COSPlay: Leveraging Task-Level Parallelism for High-Throughput Synchronous Persistence

Marina Vemmou
mvemmou@gatech.edu
Georgia Institute of Technology
USA

Alexandros Daglis
alexandros.daglis@cc.gatech.edu
Georgia Institute of Technology
USA

ABSTRACT

A key challenge in programming crash-consistent applications for Persistent Memory (PM) is achieving high performance while controlling the order of PM updates. Managing persist ordering from the CPU typically requires frequent synchronization points, which expose the PM's high persist latency on the execution's critical path. To mitigate this overhead, prior proposals relax the persistency model and decouple persistence from the program's volatile execution, delegating persistence ordering to specialized hardware mechanisms such that persistent state lags behind volatile state. In this work, we identify the opportunity to mitigate the effect of persist latency by leveraging the task-level parallelism available in many PM applications, while preserving the stricter semantics of synchronous persistence and the familiar x86 persistency model.

We introduce *COSPlay*, a software-hardware co-design that employs coroutines and rapid userspace context switching to hide persist latency by overlapping persist operations across concurrent tasks. Modest CPU extensions enable the hardware to fully overlap persists of different contexts, while preserving intra-context ordering to meet crash consistency requirements. *COSPlay* boosts the throughput of crash-consistent applications by up to 1.7× on systems with basic PM support. For systems with higher persist latency due to added backend memory operations, such as encryption and deduplication, *COSPlay*'s performance gains grow to 2.2 – 7.3×.

CCS CONCEPTS

• **Hardware → Memory and dense storage**.

KEYWORDS

persistent memory, persist ordering, crash consistency, task-level parallelism, coroutines

ACM Reference Format:
Marina Vemmou and Alexandros Daglis. 2021. COSPlay: Leveraging Task-Level Parallelism for High-Throughput Synchronous Persistence. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3466752.3480075

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480075

1 INTRODUCTION

Persistent Memory (PM) has been a much-anticipated technology, as it combines the persistence of storage with performance comparable to DRAM. While simply replacing conventional storage with PM can drastically boost the performance of data-intensive applications, reaping the full benefits of PM and keeping processors highly utilized requires redesign efforts spanning the entire system stack, from hardware, to software libraries and runtime systems. With PM recently becoming commercially available [36], these efforts are now more timely than ever.

PM provides applications with direct access to a persistent byte-addressable domain through a load/store interface, at near-DRAM performance. While this feature is a major performance booster for crash-consistent applications, a key challenge lies in ensuring that updates reach PM (i.e., they persist) in the intended order. Unless explicitly controlled, persist order differs from store order, due to the deep cache hierarchy between the CPU and PM. While store order is dictated by the memory consistency model, persist order is dictated by the *memory persistency model*. Although several persistency models have been proposed [11, 25, 37], the only currently available and widely used one is the x86 persistency model, which implements an instance of epoch persistency. Under x86 persistency, programmers use the `clwb` instruction to explicitly indicate the updated cache blocks that should be written back to PM. The `sfence` instruction is used as a persist barrier to enforce a desired order across `clwbs`, by stalling the execution of younger `stores`/`clwbs` until all older ones have completed. While persist barriers allow programmers to control the order of groups of persists, they place persist latency on the critical path. Therefore, the latency of persist operations has a direct impact on performance.

To alleviate the effect of persist latency on performance, Asynchronous DRAM Refresh (ADR) support in modern PM-enabled systems allows PM updates to be considered persistent when they reach the write queue of the memory controller, rather than the PM device itself [18]. However, compared to volatile DRAM, PM's persistence property introduces additional constraints for the stored data, such as the need to preserve data integrity and confidentiality. Therefore, PM systems may need to apply additional operations—such as encryption, integrity protection, deduplication—at the memory controller. Such Backend Memory Operations (BMOs) [29] increase the latency of persist operations. As persist barriers expose this prolonged persist latency on the critical path, BMOs can drastically degrade the performance of crash-consistent applications.

We posit that the task-level parallelism inherent in a large class of crash-consistent applications can be leveraged to ameliorate the long-latency effect of persist barriers. By associating independent application-level tasks with different logical contexts, when

one context is stalled waiting for its updates to persist, another context can make forward progress. We achieve that via software-hardware co-design. On the software front, we employ coroutines, a lightweight software component for preemptive multitasking, to exploit an application's task-level parallelism and rapidly switch between tasks. Coroutines are becoming increasingly relevant in the upcoming era of microsecond-scale computing [2] and have already been effectively used for network and remote memory access latency hiding in distributed systems [23, 33, 48] and databases [15, 20, 38]. In addition, we introduce ISA extensions to associate persist operations with their context, and modest CPU hardware extensions to allow inter-context overlap of persists, while preserving the intended intra-context ordering. Our approach—*CO*routines for *S*ynchronous *P*ersistence (*COSPlay*)—leaves the memory hierarchy and the CPU's most performance-critical microarchitectural components unmodified.

Prior work removes persist latency from the critical path by relaxing semantics: either by resorting to relaxed persistency models [11], or by resorting to *asynchronous* task persistence—i.e., allowing a task to complete its execution and persist its PM updates at a later point in time [25, 28]. Compared to these approaches, *COSPlay* trades off general applicability in favor of preserving stricter semantics. *COSPlay* is specifically specialized for applications with inherent task-level parallelism, but operates under the predominant x86 persistency model familiar to programmers and provides *synchronous* persistence guarantees—i.e., a task's CPU execution may only complete when all of its updates have persisted. Our evaluation on a range of PM benchmarks commonly used in the literature [10, 13, 24, 29, 30] demonstrates that *COSPlay* can effectively hide the long persist latencies introduced by BMOs and exposed by `sfences`, delivering throughput improvements of $1.06 - 7.3\times$, depending on the granularity of the application's tasks and the underlying system's persist latency.

In summary, we make the following contributions:

- We leverage the rapid task switching enabled by coroutines as an effective software-based technique to hide PM's high persist latency hampering the performance of crash-consistent programs.
- We introduce *COSPlay*, a software-hardware co-design employing modest hardware extensions to drastically improve the throughput of crash-consistent applications with inherent task-level parallelism. *COSPlay* preserves the desirable strict semantics of synchronous task persistence and the x86 persistency model.
- We thoroughly evaluate *COSPlay*, comparing against several alternative software and hardware approaches with equivalent persistency semantics.

The paper is organized as follows. §2 details the challenges of persist ordering, exacerbated by the presence of BMOs. §3 introduces our approach to addressing these challenges and the requirements for an effective design. We then present our implementation, *COSPlay*, in §4. We describe our methodology in §5 and evaluate *COSPlay* in §6. Finally, we discuss related work in §7 and §8 concludes.

2 BACKGROUND

2.1 Persist Ordering Challenges

A key complication in developing crash-consistent programs for PM is the divergence between volatile and persistent memory order. The former is dictated by the memory model (e.g., a TSO variant in modern x86 CPUs), while the latter is dictated by the *persistency model*, a much younger topic with active research activity. Currently, the practically predominant one is the *x86 persistency model*.

The x86 persistency model is an instance of epoch persistency [37] that allows programmers to explicitly control the order of individual or groups of updates to PM. The basic tools to control this order are two instructions: `clwb` and `sfence`. The `clwb` instruction causes a cache block to be written back to PM from the cache hierarchy, keeping a clean copy cached to preserve cache locality. The hardware does not implicitly provide any ordering guarantees across consecutive `clwbs` to different addresses, and concurrent `clwbs` may persist in any order. The `sfence` instruction plays the role of a persist barrier: it stalls the issuing of any new `stores` or `clwbs` in the CPU pipeline until all pending stores drain from the store queue and all pending `clwb` instructions have been acknowledged by the memory controller, indicating that their corresponding updates have persisted. Crash-consistent programs for PM use the combination of `clwb` instructions followed by an `sfence` to control the order of persistent updates: only when the `sfence` retires are prior `clwb` operations guaranteed to have persisted.

Prior work has correctly identified the x86 persistency model's main weakness: while persist ordering is an integral requirement of correct persistent applications, preserving a total order across *all* persists is often overly conservative and not required for correctness. A number of alternative models that relax ordering requirements to enhance performance have been proposed [37], with *strand* persistency allowing for the maximum amount of persist operations' overlapping. Under strand persistency, persists whose ordering must be maintained are grouped into *strands*. While intra-strand ordering is enforced, inter-strand ordering can be relaxed.

