JASS: A Tunable Checkpointing System for NVM-based Systems

Abstract—**Checkpointing (or snapshotting) a system's state has always been a problem of great interest and has found a lot of use in ensuring system reliability, record-replay debugging, job migration and running high-throughput transaction systems. In the last few years ultra-fast hardware-assisted NVM-based checkpointing schemes have come up that can collect incremental full-system checkpoints in milliseconds. Unfortunately, such systems have large overheads in terms of their write amplification (increased number of writes). This, in turn, seriously reduces the reliability and lifetime of NVM devices. We propose the first tunable scheme in this space, JASS, where given a checkpoint latency (*CL*), we near-optimally minimize the write amplification (*WA*). This allows us to run parallel programs in a *disciplined fashion*. To realize this goal, we propose many novel hardware along the way such as a rigorous method of flushing pre-checkpoint messages in the NoC, a novel DRAM scrubber and locality predictor, and a control-theoretic algorithm to guarantee a *CL* while minimizing the *WA*. We reduce *WA* by 35-96% as compared to the nearest state-of-the-art competing method and improve performance of PARSEC benchmarks by 19.4%.**

I. Introduction

Checkpointing (or snapshotting) is a classical problem in computer systems. It finds wide applications in record-replay debugging, error analysis, system backup and replication, and crash recovery. Over the years, checkpointing latencies are becoming smaller and at the same time, newer applications of ultra-fast checkpointing are also coming up. As of today, the best software-based checkpoint latencies are in the 300-500 ms range and are thus the methods of choice for Level 1 fault recovery methods in computing systems, notably HPC systems and data centers. The popular schemes in this space are CRIU [1], ULFM [2], Legio [3] and ReInit++ [4].

In the last few years, NVM-based checkpointing solutions that additionally rely on bespoke checkpointing hardware have been proposed. They are disruptive inventions because they bring down the checkpoint latency to a few milliseconds [5], [6], [7], [8], [9]. This is not exactly new in the HPC community. Way back in the times of the Bluegene/P system [10], checkpointing times were recorded to be of the order of milliseconds for low-memory-footprint workloads. However, the modern schemes are more generic (workload independent) and use fast on-chip hardware to collect checkpoints on persistent memory, as opposed to traditional in-memory checkpointing. Typically, they also collect incremental full-system checkpoints per node (including the OS state). There are many applications now that have begun to use such ultra-fast checkpointing schemes such as record-replay debugging[11], system backup and recovery[12], high-throughput transaction systems[13], [14], and ultra-fast job migration in exascale systems[15]. Given that exascale systems are expected to fail once every few hours or maybe once an hour on an average [16], the need for ultra-fast per node checkpointing is obvious. In our view, unless the Level 1 checkpoint time (per node) is of the order of milliseconds, it will be very hard to collect a *global* (semi)coordinated checkpoint quickly. Finally, note that with these requirements, an incremental, full-system checkpointing method is quite suitable to HW-based approaches because we can run our full software stack unmodified or at best add one or two instructions to tell the HW when to checkpoint.

Let us describe the problem formally. There are four variables of interest: *CL* (checkpoint latency), *ES* (epoch size *or* checkpoint interval), $MTBF$ (mean time between failures) and *WA* (write amplification). In our case, the system continues to execute during the checkpointing process. Let CL_{down} be the system downtime during checkpointing, which in our case tends to zero. Classical theory [17] says that the optimal epoch size (ES_{opt}) should be equal to $\sqrt{2 \times CL_{down} \times MTBF}$, if we wish to minimize the overall execution time. Given that $CL_{down} \rightarrow 0$ for us, $ES_{opt} \rightarrow 0$, which does not make sense because $ES \geq CL$. Hence, we need to set ES based on the application's requirements; in fact if $CL_{down} = 0$, $ES_{opt} = CL$ (keep checkpointing all the time). For instance, if a transaction takes 2 milliseconds [13], [14], and we wish to log a batch of 5 transactions before making them visible, then $ES = 10$ms, or if we want to debug at most 7 ms of an application's execution, then $ES = 7$ms.

Now, a low millisecond-scale *CL* does not come for free. We need to incur a lot of additional writes in the non-checkpointing phase of the application to ensure that when the time for checkpointing arises, the latency of the latter process is bounded by the chosen *CL*. These extra writes increase the *WA* (write amplification). We shall show in Section VI that *WA* increases super-linearly (roughly quadratically) with a linear decrease in the *CL*. The endurance of NVMs is known to be very sensitive to the number of writes and its reliability quickly decreases with increased *write stress*[18]. Hence, given an *ES*, our problem basically boils down to optimally trading off *CL* and *WA* (see [18], [19]). We thus set the value of a *CL* (based on user requirements) and minimize the *WA*.

Our proposal JASS embodies the basic realization that we *amplify writes* (create more writes to the NVM) throughout an epoch such that our target *CL* can be achieved at the epoch boundary. We shall see in Section VI that to achieve a reduction of even 1 ms in the *CL*, we will have to pay a big price in terms of the *WA* (2 $\times$). Our first contribution is a system that allows us to operate anywhere in the region of the feasible

$\langle CL, WA\rangle$ values for a workload and a given value of *ES*. In other words, given a pair of *CL* and *ES* values, we make a best effort to meet it and also minimize the *WA*. No other competing work supports this *tunability aspect*.

Many more innovations are possible because of this design philosophy. Existing NVM-based checkpointing schemes [8], [6], [7] checkpoint at the granularity of epochs (like us), however, while doing so, they introduce a very complex system that requires them to tag cache lines and sometimes DRAM rows with multiple bits, persist when there is a coherence write or eviction, maintain the state of many epochs at the same time and have elaborate data coalescing algorithms. We design a scheme that is far simpler, where the crux of our algorithm is a coherent cache flushing scheme using a variant of the classical Chandy-Lamport algorithm [20] for distributed systems. We need to maintain only a single bit per cache line and require no changes to be made to the DRAM; we propose a simple closed-loop control algorithm to ensure that the *CL* target is met while minimizing the *WA*. We compare JASS with the nearest competing work, NVOverlay [6], and show that for its constraints, our *WA* is $3\times$ lower. Given a *CL* target, we never overshoot (or undershoot) by more than 5%.

Novel ideas in our proposal: JASS

1) A novel cache flushing scheme that flushes all in-flight pre-snapshot messages from the NoC and pre-snapshot data from the caches using a variant of the Chandy-Lamport algorithm. This often overlooked aspect of cache flushing [8] is quite difficult to correctly and comprehensively implement in practice, as we found out during the course of this work.
2) A locality predictor that decides whether to keep a page in DRAM or not based on some tunable hyperparameters.
3) A method to dynamically *tune* those hyperparameters such that the target *CL* is achieved while minimizing the *WA*.
4) For the same *CL*, we reduce the *WA* by 35-96%, as compared to the state-of-the-art.
5) Because of our simpler design, we also speedup a suite of PARSEC benchmarks by 19.4% (a collateral benefit of our scheme arising from its simplicity).

