Compiler-Directed Whole-System Persistence

Jianping Zeng
Purdue University
zeng207@purdue.edu

Tong Zhang
Samsung Electronics
t.zhang2@samsung.com

Changhee Jung
Purdue University
chjung@purdue.edu

***Abstract*—Nonvolatile memory (NVM) technologies have gained increasing attention thanks to their density and durability benefits. However, leveraging NVM can cause a crash consistency issue. For example, if a younger store is evicted (persisted) to NVM from volatile caches before an older one and power failure occurs in between, it might be impossible to correctly resume the interrupted program in the wake of the failure. Traditionally, addressing this issue involves expensive persist barriers for enforcing the original store order, which not only incurs a high run-time overhead but also places a significant burden on users due to the difficulty of persistent programming.**

To this end, this paper presents cWSP, compiler/architecture codesign for lightweight yet performant whole-system persistence (WSP). In particular, cWSP compiler partitions not only user applications but also OS and runtime libraries into a series of recoverable regions (epochs), thus enabling persistence and crash consistency for the entire software stack. To achieve high-performance crash consistency, cWSP leverages advanced compiler optimizations for checkpointing a minimal set of registers and proposes simple hardware support for expediting data persistence on the cheap. Experimental results with 37 applications from SPEC CPU2006/2017, DOE Mini-apps, SPLASH3, WHISPER, and STAMP, show that cWSP incurs an average runtime overhead of 6%, outperforming the state-of-the-art work with a significant margin.

I. Introduction

Nonvolatile memory (NVM) [2], [8], [12], [13], [45], [50], [65], [77], [106], [122] technologies have been deemed an alternative to DRAM thanks to their irresistible features, *e.g.,* nonvolatility, byte-addressability, lower cost per bit, and near-zero standby power. They are now commercialized by many vendors, *e.g.,* Intel Optane persistent memory (PMEM) [54], Everspin STT-MRAM [1], and Fujitsu ReRAM [76]. Considering this, many cloud service providers and national labs—such as Microsoft Azure [28] and Argonne National Lab's Aurora [118]—already equip their server fleets with PMEM as a key to offering data-intensive workloads sufficient memory [34], [35], [63], [88], [113], [123]. However, indiscriminately replacing DRAM with PMEM incurs significant performance loss in that PMEM is slower than DRAM. According to Peng *et al.* [103], PMEM leads to 2-18x slowdown compared to DRAM for their graph benchmark applications.

Thanks to the emerging cache-coherent CXL (Compute eXpress Link) [22] technology, which offers high-bandwidth and low-latency interconnect based on PCIe interface, it is now practically possible to mitigate the performance issue of NVM. This is because CXL can enable a deeper and wider memory hierarchy at low cost. For example, local DRAM can serve as a last-level cache (LLC) positioned between the conventional L3 cache and the CXL-enabled low-tier PMEM, which is akin to Intel PMEM's *memory mode* [54] where DRAM acts as an LLC atop PMEM main memory. Adopting such deep cache hierarchy effectively lowers the chance of accessing slow NVM, making its performance drawbacks more tolerable. The upshot is that users can benefit from NVM's enticing features, such as nonvolatility, with a minimal impact on run-time performance.

Figure 1 illustrates the normalized execution time of using CXL PMEM compared to that of CXL DRAM for memory-intensive applications with 4 different cache hierarchies[1]: (1) 2-level caches comprised of 64KB 8-way L1 data cache with 4-cycle hit latency and 1MB 8-way L2 with 14-cycle hit latency; (2) 3-level caches by adding a 16MB 16-way L3 with 44-cycle hit latency; (3) 4-level caches by adding a 128MB 16-way L4 with 82-cycle hit latency [6]; (4) 5-level caches by adding a 4GB direct-mapped DRAM cache. The clear trend in the figure is that the performance loss gradually drops from 2.14x to only 1.34x along with the deeper hierarchy [6], [100]. This trend implies that the performance loss of NVM would be ignorable for the future deeper memory hierarchy enabled by CXL. More importantly, big data applications still benefit from CXL-enabled NVM owing to its high density, thereby maintaining their performance.

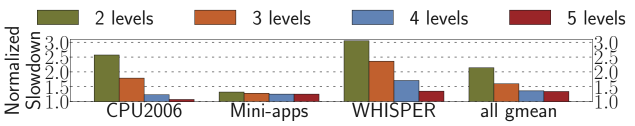

Fig. 1: Normalized slowdown of CXL PMEM main memory to CXL DRAM main memory with varying levels of caches

However, the naive use of NVM can lead to a crash inconsistency. To illustrate, suppose users try to insert a new node to the beginning of a doubly-linked list, which is done by (1) setting the new node's next pointer to the address of the old head node and (2) resetting the old head node's previous pointer to the address of the new node. Now, assume a scenario where the second store persists in NVM with the cacheline evicted from LLC while the first store is cached. If power is suddenly cut off here, the data stored in the cache is lost. This causes the new node's next pointer to become a dangling pointer, leading to inconsistent NVM states.

Considering this, Intel proposes eADR [26] to preserve the contents of volatile caches across power failure by just-

[1]Other architectural parameters are listed in Section IX.

979-8-3503-2658-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ISCA59077.2024.00074

in-time (JIT) checkpointing them to NVM right before the failure. Unfortunately, eADR requires a constant and high energy supply for the JIT checkpointing process. This becomes unsustainable, especially with the growing LLC size, *e.g.,* a 384MB L3 in the AMD EPYC 9654P CPU [52]. Even worse, eADR falls short in covering other volatile components, *e.g.,* registers and DRAM—at terabyte scale [110], which is the reason why eADR cannot guarantee crash consistency at all.

Thus, to make program persistent with NVM, users should resort to partial-system persistence (PSP) where NVM gets a separate address space next to DRAM's main memory space as with Intel PMEM's *app-direct mode* [19], [21], [29], [43], [56], [69]–[71], [107], [108], [114], [125], [126]. However, *PSP faces 4 challenges: (1) run-time or hardware cost, (2) difficulty of enabling DRAM cache, (3) programming burden, and (4) vulnerability to bugs.* First, software-based PSP schemes [78], [124], [125] require the insertion of persist barriers—*e.g.,* `clwb` and `sfence` in x86—to flush the data stored to NVM address space, while hardware-based schemes [57], [70], [95], [126] accelerate the store persistence using costly architectural support. Second, since DRAM serves as main memory, both PSP schemes cannot afford to exploit DRAM as the last-level cache (LLC)[2], thereby losing the performance benefit of the DRAM cache. Third, users are burdened with rewriting data structures with memory persistency [102] in mind, often leading to designing custom recovery logic for their crash consistency. That is why persistent programming is generally hard and error-prone [5], [86], [111], [116], [132]. Last, PSP requires a special memory allocator such as `pmalloc` [23], introducing the potential risk of persistent memory leaks. This renders already error-prone persistent programming even more complex and buggy [40], [42], [67], [84], [85], [90], [91], [97].

Given PSP's deficiencies, the interest in whole-system persistence (WSP) [58], [96], [130] is growing in both academia and industry. Since NVM serves as main memory in WSP allowing DRAM to be repurposed as LLC without hassle, WSP can enable deeper cache hierarchy and achieve high performance. Moreover, WSP transparently ensures the store persistence and the crash consistency for all kinds of program. However, WSP faces skepticism due to its complicated hardware design and substantial energy requirements [96] for flushing all the volatile states to NVM before impending power failure. The state-of-the-art WSP solution, Capri [58], addresses the skepticism to some extent but is still deemed impractical for several reasons: (1) significant storage overhead of 54KB per core for Capri's hardware buffers; (2) high amount of energy for JIT-checkpointing the buffers without power interruption; (3) complex hardware loggings and their demand for extremely high bandwidth of persist data path; and (4) inability to efficiently guarantee crash consistency for multiple memory controllers (MCs), and so on.

To this end, this paper presents cWSP, a synergistic compiler/architecture codesign to achieve lightweight yet performant WSP. The key idea is that cWSP can recover potentially inconsistent NVM states by re-executing a small portion of code. As such, cWSP compiler partitions any program including operating systems (OS) and runtime libraries—as long as they can be translated into LLVM bitcode [72]—into a series of idempotent regions (epochs) [32]. Since they are designed to be free of memory antidependence, they can be re-executed multiple times yet still generate the same correct output. This allows cWSP to resume program from the beginning of the power-interrupted region, *i.e.,* the end of the most recently persisted region. The takeaway is that cWSP obviates Capri's expensive hardware buffers and their JIT checkpointing while maintaining high performance.

Another prior work iDO [78] also leverages idempotent processing for power failure recovery. However, iDO works for only user applications and slows them down significantly; the reason is twofold: (1) iDO compiler generates superfluous memory writes to NVM which has limited write bandwidth and high write latency; (2) iDO causes the core pipeline to stall at the end of each region because 2 persist barriers are inserted before and after the region boundary. In contrast, cWSP addresses these issues with 2 pillars: (1) its compiler optimization eliminates unnecessary memory writes; (2) its hardware enables asynchronous store persistence, allowing the core pipeline to execute other instructions while persisting previous stores. In particular, cWSP persists the 8-byte data being stored to NVM through a FIFO persist path—built on Intel's existing non-temporal data path [27] with its write-combining buffer (WCB) disabled—immediately after the store instruction is committed. In this way, cWSP enables fast store persistence and lowers the bandwidth requirement for the persist path by 8x compared to all prior work relying on 64-byte data persistence [3], [41], [57], [70], [114], [126].

Last but not least, cWSP introduces a novel concept of *memory controller speculation*, aiming to efficiently ensure crash consistency in the presence of multiple memory controllers (MCs) that have non-uniform memory access (NUMA) time. This pivotal feature distinguishes cWSP from prior schemes [30], [41], [44], [57], [58], [70], [95]. They conservatively wait at each region (epoch) boundary for previous stores to persist in case the NUMA leads to a reordering of stores across regions and the resulting crash inconsistency on power failure in-between. In contrast, cWSP assumes power failure is unlikely between regions and *speculatively* persists the stores of the subsequent regions with no stall at their boundaries. As a safe net, the MCs undo-log the stores upon their arrival to handle potential misspeculation (*i.e.,* power failure occurred). Upon misspeculation, cWSP reverts the speculative NVM updates with undo logs to maintain consistent NVM states across the power failure.

The experimental results with 37 applications from SPEC CPU2006/2017 [10], [46], Mini-apps [64], [121], SPLASH3 [109], WHISPER [95], and STAMP [92] demonstrate that cWSP incurs an average of only 6% run-time overhead and a storage overhead of 176 bytes, making cWSP highly suitable for implementation on silicon, whereas the state-of-the-

[2]Unless additional hardware support is devised to use a portion of DRAM as cache leaving the rest for main memory.

art WSP work incurs a 27% run-time overhead despite its significant hardware overheads. In summary, cWSP makes the following contributions:

- cWSP is the first approach to a lightweight yet performant WSP—only a storage cost of 176 bytes (346x reduction of the state-of-the-art work's 54KB)—while supporting multiple MCs efficiently.
- cWSP eliminates the expensive yet power-hungry JIT checkpointing in prior approaches thanks to the intelligent compiler/architecture codesign.
- cWSP works well even for future CXL-based deeper and wider memory hierarchy—Section IX-C shows that cWSP incurs only a 4% run-time overhead for memory-intensive applications running on CXL-enabled NVM.
- cWSP provides a complete compiler toolchain—based on Clang/LLVM 13.0—that can rebuild the entire Linux software stack with crash consistency ensured.

