PiCL: a Software-Transparent, Persistent Cache Log for Nonvolatile Main Memory

Tri M. Nguyen
Department of Electrical Engineering
Princeton University
Princeton, USA
trin@princeton.edu

David Wentzlaff
Department of Electrical Engineering
Princeton University
Princeton, USA
wentzlaf@princeton.edu

***Abstract*—Software-transparent crash consistency is a promising direction to immediately reap the benefits of nonvolatile main memory (NVMM) without encumbering programmers with error-prone transactional semantics. Unfortunately, proposed hardware write-ahead logging (WAL) schemes have high performance overhead, particularly for multi-core systems with many threads and big on-chip caches and NVMs with low random-access performance. This paper proposes PiCL, a new WAL checkpointing mechanism that provides a low overhead, software-transparent crash consistency solution for NVMM. PiCL introduces multi-undo logging, cache-driven logging, and asynchronous cache-scan to reduce random accesses and enable good row locality at the NVM. The key idea is that: by relaxing the durability timing of checkpoints, crash consistency can be provided with less than 1% performance overhead where 1.5× to 5.0× slowdown was typical with prior work. To demonstrate the feasibility of software-transparent crash consistency, we fully implemented PiCL as an FPGA prototype in Verilog using the OpenPiton framework.**

***Index Terms*—cache memory, nonvolatile memory, parallel processing, computer crashes, checkpointing**

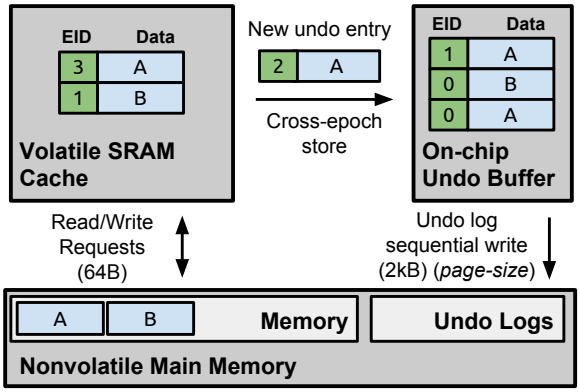


Fig. 1. A structural view of PiCL architecture. Memory writes are divided into epochs, and cache lines are tagged with EpochIDs (EID). Overwriting cache lines with older EIDs creates undo entries, which are coalesced in an undo buffer before they are written sequentially in blocks of 2KB to NVM.

I. Introduction

Nonvolatile main memory (NVMM) [1] is emerging as an important research direction, as both the means to build NVMM and the need to use NVMM are rapidly converging. Byte-addressable nonvolatile memory (NVM) products such as Intel 3D XPoint have been commercialized [2, 3], and plug-in DDR4 replacement standards like NVDIMM-F/P/N have been proposed [4] with Intel announcing DDR4 pin-compatible NVM DIMM devices for up to 6TB of addressable memory [5]. Besides lower refresh power and higher storage density than conventional DRAM, the data persistence of NVM promises instantaneous crash recovery and high availability for extreme-scale distributed systems and warehouse-scale shared-memory systems where failures are the common case.

Unfortunately, crash consistency does not come cheap. While NVM itself does not lose data on power loss, the SRAM-based cache hierarchy *does*. It reorders and coalesces memory operations such that stores arrive at memory out-of-order, leading to memory inconsistency on power failure. For instance, when a doubly linked list is appended, two memory locations are updated with new pointers. If these pointers reside in different cache lines and are not both propagated to memory when the system crashes, the memory state can be irreversibly corrupted.

To solve this problem, much of the prior work provides crash consistency through software transactional interfaces, persistent objects, or multi-versioning management [6]–[20]. Sadly, by pushing the burden of correctness to the programmers, these schemes hurt programmability, are error prone [21], and the software-directed cache flushes often lead to 5-10× performance slowdown [14].

As such, the *software-transparent* approach to crash consistency [22]–[26] is quite attractive as an alternative. Additionally, through compatibility with legacy software, the benefits of NVMM can be reaped immediately, leading to an accelerated adoption rate of persistent memory technologies. These schemes make a checkpoint of the memory state once per some epoch time interval such that on a crash event, the memory state can be readily reverted to a consistent state. To avoid copying the entire memory address space at every epoch, write-ahead logging (WAL), either redo or undo logging, is often used in conjunction with shorter checkpoints (10ms to 100ms).