Section II describes the relevant background, Sections III and IV elaborate on the design; we present our results in Section VI, described related work in Section VII and finally conclude in Section VIII.

II. Background

The nonvolatile nature of byte-addressable Nonvolatile Memory (NVMs) has inspired different approaches to use them. Most software approaches implement some form of logging or transactions, while most hardware approaches define and restrict the order of stores to memory. In order to understand these approaches, we must look at the characteristics of NVMs first.

A. NVM Cell Characteristics

All popular types of NVM cells (ReRAM, FeRAM, STT-RAM, PCM, NAND flash) have a smaller write endurance (roughly 1000X lower) when compared to DRAM [18]. When the number of writes to a cell exceeds the endurance threshold, the cell loses its ability to retain data without applied power (*persistence*). Since the major benefits of using an NVM device is *persistence*, repeated writes to the same cells are extremely problematic. As a result, **wear-leveling** is implemented to ensure that every cell sees roughly the same number of writes.

B. Wasted Writes

Although NVM devices placed on the memory bus are byte-addressable, they have an internal hardware buffer (for reading and writing) that caches data at the granularity of blocks [21]. Writes to this block are coalesced by this buffer. A block switch causes a write-back of a modified block to a cell at a different location. This relocation of blocks is part of a device-side wear-leveling technique. We formally define *write amplification* as a ratio: number of block-level writes made by the target system divided by the number of byte-level writes **intended** by the target system (assuming no checkpointing and logging).

C. Place in the Memory Hierarchy

Since NVMs are a new memory class, their position in the hierarchy is critical. Figure 1 shows the different options. Most research papers place DRAM and NVM at the same level [19], i.e., horizontal integration. This provides greater flexibility, as applications can divide their address space into persistent and non-persistent zones. The resulting problem of *data placement* is handled by software. A system can opt for *replacement integration*, replacing DRAM with NVM. Although cost-effective, such a system suffers from long read and write delays to main memory. In this paper, we opt to design a system with the more popular horizontal integration.

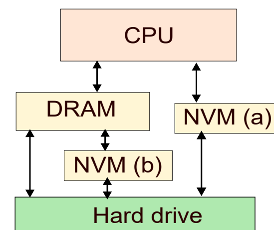

Fig. 1. Placement of the NVM in the memory hierarchy. (a) Horizontal integration puts NVM and DRAM on the same level. (b) Vertical Integration places NVM below DRAM.

D. Chandy-Lamport Snapshot

JASS collects a snapshot of a running system using a (modified) Chandy-Lamport distributed snapshotting algorithm as shown in Algorithm 1. The Chandy-Lamport snapshotting algorithm is initiated over the NoC by any node. It could also be initiated upon a message received from a remote machine (in the case of distributed coordinated checkpointing). Point-to-point links in the network are assumed to be FIFO, which is the case for an NoC.

The main idea is to propagate token messages on every link. If a node has received a token, it acknowledges the receiving

Algorithm 1 Chandy-Lamport algorithm

1: **if** Snapshot token received for the first time **then**
2: Take local snapshot.
3: Propagate token to all neighbors.
4: Send an acknowledgment to the sender.
5: **else if** Snapshot token received more than once **then**
6: Send an acknowledgment to the sender.
7: **else if** Token received and message arrives on a non-token/non-ack receiving link **then**
8: Record message.
9: **end if**

of the token. Upon the first arrival of a token at a node, a local snapshot is taken at that node. This snapshot records the local state. Since the arrival of a token signifies that the sender has taken a snapshot, any subsequent messages on that link are post-snapshot messages and are not recorded.

Note that all pre-snapshot messages (prior to arrival of the token) need to be recorded and are a part of the checkpoint.

III. JASS: Overall Design

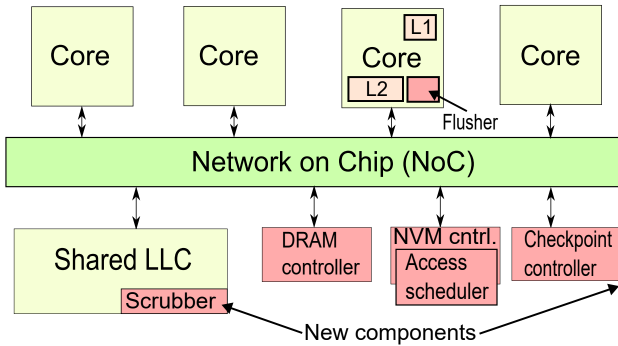

Fig. 2. Design of the overall system. The red (darkly shaded) regions are the new components in JASS.

JASS is a full-system *user-transparent* snapshotting system. The overview of the system is shown in Figure 2. Figure 3 shows the sequence of events as they occur within JASS. Just like our predecessors [5], [6], [7], [8], we are *epoch-persistent*. The system can be seamlessly started from its last epoch boundary (standard assumption).

Unlike previous works where multiple epochs can be alive (unpersisted) at the same time, we need to maintain the state for only 2 epochs (1-bit state): pre-snapshot and post-snapshot. This reduces cache pollution. At the end of the current epoch, the sense *reverses* (post-snapshot in the current epoch becomes pre-snapshot in the next epoch). Every cache line is tagged with this bit; however, the main memory (DRAM) or the transfer protocol are left unmodified (quite unlike NVOverlay and Donuts). Both epochs have separate DRAM page (simply called a page henceforth) tables that refer to the modified working sets of the epochs. A DRAM scrubbing scheme persists the modified pages for the epoch that is being persisted. **Note** that a *page* is defined as a 256-byte region of contiguous memory in this paper (not the 4 KB virtual memory page, which we refer to as a VM-page). The persist path is the same as the store path (caches $\rightarrow$ DRAM $\rightarrow$ NVM); this reduces *WA*.

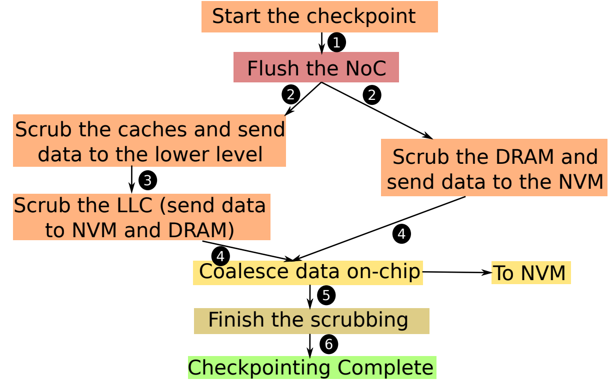

Fig. 3. Sequence of events when a checkpointing operation begins. Arrows with the same number indicate the events start executing simultaneously.

