Capri: Compiler and Architecture Support for Whole-System Persistence

Jungi Jeong*
Purdue University
West Lafayette, Indiana, USA
jungijeong@purdue.edu

Jianping Zeng
Purdue University
West Lafayette, Indiana, USA
zeng207@purdue.edu

Changhee Jung
Purdue University
West Lafayette, Indiana, USA
chjung@purdue.edu

ABSTRACT

This paper investigates whole-system persistence (WSP) that ensures hassle-free crash consistency for all programs while simultaneously leveraging both advantages of the non-volatile memory technologies: high-density and in-memory persistence. Despite the promising characteristics, there are two challenges that must be addressed to make WSP a reality. First, programs must be able to resume the execution from where they had a failure. Second, failure recovery must be offered to any program including the OS in a transparent manner while minimizing persistence overheads.

To this end, this paper presents Capri, a compiler and architecture co-designed scheme for region-level whole-system persistence. First, the Capri compiler partitions program into a series of regions whose boundaries serve as recovery points. Then, the Capri architecture provides the regions with crash consistency through hardware-based atomic updates. Finally, with the novel interplay between the architecture and the compiler, Capri provides failure atomicity on the cheap, i.e., with 0%, 12.4%, and 9.1% performance overheads for SPEC CPU2017, STAMP, and Splash3 benchmarks, respectively.

CCS CONCEPTS

• **Hardware → Emerging architectures**; *Emerging languages and compilers.*

KEYWORDS

whole-system persistence, compiler/architecture co-design

ACM Reference Format:
Jungi Jeong, Jianping Zeng, and Changhee Jung. 2022. Capri: Compiler and Architecture Support for Whole-System Persistence. In *Proceedings of the 31st Int'l Symposium on High-Performance Parallel and Distributed Computing (HPDC '22), June 27-July 1, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3502181.3531474

1 INTRODUCTION

Advanced non-volatile memory (NVM) technologies, such as Intel's Optane PMem [3], provide both high-density and in-memory persistence, realizing the full potential to unify the main memory and storage devices. This leads to the advent of persistent memory programming for so-called partial-system persistence (PSP), e.g., Optane's *app-direct mode*, where DRAM is used as the main memory while NVM serves as a persistent heap. In PSP, programmers must delineate a piece of code which requires crash consistency and explicitly manage both volatile and non-volatile data (objects) using dedicated interfaces such as `pmalloc` or persistent transactions [11, 17–19, 26, 29, 46, 78, 79]. However, the persistent memory programming is difficult and often necessitates custom data structure design and application-specific recovery code tailored to particular data structures [28, 31, 42, 49, 65, 70], thus being limited to a small set of programs such as in-memory index structures/databases or key-value stores [4, 5, 12, 30, 38, 40, 41, 43, 56, 60, 77, 82].

1.1 Motivation

Unfortunately, this limitation hinders most users from readily taking advantage of both high-density and in-memory persistence of NVM. With that in mind, as an alternative to the *app-direct mode*, Intel proposes a *memory mode*, where DRAM is vertically integrated as a cache on top of NVM. In particular, since NVM here is used as the high-density yet volatile main memory [3], it does not provide in-memory persistence at all. This implies that in the memory mode, users have no choice but to put up with data loss in case of power failure, unless they resort to the app-direct mode at the expense of the persistent memory programming difficulty.

To solve the problems, this paper studies whole-system persistence (WSP) that simultaneously enables high-density and in-memory persistence and satisfies the following requirements. First, WSP must be able to restore the entire system on failure recovery *no matter how deep the volatile cache and memory hierarchy is* as in off-chip DRAM cache as Optane's *memory mode*. Second, failure recovery should be offered *to any programs* (instead of being limited to in-memory databases and key-value stores) *in a software-transparent manner*, which is desired as a variety of recent works confirm that persistent programming is error-prone [20, 57–61, 67].

1.2 Limitation of state-of-the-art approaches

The key approach to achieving whole-system persistence is to flush all data in volatile media (e.g., register files, CPU caches, and DRAM) into non-volatile memory before the impending power failure. For example, Narayanan et al. proposed to use residual energy and persist all volatile status when power is about to be cut off [66]. Similarly, Intel recently announced extended ADR (eADR) support that includes on-chip caches in the persistent domain [1]. However, it turns out that eADR must secure an excessive amount of residual energy [7] to persist the deep cache hierarchy of HPC manycore processors, which gets worse for the off-chip DRAM cache as in the memory mode of Intel Optane. Apart from that, eADR does

*Now at Google.

HPDC '22, June 27-July 1, 2022, Minneapolis, MN, USA

ACM ISBN 978-1-4503-9199-3/22/06.
https://doi.org/10.1145/3502181.3531474

not protect other volatile states such as register files and internal buffers in the processor pipeline. This limitation makes it practically impossible to realize whole-system persistence at low cost.

1.3 Overall Design

To this end, this paper proposes Capri, a compiler and architecture co-design scheme that achieves *region-level whole-system persistence*. Capri partitions programs into a series of regions, where each region boundary serves as a recovery point. For this purpose, the Capri compiler inserts region boundary instructions to delineate the *region formation*. Furthermore, it instruments register-checkpointing instructions required for resuming the power-interrupted programs from the last committed and persisted region boundaries after failure recovery. Then, all store instructions within a region are carefully handled by hardware, which leverages the *hardware non-volatile proxy buffer* as the safety net to prevent partial updates. The proxy buffer guarantees that modified data in a region are not released to the non-volatile main memory before the region is completed.

Capri's region formation and proxy buffer showcase a novel interplay between compiler and architecture, enabling whole-system persistence *without program changes* while *simultaneously leveraging high-density and in-memory persistence*. That is, the Capri compiler uses the number of stores as a region criterion, i.e., the resulting regions contain no more than the threshold number of stores therein (e.g., 256 by default, including both regular and checkpointing stores). This threshold determines the hardware proxy buffer size and prevents overflow, rendering hardware design simpler.

Challenge #1: Stale Read Prevention. Previous studies with two data paths to NVM (e.g., the regular data path through caches and the new persist path from proxy buffers) have decided to drop dirty cache blocks on their last-level cache eviction (from the regular path) to simplify the sequential persist order [10, 25, 34, 35, 45, 64, 71]. However, such a design decision complicates the NVM read operations—whenever searching data in NVM, it must look up the proxy buffer simultaneously, increasing both read latency and energy consumption. Such *indirect read* has been mitigated in various ways, e.g., HW bloom filter [35, 64], cache coherence [45], or speculation [34], but they all come at the cost of significant hardware and software complexity.

Solution #1: Undo+Redo Logging. Capri eliminates the indirect read problem by allowing NVM updates from the *both* dirty cache writeback and the proxy buffer. Therefore, memory loads and stores happen in the same way as the commodity architecture. However, dirty cache writeback may break correctness by persisting regions out of order (see Section 5.4). To preserve crash consistency with this distinctive architecture, Capri uses *undo+redo logging* that keeps data of both before and after the update. Thus, even if the dirty cache writeback persists regions out of order, the undo value (e.g., before the update) can safely restore the previous state across a power failure.

Challenge #2: Checkpoint Overheads. By its nature, whole-system persistence (in line with persistent memory programs [7, 8, 23, 25, 34, 45, 64, 71]) should come with performance overheads—compared to volatile execution—and complex hardware changes to control the persist order. For example, with WSP, all store instructions in a region must be reflected inside the non-volatile main memory before proceeding to the next region. Furthermore, register-checkpointing stores incur non-negligible pressure to NVM, leading to a substantial slowdown. Therefore, the primary design goal of Capri is to *lengthen the region size* as much as possible to lessen checkpointing stores. The longer regions are desirable since they reduce the number of checkpointing stores and unburden the pipeline by less dynamic instruction counts.