II. Background and Motivation

A. Persist Path and Stale Read Issue

Prior PSP schemes [3], [57], [70], [95], [114], [126] utilize the existing non-temporal path [27] as a dedicated persist path of NVM stores—and thus drop their dirty cacheline evictions from LLC—to deliver the data to NVM in order, achieving *strict persistency [102]*. Here, data merged on L1 data cache are also placed on the persist path so that it directly transfers the data to NVM, bypassing the lower-level caches and thus avoiding costly persist barriers. However, these schemes come with some challenges. First, they demand a *high-bandwidth* persist path, which is not always feasible, as they persist a 64-byte cacheline to NVM for every 8-byte store merged into L1 data cache. Moreover, they delay the persistence of a store until it is merged into L1 data cache, significantly impacting the performance, especially given the high L1 data cache miss rate—22% for `470.lbm` of CPU2006 in our simulation.

More importantly, the use of the persist path without caution can cause a *stale read issue* [57], leading to wrong program output due to the lack of ordering guarantees between the persist path and the regular (cache) data path. Figure 2 (a) and (b) show how the issue occurs; in this example, `str 100,[A]`; `str 200,[A]`; and load `ldr r0,[A]` all access the same memory location *A*. Here, the two stores are merged into the same cacheline—which is later silently dropped from LLC as in prior schemes [3], [57], [70], [95], [126]—while the two memory updates are sent to NVM through the persist path. Suppose the cacheline is dropped (①) from LLC before `str 100,[A]` and `str 200,[A]` persist because of congestion in the persist path. If the core pipeline encounters an LLC miss for the load when only the first store has persisted (②), then the load ends up reading an outdated value from NVM—instead of the up-to-date value of 200.

To address the issue, prior work such as BBB [3] and DPO [70] include front-end persist buffers (PBs)—containing the stores for the example shown in Figure 2 (a)—in the cache coherence domain. That way, the load in Figure 2 (a) reads

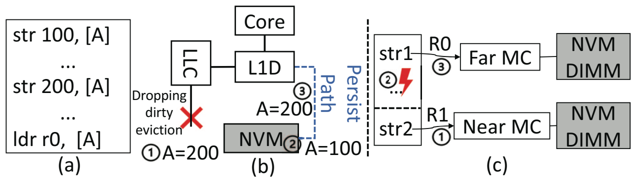

Fig. 2: (a) Original assembly code; (b) stale read issue occurred; (c) crash inconsistency for multiple MCs

the up-to-date data from the PBs. However, the prior work significantly complicates the already complex cache coherence protocol. On the other hand, other prior work like HOPS [95] delays the loads missing in LLC—in case the up-to-date data are pending on the persist path—until they persist in NVM. For this purpose, HOPS requires a bloom filter near memory controllers to check if the data are pending. The implication is that every NVM store must pay for long latency to access the backend bloom filter, which might hurt the performance. Either way, these prior schemes come with notable overheads, rendering them unsuitable for lightweight yet performant WSP targeting the CXL-enabled deeper memory hierarchy.

B. Multiple Memory Controllers and Crash Inconsistency

The presence of multiple memory controllers (MCs) in a server system [53] poses a daunting challenge in maintaining the FIFO ordering of the persist path, which is the basis for crash consistency, though they enable large memory space. As shown in Figure 2 (c), due to non-uniform memory access across MCs, a younger store—*e.g.,* `str2` in the 2nd region R1—could persist in NVM (①) before an older one (③) if they are destined to different MCs. This can cause inconsistent NVM states when power failure occurs in between (②). To address this issue, many prior proposals [30], [41], [57], [58], [70], [95] simply wait at region boundaries for prior stores to be persisted, thus degrading the performance.

C. Region-Level WSP with Persist Path

Capri [58], the state-of-the-art WSP scheme, addresses the challenges of persisting stores with compiler/architecture codesign for a wide range of applications[3]. Capri relies on a hardware-managed redo buffer [60], [61], [99] as the basis for crash consistency. For this reason, Capri compiler partitions input program into a series of recoverable regions with the buffer size in mind, preventing the buffer overflow during region execution and ensuring correct power failure recovery.

Figure 3 (a) shows Capri's high-level architectural diagram. During region execution, Capri copies the dirty cachelines touched by the region's stores to the redo buffer—next to the L1 data cache as shown in the figure. Each cycle, Capri attempts to transfer the data in the redo buffer to NVM through the persist path. For failure-atomic region persistence, Capri employs a 2-phase approach. It moves the redo buffer entries to a battery-backed proxy buffer—managed by the memory controller—and then from there to NVM media. The

[3]We get Capri compiler's source code from authors and figure out that it cannot compile runtime libraries though it covers the OS and user code.

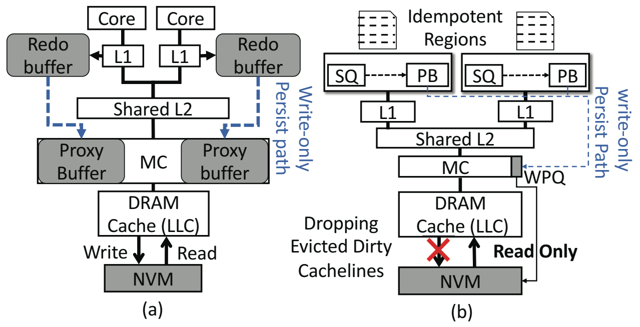

Fig. 3: (a) Capri architecture for PMEM memory mode; (b) cWSP architecture for PMEM memory mode; shaded boxes are in persistence domain; round boxes are newly proposed by either architecture; the thin persist path of cWSP indicates its lower bandwidth requirement

2-phase persistence ensures that either the proxy buffer or NVM remains intact across power failure.

However, such a redo buffer approach forces the CPU pipeline to stop at each region end until all its buffered stores are moved to the persistent domain (proxy buffer) before preceding to the next region, causing significant slowdown. To solve this issue, Capri lets the redo buffer battery-backed as well. This allows the next region to immediately start in that the prior one has buffered its stores already in the persistent domain (redo buffer).

D. *Limitations of Prior WSP Work*

Capri faces 5 issues, rendering its practical use impossible. First, its hardware buffers incur a high storage cost, totaling (N + 1) × M × 18KB, where N corresponds to the memory controller count, M to the core count. For example, Capri results in a storage overhead of 88MB for AMD 128-core EPYC 9754 processors with 12 MCs [53]. Second, Capri requires a considerable amount of energy at all times for JIT checkpointing, leading to battery maintenance burden and environmental impact [68], [93], [104], while power failure is scarce in server fleets. Third, Capri relies on an over-complex redo+undo logging to recover potentially inconsistent NVM status by using undo or redo logs depending on where the 2-phase persistence is power-interrupted. This complex hardware logging scheme ends up amplifying NVM writes by 8x and demands an extravagant bandwidth for the persist path.

Forth, Capri incurs extra hardware cost for resolving the stale read issue. That is, Capri delays DRAM cache eviction to scan the proxy buffer and invalidate the matched proxy buffer entry of the same address. Even if no matching entry is found, which is a common case, Capri cannot release the DRAM cache eviction. That is because the data being matched might be pending on the persist path; therefore, Capri should wait for the worst-case data delivery latency in case the data is to be found within the latency [58]. Finally, Capri causes a high run-time overhead for server-class cores with many MCs [53] due to frequent persistence stalls at the end of short regions—29 instructions in regions on average.

III. cWSP Overview

Figure 3 (b) shows the architectural diagram of cWSP. In particular, cWSP's persist path connects each core to MC unlike Capri's starting from L1 data cache. As will be shown in Section IX-C, cWSP works well for CXL-based NVM, in which case the persist path ends at CXL Home Agent [22], though cWSP assumes the less complex Intel PMEM memory mode by default for a fair comparison to the state-of-the-art work Capri.

A. *Region-Level Crash Consistency for All*

To achieve crash consistency for the entire Linux software stack, cWSP compiler is capable of partitioning any C/C++ program including the OS kernel into a series of idempotent regions [32], [78], [80], [81], [83], [133], that are free of memory antidependence also known as write-after-read dependence, serving as the basis for *recovery-via-resumption*. Similarly, cWSP also ensures crash consistency for C/C++ libraries and the Linux kernel by partitioning their functions such as `malloc` and `sbrk`; see Section IV for details.

B. *Asynchronous Store Persistence*

Unlike all prior work [3], [44], [57], [58], [78], [95], [125], cWSP *for the first time* decouples store persistence from cache access. That is, cWSP persists the data being stored as soon as the store is committed. To achieve this, cWSP repurposes Intel's write-combining buffer (WCB) as a volatile persist buffer (PB) that connects from store queue (SQ) to the memory controller (MC) as shown in Figure 3 (b). Each time a store is committed, its data is copied to the PB and then transferred to the MC along the persist path in the background. The implication is twofold: (1) cWSP persists stores at 8-byte granularity and thus brings an eightfold reduction in the persist path bandwidth, compared to the prior work based on 64-byte cacheline granularity; (2) cWSP exerts practically no pressure on the SQ, which would otherwise slow down the core pipeline execution. Further details are deferred to Section V-A.

C. *Memory Controller (MC) Speculation for Multiple MCs*

To ensure high-performance crash consistency even in the presence of multiple memory controllers (MCs), cWSP proposes *memory controller speculation*. While the stores of a region are on their way to NVM locations, cWSP *speculatively* persists the following regions' stores with the data logged in NVM, despite non-uniform memory access across MCs, assuming power failure is unlikely in the meantime. cWSP leverages undo logging to enable in-place updates and avoid costly read redirection. If misspeculation (*i.e.,* power failure) occurs, cWSP reverts the speculative NVM updates using the undo logs, thereby maintaining consistent NVM states across power failure; details are provided in Section V-B.

D. *Power Failure Recovery Protocol*

Since the memory controller speculation of cWSP allows multiple regions to be persisted concurrently, care should be taken to ensure correct power failure recovery. It is possible that these regions have at least some of their stores persisted before power failure. This paper calls such regions *unpersisted*. On the other hand, a region is called *persisted* only after its stores are all persisted.

In the wake of power failure, cWSP resumes the interrupted program in 3 steps with identifying a boundary between persisted and unpersisted regions: (1) reverting speculative NVM updates using undo logs; (2) preparing the inputs to the oldest *unpersisted* region, the entry of which serves as the recovery point; and (3) restarting the region from the beginning; details are found in Section VII.