A. Checkpoint Initiation

JASS has two ways to initiate a checkpoint. A clock-driven method takes a periodic checkpoint of the system. If the system decides that, in the rare event of a failure, it is okay with losing a maximum of η minutes of work, we set the period between checkpoints to η minutes. This is the epoch size, *ES*. The system sets this value, and we expose this using a privileged instruction. This mode can be turned off if periodic checkpoints are not desirable. The other method is to initiate a checkpoint before an *event of interest* like a system call or I/O instruction. A machine can also initiate a checkpoint in a large distributed system when it gets a message from a remote node.

B. Process of Taking a Checkpoint

We can divide the process of taking a checkpoint into three distinct phases: ① Flush the pre-snapshot messages in the NoC and the lines in the high-level caches to the LLC, ② scrub the caches and write the changes to the NVM, ③ scrub the main memory and move all the modified pages (since the last epoch boundary) to the NVM.

C. Incremental Checkpoints and Recovery

JASS takes incremental checkpoints. The NVM device has a page table in a known location that is used during recovery. We use atomic logging (similar to [6]) to ensure recovery of the page table data structure. Post-crash recovery is simple but time-consuming. First, we read the page table and move pages to DRAM one by one. When this is done, we restore the register state in all the cores. The system can then resume.

IV. JASS: Detailed Design - Caches

As described in the previous section, a checkpoint can be initiated for a variety of reasons: periodic timer interrupt, before a system call or I/O operation, or as a part of a coordinated activity in a distributed system. Regardless of the reason, the flow of actions is the same. A checkpoint controller starts the process and initiates the Chandy Lamport algorithm by sending a checkpoint message (snapshot token) to its neighboring nodes on the NoC (see Section II).

A. The Process of Snapshotting

Upon receiving a snapshot token at a router on the NoC, the router checks if it has received one before. If it has not received a token, the connected elements are notified to take a snapshot, and the router propagates the token to all its neighbors. Every core has a single bit initialized to zero. This bit is attached to every message and cache line in the system. Let us call this bit the *snapshot* bit. Consider the case when we are taking the first snapshot.

If the snapshot bit is 0, we know that the message/cache line associated with the bit is pre-snapshot. Once an element is made aware of an ongoing snapshot, all further message generation and cache modifications happen by appending a 1 bit instead of a 0 bit, thereby distinguishing between pre-snapshot and post-snapshot data/messages.

After snapshot completion, we can change the sense of the bits: now 1 stands for a pre-snapshot state and 0 stands for a post-snapshot state.

1) Snapshotting Cores and Private Caches: Snapshotting a core is simple. We first flush the pipeline and then store the architectural register state in a known location in the NVM. The core is thus assumed to be checkpointed. All subsequent memory writes are deemed to be post-snapshot writes. Snapshotting a cache also entails changing the state of future evictions to post-snapshot writes. We assume that both the cores and caches contain a snapshot bit, whose sense changes every epoch.

B. Flushing Messages on the NoC

Correctly flushing pre-snapshot messages from the NoC is complicated and even though it looks simple, it is not so. This is something that has not gotten its due share of importance in prior work. Many a time architects assume that a solution is simple and something at design time can be worked out; however, this is one such instance where this assumption is not true. There are a lot of hidden subtleties and doing this correctly is quite difficult. It is possible that a pre-snapshot message gets stuck for a long time in a router. We must make sure that this message reaches its destination before we start scrubbing data in the lower-level caches. To achieve this, we must perform some form of *termination detection* that makes sure that there are no messages in the NoC with a pre-snapshot tag.

Algorithm 2 describes the operation of routers when a snapshot is going on. R_i describes the ith router. *buffer* represents the internal buffers of a router. Each router can be in one of two states: token received (TR) or no token received (NTR). When the router is in state NTR, and the incoming message is not a token, normal operations continue. Algorithm 2 augments the classical Chandy-Lamport algorithm [20] with a few additional actions (see Section II). To start with, note that the checkpoint token gets the **highest priority** in the routers, which preserves the abstraction of FIFO channels with respect to the token.

The main idea is to keep a count of the all pre-snapshot messages in the system and ensure that all of them reach

Algorithm 2 Flushing Algorithm

```
1: Initially, ∀i, state(R_i) = NTR                          ▷ No token received
2: Router R_i receives message M
3: if state(R_i) = NTR ∧ type(M) = TOK then                 ▷ First token received
4:     state(R_i) = TR
5:     xcount = 0
6:     for M_j ∈ buffer(R_i) do
7:         mark(M_j)
8:         xcount ← xcount + 1                              ▷ Mark all in-flight messages
9:     end for
10:    Send xcount to the checkpoint controller, pcount ← 0
11:    for node ∈ neighbors(R_i) ∪ tile_elements(R_i) do
12:        Send TOK to node                                 ▷ Propagate to all neighbors
13:    end for
14: else if state(R_i) = TR then                            ▷ Already received
15:    if dest(M) = i ∧ type(M) ≠ TOK then                  ▷ Leaving the NoC
16:        pcount ← pcount + 1                  ▷ pcount is the msg processed count
17:    end if
18: end if
19: Periodically:
20:    Send pcount to the checkpoint controller
21: if ∀i, R_i has taken a checkpoint ∧ ∑pcount_i = ∑xcount_i then
22:    Flushing complete
23: end if
```

their destinations before we begin the next phase. We achieve this in a distributed way, having each router count messages internally (expressed in the *xcount* variable), and later sharing this information with the checkpoint controller, which stores the value. Periodically, all the destination routers send the count representing the total number of messages processed (*pcount*) to the checkpoint controller. When $\sum pcount_i$ is the same as $\sum xcount_i$ and all the routers successfully take their checkpoint, we can conclude that all pre-snapshot messages in the NoC have reached their destination.

Theorem IV-B.1: Algorithm 2 will correctly flush all pre-snapshot messages from the system.

Proof IV-B.1: Since the token has the highest priority, a post-snapshot message never reaches a router in state NTR. All messages buffered at the routers will be counted. For messages in transit, a small timer (set to the delay of the link) is set for all the router's links that changed their state from NTR to TR. Pre-snapshot message received within this timer are counted. This leads to all pre-snapshot messages being counted exactly once.

As messages reach their destination, the process count (*pcount*) increases. When the sum of these counts across all routers equals the number of pre-snapshot messages, we can conclude that all messages have reached their destination, leaving no pre-snapshot message in the NoC.

C. Cache Operations

After we take a snapshot at a cache, normal operations continue. The pipeline or upper levels will keep sending requests, albeit marked post-snapshot, and the caches will continue to service them. However, we must reconsider some cache operations that may depend on the snapshot bit.