Solution #2: Compiler Optimizations. Capri reduces checkpoint overheads via 1) region size extension and 2) unnecessary checkpoints removal. First, we found that although a large number (e.g., 1k stores) is given as a threshold, many of the resulting regions contain fewer stores due to *short loops* [1] in programs (see Section 4.3 and Figure 11). In light of this, Capri presents *speculative loop unrolling* that unrolls the loop even if iteration counts are unavailable at compile time. In this way, the Capri compiler significantly extends the region sizes for short loops in programs. Second, Capri leverages existing compiler analysis to reduce checkpoint overheads further [39], i.e., it removes register-checkpointing stores if their register values can be reconstructed by other register values at recovery time. Finally, Capri rearranges checkpointing stores to prevent repeated checkpoints of the same register inside the loop body.

1.4 Experimental methodology and evaluation results

For evaluation, we used the full-system simulation mode of a cycle-accurate architecture simulator [9]. Notably, we re-compiled the entire Linux Kernel with our Capri compiler to include the OS in the whole-system persistence domain. Our experiments demonstrate that Capri accomplishes lightweight whole-system persistence only causing 0%, 12.4%, and 9.1% performance overheads in a geometric mean for SPEC2017, STAMP, and Splash3 benchmarks, respectively. Although a naive approach may slow down the benchmark up to 2X, our novel architecture and compiler interaction achieves very low performance overheads. Consequently, Capri makes it possible to accommodate all programs as first-class citizen in the world of persistence.

2 WHOLE-SYSTEM PERSISTENCE: DESIGN GOALS

2.1 SW-Transparent Failure-Atomicity

We pursue an entirely transparent approach that does not require program changes. The existing persistent applications in partial-system persistence need to be re-written. For example, programmers should identify which data (e.g., objects) to place in NVM and annotate them within transactions, such that the manually-crafted recovery protocol can restore them to the correct and consistent state. Unfortunately, ensuring failure-atomicity in persistent applications often leads to a substantial change in application design [60, 61]. Also, such modifications can be more error-prone [20, 57–61, 67], breaking the failure-atomicity guarantee. To

[1] The loop body limits region sizes since it is challenging to measure the exact dynamic store counts at the compile time if loop counts are unknown. See Section 4.3 for details.

make NVM programming more straightforward and less buggy, multiple studies have presented frameworks that abstract complex low-level techniques [24, 33, 44, 73] and spot potential buggy codes [20, 57–61, 67]. On the other hand, this study aims to *execute programs without changes, eliminating error-prone modifications with whole-system persistence.*

2.2 Whole-System Failure-Atomicity

The second goal of this study is to provide failure-atomicity for *all* applications by making the whole system persistent. Previous proposals for partial-system persistence have been limited for the following reasons. First, they protect and recover data only allocated in NVM specified by programmers. This partial persistence results in all volatile states in the register file, on-chip caches, and DRAM disappearing when a power failure happens. Second, therefore, the crash recovery necessitates restarting the program from the beginning. However, such convention limits its applicability to particular application domains such as in-memory databases [40, 43, 70] or key-value stores [12, 30, 38, 56, 77, 82], which are backed by durable data structures [31, 42, 49, 65]. On the other hand, this study targets whole-system persistence that *restores the entire program states from the register file to NVM* and *resumes them from where they were interrupted.* Furthermore, these benefits are generally available for even applications that do not have transactional semantics, such as general-purpose CPU applications or multi-threaded scientific workloads.

3 CAPRI: COMPILER/ARCHITECTURE CO-DESIGN FOR WHOLE-SYSTEM PERSISTENCE

Capri pursues whole-system persistence while leveraging both the high-density and non-volatility of NVM. That is, Capri targets the vertically integrated hybrid memory (e.g., Intel Optane's memory mode) that uses NVM as the main memory and DRAM as hardware-managed off-chip caches. This section explains how Capri guarantees region-level whole-system persistence without program changes by 1) compiler-directed region partitioning and 2) architecture-enforced region failure-atomicity.

3.1 Region-level Whole-System Persistence

Capri provides region-level persistence that partitions the program into a series of recoverable regions. At region boundaries, programs produce checkpoints containing the current PC offset and live-out register values (e.g., will be used in the following regions) so that the recovery procedure can correctly restore register values. Furthermore, data updated within a region by store instructions must be persisted in NVM to proceed to the next region. In the end, Capri restores the entire program state after a power failure using checkpointed register values and data in NVM. Then, it resumes programs from the beginning of the regions where they had a failure. Note that checkpointed register values and NVM data are sufficient to reconstruct the entire program state since the Capri architecture guarantees all store instructions in each region recoverable from a crash.

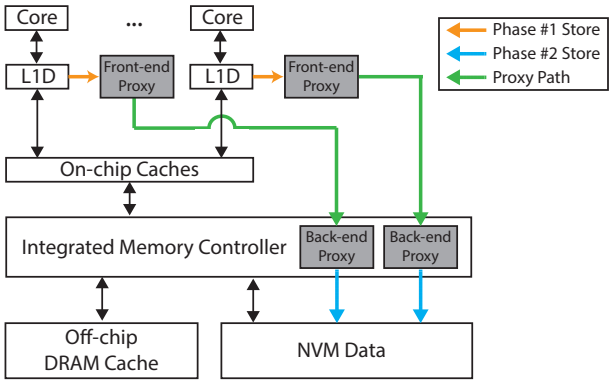


Figure 1: Capri architecture overview. Capri uses NVM as the main memory and DRAM as off-chip caches.

3.2 Compiler-directed Region Partitioning

The Capri compiler is responsible for placing region boundary instructions and register-checkpointing stores. In particular, it uses the number of stores as a region criterion to balance architecture and compiler efforts. For example, it places region boundaries such that the number of *dynamic* store instructions in each region does not exceed the given threshold (e.g., 256 stores). Then, the core hardware component in the Capri architecture—the non-volatile proxy buffer (see Section 5.2)—is configured based on the threshold used in the Capri compiler such that it prevents hardware overflow.

Along with the region boundary placement, the Capri compiler calculates the minimal register set required for the recovery. First, before proceeding to the next region, the Capri compiler performs static analysis over the control flow graph to identify live-in registers to the next region. Then, it generates register-checkpointing stores just like regular stores. In case of failure recovery, these checkpointed registers are reloaded to restore the live-in register set of the interrupted region.

3.3 Architecture-supported Region Failure-Atomicity

Given the region formation and register-checkpointing stores delineated by the Capri compiler, the Capri architecture ensures failure-atomic execution for each region. Unlike previous studies that assumed failure-atomic software [7, 8, 25, 34, 45, 64, 71], architecture should provide atomic execution in a software transparent manner. In particular, all stores in a region should become persistent in an all-or-nothing-based manner before the next region persists, allowing programs to restart from the beginning of the interrupt region after failure recovery.

For this purpose, Capri supports *two-phase atomic stores* similar to the hardware-based write-ahead logging approach [10, 22, 27, 35, 37, 68, 72]. Figure 1 shows Capri architecture and illustrates the two-phase atomic store strategy with *the decoupled proxy buffer architecture.* The front-end proxy buffer is placed alongside the L1 data cache, while the integrated memory controller contains the back-end proxy buffer. Note that both the front- and back-end proxy buffers are non-volatile (e.g., battery-backed SRAM buffers similar

to the previous study [7]). Having non-volatile proxy buffers is critical to reducing the gap between volatile and persistent memory.

The two-phase atomic store guarantees failure-atomicity with undo+redo logging. The undo+redo logging has an advantage over undo or redo logging strategies. The first phase (colored orange in Figure 1) generates proxy entries for all store instructions since whole-system persistence considers all data as persistent data. Once creating proxy entries of all stores in a region, the first phase completes. These proxy entries will travel through the proxy data path. This uncacheable and separate path that directly connects the front-end proxy to the per-core back-end in the memory controllers. The proxy buffer guarantees that any store instructions are not reflected on NVM until the first phase of atomic store completes. Therefore, it is the safety net to prevent partial updates visible after failure recovery. After the first phase completes, the second phase (colored blue) copies proxy entries from the proxy buffer to NVM data. Note that the Capri architecture ensures the first phase to happen before the NVM updates while the second phase can occur at any time after the first phase (see Section 5.1). Lastly, the Capri architecture guarantees the in-order persistence of regions.