StrandWeaver [11] is the first implementation of strand persistency, showcasing that overlapping persists as allowed by strand persistency yields considerable performance gains. However, hardware implementations of strand persistency are not a panacea. Compared to strict ordering, relaxed ordering is arguably more arduous to reason about (e.g., consider TSO versus RMO/WO). In addition, support for strand persistency introduces considerable hardware complexity. In order to preserve correct persistence ordering while allowing the core to run ahead, StrandWeaver introduces a hardware-based offload model. Ordering constraints are maintained by a set of new, per-core hardware entities, the primary being the Strand Buffer Unit. Finally, although StrandWeaver *asynchronously* preserves the intended persist order, by ensuring it follows volatile memory order later in time, such temporal decoupling is not applicable when the application requires the stronger semantics of *synchronous* persistence. For applications that feature fine-grained tasks, where a task can only be considered completed when it is also guaranteed to be durable (i.e., all of its updates have persisted), the frequent need for explicit synchronization of the decoupled

volatile and persistent state (by using a `JoinStrand` in StrandWeaver), limits performance gains. We quantify these limitations in our evaluation (§6.3).

Finally, while it is too early to declare that relaxed persistency models such as strand persistency are too complicated or expensive to be practically adopted, it is highly likely that x86 persistency will dominate for years to come. We thus focus on the x86 persistency model in this paper, and propose mechanisms to improve performance on systems that abide by its semantics.

2.2 Persist Latency and BMOs

Crash-consistent programs make periodic use of persist barriers to ensure that updates have persisted in the intended order. `sfence` places the latency of writebacks to persistent memory, triggered by `clwb` operations, on the critical path. As the write latency of PM is significantly higher than DRAM (e.g., 500ns for Optane DIMM [19]), frequent occurrence of such synchronous events can dramatically hamper performance. To address this serious drawback, Intel's Asynchronous DRAM Refresh (ADR) [18] extends the underlying PM's effective persistence domain to the memory controller's write-back queues. Thus, the latency exposed by the `sfence` is that of reaching the memory controller's writeback queues rather than the PM device itself.

While ADR shrinks the critical path to the persistent domain, the unique characteristics of PM introduce the need for special operations added before the memory controller, known as *Backend Memory Operations* (BMOs) [29]. As PM is by definition non-volatile, security concerns may require the encryption of data written to it. PM's limited lifetime may require the application of wear-leveling and error correction techniques, while its limited write bandwidth motivates the use of compression. The addition of BMOs adds 100s of cycles on the critical path to the persistence domain, which can significantly degrade the performance of crash consistent programs using persist barriers to control persist ordering, according to the x86 persistency model.

To illustrate the combined effect of persist ordering with BMOs under the x86 persistency model, we examine the relationship between persist latency and application throughput. We assume an ADR-enabled baseline system with a persist latency of 200 cycles [19], and vary the latency added to that persist latency as an effect of a hypothetical BMO. We use a benchmark suite commonly used in PM literature featuring programs that execute short persistent transactions and use undo logging for crash consistency. Additional methodological details can be found in §5.

Figure 1 shows the overhead of crash consistency as a function of increasing persist latency due to the introduction of BMOs. Performance is shown as slowdown compared to non-crash-consistent versions of the applications, where the order of updates to PM is not controlled in any way (i.e., no `sfences`). We sweep the persist latency added to the baseline ADR latency from 0 to 1000 cycles (0 = plain ADR), representing hypothetical BMOs added on the persist critical path. Vertical lines mark latencies of exemplar BMOs from the PM research literature.

Even in the absence of BMOs (i.e., plain ADR), crash consistency incurs a 1.07 – 3.14× slowdown. Unsurprisingly, this slowdown linearly grows with increasing BMO latency. The performance impact

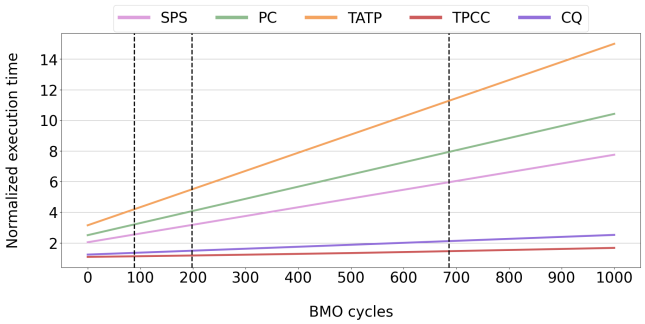


Figure 1: Slowdown of crash-consistent applications compared to their counterparts with unordered persists, as a function of persist latency (0 = plain ADR). Vertical lines indicate three exemplary BMOs (left to right): encryption [43], lightweight deduplication [50], heavyweight deduplication [50].

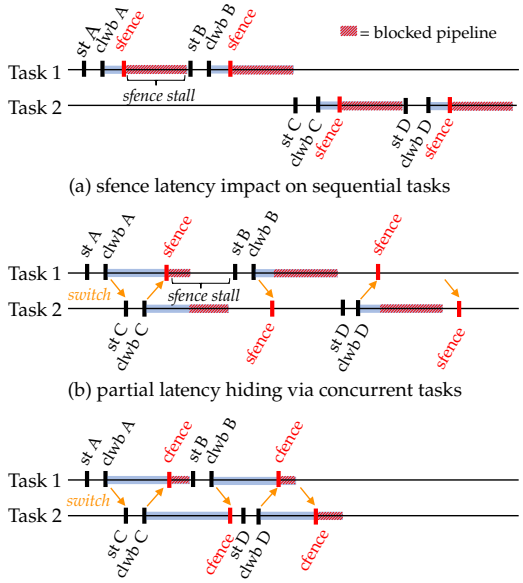
differs by application, based on the frequency of persist ordering points, but slowdowns are commonly in the range of 1.5 – 5× and more than 10× for TATP with the heaviest BMO we consider. In conclusion, persist ordering requirements for crash consistency introduce significant performance concerns, especially in the presence of BMOs that extend the latency of persist operations.

3 HIDING PERSIST LATENCY VIA TASK-LEVEL PARALLELISM

A large class of applications leveraging PM for its desirable properties, such as object stores and transactional systems, exhibit high degrees of task-level parallelism. There are multiple techniques across the system stack banking on parallelism to hide latency, and we posit that task-level parallelism can similarly be leveraged to ameliorate the long-latency effect of `sfence` operations waiting for preceding `clwbs` to complete.

Figure 2 demonstrates this opportunity with an example, where a single CPU core executes two independent tasks. The example assumes that crash consistency is achieved via write-ahead logging. In the default case (Figure 2a), task 2 starts executing after task 1 completes. Both tasks use a combination of `clwb` and `sfence` operations to guarantee the intended ordering of updates to PM and synchronous persistence—i.e., the task is considered completed only after its updates have persisted. The first `sfence` guarantees that the log will persist before the following in-place updates (A→B for task 1, C→D for task 2), while the second guarantees synchronous persistence. Blue shading indicates each `clwb`'s latency to become persistent. Red dashed lines over the blue shading indicate the duration a `clwb` is on the critical path, due to an `sfence` blocking the CPU pipeline until the `clwb`'s persistence is acknowledged.

The execution order of the two tasks implicitly imposes a full A→B→C→D ordering, placing four full persist (`clwb` + `sfence`) latencies on the critical path. As updates across tasks are semantically independent, there are intra-task but not inter-task ordering dependencies; thus, it is allowable to overlap the persists of task 1 and task 2.



(a) sfence latency impact on sequential tasks

(b) partial latency hiding via concurrent tasks

(c) improved latency hiding with context-aware fences

Figure 2: `clwb` and `sfence` impact on execution latency.

Figure 2b shows how overlapping inter-task persist latencies can partially recoup the cost of long-latency persist operations to improve performance. As soon as task 1 issues a long-latency `clwb` and before executing the following `sfence`, execution switches to task 2, which issues its `clwb` to address C, thus partially overlapping the latency to persist task 1's address A and task 2's address C. After issuing task 2's `clwb`, the CPU switches back to task 1. If task 1's previously issued persist is still pending, the CPU needs to still stall until the persist is acknowledged, to guarantee the intended persist ordering within task 1; thus, an `sfence` is still required. As the `sfence`'s semantics require waiting for *all* preceding `store`/`clwb` operations to complete, task 1's `sfence` stalls the CPU until task 2's `clwb C` has also completed, thus limiting the benefits of inter-task `clwb` overlapping.

The desired effect for task 1's `sfence` is to be *context aware* and only wait for task 1's `stores`/`clwbs`. Instead of waiting for *all* pending `clwbs`, as `sfence` does, a hypothetical context-aware fence (`cfence`) only waits for `clwbs` associated with a specific context/task. Figure 2c illustrates how such a `cfence` can significantly improve the overlap of the two tasks' long-latency persists.

In summary, Figure 2's example conceptually illustrates the opportunity for throughput improvement of task-parallel crash-consistent applications, by overlapping persist latency across independent tasks. Effectively taking this approach from concept to practice comes with two key requirements:

► **Requirement 1:** The overhead of context (task) switching must be sufficiently smaller than a persist's latency.

► **Requirement 2:** CPU hardware needs the ability to distinguish between persists of different tasks, and selectively stall only when persists of the current task are still pending.

Next, we introduce *COSPlay*, our mechanism leveraging software-hardware co-design to meet these two requirements.