IV. cWSP Compiler and Runtime System

A. *Cutting Memory Antidependence*

To partition program into a series of idempotent regions, it is a critical step to ensure the absence of memory antidependencies within each region. For this purpose, cWSP uses the same idempotent processing algorithm developed by De Kruijf *et al.* [32]. First, cWSP compiler treats function callsites and synchronization points—such as atomic operations and memory fences—as initial region boundaries. cWSP also inserts a region boundary at the header of each loop, forming a region per iteration; of course, extra boundaries are inserted in the loop body to split other memory antidependence therein. Second, cWSP compiler computes a set of cutting points for antidependence pairs of memory using LLVM's *alias analysis*. Later, cWSP compiler uses a hitting set algorithm to find out the best partitioning strategy. As Figure 4 (a) shows, a region boundary separates `r2 = ldr [r0]` and `str r1,[r0]` and keeps them in two separate regions.

B. *Checkpointing Live-Out Registers*

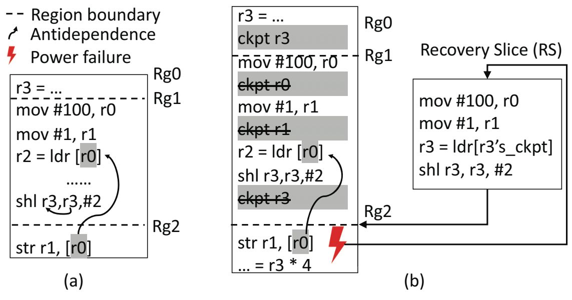

Fig. 4: (a) Cut memory antidependence; (b) inserts the checkpoints for live-out registers and then prunes all 3 checkpoints in region Rg1; note that region Rg0/R1 are already persisted before power failure (⚡), whereas Rg2 is not

However, solely preventing memory antidependence within regions is insufficient to achieve WSP, as volatile registers lose their data upon power failure. To address this issue, cWSP compiler checkpoints (saves) registers to a designated storage in NVM, indexed by architectural registers and managed by cWSP hardware. cWSP compiler first calculates a set of *live-out* registers for each region using LLVM's *liveness analysis* and then checkpoints their values to NVM. Figure 4 (b) shows that `ckpt r3` is inserted in region Rg0 since `r3` is *live-out*—*i.e.,* it is used by some later region(s).

C. *Pruning Register Checkpoints*

To mitigate the potential increase in write pressure on the persist path caused by inserted checkpoints (essentially store instructions), cWSP leverages the optimal checkpoint pruning algorithm of Penny [66]. We found out that this optimal checkpoint pruning, originally designed for soft error resilience, can efficiently eliminate redundant checkpoints without compromising the crash consistency guarantee. The intuition behind the checkpoint pruning is that many checkpoints are unnecessary if they can be reconstructed using immediate values and/or the remaining checkpoints at recovery time. For example, all 3 checkpoints in region Rg1 are eliminated as shown in Figure 4 (b), improving the performance greatly (see Section IX-B). Across power failure (⚡) occurred in region Rg2, cWSP's recovery runtime first executes Rg2's recovery slice (RS)—on the right of Figure 4 (b)—to reconstruct the values of region Rg2's 3 live-in registers. As shown in the RB, $r0$ and $r1$ are reconstructed from 100 and 1, respectively, while $r3$ is done by (1) loading the value checkpointed in region Rg0 and (2) applying the shift instruction over the value. With these input registers restored, cWSP then resumes the interrupted program from the beginning of the region Rg2.

D. *cWSP Runtime and Linux Kernel*

Ensuring crash consistency for the entire software stack—covering user program, runtime libraries, and the Linux kernel—is crucial for the successful implementation of cWSP. However, this is not adequately addressed in previous approaches [58], [84], [95], [128], due to the lack of C library in LLVM community and the incompatibility between the Clang/LLVM compiler and the GNU C library `glibc` [36]. To overcome this obstacle, cWSP introduces a comprehensive crash-consistent runtime for the first time. We patch essential libraries, including `glibc`, LLVM C++ library `libcxx` [39], LLVM `compiler-rt` [37], and LLVM stack unwinding library `libunwind` [38]. In addition, we patch the configuration of `glibc` to allow for its compilation with cWSP compiler. In particular, all the assembly files pertaining to x86_64 are manually patched to insert region boundaries and checkpoints. It is also feasible to lift assembly code up to LLVM bitcode using mature lifting tools, *e.g.,* Remill [101], in which case cWSP compiler optimizations can be automatically applied along with the recoverable region formation.

V. cWSP Hardware Implementation

A. *Asynchronous Store Persistence: Challenges and Solutions*

1) Preventing Stale Read Issue on the Cheap: Recall that the stale read issue arises on LLC load misses. That is because there is no ordering guarantee between the persist path and the regular path where LLC silently drops dirty cachelines on their eviction, though the data of committed stores move to both paths in our case. That is, there is a potential for a race condition [98] between (1) the read on the regular data path and (2) the write on the persist path. Fortunately, we found out that the stale read issue almost never occurs due to the faster persist path, *i.e.,* data being read by those loads missing LLC are sure to have already persisted in NVM. The *load-after-persist* order is made most of the time in that the data carried over the persist path can directly head to NVM whereas they go through multiple levels of caches in the regular path.

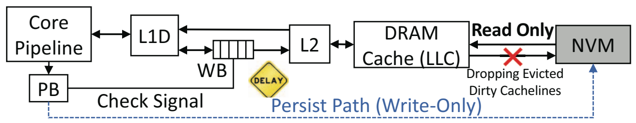

Fig. 5: Solving stale read issue by delaying dirty cacheline writeback from the WB of the private L1D to the shared L2

Given the rare occurrence of the stale read issue, any possible solution must be lightweight enough to minimize the impact on the core pipeline execution. With that in mind, cWSP enforces the *load-after-persist* order occasionally, *i.e.,* when the data is about to reach the shared L2 on the regular path. That is, cWSP only needs to ensure that no writeback is made to the L2 until the same data—once placed in the persist path—is eventually written to NVM. To achieve this, cWSP delays the writeback of dirty cachelines from the private L1D's write buffer (WB) to the shared L2, provided the persist path has not yet flushed the corresponding data to NVM. As shown in Figure 5, when a cacheline at the WB head is about to be drained, a check signal with the cacheline address is issued to search for a matching entry in the PB. If found, cWSP holds the writeback of the head until the matched PB entry is persisted in NVM.

Coherence-Agnostic PB: The upshot of the above simple technique is that the PB is out of cache coherence domain, *i.e.,* the entire caches and the coherence protocol both remain the same. That is because cWSP ensures a memory read always retrieves the up-to-date data either from the caches on their hits or from NVM when missing in the DRAM cache (LLC). Thus, accessing the PB for loads becomes unnecessary, while prior work [3], [70], [95] consults PB either directly from the core or through cache coherence requests for loads thereby complicating already complex cache coherence mechanisms.

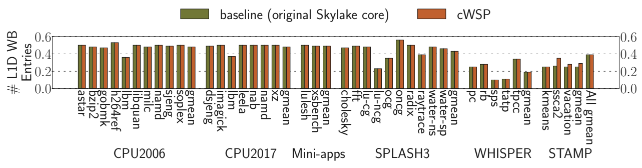

Fig. 6: Average occupancy of the WB of L1 data cache for baseline and cWSP

At first glance, one might think that delaying the WB writeback could potentially slow down pipeline execution, especially when the WB is full and a new WB entry needs to be allocated for an incoming dirty eviction. However, our empirical evaluation shows that this delay has no adverse effect on performance at all (see Section IX-B). Figure 6 shows that both the baseline and cWSP maintain an average occupancy of only 0.39 WB entries, implying minimal pressure on the WB. This negligible impact can be attributed to two factors. First, the persist path is way faster than the regular path. When a dirty cacheline is about to be written back from the WB to L2, its corresponding PB entry is most likely persisted in NVM already, resulting in no matching found in the PB. Second, content-addressable memory (CAM) searching for the (50-entry) PB can complete in just 1 cycle (0.5*ns*), causing technically no delay on the WB writeback; this is supported by IBM's report on a 0.6*ns* CAM search time for a 64x72 CAM with 65*nm* CMOS technology [4].

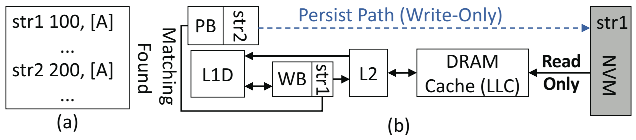

Fig. 7: (a) assembly code; (b) a false positive

False Positives: Note that cWSP guarantees the absence of *false negatives* for the PB searching, since it creates a PB entry for each store before the data is merged into L1D. However, theoretically, cWSP could mis-identifies a PB entry though it does not collide with the WB head entry, causing false positives and unnecessary delay in the WB writeback. Nevertheless, we never observed false positives in our experimentation, owing to the huge speed gap between the regular path and the way faster persist path. Figure 7 illustrates a hypothetical false positive scenario. Consider two committed stores—$str1$ and $str2$—writing values 100 and 200, respectively, to the same memory address A. In this scenario, $str1$'s data (100) has already been persisted in NVM via the persist path, and its corresponding dirty cacheline has been evicted from the L1 data cache to the WB. Meanwhile, the core pipeline allocates a PB entry for $str2$, which misses the L1 data cache. Here, if the WB is about to flush $str1$'s dirty cacheline to the L2, $str2$'s PB entry is mistakenly perceived as a match with $str1$ at the WB head, causing a false positive.

2) Lowering Persist Path Bandwidth at No Cost: Due to the 8-byte data granularity of the persist path, cWSP's write pending queue (WPQ) maintains 8-byte entries as well. Therefore, care must be taken to ensure correctness in that the memory system transfers data at a 64-byte granularity. If a load misses the LLC (DRAM cache) and encounters a WPQ hit, then it can only get the corresponding 8-byte WPQ entry failing to retrieve the remaining 56-byte data, in which case program correctness is broken.

To address this potential incorrectness without complex hardware support, cWSP simply postpones serving those loads hitting the WPQ until the matching WPQ entry persists in NVM. In particular, this delay has no practical impact on performance owing to the remarkably low WPQ hit ratio. Figure 8 shows 0.98 hits per 1 million instructions. Such a low hit ratio has a twofold implication: (1) with an increasingly deeper memory hierarchy where fewer read requests reach the

NVM, the WPQ hit ratio gets even lower; (2) cWSP effectively expands the WPQ's capacity by eightfold—compared to conventional WPQ whose entry size is 64-byte—without requiring additional storage. Consequently, cWSP is well-suited for the future deeper/wider memory hierarchy.

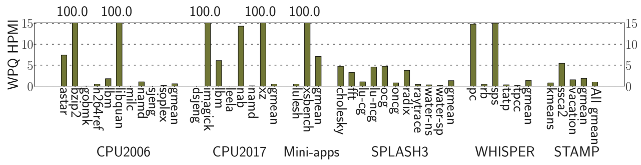

Fig. 8: WPQ hits per 1 million instructions

B. *Memory Controller Speculation*

1) Store Persistence without Stalling at Region Boundaries: With multiple memory controllers (MCs), the stores of a younger region could persist before those of older regions due to the non-uniform memory access (NUMA) time of the MCs. As mentioned earlier in Section II-B, this out-of-order region persistence breaks the FIFO nature of the persist path that serves as the basis for crash consistency. As such, prior schemes [30], [41], [57], [58], [70], [95] resort to stalling the core pipeline at each region boundary (transaction end) until every store of the region persists. That is, they do not allow inter-region persist reordering, leading to high performance loss server-class cores backed with many MCs [53].