Since the front- and back-end proxy buffers are non-volatile (e.g., battery-backed SRAMs), their contents are drained to NVM at the time of power interrupt. Later, the recovery processes use them to restore the consistent non-volatile main memory state (see Section 5.4).

Lastly, to the best of our knowledge, supporting non-recoverable operation such as I/O operations has been remained as the open problem. That being said, since the Capri compiler places a region boundary at the function calls, the function that implements I/O operations is treated as a separate region—though it cannot be recovered due to the I/O operation. We believe that Capri can simply handle I/O operations by checkpointing necessary status (e.g., before I/O operation starts). That way, the interruped I/O operation can be restarted after recovery.

4 CAPRI COMPILER

The Capri compiler provides the region formation, which is the building block of whole-system persistence in Capri. It primarily performs two tasks on the binary. First, the Capri compiler partitions the binary into fine-grained regions, considering the proxy buffer capacity used as a safety net of the atomic store release. Then, the compiler inserts checkpointing store instructions to preserve the live-out registers to NVM.

Although compiler-directed region formation and checkpointing stores are not new [15, 16, 32, 47, 52–55, 74, 84, 85], the interplay between compiler and architecture makes Capri stand out among the previous studies. In particular, the goal of the Capri compiler is to delineate regions as large as possible and to minimize NVM writes (as checkpointing stores amplify NVM writes) such that architecture ensures failure atomicity without significant performance loss.

4.1 Region Formation

At first glance, region formation appears to be trivial; for example, one can count the store instructions while traversing the control flow graph (CFG) and insert boundaries before it exceeds a given threshold. However, region formation is circularly dependent on calculating the checkpoint set since checkpoint store instructions are counted as regular stores. Thus, Capri cannot calculate the checkpoint set and its location before determining the region boundaries. As a result, placing region boundaries requires the binary containing register checkpoint stores, which in turn needs to identify the boundaries first.

To break the circular dependence, Capri uses the following heuristic. First, Capri considers all basic blocks in the CFG as initial regions and calculates the number of checkpoints to be instrumented in each region. In particular, Capri places a region boundary at all the entry/exit points of functions and the beginning of each loop header. Also, the Capri compiler considers memory fences and atomic operations as region boundaries since they are critical to guarantee correctness for multi-threaded programs [47, 55]. Then, Capri splits the basic blocks if they have region boundaries in the middle to ensure all regions starts at the beginning of basic blocks, which helps compute the next step.

Next, while traversing the CFG, Capri attempts to combine initial regions into larger regions as much as possible. By combining them, Capri can remove many checkpoint instructions (i.e., stores) because the registers can be no longer live-out to the later regions after combining. The region criterion ensures that the number of stores in each region will not overflow the threshold, including the checkpoint stores. Moreover, Capri does not insert cache-flush instructions (e.g., dccvap in ARM or clwb in X86).

4.2 Register-Checkpointing Stores

Capri leverages a novel compiler analysis to identify the minimal register state necessary for restoring a recovery point (i.e., the most recently committed region boundary) [53]. The checkpoint-set analysis investigates instructions that update registers and checkpoints the resulting value if used in later regions. In particular, the Capri compiler is interested in the last instructions that update the same registers. If the compiler identifies those instructions, it inserts checkpoint stores (i.e., regular store instructions with the register values as operands) immediately following them to store the updated values in a reserved memory location.

The checkpointed register values are used when recovering from a power failure. They have fixed destination addresses such that it is easily accessible through an index within an array during the recovery process. For this purpose, the Capri compiler allocates a global array where all registers have mapped into the dedicated slots. For example, r0 is mapped into the index zero. This global array checkpoint storage is feasible since the Capri compiler creates checkpoints for the architectural register, which is statically fixed in number. Please refer to Section 5.4 for the recovery protocol.

4.3 Extending Region with Speculative Loop Unrolling

The Capri compiler guarantees that all regions have at most as many stores as the threshold. However, this does not mean that regions will have the exact threshold number of stores. Instead, many regions have fewer stores than the threshold—because of *short loops* in programs. For example, the compiler makes conservative decisions when dealing with loops since it is challenging to measure

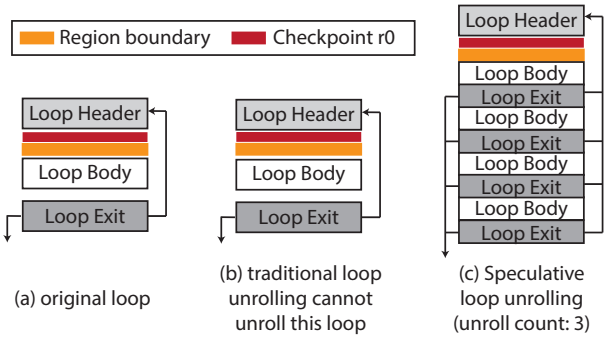


Figure 2: Capri speculatively unrolls the loop of static-unknown iteration count.

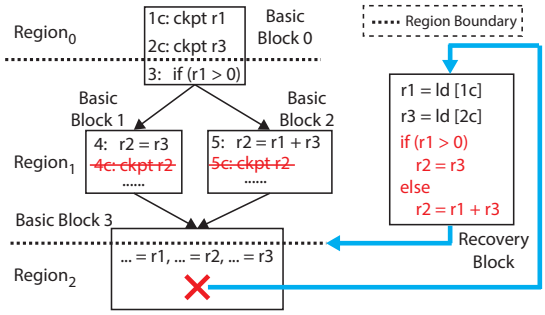


Figure 3: Capri prunes checkpoints if they can be reconstructed at the recovery time.

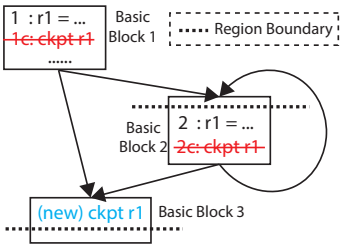


Figure 4: Moving checkpoint stores out of a loop to prevent repeated checkpoints of the same register.

the exact dynamic store counts at the compile time if loop counts are unknown. Therefore, it places a region boundary in the loop header, limiting the region size into the loop body, as shown in Figure 2(a). Furthermore, as a result, the value of register *r*0 holding the loop index variable is repeatedly checkpointed for every loop iteration.

However, such short regions do not match the goal of the region formation that pursues as large region sizes as possible to reduce persistence overheads. Capri aims to maximize the region length since it reduces the register-checkpointing stores and the region boundary instructions, which leads to unburdening the core pipeline and the memory pressure.

To that end, Capri presents a novel loop unrolling strategy, *speculative loop unrolling* that can be applied even if iteration counts are unknown at the compile time and thus can lengthen the region size. Speculative loop unrolling duplicates the loop body and loop exit condition simultaneously. Figure 2(c) illustrates the resulting code with speculative unrolling when the unrolling count is 3. Since the code size is extended with multiple loop bodies, the Capri compiler can insert region boundaries that produce longer lengths while enforcing the region threshold (i.e., without overflowing). As a result, with speculative loop unrolling, the Capri compiler forms almost 3x longer regions and reduces checkpointing stores for register *r*0 by 3x as well. On the other hand, the traditional loop unrolling cannot increase the region size if loop exit condition is not a static-known constant, as shown in Figure 2(b).

However, this speculative loop unrolling strategy does not benefit general purposes since it introduces complex control flow which can hurt other compiler optimizations, *e.g.,* instruction scheduling and pointer analysis. That being said, this speculative loop unrolling is particularly tailored for region-level whole-system persistence where the larger region comes with better performance. Our evaluation shows that the speculative loop unrolling improves benchmark performance significantly.