Regrettably, there are two critical scalability problems in prior work. First, cache flushes are mandatory at each epoch. In CPUs with large (64MB+) on-chip caches, the cache flush latency dominates both the commit and execution time leading to high overhead. Furthermore, these cache flushes are often *synchronous*, which means they "stop-the-world" and stall the

whole system until completely finished. Second, both redo and undo logging add significantly more random accesses to the NVM storage. Because NVMs have random access performance more than 10× worse than conventional DRAM [27]–[29], these logging patterns are costly. For both problems, adding a DRAM cache layer to absorb the writes does not help because these writes necessarily have to be persisted in the NVM layer to guarantee checkpoint *durability*.

In this paper, we propose PiCL (pronounced *pickle*) to provide software-transparent crash consistency at low cost. PiCL implements a unique solution of *multi-undo logging*, *cache driven logging*, and *asynchronous cache scan* to (1) take the cache flushes off the critical path and (2) coalesce undo writes to better match the page-based performance characteristics of NVMs. At the surface, these ideas are similar to well-known concepts in databases and file-systems such as *group commit* [11], but they are largely unexplored in the context of software-transparent crash consistency.

Some of the key mechanisms are demonstrated in Fig. 1. First, by tagging cache lines with Epoch IDs (EID), the volatile cache hierarchy is able to track data of different epochs simultaneously. Second, when cross-epoch stores are detected, the pre-store data is saved as an undo entry and is collected in an on-chip undo buffer. This on-chip buffer then periodically flushes many entries together to maximize the sequential write bandwidth of NVMs. As a result, PiCL minimizes random accesses and provides crash consistency with less than 1% performance overhead where a performance loss of 1.5× to 5.0× was typical with prior work for an eight-core CPU with 16MB last-level cache.

Besides low performance overhead, the architecture of PiCL is simple. Cache eviction policy is unmodified. PiCL also does not need translation tables [26], nor NVM caches [9, 30], nor sophisticated persistent memory controllers [14]–PiCL is designed to work with off-the-shelf DDR-like NVDIMMs [5]. To further demonstrate the feasibility of PiCL, we implemented it as an FPGA prototype in the OpenPiton open-source manycore framework [31] and the necessary interrupt handlers as a complete prototype. We summarize the implementation and report its overheads when synthesized to a Xilinx Genesys2 FPGA.

Specific contributions of this paper include:

- We introduce *multi-undo logging*, an improvement to undo logging that allows concurrent logging of multiple epochs.
- We introduce *cache-driven logging*, a low-overhead technique that tracks cache modifications across epochs and writes back undo data directly from the processor. Cache-driven logging breaks the costly *read-log-commit* access sequence of undo-based logging schemes.
- We combine multi-undo logging and cache-driven logging to purposefully overcome cache flushes and minimize row buffer misses at the NVM. The resulting design is simple, requiring no change to existing DDR NVDIMM interfaces.
- We evaluate PiCL across SPEC2006 applications, and find that cache flushes and non-sequential logging in prior work significantly introduce overheads, especially in multi-core, multi-programmed workloads.
- We implement PiCL in OpenPiton to demonstrate the feasibility of software-transparent crash consistency.

TABLE I
GLOSSARY OF DIFFERENT EPOCH STATES.

Epoch	A time interval in which all writes within share the same epoch ID (EID).
Executing epoch	An uncommitted epoch. The EID of this epoch is the same as the `SystemEID`.
Committed epoch	An epoch that has finished, but not necessarily *persisted* to NVM.
Persisted epoch	An epoch with its data fully written to NVM. The system can be reverted to the memory state of a persisted epoch. Abbreviated as `PersistedEID`.

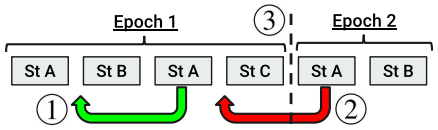


Fig. 2. Write ordering with epochs. ① Writes within an epoch can be reordered and coalesced, but ② cannot across epoch boundaries ③. Epoch boundaries are also where all modified data must be flushed to persist the current epoch.