Read: Reading from a line does not interfere with our snapshot. Since future writes on the same core will not be a part of the checkpoint, locally servicing a read even from a pre-snapshot marked line is correct.

Write: Locally servicing a write to a line may overwrite the data that is marked pre-snapshot. As a result, we must evict the line to the lower level before we effect the write. If the lower level receives this write but has a line marked as pre-snapshot, we overwrite that value because values at higher levels are more recent. In this way, the cache hierarchy coalesces writes internally.

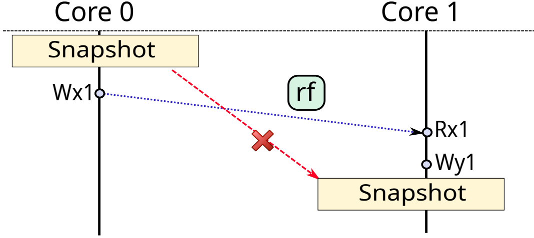

Fig. 4. An rf-edge (read-from) cannot cross a snapshot token

GETS: A GETS (get a copy of the block in the shared state) request does not lead to a write to the line. However, one might argue that a GETS can cause inconsistency in snapshotted data. Figure 4 illustrates the problem. Consider a write to x ($Wx1$) by core 0. We then have a read to x by core 1. Core 1 reads x as 1, thereby creating an rf (*read-from*) edge. Now, at core 1, a write to y executes. We will have an inconsistent snapshot if $Wx1$ is not included in the snapshot while $Wy1$ is included.

The above situation implies that the writer (**Wx1**) is post-snapshot, since it is not included in the snapshot while a causally related write (**Wy1**) is included in the snapshot. However, this situation can never arise. Since the token has the highest priority in our network (discussed above), the reading cache in Core 1 will receive the token from the sister cache (Core 0) before any subsequent write from it. Hence, as shown in Figure 4, in our system, a token cannot cross an **rf**-edge in a hypothetical dependence graph. Consequently, **GETS** messages can be serviced normally.

GETX: A GETX (get a copy in the exclusive state) message will write to the line. As a result, we must writeback this line to the lower level before forwarding it to the requesting cache.

EVICT/INVALIDATE: We send pre-snapshot information to the next level along with the line.

D. Cache Scrubbing

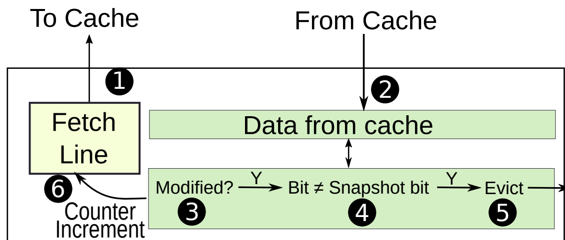

Fig. 5. The implementation of the scrubber

Figure 5 shows the implementation of our scrubber in a cache. The idea is to read entire rows and send pre-snapshot data to lower levels. Scrubbing can be made more efficient by delaying the time between consecutive scrubs. However, the checkpoint's latency depends on our scrubber's frequency since a long-lived pre-snapshot line can be present in the last row of our cache (one that the scrubber touches at the end). We, therefore, present a tunable parameter called *scrubbing-step*. We perform a scrubbing cycle of the cache at each *scrubbing-step*. The lower its value, the more frequently the scrubber activates.

To avoid a scenario where an evicted pre-snapshot line reaches a lower level after the lower level has scrubbed the set for that line, we serialize the scrubbing at different levels. First, we scrub the L2 cache after flushing finishes (in our setup). When this is over, a broadcast from the checkpoint controller tells the next level caches to start scrubbing. Serializing scrubbing in this way does not affect the latency of our checkpoints since the DRAM turns out to be the bottleneck.

V. JASS Detailed Design - DRAM

In this section, we discuss DRAM scrubbing, which is required to persist pages in the DRAM that are a part of the epoch that needs to be checkpointed. This need arises because we do not persist all writes to the NVM by default. Three research questions emerge.

- RQ1: How and when should the DRAM be scrubbed?
- RQ2: How to enforce an upper-bound on the number of unpersisted pages in the DRAM?
- RQ3: How to merge the DRAM and cache data before writing to the NVM?

A. RQ1: DRAM Scrubbing

If we were to scrub the DRAM like the caches, we would need to walk and read the whole DRAM for every epoch, which is not only wasteful but also unnecessary, and it makes maintaining a bound on the *CL* difficult. Any epoch will populate a fraction of the DRAM that is a part of the working set of that epoch. Due to the incremental nature of our snapshot, the additional pages in the working set need to be persisted (referred to as the modified working set).

We must, therefore, track the working set of each epoch. We use a per-epoch DRAM page table that only tracks the modified pages; **note that in the DRAM no page address re-mapping/translation is done**. This page table is similar to page tables used by the OS (albeit conceptually). However, unlike OS page tables, this page table is for the entire system that covers all used physical addresses. Recall that a page in our discussion refers to a 256-byte region (a 4 KB region is a VM-page).

The design of the page table is shown in Figure 6. This table is indexed whenever we issue a write to the DRAM. Since writes are not on the critical path, this does not hurt the performance of the applications. Walking the DRAM and persisting pages belonging to the current epoch is achieved by walking through this page table. The least significant 8 bits are used as an offset. We implement our page table as a four-level tree, indexed using 10 bits at each level. At a non-leaf level

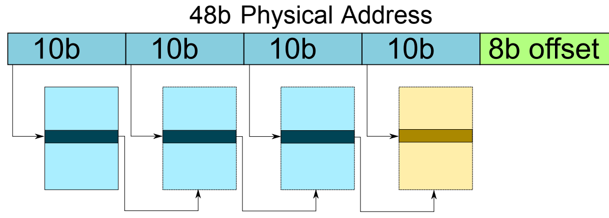

Fig. 6. Per-epoch DRAM page table.

(marked as blue in Figure 6), we store a physical address for the next page and a valid bit. The final level (marked as yellow in Figure 6) holds a *2-bit private counter* for every DRAM page (to be discussed next) as well as a *valid* bit. It is the valid bits at the last level that signify whether a DRAM page belongs to an epoch's modified working set or not.

We use an incremental scheme to walk through this page table. The idea is to balance the scrubbing traffic and regular post-epoch DRAM traffic;note that we do not stall the execution while taking a checkpoint. Since we know the page size at each level, it is more efficient to have a per-level hierarchical counter in the memory controller (to maintain the progress of the page-walk process: one counter for each level of the page table). Alongside this counter, we cache each level's 48b starting page address. To walk the DRAM, we continually increment the lowest level counter to find a page to persist. If the resulting page has its valid bit set in the page table, it is read and persisted. This process continues until the last level counter reaches 2^{10} (the number of entries in each page at a level). When this happens, we calculate the entry's index corresponding to the last level of the page table by incrementing the counter stored in the previous level until we find a valid page. The page found, if not already cached (in a 3KB cache at the memory controller), is read from the DRAM, and the address stored in its 48b field is copied to the last-level counter as the base address for future walks. The process continues up the hierarchy until all valid pages have been walked. In this way, the hierarchical counter implements a nested **for** loop in hardware.