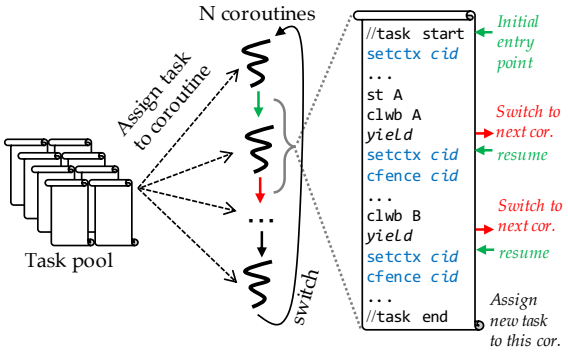


Figure 3: *COSPlay* overview: task assignment to window of *N* concurrent coroutines with round-robin scheduling.

4 HIGH-THROUGHPUT SYNCHRONOUS PERSISTENCE WITH COSPLAY

We take a software-hardware co-design approach to build a system that meets §3's two requirements. At the core of our approach lies the lightweight software control structure of coroutines, which we rely on for rapid context switching. Coroutines are a versatile tool for modern software development, offering as low context-switching latency as 6ns [3], and are becoming increasingly popular in the era of microsecond-scale computing [2]. Coroutine switching is several orders of magnitude faster than traditional OS thread switching, which typically lies in the 5–20µs range [26, 45] (i.e., 1–2 orders of magnitude slower than the slowest persist latency we consider). Due to their agility, coroutines are already being used in various domains for throughput boost via latency hiding, such as in distributed memory systems [23, 33, 35, 48] and databases [15, 20, 38]. As a tool of growing importance, coroutine support has been included in the C++20 standard [8].

Figure 3 provides a high-level overview of *COSPlay*'s operation. We target applications with inherent task-level parallelism. We use a coroutine library (§4.2) to map each task to a coroutine structure and explicitly switch from one coroutine to the next (*yield* in Figure 3) after a synchronous persist operation is issued. We introduce two new instructions (marked in blue in Figure 3 and detailed in §4.1) to associate instructions with different *contexts* and enable the required semantics of a context-specific `sfence`. Finally, simple hardware extensions enable the CPU pipeline to efficiently distinguish among persist operations of different contexts (§4.3).

4.1 ISA Extensions

We introduce two new instructions to expose the notion of different software contexts to the CPU and enable context-aware persist ordering: `setctx cid` and `cfence cid`. A `setctx cid` indicates that all following instructions belong to context `cid`, until a subsequent `setctx cid'`, with `cid'` ≠ `cid`, is encountered. `cfence` is a context-specific `sfence`. A traditional `sfence` orders *all* of its preceding `stores` and `clwbs` (i.e., `stores` are globally visible and `clwbs` have reached the persistent domain), with respect to its trailing `stores` and `clwbs`. `cfence` offers the same guarantee, but only for `stores` and `clwbs` associated with the same context `cid`: a `cfence cid`

can only retire after all preceding stores/clwbs associated with context cid have completed.

4.2 Software Interface

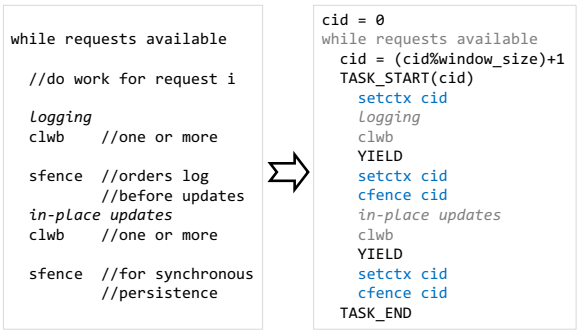
We build *COSPlay*'s API upon a typical coroutine software interface used for task-parallel code. Our implementation uses a high-performance open-source coroutine library [5]. We introduce three macros to define and manipulate task behavior. TASK_START and TASK_END mark the beginning and end of a task, respectively. YIELD invokes the library's coroutine switch functionality and transfers control to our scheduler, which employs simple round-robin scheduling across available contexts.

Figure 4 outlines the transformation from baseline sequential code with available task-level parallelism to task-parallel *COSPlay* code. Black fonts in Figure 4b indicate coroutine code. The TASK_START and TASK_END macros enclose a task that is mapped to a coroutine. The runtime manages *window_size* concurrently active coroutines and cycles through them every time a YIELD is encountered. YIELD invocations must be explicitly placed in the task's source code. For the purpose of hiding persist latency, the process of placing these YIELDs in the code is straightforward: every sfence used for persistence ordering reasons indicates a long-latency event of waiting for clwbs to become persistent. Hence, we place a YIELD before every sfence of the original code, to overlap the clwbs of one context with computation and clwbs of another.

Blue fonts in Figure 4b indicate *COSPlay*'s newly introduced instructions. A setctx cid instruction is placed at the beginning of each task, to implicitly associate the task's following instructions with context cid. A sequence of setctx cid, cfence cid instructions replaces the baseline code's sfence. setctx restores the context id associated with the task and cfence enforces the task's intended persist ordering by blocking until any previously pending stores/clwbs associated with the same cid have completed.

As execution progresses, tasks are gradually assigned to the set of provisioned *window_size* contexts. When a task completes, a new one is spawned in the same context. In our current implementation, the number of used contexts/coroutines (*window_size*) is defined at program launch time. Runtime *window_size* adaptation to dynamic application behavior is an interesting future work direction.

From the programmer's perspective, leveraging *COSPlay* only requires identifying the available task-level parallelism in the original code and explicitly annotating it with our coroutine library macros. Beyond that, the placement of all *COSPlay*-specific additions (blue code in Figure 4b) is standard with respect to coroutine primitive placement and can be incorporated in the coroutine library to remain completely hidden by application programmers. Note that we do not claim that breaking up an arbitrary application into parallel tasks is trivial. Instead, we posit that upgrading applications with inherent task-level parallelism to be *COSPlay*-compatible is straightforward. For example, applications performing transaction processing or dynamic data structure (e.g., hash table, list, tree, graph) manipulations are a good fit, as a transaction or data structure traversal is naturally a self-contained task.



(a) Baseline code (b) COSPlay-enabled code

Figure 4: Pseudocode transformation example. In (b), black fonts indicate standard coroutine code to exploit task-level parallelism; blue fonts indicate *COSPlay* additions to parallelize inter-task persist operations, while preserving the required intra-task persist ordering.

4.3 Hardware Support

COSPlay's modest hardware modifications solely pertain to enabling the cfence instruction semantics for context-aware CPU stalling. A new architecturally visible control register, the *context register*, holds the currently executing context's unique cid. Whenever a setctx cid instruction is executed, the context register is set to cid.

All store and clwb instructions are associated with the context register's contained cid at the time they are issued to the CPU's Store Queue (SQ) and extend the SQ with a cid field per entry. To ensure correct assignment of cid to individual store/clwb operations, the CPU's issue stage observes a read-after-write dependency on the context register between setctx and younger store/clwb instructions.

Due to the lack of publicly available details regarding clwb handling in modern CPU microarchitectures, we hypothesize the existence of the *WriteBack Buffer* (WBB), a structure responsible for tracking the status of pending clwbs. We assume clwbs are issued into the SQ like stores, and are differentiated by setting an *is_clwb* bit. clwb entries have a valid address field, but no data, so they cannot forward values to subsequent loads. As the TSO memory model does not enforce an order between stores and clwbs to different addresses, we assume that a clwb moves to the WBB as soon as it reaches the SQ's head, if there is no pending clwb to the same address in the WBB (to ensure that clwbs to the same address reach the persistent domain at the right order). The clwb stays in the WBB until an acknowledgement that the cache line has reached the persistent domain is received. All clwbs in the WBB can proceed in parallel and may complete out of order, as no ordering between clwbs is required.

Figure 5 illustrates the aforementioned components of the CPU microarchitecture, including our extensions to support *COSPlay*. We extend every SQ and WBB entry with a cid field that associates each store and clwb with a context. The cid fields are used to implement the functionality of our introduced cfence.