II. BACKGROUND

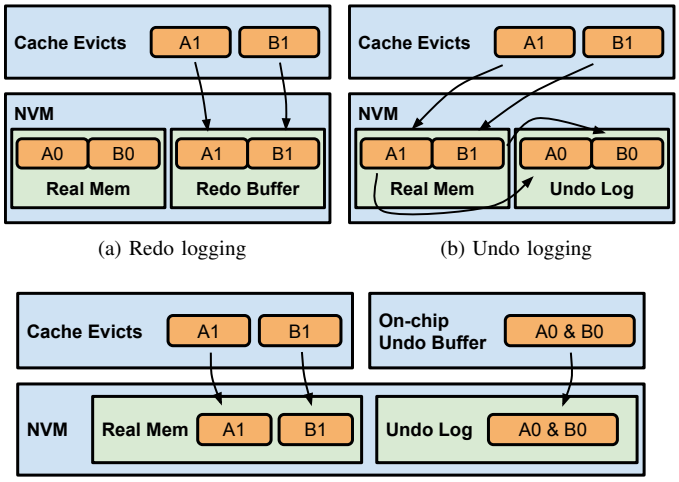
In this section, we describe three key concepts to understand the motivation of this work: epoch-based checkpointing, weaknesses of prior work in write-ahead logging, and the difference in performance between sequential and random access of byte-addressable NVMs.

A. Epoch-based Checkpointing

Over the lifetime of a program, a single cache line can be modified multiple times, and depending on the logging strategy being used, the bandwidth requirement can be drastically different. Instead of strictly enforcing the write order for *all* writes, epoch-based checkpointing (shown in Figure 2) allows full reordering to occur within a given time interval–or *epoch*. The epoch length or checkpoint interval varies depending on usage, but for relevant prior work that uses checkpointing, it is typically between 10ms to 100ms [22]–[25, 32]. Smaller checkpoints are necessary to achieve high availability[1], though PiCL is generally agnostic to checkpoint lengths and has reliable performance when using checkpoints of up to 1000ms.

One requirement of epoch-based checkpointing is that for correctness all modified cache lines must be synchronously flushed and persisted in NVM at the end of said epoch (such as at ③ for `Epoch1` in Figure 2). If modified data are not completely flushed or if the system is allowed to run while flushing, memory consistency is not guaranteed on a power loss. This cache flush is mandatory when the *commit* and *persist* operations happen atomically, which is often the case for prior WAL proposals.

[1]To achieve 99.999%, system must recover within 864ms [24]



(a) Redo logging (b) Undo logging

(c) PiCL

Fig. 3. Cache eviction handling in redo and undo logging. (a) Redo logging always appends new data into a redo buffer, while (b) undo logging writes in-place, but needs to make and append old data into an undo log first. (c) PiCL is based on undo logging, but sources (multiple) undo data directly from the on-chip undo buffer.

Problem: cache flushes are not scalable to systems with large on-chip caches. Most of the overhead of committing a checkpoint comes from writing dirty cache data back to memory: on average, it takes 1ms to write back a 2MB cache to DRAM [24]. While 1ms (or 10% of a 10ms checkpoint) is not unacceptable for some use cases, on-chip SRAM caches are rapidly growing in size. Intel and AMD server-class CPUs can have up to 64MB of L3 [33, 34] and 128MB of L4 [33], while the IBM mainframe z14 [35] has up to 128MB of L3 and 672MB of L4 cache. While the memory controller might be able to reorder these transactions to exploit row buffer locality, if cache contents do not have good data spatial locality in the first place, the memory controller would have to constantly open and close large memory pages just to write 64 bytes of data.

As a side note, while most literature in databases uses the term "commit" to indicate finality of a transaction, it has a slightly different semantic for NVM crash consistency. To be precise, when a checkpoint is committed, it is guaranteed atomicity, consistency, and isolation–but not durability, or the permanence of the changes. Only when the checkpoint is *persisted* in NVM that it is made durable. Table I gives an overview of the differences between the executing, committed, and persisted states.

B. Undo and Redo Write-ahead Logging

In database systems and file systems, write-ahead logging (WAL) is a family of techniques for providing atomicity and durability. Applied to checkpointing, atomicity requires that epochs are restored as a whole unit, and durability ensures that the data of the *persisted* epochs does not get corrupted by newer *uncommitted* epochs. Writes are propagated to NVM mainly in the form of *cache evictions* and cache flushes at the end of an epoch. There are two basic approaches in WAL: redo logging and undo logging.

In redo logging [36] (shown in Figure 3a), cache evictions are temporarily held in a redo buffer located in NVM to preserve main memory consistency until the next commit phase. Similar to a CPU *write buffer*, this redo buffer is snooped on every memory accesses to avoid returning outdated data.