B. RQ2: Limiting the Number of Pages in DRAM

It is necessary to bound the maximum number of pages to be walked in the DRAM to limit the latency of the checkpoints. Failing to do this results in an increase in the minimum time it takes to snapshot. Hence, we must periodically persist pages from the current epoch in the hope that a future snapshot will arrive sooner than a modification to those speculatively-persisted pages. This task calls for using a *locality predictor* (Figure 7) alongside our DRAM scrubber.

We have not seen any prior work that treats a running epoch as a future to-be-persisted epoch. All previous works have assumed that the line between a running epoch and a persisting epoch is the epoch boundary. By persisting parts of the running epoch, we blur the epoch boundaries. However, the recovery epoch must remain consistent. Therefore, these speculated persists are not merged with the recovery page table until we reach the epoch boundary.

The upper bound that our locality predictor must work with is defined by the CL and is calculated by the checkpoint controller. Let l be the CL value (in seconds) and f be the frequency of the system. The number of cycles in which the snapshot must complete is $c = lf$. If it takes k cycles to persist a page, then the upper bound for an epoch is simply

$$n \leq \frac{c}{k} \tag{1}$$

However, this does not take into account the non-determinism on a real system and makes simplistic assumptions. Let us thus use this as a baseline result and design a more sophisticated algorithm.

C. Dynamic Tuning of the Locality Predictor

It is important for the locality predictor to know when to persist a page. From Equation 1, we know the value of n that roughly corresponds to the target CL. If we were to keep no pages in DRAM that need to be persisted (similar to NVOverlay), we will achieve a checkpoint latency equivalent to the scrubbing time of caches. Let us call this the minimum latency CL_{min}. The acceptable observed latency (CL_{obs}) must therefore lie in the interval (CL_{min}, CL).

The checkpoint controller, working with the memory controller tunes the locality predictor for minimizing WA. The idea is to achieve a CL_{obs} close to CL. This means that the locality predictor should not be over-aggressive. An over-aggressive locality predictor will increase the checkpoint latency because more pages that need to be persisted will be in memory; if it is less aggressive, then the WA will increase.

To tune it, an epoch is divided into 50 sub-epochs. At the start of each sub-epoch, the memory controller calculates how close the epoch's modified set is to the theoretical maximum number of modified pages (n). Tracking any epoch's modified set requires a counter at the memory controller, which is incremented after a new page is added into the page table. Let the difference divided by n be δ. In this paper, we tune the aggressiveness of the predictor by changing its activation rate $\mathscr{R}$ (how frequently it runs). It can vary from 512-256K cycles. It varies linearly with δ.

D. Locality Prediction

To make the prediction, we allocate a private 2-bit counter for each DRAM page and a shared 3-bit saturating counter for a group of 64 contiguous pages. The private counters are stored alongside the last level page table entry for an epoch, whereas the shared counters are stored separately. It can never be the case that a page is a part of both epochs' modified working set because we do not do any further address re-mapping/translation: we thus need to persist the earlier avatar of the page first. Therefore, we can safely store the private counter alongside the DRAM page-table entry. On the other hand, for shared counters, it is expected that a small subset of these counters will be used (due to locality) during a given

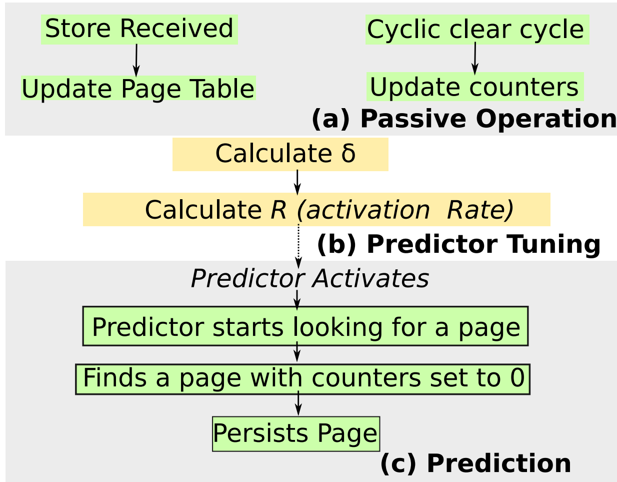

Fig. 7. Operations of JASS that occur irrespective of whether a checkpoint is in progress or not. (a) shows passive operation. (b) and (c) shows predictor functions.

time. As a result, separately storing these is a better idea. If these counters were present in the page table, a shared counter jump would mean a jump that is 64 pages away.

Every time we walk the DRAM page table, looking for a page to persist, we do a cyclic clear of bits in private and shared counters. This cyclic clear will clear bit 0 of the private counter in one operation and bit 1 in the next operation. The main idea is to capture both temporal and spatial locality. The shared counter captures spatial locality since nearby addresses share this counter. The cyclic clear captures temporal locality since it will eventually clear all bits of shared and private counters to zero for pages that have not been accessed in a long time. We can *persist* pages speculatively if we see that they are a part of the current epoch with their shared and private counters set to zero.

E. RQ3: Mispredictions and Data Coalescing

To reduce write amplification, we *coalesce* data from the upper levels with data in the DRAM using an access scheduler (similar to those used in modern GPUs [22]). To achieve this, the first time a request from the cache for a block reaches the access scheduler, a request is sent to the DRAM scrubber asking it to scrub the page that belongs to the cache line. The scrubber obliges if this page is present in the pre-snapshot epoch table. If the page is absent, the scrubber sends a *nak* (negative acknowledgement) to the access scheduler.

The locality predictor may predict to persist a page before a request from the access scheduler arrives. This is the case of misprediction since a line from the cache has reached the access scheduler, whereas the predictor did not expect this. As a result, the misprediction penalty is that we send a *nak* to the access scheduler, which must then persist the page again, creating wasted writes. Correctness is never violated since the recovery table in the NVM is never updated speculatively. We don't describe the maintenance of this table and checkpoint recovery in detail because it is the same as that used by other competing work [6].

F. Tunable Parameters

Our system tunes the following values. The system determines these values using the $\langle ES, CL \rangle$ pair provided to the system. However, these parameters are not directly accessible to the user or the OS. This tuple is sent along with the token by the checkpoint controller.

1) *scrubbing-step* : Frequency of the cache scrubber.
2) *scrubbing-granularity*: Number of rows (or sets) to scrub in one scrubbing-cycle.
3) *memory-walk step*: Frequency of the DRAM walker.