4.4 Redundant Checkpoint Stores Elimination

The Capri compiler checkpoints live-in registers of regions to the memory in case of failure recovery, which degrades the performance and induces more memory writes which is harmful especially for NVM [13, 21, 36, 63, 75, 76, 80, 83, 84, 86]. This section describes compiler techniques to reduce the number of checkpoint stores by 1) removing them if they can be reconstructed at the recovery process and 2) moving them out of a loop to prevent repeated checkpoints of the same register.

4.4.1 Optimal Checkpoint Pruning. Although the Capri compiler identifies the minimal checkpoint set required for restoring a region (see Section 4.1), register-checkpointing stores can be further reduced with the following key insight. Checkpoints can be omitted if register values can be reconstructed using other checkpointed values available at recovery time. Capri leverages the optimal checkpoint pruning in the recent study [39]. The optimal pruning algorithm detects unnecessary checkpoints and replaces them with recovery codes to restore the pruned value.

Figure 3 shows how Capri removes the checkpoint stores and generates the reconstruction code. Suppose that a system crash or power failure happens in region #2. Region #2's input registers r1, r2, and r3 have been checkpointed in the prior regions, e.g., line 1c, 2c, and 4c/5c. Note that one of r2's checkpoint at lines 4c and 5c happens in region #1, depending on the path taken in the branch at line 3. Here, Capri can safely remove both checkpoints at line 4c and line 5c since the checkpointed r1 and r3 can reconstruct r2's value, as illustrated in the recovery block in Figure 3. The recovery block of region #2 executes the backward slice of the pruned checkpoint, including the branch to reconstruct r2 according to the checkpointed predicate r1, and jumps back to the recovery PC, i.e., the beginning of region #2.

4.4.2 Moving Checkpoints Out Of Loops. For checkpoint stores remaining after pruning, the Capri compiler explores opportunities

to further reduce the number of checkpoints by rearranging them. In particular, checkpointing stores are necessary for persisting registers values used in the later regions to restore the recovery points, which are the region boundaries. Therefore, the Capri compiler has the freedom to rearrange or delay the checkpoint stores from the original locations—immediately after the last load instruction updating the register—down to any points before the region boundary without any harm.

Capri leverages this observation to further reduce checkpointing stores by moving checkpoint stores in a loop to out of it, similar to loop invariant code motion (LICM) optimization. In Figure 4, the Capri compiler moves r1's checkpoint at line 2c out of a loop, reducing repeated checkpoints for the same register. In addition, another checkpoint at line 1c can be removed since the r1's checkpoint moved to the region boundary.

5 CAPRI ARCHITECTURE

5.1 2-Phase Atomic Store with Undo+Redo Logging

Capri uses the two-phase store protocol with undo+redo logging to provide failure-atomicity. Using the undo+redo-based approach brings the following advantages over undo- or redo-based ones.

5.1.1 Indirect Read-Free. The Capri architecture does not change data movements within the regular memory hierarchy. Thus, it provides indirect read-free; memory loads and stores perform the same way as the current architecture. On the other hand, the previous redo logging approaches have dealt with the so-called indirect read problem. In particular, since redo logging keeps newly updated values in the log area before transactions or atomic regions commit, a memory load at some levels must pay additional search overheads to seek the latest value in the log.

The previous studies have proposed various ways to deal with this problem. In common, they change the way program updates NVM such that dirty cache blocks are silently dropped while only the logs update them [10, 34, 45, 64]. For example, Jeong et al. and Nalli et al. use HW bloom filters to reduce indirect reads [35] or delay the reads [64]. In addition, Kolli et al. incorporate the cache coherence to eliminate indirect reads [45]. Furthermore, Cai et al., introduces a fast buffer to minimize the search costs at the memory controller [10], and Jeong et al. speculate that indirect reads are not required because they have a rare chance to find the value in the log [34]. On the other hand, Capri eliminates indirect reads without altering the regular data path (i.e., not dropping dirty cache blocks). Thus, the program never accesses the proxy buffer during execution.

5.1.2 Asynchronous Region Persistence. Similar to the redo-based approach, undo+redo logging provides asynchronous persistence. Therefore, the Capri architecture enforces region-level persistence (almost) without stalling or delaying program execution. Instead, the persistence of store instructions happens in the background while execution proceeds in the following region. This advantage will be limited if using undo principle.

5.1.3 Compared to the previous undo+redo approach [68]. Although the previous work has already employed the undo+redo principle in persistent memory [68], Capri has the following new contributions over the previous work. First, Capri relies on a two-phase atomic store to persist regions without explicit cache flushes. Hence, it does not change the cache structure at all. In contrast, the previous study periodically scans the cache hierarchy to flush dirty data to NVM with hardware tag extension. Second, Capri aims for whole-system persistence that makes any programs crash-consistent, while the previous study assumes persistent applications with transactions.

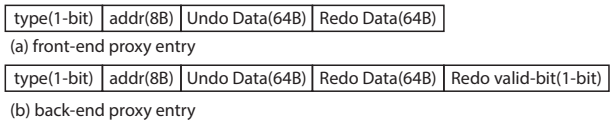


Figure 5: Front-end and back-end proxy entries.

5.2 Decoupled Proxy Buffer Architecture

The two-phase atomic store uses the proxy buffer as the safeguard to prevent partial updates visible after failure recovery. That is, each phase of store release is strictly ordered by intervening in the proxy buffer as a bridge. Thus, before completing the first phase, the second phase does not begin.

5.2.1 Front-End Proxy and the First Phase of Stores. The first phase of the atomic store generates proxy entries that consist of the home address and two cache lines (e.g., one for before and another for after the update), as shown in Figure 5(a). Capri creates proxy entries for every store instruction in the L1 data cache. In case of a cache miss, it waits until the cache fetches and allocates a block. Then, Capri obtains the old cache line (e.g., before the update) and the new cache line (e.g., after the update) and stores them in the front-end proxy buffer. Note that the front-end proxy will merge proxy entries with the same address within a region. If not merged, it appends the entry at the end of the front-end proxy buffer. The first phase completes when all stores in regions create associated proxy entries since the front-end proxy buffer is non-volatile.

It is essential to persist regions sequentially. To delineate regions within the proxy buffer, region boundary instructions also occupy the front-end proxy buffer. A single-bit indicator in the entry (e.g., type as shown in Figure 5) determines whether the entry is region boundary. The region boundary entry contains no address and data, but it only serves as a delimiter between entries. In addition, the front-end proxy buffer does not merge proxy entries even if two entries have the same address when they belong to the different regions. This way, proxy entries in each region become persistent in sequential order.

The front-end proxy buffer is a fixed and small non-volatile buffer (e.g., battery-backed SRAMs). Please refer to Section 6.1 for detailed information. The front-end proxy buffer flushes entries as much as possible to make space so as not to block the pipeline. In other words, the core pipeline does not stall as long as the front-end proxy has space to allocate new entries.

Optimizations: Since the front-end proxy buffer pressures the memory side with additional traffic, it is important to reduce the size of proxy entries. To mitigate memory pressure derived from the front-end proxy, Capri employs the following optimizations. First,

the front-end proxy buffer does not allocate entries for register-checkpointing stores since their recovery protocol is different from non-checkpointing (e.g., regular) stores (see Section 5.4 for the recovery protocol of Capri). Instead, Capri keeps dedicated register file storage near the front-end proxy buffer. Hence, only regular store instructions populate the front-end proxy buffer entries. Furthermore, as a result of compiler-directed region partitioning, there are many regions without stores. In this case, the front-end proxy buffer does not allocate an entry for the region boundary to save traffic to the back-end.