To achieve correct yet performant crash consistency in the presence of multiple MCs, we make two observations. First, despite the NUMA effect, the resulting persist reordering within a region is not harmful, since it can be correctly recovered (re-executed) thanks to cWSP's idempotent region formation. So, we are only concerned about the inter-region persist reordering that makes the idempotent recovery incorrect—since idempotence holds on a per-region basis. Second, nevertheless, the inter-region persist reordering can be alright, provided it is not caught by power outages. Even if they occur, cWSP's recovery runtime can leverage conventional logging to revert the out-of-region-order persists that might corrupt the input(s) to the oldest unpersisted region—being re-executed by the recovery protocol (Section III-D); and the recovery cost should be insignificant given the rarity of power failure. Those observations inspire us to develop *memory controller speculation*, assuming that the oldest unpersisted region never encounters power failure. With that in mind, cWSP keeps speculatively persisting the data of the following regions, without waiting for the oldest to get persisted.

Note that the oldest unpersisted region is non-speculative and vice versa in that the prior one has already been persisted, whereas the following regions are under speculation. In case power failure interrupts the persistence of the oldest unpersisted region (*i.e.,* misspeculation), cWSP undo-logs any data being speculatively persisted. This allows cWSP to restore the memory status to point where the oldest unpersisted region is about to start—for the correct re-execution of the idempotent region (Section III-D).

To illustrate, consider 4 consecutive regions: Rg0, Rg1, Rg2, and Rg3. Suppose Rg0 has been persisted. Rg1 is currently the oldest unpersisted region, *i.e.,* it is non-speculative, whereas Rg2 and Rg3 are under speculation. While Rg1 persists its data being stored in NVM, cWSP *speculatively* persists and undo-logs the data of Rg2 and Rg3, preparing them for potential reversal in the event of power failure. When the speculation turns out to be true, *i.e.,* Rg1 has persisted all its stores without power interruption, Rg2 thus becomes non-speculative, which causes Rg2's logs to be deallocated. Here, Rg3 still remains speculative though.

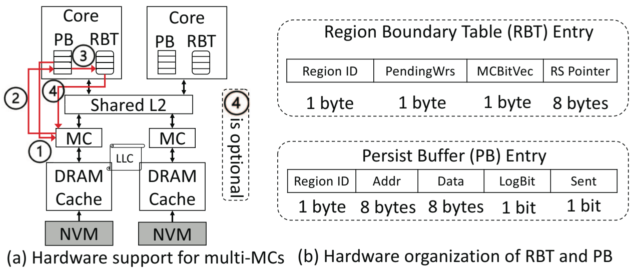

Fig. 9: Hardware organization for MC speculation; RBT is newly proposed, while PB is built on Intel WCB

To track speculation state and perform its corresponding actions, cWSP should recognize (1) if a region is speculative (or non-speculative) and (2) if it is persisted. For this reason, cWSP tracks two kinds of information, *i.e.,* speculation and persistence metadata, for each region. As shown in Figure 9, cWSP prepares two FIFO queues[4]: the region boundary table (RBT) for the speculation metadata and the persist buffer (PB) for the persistence metadata, respectively. At a high level, each region is treated speculative upon entry into the RBT and remains so until it moves to the RBT head that always points to a non-speculative region. The RTB head entry is removed as soon as its corresponding region (*i.e.,* the oldest unpersisted one) is persisted. The implication is that RBT size determines the number of speculative regions.

Specifically, when the core pipeline commits a region boundary instruction, cWSP allocates an RBT entry for the current region being started. The RBT entry contains 4 items: (1) `Region ID`, a hardware-managed counter that atomically increases to ensure unique ID allocation across all cores; (2) `PendingWrs` indicating the number of unpersisted stores in the region; (3) `MCBitVec` tracking the IDs of the MCs to which the region's stores are directed; and (4) `RS Pointer` referring to the starting address of the region's recovery slice (RS), which is encoded in the region boundary instruction; Section VII details RS. Similarly, when a store instruction commits, cWSP performs two actions. First, a PB entry is allocated to track its persistence status. Here, each PB entry contains 5 items: (1) `Region ID`—never overflowing as at most 128 regions are allowed to be persisted concurrently, given 8 cores and 16 RBT entries by default—of the current

[4]They have one read/write port and one search port to complete CAM searching in one cycle.

region which is retrieved from the RBT tail entry; (2) store address `Addr`; (3) `Data` being stored; (4) a boolean `LogBit` telling if the store is from a speculative region and thus should be undo-logged; and (5) a boolean `Sent` stating if the store has been delivered to NVM. Second, for the committed store, cWSP increases the RBT tail entry's `PendingWrs` by 1 and updates its `MCVectBit` with the store's MC ID.

As shown in Figure 9, the coordination between the RBT and the PB is crucial for keeping the speculation status of every region up-to-date. Each cycle the PB keeps sending its entry to the target MC (①) with the `Sent` set in a pipelined manner—unless the WPQ is full, which is not common (see Section IX-L). Technically, the first 4 items of the PB entry (*i.e.,* `Region ID`, `Addr`, `Data`, and `LogBit`) are sent to the WPQ of the target MC[5]. Upon the arrival of the `Data` at the WPQ, it is considered persisted—as the WPQ is in the persistent domain [24]—and undo-logged if the `LogBit` is set. Simultaneously, the MC acknowledges the PB (②), which deallocates the entry if it is the head of the PB. Then, cWSP identifies the RBT entry corresponding to `Region ID` and decreases its `PendingWrs` by 1 (③). Finally, if the `PendingWrs` becomes zero with the entry pointed by the RBT head (*i.e.,* the non-speculative region is now persisted), cWSP deallocates the entry, making the following region non-speculative; this results in (1) reclaiming its undo logs (see Section V-B2) and writing the RBT head entry's `RS Pointer` to NVM for future power failure (④).

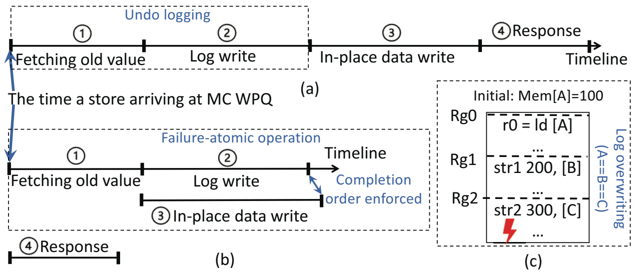

Fig. 10: (a) Naive undo logging at MC; (b) cWSP hardware undo logging at MC; (c) Log overwriting issue; Rg0 is non-speculative, while Rg1 and Rg2 are speculative

2) Hardware Undo Logging: Since the undo logging is on the critical path for every NVM write, its implementation should be performant. Figure 10 (a) shows how the critical path of each NVM write is extended by a naive implementation with fetching the old value from the address of the store (①); performing the log write (*i.e.,* the address and the value) (②); performing the in-place data write (③); and responding to the core (④). Obviously, this causes a high run-time overhead.

To this end, cWSP proposes asynchronous undo logging, *i.e.,* the MC immediately acknowledges a store arriving there (④), while its data is undo-logged and written to NVM in the background (①-③) as shown in Figure 10 (b). This allows the latter to be off the critical path. To achieve this correctly, cWSP requires that for each store, the undo logging and the data write should be failure-atomic as a whole. That is, the MC should secure enough energy for completing the entire operation (①-③) without power interruption in between—as Intel ADR secures the energy necessary for flushing all WPQ entries [24]. That way when power is about to be cut off, cWSP guarantees to flush every WPQ entry with its data undo logged. Note that the MC starts the undo logging of data being stored (①-②) as soon as it gets to WPQ. Nevertheless, when the WPQ entry is about to be flushed to NVM, its undo-logging might not have been finished, in which case the MC should hold the flushing until their completion order (②→③) is enforced for correct recovery. The rationale here is that the failure-atomic operation—including the undo logging—is only possible for the entries present in WPQ, not those already removed there.

In particular, care must be taken to prevent undo logs from being overwritten, which would otherwise corrupt NVM states causing incorrect power failure recovery. Figure 10 (c) shows how the undo log overwriting issue arises. Here, Rg0 is non-speculative, *i.e.,* the oldest unpersisted region to be re-executed in case of power failure, while Rg1 and Rg2 are speculative. Suppose addresses `A`, `B`, and `C` happen to be the same, *i.e.,* `str1`'s log is overwritten by `str2`'s log if they share the same log location. When power failure (ϟ) occurs in Rg2, cWSP mistakenly uses the `str2`'s log to revert Rg1's speculative NVM updates, resulting in inconsistent NVM states. That is because *ld* in Rg0 incorrectly reads 200 (not 100) when it restarts in the wake of the power failure.

To this end, cWSP leverages *append-only* logging for eliminating the overwriting within a region and across regions. The implementation principle here is twofold: (1) lightweight log management without additional hardware support and (2) simple log deallocation with no search cost. In light of this, cWSP requires that each MC should (1) maintain the logs of stores arriving there in its local NVM space—rather than resorting to centralized logging with inter-MC communication—(2) manage the logs on a per-region basis such that each region's logs can be deallocated using the `Region ID`; upon receiving the first store of a speculative region, the target MC allocates a log array for the `Region ID` in its own log area; once a region gets non-speculative, its idempotent recovery no longer requires its own logs—though it needs those of the following speculative region(s). This implies that cWSP can safely deallocate the logs of the non-speculative region without compromising the recovery guarantee. To achieve that, cWSP consults the `MCVecBit` of the RBT head—referring to the non-speculative region—and signals its target MCs to reclaim the log arrays corresponding to the `Region ID`. Notably, the size of the log area is limited since each region has only a handful of stores (4 on average) and the number of regions being concurrently persisted is capped by the RBT size.

VI. Crash Consistency for System Calls

In the pursuit of whole-system persistence, cWSP faces a challenge in ensuring consistent NVM states during system

[5]This requires one bus transaction for x86_64 since an `Addr` occupies only 48 bits thus being encoded with `Region ID` and `LogBit` into an 8-byte.

calls that require context switch from the user space to the kernel space. This challenge arises because the `entry function`—invoked by every system call—is implemented using assembly code, and it cannot be partitioned by cWSP compiler into idempotent regions. As a result, the *entry function* is not recoverable if power failure occurs therein.