VI. Experimental Results

In this section, we implement and evaluate JASS on Tejas [23], a cycle-approximate Intel PIN-based [24] architectural simulator. We ran benchmarks from the PARSEC [25] benchmark suite, with each program executing for a minimum of one billion instructions.

A. Setup

Parameter	Value
Number of cores	8
Pipeline type	Out-of-order
Frequency	3200MHz
Private Caches	L1 L2
Coherence	L2
L1 Cache	32K 4-cycle WT 8-way 64B block
L2 Cache	256K 8-cycle WB 8-way 64B block
L3 Cache	8-way Tiled SNUCA LLC
L3 Cache Bank	2M 30-cycle WB 8-way 64B block
NoC Topology	TORUS
DRAM ranks	2 ranks, 8 banks per rank
NVM ranks	2 ranks, 8 banks per rank
DRAM page size	2Kb
NVM page size	2Kb

TABLE I
System setup. Our system setup is in line with recent work in the area of persistent memory architectures [26], [27]

Table I show the system setup used for evaluating JASS along with the nearest competing works in the area: ThyNVM[8], PiCL[7] and NVOverlay[6]. The design choices used for ThyNVM, PiCL and NVOverlay were chosen to minimize *WA* (for a fair comparison). L2 is coherent in all cases. The shared LLC is the L3 level, which is distributed across the chip. We use the popular SNUCA [28] scheme. The system uses a torus NoC with eight cores and eight LLC tiles. The NoC also has nodes for the directory and memory controllers. All implementations include an on-chip persistent buffers that coalesce data before writing to the NVM.

B. Performance (∝ 1/(Simulated Execution Time))

In this section we compare the performance of ThyNVM, PiCL, NVOverlay and JASS. Figure 8 shows the normalized performance (reciprocal of execution time) relative to JASS. Contrary to PiCL and ThyNVM, NVOverlay and JASS do not affect the critical path (loads do not get stalled). While PiCL

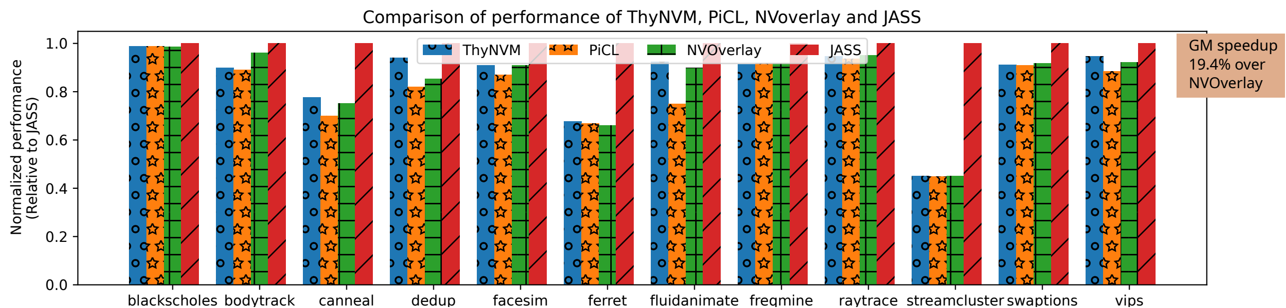

Fig. 8. Normalized performance ThyNVM, PiCL and NVOverlay with respect to JASS.

checks undo logs for all reads, ThyNVM's address remapping scheme has performance implications. While NVOverlay adds no latency to its read operations, the system persists data upon eviction from the coherent domain, increasing NoC traffic. Because of our non-intrusive mechanisms, we get a speedup of 19.4$ over NVOverlay.

C. Write Amplification

In this section, we compare the write amplification of JASS with ThyNVM, PiCL and NVOverlay. Figure 9 shows an improvement of at least 2× on most benchmarks. This improvement is largely attributed to the fact that data is not persisted on to the NVM upon cache eviction. Moreover, we see a fair amount of coalescing taking place in the entire hierarchy (compared to only L1 and L2 for NVOverlay).

D. Insights into the Operation of the System

Table II shows salient features of the execution: *CL*, % of pre-snapshot data in the caches, DRAM, degree of coalescing, misprediction rate of the locality predictor, and the LLC MPKI (misses per kilo instructions). In this $CL = 5$ ms. It is evident from column 2 in Table II that we achieve our *CL* target (within ± 5%). Since DRAM and caches begin scrubbing independently, the slowest of the two will dictate the total time. Inevitably, DRAM scrubbing is the limiting factor. The caches are more or less fully full with pre-snapshot data. *Canneal* and *Fluidanimate* are notable exceptions given their access patterns. The DRAM (%) numbers are relative: the numerator is the pre-snapshot footprint in the DRAM and the denominator is the total size of the snapshot. The degree of coalescing is small (0.4 to 23%); most values are less than 10%.

The LLC MPKI gives us an insight into the locality of the application. Benchmarks with small (*blksch*) and medium (*swap,vips*) working sets have a smaller LLC MPKI due to increased locality. This distinction will help us understand the write amplification trends (in the next section).

% DRAM-*WA* represents the additional writes that we need to perform owing to imperfect locality prediction with reference to a scenario where we have a perfect predictor. We see an additional 8.7-50.1% writes. This leads to write amplification.

We also tune our parameters dynamically. In our setup, we set the initial values of *scrubbing-step* and *memory-walk step* to 1K cycles each and *scrubbing-granularity* is set to 1 (one cache row at a time). We allow the *memory-walk step* to be tuned within the range $(0.5K, 256K)$ cycles. These values represent a vast tuning range for our experiments. Since cache scrubbing is faster than DRAM scrubbing, we have more opportunities to steal a cache's cycles or even just a read port to read entire rows very quickly. This causes the caches to rarely tune their *scrubbing-step* values.

E. Sensitivity to Input Parameters

In this section we run two experiments to compare JASS to NVOverlay. In the first experiment, we run JASS with five different epoch sizes. In the second experiment, we run JASS on five different latency constraints.

1) Epoch Size (ES): We evaluate JASS with five different epoch sizes (2.5 ms, 3 ms, 3.5 ms, 4 ms and 4.5 ms) and study the effects of write amplification with increasing *ES*. *CL* is set to 2 ms. The results in Figure 10 show that *WA* `generally` reduces with increasing epoch sizes. The slope of the line is negative with increasing *ES*, highlighting the saturation effect as expected.

The point of saturation is different for every benchmark. For the benchmarks *bodytrack, blacscholes, raytrace*: moving from 2.5 ms to 3 ms increases the *WA*; For the benchmark *canneal*, there is a significant increase from 4 to 4.5 ms. These outliers are happening because of the non-deterministic nature of coalescing.