The ordering semantics of cfence are implemented using a resource-allocation blocking mechanism. When a cfence cid is

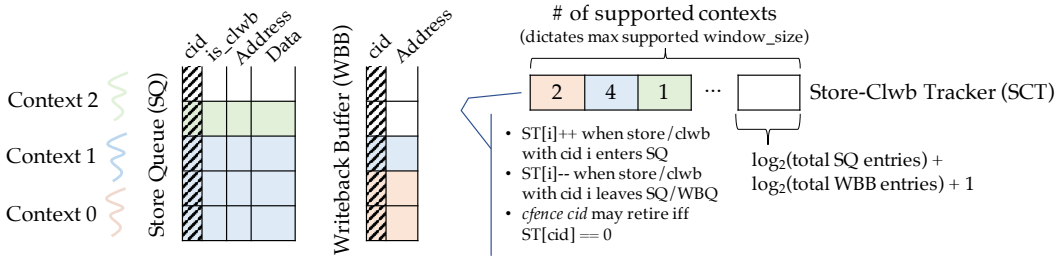


Figure 5: Extensions in CPU pipeline hardware structures for *COSPlay*.

executed, it blocks the allocation of SQ entries while any instructions with the same `cid` still reside in the SQ or WBB. The `cfence` can commit as soon as these instructions are drained. This behavior differs from a traditional `sfence`, which blocks SQ allocation until both structures (SQ and WBB) are *fully* drained. `cfence` thus introduces a higher level of complexity, as instead of simply checking whether the SQ and WBB are empty, it must perform a fully associative search in both structures to look for outstanding operations associated with the context of interest. To avoid this expensive search, we introduce a new structure, the *Store-Clwb Tracker* (SCT). The SCT is a very small, direct-mapped structure that tracks per-context pending `stores` and `clwbs`. The SCT's number of entries dictates the maximum number of supported contexts, and each entry acts as a context-specific counter. Each entry's size is $\log_2(\#SQ\ entries + \#WBB\ entries) + 1$ bits. When a `store` or `clwb` of context *i* enters the SQ, SCT[i] is incremented. Similarly, when a `store`/`clwb` of context *i* leaves the SQ/WBB, SCT[i] is decremented. `cfence cid` blocks until SCT[cid] becomes zero, signifying that all `stores`/`clwbs` associated with context cid have become globally visible/persistent.

Hardware overhead. Table 1 summarizes the per-core hardware added to enable *COSPlay*. For example, a system that features a SQ and WBB of 32 entries each, and supports 16 contexts, requires total SRAM addition of 32×4 bits + 32×4 bits + 16×7 bits = 46 bytes. The context register is accommodated by the out-of-order processor's physical register file by extending the Register Alias Table.

Context (cid) collisions. The coroutine library manages *window_size (WS)* concurrent coroutines, which the hardware maps to *C* hardware-provisioned contexts. While application software may oversubscribe the available hardware contexts by instantiating a $WS > C$, the maximum achievable persist concurrency attainable is $min\{WS, C\}$. Such oversubscription does not raise correctness implications, as *COSPlay*'s `cfence cid` will default to a conservative behavior, effectively bundling all coroutines that use the same `cid` together: `cfence cid` waits for all `stores`/`clwbs` marked with the same `cid` to complete before retiring.

Operational overhead. Microarchitectural additions to enable *COSPlay* are of negligible complexity compared to a typical out-of-order processor. Attaching the `cid` retrieved from the context register to `stores` and `clwbs` introduces an additional register read for these instructions, but does not increase port requirements for the register file, as both instructions have a single source operand

Entity	Added hardware
SQ extension	log_2(# contexts) `cid` bits per entry
WBB extension	log_2(# contexts) `cid` bits per entry

Store-Clwb Tracker	entries: # contexts; n-bit adder
	n = (log_2(# SQ entries + # WBB entries) + 1) bits per entry

Table 1: *COSPlay* per-core hardware extensions.

by default. Compared to `sfence`, the semantics of `cfence` functionally require a more selective search in the SQ and WBB, but the SCT structure allows completing this check with a simple indexed lookup into a tiny structure instead of an associative search. None of these extensions are on the critical path of loads.

4.4 Synchronization Considerations

The order at which updates of different tasks, and thus contexts, are executed is not determined by *COSPlay*, but rather by the—orthogonal to our mechanisms—synchronization primitives used by the programmer. A program with correct synchronization should contain no races in volatile memory. Under our target semantics of synchronous persistence (i.e., a task does not complete before its updates are persistent), this guarantee extends to persistent memory. We assume that all critical sections are appropriately protected against data races: only one task/context can issue `stores` and `clwbs` to a given memory location at a time, and the application waits for persists to complete before it exits the critical section, to achieve synchronous persistence. If a race inherently exists in the program, then there are no guarantees regarding the absolute order of updates to persistent memory, as would be the case even without *COSPlay*'s involvement.

COSPlay is geared towards applications with inherent task-level parallelism that generally employ fine-grained locking mechanisms and are scalable (i.e., conflicting critical sections are infrequent). Nevertheless, special care is required to prevent deadlocks, which would arise if the program switches to a task that tries to acquire an already acquired lock, by blocking or spinning. Following an approach similar to test-and-set, if a context's task attempts to acquire a lock and fails, it immediately yields to the next context. This requirement is not *COSPlay*-specific, as similar care should be taken in any scenario of parallel code leveraging coroutines.

4.5 Putting It All Together

We conclude this section with a step-by-step example demonstrating *COSPlay*'s operation. Figure 6a shows code snippets of two tasks, each assigned to a context, 0 and 1, respectively. For ease

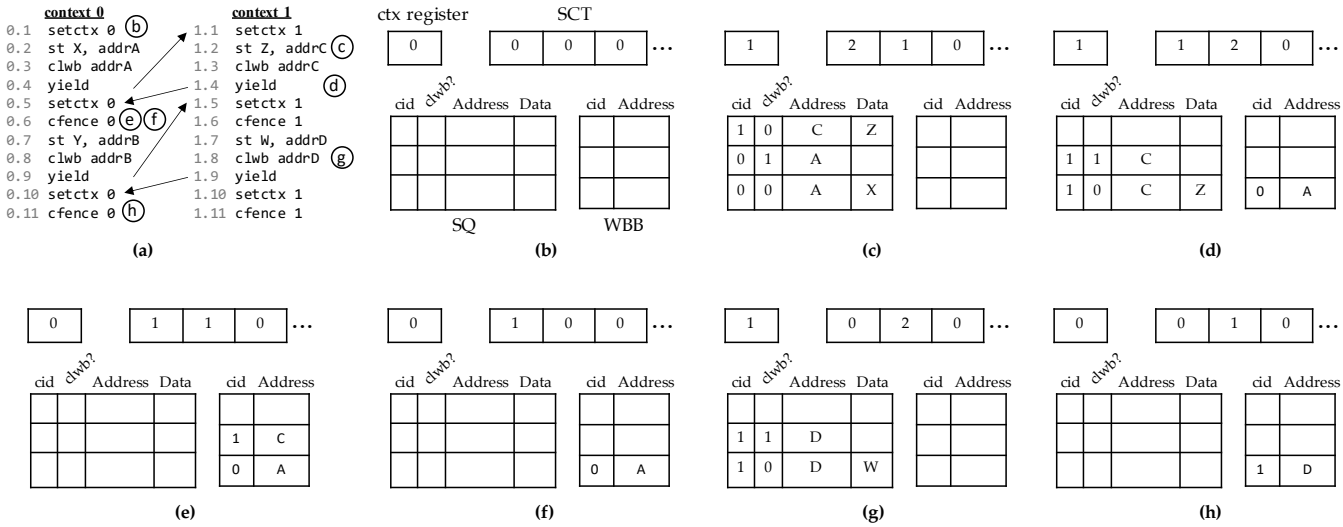


Figure 6: *COSPlay* running example.

of reference, we tag each instruction with `<context id>.<line number>`. Circled letters next to instructions point to the subfigure corresponding to the hardware state when that instruction is executing. For example, ⓑ on Figure 6a indicates that Figure 6b shows the hardware state when instruction `0.1` is executed. In Figure 6b, the context register is set to 0, and the rest of the structures of interest—Store Queue (SQ), Writeback Buffer (WBB), and Store-Clwb Tracker (SCT)—are still unused.

Figure 6c shows hardware state after instructions `0.1`–`0.4` have executed, the runtime has switched from context 0's coroutine to context 1's coroutine, and context 1's instructions `1.1`–`1.2` have executed. `1.1` has set the context register to 1. With a `store` and a `clwb` from context 0 and a `store` from context 1 pending in the SQ, SCT[0] = 2 and SCT[1] = 1.

In Figure 6d, `0.2` has drained to global memory, `0.3` has moved from the SQ to the WBB, and `1.2` has entered the SQ. Thus, SCT[0] = 1 and SCT[1] = 2, corresponding to each context's total pending entries in the SQ and WBB.

In Figure 6e, the runtime has switched back to context 0's coroutine. `0.5` has set the context register to 0, `1.2` has drained to the memory hierarchy, decrementing SCT[1], and `1.3` has moved from the SQ to the WBB. Execution blocks at `0.6`, as `cfence 0` finds SCT[0] ≠ 0, indicating that there are still pending `stores`/`clwbs` associated with context 0.

In Figure 6f, `1.3: clwb addrC` completes and leaves the WBB before the preceding `0.3: clwb addrA`, decrementing SCT[1] to 0. Such `clwb` reordering is allowable and may occur for a range of reasons—for example, if `addrA`'s home location resides in a remote socket's memory range. `0.6: cfence 0` can still not retire, as SCT[0] ≠ 0.