Problem: redo buffers are not scalable to large multi-core systems. Like the CPU write buffer, the redo buffer is typically a fixed-size, associative structure and thus is not suitable for workloads with large write sets nor for multi-core systems with many concurrent threads. When there are more writes, the buffer overflows more often. On each buffer overflow, the system is forced to abort the current epoch prematurely, leading to shorter epochs and more disrupting cache flushes.

In undo logging [36, 37] (Figure 3b), on a cache eviction, the undo data is first read from its canonical memory address. Then, this data is persisted into an undo buffer in NVM. Finally, the eviction is written in-place in memory. We refer this sequence as the *read-log-modify* access sequence. If the system crashes or loses power, it reverts these writes by applying the entries in the undo buffer to restore consistency of the last checkpoint.

Problem: undo logging has poor data spatial locality. For every cache eviction, the *read-log-modify* access sequence is performed to ensure correctness. At worst, every write is now one read and two writes. At best, multiple undo entries can be coalesced and written to the NVM log as a group. Even with this optimization however, we still have to pay the random-access cost of first reading multiple undo data then writing updated data which need to be done separately.

Prior work in SW-transparent WAL: The closest prior work to PiCL is ThyNVM [26]. ThyNVM is a redo-based WAL design where memory translation tables are used to maintain both the committed and the volatile execution versions of the data. It has a mixed checkpoint granularity of both block-size and page-size, which can lead to good NVM row buffer usage for workloads with high spatial locality. ThyNVM also overlaps the checkpoint phase with the execution phase to minimize stalling, although it is still subjected to a synchronous cache flush stall at every checkpoint, and requires a translation table which is not scalable to large multi-core systems.

PiCL is also closely related to previous hardware-driven high-frequency checkpointing designs including FRM [22, 23] and other high-frequency checkpoint designs [24, 25] which rely on undo logging for crash recovery.

It is worth noting that decreasing the checkpoint frequency is unlikely to eliminate all performance overhead. For instance, undo logging still requires a *read-log-modify* operation for each cache eviction. For redo logging, the translation table has a hard limit on the write set and the duration of a checkpoint.

A related but different class of work is *epoch-based persistency*, where writes are separated into *persist barriers* by the programmers. We give further discussions in §VII, but key differences are that (1) persist barriers are not SW-transparent, and (2) *persist barriers* are typically much shorter than a checkpoint.

TABLE II
COMPARISON OF PICL AND PRIOR WORK IN SOFTWARE-TRANSPARENT WRITE-AHEAD LOGGING

	FRM	Journaling	ThyNVM	PiCL
Async. cache flush	✗	✗	✗	✓
Single-commit overlap	✗	✗	✓	✓
Multi-commit overlap	✗	✗	✗	✓
Undo coalescing	✗	N/A	N/A	✓
Redo page coalescing	N/A	✗	✓	N/A
Second-scale epochs	✓	✗	✗	✓
No translation layer	✓	✗	✗	✓
Mem. ctrl. complexity	Medium	Medium	High	Low

C. Performance of Byte-Addressable NVM

While byte-addressable NVMs are much faster than NAND flash SSDs, they still have significantly lower random access performance than DRAM. This is especially true for capacity-optimized storage-class memory (SCM) NVMs [38, 39] where row buffers are large, and a row miss latency is more than 300ns [27]–[29] (or more than 10× that of DRAM). In fact, early commercial products (namely the Intel 3D XPoint) are characterized to have 32k random I/O operations per second (IOPS) for write per chip [40], which when scaled to 10 chips (a typical amount for a memory DIMM), write throughput would still be only 320k IOPS, or around 3000ns per row buffer write miss.

It is therefore imperative that memory requests to NVMs maximize spatial locality. Experimental data [41] shows that peak throughput is only reached when writing 4KB data, which suggests a row buffer size of at least 2KB [27, 28] in current products. Row buffer size is likely to grow even larger with multi-level cell [42] (MLC) NVM as inferred from studying the source code of NVMAIN [43].

III. MOTIVATION & DESIGN

All in all, PiCL's problem statement can be summarized as follows: prior work in software-transparent WAL is not scalable to multi-core systems. Neither undo logging nor redo logging has good data spatial locality, and both approaches require synchronous cache flushes which is not scalable with growing cache sizes. To solve these problems, we propose three novelties in PiCL:

1) A true decoupling of the execution and checkpointing phases through *multi-undo* logging. Multi-undo logging allows multiple logical commits to be in-flight while still maintaining a single central undo log, removing the need for synchronous cache flushes.
2) By versioning individual cache lines, *asynchronous cache scan* further removes cache flushes from the critical path and asynchronously executes checkpoint phases to minimize performance overhead.
3) Buffered undo logging through *cache-driven logging*. By preemptively sourcing undo data directly from the on-chip cache, it is now possible to buffer these entries before writing them to NVM.

