 1% of pages vs. shuffles: The expected lifetime converges to the ideal lifetime for shuffles > 8192, (b) Write-amplification due to shuffle writes: Kevlar performs 5% additional writes with 8192 shuffles, (c) Lifetime of 1% of pages, accounting for shuffle writes: The lifetime of PM peaks at 8192 shuffles, following which shuffle writes become significant.

We illustrate how the write rate of the hottest pages compares to the mean as a function of the number of shuffles in Fig. 3.1 (b). Note that our approach can estimate the wear rate at any percentile, but we present results primarily for the 99^{th} percentile. Without shuffles, there is a large disparity between the most-written 1% of pages and the mean. The gap rapidly shrinks with additional shuffles. Given the hottest pages' write rates in Fig. 3.1(b), we compute lifetime of a device with a 10^7 write endurance.

Tracing Methodology. We collect write-back traces for a set of applications (detailed in Section 3.4) using the DynamoRio [44] instrumentation tool and its online cache simulation client `drcachesim`. Since `drcachesim` can simulate only a two-level cache hierarchy with power-of-two cache sizes, we model an 8-way 256KB L2 cache and 32MB 16-way associative L3 cache, which is close to the configuration of the physical system on which we evaluate our Kevlar prototype (described in Table 3.1). We instrument loads and stores to trace all memory references and run `drcachesim` online to simulate the system's cache hierarchy. We record writebacks from the simulated LLC to PM. We then extract write rate distributions to analyze expected PM lifetime under shuffling.

Determining optimal shuffles. In Fig. 3.2(a), we show the lifetime, normalized to what is possible under ideal wear leveling, as a function of the number of shuffles. We assume some redundancy in the PM device similar to prior works [173, 174] and define its lifetime as the time when 1% of pages are expected to fail. Note that the lifetime under

ideal wear leveling is the device endurance divided by the application's average write-back rate. As shown in Figure 3.2(a), frequently written virtual pages are mapped to a different set of physical pages after every shuffle, leading to improved device lifetime with more shuffles. Interestingly, for all applications, after about 8192 shuffles, the expected lifetime converges to that of ideal wear leveling (i.e., the write distribution is uniform). Note that we do not consider the additional writes incurred due to remapping virtual-to-physical page mappings after each shuffle in Figure 3.2(a).

Figure 3.2(b) shows the write amplification caused due to the shuffle operations. The write amplification shows the ratio of the total writes incurred after shuffling as compared to the application's PM writes. The write amplification can be higher than 1.4x (40% additional writes) for greater than 2^{16} shuffles as shown in Figure 3.2(b).

Peak lifetimes occur when memory is shuffled 8192 times over the device lifetime. With 8192 shuffles, we perform 5% additional writes for wear leveling. Fig. 3.2(c) shows the writes due to shuffle operations, which may grow to dwarf the application's writes if shuffles are too frequent (*i.e.* >16384).

Discussion. Shuffling memory 8192 times over the PM device lifetime uniformly distributes PM writes. However, the lifetime achievable via even ideal wear leveling is limited by an application's average write rate. For our applications, this lifetime is only 2.3 to 2.8 years for a device that wears out after 10^7 writes (see Fig. 3.2(c)). Wear leveling alone may be insufficient to meet lifetime targets.

To achieve desired lifetimes, we must augment Kevlar's wear-leveling mechanism with a wear-reducing mechanism. The key challenge for wear reduction is to monitor the wear to each virtual page at low overhead. There is no straightforward mechanism for the OS to directly monitor device wear at fine granularity. PM devices incur wear only when writes reach the device. Write-back caches absorb much of the processor write traffic, so the number of stores to a location can be a poor indicator of actual device wear. Current x86 hardware can count writebacks per memory channel, but provides no support for finer-grain

(e.g., page or cache line) monitoring. Mechanisms that monitor writes via protection faults (e.g., [16, 72]) incur high performance overhead and fail to account for wear reduction by writeback caches, grossly overestimating wear for well-cached locations. Instead, Kevlar builds a software mechanism to estimate per-page wear intensity.

3.2.2 Wear Estimation

We design a wear-estimation mechanism that approximately tracks hardware cache contents to estimate per-page PM write-back rates. Our mechanism builds upon Intel's PEBS performance counters [101] to sample store operations executed by the processor. Note that, although we focus on Intel platforms, other platforms—AMD Instruction Based Sampling [63] and ARM Coresight Trace Buffers [21]—provide analogous monitoring mechanisms. Kevlar's write estimation mechanism monitors the retiring stores to maintain an estimate of hardware cache contents.

Monitoring stores. PEBS captures a snapshot of processor state upon certain configurable events. We configure PEBS to monitor `MEM_UOPS_RETIRED.ALL_STORES` events. As stores retire, PEBS can trigger an interrupt to record state into a software-accessible buffer; we record the virtual address accessed by the retiring store.

Although accurate, sampling every store with PEBS is prohibitive. Instead, we rely on systematic sampling to reduce performance overhead: we configure PEBS with a *Sample After Value* (SAV). For a SAV of n, PEBS captures only every n^{th} event. Like prior work [145], we choose prime SAVs to avoid bias from periodicities in the systematic sampling. We explore the accuracy and overhead of SAV alternatives in Section 3.5.1.

We obtain the virtual addresses of sampled stores to estimate per-page write-back rates. A naive strategy to compute write-back rates is to assume that each sampled store results in a write-back. However, with write-back hardware caches, a PM write occurs only when a dirty block is evicted from the cache hierarchy; many stores coalesce in the caches. Indeed, in our applications, the naive strategy drastically over-estimates writebacks (see

Section 3.5.1). Consequently, we design an efficient software mechanism that estimates temporal locality due to hardware caches to predict which stores incur write-backs.

Estimating temporal locality. Prior mechanisms have been proposed to estimate temporal locality in storage [202, 203] or multicore [182, 181, 33] caches. These mechanisms maintain stacks or hashmaps to compute reuse distances for accesses to sampled locations. Instead, we focus on modeling temporal locality in hardware caches to estimate LLC write-backs using sampled stores. We estimate temporal locality by using a Bloom filter [40] to approximately track dirty memory locations stored in the caches. For each store sampled by PEBS, we insert its cache block address into the Bloom filter. (Algorithm 1: Line 12-14). Whenever a new address is added to the filter, we assume it is the store that dirties the cache block, and hence will eventually result in a writeback. Further stores to the same cache block will find their address already present in the Bloom filter; we assume these hit in the cache and hence do not produce additional write-backs. Thus, the Bloom filter maintains a compact representation of likely dirty blocks present in the cache.

Bloom filters have a limited capacity; after a certain number of insertions into the set, their false positive rate increases rapidly. We size the Bloom filter such that it can accurately (less than 1% false positives) track a set as large as the capacity of the processor's last-level cache (LLC), which is roughly 700K cache blocks on our evaluation platform. We clear the Bloom filter when the number of insertions reaches this size (Algorithm 1: Line 19-29).

Of course, after clearing the filter, Kevlar would predict a sudden false spike in write-back rates. We address this by using two Bloom filters; Kevlar probes both filters but inserts into only one "active" filter at a time (Algorithm 1: Line 3, 12-17). When the active filter becomes full, we clear the inactive filter and then make it active. As such, at steady state, one filter contains 700K cache block addresses, while the other is active and being populated (Algorithm 1: Line 12-17). We assume a cache block will result in a store hit (no additional writeback) if it is present in either filter (Algorithm 1: Line 6-10).

In essence, our tracking strategy filters out cache blocks that have write reuse dis-

tances [117] of about 700K or less, as such writes are likely to be cache hits. Effectively, we assume that dirty blocks are flushed from the cache primarily due to capacity misses, which is typically the case for large associative LLCs [88, 219]. Note that our estimate of the cache contents is approximate. For example, the Bloom filters do not track read-only cache blocks. Moreover, due to SAV, only a sample of writes are inserted. The mechanism works despite these approximations because: (1) frequently written addresses are likely to be sampled and inserted into the filters—it is these addresses that are most critical to track; and (2) few addresses have reuse distances near 700K—reuse distances are typically much shorter or longer, so the filters are effective in estimating whether or not a store is likely to hit. Although Kevlar approximates writebacks by sampling retiring stores, our goal in Kevlar is to measure relative hotness of the pages as opposed to absolute writebacks per page. We show the accuracy of our estimation mechanism to identify writeback intensive pages later in Section 3.5.1.

Estimating write-backs. PEBS provides the virtual address of sampled stores. Our handler then walks the software page table to obtain the corresponding physical frame (Alg. 1: Line 7). In our Linux prototype, we maintain a writeback count in `struct page`, a data-structure associated with each page frame. When we sample a store, we update the counter for the corresponding physical page as shown in Alg. 1: Line 8. Kevlar uses the estimated writebacks to identify writeback-intensive pages.

3.2.3 Wear Reduction

As shown in Sec. 3.2.1, Kevlar's wear-leveling mechanism can achieve only 2.3- to 2.8-year lifetime for a PM device that wears out after 10^7 writes. Our goal is to achieve a lifetime target for a low-endurance PM device by migrating heavily written pages to DRAM. We assume a nominal lifetime goal of four years. This target is software-configurable; we discuss longer targets in Section 3.5.2.

Consider an application with a memory footprint of N physical PM pages and a given

lifetime target, the write rate to the PM B writes/sec to achieve the lifetime target can be computed as:

$$B = \frac{Endurance \times N}{Lifetime} \tag{3.4}$$

We use Eq. 3.4 to compute the number of writes the application may make per 1GB (*i.e.* N = 256K small pages) of PM footprint. For a given lower-bound endurance of 10^7 writes and a 4-year lifetime, writebacks must be limited to 20K writes/sec/GB. Configuring a different target lifetime or device endurance changes the allowable threshold.

One approach is to use wear leveling (as described in Sec. 3.2.1) by provisioning additional reserve capacity such that the target lifetime is met. This strategy is applicable both when PM is used for persistent storage or capacity expansion. For instance, with N pages in an application, and average write rate of B' writes/sec/GB, the reserve capacity R to achieve a 4-year lifetime is given by:

$$R = \frac{N \times B'}{2 \times 10^4} \tag{3.5}$$

When the application write rate is high relative to the device endurance, the required reserve can undermine any cost advantages, as we show later in Section 3.5.3. Instead, for capacity expansion, we propose wear reduction by migrating the hottest pages to high-endurance memory (DRAM). Kevlar regulates the average write rate to the pages that remain in PM to 20K writes/GB/sec such that we achieve the desired lifetime of four years.

3.2.3.1 Page migration

Kevlar uses its write-back estimation mechanism to measure per-page PM writeback rates and migrate the most write-intensive pages to DRAM. Kevlar must regulate average PM writeback rate to 20K writes/GB/sec to achieve a 4-year lifetime. Kevlar uses `IMC.MC_CHy_PCI_PMON_CTR` counters in the memory controller to count `CAS_COUNT.WR` events, which measure write commands issued on the memory channels. Such counters

Algorithm 1 Write-back estimation mechanism

```
1: Inputs:
       PEBS record rec, Bloom Filter filterA, Bloom Filter filterB
2:
3: Initialize:
       filterA.isActive = True
       filterB.isActive = False
       activate = LLC_CACHE_BLOCKS
4:
5: blockAddr = rec.strAddr » log2(LLC_BLOCK_SIZE)
6: if !filterA.isPresent(blockAddr) and !filterB.isPresent(blockAddr) then
7:     pageStruct = doPageWalk(blockAddr)
8:     pageStruct.WBCount+=1
9:     memRef+=1
10: end if
11:
12: if filterA.isActive and !filterA.isPresent(blockAddr) then
13:     filterA.add(blockAddr)
14: end if
15: if filterB.isActive and !filterB.isPresent(blockAddr) then
16:     filterB.add(blockAddr)
17: end if
18:
19: if activate == memRef then
20:     filterA.isActive = !filterA.isActive
21:     filterB.isActive = !filterB.isActive
22:     if filterA.isActive then
23:         filterA.clear()
24:     end if
25:     if filterB.isActive then
26:         filterB.clear()
27:     end if
28:     activate+=LLC_CACHE_BLOCKS
29: end if
```

already exist in DRAM controllers, and analogous counters exist on other hardware platforms (*e.g.* ARM's L3D_CACHE_WB performance monitoring unit counter [23]). This aggregate measure allows us to determine whether pages must be migrated from PM to DRAM (or can be migrated back) to maintain the target average rate of 20K writes/GB/sec.

Migrating hot-pages to DRAM. Kevlar computes the PM writeback rate at a fixed 10-second interval. If the average writeback rate exceeds 20K writes/GB/sec during an interval, Kevlar enables PEBS and samples the retiring stores as explained in Section 3.2.2. Kevlar estimates the PM writeback rate at 4KB-page granularity. When migration is needed, Kevlar scans writeback counters for all page frames and sorts them by their estimated write-back counts. Kevlar then migrates the hottest 10% of pages to DRAM. It continues monitoring for an additional interval. Kevlar ceases migration, disables PEBS monitoring, and clears write-back counters when the write-back rate falls below 20K writes/GB/sec. With

this monitoring and migration control loop, Kevlar achieves our lifetime target with 1.2% performance impact.

Migrating cold pages to PM. An application's access pattern might change over its execution, so pages migrated to DRAM may become cold. To minimize the application footprint in DRAM, it is desirable to migrate cold pages back to PM. If Kevlar observes five consecutive intervals with a PM writeback rate below 20K writes/GB/sec, it re-enables PEBS for a 10-second interval, estimates the write-back rate of pages in DRAM, and migrates 10% of cold pages from DRAM back to PM.

3.3 Implementation

We implement Kevlar in Linux kernel version 4.5.0. We use the Linux control group mechanism [152] to manage Kevlar specific configuration parameters.

Wear leveling. Kevlar should shuffle the entire application footprint once every 4.2 hours to achieve uniform wear leveling over a lifetime of 4 years. Instead of gang-scheduling the shuffle operations together every 4.2 hours, Kevlar periodically shuffles a fraction of application footprint. Kevlar maintains a shuffle bit in the `struct page` associated with each page frame to indicate whether the page was shuffled within the current shuffle interval. Kevlar scans the application pages every 300-sec *shuffle interval* to identify the pages that are yet to be shuffled. It randomly chooses a fraction of pages to be shuffled in this shuffle interval by equally apportioning the total number of pages yet to be shuffled to the time remaining in a 4.2 hour shuffle operation.

The fraction of pages are then shuffled following these steps: (1) Kevlar selects a pair of application pages in PM to be swapped. (2) It locks the page table entries for both pages so that any intermediate application accesses stall on page locks. (3) It allocates a temporary page in DRAM (for capacity workloads) to aid in swapping the contents of the two pages in PM. (4) Once the pages are swapped, Kevlar restores the page table entries so that the virtual addresses now map to the swapped pages, unlocks the pages, and deallocates the

temporary DRAM page. (5) Once shuffled, Kevlar records this event in the shuffle bit in page frame's `struct page` of the two pages.

Note that, we use a temporary page mapped in DRAM to limit wear in PM due to shuffle. For persistent applications, we map the temporary page in PM to ensure that the page contents are persistent in case of intermediate failure. Once all the pages are shuffled, Kevlar clears the shuffle bit in `struct page` and initiates the next shuffle.

Wear estimation. Kevlar initializes PEBS with `MEM_UOPS_RETIRED.ALL_STORES` event and a SAV to sample the retiring stores for wear estimation. We determine SAV empirically to ensure that the monitoring has negligible performance overhead. Kevlar implements two Bloom Filters, each of size 840KB and a capacity of 700K cache blocks, corresponding to the 45MB LLC of our system. We size the Bloom filter to achieve less than 1% false positives. As explained in Section 3.2.3.1, Kevlar performs a software page table walk to identify the page frames being accessed by the sampled store, and records writeback counts in `struct page`.

Wear reduction. Kevlar monitors PM writeback rate at a 10-second *migration interval* to determine if it needs to initiate hot/cold page migration between DRAM and PM. If the PM writeback rate triggers a migration, Kevlar scans the application pages and identifies the top 10% hot (or cold) pages to be migrated to DRAM (or PM). It performs migration using a mechanism similar to the page shuffles in wear leveling: it locks the page to be migrated, copies its contents to a newly allocated page in DRAM (or PM), updates page table entries, and unlocks the page. If no migration is triggered, Kevlar disables PEBS sampling counters to minimize performance monitoring overhead.

3.4 Methodology

We next discuss details of our prototype and evaluation.

Core	Intel Xeon E5-2699 v3, 2.30GHz 36-core (72 hardware threads) Dual-socket x86 server
L1 D&I Cache	32KB, 8-way associative
L2 Cache	256KB, 8-way associative
Shared LLC	45MB, 20-way associative
DRAM	256GB per socket
Operating System	Linux Kernel 4.5.0

Table 3.1: **System Configuration.** Server configuration used for our evaluation.

3.4.1 Emulating Persistent Memory

A system with byte-addressable persistent memory is not yet commercially available. Hence, we emulate a hybrid PM-DRAM memory system using a dual-socket server. We run the application under test on a single socket and treat memory local to that socket as DRAM. Conversely, we treat memory of the remote socket as PM. Note that the local and remote nodes are cache coherent across the sockets. Since each chip has its own memory controllers, we use the performance counters in each memory controller to monitor the total accesses to each device and distinguish "PM" and "DRAM" accesses.

Using this emulation, our Kevlar prototype incurs the actual performance overheads of monitoring and migration that would occur in a real hybrid-memory system. However, the latency and bandwidth differential between our emulated "PM" and "DRAM" is only the gap between local and remote socket accesses. The performance differential between DRAM and actual PM devices is technology dependent and remains unclear, but is likely higher than in our prototype. We expect relative performance overhead of our mechanism (as detailed later in Section 3.5.4) to be lower on a system with a high differential between DRAM and PM devices. Our results represent a high estimate of the Kevlar's performance overhead.

Nevertheless, our contributions with respect to wear management are orthogonal to the performance aspects of replacing DRAM with PM, which have been studied in prior work [16, 116, 172]. We focus our evaluation on quantifying the effectiveness and overheads of Kevlar's mechanisms.

3.4.2 System Configuration

We run our experiments on a dual-socket server with the configuration listed in Table 3.1. We use the Linux control group mechanism [152] to isolate the application to a particular socket. We pin application threads to execute only on CPUs on the local node, but map all memory to initially allocate in the remote node using Linux's `memory` and `cpuset cgroups`, modeling a system where DRAM has been replaced by PM. Kevlar expects a lifetime goal for the PM device as an input, and performs wear leveling, estimation, and reduction for all the processes in the `cgroups`. The test applications use all 18 CPU cores of the local node with hyper-threading enabled. For client-server benchmarks, we run clients on another system to avoid performance interference.

As explained in Section 3.2.2, we use Intel's PEBS counters to estimate PM page writeback frequency. We isolate these counters to monitor only accesses from the application under test using Linux's `perf_event` cgroup mechanism. Thus, spurious store operations from background processes or the kernel do not perturb our measurements.

We measure the write rate to the PM (*i.e.* remote DRAM) using the performance counters in the memory controller. Unlike PEBS counters, these counters lie in a shared domain and cannot be isolated to count only events for a particular process. However, we have measured the write rate of the background processes in an idle system and find that they constitute less than 1% of the total writeback rate observed during our experiments.

3.4.3 Benchmarks

We study two categories of applications. We report memory footprints of the benchmarks under study in Figure 3.8.

3.4.3.1 Capacity Expansion Workloads

We evaluate both the wear-leveling and wear-reduction mechanisms of Kevlar for the following benchmarks in a "capacity expansion" PM use case.

NoSQL applications. Aerospike [1, 192], and Memcached [14] are popular in-memory NoSQL databases. We use YCSB clients [57] to generate the workload to Aerospike and Memcached. We evaluate 400M operations on 4M keys for Aerospike and 100M operations on 1M keys for Memcached. We configure each record to have 20 fields resulting in a data size of 2KB per record. As we are interested in managing wear in write-intensive scenarios, we configure YCSB for update-heavy workload with a 50:50 read-write ratio and Zipfian key distribution.

MySQL. MySQL is a SQL database management system. We drive MySQL using the open-source TPCC [197] and TATP [164] workloads from oltpbench[60]. TPCC models an order fulfillment business and TATP models a mobile carrier database. In each, we run default transactions with a scale-factor of 320 for 1800 secs.

3.4.3.2 Persistent Workloads

We evaluate persistent applications from the WHISPER benchmark suite [160], which use the Intel PMDK libraries [8] for persistence. These applications divide their address space into volatile and persistent subsets. The persistent subset must always be mapped to PM to ensure recoverability in the event of power failure. As such, Kevlar may not migrate pages in the persistent subset to DRAM. We instead rely only on wear leveling to shuffle these pages in PM. However, we allow pages in the volatile subset to migrate to DRAM if the aggregate write rate to all pages exceeds 20K writes/GB/sec.

Linux presently provides no mechanism to label pages as persistent or volatile. WHISPER benchmarks use Linux's `tmpfs` [191] memory mapped in DRAM to emulate persistency, and the persistent pages are allocated in a fixed address range. We hardcode this address range in our experiments to prevent page migrations to DRAM.

We select the two NoSQL applications, Redis and Echo, from WHISPER. Redis is a single-threaded in-memory key-value store. We configure a Redis database comprising 1M records, each with 10 fields. We use YCSB clients to perform key-value operations on the

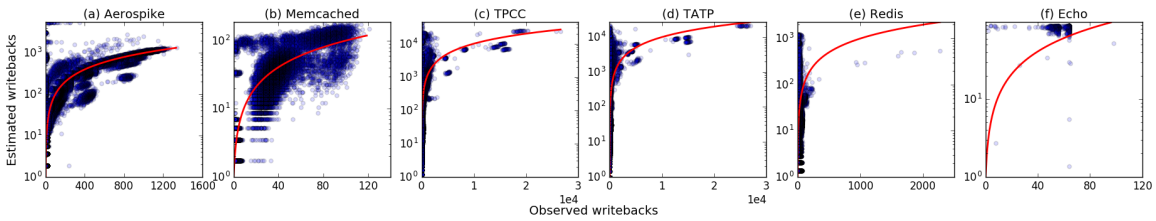


Figure 3.3: **Estimated writebacks vs. observed writebacks.** We compare the estimated writebacks with observed writebacks obtained from memory access tracing. Each point on the scatter plot represents the number of writebacks to a page. The red line on each plot represents the ideal prediction curve.

Redis server with a Zipfian distribution. For our evaluation, we run 40M operations with an update-heavy workload with a 50:50 read-to-write ratio. For echo, we use the configuration provided with the WHISPER benchmark suite and evaluate it using 2 client threads each running 40M operations.

3.5 Evaluation

We evaluate Kevlar's wear-management mechanisms.

3.5.1 Modeling Wear Estimation

We first evaluate the accuracy of Kevlar's wear-estimation mechanism as described in Section 3.2.2. We collect a ground-truth writeback trace for each application using the online cache simulator `drcachesim` in Dynamorio [44] with a tracing infrastructure described in Section 3.2.1. We model the PEBS sampling mechanism and bloom filters in `drcachesim` to record the estimated writeback rate. We compare the ground-truth writebacks against the estimates provided by the emulation of PEBS sampling and our Bloom filters.

Comparison with ideal mechanism. In Figure 3.3, we show estimated writebacks (vertical axis) and ground-truth observed writebacks (horizontal axis) for each application for one 10-sec sampling interval. We use log-linear scale[1] to highlight accuracy of our

[1]We use log-linear scale to highlight estimated and observed writebacks to hot pages that are crucial for

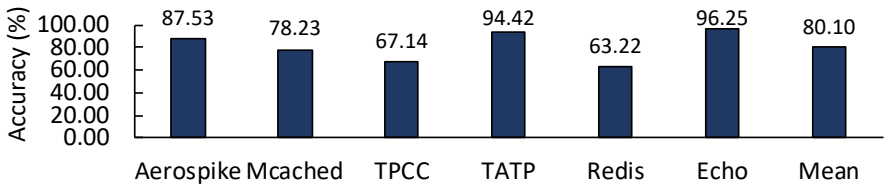


Figure 3.4: **Comparison of top 10% estimated hot pages to top 10% observed hot pages.** Kevlar's wear estimation identifies 80.10% (avg.) of the 10% hottest written pages correctly.

mechanism for higher write rate. As instrumentation results in application slowdown, we expand the 10-second sampling duration by the slowdown due to instrumentation measured for each workload. Due to the log-linear scale, we plot a red curve in the Figure to show the ideal prediction curve, where estimated and observed writebacks match. For all applications, Figure 3.3 (a-f) indicates that the estimated writebacks correlate closely to the ideal curve. Echo performs cache flush operation following each store to flush dirty cache blocks to PM. As a result, we observe 64 write-backs per page (owing to 64 cache blocks in a 4KB page) for nearly all pages. As shown in Figure 3.3(f), Kevlar is able to measure write-backs to these pages.

Prediction accuracy. Next, we compare the top 10% heavily written pages as estimated by Kevlar's wear-estimation mechanism to the top 10% hottest observed (ground-truth) pages. Figure 3.4 shows the percentage of heavily written pages correctly estimated by Kevlar. Kevlar correctly estimates 80.1% hottest pages on average and up to 96.3% hottest pages in Echo as compared to the ground truth.

We also demonstrate the accuracy of Kevlar's prediction mechanism by measuring root-mean-squared (RMS) error between estimated and observed writebacks. The RMS error reports the standard-deviation of the difference between estimated and observed writebacks. We study the impact of hardware cache modeling using our Bloom filter mechanism by comparing Kevlar's prediction mechanism with a mechanism without the Bloom filter. Figure 3.5 shows the RMS error of our writeback prediction mechanism normalized to

our study. In contrast, a log-log scale discretizes lower writeback values and hides comparison between observed and estimated writebacks for hot pages.

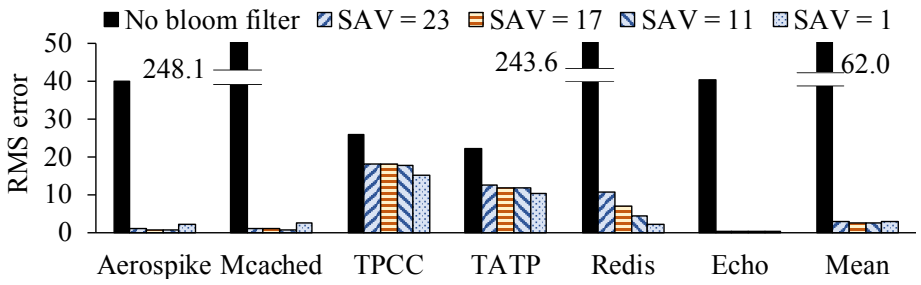


Figure 3.5: **RMS Error with cache modeling.** Kevlar achieves 20× lower RMS error than a mechanism without cache modeling.

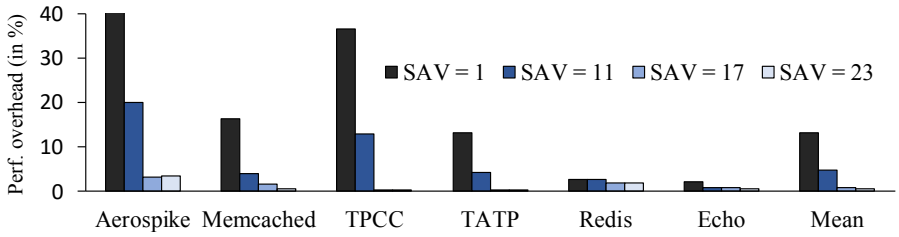


Figure 3.6: **PEBS sampling overhead.** Runtime overhead due to sampling every retiring store is 13.2% (avg.). We configure PEBS SAV = 17 in Kevlar with < 1% overhead.

the average writeback rate of the application for different PEBS SAV values. We choose prime numbers for PEBS SAV to avoid periodicities in systematic sampling.

As compared to a mechanism that does not model cache contents, we observe 100.0× and 106.8× improvement in RMS errors for Memcached and Redis, respectively, with our estimation mechanism (with SAV = 1). Overall, the Bloom filters can approximate the dirty cache contents well, allowing it to estimate writebacks with 21.6× lower RMS error on average. The Bloom filters are critical to avoiding overestimation of writebacks in Aerospike, Memcached, and Redis by estimating temporal locality of memory accesses. Note that, as shown in Figure 3.5, the standard deviation of the difference between absolute values of estimated and observed writebacks is 2.85× that of the mean for SAV of 1. Although the estimated writebacks are not accurate when compared to absolute values, our goal in Kevlar is to measure the relative hotness of the pages. As shown earlier in Figure 3.4, Kevlar identifies 80.1% of the 10% hottest pages correctly.

Configuring PEBS SAV. We study the RMS error in Figure 3.5 and runtime performance overhead in Figure 3.6 for different PEBS SAV values. Figure 3.6 shows the mon-

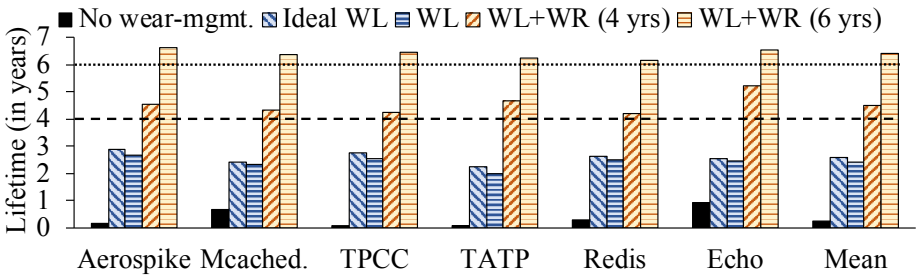


Figure 3.7: **PM Lifetime.** Kevlar achieves greater than 4 years of lifetime; 11.2× (avg.) higher than no wear leveling.

itoring overhead for different SAVs when compared to the application runtime without PEBS monitoring. Upon sampling a store, PEBS triggers an interrupt and records architectural state in a software buffer, which can lead to a performance overhead. Taking an interrupt on every retiring store results in substantial performance overhead. Indeed, with SAV=1, the performance overhead due to PEBS sampling can be as high as 112.9% (in Aerospike), and 13.2% on an average. In contrast, the performance overhead in persistent applications, Redis and Echo, is less than 3% as we sample only stores to volatile pages, which may be migrated between PM and DRAM. Interestingly, with SAV of 17, the average performance overhead due to sampling is less than 1% (avg.) with no substantial degradation in RMS error. As we do not see any substantial performance gains for SAV > 17, we configure PEBS to sample one in every 17 stores in Kevlar.