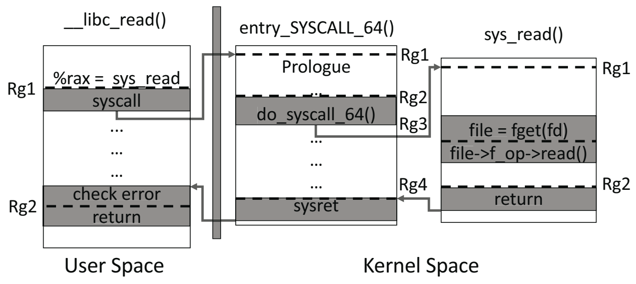

Fig. 11: Region formation for Linux system calls

To address the above challenge, we manually delineate region boundaries and insert register checkpoints in the *entry function*, *i.e.,* `entry_SYSCALL_64` in the assembly file `entry_64.S`. The overhead caused by these checkpoints is minimal, since a typical system call involves more than 4000 instructions [117], though other auxiliary functions called by `entry_SYSCALL_64` should also be instrumented. Figure 11 shows that 2 region boundaries are inserted at the entry and exit points of `entry_SYSCALL_64`, and another region boundary is inserted right before the callsite `do_syscall_64`; it transfers the program control to the beginning of `sys_read` function pointed to by the input register `%rax`; region boundaries for other callsites are omitted in the figure. With the help of the manual annotation on `entry_SYSCALL_64` and its auxiliary functions, cWSP can ensure crash consistency for all in that cWSP compiler already recompiles the `glibc` and the Linux kernel.

VII. Recovery Protocol

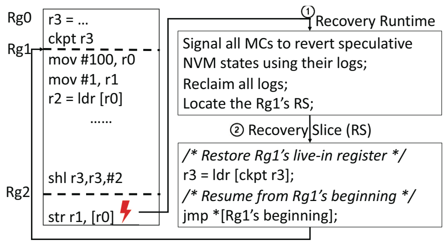

Fig. 12: Recovery process for the interrupted (ϟ) Rg1 and Rg2

In the wake of power failure, cWSP follows its recovery protocol to resume the power-interrupted program from the recovery point—the beginning of the oldest unpersisted region. That is, for the preparation of the region re-execution, cWSP's recovery runtime (1) leverages undo logs to make NVM states consistent and (2) jumps to the region's recovery slice (RS) where its live-in registers are restored. Figure 12 shows how the recovery protocol works. Suppose Rg0 has already been persisted, and Rg1 is the oldest unpersisted region while Rg2 is speculative. When power failure interrupts Rg1 and Rg2, cWSP's runtime first signals all MCs to revert speculative NVM states; each MC processes its own per-region logs in a reverse chronological order of `Region ID` and then deallocates all its logs (①). The runtime then jumps to Rg1's RS (②) that restores its live-in register $r3$. Finally, at the end of the RS, it transfers the program control back to the beginning of Rg1, and the program resumes the execution as is thereafter.

VIII. Discussion

Recovery for Multi-Cores: To ensure correct power failure recovery for multi-threaded applications on multi-core processors, cWSP maintains inter-thread dependency [59] by treating synchronization primitives, such as *atomics* and *fences*, as region boundaries as with prior techniques [11], [49], [55], [57], [78], [130]. That way, cWSP ensures that stores prior to synchronization primitives are not only merged into the L1 data cache but also persisted before the primitives are committed. As a result, for data-race-free (DRF) program assumed by C/C++ 11 onward, a dependent thread can only enter a critical section after a source thread has already persisted the stores of the section and exited the section. The implication is twofold: (1) upon power failure, there is at most one thread in the same critical section; (2) in the wake of power failure, each thread resumes its execution from the end of the latest persisted section (region) independently without the need to track the happen-before relationship among threads.

Why Not Use Store Queue as Persist Buffer: Utilizing the store queue (SQ) as a persist buffer would result in stores being held in the SQ for an extended period, thereby putting more pressure on the SQ. While enlarging the SQ could alleviate the pressure to some extent, it would also increase the latency of the critical store-to-load forwarding [112], [119], thus affecting the core pipeline performance.

I/O and Device States: To the best of our knowledge, supporting irrevocable operations like I/O has been an open problem. Despite, cWSP can be extended to have battery-backed redo buffers—organized as a FIFO queue—to ensure consistent I/O device states across power failure. We suggest that the number of the redo buffers should match the RBT size with each buffer indexed by a `Region ID`. This allows multiple regions to be persisted concurrently as with the RBT. During the execution of a region, its I/O operations are held in the corresponding redo buffer. Once the oldest unpersisted region becomes persisted, *i.e.,* all its I/O operations already arrive at the corresponding redo buffer, cWSP flushes their data to the corresponding devices.

In the event of power failure, cWSP performs two actions for recovery. First, it exploits the system's ACPI (Advanced Configuration and Power Interface) [62], [96] to save device states—including internal buffers and registers—to NVM. Second, cWSP examines the FIFO queue from front to rear to flush I/O data of each persisted region to their target devices. To ensure in-order region persistence, cWSP stops such an examination when an unpersisted region is encountered even if there might exist following persisted regions. As such, the device states get consistent back with those when the oldest unpersisted region started in the first place. When power comes back, cWSP's runtime resumes the execution of the device driver code from the beginning of the oldest unpersisted region—which is the recovery point as always.

Software Bugs: Software bugs can corrupt memory data and in turn lead to system crash. However, this is different from what cWSP pursues since they are two different problems. For example, any existing systems that maintain crash consistency, including databases and long-running machine learning (ML)/high performance computing (HPC) applications, could still experience crash caused by software bugs.

No Power Failure Recovery Test: At this moment, cWSP does not conduct experiments for system-level recovery from power failure, which we admit is a limitation of the current evaluation. We leave addressing the limitation for our future work. Nevertheless, the recovery overhead of cWSP would be negligible since it re-executes only tens of instructions in power-interrupted regions as described in Section IX-E.

IX. Evaluation and Analyses

We implement our compilation optimizations atop LLVM [72] that compile all the evaluated applications with -O3 flag; they are statically linked against cWSP runtime. Our compiler passes consist of about 4000 LOC with comments excluded.

We implement our hardware design atop gem5 [9] simulator to model an 8-core Skylake processor [33] with 2 memory controllers (MCs). Each of them manages DRAM as an off-chip direct-mapped LLC as with Intel PMEM's memory mode. Each core is equipped with a 64KB 8-way private L1 data cache with 4-cycle hit latency and a 32KB 8-way private L1 instruction cache with 3-cycle hit latency. All the 8 cores share a 16MB 16-way L2 cache with 44-cycle hit latency. The DRAM cache is configured to 4GB DDR4 2400 8x8. We set NVM main memory to 32GB with read/write latency of 175*ns*/90*ns* [126], [127]. Each MC has a 24-entry battery-backed WPQ, while RBT/PB sizes are set to 16/50. The round-trip latency of the persist path is set to 20*ns* (40 cycles) as with prior schemes [57], [58], which is considered conservative as a prior work Hermes [7] assumes a separate data path of 36-cycle round-trip latency. In addition, cWSP's persist path requires a bandwidth of only 4GB/s, which is realistic considering a 25GB/s DRAM bus [25]. We treat the original program running on the original hardware platform without crash consistency support as our baseline.

To highlight WSP's benefits, we evaluate a variety of benchmarks, *e.g.,* CPU2006/2017 [10], [46], SPLASH3 [109], WHISPER [95], STAMP [92], and Mini-apps [64], [121]. We simulate CPU2006/2017 program with reference input and modify the source code of WHISPER to stress the DRAM cache. all SPLASH3/WHISPER/STAMP applications are simulated in gem5 FS mode. As with prior schemes [31], [32], [51], [75], [82], [108], [115], [129], [131], we synchronize the simulation window by measuring the number of function calls—a constant across different binary versions generated by varying compiler optimizations—in the baseline to fast-forward 5 billion instructions and then simulate 1 billion instructions in gem5's O3CPU model.

A. Run-time Overhead Analysis

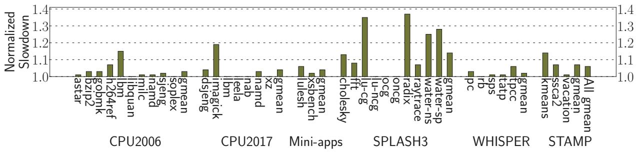

Fig. 13: Normalized slowdown of cWSP to the baseline; the persist path bandwidth is 4GB/s; lower is better

Figure 13 shows that cWSP incurs an average of only 6% run-time overhead across 37 applications. Notably, cWSP incurs higher overheads for SPLASH3 applications, *e.g.,* `lu-contig` and `radix`. This is because: (1) their baselines have a short execution time due to their low L1 data cache miss rates (∼2%); they have good data locality due to many sequential/repeated writes; (2) these sequential/repeated writes exert a high pressure on the persist path, overflowing the PB/WPQ frequently and thus causing the higher overheads. In contrast, other applications exhibit less normalized run-time overheads due to less frequent NVM writes.

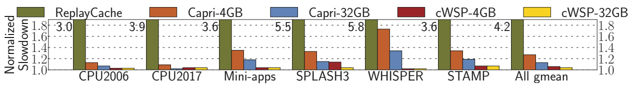

Fig. 14: Normalized slowdown of cWSP and other WSP schemes; lower is better; the numbers after dash in the legend indicate the persist path bandwidth

In addition, we compare cWSP with two prior WSP schemes, ReplayCache [128] and Capri [58], to underscore the exceptional performance of cWSP. ReplayCache is adapted to to the evaluated server-class processor since it was originally designed for energy harvesting systems [16]–[18], [20], [48], [79], [87], [89], [105], [134] where WSP is the norm. In the evaluation of cWSP and Capri, we consider two persist path bandwidth configurations: a practical 4GB/s and an ideal 32GB/s. As shown in Figure 14, cWSP outperforms both prior schemes across all benchmarks. ReplayCache results in a significant slowdown (4.3x) due to its software-oriented design, while Capri backed with 4GB/s persist path bandwidth incurs an average of 27% run-time overhead due to the contention on the persist path. Only with the ideal persist path bandwidth, can Capri be on par with cWSP.

B. Performance Impact of Each Optimization

To show how each cWSP optimization affects the run-time overhead, we break down the overhead as shown in Figure 15.

Region Formation: reveals that cWSP's region formation incurs an average of only 4% run-time overhead.

Persist Path: is the combination of above optimization with persisting stores to NVM through the persist path. This increases the average run-time overhead to 10% primarily because of the contention for the persist path.

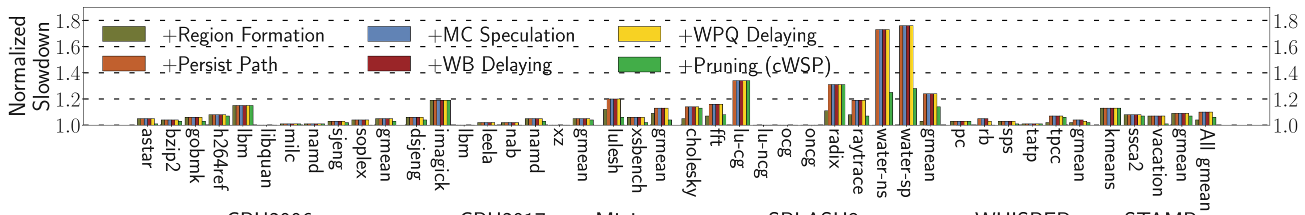

Fig. 15: The performance impact of each cWSP optimization; lower is better

MC Speculation: is the combination of all above optimizations with MC speculation. The resulting overhead remains the same since a 16-entry RBT is sufficiently large to cover the persist path latency; details deferred to Section IX-H.