5.2.2 Back-End Proxy and the Second Phase of Stores. The second phase of the atomic store moves data from the back-end proxy to NVM home address (i.e., NVM is the non-volatile main memory). The memory controller contains per-core back-end proxy buffers. Figure 5(b) illustrates the back-end proxy entry. The back-end proxy buffer receives proxy entries from the front-end through the proxy data path, separated and dedicated data path for each core to each back-end buffer. Note that the back-end does not flush entries until it accepts the region boundary entry. Instead, once the back-end receives the region boundary entry, it moves redo data of all proxy entries of the given region depending on the redo valid-bit (see Section 5.3 for discussion about the redo valid-bit). Lastly, Capri performs the second phase of stores in the granularity of regions (e.g., proxy entries between region boundary entries).

The interplay of architecture and compiler determines the back-end proxy buffer sizes. Since the back-end proxy must be large enough to accommodate all proxy entries of any given single region, it must be greater or equal to the number of proxy entries multiplied by the threshold given by the compiler. This requirement is essential to provide the two-phase atomic store protocol. If the back-end proxy does not buffer all proxy buffer entries of the region, partial updates remain after crash recovery.

5.3 Enforcing the Persist Order

Capri proposes a distinctive architecture that allows both dirty cache line writeback and separate data paths to update NVM, while all related studies permit either only one of them to do so [10, 22, 25, 34, 35, 37, 45, 64, 68]. In particular, Capri does not guarantee the persist order between the regular path (e.g., through on-chip and off-chip caches) and proxy path. Hence, the persists between two paths can come in any order, resulting in an inconsistent main memory state if not properly handled. Although such inconsistency does not harm crash consistency, it can leads to a stale read problem.

5.3.1 Stale Reads. Consider an example in Figure 6. The program executes two regions that both modify address A to 10 and 20, respectively. The proxy path generates separate proxy entries for stores in regions #1 and #2 without merging since they belong to different regions. On the other hand, two stores are merged in the cache hierarchy. Hence, the regular path causes only a single writeback request (assuming the MOESI coherence protocol to minimize unnecessary writeback). Since cache writeback requests tend to arrive later than proxy entries from the proxy path (or non-temporal paths) [34, 71], the arrival order is ❶ ➜ ❷ ➜ ❸ in most cases. If so, this persist order does not harm memory consistency, and thus no stale reads happen most of the time.

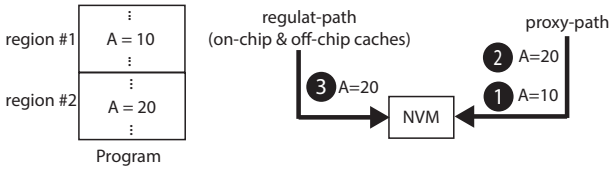


Figure 6: The stale read problem due to unordered persist between the regular and proxy paths.

However, for a rare chance, the order could change due to the delayed second phase of a region or a particular access pattern in the program that evicts the cache line quickly. In particular, ❶ ➜ ❸ ➜ ❷ or ❸ ➜ ❶ ➜ ❷ can be possible (note that ❶ ➜ ❷ are always serialized by the proxy path). For the former case (i.e., ❶ ➜ ❸ ➜ ❷), it does not corrupt the correctness of Capri. Instead, the last store (e.g., ❷) is unnecessary and thus wastes NVM bandwidth. On the other hand, the latter case (i.e., ❸ ➜ ❶ ➜ ❷) causes stale reads if the load request for A arrives between ❶ and ❷. In that case, although the correct and latest value is 20, the load request reads 10 from NVM.

Although this misordering may cause stale reads, it does not corrupt recovery. In particular, whether the power failure happens in all combinations, Capri can always recover to the latest recovery point (e.g., the region boundary) since the recovery procedure relies on undo+redo logging. Please see Section 5.4 for more details.

5.3.2 Stale Read Prevention. Capri introduces two techniques that manipulate the redo valid-bit in the proxy entry to prevent stale reads. First, it scans the back-end proxy buffer when dirty cache block writeback happens and unsets (e.g., invalidates) the redo valid-bit of entries that match the address. Then, it skips proxy entries with the redo valid-bit unset during the second phase atomic store. For example, in two scenarios above (e.g., ❶ ➜ ❸ ➜ ❷ and ❸ ➜ ❶ ➜ ❷ in Figure 6), Capri scans the back-end proxy buffer when the dirty writeback (e.g., ❸) appears from the regular path and unsets the redo valid-bit of A in the back-end proxy buffer. That way, the last store (e.g., ❷) in the first example will not happen, saving NVM bandwidth. Similarly, both proxy entries in regions #1 and #2 will be skipped during the second phase store in the latter example. Therefore, the stale read problem does not occur since NVM always contains the latest value.

Second, when cache writeback occurs, Capri monitors the incoming entries from the proxy buffer to handle the case when the back-end proxy buffer does not populate entries until the write-back happens. For example, two entries (e.g., ❶ and ❷ in Figure 6) are not populated yet when the writeback (e.g., ❸) appears. Then, the scanning approach cannot mitigate the stale reads. Note that this case is extremely infrequent if considering the latency of the deep cache hierarchy, including on-chip and off-chip caches. Even though such rare events occur, Capri can prevent the stale read problem by monitoring the proxy path if the proxy entries to the same address appear within the worst-case latency[2]. If ❶ and ❷ arrive within the window, their redo valid-bits are unset. Hence,

[2]The worst-case latency of the proxy path is determined in hardware design time based on hardware specification. All packets are guaranteed to arrive to the destination within this latency.

Capri does not copy them to NVM on the second phase of stores. If not, the monitoring ends without doing anything.

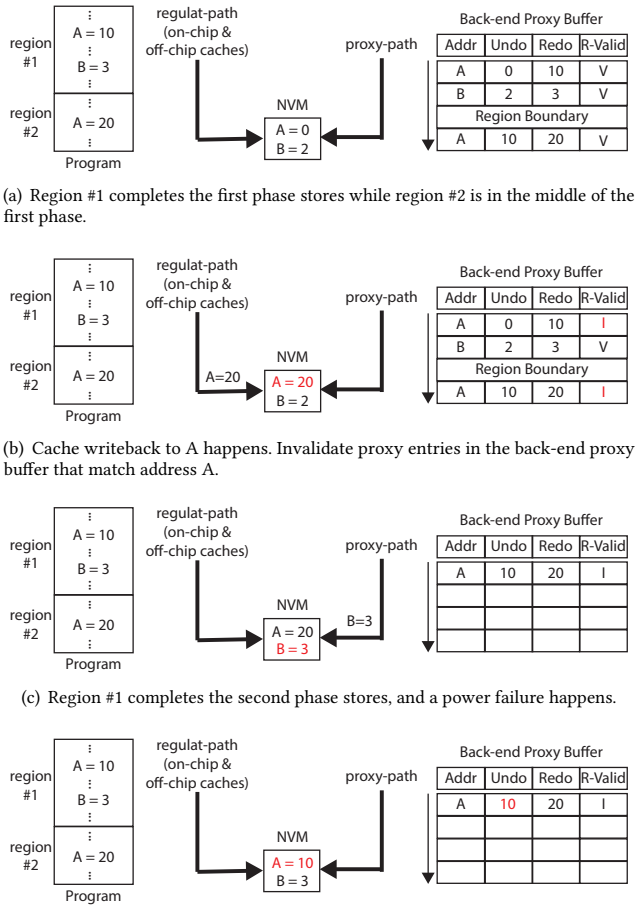
5.4 Crash Recovery with Undo+Redo Logging

5.4.1 Recovery Protocol. Capri provides a safe recovery protocol for whole-system persistence that does not require application-specific recovery codes. That is, the operating system spawns recovery threads when the system reboots after failure. First, they restore the main memory (NVM) state using proxy entries left in the proxy buffer. In particular, Capri determines how to restore regions depending on the two-phase store status (e.g., whether the first phase has finished or not). For this purpose, the recovery threads check the region boundary entry as it serves as the commit marker. If one exists after data proxy entries, it indicates that the region has been interrupted after completing the first phase. If not found, the region did not finish the first phase. For regions that have completed the first phase before failures, the recovery threads copy redo data of proxy entries of the region to NVM. On the other hand, regions that have not completed the first phase will be rollbacked. In particular, undo data in proxy entries of these regions are used to restore the value before executing them.