3.5.2 PM Lifetime

We study Kevlar for lifetime targets of four and six years. We compare Kevlar's wear-management mechanisms to a baseline with no wear leveling. We make a conservative assumption that a write to a physical page modifies all locations within that page for Kevlar's wear-management mechanisms. In contrast, we measure lifetime for the baseline via precise monitoring at cache-line granularity.

Wear leveling alone. We first consider lifetime for the PM device achieved by Kevlar's wear-leveling mechanism alone. As discussed in Section 3.2.3, to achieve a four- (or six-)

year lifetime until 1% of locations wear out on a PM device that can sustain only 10^7 writes, the average write rate must be below 20,000 (or 13,333) writes/GB/second. Even after wear leveling, all of the applications we study incur a higher average write rate when their entire footprints reside in PM. We also show lifetime due to ideal wear leveling in Figure 3.7 when writes are uniformly remapped in PM. Although wear leveling substantially improves PM lifetimes over a baseline of no wear leveling, it falls short of achieving the four-year and six-year lifetime targets for all applications. As compared to the baseline with no wear leveling, Kevlar with only wear leveling achieves an average lifetime improvement of 9.8× with 31.7× improvement in lifetime for TPCC.

Wear leveling + wear reduction. Wear reduction can improve application lifetimes to meet our target while moving only a remarkably small fraction of the application footprint to DRAM. Kevlar in wear leveling + wear reducing mode aims to limit the write-back rate to the PM at 20K (or 13.3K) write/GB/second for four (or six) year lifetime target, by identifying the "hottest pages" that are being frequently written back and migrating them to DRAM.

Owing to the writeback rate limit imposed by Kevlar's wear-reducing mechanism, as indicated in Figure 3.7, the lifetime with wear leveling + wear reduction exceeds the configured target of four and six years for all applications. Kevlar's wear leveling + wear reduction mode (for a 6-year lifetime configuration) achieves the highest lifetime improvement of 80.7× for TPCC, with an average improvement of 26.1× when compared to no wear leveling.

High-endurance PMs: Absent wear-management mechanisms, a PM device that can sustain 10^8 writes would wear out within 9.8 months. Moreover, for PM devices with endurance 10^8 - 10^9, wear-leveling mechanism would be sufficient to achieve the desired lifetimes of 4- and 6-years. For instance, our wear-leveling mechanism alone can achieve a lifetime of 24.0 years (average) for a PM device that can sustain 10^8 writes. Kevlar would not trigger wear-reduction mechanism for PMs with high write endurance as the

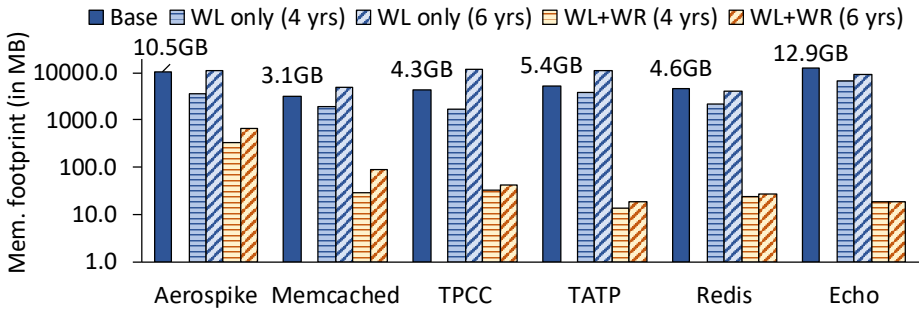


Figure 3.8: **Application footprint in PM and DRAM.** Kevlar migrates $< 1\%$ of application footprint to DRAM. Blue and orange bars represent application footprint in PM and DRAM respectively.

application write-back rate would be lower than configured threshold. Nevertheless, the endurance numbers of commercial PM devices (i.e. Intel's 3D XPoint) are not publicly available. As such, we can configure the endurance of a PM device in Kevlar.

3.5.3 Memory Overhead

Figure 3.8 shows the baseline memory footprint of the applications, and an additional memory footprint in DRAM necessary to host the most frequently written PM pages that are migrated by Kevlar. In addition, we also show the reserve footprint that can be mapped in PM to achieve the lifetime targets using wear-leveling mechanism alone as outlined in Equation 3.5.

Wear reduction for persistency applications. For the WHISPER benchmarks that rely on persistency (Redis & Echo), the pages in the persistent set must always remain in PM. Nevertheless, some fraction of these applications' footprints are volatile and may reside in PM or DRAM. We initially map the entire footprint to the PM and allow only volatile pages to migrate to DRAM. As a majority of memory accesses are made to the volatile footprint in these applications [160], the wear-reducing mechanism can achieve a 4 year lifetime by migrating only 23.6MB of footprint to DRAM.

Reserve PM required can be significant. The amount of PM reserves required to ensure that the target lifetime be met are significant. It can be as high as 2.7× for TPCC and 2.0× for TATP for a six-year lifetime (1.3× average across all the benchmarks). The

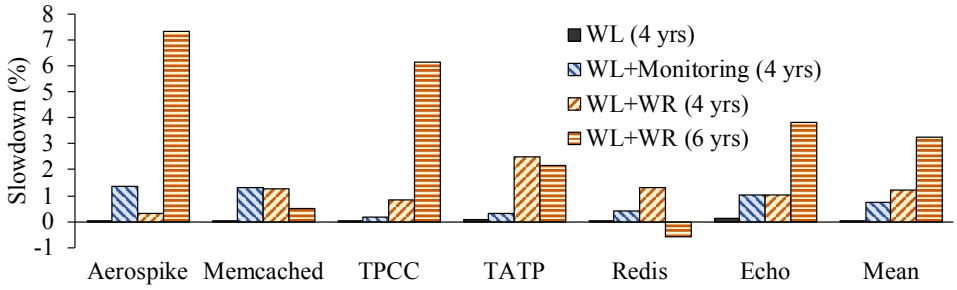


Figure 3.9: **Performance overhead.** Overhead of page monitoring and migration in Kevlar is 1.2% (avg.) in our applications.

required reserve capacity may undermine the cost advantages of capacity expansion offered by PMs.

Reserve DRAM required is much smaller than reserve PM. As can be seen from Figure 3.8, the reserve DRAM required is much smaller than the reserve PM required. This difference is due to a difference in the write endurance of DRAM (practically infinite) and the cell endurance we assume for PM (10^7 writes). Note that Kevlar's goal is to limit wear while maximizing application footprint in PM (especially for the capacity expansion use-case) and achieve configured device lifetime. Thus, it migrates only the heavily written application footprint from PM to DRAM. In contrast, prior mechanisms [16, 116] aggressively migrate pages to DRAM and limit application performance degradation resulting from slower PM accesses. Kevlar migrates less than 1% of the application's footprint to DRAM for four- and six-year lifetime targets, on average.

3.5.4 Performance Overhead

Next, we present application slowdown due to Kevlar.

Page shuffle overhead. Figure 3.9 illustrates the slowdown (lower is better) in applications resulting from our wear leveling, wear estimation, and page migration. The shuffle mechanism incurs a negligible average performance overhead of 0.04% (highest 0.1% in Echo) over the baseline with no wear leveling.

Overheads from Kevlar's monitoring and migration. As explained in Section 3.2.3,

we configure PEBS with SAV of 17, and further reduce performance overhead by filtering store addresses using the Bloom filters. We observe up to 1.3% slowdown from our PEBS sampling in Aerospike, with even lower overheads in the remaining applications. Redis observes a net gain (as much as 0.9%) when we enable migration and relocate their frequently written pages to DRAM because the local NUMA node (representing DRAM) is faster than the remote node (representing PM) in our prototype. We expect the performance gains to be more pronounced with PMs that are anticipated to exhibit higher memory latency than remote DRAM in our prototype. On an average, we see 1.2% (or 3.2%) slowdown due to our wear-management mechanisms to achieve the lifetime goal of four (or six) years.

CHAPTER IV

Persistency for Synchronization-Free Regions

4.1 Introduction

Emerging persistent memory (PM) technologies, such as Intel and Micron's 3D XPoint, aim to combine the byte-addressability of DRAM with the durability of storage [102]. Unlike traditional storage devices, which provide only an OS-managed block-based interface, PM offers a load-store interface similar to DRAM. This interface enables fine-grained updates and avoids the hardware/software layers of conventional storage, lowering access latency.

Although PM devices are nascent, the best way to integrate them into our programming systems remains a matter of fierce debate [171, 201, 54, 64, 122, 112, 99]. The promise of PM is to enable data structures that provide the convenience and performance of in-place load-store manipulation, and yet persist across failures, such as power interruptions and OS or program crashes. Following such a crash, volatile program state (DRAM, program counters, registers, etc.) are lost, but PM state is preserved. A *recovery* process can then examine the PM state, reconstruct required volatile state, and resume program execution. The design of such recovery processes is well studied in specialized domains, such as databases and file systems [157, 51, 83, 140], but open questions remain for general programming systems.

Reasoning about the correctness of recovery code requires precise semantics for the

allowable PM state after a failure [171, 52, 201, 56, 54, 47]. Specifying such semantics is complicated by the desire to support concurrent PM accesses from multiple threads and optimizations that reorder or coalesce accesses.

Recent work has proposed memory *persistency models* to provide programmers with such semantics [98, 22, 171, 56, 112]. Such models say that a PM access has *persisted* when the effects of that access are guaranteed to be observed by recovery code in the event of a failure. Similar to memory consistency models, which govern the visibility of writes to shared memory, persistency models govern the order of persists to PM. Notably, many persistency models allow the *persist* of a PM access to lag its visibility, enabling overlap of long PM writes with subsequent execution. Both industry [98, 22] and academia [56, 171, 62] have proposed candidate persistency models, but most of these have been specified at the abstraction level of the hardware instruction set architecture (ISA). Such ISA-level persistency models do not specify semantics for higher-level languages, where compiler optimizations may also reorder or elide PM reads and writes.

Language-level persistency [123] proposes extending the memory models of high-level languages, like C++11 and Java, with persistency semantics. In this paper, we argue that the language-level semantics proposed to date, *Acquire-Release Persistency* (ARP) for C++11, are deeply unsatisfying, as they fail to extend the "sequential consistency for data-race-free programs (SC for DRF)" guarantee enjoyed in fault-free execution to recovery code [42]. ARP specifies semantics that prescribe ordering constraints at the granularity of individual accesses. Although ARP bounds the latest point (with respect to other memory accesses) at which a PM store may persist, it does not generally preclude PM stores from persisting *early*, ahead of preceding accesses in memory (visibility) order. As such, the set of states a recovery program might observe includes many states that (1) do not correspond to SC program executions, and (2) could never arise in a fault-free execution, posing a daunting challenge for recovery design.

Reasoning about recovery can be greatly simplified by providing *failure atomicity* of

sets of PM updates. Failure atomicity assures that either all or none of the updates in a set are visible after failure, reducing the state space recovery code might observe. Atomicity (beyond the PM access granularity) can be achieved via logging [54, 201, 111, 47], shadow buffering [56], or checkpointing [178] mechanisms, which can be implemented in hardware [111, 178], as part of the programming/runtime system [47], or within the application [201, 54, 56].

ATLAS [47] argues to simplify recovery design by guaranteeing *failure-atomicity of outer-most critical sections*. Under such semantics, the language/runtime guarantees that recovery will observe a PM state as it existed when no locks were held by an application. However, we argue that this approach suffers from three key deficiencies: (1) its semantics are unclear for PM updates outside critical sections, (2) it does not generalize to other synchronization constructs (e.g., condition variables), (3) it requires expensive cycle detection among critical sections on different threads to identify sets that must be jointly failure-atomic, which leads to high overhead.

Instead, we propose persistency semantics that provide precise failure-atomicity at the granularity of *synchronization free regions* (SFRs)—thread regions delimited by synchronization operations or system calls. Prior works have used the SFR abstraction to define language memory models [149, 143] and to identify and debug data-races [143, 66, 37]. Under failure-atomic SFRs, the state observed by recovery will always conform to the program state at a *frontier* of past synchronization operations on each thread.

We argue that failure-atomic SFRs strike a compelling balance between programmability and performance. In a well-formed program, SFRs must be data-race free. This property allows us to extend the SC-for-DRF guarantee to recovery code and avoid the unclear semantics of ARP. Moreover, our approach avoids the limitations of ATLAS-like approaches.

We implement failure-atomic SFRs in a C++11 implementation (built on LLVM [129] v3.6.0). A programmer annotates variables that should be allocated in a persistent address

space. Our compiler pass and runtime system introduce undo-logging that enables recovery to PM state of a prior SFR frontier, from which application-specific recovery can then reconstruct volatile program state. We consider two designs that strike different trade-offs in simplicity vs. performance.

SFR-atomicity with coupled visibility: In this design, the persistent state lags the frontier of execution by at most a single (incomplete) SFR; recovery rolls back to the start of the SFR upon failure. This approach admits simple logging, but exposes the latency of PM flushing and commit.

SFR-atomicity with decoupled visibility: In this design, we allow execution to run ahead of the persistent state. We defer flushing and commit to background threads using a garbage-collection-like mechanism. In this design, we propose efficient mechanisms to ensure that the SFR commit order matches their execution order.

In summary, we make following contributions:

- We make a case for failure atomicity at SFR granularity and show how this approach provides precise PM semantics and is applicable to arbitrary synchronization primitives, such as C++11 atomics.
- We demonstrate how SFR-atomicity with coupled visibility simplifies logging, resulting in an average performance improvement of 63.2% over the state-of-the-art ATLAS design [47].
- We further observe that ordering of logs is sufficient for recoverability and propose SFR-atomicity with decoupled visibility. With this design, we show a further performance improvement of 65.5% over ATLAS.

4.2 Design Overview

We extend the C++ memory model with durability semantics for multi-threaded programs. We leverage synchronization operations to establish SFR boundaries and assure

failure atomicity at this granularity, as shown in Figure 2.2(c). An SFR is a region of code delimited by two synchronization operations (or system calls) [143, 168]. If a synchronization operation has store semantics and modifies a location in PM, it forms its own, single-instruction region ordered before a second SFR it delimits comprising subsequent writes until the next synchronization. C++ requires that SFRs be data-race free, and, in turn, guarantees serializability of SFRs, despite any compiler and hardware optimizations that reorder accesses within SFRs to gain performance [42, 15]. That is, programs are guaranteed to behave as if the updates made within SFRs become visible to all other threads atomically at the synchronization operation that terminates the SFR. Note that C++ provides no semantics for programs with unannotated data-races.

The key advantage of providing failure atomicity at SFR granularity is that it allows us to extend the appearance of SC-for-DRF behavior to recovery code as well as fault-free execution. In the absence of SFR atomicity, loads that observe PM state after failure in effect race with the PM updates that may or may not have completed within the SFR running at the point of failure. As such, C++ places no constraints on the state recovery may observe. Under failure-atomic SFRs, the state in PM at recovery follows the program state at a frontier of past synchronization operations on each thread.

C++ provides synchronization operations that assure SC-for-DRF. Specifically, we study the inter-thread and intra-thread *happens-before* ordering prescribed by synchronization operations in multi-threaded applications to order memory accesses. We extend these guarantees to ensure that the memory accesses within SFRs become persistent in an order consistent with the constraints on when they may become visible. We formalize these requirements later in Section 4.4.2.

Further, we propose compile and runtime mechanisms to provide failure atomicity at SFR granularity. We implement a compiler pass in LLVM v3.6.0, which instruments synchronization operations and PM accesses with undo-logging operations. In a traditional undo logging scheme, the state of the memory locations to be updated is first recorded in

undo logs. Once the logs persist, in-place mutations of data structures may be made. Once the mutations are complete, state is committed by invalidating and discarding corresponding log entries. We investigate two logging designs that vary in simplicity and performance.

SFR-atomicity with coupled visibility: In this design, the visibility of the program state in volatile caches is coupled with its persistent state in PM. The in-place PM mutations are flushed at the end of each SFR and the undo log is immediately committed. Thus, the committed state lags the frontier of execution by at most a single (currently executing) SFR; recovery rolls back to the start of the SFR, minimizing the state lost on a failure. This approach admits a simple logging design where there is only a single uncommitted SFR per thread and logs are entirely thread-local. However, it exposes flush and commit latency on the critical execution path.

SFR-atomicity with decoupled visibility: Instead, we can allow execution to run ahead of the persistent state by deferring flush and commit. In this approach, the persistent state still comprises a frontier of SFRs on each thread, but may arbitrarily lag execution. We use a garbage-collection-like mechanism to periodically flush PM state and commit logs. This approach can hide the latency of flushing and commit with execution of additional SFRs. The key challenge is that the SFR commit order must match their execution order. We describe efficient mechanisms to ensure correct commit.

4.3 SFR Failure Atomicity

We next describe the logging mechanism we propose to provide failure-atomicity for SFRs.

4.3.1 Logging

In both variants of our system, we use undo logging to provide failure atomicity of SFRs. For the synchronization operation that begins an SFR, and every PM store operation within the SFR, our compiler pass emits code to construct an undo log entry in PM. Fig-

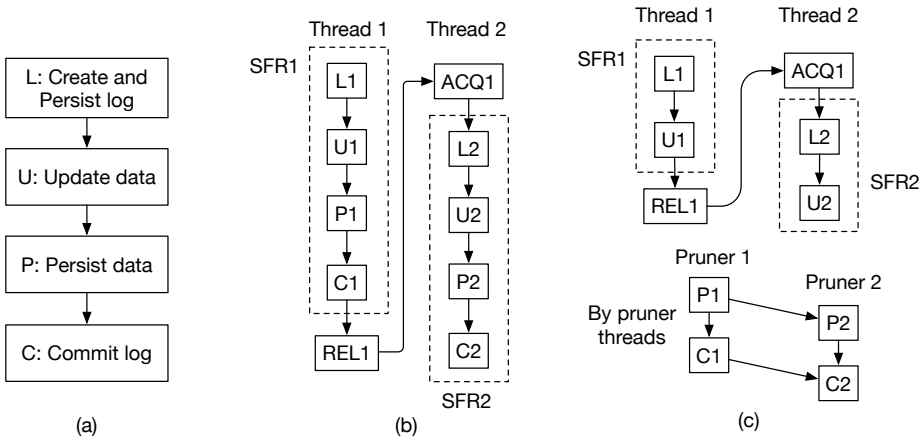


Figure 4.1: **Logging mechanisms in Coupled-SFR and Decoupled-SFR implementations.** (a) Steps in undo logging mechanism, (b) Undo log ordering in Coupled-SFR when SFRs are durability-ordered, and (c) Undo log ordering in Decoupled-SFR when SFRs are durability-ordered.

ure 4.1 illustrates the high-level steps our scheme must perform. Undo logs are appended to thread-local log buffers in PM. The log entry records the values PM locations had at the start of the SFR, before any mutation. The log entry is then persisted by explicitly flushing it from volatile caches to the PM (step L). Next, our compiler pass emits an ISA-level memory ordering barrier (to order the flush with subsequent writes) and the store operation that updates the persistent data structure in place (step U). This update may remain buffered in volatile write-back caches or it may drain to PM due to cache replacement, unless we explicitly flush it. Once updates have been explicitly flushed and persisted (step P), the corresponding undo log entries may be committed (step C). The commit operation marks logs to be pruned, discarded and reused. Our two atomicity schemes differ in when and how they perform these latter two steps.

As shown in Figure 4.1(a), the partial updates within an SFR are recoverable only when the steps outlined above are performed in order [126]. For instance, undo logs must be created and persist before in-place mutations may be made. Otherwise, it is possible that the mutations are written-back from caches to PM before the undo log persists. If failure occurs in the interim, the state as of the start of the SFR cannot be recovered. Similarly, undo logs may be committed only after the in-place mutations persist. We ensure proper ordering

between the operations by using mechanisms of an underlying ISA-level persistency model. In the case of Intel x86, this requires `CLWB` (or `CLFLUSHOPT` or `CLFLUSH` in older processors) to flush writes and `SFENCE` to order with respect to subsequent operations.

In case of a failure, recovery code begins by inspecting the uncommitted undo logs. It processes these logs, rolling back updates that may have drained from uncommitted SFRs. After rollback, the PM state will correspond to the state that existed at the start of some *frontier* of SFRs on each thread. Subsequent recovery operations (e.g., to prepare volatile data structures) are assured they will not observe updates from any partially executed SFR.

Log structure: We adopt an undo log organization similar to ATLAS [47]. Each thread manages a thread-local header, located in a pre-specified location in PM, which points to a linked list of undo log entries. As the undo logs are thread-private, threads may concurrently append to their logs. The order of entries in each undo log reflects program order. Log entries include the following fields:

- **Log type**: Entry type, one of `STORE`, `ACQUIRE`, or `RELEASE`
- **Addr**: Address of the access
- **Value**: Value to which to recover for `STORE` operations, or the log count (see Section 4.3.3) for `ACQUIRE`, or `RELEASE`
- **Size**: Access size
- **Next**: Link to next log entry

4.3.2 SFR-atomicity with Coupled Visibility

Our first design, SFR failure-atomicity with coupled visibility (Coupled-SFR), couples execution (more precisely, visibility of PM reads and writes) and persist of PM updates—persists may lag execution only until the start of the next SFR. Execution and persistency advance nearly in lock-step.

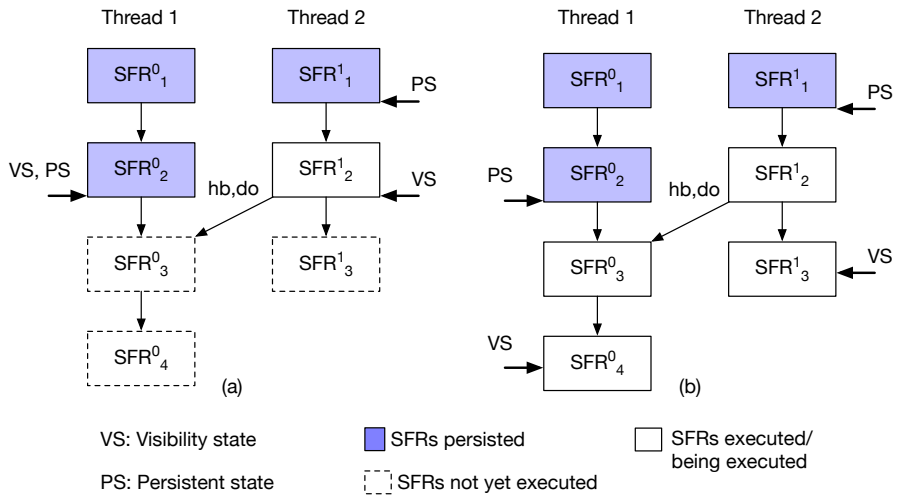


Figure 4.2: **Example persistent and execution states.** Persistent and execution state of SFRs in (a) Coupled-SFR, and (b) Decoupled-SFR.

Logging: Under Coupled-SFR, updates within an SFR are flushed and persist at the end of the SFR. Our compiler pass emits code to create undo logs, mutate data in place, flush mutations, and commit logs as described in Section 4.3.1. We emit log creation code for each PM store as shown in Figure 4.3(a). Before the SFR's terminal synchronization, an `SFENCE` instruction is emitted to ensure that all PM mutations persist before any writes in the next SFR.

Failure and Recovery: Each thread maintains only undo logs for its incomplete SFR. Upon failure, recovery code rolls back updates from the partially completed SFR on each thread using the logs. Subsequent recovery code observes the PM state as it was at the last synchronization operation prior to failure on each thread.

Discussion: The central advantage of Coupled-SFR is that each thread must track only log entries for stores within its still-incomplete SFR, and does not interact with any other thread. The thread-private nature of our commit stands in stark contrast to ATLAS, which must perform a dependency analysis and cycle-detection across all threads' logs to identify log entries that must commit atomically. Because accesses within SFRs must be data-race free, there can be no dependences between accesses in uncommitted SFRs; all inter-thread dependencies must be ordered by the synchronization commencing the SFR, and hence

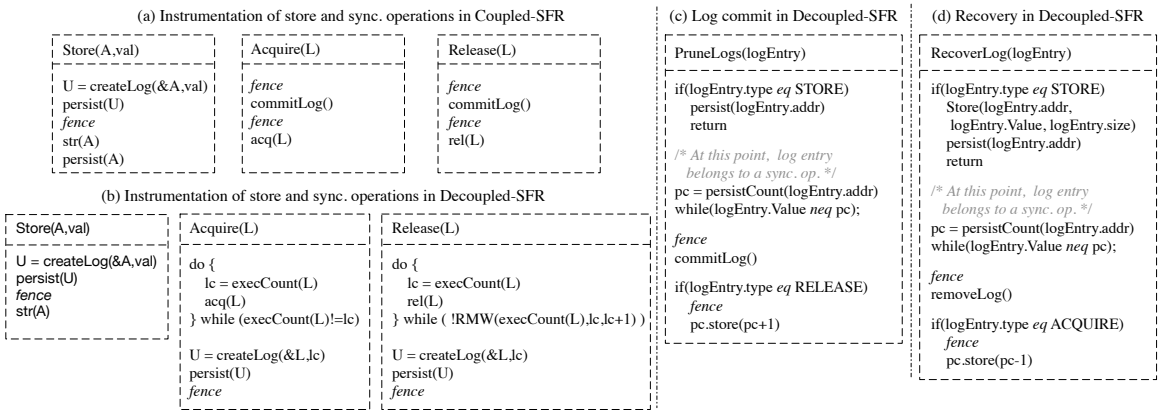


Figure 4.3: **Logging code instrumentation in Coupled-SFR and Decoupled-SFR designs.** (a) Instrumentation for store and synchronization operations in Coupled-SFR design. (b) Instrumentation for store and synchronization operations in Decoupled-SFR design. Acquire (acq) and release (rel) operations are atomic loads and atomic stores to the synchronization variable L. (c) Pseudo-code for log commit operation by pruner threads in Decoupled-SFR design. (d) Pseudo-code for recovery operation at failure in Decoupled-SFR design.

may depend only on committed state. The PM state after recovery is easy to interpret, as it conforms to the state at the latest synchronization on each thread.

However, the downside of Coupled-SFR is that there is relatively little scope to overlap the draining of persistent writes with volatile execution—execution stalls at the end of the SFR until all PM writes are flushed and the log is committed, potentially exposing much of PM persist latency on the critical path. Figure 4.2(a) illustrates an example of how high persist latencies can delay execution. In Figure 4.2(a), the program state on Thread 2 is stalled while the updates in SFR_2^1 remain pending to persist. These stalls further delay execution on Thread 1, as SFR_3^0 is ordered after SFR_2^1 by synchronization operations.

4.3.3 SFR-atomicity with Decoupled Visibility

The key drawback of Coupled-SFR is that it exposes the high latency of persists and log commits on the execution critical path. Instead, we decouple the visibility of updates (as governed by cache coherence and the C++ memory model) from the frontier of persistent state; that is, we can allow persistent state to lag execution—an approach we call Decoupled-SFR. Nevertheless, Decoupled-SFR must still assure that recovery will roll PM

state back to some prior state that conforms to a frontier of synchronization operations on each thread. To ensure that persistent state does not fall too far behind (which risks losing forward progress in the event of failure), we periodically invoke a flush-and-commit mechanism, much like garbage collection in managed languages. This mechanism flushes in-place updates and commits logs. However, the key invariant this mechanism must maintain is that SFRs commit in an order consistent with their execution. We next describe how we ensure this property.

Logging: Program state is recoverable if undo logs persist in the order the SFRs are executed (more precisely, the partial order in which they became visible, according to the C++11 memory model). In case of failure, undo logs are processed in reverse order to recover program state to the start of committed SFRs. The key departure of Decoupled-SFR from Coupled-SFR is that we defer flush and commit to perform them in the background, off the critical execution path.

In Figure 4.1(c), we illustrate logging under Decoupled-SFR. Like Coupled-SFR, our compiler pass emits logging code in advance of in-place PM mutations. In addition, we emit log entries for all synchronization operations. Read synchronization operations create `ACQUIRE` log entries, while write and read-modify-write emit `RELEASE` entries. If a `RELEASE` is to a location in PM, we emit first a `STORE` and then a `RELEASE` log for it. Log entries are appended to thread-local logs in creation (program) order. Pseudo-code for the instrumentation of store, acquire and release operations are shown in Figure 4.3(b). Unlike Coupled-SFR, we do not emit flush or commit code as part of the SFR. Instead, we delegate these operations to *pruner threads*, which operate periodically on the logs. We next explain how we maintain correct commit order for SFRs.

Ordering commit: Each program thread has an accompanying *pruner thread* that flushes mutations and commits the log on its behalf. Like garbage-collection, pruner threads are invoked periodically to commit and recycle log space.