2) Checkpoint Latency (CL): We compare JASS with five different *CL* values: 1, 2, 3, 4, and 5 ms. We show that we always meet the *CL* constraint (refer to Figure 11). We see that given a latency, JASS meets it. CL_{obs} (observed *CL*) is always within 5% of the target *CL*. The corresponding write amplification is shown in Figure 12. This figure illustrates the tradeoff between write amplification and the checkpoint latency. As with epoch size, our latency graph has a negative slope, i.e., with increasing latency, write amplification decreases. JASS is unique in this space. Our system, given a *CL*, will `minimize` the write amplification as well as `meet` the checkpoint latency.

Although the general trend of Figure 12 is a downward slope, there exist some outliers. Outliers in Figure 12 are a result of a similar, albeit subtle, impact of checkpoint latency on time between checkpoints as seen with increasing *ES* . *bodytrack* is the most notable outlier here, as its *WA* increases at 3,4 and 5 ms as compared to 2 ms. Since slower checkpoints (large

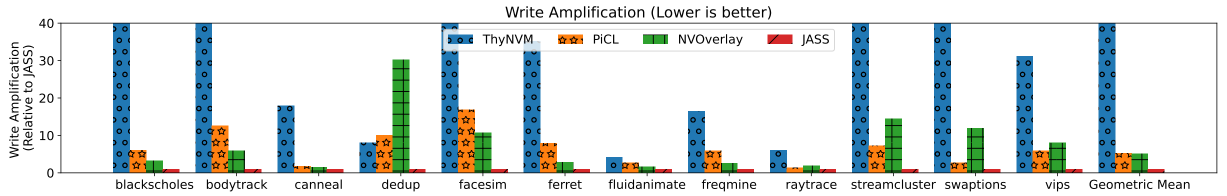

Fig. 9. Normalized write amplification of ThyNVM, PiCL and NVOverlay with respect to JASS.

Workload	CL (ms)	% Cache	% DRAM	% Coalescing	LLC MPKI	% DRAM-*WA*
Blk.	5.25	88	12	13	0.20	50.16
Body.	5.0	78	22	2	0.14	25.6
Cann.	5.0	40	60	23	0.48	49.1
Ded.	5.0	81	19	4	17.6	23.1
Face.	5.0	84	16	0.4	0.91	8.7
Ferr.	5.05	78	22	9.5	0.33	50.7
Fluid.	5.0	68	32	1.6	0.51	10.7
Freq.	5.0	86	14	4	0.89	20.6
Ray.	5.0	84	16	1.5	0.78	15.2
Strm.	5.1	84	16	0.9	0.55	18.7
Swap.	5.23	86	14	5.5	0.09	25.3
Vips	5.01	84	16	2.6	0.17	49.9

TABLE II

DIFFERENT EXECUTION STATISTICS THAT PROVIDE AN INSIGHT INTO THE WORKING OF THE SYSTEM. TIME VALUES FOR THE LONGEST EPOCH ARE REPORTED FOR A LATENCY CONSTRAINT OF 5 MS.

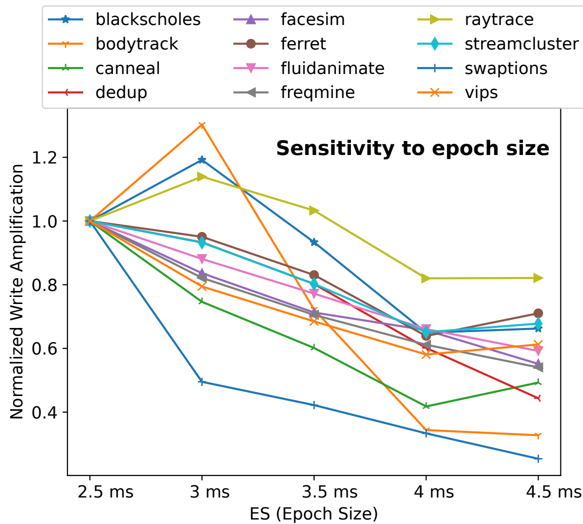

Fig. 10. Normalized write amplification of JASS with increasing epoch size.

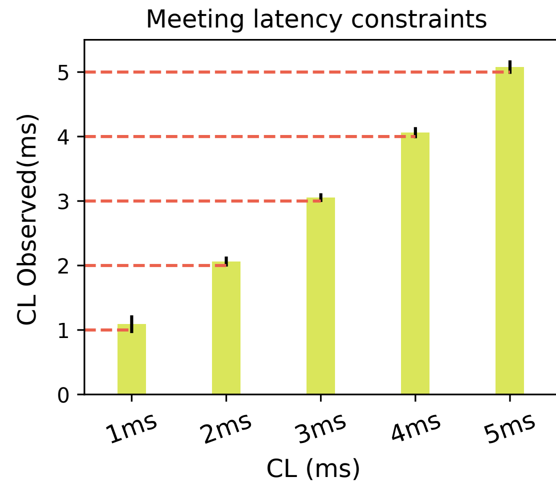

Fig. 11. Maximum checkpoint completion time for JASS under different checkpoint latency constraints.

CL) will push the completion of a checkpoint to a future point, JASS remains quiescent after the completion of the checkpoint. This causes the time between checkpoints to increase, which interferes with the access patterns of the outliers.

VII. Related Work

In this section, we look at system-level checkpointing proposals: ThyNVM [8], PiCL [7] and NVoverlay [6].

ThyNVM [8] divides each program's execution into time-ordered epochs. Each epoch has two phases: an execution phase and a checkpointing phase. To avoid application stalls, the checkpointing phase of an epoch runs concurrently with the execution phase of the next epoch. ThyNVM incurs a large write amplification since it maintains three versions of data at any given point of time (current, last, second-last). All three copies are required since a failure can corrupt both the working copy and the checkpoint in progress.

PiCL [7] aims to improve on ThyNVM by reducing the epoch sizes further. It uses undo logging in the caches. Although not a problem at the level of blocks (since logs are persisted in batches), the system produces a log for every write operation on the cache. Such a scheme is prohibitive since the baseline write amplification is high because of the writes to the log.