At the surface, these ideas are similar to well-known concepts in databases and file-systems such as *group commit* [11], but they are largely unexplored in the context of software-transparent crash consistency. For instance, to the best of our knowledge, none of the prior work flushes the cache asynchronously. While recent redo-based schemes like ThyNVM [26] enables asynchronous execution and checkpointing, the overlapping degree is limited to one checkpoint, and cache flushes are still synchronous. *Group commit* is often employed as a means to amortize the cost of persistency, but it is unclear how it can be applied to cache evictions. PiCL, for the first time, applies cache-driven logging to coalesce and efficiently write multiple undo entries. As a summary, Table II gives an overview of the features that PiCL is contributing to the state-of-the-art.

Ultimately, PiCL is making the trade-off between performance and durability (i.e., the permanence of checkpoints). By allowing multiple checkpoints to be in-flight, it delays the persist operations which can adversely affect I/O consistency. We further discuss this topic in §IV-C.

In the following sections, we start with undo logging as a baseline design for PiCL, then incrementally add cache-driven logging, asynchronous cache scanning (ACS), and multi-undo logging to achieve the above stated goals.

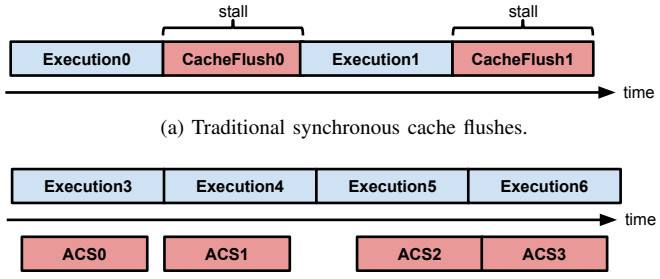
A. Base Design: Undo Logging

While there are drawbacks with undo logging, we nonetheless based PiCL on undo logging. The key advantage is that, unlike redo logging, a translation table is not needed for correctness as all write-backs and cache evictions are written to the canonical memory locations. Being based on undo logging, PiCL also inherits other desirable characteristics, such as requiring little change to existing memory controller architecture. However, the main performance problem of undo logging is the *read-log-modify* access sequence that is required on every cache eviction to create and persist the necessary undo entries. For NVMs with low random access IOPS, this access sequence severely reduces the effective NVM bandwidth.

B. Cache-Driven Logging

To fix the *read-log-modify* sequence performance problem, we propose *cache-driven logging*. Instead of accessing the NVM to make undo entries, we tag each cache line with an epoch ID (EID) such that the old data blocks can be identified and copied from the CPU cache to the NVM directly and without requiring a read operation. We call this technique cache-driven logging, as the cache actively, yet lazily, writes back appropriate undo entries based on the EID difference between the residing cache line and the current system EID.

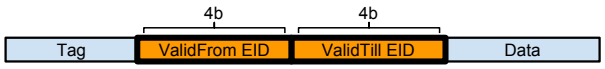
On-chip undo buffer: It is also simple to buffer many entries together on-chip and write them all together in one single IO access to the NVM. As a baseline we consider 2KB buffers to match the NVM row buffers assuming they are of the same size, give or take a few multiples to account for double buffering and for multiple channels of NVM. It is worth noting that performance degradation stemming from memory queuing delay can occur with a very large on-chip undo buffer, but it is minimal at 2KB.



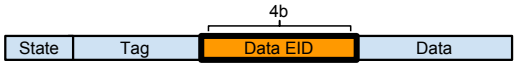
(a) Traditional synchronous cache flushes.

(b) With ACS.

Fig. 4. Execution flow comparing cache flushes and Asynchronous Cache Scanning (ACS). ACS does not run in lock-step with volatile execution like cache flushes do. Here, the gap between persisting and committing an epoch is three (ACS-gap).