In Figure 6g, `clwb addrA` has completed and left the WBB, allowing context 0's following instructions to proceed. Instructions `0.6`–`0.9` have also completed and the runtime has switched back to context 1's coroutine. `1.5` has set the context register to 1 and `1.6: cfence 1` immediately retires as SCT[1] = 0. Instructions `1.7`–`1.8` add a `store` and a `clwb` in SQ, incrementing SCT[1] to 2.

Finally, in Figure 6h, `0.11: cfence 0` finds SCT[0] = 0 and can thus immediately retire. At this point, the task assigned to context 0's coroutine completes. The runtime switches back to the only remaining active coroutine on context 1, which also completes as soon as the last pending `clwb` of context 1 completes, allowing `1.11: cfence 1` to retire.

Overall, *COSPlay* allows long-latency `clwb` operations from different tasks to overlap, as shown in Figure 6e, and relies on the semantics of our introduced `cfence` to preserve the intended intra-task persist ordering guarantees.

5 METHODOLOGY

Modeled system. We use the ZSim simulator [44] to model a core modeled after the Cascade Lake microarchitecture [17]. All our experiments use a single core, as *COSPlay* is a technique seeking to improve an individual core's utilization. Table 2 summarizes the used simulation parameters.

Modeled BMOs. While we start from a baseline ADR system, where persist latency is only that of reaching the memory controller's writeback queues, our evaluation focuses on systems featuring Backend Memory Operations (BMOs). We consider a number of BMOs used in prior work [29]:

- Encryption (AES-128) [43]: 100-cycle latency.
- Light deduplication (crc-32) [50]: 200-cycle latency.
- Heavy deduplication (md5) [50]: 700-cycle latency.
- Hypothetical BMO combination: 1000-cycle latency.

CPU	x86-64 core modeled after Cascade Lake, 2.2GHz, OoO, 4-wide dispatch/retirement, 224-entry ROB, 56-entry SQ, 32-entry WBB
L1 Caches	Split L1d/i, 32KB 8-way, 64B blocks, 4-cycle access
L2 Caches	1MB, 16-way, 10-cycle access
LLC	Inclusive, 16MB, 16-way, 30-cycle access
NoC	Crossbar, 8-cycle latency
PM	Modeled after Optane [19]: 600-cycle read latency, 200-cycle write latency to controller, 4GBps/2GBps peak read/write bandwidth

Table 2: System parameters for simulation on ZSim.

Name	Description	CKI*	SKI*	CII*
SPS	Random swaps of array elements	21.7	10.9	3.5×
PC	Modify values in a hash-table	20.0	20.0	5.3×
TATP	`update location` transaction in TATP	30.3	30.3	8.1×
TPCC	`New Order` transaction in TPC-C	23.2	2.7	1.2×
CQ	Insert/Delete entries in a queue	19.5	2.3	1.5×

*CKI: `clwb`/kInstruction, SKI: `sfence`/kInstruction, CII: *COSPlay* Instruction Increase

Table 3: Evaluated benchmarks.

We refer to these BMOs as *Enc, LD, HD, Comb*, respectively. For each BMO, the indicated number of cycles is added to the raw latency of a cache line reaching the memory controller. We assume that the system's BMO throughput is sufficiently provisioned to keep up with the available PM bandwidth—i.e., BMOs introduce a latency overhead, but never a throughput bottleneck.

Benchmarks. We use a set of benchmarks that were originally introduced by Kolli et al. [24] and have since been extensively used in PM research [10, 13, 29, 30], summarized in Table 3. The CKI and SKI columns indicate each benchmark's `clwbs` and `sfences` per 1000 instructions, respectively. The CII column shows the increase in per-task dynamic instruction count, due to *COSPlay* coroutine library calls. The smaller the benchmark's tasks and the higher the `sfence` frequency (i.e., more coroutine switching per task), the larger the increase in executed instructions. TATP features the smallest instruction footprint per task, resulting in the highest instruction bloat of 8.1×.

All benchmarks are written in C++ and employ undo logging for crash consistency. A coroutine that attempts to acquire an already held lock yields to allow another task attached to a different coroutine, as described in §4.4. All benchmarks except for CQ leverage fine-grained locking resulting in very low probability of lock collision, and hence inherently high task-level parallelism. In CQ's original implementation, queries access a single shared queue protected by a single lock, effectively limiting the parallelism degree to one. To demonstrate *COSPlay*'s limitations and applicability as a function of available task-level parallelism, we evaluate three versions of CQ*X*, with *X*={*4, 16, 64*}, where *X* denotes the number of queues instantiated. Every new task randomly selects one of the *X* available queues to operate on.

Evaluated system configurations. We employ six system configurations to highlight *COSPlay*'s operation and performance improvements:

- *Baseline* uses undo logging and achieves crash consistency by controlling log and in-place update persist order using `sfences`.
- *Unordered* represents a system that is completely free of any persist-induced CPU stalls and serves as an upper performance bound. By removing all `sfences`, persist latency is never on the critical path, but applications are not crash consistent anymore, as updates to PM are not explicitly ordered in any way.
- *StrandWeaver* is the hardware implementation of strand persistency, modeled after prior work [11]. *StrandWeaver* introduces programming primitives to allow developers to mark independent persists within a task and provisions new hardware structures to allow these independent persists to be overlapped. To achieve synchronous task persistence, a *JoinStrand* operation at the end of each task ensures that all persists belonging to the task have committed before continuing execution. We evaluate the default *StrandWeaver(4,4)* configuration, featuring four strand buffers of depth four each, and an enhanced *StrandWeaver(8,8)* configuration.
- *COSPlay* is our proposed design to hide persist latency by leveraging task-level parallelism, coroutines, and minimal hardware support, as introduced in §4.
- *Software batching (SBatch)* evaluates a software-only technique that leverages task-level parallelism by employing coroutines similarly to *COSPlay*. With a coroutine window size of N, *SBatch* only invokes an `sfence` every Nth coroutine, thus placing persist latency on the critical path only once after cycling through N tasks. Figure 7b graphically illustrates SBatch's operation for $N = 3$ as compared to *COSPlay* with the same window size (Figure 7a).
- *FreeSwitch* is a configuration to assess *COSPlay*'s coroutine switching overhead and an upper performance improvement bound for hypothetical coroutine library implementations with faster switching. We model a zero-cost switch by annotating each application's calls into coroutine switching library code (including the round-robin scheduling logic) and fast-forwarding these instructions in the simulator, thereby ignoring their effect on performance.

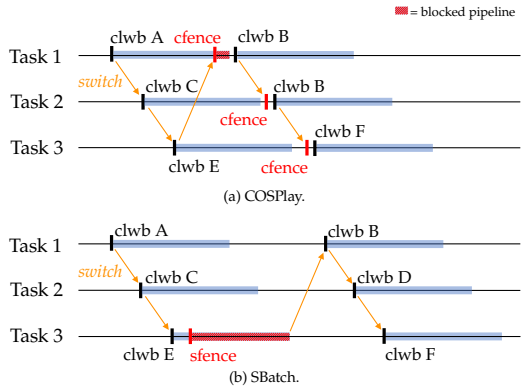


Figure 7: Illustrative comparison of persist overlap effect using *COSPlay* and SBatch (window size = 3).

6 EVALUATION

6.1 Sensitivity to Persist Latency

We start our evaluation by comparing *COSPlay* to *Baseline*. Figure 8 demonstrates the achieved speedup for each of our evaluated benchmarks, for the best-performing window size. Numbers above each bar indicate the window size used in each case. Unsurprisingly, the longer the persist latency, the higher the performance improvement opportunity for *COSPlay*. In addition, the optimal window size grows with BMO latency, as hiding the longer persist latency requires higher concurrency. For plain ADR, *COSPlay* achieves a speedup of up to 1.7×. The benefit for some applications is marginal, as the overheads of coroutine switching largely offset the gain from hiding the relatively low latency of persists. For our shortest evaluated BMO (Enc), *COSPlay* achieves a speedup of 1.04 − 2.24×, which gradually grows with BMO latency to 1.1 − 2.8×, 1.4 − 5.5×, 1.5 − 7.3×, for LD, HD, Comb, respectively.

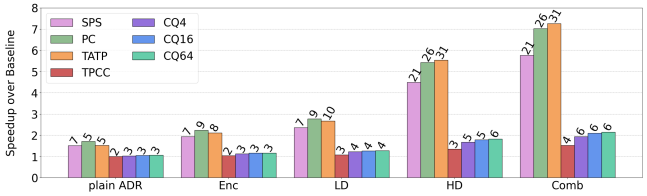
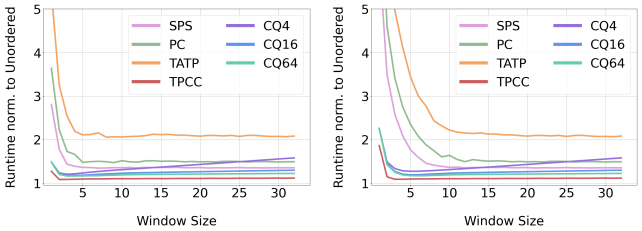


Figure 8: *COSPlay* performance for different persist latencies. Over-bar numbers indicate the best window size.