NVOverlay [6] defines epoch boundaries by monitoring coherence traffic, thus creating prohibitively small epochs. The system uses a distributed vector clock with no global

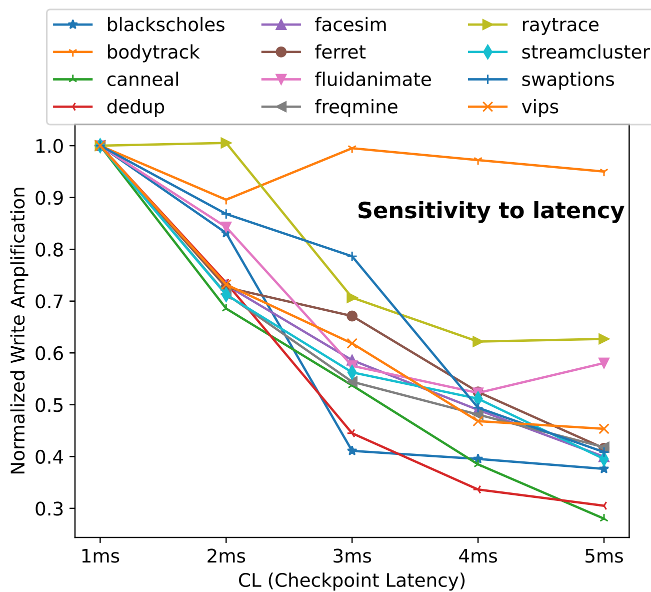

Fig. 12. Tradeoff between latency and write amplification.

synchronization. This causes some cores to run ahead of others (in terms of the epoch number). Designed for high-frequency checkpointing, NVOverlay tags each cache line with a 16-bit epoch id. Moreover, the system tags each DRAM row using ECC bits, which is not practical. A multi-version snapshotting mechanism manages different versions of page tables for each epoch in DRAM, while a single *recovery epoch* page table is present in the NVM. To reduce writes to NVM, NVOverlay unmaps data in the NVM at cache line granularity while mapping is done at page granularity. This causes the system to occasionally perform garbage collection of sparsely mapped pages.

VIII. Concluding Remarks

There are two key takeaway points from the design of JASS: *tunability* and *efficiency*. Any target *CL* has a *WA* cost, and thus minimizing the number of additional writes is of paramount importance. There is a need to design a very efficient control algorithm. Moreover, the downtime of a checkpointing algorithm can be made near-zero and we can efficiently flush the in-flight messages and scrub the DRAM to ensure that the work that needs to be done during checkpointing always remains within bounds. JASS requires very little HW modifications and has dual benefits: *WA* reduction by 35-96% and a speedup of 19.4% as compared to the nearest state-of-the-art competitor NVOverlay.

References

[1] R. S. Venkatesh, T. Smejkal, D. S. Milojicic, and A. Gavrilovska, “Fast in-memory criu for docker containers,” in *ISMS*, 2019.

[2] N. Kohl, J. Hötzer, F. Schornbaum, M. Bauer, C. Godenschwager, H. Köstler, B. Nestler, and U. Rüde, “A scalable and extensible checkpointing scheme for massively parallel simulations,” *The International Journal of High Performance Computing Applications*, vol. 33, no. 4, pp. 571–589, 2019.

[3] R. Rocco, D. Gadioli, and G. Palermo, “Legio: fault resiliency for embarrassingly parallel mpi applications,” *The Journal of Supercomputing*, vol. 78, no. 2, pp. 2175–2195, 2022.

[4] G. Georgakoudis, L. Guo, and I. Laguna, “Reinit: Evaluating the performance of global-restart recovery methods for mpi fault tolerance,” in *ICHPC*, 2020.

[5] P. Ekemark, Y. Yao, A. Ros, K. Sagonas, and S. Kaxiras, “Tsoper: Efficient coherence-based strict persistency,” in *HPCA*, 2021.

[6] Z. Wang, C.-H. Choo, M. A. Kozuch, T. C. Mowry, G. Pekhimenko, V. Seshadri, and D. Skarlatos, “Nvoverlay: Enabling efficient and scalable high-frequency snapshotting to nvm,” in *ISCA*, 2021.

[7] T. M. Nguyen and D. Wentzlaff, “Picl: A software-transparent, persistent cache log for nonvolatile main memory,” in *MICRO*, 2018.

[8] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu, “Thynvm: Enabling software-transparent crash consistency in persistent memory systems,” in *MICRO*, 2015.

[9] K. Kruger, R. Pannain, and R. Azevedo, “Donuts: An efficient method for checkpointing in non-volatile memories,” *Concurrency and Computation: Practice and Experience*, 2023.

[10] G. Zheng, X. Ni, and L. V. Kalé, “A scalable double in-memory checkpoint and restart scheme towards exascale,” in *DSN*, 2012.

[11] A. Miraglia, D. Vogt, H. Bos, A. Tanenbaum, and C. Giuffrida, “Peeking into the past: Efficient checkpoint-assisted time-traveling debugging,” in *ISSRE*, 2016.

[12] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, “Mojim: A reliable and highly-available non-volatile memory system,” in *ASPLOS*, 2015.

[13] S. Wang, J. Liagouris, R. Nishihara, P. Moritz, U. Misra, A. Tumanov, and I. Stoica, “Lineage stash: fault tolerance off the critical path,” in *SOSP*, 2019.

[14] A. Khorguani, T. Ropars, and N. De Palma, “Respct: fast checkpointing in non-volatile memory for multi-threaded applications,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 525–540.

[15] E. Meneses, X. Ni, G. Zheng, C. L. Mendes, and L. V. Kale, “Using migratable objects to enhance fault tolerance schemes in supercomputers,” *IEEE transactions on parallel and distributed systems*, vol. 26, no. 7, pp. 2061–2074, 2014.

[16] D. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel, “Resilience-aware resource management for exascale computing systems,” *IEEE Transactions on Sustainable Computing*, vol. 3, no. 4, pp. 332–345, 2018.

[17] J. W. Young, “A first order approximation to the optimum checkpoint interval,” *Communications of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[18] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao, “Emerging nvm: A survey on architectural integration and research challenges,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 23, no. 2, nov 2017.

[19] A. Baldassin, J. a. Barreto, D. Castro, and P. Romano, “Persistent memory: A survey of programming support and implementations,” *ACM Comput. Surv.*, vol. 54, no. 7, jul 2021.

[20] N. A. Lynch, *Distributed algorithms*. Elsevier, 1996.

[21] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. S. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, “Basic performance measurements of the intel optane DC persistent memory module,” *CoRR*, vol. abs/1903.05714, 2019.

[22] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, “Memory access scheduling,” *SIGARCH Comput. Archit. News*, vol. 28, no. 2, p. 128–138, may 2000.

[23] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter, “Tejas: A java based versatile micro-architectural simulator,” in *PATMOS*, 2015.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” *SIGPLAN Not.*, vol. 40, no. 6, jun 2005.

[25] C. Bienia, “Benchmarking modern multiprocessors,” Ph.D. dissertation, Princeton University, January 2011.

[26] S. Yadalam, N. Shah, X. Yu, and M. Swift, “Asap: A speculative approach to persistence,” in *HPCA*, 2022.

[27] M. Vemmou and A. Daglis, “Cosplay: Leveraging task-level parallelism for high-throughput synchronous persistence,” in *MICRO*, 2021.

[28] S. R. Sarangi, *Advanced Computer Architecture*, 1st ed. India: McGraw Hill, 2021.