Recoverability requires that logs are pruned—committing the updates in the corre-

sponding SFR—in the same order as the SFRs are executed, else the state after recovery will not correspond to a state consistent with fault-free execution. As such, our logging mechanism must log the *happens-before* ordering relations between SFRs (as governed by the C++11 memory model) and commit according to this order. We record happens-before by: (1) adding acquire / release annotations to the per-thread logs, (2) maintaining per-thread logs in program order (thereby capturing intra-thread ordering), and (3) tracking order across threads by maintaining a monotonic sequence number across release / acquire pairs. We refer throughout to Figure 4.3(b), which illustrates pseudo-code for our instrumentation.

We associate a sequence number `execCount(L)` with each synchronization variable `L`. We use a lock-free hashmap to record `execCount(L)` for each synchronization operation, allowing lock-free concurrent access/update of the counters. The hashmap is located in volatile memory for faster accesses, because we do not need the hashmap for recovery and can reinitialize it post-failure. For simplicity, our implementation assumes `execCount(L)` is large enough that we can ignore wrap-around.

For operations with release semantics (see Figure 4.3(b) Release), the instrumentation code observes the current value of `execCount(L)`. Then, `execCount(L)` is incremented with an atomic memory access. The loop in this pseudo-code accounts for the possibility of racing `RELEASE` operations. A log entry is emitted reflecting the identity of the synchronization variable `L` and the observed value of `execCount(L)`, which is recorded in the log entry's `Value` field.

A subsequent `ACQUIRE` operation that synchronizes-with a `RELEASE` observes the sequence number of that release (see Figure 4.3(b) Acquire). Note that it is critical that the acquire operation and the observation of the sequence number are atomic, which we arrange by reading the `execCount(L)` field twice, before and after the acquire—a mismatch indicates two racing release operations (unlikely in well-structured code), which we handle by synchronizing again.

Log commit: The pruner threads must together commit logs in sequence number order. We use a second monotonic counter per synchronization variable, the *persist counter* (`persistCount(L)`), also placed in a lock-free hashmap, to synchronize and order SFR commit across pruner threads.

The pseudo-code for log commit and pruning is depicted in Figure 4.3(c). Each pruner thread processes its thread-private log starting at the entry indicated in its corresponding log header. Upon reaching an entry for a synchronization operation, the pruner thread may need to wait for other pruner threads to ensure commit is properly ordered. We consider each kind of log entry in turn:

`STORE`: The pruner thread ensures the corresponding mutation is persistent by flushing the corresponding address with a `CLWB` operation (using the `Addr` field recorded in the log).

`ACQUIRE`: The pruner thread spins on `persistCount(L)` until it equals `execCount(L)` recorded in the `Value` field of the log entry. This spin awaits commit of the SFR with which the acquire synchronized. The SFR is then committed.

`RELEASE`: The pruner thread spins on `persistCount(L)` until it equals `execCount(L)` recorded in the `Value` field of the log entry. This spin waits for commit of the preceding release of the same synchronization variable. Then, a *fence* is issued to ensure the `CLWB` operations of any preceding `STORE` log entries are ordered before commit. The SFR may then be committed. After commit, `persistCount(L)` is incremented, which unblocks the pruner thread that will commit the next SFR for this synchronization variable. Note, again, the need for a memory fence after commit to ensure that the commit operation is ordered before subsequent commits and the increment of `persistCount(L)`.

Log pruning: Pruner threads prune (discard) log entries when an SFR is committed. To prune a group of entries belonging to an SFR, the pruner atomically modifies the pointer in its log header to point to a later log entry. The log space may then be freed/recycled. As the log entries belonging to an SFR are committed atomically, only after the updates within the SFR have persisted, the pruner threads guarantee SFR failure-atomicity.

Failure and Recovery: In Decoupled-SFR, the state after failure and recovery must conform to a frontier of past synchronization operations on each thread. Recovery code inspects the uncommitted undo logs and rolls back updates in the reverse order of log creation. Much like the commit operation of pruner threads, the recovery code uses the `execCount(L)` sequence number recorded in the `Value` field of log entries to apply undo logs in reverse order. The pseudo-code for this recovery is shown in Figure 4.3(d). First, the recovery process scans all undo logs and records the highest observed sequence number for each synchronization variable in a hashmap. Then, `STORE` log entries are replayed in reverse creation order to roll back values in PM. As the logs roll back, replayed log entries are pruned when traversing `ACQUIRE` or `RELEASE` entries, thus allowing recovery even in the event of multiple/nested failures. Once PM state is recovered, application-specific recovery code takes over to reconstruct any necessary volatile state.

Optimizations: We enable certain optimizations to make log pruning more efficient. First, we can often commit batches of SFRs atomically. If `persistCount(L)` matches the `Value` in all synchronization log entries for consecutive SFRs (i.e., no need to wait), we commit them together. Second, a pruner thread processes `STORE` log entries for a single SFR together: it issues multiple `CLWB` operations to flush updates in parallel. Importantly, processing entries as a group allows us to coalesce multiple updates to the same address within an SFR. Note that we still log all writes to the same memory addresses within the SFR separately, which avoids the need to check if the memory address has previously been logged within the SFR on the critical execution path.

Finally, if a pruner thread commits its last log entry, it blocks to conserve CPU. Execution threads wake all pruners when log entries accumulate above some threshold. Note that, since pruner threads may have to wait for one another to process dependent log entries, they should be gang-scheduled.

Discussion: Under Decoupled-SFR, persistent state may arbitrarily lag execution state. Hence, although recovery arrives at a state consistent with a synchronization frontier, for-

ward progress may be lost. Programmers must be aware of this possibility. If state loss is not desired (e.g., if the program will perform an operation with an irrecoverable side-effect), Decoupled-SFR provides a *psync* operation, which stalls execution and triggers pruner threads to drain their logs.

4.4 Durability Invariants

We briefly discuss invariants that a logging implementation must meet to ensure failure-atomicity of SFRs and describe how the Coupled-SFR and Decoupled-SFR implementations ensure these invariants.

4.4.1 Preliminaries

We introduce a notation to describe persist ordering, following the approach in prior works [123, 124], and present a summary of persist ordering as it relates to the C++ memory model. C++ provides atomic (`std::atomic<>`) primitives, which allow programmers explicit control over the ordering of memory accesses. Atomic variables may be loaded and stored directly (without, e.g., a separate mutex) and hence facilitate the implementation of a wide variety of synchronization primitives. We formalize persist ordering using the following notation for memory operations to a location l from a thread i.

- ACQ_l^i: an atomic load or read-modify-write
- REL_l^i: an atomic store or read-modify-write
- M_x^i: a non-atomic operation on memory location `x`

We indicate ordering constraints among memory events with the following notation:

- $M_x^i \leq_{sb} M_y^i$: M_x^i is sequenced-before M_y^i in thread i
- $REL_l^i \leq_{sw} ACQ_l^j$: A release operation on location l in thread i "synchronizes with" an acquire operation on location l in thread j.

- $M_x^i \leq_{hb} M_y^j$: M_x^i in thread i happens-before M_y^j in thread j

The C++ memory model achieves inter-thread ordering using the "synchronizes-with" ordering relation and intra-thread ordering using the "sequenced-before" ordering relation. The "happens-before" relation is the transitive closure of "synchronizes-with" $\leq_{sw}$ and "sequenced-before" $\leq_{sb}$ orderings.

Memory operations must follow the *sequenced-before* ordering relations within a thread. A release operation REL_l^i orders prior memory access M_x^i and an acquire operation ACQ_l^i orders subsequent memory access M_y^i on thread `i`. Further, the C++ memory model achieves the inter-thread ordering using the "synchronizes-with" order relation between an acquire and release operation. A release operation REL_l^i in thread i synchronizes-with the acquire operation ACQ_l^j in thread j. The synchronizes-with relation orders memory access M_x^i in thread i with memory access M_y^j in thread j:

$$(M_x^i \leq_{sb} REL_l^i \leq_{sw} ACQ_l^j \leq_{sb} M_y^j) \rightarrow M_x^i \leq_{hb} M_y^j \tag{4.1}$$

We now use the happens-before ordering relation between the memory accesses to define the order in which SFRs must be made durable in PM.

4.4.2 SFR Durability

Atomic loads, stores, and read-modify-write operations delimit SFRs. We say that a store operation is *visible post-recovery* if the effects of the store may be observed by code that runs after failure and recovery. Our logging designs must ensure that an SFR is failure-atomic:

Atomicity Invariant*: If there exists a PM update within an SFR that is visible post-recovery, then all updates in the SFR must be visible post recovery.*

The *Atomicity Invariant* guarantees that the updates within an SFR are not partially visible after failure. We say that an SFR is *durable* if all its updates are visible post-

recovery.

Further, our logging must ensure that SFRs become durable in an order consistent with the C++11 memory model. We use the happens-before ordering relation between the memory accesses to prescribe the order SFRs must be made durable.

Suppose SFR^i and SFR^j denote SFRs on threads i and j respectively. Consider memory operations M_x^i and M_y^j on threads i and j respectively, such that $M_x^i \in SFR^i$, and $M_y^j \in SFR^j$. We say that SFR^i is *durability-ordered* before SFR^j if:

$$\exists \left(M_x^i \in SFR^i, M_y^j \in SFR^j\right) | M_x^i \leq_{hb} M_y^j, SFR^i \leq_{do} SFR^j \tag{4.2}$$

where $SFR^i \leq_{do} SFR^j \rightarrow SFR^i$ must be made durable before SFR^j.

Finally, we require that durability-order between SFRs is transitive and irreflexive:

$$(SFR^i \leq_{do} SFR^j) \wedge (SFR^j \leq_{do} SFR^k) \rightarrow SFR^i \leq_{do} SFR^k \tag{4.3}$$

Following Equation 4.2, logging must satisfy:

Durability Invariant*: If an SFR is durable, SFRs that are durability-ordered before it must also be durable.*

Note that the SFRs are unordered if there exists no transitive durability-ordering relation between them. The key correctness requirement of the recovery mechanism is that the state that the recovery code observes after failure must be consistent with the ordering constraints expressed in Equation 4.2-4.3. We now describe how our designs, Coupled-SFR and Decoupled-SFR, satisfy the atomicity invariant to guarantee SFR failure-atomicity and the durability invariant to ensure SFR durability is properly ordered.

4.4.3 Coupled-SFR

Under Coupled-SFR, each thread maintains a thread-local pointer to a list of log entries for at most one incomplete SFR. The log is committed atomically using `commitLog` as

shown in Figure 4.3(a) before the synchronization operation that ends the SFR is executed. `CommitLog` atomically prunes the entire list of undo log entries by zeroing the pointer in the thread-local header. The SFR is durable when the logs commit. This atomic commit satisfies the *Atomicity Invariant*, thereby ensuring failure-atomicity of SFRs.

Figure 4.1(b) illustrates the SFRs, `SFR1` and `SFR2`, as ordered by the happens-before ordering relation. Note that execution of the memory accesses in `SFR1` are ordered before those in `SFR2` by the happens-before ordering relation between `REL1` on thread 1 and `ACQ1` on thread 2. The ordering relation between `REL1` and `ACQ1`, implies `SFR1` is durability-ordered before `SFR2` by Equation 4.2. As shown in Figure 4.1(b), `SFR1` becomes durable in the commit stage (step `C1` in Figure 4.1(b)) before the release operation. Further, the subsequent acquire operation is sequenced-before the commit operation (step `C2` in Figure 4.1b) in `SFR2`. The two ordering relations guarantee that `SFR1` becomes durable before `SFR2` in Coupled-SFR.

4.4.4 Decoupled-SFR

Similar to Coupled-SFR, under Decoupled-SFR, each thread maintains a thread-local pointer to the head of its undo logs. The pruner threads commit logs atomically by adjusting the log header to point to a subsequent log entry for a synchronization operation, as shown in Figure 4.3(c). The atomic commit ensures that one (or more) SFRs are made durable atomically.

Figure 4.1(c) shows the order of creation of undo logs for `SFR1`, which is *durability-ordered* with `SFR2`. The durability-order relation implies that `SFR1` must be made durable before `SFR2`. During execution, Decoupled-SFR assigns ascending sequence numbers to the synchronization operations. The log entry corresponding to the release operation records a sequence number from `execCount(L)`, atomically increments it and then performs the release operation. Consequently, the acquire operation that synchronizes-with the release operation records the updated sequence number in its log entry followed by

executing SFR2. As shown in Figure 4.3(c), the pruner threads commit the log entry in ascending sequence number order. Thus, the logs for SFR1, which are sequenced-before the release operation, commit before SFR2, which are sequenced-after the acquire operation. The two ordering relations guarantee the durability of SFR1 before SFR2.

4.5 Evaluation

We implement a compiler pass that can emit code for both our logging approaches in LLVM [129] v3.6.0. The compiler pass instruments stores and synchronization operations to create undo logs according to the pseudo-code in Figure 4.3. We also provide a library containing the recovery code that rolls back undo logs upon failure, recovering to a frontier of past synchronization operation, and the runtime code for log pruning in Decoupled-SFR. We first describe our experimental framework including our system configuration, the benchmark suite that we use, and the designs we consider in our experiments.

System configuration: We perform our experiments on an Intel E5-2683 v3 server class machine with 14 physical cores, each with 2-way hyper-threading, operating at a frequency of 2.00GHz. Since byte-addressable persistent memory devices are not yet commercially available, we use Linux `tmpfs` [191], memory-mapped in DRAM, to mimic the persistent address space of a PM-enabled system. Note that it is widely expected that the access latency of actual PM devices will be higher than that of DRAM (likely by 2-10x) [209]. In our experimental setup, we expect to underestimate the cost of flushing mutations to PM in ATLAS and Coupled-SFR. In Decoupled-SFR, because we delegate flush operations for in-place updates to the pruner threads, we expect to hide the flush latency. Hence, we expect to obtain similar performance for Decoupled-SFR even with slower PM devices. As such, we believe our evaluation is conservative in estimating the performance advantage of Decoupled-SFR over the alternatives.

Our Haswell-class server machine does not offer `clwb` instructions, instead providing a `clflush` operation to flush the data out of the cache hierarchy to the memory controller.

Benchmark	Description
Concurrent queue (CQ)	Insert/Delete nodes in a queue
Array Swap (SPS)	Random swap of array elements
Persistent Cache (PC)	Update entries in persistent hash table
RB-tree	Insert/Delete nodes in RB-Tree
TATP	Update location trans. from TATP [164]
Linked-List (LL)	Update/Insert/Delete nodes in a linked-list
TPCC	New Order trans. from TPCC [197]

Table 4.1: **Benchmarks.** Set of multi-threaded micro-benchmarks and benchmarks used to study Coupled-SFR and Decoupled-SFR designs.

Systems supporting `clwb`, which avoids some undesirable overheads of `clflush`, are expected to be available in the near future. To our knowledge, no available x86 platform provides mechanisms to ensure that data are indeed flushed to memory. Instead, Intel presently requires the memory controller in PM-enabled systems to guarantee durability (e.g., via battery backup or flush-upon-failure) [100]. As a result, we rely on `sfence` operation to order the drain of updates.

Benchmarks: We study a suite of seven write-intensive multi-threaded benchmarks and micro-benchmarks, listed in Table 4.1, which have been used in prior studies of persistent memory systems [171, 123, 54, 111]. The Concurrent Queue (CQ), similar to that of prior works [171, 123], inserts and removes nodes from a shared persistent queue. The Array Swap, RB-tree and Persistent cache (PC) are similar to the implementations in NV-Heaps [54]. Our TATP benchmark executes the update location transaction of the TATP database workload [164], which models the home location registration database of a telecommunications provider. Our TPCC benchmark executes the new-order transaction from the TPCC database workload [197], which models an order processing system. The Linked-List benchmark uses a hand-over-hand locking mechanism to update, insert, and remove nodes in a persistent linked-list. All the benchmark run 12 concurrent execution threads and perform 10M operations on the persistent data structure.

Design options: We compare the following designs: (a) ATLAS: a state-of-the-art logging approach that provides failure-atomicity of outermost critical sections, (b) Coupled-SFR: our mechanism for SFR failure-atomicity with coupled visibility, (c) Decoupled-SFR:

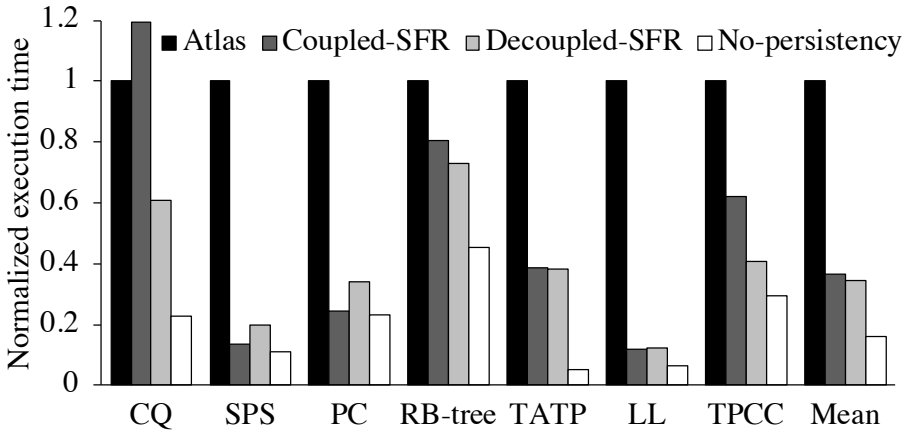


Figure 4.4: **Execution time.** Execution time of Coupled-SFR and Decoupled-SFR designs normalized to ATLAS. No-persistency design, with no durability guarantees, shows an upper bound on performance.

our mechanism for SFR failure-atomicity with decoupled visibility, and (d) No-persistency: a design that provides no recoverability of the program. We include No-persistency to show an upper bound for our performance improvements and quantify the cost of recoverability. No-persistency provides no recovery guarantees.

4.5.1 Performance Comparison

Figure 4.4 contrasts the execution time of Coupled-SFR and Decoupled-SFR with that of ATLAS. In this experiment, we perform two operations per SFR for the concurrent queue (CQ), persistent cache (PC), array swap (SPS), RB-tree, and linked-list (LL). The other two benchmarks TATP, and TPCC implement open specifications and so each SFR includes as many write operations as are required to implement the mandated behavior of update location and new order transactions, respectively. ATLAS performs the slowest in all benchmarks (except in CQ) because it records the order of execution of critical sections (as opposed to Coupled-SFR), and flushes the PM mutations within each critical section on the critical execution path (as opposed to Decoupled-SFR). Decoupled-SFR enables light-weight recording of SFR order and performs flush and commit operations on pruner threads, off the critical execution path. As a result, Decoupled-SFR achieves up to

80.1% and 66.0% performance improvement in array swap and persistent cache, respectively, which employ fine-grained locking and have the highest concurrency. Linked-list uses hand-over-hand locking and must acquire several locks in the linked-list before operating on a node. Decoupled-SFR performs best with 87.5% improvement in Linked-list, as it greatly simplifies logging as compared to the ATLAS.

It is interesting to note that Coupled-SFR performs better than Decoupled-SFR in array swap, persistent cache, and linked-list. This might seem counter-intuitive, as Coupled-SFR admits simpler logging at the cost of committing logs at every synchronization operation. However, these benchmarks perform only two stores per SFR. As a result, the cache flush operations on the critical path under Coupled-SFR incur less overhead than the more complex logging code of Decoupled-SFR.

As the number of stores per critical section grows, ATLAS fails to scale. ATLAS does not support concurrent commit and must rely on only a single helper thread to commit and recover log entries. Therefore, as the number of PM writes scales with the number of execution threads, the single helper thread can no longer keep up with the required commit rate and the log grows until available log capacity is exhausted. On the contrary, both Coupled-SFR and Decoupled-SFR perform distributed pruning and do not suffer from this issue.

CQ has no concurrency as all the threads contend to acquire a single lock to access the queue. Coupled-SFR performs worse than ATLAS in CQ as the flush and commit operations are done in the critical execution path by each thread, incurring delay. We show a separate comparison between Coupled-SFR and Decoupled-SFR with a varying number of PM writes per SFR in Section 4.5.4.

4.5.2 Logging Overhead

We study the overhead of each of the various steps performed in logging for our Coupled-SFR and Decoupled-SFR designs. In Figure 4.5, we incrementally enable steps in undo

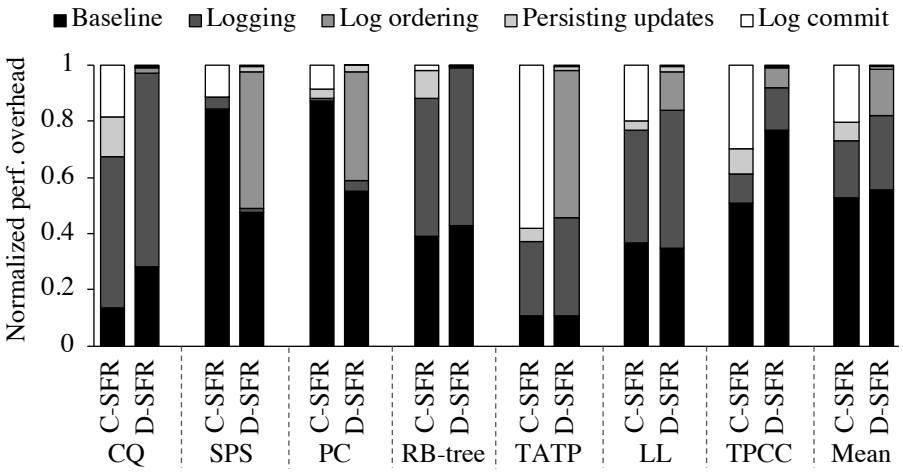


Figure 4.5: **Logging overhead.** Distribution of logging overhead in Coupled-SFR and Decoupled-SFR designs.

logging and study the distribution of execution time in each step. Note that none of these incomplete designs implement a recoverable system; we study them only to quantify overheads.

In Coupled-SFR, the majority of time is spent in creating the logs entries and flushing them to PM. Note that there is no overhead in Coupled-SFR due to log ordering as the log entries are committed at the end of each SFR. Overall, Coupled-SFR spends 39% of the execution time in flush and log commit when there are two operations per SFR.

In contrast, Decoupled-SFR spends less than 1% of execution time flushing updates and committing logs as these operations are performed by pruner threads in the background. The remaining 1% overhead is due to the pruning of the final few logs when the benchmarks complete. Our result indicates that the pruner threads are able to keep up with program execution. We also measure the log size overhead in the Decoupled-SFR design. Across our experiments, the log size in Decoupled-SFR is typically less than a few KB and never grows above 100 KB. On average, log creation costs 26.6% and recording of log order costs 16.3% of the total execution time in Decoupled-SFR.

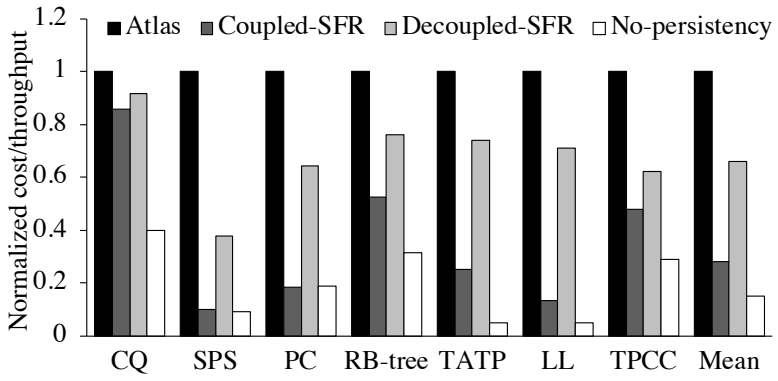


Figure 4.6: **CPU cost per throughput.** CPU cost per throughput of Coupled-SFR and Decoupled-SFR normalized to ATLAS. The No-persistency design shows cost/throughput for a non-recoverable implementation.

4.5.3 CPU Cost per Throughput

We next evaluate the cost of the background activity required by both ATLAS and Decoupled-SFR to commit their logs. Although the pruner/helper threads do not delay execution on the critical path, they nonetheless consume CPU resources and therefore can increase the total CPU cost to complete the benchmarks. We measure this overhead by dividing the total CPU utilization (in CPU-seconds) consumed by all threads over the course of benchmark execution by the achieved throughput (operations/transactions per second). For this metric, lower is better (less CPU overhead per unit of forward progress). Figure 4.6 shows the normalized CPU-cost per throughput of each benchmark for all four designs. We find that the cost of Coupled-SFR is the lowest as compared to ATLAS and Decoupled-SFR, as the threads executing the program commit the logs themselves. As we create as many pruner threads as there are execution threads in the program, Decoupled-SFR requires higher CPU resources to flush and commit the logs. In concurrent queue, which (despite its name) has no concurrency, the cost per throughput of Decoupled-SFR is equivalent to ATLAS, because there exists a single total order across all logs on all threads, and so the actions of the pruner threads are serialized. As No-persistency does not create any logs, it has the lowest cost per throughput of all designs, and illustrates the cost of recoverability. Overall, Coupled-SFR and Decoupled-SFR have 72.1% lower and 33.2%

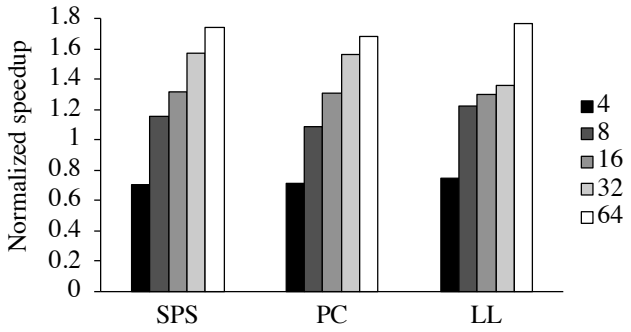


Figure 4.7: **Performance study with different SFR sizes.** Sensitivity study showing speedup of Decoupled-SFR normalized to Coupled-SFR with increasing number of stores per SFR.

lower CPU-cost per throughput than ATLAS.

4.5.4 Sensitivity Study of Operations/SFR

The size of logs varies with the number of store operations performed in SFRs. We perform a sensitivity analysis to study how the performance of our designs compare as the number of stores per SFR increases. Figure 4.7 illustrates the performance of Decoupled-SFR for the four benchmarks, normalized to Coupled-SFR. With two stores per SFR, we see that Coupled-SFR performs better than Decoupled-SFR. Decoupled SFR is slower because the overhead of creating and updating the `execCount(L)` and `persistCount(L)` to maintain undo log order in Decoupled-SFR is higher than the performance gain of delegating flush operations for only two stores to pruner threads. As the number of store operations increase, the flush operations and log commits delay execution in Coupled-SFR. As a result, at 64 stores per SFR, Decoupled-SFR performs 1.74x faster than Coupled-SFR. For benchmarks such as CQ, RB-tree and TPCC, we have already shown in Figure 4.4 that Decoupled-SFR performs 1.98×, 1.53× and 1.10× better than Coupled-SFR, respectively.

CHAPTER V

Relaxed Persist Ordering Using Strand Persistency

5.1 Introduction

Persistent memory (PM) technologies, such as Intel and Micron's 3D XPoint, are here — cloud vendors have already started public offerings with support for Intel's Optane DC persistent memory [7, 13, 2, 20]. PMs combine the byte-addressability of DRAM and durability of storage devices. Unlike traditional block-based storage devices, such as hard disks and SSDs, PMs can be accessed using a byte-addressable load-store interface, avoiding the expensive software layers required to access storage, and allowing for fine-grained PM manipulation.

Because PMs are durable, they retain data across failures, such as power interruptions and program crashes. Upon failure, the volatile program state in hardware caches, registers, and DRAM is lost. In contrast, PM retains its contents—a *recovery* process can inspect these contents, reconstruct required volatile state, and resume program execution [50, 83, 140, 157].

Several *persistency models* have been proposed in the past to enable writing recoverable software, both in hardware [98, 22] and programming languages [47, 78, 123, 124, 77]. Like prior works [171, 122, 112], we refer to the act of completing a store operation to PM as a *persist*. Persistency models enable two key properties. First, they allow programmers to reason about the order in which persists are made [171, 112, 122, 160]. Similar to mem-

ory consistency models [15, 42, 73, 127, 146], which order visibility of shared memory writes, memory persistency models govern the order of persists to PM. Second, they enable failure atomicity for a set of persists. In case of failure, either all or none of the updates within a failure-atomic region are visible to recovery [126, 54, 201, 56].

Recent works [47, 78, 8, 123, 201, 54, 123, 121, 207] extend the memory models of high-level languages, such as C++ and Java, with persistency semantics. These language-level persistency models differ in the synchronization primitives that they employ to provide varying granularity of failure atomicity. These persistency models are still evolving and are fiercely debated in the community [54, 201, 8, 47, 78, 123, 105]. Specifically, ATLAS [47], Coupled-SFR [78, 121], and Decoupled-SFR [78, 121] employ general synchronization primitives in C++ to prescribe the ordering and failure atomicity of PM operations. Other works [8, 201, 54, 126] ensure failure atomicity at a granularity of transactions using software libraries [201, 54, 126] or high-level language extensions [8].

These language-level models rely on low-level hardware ISA [98, 22] primitives to order PM operations. For instance, Intel x86 systems employ `CLWB` instruction to explicitly flush dirty cache lines to the point of persistence and `SFENCE` instruction to order subsequent `CLWB`s and stores with prior `CLWB`s and stores [98]. Under Intel's persistency model, `SFENCE` enforces a bi-directional ordering constraint on subsequent persists and introduces high-latency stalls until prior `CLWB`s and stores complete. In this paper, we note that `SFENCE` introduces stricter ordering constraints than required by high-level programming languages and that the persist order can be decoupled from the visibility of PM operations while still guaranteeing correct failure recovery.