Second, the recovery threads restore the register values for each physical core using the register checkpointing storage in NVM. When the Capri compiler generates register-checkpointing stores, their destination addresses are fixed since the number of architectural registers is statically determined in the ISA. Hence, the recovery threads reload the architectural register values using the predefined mapping between the register file and physical addresses.

Once the recovery protocol completes failure recovery, programs can resume from the beginning of the interrupted regions. Note that restoring the non-volatile main memory and register values in cores is sufficient for whole-system persistence since previous regions have already persisted in the main memory.

5.4.2 Handling Cache Writeback. Capri guarantees sequential persistence of regions for crash recovery. However, the dirty cache writeback may break the sequential persist order of regions since Capri allows NVM updates from both caches and proxy paths. For example, Figure 7 illustrates the program executing two regions. Suppose that region #1 completes the first phase stores while region #2 performs the first phase (Figure 7(a)). All back-end proxy entries are already allocated on the right side of the figure. Then, in Figure 7(b), the cache writeback updates address A before the second phase of region #1 (e.g., proxy entries still exist). Hence, the redo valid-bits of entries with address A become unset (see Section 5.3). Next, in Figure 7(c), assume that a power failure happens after region #1 completes two-phase atomic stores while region #2 does not finish the first phase. However, NVM is inconsistent as A is supposed to be the value 10 updated by the store in region #1. As shown in Figure 7(d), Capri can restore the value of A to 10 using the undo data of proxy entries A in the proxy buffer.



(a) Region #1 completes the first phase stores while region #2 is in the middle of the first phase.

(b) Cache writeback to A happens. Invalidate proxy entries in the back-end proxy buffer that match address A.

(c) Region #1 completes the second phase stores, and a power failure happens.

(d) Crash recovery rollbacks A=10, which is the end of region #1, using the undo value of A in the back-end proxy buffer.

Figure 7: Undo+redo logging enables crash recovery even with unordered persists between regular and proxy paths.

Table 1: Simulator configuration.

Processor	ARMv8 (64-bit) ISA, 2GHz, 8way-OoO 128/72-entry Ld/St Queue
L1 I/D Cache	32/32KB, 8-way, private 2ns hit latency (1ns tag/1ns data latency)
L2 Cache	16MB, 16-way, shared 20ns hit latency (10ns tag/10ns data latency)
Integrated Mem Ctrl.	32/64-entry DRAM read/write queue 32/16-entry NVM read/write queue
DRAM	8GB, DDR4 2400MHz
NVM	32GB, Read = 150ns, Write = 300ns
Proxy Path	20ns latency

6 EVALUATION

6.1 Methodology

We implemented compiler techniques described in Section 4 in the LLVM 13.0 compiler infrastructure [48] and architecture support explained in Section 5 in the gem5 simulator [9]. We conduct our simulation with ARMv8 (64-bit) ISA, modeling an 8-core out-of-order processor with the vertically-integrated hybrid memory such

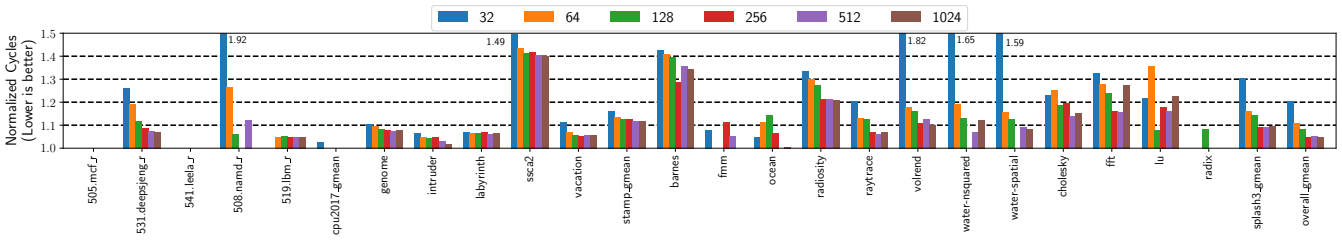


Figure 8: Normalized execution cycles with different store thresholds.

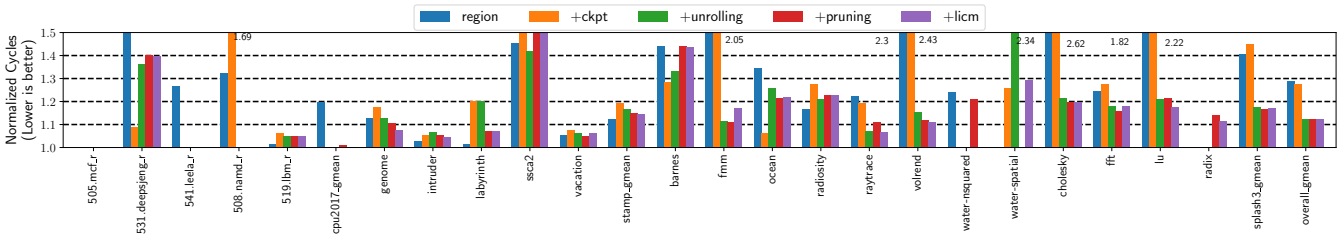


Figure 9: Normalized execution cycles with different compiler optimizations.

as the memory mode of the Intel Optane [3]. Table 1 shows the simulation configuration in detail. We configured the integrated memory controller that governs DRAM and NVM with a separated read/write queue. The hardware-managed DRAM cache has 64B of cache block size [81] and uses a direct-mapped policy. Also, we set the write-pending queue (WPQ) size of 16 entries [81] while the WPQ is in the part of the persistent domain. The front-end proxy buffer size is fixed to 32 entries (e.g., 4KB), while the number of back-end proxy buffer entries is the same as the threshold. For instance, the threshold of 256 requires 32KB per-core non-volatile storage (e.g., battery-backed SRAMs) for the back-end proxy buffer.

We evaluated Capri with both single-threaded (SPEC CPU2017 [6] and STAMP [62]) and multi-threaded (Splash3 [69]) benchmarks. All benchmarks were compiled with standard -O3 optimization, and the default threshold number of stores in each region is 256. We ran these benchmarks using the full-system simulation mode of gem5 with ARM Linux Kernel 4.14.239 compiled with the Capri compiler. Therefore, the operating system contains the region formation and register-checkpointing stores. For SPEC CPU2017, we fast-forwarded 10 billion instructions and simulated the following 2 billion instructions (excluding region boundary and checkpoint store instructions). On the other hand, we simulated the entire program execution for Splash3 and STAMP. Furthermore, we compiled STAMP benchmarks as a sequential program. Lastly, all results are normalized to the unmodified programs that do not have region boundary instructions and checkpoint stores.

6.2 Performance Evaluation

Different Store Thresholds. Figure 8 shows execution cycles of each benchmark from SPEC CPU2017, STAMP, and Splash3 while varying the store thresholds used to determine the region formation in the Capri compiler. In addition, we synergically applied compiler optimizations such as speculative unrolling, checkpoint pruning, and LICM and plotted the best combination of them.

The trend clearly shows that the larger thresholds (and thus longer regions) provide better performance because of fewer checkpointing stores and the less dynamic instruction count increase. For example, there are clear performance improvements when increasing the threshold from 32 to 64 for several benchmarks such as 508.namd_r from CPU2017, ssca2 from STAMP, and volrend, water-nsquared, and water-spatial from Splash3. As a result, Capri, with the threshold of 32, incurs a 20% slowdown compared to volatile execution, while increasing the threshold to 64 halves the slowdown, on average.