(a) Enc (100 cycles). **(b) HD (700 cycles).**

Figure 9: Performance sensitivity to window size.

Benchmarks with the highest SKI values—SPS, PC, TATP—benefit the most from *COSPlay*; an expected behavior, as higher SKI values indicate more frequent stalls due to serialized persists. Although TATP demonstrates the highest SKI value, it benefits slightly less than the benchmark with the second-highest SKI, PC, due to a considerably higher instruction count overhead incurred by *COSPlay*. TPCC and CQ both show more modest performance gains, due to their significantly lower SKI value. The maximum achievable speedup for CQ is affected by the available parallelism degree, implicitly dictated by the number of provisioned queues—e.g., 1.7×, 1.8×, 1.8× for CQ4, CQ16, CQ64, respectively, in the case of HD. Interestingly, for the longer BMOs, the best window size for CQ4 exceeds the application's maximum available parallelism degree of four. Due to random queue selection, each task assigned to a coroutine is likely to pick a queue with an already acquired lock. Thus, limiting the number of concurrent coroutines to four leaves available parallelism on the table. For longer BMOs, the gains of window size overprovisioning to maximize the extracted parallelism outweigh the cost of fruitless coroutine switching due to finding a lock already acquired.

6.2 Sensitivity to Window Size

Figure 9 shows *COSPlay*'s sensitivity to window size for Enc and HD, with its performance normalized to the—non-crash consistent—*Unordered* configuration. The general trends we observe are similar for both BMOs. With a window size of one, *COSPlay*'s `cfence` becomes functionally equivalent to a normal `sfence`, as only a single context is at any point active, thus always exposing the persist latency on the critical path. Due to the added overhead of the coroutine library, *COSPlay* with a window size of one performs worse than *Baseline*. As we increase the window size, performance drastically improves, reaching a knee of diminishing returns beyond a window size of 2–5 for Enc and 5–10 for HD. That shift to the right is expected, as hiding the longer BMO latency requires more parallelism.

All applications except for CQ plateau after the window size is sufficient to hide the persist latency. As the window size grows further, it is expected to see this plateau turn into a bathtub due to cache thrashing. Because of the small size and little data locality of tasks, we only start observing an increase in cache miss ratios, negatively affecting performance, at window sizes beyond 32. CQ exhibits a bathtub curve even within Figure 9's window size range, due to its limited number of locks constraining concurrency: a coroutine switch activating a task that finds a lock acquired is pure overhead.

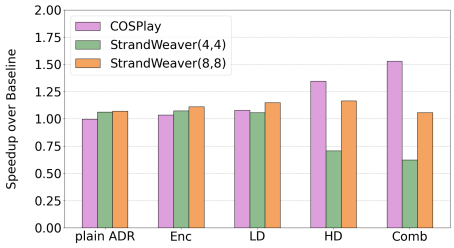


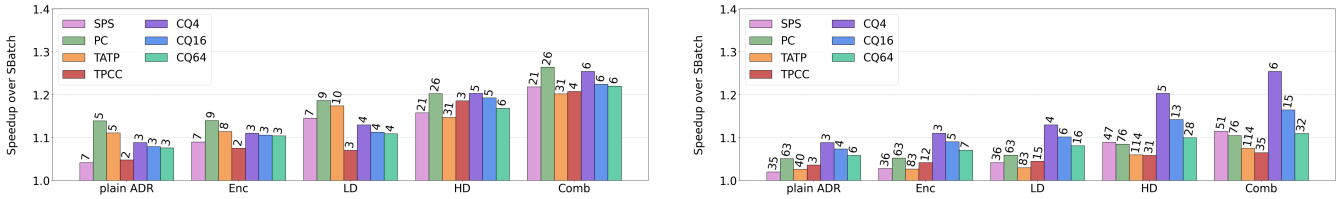
Figure 10: *COSPlay* and StrandWeaver on TPCC.

For Enc and HD, *COSPlay* brings the performance of crash consistent programs within 1.08 – 2.07× of *Unordered*, a significant improvement from the 1.12 – 11.44× performance gap between *Unordered* and *Baseline*, which does not exploit task-level parallelism. The remaining gap from *Unordered* is primarily attributed to the considerable instruction overhead (CII in Table 3) introduced by *COSPlay*'s coroutine management code.

6.3 Comparison to StrandWeaver

StrandWeaver [11] is arguably the prior work most closely related to *COSPlay*. *StrandWeaver* ameliorates the performance impact of persist ordering in two steps. Programmers are trusted to identify *strands*, i.e., subsets of persists within a task that need to be ordered. *StrandWeaver*'s hardware then ensures that intra-strand ordering is maintained, while inter-strand ordering can be relaxed for performance gains. Ordering policies are enforced by an out-of-core unit, the Strand Buffer. In our applications, a set of in-place updates with their corresponding undo log comprise a single strand, as there is a logical dependency that requires the log and in-place update to persist in order. Multiple log/in-place update sets within a task represent independent strands.

All our applications except for TPCC feature a single pair of `clwb` batches for undo logging and in-place updates per task (as per Figure 4(a)'s example code), which need to be ordered, and thus cannot be overlapped. Because of these ordering constraints, such short tasks would amount to a single strand per task, making *StrandWeaver* inapplicable as no inter-strand parallelism can be extracted. In addition, to achieve synchronous persistence (i.e., a task is only considered completed when its updates are persistent), a `JoinStrand` is required at each task's end. `JoinStrand`'s effect is similar to `sfence`'s: it blocks until `clwbs` from *all* previous strands have drained. The compound effect of single-strand tasks and a `JoinStrand` at the end of each task is that the relative *COSPlay*



(a) Iso-window-size comparison. Over-bar numbers indicate the used window size, which is the best-performing window size for *COSPlay*.

(b) Comparison using best window size for each configuration. Over-bar numbers indicate the best-performing window size for *SBatch*.

Figure 11: Speedup of *COSPlay* over *SBatch*.

versus *StrandWeaver* performance for these applications is identical to *COSPlay* versus *Baseline,* shown in Figure 8. However, *StrandWeaver* can accelerate TPCC, where each task comprises a dozen undo logging/in-place update pairs of `clwb` batches.

Figure 10 shows *StrandWeaver*'s performance improvement over *Baseline* for TPCC. For ADR, *StrandWeaver(4,4)* and *StrandWeaver(8,8)* perform similarly, corroborating the original work [11]. However, as persist latency grows, we find that (4,4)'s buffer provisioning depth is not sufficient to accommodate every strand's full batch of `clwbs`, causing the CPU to stall more often than the single `JoinStrand` per task. Ultimately, for the longest persist latencies, this additional blocking results in *Baseline* outperforming (4,4), as *Baseline* allows *all* `clwbs` before an `sfence` to proceed in parallel, while *StrandWeaver* limits intra-strand concurrency to the depth of the Strand Buffer (which in (4,4) is four). This case exhibits how *StrandWeaver*'s statically partitioned hardware resources introduce unnecessary stalls when either of the provisioned depth or width is saturated (i.e., `clwbs` within the same strand exceed buffer depth, or a wider buffer to accommodate more parallel strands is needed). In contrast, COSPlay's SQ and WBB resources are dynamically partitioned as necessary.

StrandWeaver(8,8) provisions sufficient Strand Buffer depth for the examined application to mitigate (4, 4)'s resource limitation, thus enabling both intra- and inter-strand `clwb` concurrency sufficient to hide the latency of longer BMOs. As a result, (8,8) outperforms (4,4) by ~60% for HD/Comb. Ultimately, the single *JoinStrand* blocking point at the task's end caps *StrandWeaver*'s performance; we experimentally verified that overprovisioning Strand Buffer beyond (8, 8), width- or depth-wise, does not help.

For an ADR system, *StrandWeaver(8,8)* outperforms *COSPlay* by 9%, as it reduces CPU stalls for persists from 12 per task to only a single one at the end, without *COSPlay*'s instruction overhead. The performance gap gradually shrinks as persist latency grows, rendering the single `JoinStrand` per task and the associated stall to drain all buffers increasingly detrimental. In contrast, *COSPlay*'s `cfences` enable continuous execution, never requiring an explicit full drain, because they selectively check only for the completion of the associated context's `stores`/`clwbs`. As a result, *COSPlay* outperforms *StrandWeaver(8,8)* with long BMOs—by 22%/55% for HD/Comb, respectively.

Ultimately, in terms of performance, neither StrandWeaver nor *COSPlay* is always strictly superior, as the outcome depends on the nature of the target application and the system's configuration. In the absence of task-level parallelism (TLP), *COSPlay* is not applicable. In contrast, if persist latency is high or the application features short tasks with the requirement of synchronous persistence, *COSPlay*'s TLP-extracting approach is essential to boost throughput. Furthermore, *COSPlay* requires significantly simpler hardware extensions, and relies on the familiar to programmers and already widely employed x86 persistency model. Combining StrandWeaver with coroutines to leverage TLP is not effective, as StrandWeaver's only primitive to synchronize volatile execution with persistence (`JoinStrand`) waits for *all* previous `stores`/`clwbs` of *all* strands to persist before unblocking the core. In contrast, *COSPlay*'s `cfence cid` primitive allows the CPU to selectively block for only pending persists of the current context.