Prior research proposals relax ordering constraints by proposing relaxed persistency models [171, 125, 112, 160] in hardware and/or build hardware logging mechanisms [111, 62, 166, 113] to ensure failure-atomic updates to PM. These works propose relaxed persistency models, such as epoch persistency [112, 56, 160], that implement *persist barriers* to divide regions of code into *epochs*; they allow persist reordering within epochs and disallow

persist reordering across epochs. Unfortunately, epoch persistency labels only consecutive persists that lie within the same epoch as concurrent. It fails to relax ordering constraints on persists that may be concurrent, but do not lie in the same epoch. In contrast, hardware logging mechanisms [111, 166, 62, 187] aim to provide efficient implementations for ensuring failure atomicity for PM updates in hardware. These works ensure failure atomicity for transactions by emitting logging code for PM updates transparent to the program. These mechanisms impose fine-grained ordering constraints (*e.g.* between log and PM updates) on persists but propose fixed and inflexible hardware that fails to extend to a wide range of evolving language-level persistency models.

In this work, we propose StrandWeaver, which formally defines and implements the *strand persistency* model to minimally constrain ordering on persists to PM. The principles of the strand persistency model were proposed in earlier work [171], but no hardware implementation, ISA primitives, or software use cases have yet been proposed. The strand persistency model defines the order in which persists may drain to the PM. It decouples persist order from the write visibility order (defined by the memory consistency model)—memory operations can be made visible in shared memory without stalling for prior persists to drain to PM. To implement strand persistency, we introduce three new hardware ISA primitives to manage persist order. A *NewStrand* primitive initiates a new *strand*, a partially ordered sequence of PM operations within a logical thread—operations on separate strands are unordered and may persist concurrently to PM. A *persist barrier* orders persists within a strand—persists separated by a persist barrier within a strand are ordered. Persist barriers do not order persists that lie on separate strands. A *JoinStrand* primitive ensures that persists issued on the previous strands complete before any subsequent persists can be issued.

StrandWeaver proposes hardware mechanisms to build the strand persistency model upon these primitives. StrandWeaver implements a *strand buffer unit* alongside the L1 cache that manages the order in which updates drain to PM. The strand buffer unit enables

updates on different strands to persist concurrently to PM, while persists separated by persist barriers within a strand drain in order. Additionally, StrandWeaver implements a *persist queue* alongside the load-store queue to track ongoing strand persistency primitives. The persist queue guarantees persists separated by `JoinStrand` complete in order even when they lie on separate strands.

StrandWeaver decouples volatile and persist memory order and provides the opportunity to relax persist ordering even when the system implements a conservative consistency model (*e.g.* TSO [170]). Unfortunately, programmers must reason about persist order at the abstraction of the ISA, making it burdensome and error-prone to program persistent data structures. To this end, we integrate the ISA primitives introduced by StrandWeaver into high-level language persistency models to enable programmer-friendly persistency semantics. We build a logging design that employs StrandWeaver's primitives to enforce only the minimal ordering constraints on persists required for correct recovery. We showcase the wide applicability of StrandWeaver primitives by integrating our logging with three prior language-level persistency models that provide failure-atomic transactions [8, 201, 54], synchronization-free regions [78], and outermost critical sections [47], respectively. These persistency models provide simpler primitives to program recoverable data structures in PM—programmer-transparent logging mechanisms layered on top of our StrandWeaver hardware hide low-level hardware ISA primitives and reduce the programmability burden.

In summary, we make the following contributions:

- We formally define primitives for strand persistency that enable relaxed persist order, decoupled from visibility of PM operations.
- We propose StrandWeaver, hardware mechanisms to implement the primitives defined by the strand persistency model. Specifically, we show how the strand buffer unit and persist queue can order and schedule persists concurrently.
- We build logging designs that rely on low-level hardware primitives proposed by

StrandWeaver and integrate them with several language-level persistency models.

- We evaluate StrandWeaver to show that it relaxes persist order to outperform Intel's persistency model by up to 1.97× (1.45× avg.).

5.2 Strand Persistency Model

Strand persistency divides thread execution into *strands*. Strands constitute sets of PM operations that lie on the same logical thread. Ordering primitives enforce persist ordering within strands, but persists are not individually ordered across strands. We use the term "strand" to evoke the idea that a strand is a part of a logical thread, but has independent persist ordering. Strand persistency decouples the visibility and persist order of PM operations. The consistency model continues to order visibility of PM operations—PM operations on separate strands are visible in an order enforced by system's consistency model.

Strand primitives. Strand persistency employs three primitives to prescribe persist ordering: a *persist barrier* to enforce persist ordering among operations on a strand, *NewStrand* to initiate a new strand, and a *JoinStrand* to merge prior strands initiated on the logical thread. PM accesses on a thread separated by a persist barrier are ordered. Conversely, `NewStrand` removes ordering constraints on subsequent PM operations. `NewStrand` initiates a new strand—a strand behaves as a separate logical thread in a persist order. Persists on different strands can be issued concurrently to PM. Note that persist barriers, within a strand, continue to order persists on that strand. The hardware must guarantee that recovery software never observes a mis-ordering of two PM writes on the same strand that are separated by a persist barrier. Finally, `JoinStrand` merges strands that were initiated on the logical thread. It ensures the persists issued on the prior strands complete before any subsequent persists are issued.

In this work, we propose StrandWeaver to define ISA extensions and build the strand persistency model in hardware. Further, we provide techniques to map the persistency

semantics offered by high-level languages to strand persistency.

5.2.1 Definitions

The strand persistency model specifies the order in which updates persist to PM. We formally define the persist order enforced under strand persistency using notation similar to prior works [122, 126].

- M_x^i: A load or store operation to PM location x on thread i
- S_x^i: A store operation to PM location x on thread i
- PB^i: A persist barrier issued by thread i
- NS^i: A `NewStrand` issued by thread i
- JS^i: A `JoinStrand` issued by thread i

Persist memory order (PMO) is an ordering relation that describes the ordering of memory operations to PM defined by the system's persistency model.

- $M_x^i \leq_{po} M_y^i$: M_x^i is program ordered before M_y^i
- $M_x^i \leq_p M_y^i$: M_x^i is ordered before M_y^i in PMO

We now define the ordering constraints that are expressed by the primitives under strand persistency.

Intra-strand ordering. `NewStrand` operation initiates a new strand and clears all the ordering constraints in PMO on subsequent memory operations. A persist barrier orders PM operations within a strand. Thus, two memory operations that are not separated by a `NewStrand` are ordered in PMO by a persist barrier.

$$\begin{aligned} &\left(M_x^i \leq_{po} PB^i \leq_{po} M_y^i\right) \wedge \left(\nexists NS^i : M_x^i \leq_{po} NS^i \leq_{po} M_y^i\right) \\ &\rightarrow M_x^i \leq_p M_y^i \end{aligned} \tag{5.1}$$

Additionally, `JoinStrand` introduces ordering constraints on PM operations to different memory locations that lie on separate strands. Note that a persist barrier does not order persists on different strands. JoinStrand orders persists initiated on prior strands with the persists on subsequent strands.

$$M_x^i \leq_{po} JS^i \leq_{po} M_y^i \rightarrow M_x^i \leq_p M_y^i \tag{5.2}$$

Thus, memory operations separated by `JoinStrand` are ordered in PMO.

Strong persist atomicity. Persists to the same or overlapping memory locations follow the order in which memory operations are visible (as governed by the consistency model of the system)—this property is called *strong persist atomicity* [171]. Similar to consistency models that ensure *store atomicity* by serializing memory operations to the same memory location through coherence mechanisms, strong persist atomicity serializes persists to the same memory location. We preserve strong persist atomicity to ensure that recovery does not observe side-effects due to reorderings that would not occur under fault-free execution of the program.

$$S_x^i \leq_{po} S_x^j \rightarrow S_x^i \leq_p S_x^j \tag{5.3}$$

Conflicting persists that lie on different strands or logical threads are ordered through strong persist atomicity.

Transitivity. Finally, persist order is transitive and irreflexive:

$$\left(M_x^i \leq_p M_y^j\right) \wedge \left(M_y^j \leq_p M_z^k\right) \rightarrow M_x^i \leq_p M_z^k \tag{5.4}$$

Persists on racing strands or threads (two or more strands or threads that consist of racing memory accesses) can occur in any order, unless ordered by Equations 5.2-5.4.

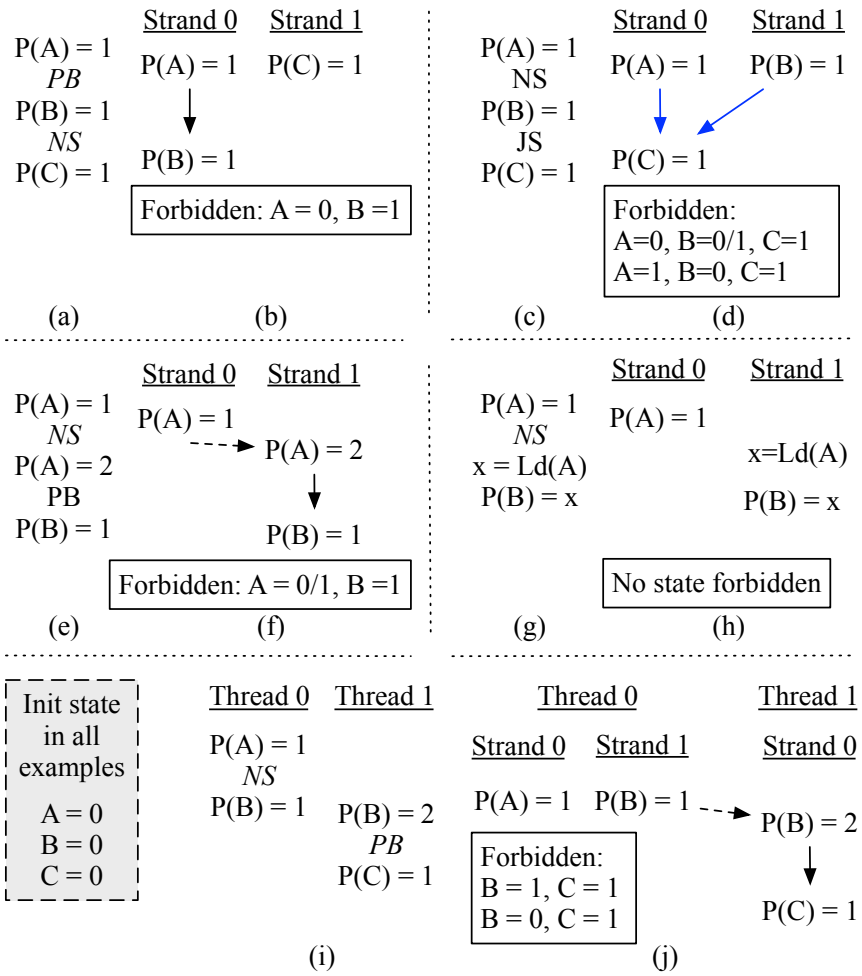


Figure 5.1: **Persist order due to StrandWeaver's primitives.** Figure uses following notations for strand primitives: PB: persist barrier, NS: `NewStrand` and JS: `JoinStrand`. In each case, we also show the forbidden PM state. Black solid arrow, blue solid arrow, and black dotted arrow show order due to persist barrier, `JoinStrand`, and SPA, respectively. (a,b) Intra-strand ordering due to persist barrier, (c,d) Inter-strand ordering due to `JoinStrand`, (e,f) Persist order due to SPA, (g,h) Loads to the same PM location do not order persists, (i,j) Inter-thread ordering due to SPA.

5.2.2 Persist Ordering

Figure 5.1 illustrates persist ordering under different scenarios due to strand persistency primitives.

Intra-strand persist concurrency. Figure 5.1(a) shows example code that employs `NewStrand` to issue persists concurrently on different strands, and a persist barrier to order persists within a strand. Persist barrier `PB` orders persist `A` before persist `B` (Equation 5.1) on strand `0` as shown in Figure 5.1(b). The `NewStrand` operation clears ordering constraints

on following persists due to the previous persist barrier PB (Equation 5.1) and initiates a new strand 1. Persist barrier PB does not order persists that lie on different strands. Persist C lies on strand 1, and can be issued to PM concurrent to persists A and B.

Inter-strand persist ordering. Figure 5.1(c) shows example code that orders persists using JoinStrand. JoinStrand merges strands 0 and 1 to ensure that persists A and B are ordered in PMO before persist C (as per Equation 5.2) as shown in Figure 5.1(c,d). Figure 5.1(d) shows forbidden PM state that requires persist C to reorder before persists A and B and so, would never occur under strand persistency.

Inter-strand strong persist atomicity. Strong persist atomicity (SPA) governs the order of persists on different strands or threads to the same or overlapping memory locations (as per Equation 5.3). SPA orders persists as per their visibility enforced due to program order or cache coherence. Figure 5.1(e) shows an example of conflicting persists that occur on separate strands within a thread. Persist A on strand 0 is ordered before persist A on strand 1 as corresponding stores to the memory location A follow their program order [26]. Note that, persist B on strand 1 is ordered after persist A on strand 0 due to transitivity (as per Equation 5.4)—this relationship guarantees that recovery never observes the PM state shown in Figure 5.1(f).

Note that, a conflicting load to PM on another strand does not establish persist order in PMO (as per Equations 5.1 and 5.3). As shown in Figure 5.1(g), although load A is program-ordered after persist to A, persist B on strand 1 can be issued concurrently to PM. Although visibility of the memory operations is ordered, persists can be issued concurrently on the two strands—PM state (A=0, B=1) is not forbidden. Persist order due to SPA on separate strands can be established by having write-semantics for both memory operations to the same location (*e.g.* read-modify-write instead of loads). Alternatively, persist order across strands can be achieved using JoinStrand as shown in Figure 5.1(c,d).

Inter-thread strong persist atomicity. Similar to inter-strand order, SPA orders persists that occur on different logical threads. Figure 5.1(i) shows an example execution on

two threads. On thread `0`, persists `A` and `B` lie on different strands and are concurrent, as shown in Figure 5.1(j). If a store to memory location `B` on thread `0` is ordered before that on thread `1`, order that is established through cache coherence, they are ordered in PMO. SPA orders persist `B` (and following persist `C` due to intervening persist barrier) on thread `1` after persist `B` on thread `0`.

Establishing inter-thread persist order. Persists on different strands or threads may occur in any order, unless ordered by Equations 5.2-5.4. Synchronization operations establish *happens-before* ordering relation between threads [128, 31], ordering visibility of memory operations, but do not enforce persist order. Persists can potentially reorder across the synchronizing lock and unlock operations. This reordering can be suppressed by placing a `JoinStrand` operation before unlock and after synchronizing lock operations. Synchronizing lock and unlock operations establish a happens-before ordering relation between threads, and `JoinStrand` operations prevent any persists from reordering across synchronizing operations. Note that locks may be persistent or volatile. If locks reside in PM, persists resulting from lock and unlock operations are ordered in PM due to SPA. Thus, recovery may observe correct lock state and reset it after failure [4].

5.3 Hardware Implementation

We now describe hardware mechanisms that guarantee these persist orderings.

Microarchitecure. We implement StrandWeaver's persist barrier, `NewStrand`, and `JoinStrand` primitives as ISA extensions. A persist occurs due to a voluntary data flush from volatile caches to PM using a `CLWB` operation, or a writeback resulting from cacheline replacement. We use `CLWB`, which is *issued* to write-back caches by the CPU, to flush dirty cache lines to the PM controller. Note that `CLWB` is a non-invalidating operation—it retains a clean copy of data in caches. A `CLWB` *completes* when the CPU receives an acknowledgement of its receipt from the PM controller.

Architecture overview. Figure 5.2 shows the high-level architecture of StrandWeaver.

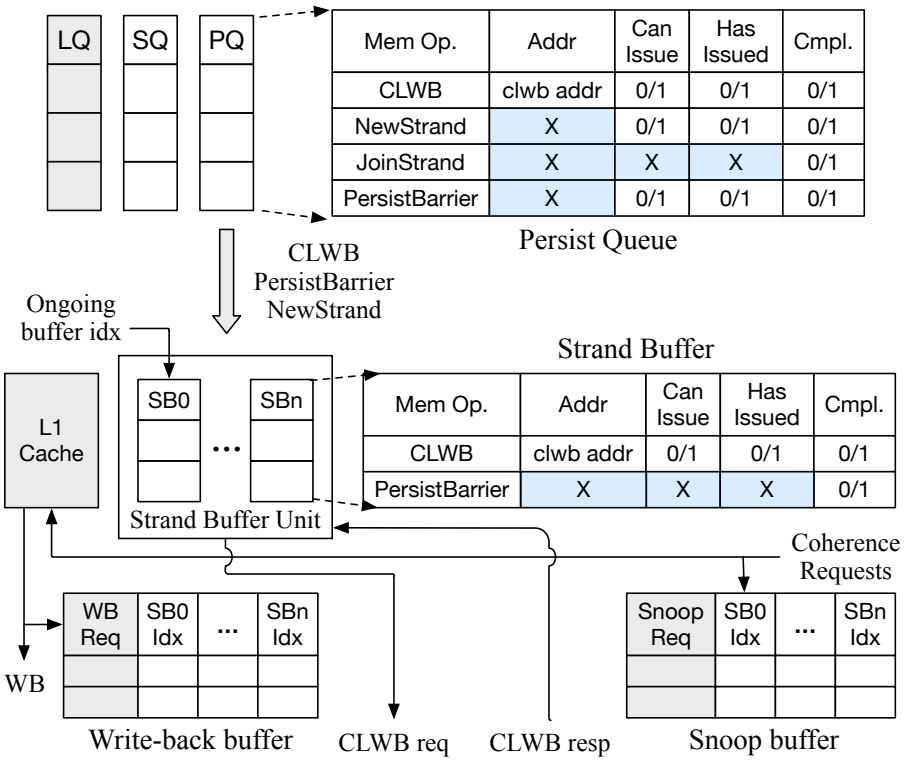


Figure 5.2: **StrandWeaver architecture.** Persist queue and strand buffer unit implement persist ordering due to primitives in strand persistency model.

The Persist queue and strand buffer unit jointly enforce persist ordering. The persist queue, implemented alongside the load-store queue (LSQ), ensures that `CLWB`s and stores separated by a persist barrier within a strand are *issued* to the L1 cache in order, and `CLWB`s separated by `JoinStrand` *complete* in order. The strand buffer unit is primarily responsible for leveraging inter-strand persist concurrency to schedule `CLWB`s to PM. It resides adjacent to the L1 cache and comprises an array of strand buffers that may issue `CLWB`s from different strands concurrently. Each strand buffer manages persist order within a strand and guarantees that persists separated by persist barriers within that strand complete in order. The strand buffer unit also coordinates with the L1 cache to ensure that persists due to cache writebacks are ordered as per PMO. It also tracks cache coherence messages to ensure that inter-thread persist dependencies are preserved.

Persist queue architecture. Figure 5.2 shows the persist queue architecture and operations appended to it by the CPU pipeline. The persist queue manages entries that record

ongoing CLWBs, persist barriers, NewStrand, and JoinStrand operations. Its architecture resembles that of a store queue—it supports associative lookup by address to identify dependencies between ongoing stores and CLWBs. Figure 5.2 also shows Addr, CanIssue, HasIssued, and Completed fields per entry in the persist queue. The Addr field records the memory address for an incoming CLWB operation that needs to be flushed from caches to the PM. The CanIssue field is set when an operation's persist dependencies resolve and the operation is ready to be issued to the strand buffer unit. CLWBs, persist barriers, and NewStrand are issued to the strand buffer unit when CanIssue is set; HasIssued is set as they are issued. The Completed field is set when the persist queue receives a completion acknowledgement for the operation. An operation can retire from the queue when Completed is set.

Persist queue operation. The persist queue tracks persist barriers to monitor intra-strand persist dependencies. On insertion, a persist barrier imposes a dependency so that CLWBs and stores are ordered within its strand. It orders issue of prior stores before subsequent CLWBs, and prior CLWBs before subsequent stores. These constraints ensure that stores do not violate persist order by updating the cache and draining to PM via a cache writeback before preceding CLWBs. The persist queue also coordinates with the store queue to ensure that younger CLWBs are issued to the strand buffer unit only after elder store operations to the same memory location. On CLWB insertion, the persist queue performs a lookup in the store queue to identify elder stores to the same location. This lookup is similar to that performed by the load queue for load-to-store forwarding [41, 206].

CLWBs, persist barriers, and NewStrand operations in the persist queue are issued to the strand buffer unit in order. Note that, unlike Intel's persistency model, which stalls stores separated by SFENCE until prior CLWBs *complete* (as described in Section 2.5), persist barriers stall subsequent stores only until prior CLWBs have *issued*. JoinStrand ensures that CLWBs and stores issued on prior strands complete before any subsequent CLWBs and stores can be issued. Unlike a persist barrier, JoinStrand stalls issue of subsequent CLWBs and

stores until prior CLWBs and stores *complete*. On JoinStrand insertion, the persist queue coordinates with the store queue to ensure that subsequent stores are not issued until prior CLWBs complete. As JoinStrand is not issued to the strand buffer unit, its CanIssue and HasIssued fields are not used.

CLWBs, persist barriers, and NewStrand operations complete when the persist queue receives a completion acknowledgement from the strand buffer unit. JoinStrand completes when prior CLWBs, persist barrier, and NewStrand are complete and removed from the persist queue, and prior stores are complete and removed from the store queue.

Strand buffer unit architecture. The strand buffer unit coordinates with the L1 cache to guarantee CLWBs and cache writebacks drain to PM and complete in the order specified by PMO. It maintains an array of strand buffers—each strand buffer manages persist ordering within one strand. CLWBs that lie in different strand buffers can be issued concurrently to PM. Strand buffers manage ongoing CLWBs and persist barriers and record their state in fields similar to the persist queue. The CanIssue and HasIssued fields mark when a CLWB is ready to issue and has issued to PM, respectively. The strand buffer retires entries in order when operations complete.

Strand buffer unit operation. The strand buffer unit receives CLWB, persist barriers, and NewStrand operations from the persist queue. In the strand buffer unit, the ongoing buffer index points to the strand buffer to which an incoming CLWB or persist barrier is appended. This index is updated when the strand buffer unit receives a NewStrand operation indicating the beginning of a new strand. Subsequent CLWBs and persist barriers are then assigned to the next strand buffer. StrandWeaver assigns strand buffers upon NewStrand operations in a round-robin fashion. The strand buffer unit acknowledges completion of NewStrand operations to the persist queue when it updates the current buffer index.

Each strand buffer manages intra-strand persist order arising from persist barriers. It orders completion of prior CLWBs before any subsequent CLWBs can be issued to PM. On insertion in a strand buffer, a persist barrier creates a dependency that orders any subse-

quent CLWBs appended to the buffer. A persist barrier completes when CLWBs ahead of it complete and retire from the strand buffer. On completion of a persist barrier, the strand buffer resolves dependencies for subsequent CLWBs and marks them ready to issue by setting CanIssue. When CLWBs are inserted, the strand buffer performs a lookup to identify any persist dependencies from incomplete persist barriers. If there are none, the strand buffer immediately sets CanIssue.

When its dependencies resolve (when CanIssue field is set), the strand buffer issues a CLWB—it performs an L1 cache lookup to determine if the cache line is dirty. If so, it flushes the dirty cache block to PM and retains a clean copy in the cache. Upon a miss, it issues the CLWB to lower-level caches. When the CLWB is performed, HasIssued is set.

The strand buffer receives an acknowledgement when a CLWB completes its flush operation. It marks the corresponding entry Completed and retires completed entries in order.

Managing cache writebacks. PM writes can also happen due to cache line writebacks from write-back caches. The persist queue does not stall visibility of stores following persist barriers until prior CLWBs complete—it only ensures that prior CLWBs are issued to the strand buffer unit before any subsequent stores are issued. Thus, stores might inadvertently drain from the cache before ongoing CLWBs in the strand buffer unit complete. StrandWeaver extends the write-back buffer, which manages in-progress writebacks from the L1 cache, with a field per strand buffer (as shown in Figure 5.2) that records the tail index of the buffer when the L1 cache initiates a writeback. The write-back buffer drains writebacks only after the strand buffers drain operations beyond these recorded indexes. This constraint guarantees that older CLWBs complete before subsequent writebacks are issued, and thus prevents any persist order violation. Note that, since CLWBs never stall in strand buffers to wait for writebacks, there is no possibility of circular dependency and deadlock in StrandWeaver.

Enabling inter-thread persist order. As explained earlier in Section 5.2, strong persist atomicity establishes order on persists to the same memory location across different

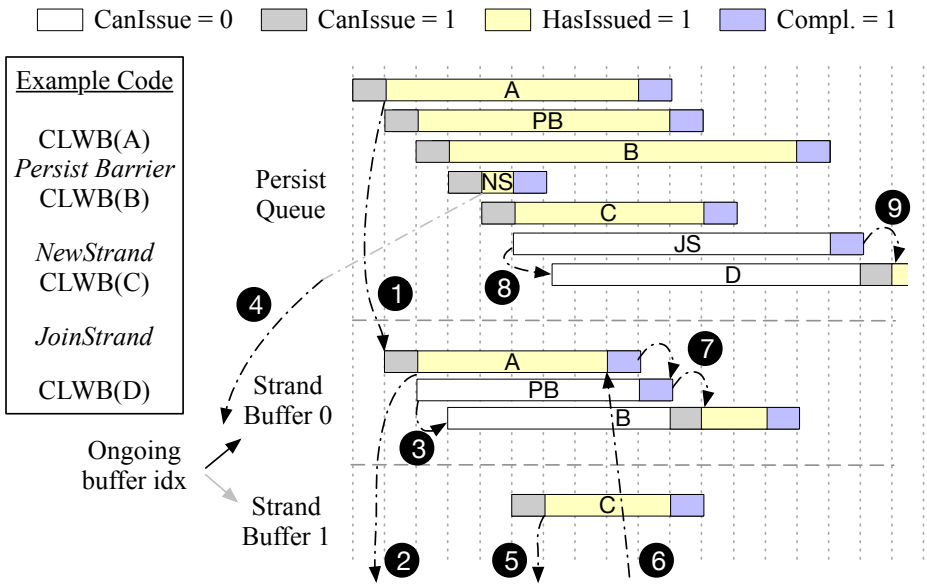


Figure 5.3: **Running example.** Figure uses following notations. PB: persist barrier, NS: `NewStrand`, JS: `JoinStrand`.

threads—persists follow the order in which stores become visible. As cache coherence determines the order in which stores become visible, we track incoming coherence requests to the L1 cache to establish persist order. If a cache line is dirty in the L1 cache, other cores might steal ownership and persist the cache line before ongoing `CLWB`s in the strand buffer complete (violating the required order shown in Figure 5.1(i,j)). Similar to the write-back buffer, we provision per-strand-buffer fields in the snoop buffers that track and respond to ongoing coherence requests. On an incoming read-exclusive coherence request, if the corresponding cache line is dirty, we record the tail index of the strand buffer in the snoop buffer. The read-exclusive request stalls until the strand buffers drain to the recorded index. This stall ensures that `CLWB`s that are in progress when the coherence request was received complete before the read-exclusive reply is sent. Again, there is no possibility of circular dependency/deadlock.

PM controller. We do not modify the PM controller; we assume it supports ADR [100, 99] and so lies in the persistent domain. When the PM controller receives a `CLWB`, it returns an acknowledgement to the strand buffer unit.

5.3.1 Example

Figure 5.3 shows an example code with the desired order on persists prescribed by PMO. We show a step-by-step illustration of operations as executed by StrandWeaver. ❶ `CLWB(A)` is appended to an entry in the persist queue, and is issued to the strand buffer unit, as it encounters no earlier persist dependencies. Since the current buffer index is `0`, `CLWB(A)` is added to strand buffer `0`. ❷ `CLWB(A)` is issued and performs an L1 access to flush the dirty cache line. ❸ A persist barrier and `CLWB(B)` are appended to strand buffer `0`; `CLWB(B)` stalls and waits for the preceding persist barrier (and `CLWB(A)`) to complete. ❹ `NewStrand` from the persist queue updates the ongoing buffer index in the strand buffer unit to `1`. Consequently, subsequent `CLWB(C)` is appended to strand buffer `1`. ❺ As `CLWB(C)` incurs no prior dependencies in its strand buffer `1` due to persist barriers, it issues to PM concurrent to `CLWB(A)`. ❻ The strand buffer unit receives a completion for `CLWB(A)`; the operation is complete. ❼ As `CLWB(A)` and the persist barrier complete, the ordering dependency of `CLWB(B)` is resolved, and it issues. ❽ JoinStrand stalls issue of `CLWB(D)` until prior `CLWB`s complete. ❾ When the persist queue receives a completion acknowledgement for `CLWB(A)`, `CLWB(B)`, and `CLWB(C)`, `JoinStrand` completes and `CLWB(D)` is issued to the strand buffer unit.

5.4 Designing Language-level Persistency Models

The strand persistency model decouples persist order from the visibility order of memory operations—it provides opportunity to relax persist ordering even in the presence of conservative consistency models (*e.g.* TSO [170]). Unfortunately, programmers must reason about memory ordering at the ISA abstraction, making it error-prone and burdensome to write recoverable PM programs. Recent efforts [47, 78, 8, 123, 201, 54, 123, 121, 207] extend persistency semantics and provide ISA-agnostic programming frameworks in high-level languages, such as C++ and Java. These proposals use existing synchronization prim-

itives in high-level languages to also prescribe order on persists and enforce failure atomicity for groups of persists. Failure atomicity reduces the state space visible to recovery and greatly simplifies persistent programming by ensuring either all or none of the updates within a region are visible in case of failure.