WB Delay: is the combination of all above optimizations with delaying the writeback from the L1D's WB; there is no extra overhead incurred (see Section V-A1 for the reason).

WPQ Delay: is the combination of all above optimizations with delaying the response from the WPQ in MC for loads hitting in the WPQ. There is no observable increase in the run-time overhead (see Section IX-A for the reason).

Pruning (cWSP): uses all above optimizations along with checkpoint pruning, lowering the average run-time overhead to only 6%. This technique dramatically reduces the overheads of certain applications, *e.g.,* `water-ns` and `LULESH`.

C. Run-Time Overhead Analysis for CXL-Based NVM

TABLE I: CXL memory devices

Device	CXL IP	Memory Technology	Max. Bandwidth	Latency (read/write)
CXL-A (NVDIMM)	Hard IP	DDR5-4800	38.4GB/s	158ns/120ns
CXL-B (NVDIMM)	Hard IP	DDR4-2400	19.2GB/s	223ns/139ns
CXL-C (NVDIMM)	Soft IP	DDR4-3200	25.6GB/s	348ns/241ns
CXL-D (PMEM)	Simulation	Intel Optane	6.6GB/s for read 2.3GB/s for write	245ns/160ns

To showcase the scalability of cWSP for the future far CXL-based NVM, we model three CXL NVDIMMs (CXL-A to CXL-C) in our simulator with the parameters from a recent empirical analysis of CXL DRAM memory [120]. Additionally, we model another CXL PMEM (CXL-D) by adding 70*ns* CXL interconnect latency [74] to the existing PMEM technology [127]. We keep all other parameters the same as those listed in Section IX, except for the latency and bandwidth parameters of the CXL-based NVM.

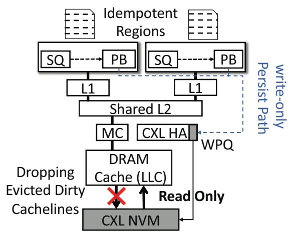

Fig. 16: cWSP architecture for CXL-based NVM; local DRAM served as an LLC atop the NVM

Figure 16 depicts the high-level architecture of cWSP where local DRAM works as an LLC atop of CXL-based NVM. Note that the persistence domain just moves from the battery-backed WPQ of conventional MC—as shown in Figure 3 (b)—to the one of CXL Home Agent (HA)[6], keeping the persist path length technically the same. The implication of the battery-backed WPQ of the HA is that the data being stored become persistent as soon as they arrive in the WPQ. In other words, on power failure occurs, data buffered in the WPQ are ensured to be flushed to the CXL-based NVM through the internal buffers along the way, *i.e.,* cWSP does not have to pay for the long latency of traveling from the HA to the CXL-based NVM. With the help of the same persist path length, cWSP maintains high performance for such a deep cache hierarchy.

[6]It controls the communication between the processor core and the CXL-based NVM, *e.g.,* translating load/store requests into PCIe transactions [22].

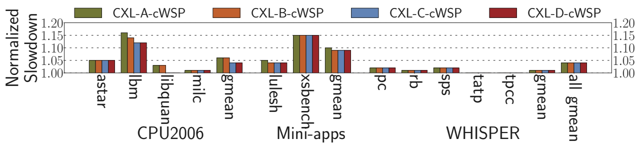

Fig. 17: Normalized slowdown of cWSP to the baseline (original program on CXL devices without crash consistency support) with varying CXL configuration; lower is better

Figure 17 shows the normalized slowdown of cWSP across selected memory intensive applications with varying CXL devices. The memory footprints of those program range from 2.5GB to 6GB. Notably, cWSP maintains an average of only 4% run-time overhead, regardless of the speed of the underlying CXL memory. Intriguingly, cWSP exhibits a slightly higher overhead with faster CXL memory. This is because cWSP benefits less from the speed enhancement in comparison to the baseline—primarily due to store persistence, leading to a higher normalized overhead. This trend aligns with the observation made with fast NVM technology, as detailed in Section IX-M. Note that cWSP incurs higher overheads for some applications, *e.g.,* `lbm` and `XSBench`, due to more RBT overflow caused by their shorter regions; see Section IX-E.

D. Comparison to Partial-System Persistence

To highlight the benefits of enabling DRAM as a cache, we implement an optimized version of BBB [3] that behaves as an ideal PSP scheme like Intel eADR with DRAM disabled. We believe that this ideal PSP attains the performance akin to LightPC [73], a system that replaces DRAM with slower PCM RAM and relies on the modified OS to flush the entire volatile data to NVM right before power failure. We then compare cWSP to this ideal PSP scheme across those selected memory intensive applications; their L2 miss rates range from 20% to 100%. Figure 18 shows that cWSP incurs an average of only 3% run-time overhead, thanks to enabling the DRAM cache. In contrast, the ideal PSP scheme causes a substantial

52% performance slowdown on average in that every single memory operation must access slower NVM.

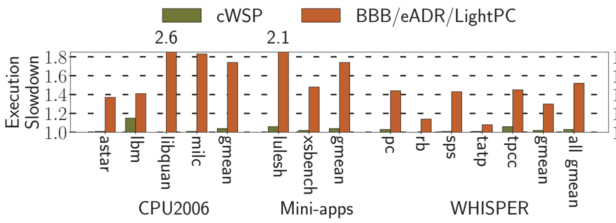

Fig. 18: Normalized slowdown of cWSP and the ideal PSP (BBB/eADR/LightPC) to the baseline; lower is better

E. Region Characteristics

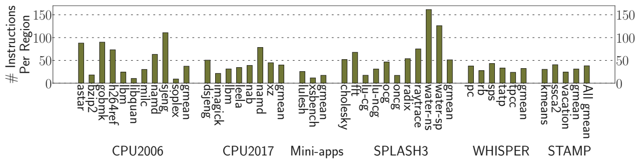

Fig. 19: Average number of instructions in regions

Given the critical role of idempotent region size in influencing power failure recovery time and the efficiency of the asynchronous store persistence, we collect the number of dynamic instructions in each region and report the average numbers in Figure 19. It shows that there are 38.15 instructions in each region on average, which signifies cWSP's fast failure recovery. Furthermore, with a 16-entry RBT, cWSP overlaps the long persistence latency of the oldest unpersisted region with the execution latency of 572 (16x38.15) instructions.

F. Impact of Deeper Cache Hierarchy

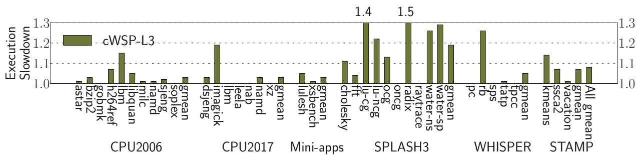

Fig. 20: cWSP's slowdown with L3 cache

To show how cWSP performs for a deeper cache hierarchy, *i.e.,* a 3-level SRAM cache atop DRAM cache, we add a shared 16-way set-associative writeback L3 cache of 44-cycle hit latency to both cWSP and the baseline. We also change the existing L2 cache in Figure 3 (b) to a private 8-way set-associative L2 with 1MB and 14-cycle hit latency. Figure 20 depicts that cWSP still incurs a low run-time overhead, *i.e.,* only 8% on average, thanks to the efficient asynchronous store persistence; see Section IX-H for details.

G. Impact of Persist Path Bandwidth

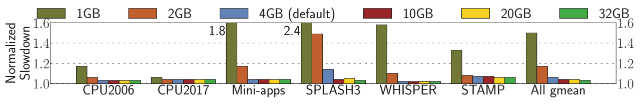

Fig. 21: cWSP's slowdown with varying persist path bandwidth from 1GB/s up to 32GB/s; lower is better

Since cWSP persists stores to NVM through the persist path, its bandwidth plays a key role in determining the overall performance. To explore the impact of persist path bandwidth on cWSP's performance, we conduct experiments with varying persist path bandwidth, from 1GB/s up to 32GB/s, as shown in Figure 21. The key trend is that the run-time overhead of cWSP decreases as the bandwidth rises. Thanks to cWSP's 8-byte persist granularity, cWSP's run-time overhead remains the same once the bandwidth rises beyond 10GB/s, confirming cWSP's low demand for persist path bandwidth.

H. Sensitivity to Region Boundary Table (RBT) Size

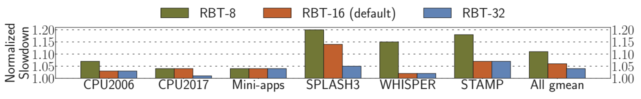

Fig. 22: cWSP's normalized slowdown with varying RBT size

The region boundary table (RBT) is so critical that its size highly affects how frequently the core pipeline stalls; the core pipeline stalls at a region boundary if RBT is full. We systematically vary RBT size from 8 to 32 to assess cWSP's performance. As shown in Figure 22, cWSP's run-time overhead rises to 11% on average and up to 20% for SPLASH3, when setting RBT size to 8. This is because the regions of SPLASH3 program are relatively short, leading to more pipeline stalls for a 8-entry RBT. Here, cWSP's run-time overhead decreases to only 4% with a 32-entry RBT.

I. Sensitivity to Persist Path Latency

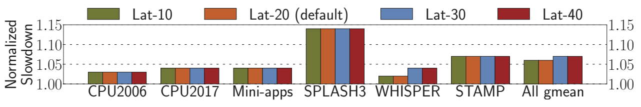

Fig. 23: cWSP's slowdown with varying persist path latency from 10*ns* to 40*ns*

To show the impact of persist path latency on cWSP's performance, we vary the persist path latency from 10*ns* to 40*ns*. Figure 23 shows that the persist path latency can be almost entirely overlapped by region execution, even if the latency increases up to 40*ns*, thanks to the efficient asynchronous store persistence enabled by RBT. Notably, SPLASH3 exhibits a higher run-time overhead caused by more frequent NVM writes; please refer to Section IX-A for details.

J. Sensitivity to Write Buffer (WB) Size

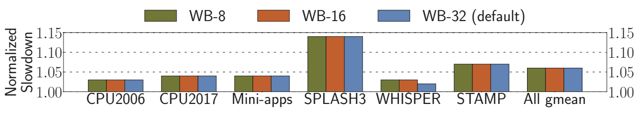

Fig. 24: cWSP's slowdown with varying L1D's WB size

To show the impact of delaying dirty cacheline writeback from the L1D's WB, we conduct a series of simulations with varying the WB size. Figure 24 depicts that cWSP's overhead remains the same no matter how small the WB is owing to the faster enough persist path; refer to Section V-A for details.