(a) An undo log entry (in NVM)



(b) An augmented cache entry

Fig. 5. Meta-data for cache lines and log entries

The key idea is that when modified cache entries are first brought on-chip, they tend to remain cached long enough for a group of unrelated undo entries to be collected and efficiently written to the log before the cache lines themselves get evicted. Central to the correctness of this approach is that cache lines cannot be evicted before their undo data are flushed from the on-chip undo buffer. If this happens, the undo data will be lost upon a power failure, leaving the memory state inconsistent. To enforce this dependency between undo and in-place writes, we add a bloom filter to detect when an eviction matches with an undo entry in the buffer. If so, the buffer is flushed first. We find that this case is rare in an LLC with sufficient associativity (e.g., 8-way), and that the false-positive rate is insignificant when a sufficiently large bloom filter is used (i.e., 4096 bits vs 32 entries capacity). This filter is cleared on each buffer flush.

C. Asynchronous Cache Scan

To remove the cache flush off the critical path, it is important to know *why* it is necessary in the first place. There are two reasons why a cache flush must happen synchronously and immediately in prior work: (1) the logging system can only ensure durability of one checkpoint, and (2) the need to immediately *persist* the checkpoint. If neither holds–as in PiCL where multi-undo logging allows logging of multiple epochs concurrently–a cache flush does not have to stall the system and can be entirely asynchronous.

PiCL further defers epoch persistency to the asynchronous cache scanning (ACS) engine. Much like a cache flush, the ACS engine scans the cache and flushes the necessary dirty cache blocks. The difference is that instead of writing back all dirty cache blocks, ACS only targets the cache blocks with EID tags matching the currently *persisting* epoch ID. When ACS found a matching cache line, the line is written to memory in-place and made clean.

Shown in Figure 4, ACS trails behind by a threshold (three epochs as shown) and does not happen in lock-step with execution. We refer to this epoch ID difference as the *ACS-gap*. This parameter is system-configurable and is independent from other parameters. If ACS-gap is set to zero, the cache scan is initiated right after the commit to persist it. ACS opportunistically accesses the EID tag array and the dirty bit array to write back the appropriate data entries. More powerfully, ACS can be delayed by a few epochs to save even more bandwidth. Upon finishing the cache scan, ACS also flushes the on-chip undo buffer if the buffer contains entries of the same epoch as the final step of persisting the epoch.

D. Multi-Undo Logging

Last, we introduce multi-undo logging. Like its baseline design, multi-undo logging stores undo entries in a contiguous memory region in the NVM such that they can be used to patch main memory and restore it to a consistent state upon a hardware failure. There are two key semantic differences. First, multiple outstanding committed-but-not-persisted epochs are allowed. Second, undo entries of different commits can be written in a single undo log without any ordering restriction; except for entries of the same address (i.e., `A2` can be persisted before `B1`, but not `A1`) to preserve the order of delivery guarantee for ACID consistency. Taken together with cache-driven logging and ACS, the implications of being able to have multiple outstanding epochs are significant. Critically, we have removed the need to stall and "stop-the-world" to flush the cache, enabling low-overhead crash consistency for systems with very large on-chip caches.

Unlike the baseline undo logging scheme where undo entries implicitly belong to the most recently committed epoch, in multi-undo logging, undo entries of different epoch steps are co-mingled. Furthermore, entries can belong to not just one, but multiple consecutive epochs. In PiCL, each entry is tagged with a pair of EIDs: a `ValidFrom` EID and a `ValidTill` EID, as shown in Figure 5a. These values reflect the epoch in which the cache block was first modified to this particular value, and when the block is again modified to a different value. In other words, they specify the validity range of the data, which is used not only during recovery, but also to indicate whether the entry is expired and can be reclaimed or garbage collected. Again, by co-mingling undo entries of different commits together in a contiguous memory block, we maximize sequential write performance at the NVM device.

E. All Together

Multi-undo example: Figure 6 depicts a sequence of writes across three epochs, showing when undo entries are created in multi-undo logging. Here, the top row indicates writes and cache evictions from the processor, and the bottom row shows the resulting undo entries and write-backs from these writes.