Finally, Capri incurs only a 5.1% performance overhead compared to volatile programs—when the threshold is 256—with the help of the novel Capri architecture. First, the front-end proxy buffer effectively reduces the volatile and persistence gap while hiding the data movement from the front to the back-end proxy in the background. Furthermore, since Capri takes the indirect read-free design, memory loads—which is more critical than stores in modern computer architecture—are never slowed down unlike prior work [34, 45, 64].

Compiler Optimization. This section discusses how the Capri compiler optimizations, i.e., speculative unrolling, checkpoint pruning, and LICM, affect application performance. Figure 9 shows execution cycles of the tested benchmarks while accumulatively applying the compiler optimizations. For instance, the blue bars indicate the performance overheads of placing region boundary instructions without checkpointing stores (thus not failure atomic). The yellow bars show the resulting overheads when checkpointing stores are inserted on top of the region formation. Similarly, the right-most purple bars correspond to the overheads of getting all our compiler optimization techniques enabled.

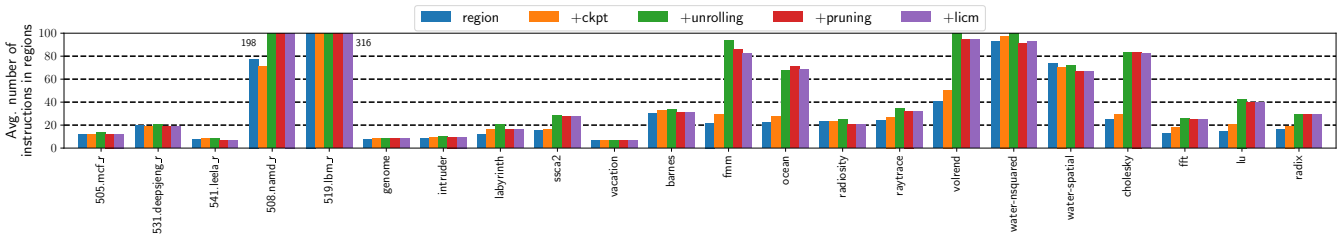


Figure 10: Average number of instructions in regions.

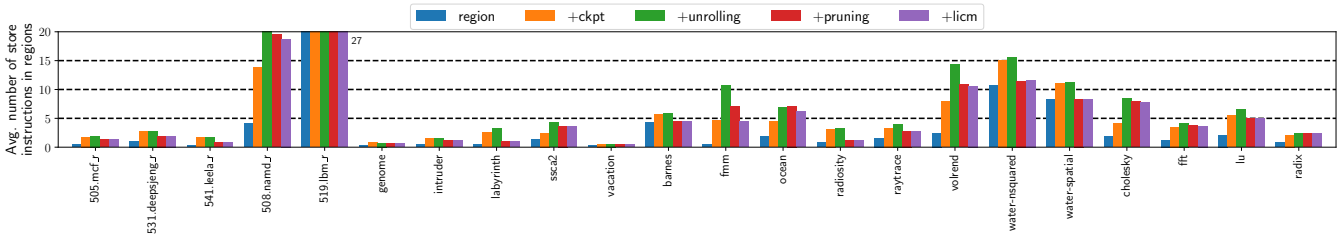


Figure 11: Average number of store instructions in regions.

First, our speculative loop unrolling turns out to be effective for many of the benchmarks since it expands region lengths significantly, as shown in Section 6.3. This speedup empirically proves that most programs contain short loops—harmful for region-level whole-system persistence, but Capri's speculative loop unrolling effectively mitigates this challenge by lengthening the loop body. Second, the best-performing set of our compiler optimization techniques differ across the benchmarks—mainly due to the non-identical application characteristics such as store density, live-out registers, and how they are defined. In addition, although checkpoint pruning and LICM techniques do not show noticeable speedup in geometric mean, they reduce NVM writes and thus are particularly beneficial in terms of improved power consumption and NVM endurance.

6.3 Region Formation

This section reports the average region length (e.g., the number of instructions therein) and the average number of stores (including checkpoints) in regions with different compiler techniques in Figures 10 and 11, respectively. Region length directly relatd to performance; for example, 508.namd, ssca2, and volrend show huge speedup when region sizes grow by using speculative loop unrolling. Checkpoint pruning and LICM optimizations reduce the region sizes since they remove checkpoint stores. Furthermore, even if the Capri compiler takes 256 stores as the threshold number to delineate the region boundaries, the resulting formation reveals that each region has fewer stores than the threshold. These results indicate that program structures such as loops and function calls are frequently happening in program execution, limiting the efficiency of the Capri's region partitioning. Since it is expected to reduce whole-system persistence costs with longer regions, our future work is to devise a new algorithm to formulate regions with having more instructions.

7 OTHER RELATED WORK

Battery-Backed Buffers: Recent advances in persistent memory technology propose to locate the battery-backed buffer close to cores to reduce persistence overheads [1, 7, 23, 66]. For example, Intel announced enhanced-ADR (eADR) that includes on-chip caches within the persistent domain [1]. This feature expects to show drastic performance improvement by removing cache flush overheads in persistent memory applications, although implementing eADR turns out to be non-trivial [2]. Besides, whole-system persistence requires similar hardware support that covers on-chip caches [66]. Furthermore, several research papers have proposed battery-backed buffers in the cache hierarchy that substantially reduces the capacitor size. For example, TSOPER uses a battery-backed buffer, atomic-group buffer (AGB), between LLC and NVM [23]. BBB places it alongside the L1 cache [7]. But, all these works need changes in the cache coherence mechanisms while Capri does not. The BPB is the proxy buffer to NVM that prevents partial updates rather than serving demand loads.

Compiler-Directed Region Partitioning: Capri leverages compiler analysis to partition program into a series of recoverable regions. Despite different region criteria and goals, prior studies leverage such compiler-directed region partitioning, e.g., iDO [50] and Penny [39] form side-effect-free idempotent regions for power failure recovery and soft error recovery, respectively. Also, other prior studies use a gated store buffer (SB) [16, 51, 55, 85]—as a redo buffer—and split program into a series of regions with the SB size in mind so that no region overflows the SB. For example, Turnstile [55] and Turnpike [85] both take advantage of the SB-aware region formation for soft error resilience, while CoSpec [16] exploits the region formation to achieve crash consistency across frequent power failure in energy harvesting systems [14–16, 51]

that do not have caches. On the other hand, Capri targets whole-system persistence for general-purpose computing platform with deep cache hierarchy.

8 CONCLUSION

To leverage both high-density and in-memory persistence benefits of NVM, the users of Intel Optane are forced to select the app-direct mode over the memory mode. As a result, only a handful of applications can resort to the both benefits at the expense of persistent programming difficulty. To address the limitation, this paper introduced Capri, a compiler/architecture co-design scheme for region-level whole-system persistence. Unlike partial-system persistence, Capri makes any programs failure-atomic without source code change while letting them enjoy both high-density and in-memory persistence simultaneously. This guarantee is particularly promising for the Optane users since Capri can free them from all the headaches of persistent programming including notorious crash consistency bugs. To achieve this, the Capri compiler generates recoverable regions while Capri architecture guarantees their execution to be crash-consistent. Furthermore, Capri's compiler and architecture optimizations enable lightweight whole-system persistence (5.1% average slowdown), thereby offering all programs high-performance persistence with increased memory space.

ACKNOWLEDGMENTS

We would like to thank the HPDC reviewers for their insightful comments and the members of the Purdue CompArch research group for early discussions on the project. This work was supported by NSF grants 1750503 (CAREER) and 1814430.