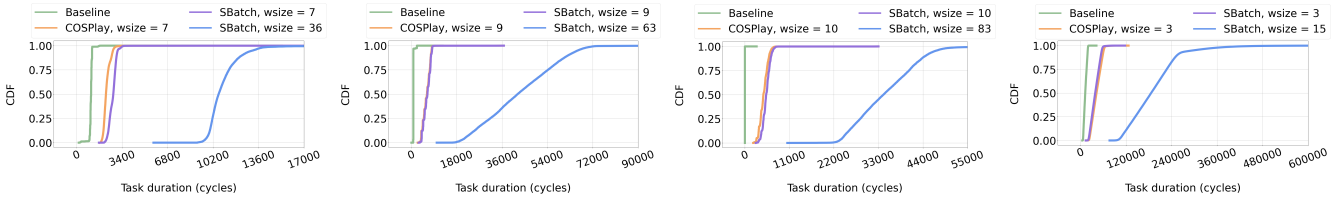
6.4 Comparison to Software Batching

COSPlay predominantly relies on a software mechanism to overlap persist latencies across multiple tasks, but also employs modest hardware extensions to do so effectively. *SBatch* is an alternative software-only approach that, similarly to *COSPlay*, uses coroutines to leverage task-level parallelism, but without any hardware support. Instead, it relies on batching persists of multiple tasks, as described in §5 and graphically illustrated in Figure 7b.

Figure 11 compares the speedup achieved by *COSPlay* over *SBatch*. Figure 11a shows an iso-window-size comparison, using the best window size for *COSPlay*, indicated by the over-bar numbers. For an ADR system, *COSPlay* outperforms *SBatch* by 1.04 – 1.14×, growing with BMO latency to 1.20 – 1.26× for Comb. While *SBatch* reduces the number of CPU-blocking persist operations by a factor of N, *COSPlay* with a sufficient window size can potentially eliminate them completely.

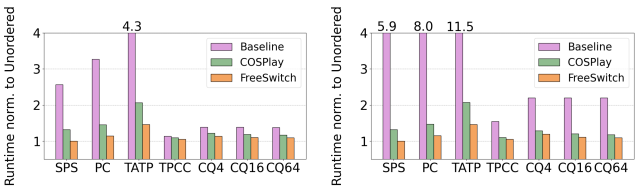
Figure 11b compares the performance of *COSPlay* and *SBatch* when the best-performing window size for each configuration is used. Over-bar numbers show the best window size for *SBatch*; remember that the best window size for *COSPlay* appears in Figure 11a. With the exception of CQ4 and CQ16, which are bound by limited concurrency, the performance gap between the two configurations drops to 1.02 – 1.11× for non-concurrency-bound applications. However, achieving that performance gap reduction requires *SBatch* to employ window sizes up to 12× larger than *COSPlay*, which comes at the cost of significantly higher individual task latency—a well-known tradeoff associated with batching.

Figure 12 shows the CDF of individual task duration. For brevity, we only show the CDFs for four of the applications and only for the LD system configuration. As demonstrated in Figure 11a and



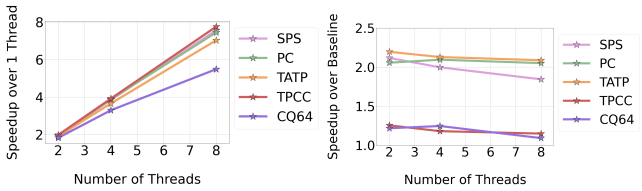
(a) SPS. (b) PC. (c) TATP. (d) TPCC.

Figure 12: Task latency CDFs for *COSPlay* and *SBatch* for light deduplication (LD) BMO (200 cycles).



(a) Enc (100 cycles). (b) HD (700 cycles).

Figure 13: Opportunity for faster context switching.



(a) Baseline scalability. (b) *COSPlay* speedup over baseline.

Figure 14: Performance impact of *COSPlay* on multicore deployment for system with light deduplication (LD) BMO.

11b, *SBatch* requires a drastically larger window size to approach *COSPlay*'s performance. Such increased window size results in *SBatch* exhibiting a median task latency increase of 4 – 8× and even more for larger percentages. Even for the same window size, *COSPlay* yields marginally lower (4 – 25% for LD) task latency than *SBatch*, as *SBatch*'s 1 in N blocking tasks slightly shifts the whole latency distribution to the right. While we only show LD in Figure 12, the individual task latency gap between *SBatch* and *COSPlay* is even wider for longer-latency BMOs, as *SBatch* employs even larger batching degrees.

6.5 Context Switching Overhead

Figure 13 shows the performance of *Baseline*, *COSPlay*, and *FreeSwitch* compared to *Unordered* for Enc and HD. Starting with Figure 13a, for the applications most sensitive to persist latency—SPS, PC, and TATP—*COSPlay* captures most of TLP's potential, improving throughput by 1.9 – 2.2×. Eliminating switching overhead can offer an additional improvement of 1.3 – 1.4×. For the least sensitive application, TPCC, switching overheads are significant: *COSPlay* improves performance over *Baseline* by just 4%, while *FreeSwitch* shows an additional 4% potential. CQ lies in between: *COSPlay* is 14% faster than *Baseline*, and *FreeSwitch* showing an additional 7% improvement opportunity. The remaining performance gap between *FreeSwitch* and *Unordered* is attributed to coroutine overheads other than switching and scheduling (which are skipped by *FreeSwitch*) and, to a lesser extent, to cache interference.

The performance gaps shift in presence of longer BMOs, as Figure 13b demonstrates for HD. *FreeSwitch*'s relative performance improvement opportunity over *COSPlay* remains very similar as in Enc. However, *COSPlay* captures most of the performance gain opportunity for all applications, due to the significantly reduced relative cost of coroutine switching as compared to persist latency.

Such analysis can help inform a decision about investing in hardware for faster context switching versus a lightweight software-focused approach with small hardware additions like *COSPlay*. For instance, a CPU with coarse-grained multithreading (CGMT) could functionally achieve the same effect as *COSPlay* with faster context switching at the cost of hardware. *FreeSwitch* with window size N could also be interpreted as a rough approximation of an N-way CGMT CPU, although, as previously mentioned, there are additional overheads introduced by the coroutine library that *FreeSwitch* does not eliminate. A key tradeoff for the design of a multithreaded core is the partitioning of its LSQ [40, §11.4.4.1], sharing conceptual similarities with StrandWeaver's Strand Buffer provisioning considerations (§6.3). Per-thread partitioned provisioning could allow context-aware ordering like the one achieved by *COSPlay*'s `cfence`, but limits the maximum LSQ capacity usable by each individual thread. *COSPlay* does not introduce such tradeoff. Finally, although a CGMT CPU would be feasible for low BMO latencies like Enc, which require window sizes of 2–9 for latency hiding, the cost would be prohibitive for longer ones like HD, where optimal window sizes fall in the 3–31 range (see Figure 8).

6.6 *COSPlay* and Multithreaded Execution

Our evaluation so far focused on single-core application deployments. However, as the applications *COSPlay* targets exhibit task-level parallelism, they can be trivially deployed in multi-threaded mode on multicore systems and achieve good scalability, as demonstrated in Figure 14a. *COSPlay* is orthogonal to multithreading, and can be combined with it to further boost application throughput. Figure 14b shows the achieved speedup as a function of application threads (each running on a dedicated core), when each thread employs *COSPlay*. Unsurprisingly, *COSPlay*'s achieved speedup diminishes with the number of threads, as higher concurrency promotes PM bandwidth contention and synchronization overheads to considerable performance determinants. Even when using 8 threads, *COSPlay* still delivers noticeable throughput gains, especially for applications with high task-level parallelism and fine-grained tasks. With only minimal hardware additions, *COSPlay* improves the

utility and efficiency of area- and power-hungry aggressive OoO cores.

7 RELATED WORK

Persistency models. The implemented persistency model dictates not only the semantics exposed to the programmer, but also the impact of a PM's persist latency on performance. Pelley et al [37] provide one of the first persistency model taxonomies, discussing strict, epoch and strand persistency. The predominant x86 persistency model is a form of epoch persistency, which relaxes the order of persists within the same epoch to improve persist overlap and hence performance. However, the remaining synchronization points still hamper performance, motivating approaches with looser persistency semantics. StrandWeaver [11] implements strand persistency [37] by diving programs into strands that can persist out of order and ensuring the expected by the programmer ordering in hardware. We extensively discussed similarities and differences of StrandWeaver and *COSPlay* in §2.1 and §6.3. DPO [25] implements a buffered strict persistency model, whereas BPFS [7] and HOPS [32], among others [1, 9, 21, 24, 42], implement different versions of buffered epoch persistency by utilizing buffers that maintain the desired order of persist operations through the memory hierarchy. Such buffering approaches remove the persist latency from the CPU's critical path and eschew program execution stalls while waiting for a persist to happen, but require extensive and often complex additions to the caches and the PM controller. More importantly, such decoupling of volatile and persistent state complicates persistency semantics, as a program's persistent state can lag arbitrarily behind its volatile state, resulting in *asynchronous* rather than *synchronous* persistence. *COSPlay* focuses on the semantics of the x86 persistency model and synchronous persistence, but its applicability is limited to applications with task-level parallelism.