K. Sensitivity to Persist Buffer (PB) Size

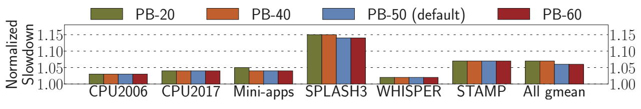

Fig. 25: cWSP's slowdown with varying PB size

As a critical component, the PB should be large enough so as not to congest the persist path frequently. Figure 25 shows that cWSP's performance is insensitive to PB size. Here, cWSP's overhead rises to only 7% even if the PB size is 20, thanks to the asynchronous store persistence; cWSP sets default PB size to 50 for maximal performance.

L. Sensitivity to NVM WPQ Size

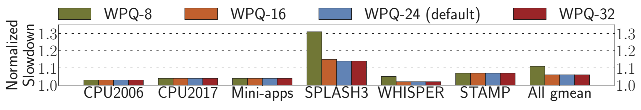

Fig. 26: cWSP's slowdown with varying WPQ size

As a shared component among multiple cores, NVM WPQ should be appropriately sized to keep the pressure on it low. Figure 26 shows that a 24-entry WPQ is large enough maintain cWSP's low run-time overhead; it is cheap to scale the WPQ size to 24 owing to its 8-byte granularity. As the WPQ size decreases to 8, cWSP still incurs a moderate run-time overhead, *i.e.,* 11% on average. Notably, cWSP incurs an up to 31% overhead for SPLASH3 due to a high pressure on the WPQ caused by its frequent NVM writes.

M. Sensitivity to NVM Technology

To analyze how NVM technologies affect the performance of cWSP, we evaluate cWSP for 3 NVM technologies: PMEM [126], STT-MRAM [15], and ReRAM [14]. Figure 27 shows that cWSP maintains its low overhead (8%), regardless of the NVM technique. Note that cWSP incurs a marginally elevated overhead with fast NVM techniques, *e.g.,* ReRAM. This phenomenon arises from the fact that cWSP benefits less from faster NVM techniques than the baseline—due to the store persistence, resulting in a higher normalized overhead; the same phenomenon appears for faster CXL devices (see Section IX-C).

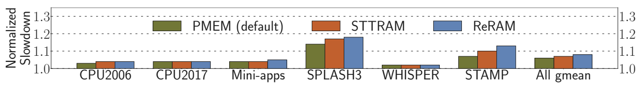

Fig. 27: cWSP's slowdown with varying NVM technologies

N. Hardware Overhead

cWSP incurs only a storage overhead of 176 bytes for the 16-entry RBT whose entry size is 11 bytes (see Figure 9). Note that cWSP does not incur hardware overhead for the PB since it can be covered by the Intel 1KB write-combing buffer (WCB) [27]. We also use CACTI [94] with 22*nm* technology to calculate the hardware overhead of the RBT. The calculation results turn out that the RBT costs only 0.001mm^2.

X. Other Related Work

In general, there are two types of application-level crash consistency schemes that are implemented by software: (1) failure-atomic sections (FASEs) protected by the outermost pair of lock and unlock as in iDO [78] and (2) persistent transactions such as Clobber-NVM [125] and LAD [44]. On one hand, iDO achieves correct power failure recovery using idempotent processing and live-out register checkpointing. However, it incurs a high run-time overhead due to introducing persist barriers at each region boundary. On the other hand, Clobber-NVM undo logs program stores as with other transaction-based schemes yet in a more intelligent way. That is, rather than undo logging every store in a transaction, Clobber-NVM does only antidependent therein—since others are to be reinitialized during the re-execution of power-interrupted transaction. In a sense, Clobber-NVM resembles the MC speculation of cWSP in that it also undo-logs cross-idempotent-region stores that might be antidependent on some prior region's loads. However, Clobber-NVM still suffers from persist barrier cost between the transactional store and its log stores. LAD [44] uses a hardware redo buffer in MC to log memory updates of transactions. Unfortunately, LAD needs to fall back to undo logging upon full redo buffer. Moreover, LAD suffers from frequent core pipeline stalls, *i.e.,* 163 cycles on average, at end of short each transaction; its size is limited to the redo buffer size.

LightPC [73] and Zhuque [47] offer crash consistency and persistence at the process level. They are inferior to cWSP for 3 reasons: (1) requiring extensive modifications on the OS source code or C library, (2) having poor performance due to the inability to enable DRAM; LightPC uses PCM RAM, while Zhuque maps the memory objects of user processes to PMEM space, and (3) consuming a lot of energy to dump entire volatile states to NVM upon power loss as with the pioneering work on WSP [96].

XI. Conclusion

This paper presents cWSP, a compiler-directed whole-system persistence (WSP) scheme. cWSP compiler partitions input program into a series of idempotent regions so that the program can correctly recover from power failure by re-executing the oldest unpersisted region. During the execution of the idempotent regions, cWSP architecture persists their data being stored in a performant way without breaking the recovery guarantee. Experimental results with 37 applications from SPEC CPU2006/2017/DOE Mini-apps/SPLASH3/WHISPER/STAMP highlight the low run-time overhead of cWSP, *i.e.,* 6% on average, achieving a 4.5x reduction compared to that of the state-of-the-art work.

Acknowledgment

We thank anonymous reviewers and our shepherd for their valuable comments. This work was supported by NSF grants 2001124 (CAREER), 2153749, and 2314681.

References

[1] S Aggarwal, H Almasi, M DeHerrera, B Hughes, S Ikegawa, J Janesky, HK Lee, H Lu, FB Mancoff, K Nagel, et al. Demonstration of a reliable 1 gb standalone spin-transfer torque mram for industrial applications. In *2019 IEEE International Electron Devices Meeting (IEDM)*, pages 2–1. IEEE, 2019.

[2] Hiroyuki Akinaga and Hisashi Shima. Resistive random access memory (reram) based on metal oxides. *Proceedings of the IEEE*, 98(12):2237–2251, 2010.

[3] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. Bbb: Simplifying persistent programming using battery-backed buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 111–124. IEEE, 2021.

[4] Igor Arsovski and Reid Wistort. Self-referenced sense amplifier for across-chip-variation immune sensing in high-performance content-addressable memories. In *IEEE Custom Integrated Circuits Conference 2006*, pages 453–456. IEEE, 2006.

[5] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.

[6] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. Memory hierarchy for web search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 643–656. IEEE, 2018.

[7] Rahul Bera, Konstantinos Kanellopoulos, Shankar Balachandran, David Novo, Ataberk Olgun, Mohammad Sadrosadat, and Onur Mutlu. Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–18. IEEE, 2022.

[8] Abhishek Bhattacharyya, Abhijith Somashekhar, and Joshua San Miguel. Nvmr: non-volatile memory renaming for intermittent computing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 1–13, 2022.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[10] James Bucek, Klaus-Dieter Lange, et al. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42. ACM, 2018.

[11] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10):433–452, 2014.

[12] Yangyin Chen. Reram: History, status, and future. *IEEE Transactions on Electron Devices*, 67(4):1420–1433, 2020.

[13] Ping Chi, Shuangchen Li, Yuanqing Cheng, Yu Lu, Seung H Kang, and Yuan Xie. Architecture design with stt-ram: Opportunities and challenges. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 109–114. IEEE, 2016.

[14] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. *ACM SIGARCH Computer Architecture News*, 44(3):27–39, 2016.

[15] Ping Chi, Cong Xu, Tao Zhang, Xiangyu Dong, and Yuan Xie. Using multi-level cell stt-ram for fast and energy-efficient local checkpointing. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 301–308. IEEE, 2014.

[16] Jongouk Choi, Hyunwoo Joe, and Changhee Jung. Capos: Capacitor error resilience for energy harvesting systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4539–4550, 2022.

[17] Jongouk Choi, Hyunwoo Joe, Yongjoo Kim, and Changhee Jung. Achieving stagnation-free intermittent computation with boundary-free adaptive execution. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 331–344. IEEE, 2019.

[18] Jongouk Choi, Larry Kittinger, Qingrui Liu, and Changhee Jung. Compiler-directed high-performance intermittent computation with power failure immunity. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 40–54. IEEE, 2022.

[19] Jongouk Choi, Qingrui Liu, and Changhee Jung. Cospec: Compiler directed speculative intermittent computation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 399–412, 2019.

[20] Jongouk Choi, Jianping Zeng, Dongyoon Lee, Changwoo Min, and Changhee Jung. Write-light cache for energy harvesting systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[21] 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 ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146, 2009.

[22] CXL Consortium. Compute express link: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/, 2023.

[23] Intel Corporation. Persistent memory programming. https://pmem.io.

[24] Intel Corporation. Deprecating the pcommit instruction. https://www.intel.com/content/www/us/en/developer/articles/technical/deprecate-pcommit-instruction.html, 2016.

[25] Intel Corporation. Memory performance in a nutshell. https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html, 2016.

[26] Intel Corporation. eadr: New opportunities for persistent memory applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html, 2021.

[27] Intel Corporation. Intel® 64 and ia-32 architectures optimization reference. https://cdrdv2-public.intel.com/671488/248966-046A-software-optimization-manual.pdf, 2023.

[28] Microsoft Corporation. Intel optane dc persistent memory, azure netapp files, and azure ultra disk for sap hana. https://azure.microsoft.com/en-us/blog/intel-optane-dc-persistent-memory-azure-netapp-files-and-more-for-sap-hana/.

[29] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 271–282, 2018.

[30] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. Lazy release persistency. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1173–1186, 2020.

[31] Marc De Kruijf and Karthikeyan Sankaralingam. Idempotent processor architecture. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 140–151. IEEE, 2011.

[32] Marc De Kruijf and Karthikeyan Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12. IEEE, 2013.

[33] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, 37(2):52–62, 2017.

[34] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *Acm sigplan notices*, 47(4):37–48, 2012.

[35] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. A case for specialized processors for scale-out workloads. *IEEE Micro*, 34(3):31–42, 2014.

[36] Free Software Foundation. Gnu c standard library. https://www.gnu.org/software/libc/.

[37] LLVM Foundation. compiler-rt runtime libraries. https://compiler-rt.llvm.org/.

[38] LLVM Foundation. A llvm-compatible unwinder. https://bcain-llvm.readthedocs.io/projects/libunwind/en/latest/.

[39] LLVM Foundation. A new c++ standard library for llvm. https://libcxx.llvm.org/.

[40] Kaan Genç, Michael D Bond, and Guoqing Harry Xu. Crafty: Efficient, htm-compatible persistent transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–74, 2020.

[41] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. Relaxed persist ordering using strand persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 652–665. IEEE, 2020.

[42] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Yashme: detecting persistency races. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 830–845, 2022.

[43] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 913–928, 2019.

[44] Siddharth Gupta, Alexandros Daglis, and Babak Falsafi. Distributed logless atomic durability with persistent memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 466–478, 2019.

[45] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[46] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[47] George Hodgkins, Yi Xu, Steven Swanson, and Joseph Izraelevitz. Zhuque: Failure is not an option,{it's} an exception. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 833–849, 2023.