REFERENCES

[1] [n.d.]. eADR: New Opportunities for Persistent Memory Applications. https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html.
[2] [n.d.]. From FLOPS to IOPS: The New Bottlenecks of Scientific Computing. https://www.sigarch.org/from-flops-to-iops-the-new-bottlenecks-of-scientific-computing/.
[3] [n.d.]. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html.
[4] [n.d.]. Lenovo Memcached-pmem. https://github.com/lenovo/memcachedpmem.
[5] [n.d.]. PMEM Redis. https://github.com/pmem/redis/tree/3.2-nvml.
[6] [n.d.]. SPEC CPU2017. https://www.spec.org/cpu2017/.
[7] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. 2021. BBB: Simplifying Persistent Programming using Battery-Backed Buffers. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.
[8] Mazen Alwadi, Vamsee R. Kommareddy, Clayton Hughes, Simon D. Hammond, and Amro Awad. 2020. Stealth-Persist: Architectural Support for Persistent Applications in Hybrid Memory Systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* (2011).
[10] Miao Cai, Chance C. Coats, and Jian Huang. 2020. HOOP: Efficient Hardware-Assisted Out-of-Place Update for Non-Volatile Memory. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*.
[11] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*.
[12] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[13] Ping Chi, Shuangchen Li, Yuanqing Cheng, Yu Lu, Seung H Kang, and Yuan Xie. 2016. Architecture design with STT-RAM: Opportunities and challenges. In *6 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*.
[14] Jongouk Choi, Hyunwoo Joe, Yongjoo Kim, and Changhee Jung. 2019. Achieving stagnation-free intermittent computation with boundary-free adaptive execution. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
[15] Jongouk Choi, Larry Kittinger, Qingrui Liu, and Changhee Jung. 2022. Compiler-Directed High-Performance Intermittent Computation with Power Failure Immunity. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
[16] Jongouk Choi, Qingrui Liu, and Changhee Jung. 2019. CoSpec: Compiler Directed Speculative Intermittent Computation. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[17] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[18] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through Byte-Addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*.
[19] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
[20] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[21] Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Helen Li, and Yiran Chen. 2008. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *45th ACM/IEEE Design Automation Conference (DAC)*.
[22] Kshitij Doshi, Ellis Giles, and Peter Varman. 2016. Atomic persistence for SCM with a non-intrusive backend controller. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[23] Per Ekemark, Yuan Yao, Alberto Ros, Konstantinos Sagonas, and Stefanos Kaxiras. 2021. TSOPER: Efficient Coherence-Based Strict Persistency. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[24] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. 2020. go-pmem: Native Support for Programming Persistent Memory in Go. In *USENIX Annual Technical Conference (ATC)*.
[25] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2020. Relaxed Persist Ordering Using Strand Persistency. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.
[26] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *USENIX Annual Technical Conference (ATC)*.
[27] Siddharth Gupta, Alexandros Daglis, and Babak Falsafi. 2019. Distributed Logless Atomic Durability with Persistent Memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[28] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[29] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-Threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*.
[30] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *USENIX Annual Technical Conference (ATC)*.
[31] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*.
[32] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[33] Jungi Jeong, Jaewan Hong, Seungryoul Maeng, Changhee Jung, and Youngjin Kwon. 2020. Unbounded Hardware Transactional Memory for a Hybrid DRAM/NVM Memory System. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[34] Jungi Jeong and Changhee Jung. 2021. PMEM-Spec: Persistent Memory Speculation (Strict Persistency Can Trump Relaxed Persistency). In *Proceedings of the International Conference on Architectural Support for Programming Languages and*

Operating Systems.

[35] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. 2018. Efficient Hardware-Assisted Logging with Asynchronous and Direct-Update for Persistent Memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[36] Yongsoo Joo, Dimin Niu, Xiangyu Dong, Guangyu Sun, Naehyuck Chang, and Yuan Xie. 2010. Energy-and endurance-aware design of phase change memory caches. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 136–141.

[37] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[38] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST)*.

[39] Hongjune Kim, Jianping Zeng, Qingrui Liu, Mohammad Abdel-Majeed, Jaejin Lee, and Changhee Jung. 2020. Compiler-directed Soft Error Resilience for Lightweight GPU Register File Protection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[40] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[41] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*.

[42] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. 2018. ClfB-Tree: Cacheline Friendly Persistent B-Tree for NVRAM. *ACM Transactions on Storage* 14, 1, Article 5 (2018).

[43] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[44] Apostolos Kokolis, Thomas Shull, and Josep Huang, Jian Torrellas. 2020. P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

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

[46] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[47] Marc de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static Analysis and Compiler Design for Idempotent Processing. In *the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[48] Chris Lattnea and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization (CGO)*.

[49] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST)*.

[50] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. 2018. IDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

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

[52] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2015. Clover: Compiler Directed Lightweight Soft Error Resilience. In *Proceedings of the 16th Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[53] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Compiler-Directed Lightweight Checkpointing for Fine-Grained Guaranteed Soft Error Recovery. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[54] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Compiler-directed soft error detection and recovery to avoid DUE and SDC via Tail-DMR. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 2 (2016).

[55] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Low-Cost Soft Error Resilience with Unified Data Verification and Fine-Grained Recovery for Acoustic Sensor Based Detection. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[56] Ruicheng Liu, Peiquan Jin, Xiaoliang Wang, Zhou Zhang, Shouhong Wan, and Bei Hua. 2019. NVLevel: A High Performance Key-Value Store for Non-Volatile Memory. In *IEEE 21st International Conference on High Performance Computing and Communications (HPCC)*.

[57] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. 2021. 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 (ASPLOS)*.

[58] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. 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 (ASPLOS)*.

[59] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. 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 (ASPLOS)*.

[60] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[61] Amirsaman Memaripour and Steven Swanson. 2018. Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software. In *IEEE 36th International Conference on Computer Design (ICCD)*.

[62] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *IEEE International Symposium on Workload Characterization*.

[63] Sparsh Mittal, Jeffrey S Vetter, and Dong Li. 2014. LastingNVCache: A technique for improving the lifetime of non-volatile caches. In *IEEE Computer Society Annual Symposium on VLSI*.

[64] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[65] Moohyeon Nam, Hokeun Cha, Youngri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST)*.

[66] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence with Non-volatile Memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[67] Ian Neal, Andrew Quinn, and Baris Kasikci. 2021. 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 (ASPLOS)*.

[68] Matheus A. Ogleari, Ethan L. Miller, and Jishen Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

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

[70] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. 2017. Failure-Atomic Slotted Paging for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[71] Sara Mahdizadeh Shahrai, Seyed Armin Vakil Ghahani, and Aasheesh Kolli. 2020. (Almost) Fence-less Persist Ordering. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[72] Seunghee Shin, Satis K. Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A Flexible and Fast Software Supported Hardware Logging approach for NVM. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[73] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. AutoPersist: An Easy-to-Use Java NVM Framework Based on Reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[74] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. 1995. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*.

[75] Guangyu Sun, Xiangyu Dong, Yuan Xie, Jian Li, and Yiran Chen. 2009. A novel architecture of the 3D stacked MRAM L2 cache for CMPs. In *IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*.

[76] Farhad Tabrizi. 2007. The future of scalable stt-ram as a universal embedded memory. *Embedded. com, February* (2007).

[77] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *USENIX Annual Technical Conference (ATC)*.

[78] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*.

[79] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference*

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

[80] Jue Wang, Xiangyu Dong, Yuan Xie, and Norman P Jouppi. 2013. i 2 WAP: Improving non-volatile cache lifetime by reducing inter-and intra-set write variations. In *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA).*

[81] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[82] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (ATC).*

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

[84] Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay Paddayuru Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. 2021. ReplayCache: Enabling Volatile Cachesfor Energy Harvesting Systems. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[85] Jianping Zeng, Hongjune Kim, Jaejin Lee, and Changhee Jung. 2021. Turnpike: Lightweight Soft Error Resilience for In-Order Cores. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[86] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. Energy reduction for STT-RAM using early write termination. In *IEEE/ACM International Conference on Computer-Aided Design-Digest of Technical Papers.*