Some models [8, 201, 54] enable failure-atomic transactions for transaction-based programs and rely on external synchronization [4, 54] to provide transaction isolation. ATLAS [47] and SFR-based [78, 121, 77] persistency models look beyond transaction-based programs to provide failure atomicity using languages' low-level synchronization primitives. ATLAS employs undo logging to provide failure atomicity for outermost critical sections—code region bounded by lock and unlock synchronization operations. In contrast, SFR-based persistency models [78, 121, 77] enable failure-atomic synchronization-free regions—code regions bounded by low-level synchronization primitives, such as acquire and release. The models enable undo logging as a part of language semantics—compiler implementations emit logging for persistent stores in the program, transparent to the programmer. We propose logging based on strand persistency primitives. We integrate our logging mechanisms into compiler passes to implement language-level persistency model semantics.

Logging implementation. Undo logging ensures failure atomicity by recording the old value of data before it is updated in a failure-atomic region. Undo logs are committed when updates persist in PM. On failure, a recovery process uses uncommitted undo logs to roll back partial PM updates. For correct recovery, undo logs need to persist before in-place updates (as shown earlier in Figure 2.3(b)). A pairwise persist ordering between an undo log and corresponding in-place update ensures correct recovery. Within a failure-atomic region, undo logs for different updates need not be ordered—logging operations can persist concurrently (as shown under the ideal ordering constraints in Figure 2.3(d)). Similarly, in-place updates may persist concurrently too, provided they do not overlap.

Figure 5.4 shows our logging mechanism, which employs StrandWeaver's ISA prim-

```
FA-Begin()          FA-End()          Store(A,val)
log_begin()         JoinStrand        log_store(L_A,A)
JoinStrand          log_end()         CLWB(L_A)
                                      Persist Barrier
                                      store(A,val)
                                      NewStrand
```

Figure 5.4: **Logging using strand primitives.** Figure shows instrumentation for failure-atomic region begin and end, and PM store operation.

itives to enable failure-atomic updates. We persist undo logs and in-place updates using `CLWB` and order these persists using a persist barrier. The persist barrier ensures that the log is created and flushed to PM before the update. Each logging operation and update is performed on a separate strand; we issue `NewStrand` after each log-update sequence, enabling persist concurrency across the independent updates. We ensure that all persists within a failure-atomic region complete before exiting the region by enclosing it within `JoinStrand` operations—these ensure that persists on different strands do not "leak" out of the failure-atomic region. The precise implementation of `log_begin()` and `log_end()` vary based on the semantics prescribed by various language-level persistency models.

Integrating with language persistency models. Under ATLAS, we initiate and terminate failure-atomic regions at the lock and unlock operations of outermost critical sections. `log_begin()` creates a log entry for the lock operation. The log entry captures happens-after ordering relations on the lock due to prior unlock operations on the same lock, similar to the mechanism employed by ATLAS [47]. `log_end()` for an unlock operation updates metadata (similar to [47]) to record happens-before ordering information required by the subsequent lock operation on that lock. `log_store()` creates an undo log entry that records the address and prior value of an update. Under SFR-based persistency, we emit `log_begin()` and `log_end()` at the acquire and release synchronization operations bracketing each SFR. As in prior work [78], `log_begin()` and `log_end()` log happens-before ordering relations in their log entries to ensure correct recovery. Under failure-atomic transactions [8, 54], `log_end()` flushes all PM mutations in the transaction

and ensures that they persist before committing the logs.

Log structure. We initialize and manage a per-thread circular log buffer in PM as an array of 64-byte cache-line-aligned log entries. Additional log entries are allocated dynamically if the log space is exhausted.

Our log entry structure is similar to prior work [78, 47]:

- `Type`: Entry type [`Store`, `Acquire`, `Release`] in ATLAS or SFR, [`Store`, `TX_BEGIN`, `TX_END`] for transactions
- `Addr`: Address of the update
- `Value`: Old value of an update in a store log entry, or the metadata for happens-before relations for a sync. operation
- `Size`: Size of the access
- `Valid`: Valid bit for the entry
- `Commit marker`: Commit intent marker for log commit

Our logging implementation maintains *head* and *tail* pointers to record the bounds of potentially valid log entries in the log buffer. Figure 5.5(a) (step ❶) shows the head and tail pointers and the valid log entries that belong to synchronization operations (marked red) and store operations (marked blue). The tail pointer indicates the location to which the next log entry will be appended—we advance the tail pointer upon creation of each log entry. We maintain the tail pointer in volatile memory so that log entries created on different strands are not ordered by updates to the tail pointer (as a consequence of strong persist atomicity, see Equation 5.3).

The head pointer marks the beginning of potentially uncommitted log entries. In Figure 5.5(a), suppose log entry `4` marks the end of a failure-atomic region. Before commit begins, we set the commit marker of the log entry that terminates the failure-atomic region as shown in step ❷ in Figure 5.5(a)—this marks that the log commit has initiated. We mark

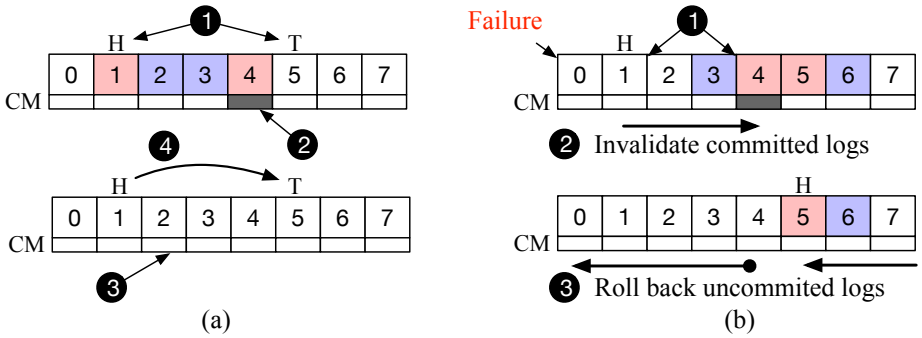


Figure 5.5: **Logging example.** Entries for store operations are shown in blue and entries for synchronization operations are shown in red. `CM` refers to the commit marker in the log entry. (a) Running example of log entry allocation and commit. (b) Running example of recovery process on failure.

undo-log entries corresponding to a failure-atomic region invalid (step ❸ in Figure 5.5(a)), and update and flush the head pointer to commit those log entries (step ❹ in Figure 5.5(a)).

On failure, the tail pointer in volatile memory is lost, and the persistent head pointer is used to initiate recovery. First, the recovery process identifies the log entries that were committed, but not invalidated prior to failure; this scenario occurs if failure happens during an ongoing commit operation. Figure 5.5(b) shows an example with the commit marker for log entry `4` set, log entries `1,2` invalidated, and log entries `3,4` yet to be invalidated (step ❶). The recovery process invalidates the log entries from the head pointer to the log entry `4` with the commit marker set, and advances the head pointer, as shown in Figure 5.5(b) (step ❷). Then, the recovery process scans the log buffer starting from the head pointer and rolls back values recorded in valid log entries in reverse order of their creation, as shown in Figure 5.5(b) (step ❸).

5.5 Evaluation

We next describe our evaluation of StrandWeaver.

Core	8-cores, 2GHz OoO 6-wide Dispatch, 8-wide Commit 224-entry ROB 72/64-entry Load/Store Queue
I-Cache	32kB, 2-way, 64B 1ns cycle hit latency, 2 MSHRs
D-Cache	32kB, 2-way, 64B 2ns hit latency, 6 MSHRs
L2-Cache	28MB, 16-way, 64B 16ns hit latency, 16 MSHRs
DRAM, PM controller	64/32-entry write/read queue, 1kB row buffer
PM	Modeled as per [107], 346ns read latency, 96ns write latency to controller 500ns write latency to PM

Table 5.1: **Simulator Specifications.** Table lists the configuration of StrandWeaver's implementation.

5.5.1 Methodology

We implement StrandWeaver in the gem5 simulator [35], configured as per Table 5.1. We model a PM device as per the recent characterization studies of Intel's Optane memory [107], as shown in Table 5.1. We configure our design with 16-entry persist queue and four 4-entry strand buffers. StrandWeaver requires a total of 144B of additional storage each in the persist queue and strand buffer unit per core. It also extends the write-back buffer and snoop buffer, 8 bits per entry each, to record a 2-bit tail index for four strand buffers. We consider other configurations for the persist queue and strand buffer unit in Section 5.5.3.

Benchmarks. Table 5.2 describes the microbenchmarks and benchmarks we study, and reports `CLWB`s issued per thousand CPU cycles (CKC) as a measure of their write-intensity. Queue performs insert and delete operations to a persistent queue. Hashmap performs updates to a persistent hash, array-swap swaps two elements in an array, RB-tree performs inserts and deletes to a persistent red-black tree, and TPCC performs new order transactions, which model an order processing system. Additionally, we study N-Store [24], a persistent key-value store benchmark, using workloads with different read-write ratios, as listed in Table 5.2. We use the YCSB engine with N-Store to generate load and modify its undo-log

Benchmarks	Description	CKC
Queue	Insert/delete to queue [171, 112]	0.78
Hashmap	Read/update to hashmap [54, 122]	4.83
Array Swap	Swap of array elements [54, 122]	4.45
RB-Tree	Insert/delete to RB-Tree [54, 112]	3.46
TPCC	New Order trans. from TPCC [197, 122]	1.58
N-Store (rd-heavy)	90% read/10% write KV workload [24]	4.41
N-Store (balanced)	50% read/50% write KV workload [24]	8.06
N-Store (wr-heavy)	10% read/90% write KV workload [24]	10.05

Table 5.2: **Benchmarks.** `CLWB`s per 1000 cycles (CKC) measures write intensity of the benchmarks in the non-atomic design.

engine to integrate our logging mechanisms. The microbenchmarks and benchmark each run eight threads and perform 50K operations on persistent data structures. As shown in Table 5.2, N-Store under a write-heavy workload is the most write-intensive benchmark and queue and TPCC are the least write-intensive microbenchmarks in our evaluation.

Language-level persistency models. As explained in Section 5.4, we design language-level implementations that map persistency semantics in high-level languages to the low-level ISA primitives defined by StrandWeaver. We implement failure-atomic transactions, outermost critical sections (ATLAS), and SFRs to evaluate StrandWeaver for each of the benchmarks.

We compare following designs in our evaluation:

BASELINE. This design implements language-level persistency models using Intel's existing ISA primitives, which divide program regions into epochs using `SFENCE`, and allow persist reordering only within the epochs. In this design, logs and in-place updates are ordered by `SFENCE`.

NO-PERSIST-QUEUE. This is an intermediate hardware design that implements the strand persistency model, but without the addition of a persist queue. Incoming `CLWB`s, persist barriers, `NewStrand` and `JoinStrand` are inserted in the existing store queue. The store queue manages the order in which `CLWB`s, `NewStrand`, and persist barriers issue to the strand buffer unit. We use this design to study the concurrency enabled by the strand buffer unit.

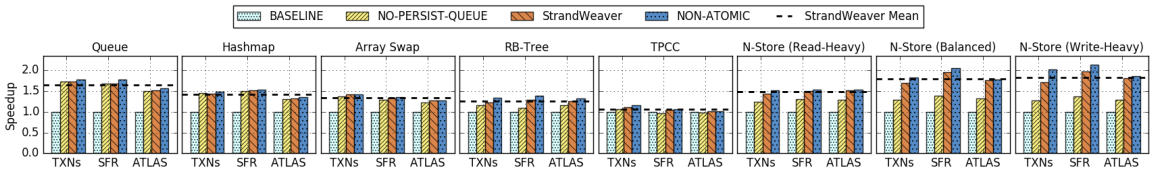


Figure 5.6: **StrandWeaver's speedup.** Speedup of StrandWeaver and Non-atomic design normalized to the baseline implementation using Intel's persistency model.

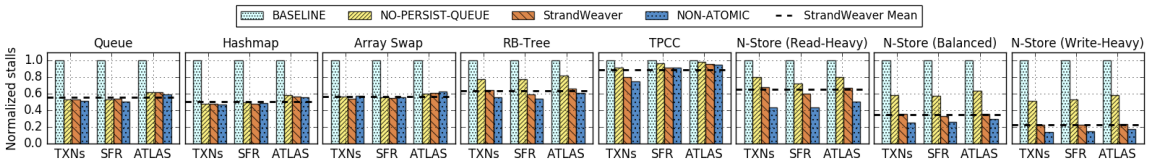


Figure 5.7: **Pipeline stalls.** CPU stalls as hardware enforces persist order. Stalls due to barriers create back pressure in CPU pipeline, and blocks program execution.

StrandWeaver. This design implements our proposal, as detailed in Sections 5.3-5.4.

NON-ATOMIC. In this design, we do not order log persists with in-place updates—we remove the `SFENCE` between the log entry creation and in-place update. Due to the absence of any ordering constraints, this design shows the best-case performance that StrandWeaver can obtain due to relaxed persist ordering. Note that, since logs are not ordered before in-place updates, this design does not assure correct recovery in case of failure.

5.5.2 Performance Comparison

Figure 5.6 shows the performance comparison for our microbenchmarks and benchmarks, implemented under the three language-level persistency models across the hardware designs. Figure 5.7 shows CPU pipeline stalls as hardware enforces persist ordering constraints—frequent stalls due to barriers fill hardware queues and block program execution.

StrandWeaver outperforms BASELINE. StrandWeaver outperforms the baseline design in all the benchmarks we study, as it relaxes persist order relative to Intel's existing ordering primitives. The baseline orders log operations and in-place updates using `SFENCE`, enforcing drastically stricter ordering constraints than required for correct recovery. As

explained in Section 2.5, SFENCE divides program execution into epochs and CLWBs are allowed to reorder/coalesce only within the epochs. Unfortunately, the persist concurrency available within epochs is limited by their small size [112]. In contrast, StrandWeaver enforces only pairwise ordering constraints between the undo log and in-place update. As a result, StrandWeaver outperforms the baseline design by 1.45× on average.

Note that we achieve speedup over the baseline even though the memory controller lies in the persistent domain and hides the write latency of the PM device. In the baseline, SFENCEs stalls issue for subsequent updates until prior CLWBs complete. The additional constraints due to SFENCE fill up the store queue, creating back pressure and stalling the CPU pipeline. StrandWeaver encounters 48.7% fewer pipeline stalls, resulting in a performance gain of 1.45× on average over the baseline. Table 5.2 shows that N-Store, under a write-heavy workload, is the most write-intensive benchmark that we evaluate. As a result, StrandWeaver achieves the highest speedup of 1.82× on average, with 77.1% fewer stalls, in N-Store.

Persist concurrency due to strands. The strand buffers issue CLWBs that lie on different strands concurrently. As shown in Figure 5.6, StrandWeaver's intermediate design—without the persist queue—achieves 1.29× speedup over the baseline on average, with 34.9% fewer pipeline stalls. Adding the persist queue prevents head-of-the-line blocking due to long-latency CLWBs in the store queue—stores on different strands may enter the store queue and issue concurrent to CLWBs. StrandWeaver attains an additional performance improvement of 1.13× over the variant without the persist queue.

Performance comparable to non-atomic design. Figure 5.6 shows performance for the non-atomic design that removes the pairwise ordering constraint between the updates and their logs. We include this design to study the limit on performance that StrandWeaver might achieve—this design does not ensure correct recovery as updates can persist before their logs. StrandWeaver incurs 3.1% slowdown in microbenchmarks and 5.7% slowdown in N-Store relative to this upper bound due to additional persist ordering within each strand.

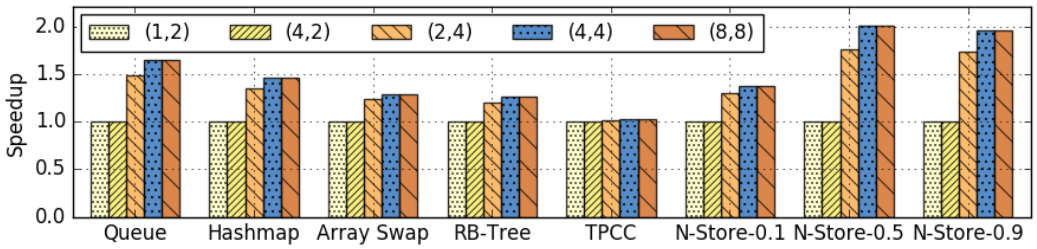


Figure 5.8: **Speedup due to different StrandWeaver's configurations.** Sensitivity study with different StrandWeaver configurations denoted as (Number of strand buffers, Number of entries per strand buffer).

Low write-intensity benchmarks. StrandWeaver achieves its lowest speedup of 5.32% on average in TPCC for the three persistency model implementations. TPCC acquires multiple locks per new order transaction to ensure isolation. As such, there is high lock acquisition overhead per failure-atomic region. As per Table 5.2, Queue has the lowest write intensity, but achieves a speedup of 1.64× on average. Queue has the least concurrency among the benchmarks we study, as all its threads contend on a single lock to serialize push and pop operations to a persistent queue. `CLWB`s fall on the critical execution path and additional ordering constraints incur execution delay.

Sensitivity to language-level persistency model. StrandWeaver's implementation that ensures failure-atomic transactions flushes in-place updates and commits logs at the end of the failure-atomic region. In contrast, the SFR implementation issues batched commits by logging happens-before relations in logs at the end of each SFR and continuing execution without stalling for log commits. ATLAS issues batched log commits too, but employs heavier-weight mechanisms to record happens-before order between the lock and unlock operation, as compared to SFR [78]. Thus, StrandWeaver achieves the highest speedup of 1.50× for SFR, followed by 1.45× speedup for failure-atomic transactions, and 1.40× speedup for ATLAS.

5.5.3 Sensitivity Study

Figure 5.8 shows the evaluation of StrandWeaver with varying number of strand buffers and entries per strand buffer. Due to space limitations, we show only the results for the SFR implementation—the performance trend for the other implementations is similar. With fewer than four entries per buffer, the strand buffer unit fails to leverage available persist concurrency on different strands, even when we configure the unit with four buffers. As we increase the number of buffer entries to four, even with two strand buffers, StrandWeaver's performance improves by 1.36×, as persists on different strands can drain concurrently. Finally, StrandWeaver's performance improves by a further 7.7% with four strand buffers and four buffer entries each. As we see no further improvement with additional state (*e.g.* eight strand buffers with eight buffer entries, in Figure 5.8), we configure the strand buffer unit with four buffers, each with four entries.

CHAPTER VI

Related Work

The adoption of PMs has been widely studied by both academia and industry in hardware design [56, 171, 112, 122, 62, 160, 223, 187, 111, 82], file systems [56, 208, 200, 211, 50, 212, 64, 210], runtime systems [205, 48, 167, 120, 25, 24, 126, 136, 47, 78, 150, 84, 77, 165, 81, 162, 95, 109, 43], persistent data structures [199, 163, 96, 49], and distributed systems [118, 222, 226, 142]. This thesis discusses works that address wear out problem in PMs in Chapter III, defines persistency semantics for high-level programming languages in Chapter IV, and proposes relaxed persistency model in hardware in Chapter V.

We discuss the relevant works that address wear out problem in PMs.

6.1 Wear-reduction Mechanisms

We first discuss techniques that reduce PM writes.

DRAM cache. Numerous works [175, 183, 68, 153] advocate placing a DRAM cache in front of PM. The DRAM cache absorbs most of the writes thereby reducing wear. A DRAM cache presents three disadvantages: (1) it sacrifices capacity that could instead be used to expand memory; (2) it increases the latency of PM writes; and (3) it is inapplicable to writes that require persistency, which must write through the cache. Like many prior works [56, 171, 112, 122, 62, 160, 223, 187, 111, 78, 80], we assume that PM and DRAM are peers on the memory bus.

Page migration. Several works [59, 176, 220, 19] propose migrating pages from PM to DRAM to reduce wear. Dhiman et al. [59] use a software-hardware hybrid solution, where dedicated hardware counters (one per PM page) that track page hotness are maintained in PM and cached in the memory controller. RaPP [176] and Zhang et al. [220] use a set of queues in the memory controller to estimate write intensiveness and perform page migrations to DRAM. However, these mechanisms propose no wear-leveling solutions for the remaining pages in PM. As such, these mechanisms may still not achieve desired PM device lifetimes. For example, RaPP can achieve a device lifetimes exceeding 3 years only if the cell endurance exceeds 10^9 [176] – insufficient for PCM-based memories with endurance of only 10^7 - 10^9 writes. Moreover, these mechanisms do not support applications that require crash consistency when using PM as storage [160].

Heterogeneous main memory: Several works [16, 116, 172] manage footprint between DRAM and PM for applications that prefer DRAMs for high performance. These works map heavily and least accessed regions of application footprint to DRAM and PM respectively.

Currently, Kevlar operates at a small (4KB) page granularity. However, huge (2MB) pages are increasingly being used to minimize performance penalties of using small pages (due to increased TLB pressure), especially in virtualized systems. Kevlar can be further extended to operate at a huge page granularity. For instance, Kevlar can be integrated with mechanisms such as Thermostat [16] to split a huge page into small pages, monitor write rate at granularity of small pages, and migrate pages between DRAM and PM. We leave evaluation of Kevlar's wear-reduction mechanism and development of shuffling strategies to operate at a huge page granularity to future work.

Other. DCW [225] performs read-compare-write operation to ensure that only the data bits that have changed are written. Bittman et al. [38] proposes data structures aimed at minimizing the number of bit-flips per PM write operation. The downside of these mechanisms is that writes become slower and consume more memory bandwidth. Flip-N-

Write [53] extends DCW and further reduces the number of bits written by inverting the bit representation of the data when it reduces the number of modified bits. Each memory location is extended with a "flip" bit to indicate if the associated data has been flipped or not. Ferreira et al. [68] enable eviction of clean cache lines over dirty cache lines at the expense of potentially slowing down future reads to evicted cache lines. Recent works, MCT [58] and Mellow Writes [218], improve the endurance by reconfiguring memory voltage levels and slowing write accesses to the PM. These proposals can achieve high device lifetime but at a significant performance overhead, especially when write latency is critical to application performance [160]. NVM-Duet [137] employs a smart-refresh mechanism to eliminate redundant memory refresh operations thereby reducing PM wear. Others [108, 224] propose solutions to manage wear when using persistent memory technologies to build caches.

6.2 Wear-leveling Mechanisms

Qureshi et al. [175], Zhou et al. [225], Security refresh [183], Online Attack Detection [174] and Start-Gap [173] observe that cache lines within a PM page do not wear out equally and propose mechanisms to remap cache lines for uniform intra-page wear. Zhou et al. [225] propose Row Shifting to rotate a PM row by one byte at a time to level intra-row wear and Segment Swapping to swap the frequently written segments with sparingly written ones. Row Shifting rotates a PM row by one byte at a time to level intra-row wear. Segment Swapping tracks PM segments for hotness and swaps the frequent written segments with sparingly written ones. Instead of using a table-based address translation mechanism, Start-Gap wear leveling [173] uses an algebraic formula as its address translation mechanism. Security refresh [183] uses a separate randomized address translation mechanism to not only achieve wear-leveling under normal operating conditions but also prevent a malicious program from intentionally wearing out certain memory locations with targeted writes. However, such secure wear leveling mechanisms incur performance overheads that are not necessary under non-malicious operation. Online Attack Detection [174]

scales the rate of wear leveling based on whether or not the memory device is under attack. All of these works rely on additional address indirection mechanisms in hardware.

Error recovery. DRM [103] gracefully degrades PM capacity as memory cells wear out by remapping corresponding virtual page to two different physical pages with non-overlapping faulty regions. When regions within a PM page become faulty, DRM maps the corresponding virtual page to two different physical pages with non-overlapping faulty regions. SAFER [184] observes that a failed cell with a "stuck-at" value is still readable, making it possible to continue to use the failed cell to store data. FREE-p [216] and NVMAlloc [158] leverage ECC and checksum mechanisms to tolerate wear out errors. Moraru [158] proposes a wear-aware memory allocation mechanism in OS, but uses checksum metadata to recover from wear outs at runtime. Awasthi et al. [28] propose error scrubbing mechanisms targeted at the "resistance drift" problem seen in PCM memory devices.

6.3 Software-based Mechanisms

This chapter discusses the related software library, runtime and checkpointing mechanisms.

Library-based mechanisms: NV-Heaps [54] and Mnemosyne [201] provide library-based application-level interfaces for building persistent objects in PM. Both provide libraries to create virtually mapped regions in persistent memory, along with primitives to update persistent data mapped to the memory. They use write-ahead logging to provide failure atomicity for transactions. SoftWrAP [75] and REWIND [48] provide software libraries to perform transactional updates to PM. SoftWrAP uses alias tables to redirect the updates within the failure-atomic transactions to a log space in DRAM and commits the updates when the transactions retire. Similar to SoftWrAP, DUDETM [136] updates the redo logs for transactions in DRAM, and then persists and merges the logs in PM. Kamino-Tx [150] avoids logging by replicating the heap, performing updates within transactions

on a working copy of the heap, and copying changes to the backup heap when transactions commit. Transactions simplify logging, both in hardware and software. However, our approach differs from software-annotated transaction-based solutions in that it is applicable to general-purpose programs that are not transaction-based, especially those that use synchronization mechanisms like conditional waits or complex locks that do not readily compose with transactional models. In this work, we seek to provide persistency semantics for arbitrary (non-transactional) synchronization.

Runtime logging solutions: NVthreads [91] extends ATLAS [47] to provide durability guarantees to lock-based programs. NVthreads uses copy-on-write to make updates within a critical section and then merges the updates to the live data at a 4KB page granularity at the end of outermost critical sections. Due to the expensive merge operations at the end of the critical sections, NVthreads suffers a high performance overhead in applications with frequent lock acquisition and release operations like the benchmarks that we study in this paper. Moreover, we extend durability semantics to more general synchronization constructs that NVthreads and ATLAS do not support. Boehm et al. [43] elaborates on the ATLAS programming model further and defines recovery semantics for updates to persistent locations both within and outside critical sections. ARP [123] and Izraelevitz et al. [109] propose language-level persistency models. Both works provide persist ordering, but fail to provide failure atomicity at a granularity larger than individual persists. Moreover, they offer unclear semantics at failure, as writes may be replaced from the cache hierarchy and persist well before other, earlier writes, exposing non-SC state to recovery. TARP [124] and Izraelevitz et al. [106] offer x86 and ARM ISA encodings of language-level persistency models. Kolli et al. [126] introduces efficient implementation of transactions, namely synchronous-commit and deferred-commit transactions, that minimize persist dependencies by deferring commit of undo logs until the transactions conflict.

WSP [161] proposes mechanisms to flush the precise architectural state of a program at the moment of failure to PM. JUSTDO logging [104] recovers an application to its state

right before the failure, and requires persisting of architectural state including stack-local variables before executing a critical section. It assumes the cache hierarchy is persistent to avoid high PM access latency when preserving volatile program state. Both WSP and JUSTDO logging fail to provide recoverability from failures other than power interruptions (*e.g.* kernel panic or application crash). SCMFS [208], BPFS [56], NOVA [211], NOVA-Fortis [212] and PMFS [64] propose filesystems that leverage low latency of PMs.

Checkpointing-based solutions: ThyNVM [178] proposes dual-scheme checkpointing mechanism to provide crash consistency support for DRAM+NVM systems. It eliminates stalls for checkpointing by overlapping execution and checkpointing. CC-HTM [76] leverages HTM to provide fine-grained checkpointing of transactions to PM. Survive [155] provides a fine-grained incremental checkpointing for hybrid DRAM+PM systems. Other works checkpoint the volatile state using cache persistence by ensuring that a battery backup is available to flush the volatile state to PM upon power failure [163], or by bypassing caches altogether [205].

Energy harvesting systems: A group of studies look at application consistency requirements for energy harvesting devices. As the energy supply for this class of devices is intermittent, these works explore mechanisms to maintain data consistency in PM while ensuring forward progress. Alpaca [147] provides a task-based programming model, where tasks present an abstraction for the atomicity of updates in PM. Alpaca requires programmers to annotate tasks and task-specific shared variables in the program. We provide a more generic mechanism built upon existing C++ synchronization. Idetic [154] and Hibernus [30] detect imminent power failure and periodically checkpoint volatile state, but may leave data in PM inconsistent [55]. This group of works propose consistency mechanisms for power failures alone, whereas we consider more general fail-stop failures.

6.4 Hardware-based Mechanisms

Pelley et al. [171, 126] proposes persistency models closely aligned with the hardware memory consistency model to order writes to PM. His work proposes strict and relaxed persistency models that vary in the constraints imposed on the updates as they persist to PM. BPFS [56] uses epoch barriers to order persists in hardware. The persists within an epoch can be reordered while persist reordering across epochs is disallowed. DPO [122], Doshi et al. [62], HOPS [160], and Shin et al. [188] propose hardware mechanisms for efficiently implementing epoch persistency models. They implement hardware structures in the cache hierarchy that record and drain persists to the PM in order. ATOM [111] improves upon undo-logging mechanism for PM by decoupling the update of undo log from the in-place update to the persistent data-structure. It relies on hardware structures in the memory hierarchy that order logs before the actual updates to PM. FIRM [209], Ogleari et al. [166], and DHTM [113] build undo- or hybrid undo-redo logging mechanisms in hardware. These hardware mechanisms primarily provide failure atomicity for transactions, but fail to extend to other synchronization primitives or other logging implementations used by high-level language persistency models [47, 78, 123]. Proteus [187] implements a software-assisted hardware solution to persist transactions atomically to PM. It involves significant modifications to the processor pipeline to record logs and order logs with respect to subsequent stores. Liu et al. [138] proposes an encryption mechanism based on counter-mode encryption. It employs hardware mechanisms to ensure atomicity of data and the associated counter used for its encryption in PM. Kiln [223] and LOC [141] provide a storage interface to PM to programmers, but rely on programmers to ensure isolation. Unlike hardware-based solutions, we use synchronization primitives in the C++ memory model to provide ordering and failure-atomicity to the PM updates.