Initially, `A`, `B`, and `C` are unmodified. In `Epoch1`, all three are modified. Assuming they were clean previously, undo

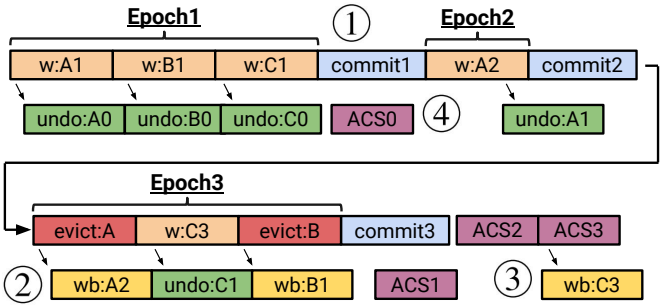


Fig. 6. Example of when entries are created in multi-undo logging. Shorthands: `w` = write to cache, `undo` = data appended to on-chip undo buffer, `evict` = evictions, `wb` = write-back data in-place.

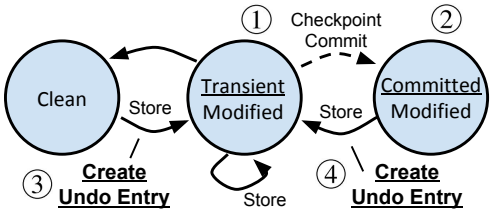


Fig. 7. Additions to the cache state transition graph in the LLC cache, assuming MESI.

entries `A0`, `B0`, and `C0` (data versions from `Epoch0`) are created and appended to the undo log. In `Epoch2`, `A` is again modified, leading to undo of `A1`. In `Epoch3`, only `C` is modified, and both `A` and `B` are evicted due to cache eviction policy.

Difference 1: dirty data are not forcibly written back at each checkpoint. This allows full overlap between execution and commit phases. A logical commit for `Epoch1` is seen at ①. In prior work, this commit would also imply persistency which requires all modified cache lines (`A1`, `B1`, and `C1`) to be flushed from the cache so to make `Epoch1` durable. In contrast, PiCL defers the persistency of checkpoints to ACS.

Difference 2: out-of-order undo creations and cache evictions. Notice that at ② in Epoch3, `A2` is evicted to memory first, logically out-of-order with regards to the undo of `C1` and cache eviction of `B1`. In some epoch-based transaction schemes [44, 45], it is necessary by semantics for data of `Epoch1` (B1 and C1) to be persisted in NVM first before `Epoch2` (A2) can be. Multi-undo allows full reordering of its operations, facilitating efficient coalescing of undo writes and putting no constraint on cache eviction policy.

Purpose of ACS: In PiCL, ACS assumes the responsibility of finding which dirty data is still in the cache, flushing them, and making checkpoints durable and available for recovery. In the example, there are four ACS operations in total, but only `ACS3` at ③ actually writes data to NVM. Take `ACS0` ④ for instance: while `A`, `B`, and `C` are dirty in the cache, they do not have to be flushed to NVM as the necessary undo entries have already been created to be used in recovery. Note that ACS will also flush the on-chip undo buffer if it has not already been flushed.

Validity range: In multi-undo logging, undo entries are tagged with `ValidFrom` and `ValidTill` EIDs, and they are determined based on when they are last modified and the current EID. For example, undo entries for `A0`, `B0`, and `C0` all will have `ValidFrom` set to 0 (last EID they were modified), and `ValidTill` field to 1 (the current epoch ID).

In contrast, undo for `C1` will be tagged $< 1, 3 >$, which means this entry should be used not only when reverting back to `commit1`, but also `commit2` (but not `commit3`).

IV. Implementation

In this section we describe the hardware implementation of the ideas presented in §III. PiCL is entirely implemented in the SRAM cache hierarchy where we add EID tags to cache lines and track updates to these tags to create undo entries. We also touch on OS support such as garbage collection of the undo log, and following that, we discuss other related topics such as I/O consistency, multi-channel memory, and possible NVM-DRAM buffer extension.

A. Caches

EID Tags: To implement the cache driven logging architecture, we add an EID tag to each cache line, as shown by Figure 5b. The EID is a small tag (e.g., 4-bit values are sufficient) used to indicate the epoch that the line was last modified in. The basic idea is to compare this EID tag with the current system EID to track when a cache line is modified across epoch boundaries (cross-EID stores), in order to make undo data prior to the modification. We will add this tag to cache lines in both the LLC and the private caches (L1/L2).

Three EIDs are of note in PiCL: (i) `PersistedEID`, or the most recent fully persisted, fully recoverable checkpoint, (ii) `SystemEID`, or the currently executing, uncommitted epoch, and (iii) per-cache block `TaggedEID` which denotes when it was last modified.