Scoped fences and barriers. *COSPlay*'s `cfence` is a form of selective `sfence` operation. Fences and barriers with selective scopes have been considered before both in the context of parallel programming [27] and more specifically in the context of PM. Gope et al [12] propose barriers with different scopes for GPU programming with PM, while work on buffered epoch persistency models consider uni-directional acquire/release fences to express finer-grained persist ordering than full `sfences` [9, 24]. *COSPlay*'s `cfence` preserves simplicity as its semantics are identical to an `sfence` within its logically independent task.

Crash consistency. Crash consistency is typically achieved through a form of logging. Applications in our evaluation rely on undo logging, but *COSPlay*'s approach is equally applicable with redo logging or with the use of higher-level libraries that hide this complexity from the programmer [4, 6, 14, 16, 31, 47]. A large body of work aims to alleviate logging overhead via hardware support. Themis [30] observes that non-temporal stores commonly used in logging often naturally persist faster than temporal stores, and proposes hardware extensions to guarantee that log-update ordering is always preserved without placing an `sfence` between them. PiCL [34] automatically generates undo-log checkpoints in the cache and controls their propagation order to PM, while opportunistically batching them to improve bandwidth utilization. ATOM [22] employs a log-manager module that controls and coordinates the logging procedure. Proteus [41] introduces new ISA primitives and CPU modifications to create and manage logs. Kiln [49] and LAD [13] remove logging altogether by speculatively buffering a transaction's updates in a persistent space (in the LLC and memory controller, respectively) and atomically committing them to the persistent domain once the transaction successfully completes. *COSPlay* is agnostic to the persist operation type (logging versus in-place update) and aims at hiding the latency of *any* such operation that may be exposed on the CPU's critical path; thus, it can be combined with any other technique that does not completely eliminate synchronous persists.

Latency hiding with coroutines. Coroutine-based programming has recently seen an increase in popularity as an effective approach to hide long-latency events (e.g., cache misses) in throughput-oriented applications that exhibit low memory-level parallelism but high task-level parallelism. Utilizing coroutines successfully for that purpose requires the programmer or compiler to statically determine when an event will incur high latency, so that a switch to a different coroutine can be scheduled. Prior work employs coroutines on data structures with known poor locality and chained-access patterns, replacing loads known to miss in the caches with prefetch instructions and a coroutine switch, thus overlapping such long-latency prefetches with another coroutine's computation. Grappa [33], DrTM+H [48], and FaSST [23] are recent examples of software distributed shared memory systems employing coroutines to hide network latency. Psaropoulos et al [38] leverage coroutines to interleave index traversals for database join operations, while Jonathan et al [20] extend the same approach to a number of different pointer-chasing data structures, such as B+ trees and hashtables. He et al [15] build a full database around the latency-hiding capabilities enabled by coroutines. As a latency-hiding technique, coroutines have also been used in PM-based systems, in an attempt to bridge the latency gap between volatile and non-volatile memory [39, 46]. Van Renen et al [46] amortize the high latency cost of frequent synchronous persist operations via the software batching approach evaluated in §6.4.

8 CONCLUSION

We presented *COSPlay*, a software-hardware co-design that combines coroutines for rapid context switching with light hardware modifications to accelerate crash-consistent applications exhibiting task-level parallelism. A key benefit of the mechanism is that it operates under the predominant x86 persistency model and preserves the strict semantics of synchronous persistence. Our evaluation and comparison against a range of alternative software and hardware approaches demonstrated that *COSPlay* represents an appealing point in the design space, striking a balance between throughput, individual task latency, and hardware requirements.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback, which helped improve this paper. We thank Aasheesh Kolli for valuable technical discussions in the early stages of this research and Anirudh Jain for his assistance with the Pin tool.

REFERENCES

[1] Mohammad A. Alshboul, James Tuck, and Yan Solihin. 2018. Lazy Persistency: A High-Performing and Write-Efficient Software Persistency Technique. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*. 439–451.

[2] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[3] boost-coroutines [n.d.]. Performance of coroutine switching in the Boost 1.74.0 C++ library. https://www.boost.org/doc/libs/1_74_0/libs/context/doc/html/context/performance.html

[4] Dhruva R. Chakrabarti, Hans-Juergen Boehm, and Kumud Bhandari. 2014. Atlas: leveraging locks for non-volatile memory consistency. In *Proceedings of the 29th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 433–452.

[5] co2 [n.d.]. CO2 - Coroutine II: A C++ await/yield emulation library for stackless coroutine. https://github.com/jamboree/co2

[6] 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 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVI)*. 105–118.

[7] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. 133–146.

[8] cpp-coroutines [n.d.]. Coroutines in C++20. https://en.cppreference.com/w/cpp/language/coroutines

[9] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. 2020. Lazy Release Persistency. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*. 1173–1186.

[10] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for synchronization-free regions. In *Proceedings of the ACM SIGPLAN 2018 Conference on Programming Language Design and Implementation (PLDI)*. 46–61.

[11] 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 47th International Symposium on Computer Architecture (ISCA)*. 652–665.

[12] Dibakar Gope, Arkaprava Basu, Sooraj Puthoor, and Mitesh Meswani. 2018. A Case for Scoped Persist Barriers in GPUs.. In *Proceedings of the 11th Workshop on General Purpose GPUs*. 2–12.

[13] 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)*. 466–478.

[14] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*. 775–788.

[15] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: Coroutine-Oriented Main-Memory Database Engine. *Proc. VLDB Endow.* 14, 3 (2020), 431–444.

[16] 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 2017 EuroSys Conference*. 468–482.

[17] Intel [n.d.]. Intel Cascade Lake Microarchitecture. https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake

[18] Intel Corporation. 2020. Intel® architecture instruction set extensions and future features programming reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf

[19] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019).

[20] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin J. Levandoski, and Gor V. Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proc. VLDB Endow.* 11, 11 (2018), 1702–1714.

[21] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient persist barriers for multicores. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 660–671.

[22] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *Proceedings of the 23rd IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 361–372.

[23] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*. 185–201.

[24] Aasheesh Kolli, Vaibhav Gogte, Ali G. Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-level persistency. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*. 481–493.

[25] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali G. Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016. Delegated persist ordering. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 58:1–58:13.

[26] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. In *Experimental Computer Science*. 2.

[27] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. 2014. Fence Scoping. In *Proceedings of the 2014 ACM/IEEE Conference on Supercomputing (SC)*. 105–116.

[28] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*. 329–343.

[29] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Manabi Khan. 2019. Janus: optimizing memory and storage support for non-volatile memory systems. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. 143–156.

[30] Sara Mahdizadeh-Shahri, Seyed Armin Vakil-Ghahani, and Aasheesh Kolli. 2020. (Almost) Fence-less Persist Ordering. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 539–554.

[31] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*. 789–806.

[32] 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 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*. 135–148.

[33] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*. 291–305.

[34] Tri Minh Nguyen and David Wentzlaff. 2018. PiCL: A Software-Transparent, Persistent Cache Log for Nonvolatile Main Memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 507–519.

[35] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos K. Aguilera. 2019. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*. 97–108.

[36] optane [n.d.]. Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html

[37] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*. 265–276.

[38] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *Proc. VLDB Endow.* 11, 2 (2017), 230–242.

[39] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the Latency Gap between NVM and DRAM for Latency-bound Operations. In *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN)*. 13:1–13:8.

[40] J.P. Shen and M.H. Lipasti. 2013. *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press.

[41] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: a flexible and fast software supported hardware logging approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 178–190.

[42] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the Long Latency of Persist Barriers Using Speculative Execution. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*. 175–186.

[43] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 339–350.

[44] Daniel Sánchez and Christos Kozyrakis. 2013. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*. 475–486.

[45] Dan Tsafrir. 2007. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Experimental Computer Science*. 4.

[46] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Building blocks for persistent memory. *VLDB J.* 29, 6 (2020), 1223–1241.

[47] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVI)*. 91–104.

[48] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*. 233–251.

[49] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 421–432.

[50] Pengfei Zuo, Yu Hua, Ming Zhao, Wen Zhou, and Yuncheng Guo. 2018. Improving the Performance and Endurance of Encrypted Non-Volatile Main Memory through Deduplicating Writes. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 442–454.