CHAPTER VII

Conclusion and Future Work

PM technologies, such as Intel's 3D Xpoint, blur the distinction between memory and storage. The byte-addressable PMs avoid the performance inefficient software layers required for block-based storage devices and enable fine-grained manipulations to the persistent data-structures. Future systems can employ PMs to store data durably, reconstruct required volatile state, and resume program execution. However, this thesis notes several challenges concerning low write endurance of PMs and inefficiencies in programming and hardware systems that need to be addressed before PMs can be integrated in the future systems. This thesis makes contributions summarized below.

7.1 Conclusion

Chapter III presented Kevlar, a wear-management mechanism for persistent memories. Kevlar relies on a software *wear-estimation* mechanism that uses PEBS-based sampling in a novel approach to estimate dirty cache contents and predict writebacks to PM. Kevlar uses a two-pronged approach to improve PM device lifetime. It uses a *wear-leveling* mechanism that shuffles PM pages every ~4 hours with an overhead of less than 0.10% achieving up to 31.7× higher lifetime as compared to PM with no wear leveling. Kevlar employs *wear-reduction* mechanism to further extend PM lifetime. It migrates the hottest pages to higher durability memory. Kevlar, implemented in Linux kernel (version 4.5.0), achieves four-

year target lifetime with 1.2% performance overhead.

Chapter IV made a strong case for building persistency semantics upon the strong foundations of the data-race-free (DRF) memory model of C++, using existing C++ synchronization operations to prescribe ordering for persists. Past works have proposed language-level persistency models prescribing semantics for updates to PM. However, we showed that the existing language-level persistency models either lack precise durability semantics or incur a high performance overhead. We made a case that failure-atomic SFRs strike a compelling balance between programmability and performance. We then examined two designs, Coupled-SFR and Decoupled-SFR, for failure-atomic SFRs that vary in performance and the amount by which the PM state may lag execution. We show that our designs simplify logging and outperform the state-of-the-art implementation by 87.5% (65.5% avg).

Chapter V proposed StrandWeaver, a hardware strand persistency model to minimally constrain orderings on PM operations. We formally defined primitives under strand persistency to specify intra-strand, inter-strand, and inter-thread persist ordering constraints. We constructed hardware mechanisms to implement strand persistency model that expose ISA primitives to relax persist order. Furthermore, we implemented the state-of-the-art language-level-persistency models that map persistency semantics in high-level languages to the low-level ISA primitives using our logging mechanism. Finally, we demonstrated that StrandWeaver can achieve 1.45× speedup on average as it can enable greater persist concurrency than existing ISA-level mechanisms.

7.2 Future Work

This thesis makes a strong case for developing efficient persistency models in future hardware systems and programming languages. However, open questions remain concerning the future applications that want to leverage PMs as storage. We leave these problems to future work.

Software systems for PMs. This thesis seeks to extend the persistency semantics of

high-level programming languages, using existing synchronization operations order persists. However, standardization of such semantics in high-level programming languages might be difficult until the PM devices are widely available. In the short term, expertly handcrafted portable software libraries (analogous to Boost [3]) that build commonly used PM data-structures (*e.g.* trie, hashmap and vector) might aid development of storage software for PMs. The software libraries may design efficient implementations of PM data-structures that impose fewer ordering constraints on PM operations — the libraries may coalesce PM accesses to improve the bandwidth utilization and reduce frequent stalls due to `CLWB-SFENCE` ordering. The storage applications developed using these software libraries resulting may accelerate adoption of PM-based storage.

The storage applications may also use PMs for their density. For instance, storage applications cache frequently used storage blocks in DRAM for faster access. The DRAM cache miss is expensive; blocks need to be fetched from a slower storage device such as Flash. A large capacity and cheaper PMs may be employed to cache frequently accessed storage blocks. Since PMs are expensive yet denser than DRAMs, several interesting design choices may be studied for a tiered systems with DRAM, PMs, and Flash.

Reliability of storage applications. PMs enable fine-grained storage data manipulation. Unfortunately, this also increases the risk of data corruption — a stray incorrectly-ordered cache-line writeback to the PM may result in an irrecoverable data in case of a failure. We require testing mechanisms that may verify the applications for their recovery correctness. The testing mechanisms may inject random failures during runtime, or employ an exhaustive approach to ensure that updates to PM are correctly ordered. Recent works [138, 139] also propose data-at-rest encryption mechanisms for PMs. The testing frameworks for PMs may also verify that plaintext data is not visible to recovery in case of failure.

Persistency models for remote PMs. Persistency models for hardware systems and programming languages have been defined earlier for the single-node local PMs alone.

These persistency models define the state the recovery would observe in case of failure. Meanwhile, several innovations in networking technologies such as RDMA have reduced network latencies to a few microseconds [85, 196]. RDMA networking technologies may be coupled with the byte-addressable PMs to build fast, reliable, and fault-tolerant distributed storage systems. The persistency models, currently ill-defined for remote PM use-cases, may be extended to define the PM state in presence of network failures. These persistency models may define the ordering constraints on updates to remote PMs, including the ordering guarantees required from networking hardware. These models may also define the mechanisms required to flush the updates from volatile hardware and NIC caches to the remote PMs.

APPENDIX

APPENDIX A

HARE: Hardware Accelerator for Regular Expressions

A.1 Introduction

Fast analysis of unstructured textual data, such as system logs, social media posts, emails, or news articles, is growing ever more important in technical and business data analytics applications [179]. Nearly 85% of business data is in the form of unstructured textual logs [12]. Rapidly extracting information from these text sources can be critical for business decision making. For instance, a business might analyze trends in social media posts to better target their advertising budgets.

Regular expressions (regexps) provide a powerful and flexible approach for processing text and unstructured data [9]. Historically, tools for regexp processing have been designed to match disk or network [144] bandwidth. As we will show, the most widely used regexp scanning tool, `grep`, typically achieves at most 100-300 MB/s scanning bandwidth on modern servers—a tiny fraction of available memory bandwidth. However, the wide availability of cheap DRAM and upcoming NVRAM [6] allows many important data corpora to be stored entirely in high-bandwidth memory. Data management systems are being redesigned for in-memory datasets [148, 194]. Text processing solutions, and especially regexp processing, require a similar redesign to match the bandwidth available in modern system architectures.

Conventional software solutions for regexp processing are inefficient because they rely

on finite automata [90]. The large transition tables of these automata lead to high access latencies to consume an input character and advance to the next state. Moreover, automata are inherently sequential [27]—they are designed to consume only a single input character per step. Straight-forward parallelization to multi-character inputs leads to exponential growth in the state space [92].

A common approach to parallelize regexp scans is to shard the input into multiple streams that are scanned in parallel on different cores [151, 159, 89]. However, the scan rate of each individual core is so poor (especially when scanning for several regexps concurrently) that even the large core counts of upcoming multicore server processors fall short of saturating memory bandwidth [195]. Moreover, such scans are highly energy inefficient. Other work seeks to use SIMD parallelism [180, 45] to accelerate regexp processing, but achieves only modest 2×-3× speedups over non-SIMD software.

Instead, our recent work on the HAWK text scan accelerator [195] has identified a strategy to scan text corpora using finite state automata at the full bandwidth of modern memory systems, and has been demonstrated for scan rates as high as 32 giga-characters per second (GC/s; 256 Gbit/s). HAWK relies on three ideas: (1) a fully-pipelined hardware scan accelerator that does not stall, assuring a fixed scan rate, (2) the use of bit-split finite state automata [135] to compress classic deterministic finite automata for string matching [18] to fit in on-chip lookup tables, and (3) a scheme to efficiently generalize these automata to process a window of characters each step by padding search strings with wildcards. We elaborate on these prior ideas in Section A.3.

HAWK suffers from two critical deficiencies: (1) it can only scan for *exact* string matches and fixed-length patterns containing single-character (`.`) wildcards, and (2) it is unable to process Kleene operators (+, *), alternation (|,?), and character classes (`[a-z]`), which are ubiquitous in practical text and network packet processing [9, 11]. These restrictions arise because HAWK's strategy for processing multiple input characters in each automaton step cannot cope with variable-length matches.

We propose *HARE* [79], the Hardware Accelerator for Regular Expressions, which extends the HAWK architecture to a broad class of regexps. HARE maintains HAWK's stall-free pipeline design, operating at a fixed 32 GC/s scan rate, regardless of the regexps for which it scans or input text it processes. Similar to HAWK, we target a throughput of 32GB/s because it is a convenient power-of-two and representative of future DDR3 or DDR4 memory systems. HARE extends HAWK in two key ways. First, it supports character classes by adding a new pipeline stage that detects in which character classes the input characters lie, extending HAWK's bit-split automata with additional bits to represent these classes. Second, it uses a counter-based mechanism to implement regexp quantifiers, such as the Kleene Star (*), that match repeating characters. The combination of repetition and character classes presents a particular challenge when consecutive classes accept overlapping sets of characters, as some inputs may match an expression in multiple ways.

We evaluate HARE through:

- An ASIC RTL implementation of a stall-free HARE pipeline operating at 1GHz and processing 32 characters per cycle, synthesized using a commercial 45nm design library. We show that HARE can indeed saturate a 32GB/s memory bandwidth—performance far superior to existing software and hardware approaches.
- A scaled-down FPGA prototype operating at 100 MHz processing 4 characters per cycle. We show that even this scaled-down prototype outperforms traditional software solutions like `grep`.

A.2 Overview

HARE seeks to scan in-memory text corpora for a set of regexps while fully exploiting available memory bandwidth.

A.2.1 Preliminaries

HARE builds on the previous HAWK architecture [195], which provides a strategy for processing character windows without an explosion in the size of the required automata. HARE extends this paradigm to support two challenging features of regular expressions: character classes and quantifiers.

HARE is not able to process all regular expressions as no fixed-scan-rate accelerator can do so; some expressions inherently require either backtracking or prohibitive automata constructions, such as determinization. Moreover, when allowing combinations of features, such as Kleene star and bounded repetitions, even building a non-deterministic automaton can incur an exponential blowup [190].

We extend HAWK to support character classes, alternations, Kleene operators, bounded repetitions, and optional quantifiers. HARE allows Kleene (+,*) operators to be applied only to single characters (or classes/wild-cards) and not multi-character sub-expressions. Nevertheless, we demonstrate that this subset of regexps covers the majority of real-world regexp use cases.

A.2.2 Design Overview

HARE's design comprises a stall-free hardware pipeline and a software compiler. The compiler transforms a set of regexps into state transition tables for the automata that implement the matching process and configures other aspects of the hardware pipeline, such as look-up tables used for character classes and the configuration of various pipeline stages.

Figure A.1 depicts a high-level block diagram of HARE's hardware pipeline. The figure depicts HARE as a six logical stages, where input text originates in main memory and matches are emitted to post-processing software (via a ring-buffer in memory). Note that individual logical stages are pipelined over multiple clock cycles to meet timing constraints. The two stages marked in orange (Character Class Unit, CCU; and Counter-based Reduction Unit, CRU) are newly added in HARE and provide the functionality to support

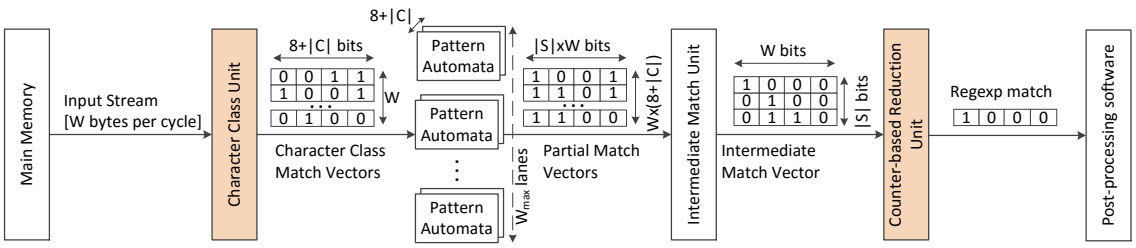


Figure A.1: **HARE block diagram.** The hardware pipeline enables stall-free processing of regexps. Shaded components are newly added relative to the baseline HAWK design.

regexps; the remaining stages are similar to units present in the HAWK baseline, which can match only fixed-length strings.

A HARE accelerator instance is parameterized by its width W, the number of input characters it processes per cycle. HARE streams data from main memory, using simple stream buffers to manage contention with other cores/units. W incoming characters are first processed by the CCU, which uses compact look-up tables to determine to which of $|C|$ pre-compiled character classes (those appearing in the input regexp) the input characters belong. The CCU outputs the original input characters (W×8 bits) augmented with additional W×$|C|$ bits indicating if each input character belongs to a particular character class.

The Pattern Automata perform the actual matching, navigating the set of automata constructed by the HARE compiler to match the sub-expressions of the input regexp. To make the state transition tables tractable, the Pattern Automata rely on the concept of *bit-split state machines* [135], wherein each pattern automaton searches for matches using only a subset of the bits of each input character. Bit-split state machines reduce the number of outgoing transition edges (to two in the case of single-bit automata) per state, drastically reducing storage requirements while facilitating fixed-latency lookups. We detail the bit-split concept and how we extend it to handle character classes in Section A.3.2.

Each pattern automaton outputs a bit vector indicating strings that *may* have matched at each input position, for the subset of bits examined by that automaton in the present cycle. These bit vectors are called *partial match vectors* or PMVs. A sub-expression of the regexp

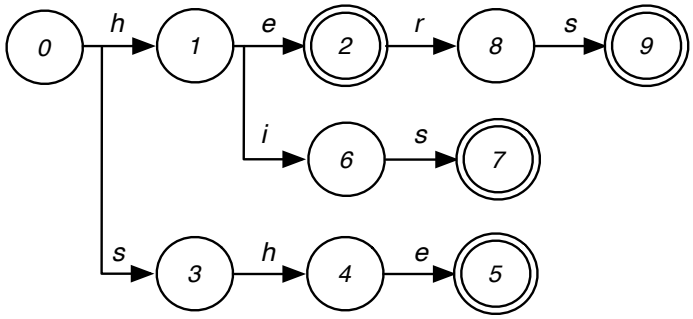


Figure A.2: **An Aho-Corasick pattern matching automaton.** Automaton for search patterns *he*, *hers*, *his*, and *she*. States *2*, *5*, *7*, and *9* are accepting.

matches in the input text only if it is matched in *all* partial match vectors. The Intermediate Match Unit computes the intersection of all PMVs, called the intermediate match vector or IMV, using a tree of AND gates.

HAWK is only able to match fixed-length strings. Variable length matches pose a problem because they thwart HAWK's strategy for addressing the multiple possible alignments of each search string with respect to the window of W characters processed in each cycle. The central innovation of HARE is to split each regexp into multiple fixed-length sub-expressions called *components* and match the components separately using the pattern automata and intermediate match unit. The next stage, the Counter-based Reduction Unit, combines separate matches of the components and resolves ambiguities that arise due to concatenated character classes to determine a final match. This stage also allows it to handle Kleene (+,*), and bounded repetition (`{a,b}`) quantifiers in the presence of (potentially overlapping) character classes. Quantifiers pose a challenge because they can match a variable number of input characters. We elaborate on these issues in in Section A.3.4

A.3 From HAWK to HARE

HARE builds on HAWK [195], which itself builds on the Aho-Corasick algorithm [18] for matching strings.

A.3.1 Aho-Corasick Algorithm

The Aho-Corasick algorithm [18] is widely used for locating multiple strings (denoted by the set S) in a single scan of a text corpus. The algorithm centers around constructing a deterministic finite automaton for matching S. Each state in the automaton represents

the longest prefix of strings in S that match the recently consumed characters in the input text. The state transitions that extend a match form a trie (prefix tree) of all strings accepted by the automaton. The automaton also has a set of accepting states that consume the last character of a string; an accepting state may emit multiple matches if several strings share a common suffix. Figure A.2 illustrates an Aho-Corasick automaton that accepts the strings {he,she,his,hers} (transitions that do not extend a match are omitted).

The classic Aho-Corasick automaton is a poor match for hardware acceleration, due to two key flaws:

• **High storage requirement**: The storage requirements of the state transitions overwhelm on-chip resources. To facilitate fixed-latency next-state lookup (essential to achieve a stall-free hardware pipeline), transitions must be encoded in a lookup table. The size of the required lookup table is the product of the number of states $|S|$ and the alphabet size $|\alpha|$, which rapidly becomes prohibitive for an ASCII text.

• **One character per step**: In the classic formulation, the Aho-Corasick automaton consumes only a single character per step. Hence, meeting our performance goal of saturating memory bandwidth (32GBps) either requires an infeasible 32-GHz clock frequency or consuming multiple characters per step. One can scale the classic algorithm by building an automaton that processes digrams, trigrams, W-grams, etc. However, the number of outgoing transition edges from an automaton grows exponentially in the width W, yielding $|\alpha|^W$ transition edges per state. Constructing and storing such an automaton for even modest W is not feasible.

A.3.2 Bit-split Automata

HAWK overcomes the storage challenge of the classic Aho-Corasick automaton using *bit-split automata* [135]. This method splits an Aho-Corasick automaton that consumes one character per step into an array of automata that operate in parallel and each consume only a subset of the bit positions of each input character. The state of each bit-split automaton now represents the longest matching prefix for its assigned bit positions, and its output function indicates the set of possibly matching strings; HAWK represents this set as a bit vector called a *partial match vector* (PMV). The output function of the original Aho-Corasick automaton is the disjunction of these PMVs, which HAWK implements via a tree of AND gates in its Intermediate Match Unit.

The bit-split technique reduces the number of outgoing edges per state. In HAWK, each automaton examines only a single input bit, hence, there are only two transition edges per state, which are easy to store in a deterministic-latency lookup table.

A.3.3 Scaling to $W > 1$

The bit-split technique drastically reduces storage, but still consumes only a single character per machine step. The primary contribution of HAWK is to extend this concept to consume a window of $W = 32$ characters per step, to search for $|S|$ strings using an array of $|S| \times W$ 1-bit automata operating in lock-step.

The key challenge to processing W characters per step is to account for the arbitrary alignment of each search string with respect to the window of W positions. For example, consider an input search string *he* in input text *heatthen*, processed four characters at a time. While *he* begins at the first position in first four-character window (*heat*), it begins at the second position in the second window (*then*).

HAWK addresses this challenge by rewriting each search string into W strings corresponding to the W possible alignments of the original string with respect to the window, padding each possible alignment with wildcard (`.`) characters to a length that is a multiple

of W. For example, for the string *he* and $W = 4$, HAWK will configure the hardware to search concurrently for <*he..*>,<*.he.*>, <*..he*>, and <*...h e...*>.

A.3.4 Challenges of Regexps

HAWK's hardware is sufficient to search for exact string matches and single-character (`.`) wildcards. However, HAWK's alignment/padding strategy is thwarted by regular expression quantifiers, because quantifiers may match a variable number of characters. To generalize HAWK's padding strategy in a straight-forward way, we must rewrite a single regexp containing a quantifier (e.g., `ab*c`) to consider all possible alignments of the prefix and all possible widths of the quantifier sub-expression, which rapidly leads to an infeasible combinatorial explosion.

HAWK's approach is further confounded by character classes, especially in cases involving multiple character classes. Consider, for example, the regular expression [a-f][o-r]ray can match six characters in the first position (characters *a* to *f*) and four characters in the second position (characters *o* to *r*). HAWK needs to enumerate the characters within the range of a character class to create all possible strings the character class can potentially match—24 patterns in the above example.

A.4 HARE Design

We now describe the details of HARE's compilation steps and hardware units. We refer readers to [195] for the details of constructing bit-split automata and microarchitectural details of the pattern automata and intermediate match unit, which we only summarize here.

A.4.1 HARE Compiler

HARE's compiler translates a set of regexps into configurations for each of its stages. The compilation process proceeds in four steps: (1) split components, (2) compute prece-

dence vectors and repetition bounds, (3) compile character classes, and (4) generate bit-split machines. Then, HARE invokes HAWK's existing compilation steps to construct bit-split automata and generate a bit stream to load into the accelerator. We describe the new compilation steps for regular expressions.

A.4.1.1 Component splitting

As previously noted, HAWK's string padding solution, which enables it to recognize matches that are arbitrarily aligned to the W character window scanned in each cycle, does not generalize to sub-expressions of a regexp that may match a variable number of characters.

Instead of pre-constructing an exponential number of pattern alignments, a key idea in HARE is to instead search for smaller, fixed-length sub-expressions of a regexp separately (and concurrently) and then confirm if the partial matches are concatenated (and possibly repeated) in a sequence that comprises a complete match. So, the first step of compilation is to split a regexp into a sequence of such sub-expressions, which we call *components*. The baseline HAWK is already able to scan for multiple fixed-length strings at arbitrary alignments; HARE configures it to search concurrently for all components comprising a regexp. The HARE compiler splits a regexp at the start and end of the operand of every quantifier (`?`, `*`, `+`, `{a,b}`) and alternation (`|`). (As previously noted, HARE does not support repetition operators applied to multi-character sequences).

Consider the example regexp `abc+de`, containing a Kleene Plus operator. The compiler splits the regexp at the operand of the Kleene Plus, `c`, resulting in three components `ab`, `c`, and `de`. The pattern automata are configured to search separately for these components (at all alignments). After reduction in the intermediate match unit, each IMV bit corresponds to a particular component detected at a particular alignment. These IMV bits are then processed in the counter-based reduction unit to identify matches of the full expression.

A.4.1.2 Compute precedence vectors

To locate a complete regexp match, HARE checks that components occur in the input stream in a sequence accepted by the regexp. As a regexp is split into multiple components, the compiler maintains a *precedence vector* that indicates which components may precede a given component in a valid match. The precedence vector for the first component is the empty set. Subsequent components include in their precedence vector all components that may precede them in a legal match. For example, a component following an optional (`?`) operator includes both the optional component and its predecessor in its precedence vector. We enumerate the rules for computing precedence vectors for each operator below. Along with the precedence vector, the compiler also records an upper and lower repetition bound for each component. For literal components (i.e., not a quantifier operand), the bounds are simply [1,1], otherwise, the bounds are determined by the quantifier.

Together, the precedence vectors and repetition bounds are used by the CRU to determine if a sequence of components (represented in the stream of IMVs consumed by the unit) constitutes a match. We next outline how to compute precedence vectors and repetition bounds for each operator.

• **Alternation** – An alternation operator (`|`) indicates that multiple components may occur at the same position in a matching input. The precedence vector for a component following an alternation includes all alternatives. For instance, for a regexp `gr(e|a)y` consisting of components `gr`, `e`, `a` and `y`, either component `e` or component `a` can appear after component `gr`. So, the precedence vectors for components `e` and `a` include component `gr`, while the vector for component `y` includes both components `e` and `a`. The lower and upper bounds for each alternative are determined by their sub-expressions (e.g., [1,1] for literals).

• **Optional quantifier** – A component followed by an optional quantifier can appear zero or one time. The successor of an optional component includes the optional component and its predecessor in its precedence vector. For example, for regexp `ab?c`, consisting of components `a`, `b`, and `c`, the precedence vector for `b` includes only `a`. However, the precedence

vector for c includes both a and b. The bounds for optional components are [1,1]. Note that the minimum bound for component b is not zero; if the component appears, it must appear at least once. The possibility that the component b may not appear is reflected in the precedence vector of component c.

• **Bounded repetition quantifier** – A bounded repetition quantifier sets a range of allowed consecutive occurrences of a component. For instance, the expression ab{2,4}c matches an input text starting with a followed by two, three, or four consecutive occurrences of b and finally terminating with c. Since all the components must appear at least once in the sequence, the precedence vector for each component includes only its immediate predecessor. The min and max bounds of component b are configured to match the bounds of the repetition quantifier *i.e.* [2,4]. Our implementation constrains bounds to a maximum of 256 to limit the width of the counters in the counter-based reduction unit.

• **Kleene Plus** – The operand of a Kleene Plus must appear one or more times in a match. Hence, each component's precedence vector includes only its immediate antecedent. For the earlier example abc+de, the precedence vector of c includes only ab and de includes only c. The max bound of a Kleene Plus operand is set to a special value indicating an unbounded number of repetitions. So, the min and max bound on components ab and de are [1,1], whereas, for c the bounds are [1,inf].

• **Kleene Star** – A Kleene Star (*), which matches a component zero or more times, is handled as if it were a Kleene Plus followed by an optional quantifier ((+)?). So, the precedence vector of its successor component includes it and its predecessor. In a regexp ab*c, the component c can either follow one or more repetitions of component b or a single instance of component a. Its precedence vector thus includes both the components a and b. Like the Kleene Plus, the bounds for the operand of a Kleene Star are set to [1,inf]. As with optional components, the minimum bound of component b is not zero; if the component appears, it appears at least once.

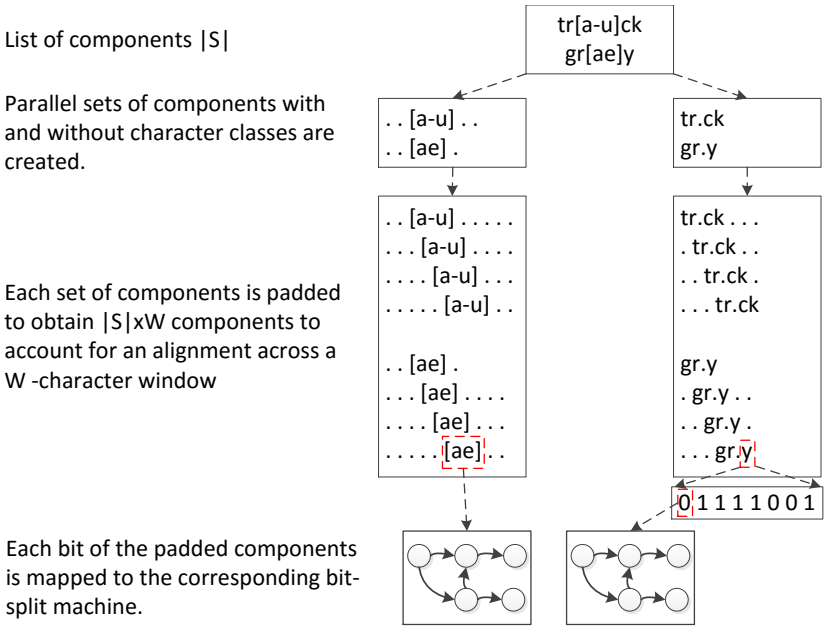


Figure A.3: **Compiling components containing character classes.** The components containing character classes are split in two, separating character classes from literals. These sets are separately padded and compiled to create bit-split automata.

A.4.1.3 Compiling character classes

Character classes define sets of characters that may match at a particular input position. For instance, the regexp `tr[a-u]ck` matches ASCII characters between `a` and `u` at the third position, including strings `track` and `truck`. The naive approach of expanding character classes by enumerating all the characters in the character class range and matching all such patterns separately rapidly leads to blowup in the size of the automata. Bit-split automata, as used in HAWK, provide no direct support for character classes and must resort to such alternation.

We observe that we can augment the eight bit-split automata that process a single character with additional automata that process arbitrary Boolean conditions, for example, whether a character belongs to a particular character class. We determine if an ASCII character belongs to a class using a simple lookup table in HARE's CCU. For each character class in the regexp, the compiler emits a 256-bit vector, wherein a given bit is set if the corresponding ASCII character belongs to the class. For instance, for the character

class [a-u], bits 97 (corresponding to a) through 117 (corresponding to u) are set. These vectors are programmed into HARE's CCU, which outputs a one when a character falls within the class. Note that our scheme can be readily extended to Unicode character ranges by replacing the lookup table with range comparators.

Next, HARE breaks components containing character classes into two separate components, one comprising only literal characters, where character classes are replaced with single-character (.) wildcards, and the second comprising only character classes, with literals replaced by wild cards. Figure A.3 illustrates the process of breaking and padding (for a 4-wide accelerator) these components for two example regexps including character classes. The regexps tr[a-u]ck and gr[ae]y consist of only a single component as they do not have any operators. The literal components are encoded in pattern automata exactly as in HAWK. The character class component uses the additional pattern automata that receive the output of the CCU. Both patterns are then padded for all possible alignments, as in the HAWK baseline.

Note that the main complexity of character classes arises in regexps where classes with overlapping character sets may occur at the same position in matching inputs (e.g., due to an alternation or Kleene operator). Placing classes into separate components facilitates their handling in the reduction stage.

A.4.1.4 Generate bit-split state machines

Once the two sets of components (one comprising only literal characters, the other comprising character classes) are generated, HARE's compiler invokes HAWK's algorithm to generate the bit-split machines processing W-characters per clock cycle. As illustrated in Figure A.3, the two sets of components are padded front and back with wildcard characters to account for their alignment within a W-character window. The compiler then generates bit-split automata for the padded components according to the algorithm proposed by Tan and Sherwood [135].

A.4.2 HARE Hardware Units

We next describe the microarchitecture of HARE's hardware pipeline, as depicted in Figure A.1 and Figure A.4.

A.4.2.1 Character Class Unit

Figure A.4 (top) illustrates the character class unit (CCU). For each character class used in a regexp, the HARE compiler emits a 256-bit vector indicating which characters belong to the class. These vectors are programmed into a W-ported lookup table in the CCU. We denote the number of classes supported by the unit as $|C|$. Each of the W characters that enter the accelerator pipeline each clock cycle probes the lookup table and reads a $|C|$-bit vector indicating to which classes, if any, that character belongs. These $|C|$-bit vectors augment the 8-bit ASCII encodings of each character and all are passed to the pattern automata units.

A.4.2.2 Pattern Automata

As described in Section A.3.3, HAWK provisions W×8 bit-split automata to process a W-wide window of 8-bit ASCII characters each clock cycle. These automata emit W×8 partial match vectors indicating which components may match at that input position. The PMVs are each $|S|$×W bits long, where $|S|$ represents the number of distinct components the accelerator can simultaneously match (our implementations use $|S|$=64). The PMVs are then output to the intermediate match unit.