[48] Ali Hoseinghorban, Amir Mahdi Hosseini Hosseini Monazzah, Mostafa Bazzaz, Bardia Safaei, and Alireza Ejlali. Coach: Consistency aware check-pointing for nonvolatile processor in energy harvesting systems. *IEEE Transactions on Emerging Topics in Computing*, 2019.

[49] 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*, pages 468–482, 2017.

[50] Yiming Huai et al. Spin-transfer torque mram (stt-mram): Challenges and prospects. *AAPPS bulletin*, 18(6):33–40, 2008.

[51] Shao-Yu Huang, Jianping Zeng, Xuanliang Deng, Sen Wang, Ashrarul Sifat, Burhanuddin Bharmal, Jia-Bin Huang, Ryan Williams, Haibo Zeng, and Changhee Jung. Rtailor: Parameterizing soft error resilience for mixed-criticality real-time systems. In *2023 IEEE Real-Time Systems Symposium (RTSS)*, pages 344–357. IEEE, 2023.

[52] Advanced Micro Devices Inc. Amd epyc 9654p. https://www.amd.com/en/products/cpu/amd-epyc-9654p, 2023.

[53] Advanced Micro Devices Inc. Amd epyc 9754. https://www.amd.com/en/products/cpu/amd-epyc-9754, 2023.

[54] Intel. Intel optane dc persistent memory quick start guide. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf, June 2020.

[55] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *ACM SIGARCH Computer Architecture News*, 44(2):427–442, 2016.

[56] Jungi Jeong, Jaewan Hong, Seungryoul Maeng, Changhee Jung, and Youngjin Kwon. Unbounded hardware transactional memory for a hybrid dram/nvm memory system. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 525–538. IEEE, 2020.

[57] Jungi Jeong and Changhee Jung. Pmem-spec: persistent memory speculation (strict persistency can trump relaxed persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 517–529, 2021.

[58] Jungi Jeong, Jianping Zeng, and Changhee Jung. Capri: Compiler and architecture support for whole-system persistence. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pages 71–83, 2022.

[59] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient persist barriers for multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 660–671, 2015.

[60] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Dhtm: Durable hardware transactional memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 452–465. IEEE, 2018.

[61] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo 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. IEEE, 2017.

[62] Asim Kadav, Matthew J Renzelmann, and Michael M Swift. Fine-grained fault tolerance using device checkpoints. 48(4):473–484, 2013.

[63] Vasileios Karakostas, Osman S Unsal, Mario Nemirovsky, Adrian Cristal, and Michael Swift. Performance analysis of the memory management unit under scale-out workloads. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2014.

[64] Ian Karlin, Jeff Keasler, and J Robert Neely. Lulesh 2.0 updates and changes. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2013.

[65] AV Khvalkovskiy, Dmytro Apalkov, S Watts, Roman Chepulskii, RS Beach, Adrian Ong, X Tang, A Driskill-Smith, WH Butler, PB Visscher, et al. Basic principles of stt-mram cell operation in memory arrays. *Journal of Physics D: Applied Physics*, 46(7):074001, 2013.

[66] Hongjune Kim, Jianping Zeng, Qingrui Liu, Mohammad Abdel-Majeed, Jaejin Lee, and Changhee Jung. Compiler-directed soft error resilience for lightweight gpu register file protection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 989–1004, 2020.

[67] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. Pactree: A high performance persistent range index using pac guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 424–439, 2021.

[68] Andreas Köhler and Lorenz Erdmann. Expected environmental impacts of pervasive computing. *Human and Ecological Risk Assessment*, 10(5):831–852, 2004.

[69] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. Language-level persistency. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 481–493. IEEE, 2017.

[70] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M Chen, and Thomas F Wenisch. Delegated persist ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[71] R Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 335–349, 2020.

[72] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[73] Sangwon Lee, Miryeong Kwon, Gyuyoung Park, and Myoungsoo Jung. Lightpc: hardware and software co-design for energy-efficient full system persistence. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 289–305, 2022.

[74] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[75] Lin Li, Vijay Degalahal, Narayanan Vijaykrishnan, Mahmut Kandemir, and Mary Jane Irwin. Soft error and energy consumption interactions: A data cache perspective. In *Proceedings of the 2004 international symposium on Low power electronics and design*, pages 132–137, 2004.

[76] Fujitsu Semiconductor Memory Solution Limited. https://www.fujitsu.com/jp/group/fsm/en/products/reram/ReRAM_whitepaper_2023e.pdf, 2023.

[77] Gang Liu, Kenli Li, Zheng Xiao, and Rujia Wang. Ps-oram: Efficient crash consistency support for oblivious ram on nvm. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 188–203, 2022.

[78] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270. IEEE, 2018.

[79] Qingrui Liu and Changhee Jung. Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6. IEEE, 2016.

[80] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Clover: Compiler directed lightweight soft error resilience. *ACM Sigplan Notices*, 50(5):1–10, 2015.

[81] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 228–239, 2016.

[82] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 25. IEEE Press, 2016.

[83] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Compiler-directed soft error detection and recovery to avoid due and sdc via tail-dmr. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(2):32, 2017.

[84] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. Pmfuzz: test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 487–502, 2021.

[85] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1187–1202, 2020.

[86] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 411–425, 2019.

[87] Yongpan Liu, Zewei Li, Hehe Li, Yiqun Wang, Xueqing Li, Kaisheng Ma, Shuangchen Li, Meng-Fan Chang, Sampson John, Yuan Xie, et al. Ambient energy harvesting nonvolatile processors: from circuit to system. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[88] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, et al. Scale-out processors. *ACM SIGARCH Computer Architecture News*, 40(3):500–511, 2012.

[89] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. Architecture exploration for ambient energy harvesting nonvolatile processors. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 526–537. IEEE, 2015.

[90] Virendra J Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[91] Amirsaman Memaripour and Steven Swanson. Breeze: User-level access to non-volatile main memories for legacy software. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 413–422. IEEE, 2018.

[92] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46. IEEE, 2008.

[93] Sunil Kumar Mohapatra, Priyadarshini Nayak, Sushruta Mishra, and Sukant Kishoro Bisoy. Green computing: a step towards eco-friendly computing. In *Emerging trends and applications in cognitive computing*, pages 124–149. IGI global, 2019.

[94] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 1:1–24, 2009.

[95] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. *ACM SIGPLAN Notices*, 52(4):135–148, 2017.

[96] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–410, 2012.

[97] Ian Neal, Andrew Quinn, and Baris Kasikci. Hippocrates: healing persistent memory bugs without doing any harm. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–414, 2021.

[98] Robert HB Netzer and Barton P Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[99] Tri M Nguyen and David 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. IEEE, 2018.

[100] Anant Vithal Nori, Jayesh Gaur, Siddharth Rai, Sreenivas Subramoney, and Hong Wang. Criticality aware tiered cache hierarchy: A fundamental relook at multi-level cache hierarchies. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 96–109. IEEE, 2018.

[101] Trail of Bits. Remill: a static binary translator from machine code instructions to llvm bitcode. https://github.com/lifting-bits/remill, 2024.

[102] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 265–276. IEEE, 2014.

[103] Ivy B Peng, Maya B Gokhale, and Eric W Green. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*, pages 304–315, 2019.

[104] Jens F Peters, Manuel Baumann, Benedikt Zimmermann, Jessica Braun, and Marcel Weil. The environmental impact of li-ion batteries and the role of key parameters–a review. *Renewable and Sustainable Energy Reviews*, 67:491–506, 2017.

[105] Shashank Priya and Daniel J Inman. *Energy harvesting technologies*, volume 21. Springer, 2009.

[106] Moinuddin K Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *2009 42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*, pages 14–23. IEEE, 2009.

[107] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 392–407, 2021.

[108] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutiu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 672–685. IEEE, 2015.

[109] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111. IEEE, 2016.

[110] Samsung. Samsung electronics unveils industry's highest-capacity 12nm-class 32gb ddr5 dram, ideal for the ai era. https://news.samsung.com/global/samsung-electronics-unveils-industrys-first-and-highest-capacity-12nm-class-32gb-ddr5-dram-ideal-for-the-ai-era, 2023.

[111] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: Memory allocation for nvram. *Adms@ Vldb*, 15:61–72, 2015.

[112] Tingting Sha, Milo MK Martin, and Amir Roth. Scalable store-load forwarding via store queue index prediction. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 12–pp. IEEE, 2005.

[113] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 205–218, 2020.

[114] Sara Mahdizadeh Shahri, Seyed Armin Vakil Ghahani, and Aasheesh Kolli. (almost) fence-less persist ordering. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 539–554. IEEE, 2020.

[115] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 141–152, 2015.

[116] 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*, pages 178–190, 2017.

[117] Livio Soares and Michael Stumm. {FlexSC}: Flexible system call scheduling with {Exception-Less} system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[118] Rick Stevens, Jini Ramprakash, Paul Messina, Michael Papka, and Katherine Riley. Aurora: Argonne's next-generation exascale supercomputer. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2019.

[119] Sam S Stone, Kevin M Woley, and Matthew I Frank. Address-indexed memory disambiguation and store-to-load forwarding. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 12–pp. IEEE, 2005.

[120] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–121, 2023.

[121] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[122] Vineet Veer Tyagi and DPCM Buddhi. Pcm thermal storage in buildings: A state of art. *Renewable and sustainable energy reviews*, 11(6):1146–1166, 2007.

[123] Ranjan Sarpangala Venkatesh, Tony Mason, Pradeep Fernando, Greg Eisenhauer, and Ada Gavrilovska. Scheduling hpc workflows with intel optane persistent memory. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 56–65. IEEE, 2021.

[124] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News*, 39(1):91–104, 2011.

[125] Yi Xu, Joseph Izraelevitz, and Steven Swanson. Clobber-nvm: log less, re-execute more. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 346–359, 2021.

[126] Sujay Yadalam, Nisarg Shah, Xiangyao Yu, and Michael Swift. Asap: A speculative approach to persistence. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[127] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 169–182, 2020.

[128] Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay Paddayuru Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. Replaycache: Enabling volatile cachesfor energy harvesting systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 170–182, 2021.

[129] Jianping Zeng, Shao-Yu Huang, Jiuyang Liu, and Changhee Jung. Soft error resilience at near-zero cost. In *2024 38th ACM International Conference on Supercomputing (ICS)*. ACM, 2024.

[130] Jianping Zeng, Jungi Jeong, and Changhee Jung. Persistent processor architecture. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1075–1091, 2023.

[131] Jianping Zeng, Hongjune Kim, Jaejin Lee, and Changhee Jung. Turnpike: Lightweight soft error resilience for in-order cores. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 654–666, 2021.

[132] Jin Zha, Linpeng Huang, Linzhu Wu, Sheng-an Zheng, and Hao Liu. A consistency mechanism for nvm-based in-memory file systems. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 197–204, 2016.

[133] Yida Zhang and Changhee Jung. Featherweight soft error resilience for gpus. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 245–262. IEEE, 2022.

[134] Yuchen Zhou, Jianping Zeng, Jungi Jeong, Jongouk Choi, and Changhee Jung. Sweepcache: Intermittence-aware cache on the cheap. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1059–1074, 2023.