HARE adds W×$|C|$ automata units to process the output of the CCU. These automata store the transition tables for character class components constructed as described in Section A.4.1.3, emitting additional PMVs representing the potential character class matches to the intermediate match unit. The (8+$|C|$)×W bit-split automata operate in lock-step, consuming the same window of W characters, and emit (8+$|C|$)×W PMVs comprising $|S|$×W bits each. Figure A.4 (middle) illustrates the pattern automata. Each cycle, an automa-

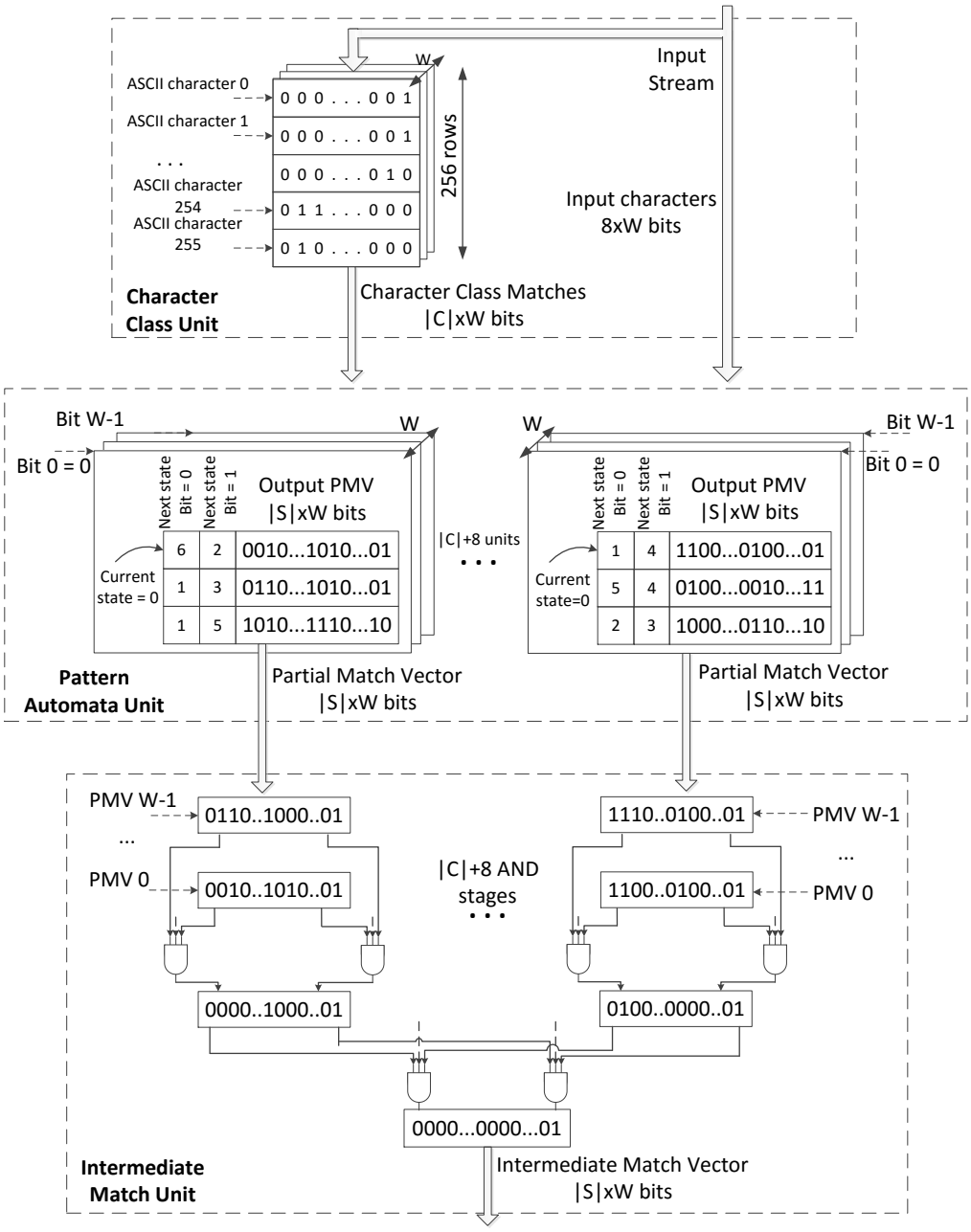


Figure A.4: **HARE's sub-units.** The character class unit compares the input characters to the pre-compiled character classes, pattern automata processes the bit streams to generate PMVs which are later reduced by IMU to compute component match.

ton consults the transition table stored in its local memory to compute the next state and corresponding PMV to emit, based on whether it consumed a zero or one. We refer readers to [195] for additional microarchitectural details of the pattern automata, which are unchanged in HARE.

A.4.2.3 Intermediate Match Unit

The intermediate match unit (IMU), as illustrated in Figure A.4 (bottom), combines partial matches produced by the W lanes of the pattern automata to produce a final match. The $W\times(8+|C|)$ PMVs are intersected (bitwise AND) to yield an intermediate match vector (IMV) of $|S|\times W$ bits. Each bit in the IMV indicates that a particular component has been matched by all automata at a specific location within the W-character window.

A.4.2.4 Counter-based reduction unit

The counter-based reduction unit (CRU): (1) determines if components appear in a sequence accepted by the regexp, (2) counts consecutive repetitions of a component, (3) resolves ambiguities among consecutive character classes that accept overlapping sets of characters, and (4) determines if the repetition counts for the components fall within the bounds set by the HARE compiler.

Our CRU design leverages the min-max counter-based algorithm proposed by Wang et al [204], which was designed to address character class ambiguities (3). Their algorithm consumes a single input character per step; we extend it to accept W-character windows per step and handle alternation operators and multi-character components. Throughout our discussion, we refer to Figure A.5, which depicts the unit and an example of a complex expression that includes several of the subtle issues the CRU must address.

The input to the CRU in each clock cycle is the intermediate match vector produced by the intermediate match unit. $IMV_{i,j}$ is a bit matrix comprising $|S|$ rows, one per component j in the regexp, and W columns, one per position i in the input window. $IMV_{i,j}$ is set if a component has been detected to end at that input position. A new IMV matrix arrives each clock cycle. Figure A.5 (top) illustrates arriving IMVs for $|S|$=5, W=4, and two clock cycles.

Internally, the CRU maintains three kinds of state, depicted in the remaining parts of Figure A.5.

Two matrices of counter-enable signals $MAX\_EN_{i,j}$ and $MIN\_EN_{i,j}$ account for the relationship between consecutive components. They track whether component j respectively may or must consume input character i to extend a match, based on the input consumed by preceding components. Loosely, if component $j-1$ matches at position $i-1$, or component j consumed character $i-1$, then these signals indicate that component j may consume character i. In our initial explanation, we assume that the precedence vector for component j includes only component $j-1$, and relax this restriction later.

The two matrices $\{MIN_{i,j}, MAX_{i,j}\}$ of counters indicate respectively the minimum number of repetitions that `must` be consumed and the maximum number of repetitions that `may` be consumed by component j to extend a match to position i. These repetition counts must be represented as a range, rather than an exact count, to handle adjacent character classes that accept overlapping character sets. In general, it is not known which input characters correspond to which components until a match is complete. Indeed, the CRU does not actually assign input characters to particular components as some regexps can match a given pattern in multiple ways. Rather, it determines if any match is possible.

Finally, a set of regexp match vectors $RMV_{i,j}$ track if the regexp matches up to and including component j at position i. $RMV_{i,j}$ is set if $MAX_{i,j}$ is above the lower repetition bound for component j and $MIN_{i,j}$ is below the upper bound, indicating that there is a feasible mapping of the input to components up to the ith character. A regexp matches at position i when the $RMV_{i,j}$ for the final component j is set.

Min-max matching for $W > 1$. We first describe our generalization of Wang's algorithm for min-max matching for $W > 1$, with reference to Algorithm 2. The min-max matching algorithm can match regexps containing a sequence of consecutive character classes when the character classes accept overlapping character sets. We describe the algorithm assuming precedence vectors form a strict chain (i.e., no `*`, `|`, `?` operators), and with only single-character components. We then remove these restrictions.

Consider a sequence of (potentially repeated) character classes $CC_1 ... CC_n$, such as

Algorithm 2 Algorithm for computing regexp match using counter-based reduction unit.

Input: Intermediate Match Vector **IMV**, number of components **|S|**, architecture width **W**, lower bounds **BL**, and upper bounds **BU**
Output: Regexp match vector *RMV*.

```
1:  MIN_EN = [[0 from 0 to |S|-1] from 0 to W-1]
2:  MAX_EN = [[0 from 0 to |S|-1] from 0 to W-1]
3:  MIN = [[0 from 0 to |S|-1] from 0 to W-1]
4:  MAX = [[0 from 0 to |S|-1] from 0 to W-1]
5:  for i = 1 to W-1 do
6:      for j = 1 to |S|-1 do
7:          MIN_EN[i][j] = RMV[i-1][j-1] || MIN[i-1][j] > 0
8:          MAX_EN[i][j] = RMV[i-1][j-1] || MAX[i-1][j] > 0
9:      end for
10: end for
11:
12: for i = 1 to W-1 do
13:     for j = 1 to |S|-1 do
14:         if MIN_EN[i][j] & IMV[i][j] then
15:             MIN[i][j] = RMV[i][j-1] ? MIN[i-1][j] + 1 : 0
16:         end if
17:     end for
18: end for
19:
20: for i = 1 to W-1 do
21:     for j = 1 to |S|-1 do
22:         if MAX_EN[i][j] & IMV[i][j] then
23:             MAX[i][j] = MAX[i-1][j] + 1
24:         end if
25:     end for
26: end for
27:
28: for i = 1 to W-1 do
29:     for j = 1 to |S| do
30:         RMV[i][j] = MAX[i][j] >= BL[i][j] & MIN[i][j] <= BU[i][j]
31:     end for
32: end for
```

`[a-d]{2,4}[abe]{2,3}`. This expression is challenging because some input texts can match the expression in multiple ways and it is generally impossible to assign input characters to specific components incrementally as the input is consumed. For example, the input `adbceb` can be matched by assigning `adbc` to CC_1 and `eb` to CC_2. However, a scheme that incrementally assigns characters might match `ad` to CC_1 and attempt to match `bce` to CC_2, at which point the match cannot be extended. The min-max algorithm resolves such ambiguous matches.

Initialization. (Lines 1-4). All counters, counter-enable, and *RMV* are initialized to zero, and the lower and upper bounds *BL* and *BU* for each component are initialized based on the bounds emitted by the HARE compiler. Each clock cycle, $IMV_{i,j}$ arrives from the intermediate match unit indicating components $0 <= j < |S|$ ending at positions $0 <= i < W$.

Determine counter-enables. (Lines 5-10). The counter-enable step captures the relationship between consecutive components and determines if the character at position i can potentially extend a match. More precisely, it determines if character at position i may potentially be consumed by component j based on whether the preceding input through $i-1$ matches the preceding regexp components up to (and possibly including) j. If $RMV_{i-1,j-1}$ is set, then component $j-1$ matches through position $i-1$, hence, character i may be the first occurrence of component j. Alternatively, if character at position $i-1$ was consumed by j, then i may be an additional repetition extending the match of component j. Note that the RMV for $j=-1$ is considered to be set at all positions in the input, meaning that first component $j=0$ may begin at any position.

Update minimum counts. (Lines 12-18). The minimum counts $MIN_{i,j}$ reflect the count of characters that must be consumed by component j because they cannot be consumed by the preceding component $j-1$. If $MIN\_EN_{i,j}$ is set, then character i may be consumed by j. If $IMV_{i,j}$ is set, then i belongs to the character class of component j. However, if character i may also be consumed by the preceding component $j-1$, as reflected by $RMV_{i,j-1}$, then it is not *necessary* for component j to consume the character and $MIN_{i,j}$ is reset, else it is incremented. The min counter, therefore, always reflects the fewest characters that can be accounted for by repetitions of component j.

Update maximum counts. (Lines 20-26) The max counters, on the other hand, reflect the largest number of characters that could be consumed by component j. As above, $MAX\_EN_{i,j}$ indicates if character i may be consumed by j, and $IMV_{i,j}$ indicates if the character matches component j. If both conditions hold, the maximum counter is incremented.

Update RMVs. (Lines 28-32). Once MIN and MAX are computed, RMV is computed as previously described; $RMV_{i,j}$ is true if MIN and MAX fall within BU and BL, respectively. A full regexp matches when $RMV_{i,j}$ for final component is set.

Example. Figure A.5 illustrates how the CRU processes `[ab][bc]+d?efc{2}` regular expression consisting of components `[ab]`, `[bc]`, `d`, `ef`, and `c`. The figure illustrates the

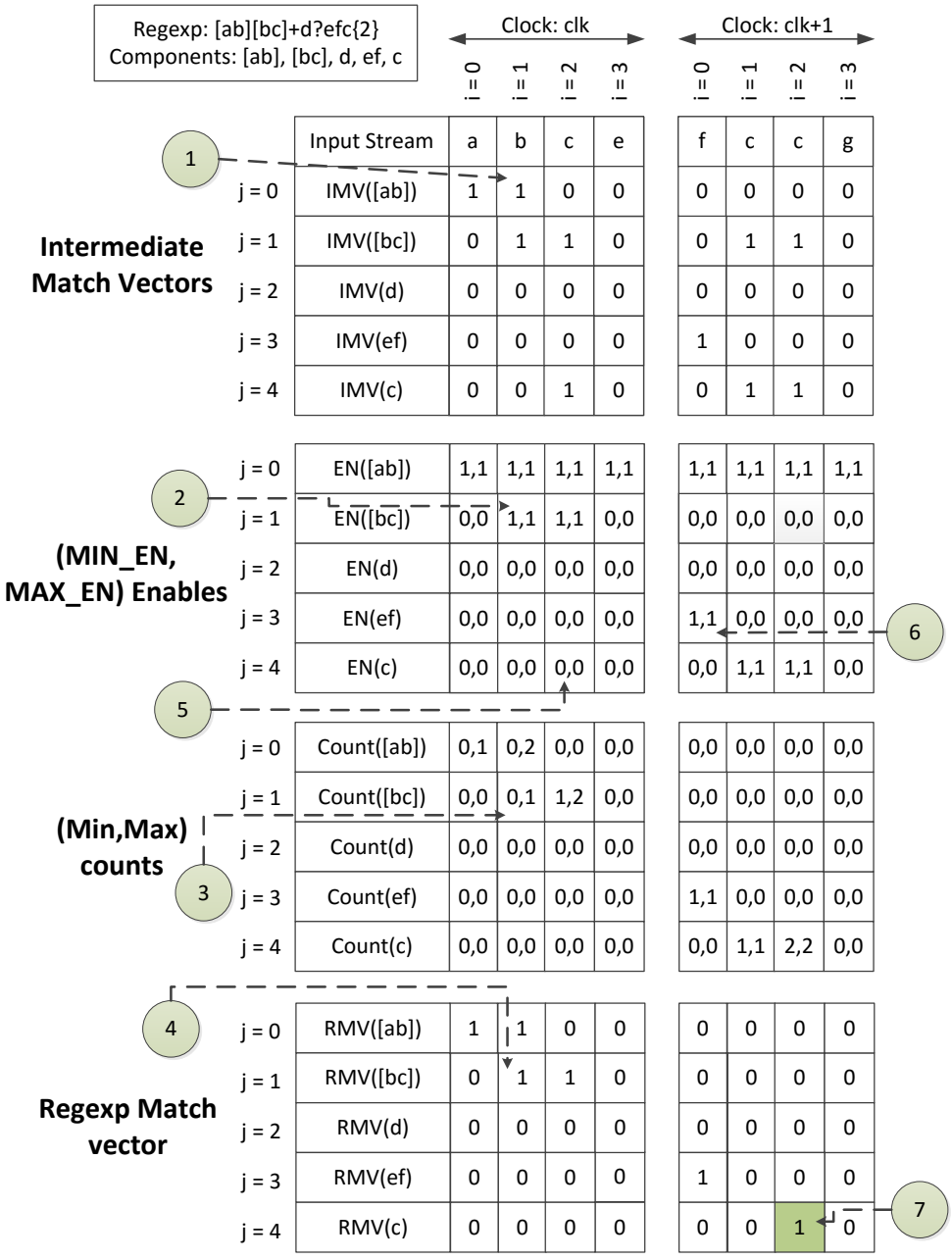


Figure A.5: **Counter-based reduction unit pipeline.** CRU combines the separate matches of the components generated by IMU. It maintains three states, namely counter enables, counters, and RMV to determine whether components of a regexp occur in a desired order.

matching process for the input string `abcefccg`. The figure shows IMVs for two clock cycles, indicating where each component has matched in the input. ① indicates where two different character classes, corresponding to components `[ab]` and `[bc]` can match input

character b at $i = 1$. Note that the counter-enables for component [ab] ($j = 0$) are always enabled and the minimum counter is always reset to zero, as a match of the regexp may begin at any point in the input. Component [ab] ($j = 0$) matches character a at $i = 0$ and increments $MAX_{0,0}$ to 1. Hence, $RMV_{0,0}$ is set, since $MIN_{0,0}$ is below upper bound BU_0 = 1 and $MAX_{0,0}$ equals lower bound BL_0 = 1.

The second character b is then processed and the counters $MIN_{1,1}$ and $MAX_{1,1}$ are enabled, since $RMV_{0,0}$ is set, enabling $MIN\_EN_{1,1}$ and $MAX\_EN_{1,1}$, indicated by ②. Furthermore, the counters $MAX_{1,0}$ and $MAX_{1,1}$ are both incremented as $IMV_{1,0}$ and $IMV_{1,1}$ are set. In other words, b can be consumed by either of the first two components.

Note that $MIN_{1,1}$ is not incremented, since b may be consumed by component $j = 0$, as indicated by ③. Since both counters for $j = 1$ satisfy the component's repetition bounds, $RMV_{1,1}$ is set, indicated by ④. When the third character is consumed, the counters $MIN_{2,4}$ and $MAX_{2,4}$ are not enabled as the preceding components did not match, indicated by ⑤.

Handling optional/alternative components. We next generalize the min-max algorithm to handle optional and alternative components. Recall that HARE's compiler emits, for each component, a precedence vector indicating the components that may precede it (see Section A.4.1.2). Rather than calculate *MIN_EN* and *MAX_EN* based solely on the immediately preceding component $j-1$, they are calculated as the logical-OR of all components in j's precedence vector. In words, component j may consume character i if any of its possible predecessors can consume character $i-1$.

Multi-character components. As originally proposed, Wang's min-max algorithm assumed the input would be consumed a single character at a time and had no need to handle multi-character components. Because PMV bits are a limited resource, it is critical for HARE to match multi-character sub-strings with a single component where possible, since HAWK provides that capability. We support multi-character components by storing the length of each component in a vector LEN_j. When indexing $RMV_{i,j}$ for a multi-character component, we right-shift the vector (in i) by $LEN_j - 1$ positions. That is, we ignore the

Processor	Dual socket Intel E5645 12 threads @ 2.40 GHz
Caches	192 KB L1, 1 MB L2, 12 MB L3
Memory capacity	128 GB
Memory type	Dual-channel DDR3-1333
Maximum memory bandwidth	21.3 GB/s

Table A.1: **Server specifications.** Server configuration used for running the software baselines.

columns of $RMV_{i,j}$ that fall within component j, and instead reference the last character of the preceding component.

We complete the preceding example to illustrate these extensions. In Figure A.5, component `ef` may be preceded by either `[bc]` or `d`. Hence, in the second clock cycle, when computing $MIN\_EN_{0,3}$ and $MAX\_EN_{0,3}$ for component `ef` as indicated by ⑥, both possible predecessors `[bc]` ($j = 1$) and `d` ($j = 2$) are considered. Moreover, since the length of `ef` is two, count-enables, *MIN*, and *MAX* are calculated by referring to $RMV_{2,j}$ rather than $RMV_{3,j}$. Ultimately as illustrated by ⑦, the expression is matched when $RMV_{2,4} = 1$ in the second cycle (indicated by the green cell), when the *MIN* and *MAX* counts for component `c` ($j = 4$) match its bound of exactly 2 repetitions.

A.5 Evaluation

We evaluate two implementations of HARE, an RTL-level design targeting an ASIC process and a scaled-down FPGA prototype to validate feasibility and correctness. We study a suite of over 5500 real-world and synthetically generated regexps. We first contrast HARE against conventional software solutions and then evaluate area and power of the ASIC implementation of HARE for different processing widths.

A.5.1 Experimental Setup

We compare HARE's performance against software baselines on an Intel Xeon class server with the specifications listed in Table A.1. We select three software baselines: *grep* version 2.10, the Lucene search-engine *lucene* [87] version 5.5.0, and the Postgres rela-

tional database *postgres* [193] version 9.5.1.

We generate input text using Becchi's traffic generator [32]. The traffic generator is parameterized by the probability of a match pM; that is, the probability that each character it emits extends a match. For instance, for a pM=0.75, the traffic generator extends the preceding match with probability 0.75 and emits a random character with probability 0.25.

We implement the HARE ASIC design in Verilog and synthesize it for varying widths W of 2, 4, 8, 16, and 32. In our ASIC implementation, we configure HARE to match at most 64 components in a single pass. We target a commercial 45nm standard cell library operating at 1.1V and clock the design at 1GHz. Although this library is two generations behind currently shipping technology, it is the latest commercial process to which we have access. We synthesize the complete design using Synopsys Design-Ware IP Suite and report the timing, area and power estimates from Design Compiler.

To validate feasibility and correctness, we implement a scaled-down design on the Altera Arria V SoC development platform. Due to FPGA limitations, we implement a 4-wide HARE design. We use the FPGA's block RAMs to store pattern automata transition tables and PMVs; the available block RAMs limit the scale of the HARE design. Due to the overheads of global wiring to far-flung block RAMs, we limit clock frequency to 100MHz. Our software compiler generates pattern automata transition tables, PMVs, and reducer unit configurations, which we load into the block RAMs.

Because of the limited on-board memory capacity and poor bandwidth to host system memory available on our platform, we synthetically generate input text on the fly on the FPGA to test the functionality of the HARE FPGA. We tested 300 synthetic and hand-written regular expressions that stress various regexp features. We generate random text using linear feedback shift registers and then use a table-driven approach to periodically insert pre-generated matches into the synthetic text and confirm that all matches are found.

Workload	Regexps	Supported	Comp.	Comp. Len
dotstar0.3	300	99.0%	3.8	14.6
dotstar0.6	300	99.0%	4.4	12.5
dotstar0.9	300	99.0%	4.9	9.9
exact-match	300	99.6%	2.1	23.4
range05	300	99.6%	2.9	18.9
range1	300	99.3%	3.4	15.2
snort	1053	85.6%	4.6	5.5
RegExLib	2673	56.4%	12.3	1.7

Table A.2: **Characteristics of regular expression workloads.** Percentage of regular expression workloads supported by HARE.

A.5.2 Regexp Workloads

We evaluate the capability and performance of HARE using a combination of human-written and automatically generated regexps from a variety of sources. Our human-written regexps are drawn from the online repository RegExLib [9] and the Snort [11] network intrusion detection library. Moreover, we derive synthetically generated regexps from the libraries provided by Becchi [32]. Table A.2 shows the characteristics of each workload, indicating the number of expressions, the fraction HARE can support, the average number of components, and the average length of components. Several regexps on RegExLib are syntactically incorrect and we therefore discard them. HARE can support up to 99% of regexps in the workloads proposed by Becchi and around 86% of the regexps in the Snort library. In addition, despite the complexity of many of the expressions on RegExLib (some involving more than 50 components), HARE can support over 56% of them. Moreover, of the regexps we do not support, 83% of the Snort regexps and 45% of the RegExLib regexps contain non-regular operators, such as back references and look-ahead; when allowing these operators the matching problem is NP-complete [17]. The remaining unsupported expressions either contain nested repetitions or apply repetition operator to multi-character sub-strings. The HARE compiler detects unsupported regexps, reports a detailed error, and does not produce false negatives. Table A.2 was derived from regexps flagged as unsupported by the compiler.

HARE resource constraints. A HARE hardware implementation imposes two funda-

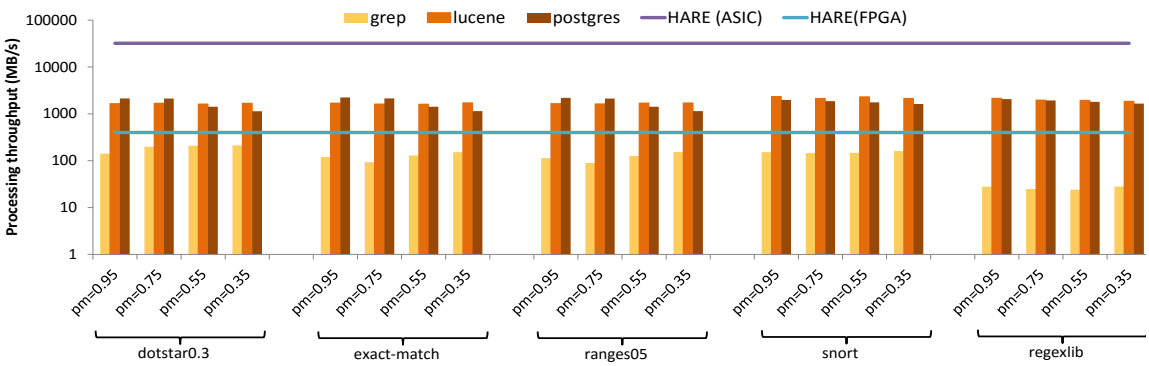


Figure A.6: **Single regexp performance comparison.** We contrast HARE's fixed 32GB/s ASIC and 400 MB/s FPGA performance against software solutions. ASIC implementation of HARE performs two order of magnitude better than the software solutions.

mental resource constraints: the number of supported character classes ($|C|$), which is constrained in the CCU and by the number of pattern automata, and the number of components in a regular expression ($|S|$), which is restricted by the number of PMV and IMV bits. Regular expressions that exceed these constraints cannot be processed in a single pass without additional software support.

Other implementation constraints, such as the maximum component length (equal to W), or the maximum precedence vector length (four per component) are automatically handled by the HARE compiler by splitting a component that exceeds the constraints into multiple components. All the workloads proposed by Becchi lie under these constraints. For Snort and RegExLib, the maximum precedence lengths of 9 and 59, respectively, exceed the hardware limit. The HARE compiler splits these components, increasing PMV utilization.

A.5.3 Performance - Scanning Single Regexp

We first contrast HARE's ASIC and FPGA performance with software baselines while scanning an input text for a single regular expression. We generate several 1GB inputs while varying pM. To exclude any time the software solutions spend materializing output, we execute queries that count the number of matches and report the count. We randomly

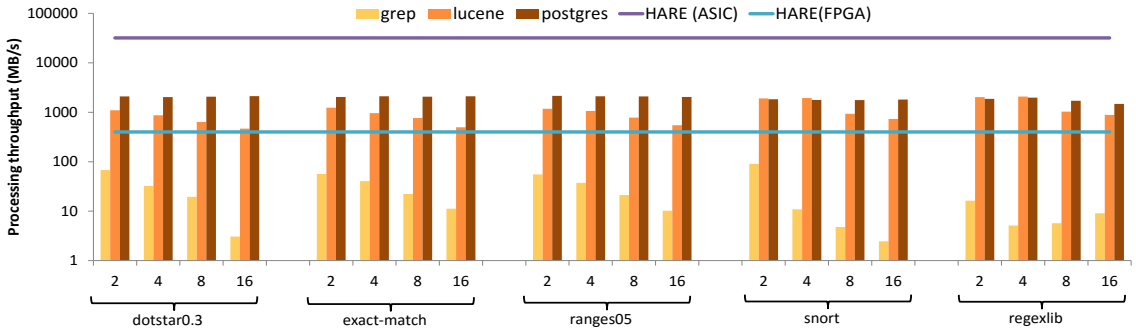


Figure A.7: **Multiple regexp performance.** The software solutions generally slow down as they search for more expressions concurrently. HARE's performance is insensitive to the number of expressions, provided the aggregate resource requirements of the expressions fit within HARE's implementation limits.

select 100 regexps from each of the eight workloads for performance tests, and report average performance over these 100 runs. In the interest of space, we report results for only three of Becchi's six benchmarks, as the remaining benchmarks show similar trends in the performance. For Lucene, we first create an inverted index of the input and do not include index creation time in the reported performance results. Similarly for Postgres, we first load the input into the database, excluding the load time from the results. We report their throughput by dividing the query execution time by the number of characters in the input text.

Figure A.6 compares the throughput of grep, Lucene, and Postgres to the fixed scan rates of the HARE designs. The software systems are configured to use all 12 hardware threads of the Xeon E5645. The 32GB/s constant processing throughput of ASIC HARE is an order of magnitude higher than the software solutions. While HARE can saturate memory bandwidth, none of the other solutions come close. Even the scaled-down FPGA HARE implementation outperforms grep, which can only process at a maximum throughput of 300MB/s. Lucene and Postgres perform consistently above 1GB/s but fall considerably short of HARE's processing throughput.

A.5.4 Performance - Scanning Multiple Regexps

Figure A.7 compares the performance of HARE and the software systems when scanning for multiple regexps concurrently (by separating a list of patterns with alternation operators). We randomly choose regexps from the workloads and vary their number from two to 16. We concatenate portions of the input text produced for each regexp (with pM=0.75) to ensure that all occur within the combined 1GB input text.

As expected, as the software systems search for more regexps, their throughput decreases. The performance of grep drops precipitously to 5MB/s when processing 16 regexps simultaneously; in practice, it is often better to perform multi-regexp searches consecutively rather than concurrently with grep. Postgres and Lucene still maintain a processing throughput of above 1GB/s even while scanning for 16 regexps. Again, note that we do not include the time Lucene and Postgres take to precompute indexes and load the input. On the contrary, HARE can still process the regexps simultaneously at constant throughput of 32GB/s.

A.5.5 ASIC Power and Area

We report the area and power requirement of ASIC HARE and its sub-units when synthesized for 45nm technology. We synthesize the HARE design for widths varying from two to 32 characters. As per our goal, we pipeline each design to meet a 1GHz clock frequency.

As shown in Figure A.8 (top), we find that the area and power requirement of HARE is dominated by the storage for state transition tables and PMVs in the pattern automata unit. Moreover, the contribution of pattern automata units to the total HARE area and power increases as the width of HARE grows, because the storage required for the bit-split machines grows quadratically with the accelerator width.

In Figure A.8 (bottom), we compare the total area and power of HARE to an Intel W5510 processor. We select this processor for comparison because it is implemented in

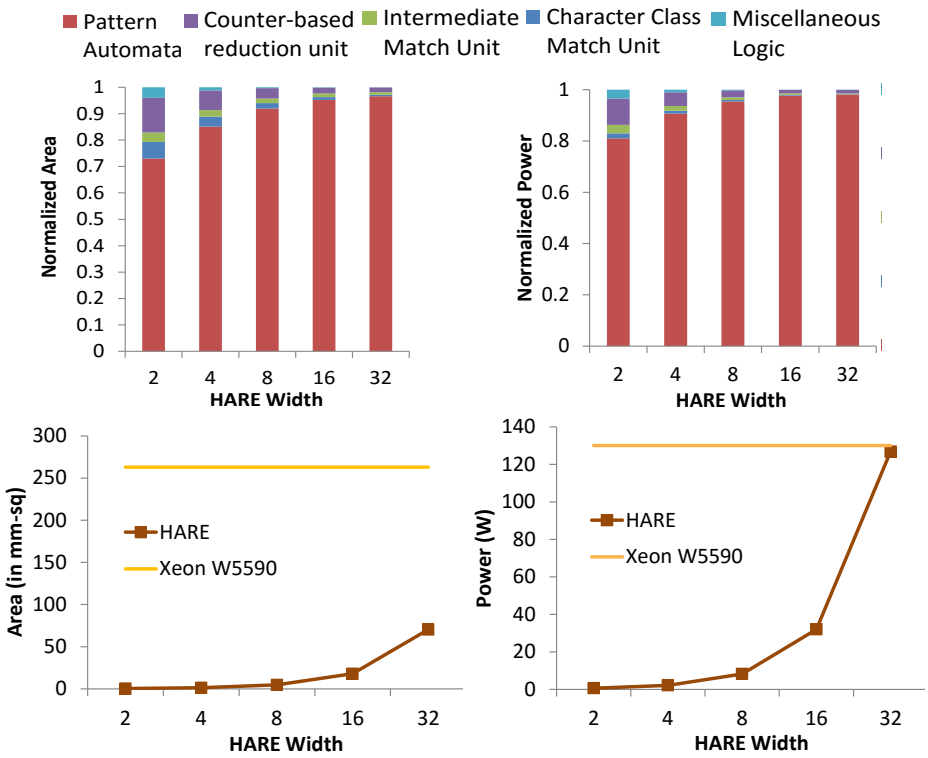


Figure A.8: **ASIC HARE area and power.** Pattern automata dominates area and power consumption of HARE due to the storage for bit-split machines. Overall, all the implementations of HARE consumes lower power than Xeon W5590.

the same technology generation as our ASIC process. We see that the 8-wide and 16-wide instances of HARE require just 1.8% and 6.8% of the area of a W5510 chip. Moreover, the 8-wide and 16-wide HARE consumes only 6.3% and 24.6% of the power of our baseline processor. Even the 32-wide instance of HARE can be implemented in 26.7% of the area while consuming lower power than the W5510. Note that the 45nm technology used in our evaluation is two generations behind the state of the art. As the area and power requirements scale with technology, HARE would occupy a much smaller fraction of chip area relative to current state-of-the-art processors.

A.5.6 FPGA Prototype

We validate the HARE design by implementing a scaled-down version on the Altera Arria V FPGA. We implement a 4-wide instance of HARE provisioning 64 components

at 100MHz. The scaled-down HARE design uses 12% of the logic and 14% of the block memory capacity of the FPGA. Since we generate input text synthetically on the FPGA, HARE scans the input at a constant throughput of 400MB/s. Even when scaled down, HARE still scans the input text 1.9x faster than grep when scanning for a single regexp and this gap widens when processing multiple regexps.

A.6 Related Work

Parallel regexp matching. Several works seek to parallelize matching by running the regexp automaton separately on separate substrings of the input and combining the results obtained on each part of the text [89]. Since each substring may start at an arbitrary point in the input, the automaton must consider all states as start states, which is problematic for large automata. PaREM [151] tries to minimize the number of states on which the automaton runs by exploiting the structure of automata that have sparse transition tables. Mytkowicz et al. [159] further optimize this concept by representing transitions as matrices and combining multiple automata executions using matrix multiplication. They also use SIMD to perform multiple lookups for different sections of the input text at once.

Parabix [134] introduces the idea of processing character bits in parallel and combining the results using Boolean operations. This design allows Parabix to exploit SIMD instructions. Cameron [46] extends the design of Parabix to directly handle non-determinism and provides a tool chain to generate marker streams, the bit-stream that mark the matches in the input text. For different regexp operations, the tool manipulates the marker stream to update the regexp matches.

The Unified Automata Processor (UAP) [67] implements specialized software and hardware support for different automata models e.g., DFAs, NFAs, and A-DFAs. This framework proposes new instructions to configure the transition states, perform finite automata transitions and synchronize the operations of parallel execution lanes. HARE's approach of using a stall-free scan pipeline with parallel bit-split automata and min-max matching bears

little similarity to UAP's implementation approach. The UAP relies on parallel processing of multiple input streams to achieve its peak bandwidth of 295 Gbit/sec, but achieves at most a 1.13 GC/s scan rate per stream. In contrast, HARE saturates a memory bandwidth of 32 GC/s (256Gbit/s) when scanning a single input stream.

ASIC and FPGA based solutions. Micron's Automata Processor [61] implements NFAs at the architecture level. Transition tables are stored as 256-bit vectors, which are then connected over a routing matrix. Counting and boolean operations are then used to count the matches of sub-expressions and combine sub-expression results. The processor can consume input strings at a line rate of 1Gbit/sec per chip.

IBM PowerEN SoC integrates RegX, an accelerator for regular expressions [144]. RegX splits regexps into multiple sub-patterns, implements separate DFAs and configures the transition tables using programmable state machines called B-FSMs [198], and finally combines the sub-results in the local result processor. RegX runs at a frequency of 2.3 GHz and achieves a peak scan rate of 9.2Gbit/sec.

A Micron Automata Processor processing 1 character/cycle consumes around 4W [61], while the IBM PowerEn RegX accelerator consumes around 2W [69]. In comparison, a 1-wide HARE implementation consumes less than 1W.

Helios [5] is another accelerator that processes regexps for network packet inspection at line rate. In addition, several works [189, 215, 156, 36, 214] propose mechanisms to match regexps on FPGAs. They focus on building a finite automaton and encode it in the logic of the FPGA. HARE's 32GB/sec (256Gbit/sec) scan rate is much more ambitious than these prior ASIC or FPGA designs.

A.7 Conclusion

Rapid processing of high-velocity text data is necessary for many technical and business applications. Conventional regular expression matching mechanisms do not come close to exploiting the full capacity of modern memory bandwidth. We showed that our

HARE accelerator can process data at a constant rate of 32 GB/s and that HARE is often better than state-of-the-art software solutions for regular expression matching. We evaluate HARE through a 1GHz ASIC RTL implementation processing 32 characters of an input text per clock cycle. Our ASIC implementation can thus match modern memory bandwidth of 32GB/s, outperforming software solutions by two orders of magnitude. We also demonstrate a scaled-down FPGA prototype processing 4 characters per clock cycle at a frequency of 100MHz (400 MB/s). Even at this reduced rate, the prototype outperforms grep by 1.5-20× on commonly used regular expressions.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Aerospike. http://www.aerospike.com/. [Online; accessed 17-Jun-2017].

[2] Available first on Google Cloud: Intel Optane DC Persistent Memory. https://tinyurl.com/gcp-release.

[3] Boost. https://www.boost.org/.

[4] C++ bindings for libpmemobj - synchronization primitives. http://pmem.io/2016/05/31/cpp-08.html.

[5] Helios Regular Expression Processor. http://titanicsystems.com/Products/Regular-eXpression-Processor-RXP.

[6] Intel and Micron Produce Breakthrough Memory Technology. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.

[7] INTEL OPTANE DC PERSISTENT MEMORY. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html.

[8] pmem.io: Persistent memory programming. https://pmem.io/pmdk/.

[9] Regular expression library. http://regexlib.com/.

[10] Reimagining the Data Center Memory and Storage Hierarchy. https://newsroom.intel.com/editorials/re-architecting-data-center-memory-storage-hierarchy.

[11] Snort. http://snort.org/.

[12] Structuring Unstructured Data. www.forbes.com/2007/04/04/teradata-solution-software-biz-logistics-cx_rm_0405data.html/.

[13] Understand and deploy persistent memory. https://docs.microsoft.com/en-us/windows-server/storage/storage-spaces/deploy-pmem.

[14] Memcached - a distributed memory object caching system, 2012.

[15] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[16] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[17] Alfred V. Aho. Handbook of theoretical computer science (vol. a). chapter Algorithms for Finding Patterns in Strings, pages 255–300. MIT Press, Cambridge, MA, USA, 1990.

[18] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6), 1975.

[19] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 62–77, New York, NY, USA, 2018. ACM.

[20] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. Sap hana adoption of non-volatile memory. *Proc. VLDB Endow.*, 10(12):1754–1765, August 2017.

[21] ARM. Embedded trace macrocell, 2011. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0014q/IHI0014Q_etm_architecture_spec.pdf`.

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

[23] ARM. Arm architecture reference manual, 2017. `https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf`.

[24] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 707–722, New York, NY, USA, 2015. ACM.

[25] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proc. VLDB Endow.*, 10(4):337–348, November 2016.

[26] Arvind Arvind and Jan-Willem Maessen. Memory model = instruction reordering + store atomicity. *SIGARCH Comput. Archit. News*, 34(2):29–40, May 2006.

[27] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 2009.

[28] Manu Awasthi, Manjunath Shevgoor, Kshitij Sudan, Bipin Rajendran, and Rajeev Balasubramonian. Efficient scrub mechanisms for error-prone emerging memories. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2012.

[29] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 211–224, Berkeley, CA, USA, 2011. USENIX Association.

[30] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):1968–1980, 2016.

[31] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 55–66, New York, NY, USA, 2011. ACM.

[32] M. Becchi, M. Franklin, and P. Crowley. A workload for evaluating deep packet inspection architectures. In *IEEE International Symposium on Workload Characterization*, 2008.

[33] N. Beckmann and D. Sanchez. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–75, Feb 2015.

[34] 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, December 2012.

[35] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[36] J. Bispo, I. Sourdis, J. M. P. Cardoso, and S. Vassiliadis. Regular expression matching for reconfigurable packet inspection. In *IEEE International Conference on Field Programmable Technology*, 2006.

[37] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Valor: Efficient, software-only region conflict exceptions. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 241–259, New York, NY, USA, 2015. ACM.

[38] Daniel Bittman, Mathew Gray, Justin Raizes, Sinjoni Mukhopadhyay, Matt Bryson, Peter Alvaro, Darrell D. E. Long, and Ethan L. Miller. Designing data structures to minimize bit flips on nvm. In *The 7th IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, August 2018.

[39] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, 2017. USENIX Association.

[40] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[41] Colin Blundell, Milo MK Martin, and Thomas F Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 233–244. ACM, 2009.

[42] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[43] Hans-J. Boehm and Dhruva R. Chakrabarti. Persistence programming models for non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 55–67, New York, NY, USA, 2016. ACM.

[44] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[45] Robert D. Cameron and Dan Lin. Architectural support for swar text processing with parallel bit streams: The inductive doubling principle. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[46] Robert D. Cameron, Thomas C. Shermer, Arrvindh Shriraman, Kenneth S. Herdy, Dan Lin, Benjamin R. Hull, and Meng Lin. Bitwise data parallelism in regular expression matching. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, 2014.

[47] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: leveraging locks for non-volatile memory consistency. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2014.

[48] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Vaglis. Rewind: Recovery write-ahead system for in-memory non-volatile data structures. *Proceedings of the VLDB Endowment*, 8(5), 2015.

[49] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. NVMOVE: Helping programmers move to byte-based persistence. In *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)*, Savannah, GA, November 2016. USENIX Association.

[50] I-C. K. Chen, C-C. Lee, and T. N. Mudge. Instruction prefetching using branch prediction information. In *Proc. of the International Conference on Computer Design*, 1997.

[51] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher M. Aycock, Gurushankar Rajamani, and David Lowell. The rio file cache: Surviving operating system crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[52] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[53] Sangyeun Cho and Hyunjin Lee. Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance. In *Proceedings of the International Symposium on Microarchitecture*, 2009.

[54] 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 *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[55] Alexei Colin and Brandon Lucia. Chain: Tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 514–530, New York, NY, USA, 2016. ACM.

[56] 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 *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[57] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[58] Zhaoxia Deng, Lunkai Zhang, Nikita Mishra, Henry Hoffmann, and Frederic T. Chong. Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in nvms. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 232–244, New York, NY, USA, 2017. ACM.

[59] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. Pdram: A hybrid pram and dram main memory system. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, 2009.

[60] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[61] Paul Dlugosch, Dean Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[62] Kshitij Doshi, Ellis Giles, and Peter Varman. Atomic persistence for scm with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 77–89. IEEE, 2016.

[63] Paul J Drongowski. Instruction-based sampling: A new performance analysis technique for amd family 10h processors. *Advanced Micro Devices*, 2007. `http://developer.amd.com/wordpress/media/2012/10/AMD_IBS_paper_EN.pdf`.

[64] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems*, 2014.

[65] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 15:1–15:16, New York, NY, USA, 2016. ACM.

[66] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. Ifrit: Interference-free regions for dynamic data-race detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 467–484, New York, NY, USA, 2012. ACM.

[67] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. Fast support for unstructured data processing: The unified automata processor. In *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[68] Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. Increasing pcm main memory lifetime. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10.

[69] Hubertus Franke, Charlie Johnson, and Jeff Brown. The ibm power edge of network processor, 2010.

[70] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, June 2005.

[71] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen-mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 183–193, New York, NY, USA, 1994. ACM.

[72] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Badgertrap: A tool to instrument x86-64 tlb misses. *SIGARCH Comput. Archit. News*.

[73] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *In Proceedings of the 1991 International Conference on Parallel Processing*, 1991.

[74] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for c programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 47–58, New York, NY, USA, 2001. ACM.

[75] E. R. Giles, K. Doshi, and P. Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, May 2015.

[76] Ellis Giles, Kshitij Doshi, and Peter Varman. Continuous checkpointing of htm transactions in nvm. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, pages 70–81, New York, NY, USA, 2017. ACM.

[77] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Failure-atomic synchronization-free regions, 2018. `http://nvmw.ucsd.edu/nvmw18-program/unzip/current/nvmw2018-final42.pdf`.

[78] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 46–61, New York, NY, USA, 2018. ACM.

[79] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. Hare: Hardware accelerator for regular expressions. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 44:1–44:12, Piscataway, NJ, USA, 2016. IEEE Press.

[80] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Strand persistency, 2019.

http://nvmw.ucsd.edu/nvmw2019-program/unzip/current/nvmw2019-final35.pdf.

[81] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Aasheesh Kolli, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Software wear management for persistent memories. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 45–63, Boston, MA, February 2019. USENIX Association.

[82] Dibakar Gope, Arkaprava Basu, Sooraj Puthoor, and Mitesh Meswani. A case for scoped persist barriers in gpus. In *Proceedings of the 11th Workshop on General Purpose GPUs*, GPGPU-11, pages 2–12, New York, NY, USA, 2018. ACM.

[83] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[84] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software persistent memory. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 319–331, Boston, MA, 2012. USENIX.

[85] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 202–215, New York, NY, USA, 2016. ACM.

[86] SAP HANA. Bringing persistent memory technology to sap hana: Opportunities and challenges, 2016. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2016/20160810_FR21_Caklovic.pdf.

[87] Erik Hatcher and Otis Gospodnetic. Lucene in action. 2004.

[88] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12):1612–1630, December 1989.

[89] Jan Holub and Stanislav Štekr. On parallel implementations of deterministic finite automata. In *Proceedings of the 14th International Conference on Implementation and Application of Automata*, 2009.

[90] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley Longman Publishing Co., Inc., 1969.

[91] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 468–482, New York, NY, USA, 2017. ACM.

[92] Nan Hua, Haoyu Song, and TV Lakshman. Variable-stride multi-pattern matching for scalable deep packet inspection. In *INFOCOM 2009, IEEE*, 2009.

[93] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan. Unified address translation for memory-mapped ssds with flashmap. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 580–591, June 2015.

[94] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, Santa Clara, CA, 2017. USENIX Association.

[95] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. Nvram-aware logging in transaction systems. In *Proceedings of the VLDB Endowment*, volume 8, pages 389–400, 2014.

[96] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, 2018. USENIX Association.

[97] Intel. Intel microarchitecture codename nehalem performance monitoring unit programming guide (nehalem core pmu). https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf.

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

[99] Intel. Persistent memory programming, 2015. http://pmem.io/.

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

[101] Intel. Intel 64 and ia-32 architectures software developer's manual, 2018. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[102] Intel and Micron. Intel and micron produce breakthrough memory technology, 2015. http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology.

[103] Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically replicated memory: Building reliable systems from nanoscale resistive memories. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010.

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

[105] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. *Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model*, pages 313–327. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[106] Joseph Izraelevitz and Michael L. Scott. Brief announcement: A generic construction for nonblocking dual containers. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 53–55, New York, NY, USA, 2014. ACM.

[107] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[108] Yongsoo Joo, Dimin Niu, Xiangyu Dong, Guangyu Sun, Naehyuck Chang, and Yuan Xie. Energy- and endurance-aware design of phase change memory caches. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, 2010.

[109] Hammurabi Mendes Joseph Izraelevitz and Michael L. Scott. Linearization of persistent memory objects under a full-system-crash failure model. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, 2016.

[110] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. Dfs: A file system for virtualized flash storage. *Trans. Storage*, 6(3):14:1–14:25, September 2010.

[111] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. Atom: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372, Feb 2017.

[112] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient persist barriers for multicores. In *Proceedings of the international symposium on Microarchitecture*, 2015.

[113] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Dhtm: Durable hardware transactional memory. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 452–465, Piscataway, NJ, USA, 2018. IEEE Press.

[114] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for nand flash memory. In *Proceedings of the 6th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '06, pages 161–170, New York, NY, USA, 2006. ACM.

[115] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, 2018. USENIX Association.

[116] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 521–534, New York, NY, USA, 2017. ACM.

[117] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *2007 25th International Conference on Computer Design*, pages 245–250, Oct 2007.

[118] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 297–312, New York, NY, USA, 2018. ACM.

[119] Jesung Kim, Jong Min Kim, S. H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.

[120] Hideaki Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[121] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, W. Wang, P. M. Chen, S. Narayanasamy, and T. F. Wenisch. Language support for memory persistency. *IEEE Micro*, 39(3):94–102, May 2019.

[122] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P.M. Chen, and T.F. Wenisch. Delegated persist ordering. In *Proceedings of the 49th International Symposium on Microarchitecture*, 2016.

[123] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 481–493, New York, NY, USA, 2017. ACM.

[124] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Tarp: Translating acquire-release persistency, 2017. `http://nvmw.eng.ucsd.edu/2017/assets/abstracts/1`.

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

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

[127] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[128] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[129] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[130] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[131] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.

[132] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 339–353, Santa Clara, CA, 2016. USENIX Association.

[133] H. L. Li, C. L. Yang, and H. W. Tseng. Energy-aware flash memory management in virtual memory system. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(8):952–964, Aug 2008.

[134] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, 2012.

[135] Lin Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 112–122, June 2005.

[136] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343, New York, NY, USA, 2017. ACM.

[137] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. Nvm duet: unified working memory and persistent store architecture. In *Proceedings of the international conference on Architectural Support for Programming Languages an Operating Systems*, 2014.

[138] S. Liu, A. Kolli, J. Ren, and S. Khan. Crash consistency in encrypted non-volatile main memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 310–323, Feb 2018.

[139] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. Janus: Optimizing memory and storage support for non-volatile memory systems. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 143–156, New York, NY, USA, 2019. ACM.

[140] David E. Lowell and Peter M. Chen. Free transactions with rio vista. In *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

[141] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-ordering consistency for persistent memory. In *Proceedings of the 32nd IEEE International Conference on Computer Design*, 2014.

[142] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, 2017. USENIX Association.

[143] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 210–221, New York, NY, USA, 2010. ACM.

[144] Jan Van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. Designing a programmable wire-speed regular-expression matching accelerator. In *Proceedings of the 45th Annual International Symposium on Microarchitecture*, 2012.

[145] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti. Laser: Light, accurate sharing detection and repair. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–273, March 2016.

[146] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. Armor: Defending against memory consistency model mismatches in heterogeneous architectures. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, 2015.

[147] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent execution without checkpoints. *Proc. ACM Program. Lang.*, 1(OOPSLA):96:1–96:30, October 2017.

[148] Stefan Manegold, Martin L. Kersten, and Peter Boncz. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *Proc. VLDB Endow.*, 2009.

[149] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. Drfx: A simple and efficient memory model for concurrent programming languages. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 351–362, New York, NY, USA, 2010. ACM.

[150] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. ACM.

[151] Suejb Memeti and Sabri Pllana. Parem: A novel approach for parallel regular expression matching. *CoRR*, 2014.

[152] Paul Menage. Memory resource controller, 2016. `http://elixir.free-electrons.com/linux/latest/source/Documentation/cgroup-v1/memory.txt`.

[153] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. Enabling efficient and scalable hybrid memories using fine-granularity dram cache management. *IEEE Comput. Archit. Lett.*

[154] Azalia Mirhoseini, Ebrahim M Songhori, and Farinaz Koushanfar. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered asics. In *2013 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 216–224, March 2013.

[155] Amirhossein Mirhosseini, Aditya Agrawal, and Josep Torrellas. Survive: Pointer-based in-dram incremental checkpointing for low-cost data persistence and rollback-recovery. *IEEE Computer Architecture Letters*, 16(2):153–157, July 2017.

[156] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling pcre to fpga for accelerating snort ids. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, 2007.

[157] 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. *ACM Transactions on Database Systems*, 17(1), 1992.

[158] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, pages 1:1–1:17, New York, NY, USA, 2013. ACM.

[159] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[160] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 135–148, New York, NY, USA, 2017. ACM.

[161] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[162] Iyswarya Narayanan, Aishwarya Ganesan, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Anand Sivasubramaniam. Getting more performance with polymorphism from emerging memory technologies. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, pages 8–20, New York, NY, USA, 2019. ACM.

[163] Faisal Nawab, Dhruva Chakrabarti, Terence Kelly, and Charles B. Morey III. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. Technical Report HPL-2014-70, Hewlett-Packard, December 2014.

[164] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. Telecom application transaction processing benchmark, 2011. `http://tatpbenchmark.sourceforge.net/`.

[165] T. Nguyen and D. Wentzlaff. Picl: A software-transparent, persistent cache log for nonvolatile main memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 507–519, Oct 2018.

[166] M. A. Ogleari, E. L. Miller, and J. Zhao. Steal but no force: Efficient hardware undo+redo logging for persistent memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 336–349, Feb 2018.

[167] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. Sofort: A hybrid scm-dram storage engine for fast data recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, 2014.

[168] Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. ...and region serializability for all. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*, 2013.

[169] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 471–484, New York, NY, USA, 2014. ACM.

[170] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: X86-tso. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[172] S. Phadke and S. Narayanasamy. Mlp aware heterogeneous memory system. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[173] Moinuddin K. Qureshi, Michele M. Franchescini, Vijayalakshmi Srinivasan, Luis A. Lastras, Bulent Abali, and John Karidis. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the International Symposium on Microarchitecture*, 2009.

[174] Moinuddin K. Qureshi, Andre Seznec, Luis A. Lastras, and Michele M. Franchescini. Practical and secure pcm systems by online detection of malicious write streams. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, 2011.

[175] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.

[176] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, 2011.

[177] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, 2016.

[178] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 672–685. ACM, 2015.

[179] Matthew Eastwood Richard L. Villars, Carl W. Olofson. *Big Data: What It Is and Why You Should Care*. IDC, 2011.

[180] V. Salapura, T. Karkhanis, P. Nagpurkar, and J. Moreira. Accelerating business analytics applications. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, 2012.

[181] D. L. Schuff, B. S. Parsons, and V. S. Pai. Multicore-aware reuse distance analysis. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010.

[182] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 53–64, New York, NY, USA, 2010. ACM.

[183] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S. Lee. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[184] Nak Hee Seong, Dong Hyuk Woo, Vijayalakshmi Srinivasan, Jude A. Rivers, and Hsien-Hsin S. Lee. Safer: Stuck-at-fault error recovery for memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, 2010.

[185] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, and Stephen W. Keckler. Scalable hardware memory disambiguation for high ilp processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 399–, Washington, DC, USA, 2003. IEEE Computer Society.

[186] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, 2016.

[187] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. Proteus: A flexible and fast software supported hardware logging approach for nvm. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 178–190, New York, NY, USA, 2017. ACM.

[188] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the long latency of persist barriers using speculative execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 175–186, New York, NY, USA, 2017. ACM.

[189] Reetinder Sidhu and Viktor K. Prasanna. Fast regular expression matching using fpgas. In *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[190] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[191] Peter Snyder. tmpfs: A virtual memory file system. In *In Proceedings of the Autumn 1990 European UNIX Users' Group Conference*, pages 241–248, 1990.

[192] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. Aerospike: Architecture of a real-time operational dbms. *Proc. VLDB Endow.*, 9(13):1389–1400, September 2016.

[193] Michael Stonebraker and Lawrence A. Rowe. The design of postgres. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, 1986.

[194] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 2013.

[195] Prateek Tandon, Faissal M. Sleiman, Michael Cafarella, and Thomas F. Wenisch. Hawk: Hardware support for unstructured log processing. In *International Conference on Data Engineering*, 2016.

[196] Mellanox Technologies. Rdma aware networks programming user manual, 2018. `http://www.seastarproject.org/`.

[197] Transaction Processing Performance Council (TPC). Tpc benchmark C, 2010. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5-11.pdf`.

[198] J. van Lunteren. High-performance pattern-matching for intrusion detection. In *25th IEEE International Conference on Computer Communications*, 2006.

[199] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the USENIX Conference on File and Storage Technologies*, February 2011.

[200] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[201] Haris Volos, Andres Jaan Tack, and Michael M. Swift E. Mnemosyne: Leightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[202] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, Santa Clara, CA, 2017. USENIX Association.

[203] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, 2015. USENIX Association.

[204] H. Wang, S. Pu, G. Knezek, and J. C. Liu. Min-max: A counter-based algorithm for regular expression matching. *IEEE Transactions on Parallel and Distributed Systems*, 2013.

[205] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, June 2014.

[206] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Mechanisms for store-wait-free multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[207] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 70–83, New York, NY, USA, 2018. ACM.

[208] Xiaojian Wu and A. L. Narasimha Reddy. Scmfs: a file system for storage class memory. In *In Proceedings of the International Conference for High Performance Computing*, 2011.

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

[210] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 427–439, New York, NY, USA, 2019. ACM.

[211] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

[212] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496, New York, NY, USA, 2017. ACM.

[213] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND ssds. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 15–28, Santa Clara, CA, 2017. USENIX Association.

[214] Y. H. Yang and V. Prasanna. High-performance and compact architecture for regular expression matching on fpga. *IEEE Transactions on Computers*, 2012.

[215] Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. Compact architecture for high-throughput regular expression matching on fpga. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.

[216] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P. Jouppi, and Mattan Erez. Free-p: Protecting non-volatile memory against both hard and soft errors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, 2011.

[217] H. C. Yu, K. C. Lin, K. F. Lin, C. Y. Huang, Y. D. Chih, T. C. Ong, J. Chang, S. Natarajan, and L. C. Tran. Cycling endurance optimization scheme for 1mb stt-mram in 40nm technology. In *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 224–225, Feb 2013.

[218] Lunkai Zhang, Brian Neely, Diana Franklin, Dmitri Strukov, Yuan Xie, and Frederic T. Chong. Mellow writes: Extending lifetime in resistive memories through selective slow write backs. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 519–531, Piscataway, NJ, USA, 2016. IEEE Press.

[219] Michael Zhang and Krste Asanovic. Highly-associative caches for low-power processors. In *Kool Chips Workshop, MICRO*, volume 33, 2000.

[220] Wangyuan Zhang and Tao Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, 2009.

[221] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In

Proceedings of the 10th USENIX Conference on File and Storage Technologies, FAST'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[222] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, New York, NY, USA, 2015. ACM.

[223] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of 46th International Symposium on Microarchitecure*, 2013.

[224] Miao Zhou, Yu Du, Bruce Childers, Rami Melhem, and Daniel Mossé. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *ACM Trans. Archit. Code Optim.*

[225] Ping Zhou, Bo Zhao, Jun Yang, and Yutao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

[226] Yanqi Zhou, Ramnatthan Alagappan, Amirsaman Memaripour, A Badam, and D Wentzlaff. Hnvm: Hybrid nvm enabled datacenter design and optimization. *Microsoft, Microsoft Research, Tech. Rep. MSR-TR-2017-8*, 2017.