Figure 7 shows the relevant state transitions in the LLC, indicating when undo entries are created. Like an ordinary LLC, a valid cache line can either be in the *clean* or *modified* state. Logically (but not physically), modified state is further disambiguated as *transient* ① when `TaggedEID` equals to `SystemEID`, or *committed* ② when they are not. Storing to transient cache blocks has no effect, while storing to committed data produces undo entries.

A line loaded from the memory to the LLC initially has no EID associated. When receiving a store request from L1, the clean data is written back as an undo entry with the `ValidFrom` tag set to `PersistedEID` ③. Otherwise, only on cross-EID stores ④ that undo entries are created with the `ValidFrom` field set to `TaggedEID`. In both cases, the EID tags are updated to the `SystemEID` value, and the `ValidTill` field of the log entry is also `SystemEID`.

The undo hooks for private caches (L1/L2), shown in Figure 8, are largely the same as the LLC: the EID tag is compared on store, and if they are different, the private cache updates the EID tag and forwards undo data entries to the

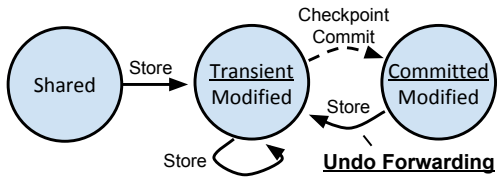


Fig. 8. Additions to the cache state transition graph in the private cache, assuming MESI. PiCL does not modify the cache coherence protocol itself, but only adds additional log forwarding actions to the LLC.

LLC (the EID tag at the LLC is also updated to `SystemEID`). Tracking the L1 cache is necessary because the LLC does not observe store hits at the L1, and it is possible for a cache line to be modified repeatedly across multiple epochs without ever be written back to the LLC. Stores are not on the critical path as they are first absorbed by the store-bufffer, and as the EID tag is fairly small (4 bits) the comparison cost is negligible compared to other cache coherence operations like TLB lookup and tag check. Additionally, undo hooks can be relocated to the L2 if the L1 is write-through.

In summary, there are three common ways in which an undo entry is created: (1) when a *clean* cache line is first modified, (2) when a *committed* modified cache line is modified, or (3) when undo data is forwarded from the private caches. Less commonly is when there are atomic (or write-through) operations at the LLC. They are handled in the same manner as described for the private caches. Notice that PiCL makes no changes to the cache coherency protocol nor to cache eviction policy–it simply adds the necessary hooks to create undo entries.

Undo Buffers: Once created, undo entries are stored directly into an on-chip buffer, waiting to be flushed to NVM in bulk to match the row buffer size (2KB in the evaluation) and maximize sequential write performance. Again, there is no need to have separate buffers–undo entries of mixed EID tags can be all coalesced in the same queue.

The undo buffer is flushed when it is full (32 entries) and double buffering can be employed to accept further incoming undo entries while the buffer is being flushed. The buffer is also flushed by ACS when persisting an EID that matches the oldest undo entry in the buffer. To be conservative, we flush the undo buffer on every ACS in the evaluations.

ACS: The asynchronous cache scan is implemented in hardware only at the LLC. The mechanism is simple: it *opportunistically* scans the EID array at the LLC for valid lines tagged with the targeted EID value (no tag checks required). For each match, if the line is dirty, it is flushed to memory and set *clean*–similar to a cache flush. If there are dirty private copies, they would have to be snooped and written back.

Note that while ACS and ordinary cache accesses occur simultaneously, there are no incorrect data races between the two. For instance, in Figure 6, if `ACS1` occurs prior to `w:A2`, then `A1` would be written to memory, and then another copy of `A1` will be appended to the undo log (as a result of modifying a clean cache line). The sole difference is an extra write to memory, and in either case correctness is preserved.

B. NVM interface and OS Interface

There is no explicit hardware requirements at the NVM interface–PiCL is designed to be compatible with existing DDR3/DDR4 interfaces to simplify implementation and reduce costs. Most of the bookkeeping tasks such as log allocation, garbage collection, and crash recovery are handled by the OS.

Log allocation: The OS can allocate a block of memory (e.g., 128MB) in NVM and pass the pointer to the PiCL hardware so that the buffer knows where to flush log data to. If the logs run out of space for any reason, the OS can be interrupted to allocate more memory and update the pointers. Memory allocations need not be contiguous as long as the necessary pointers are maintained.

Garbage collection: Using the `ValidTill` tags, the OS can determine when a log entry stored in NVM expires and can be safely discarded (garbage collected). To lower bookkeeping cost, we can group entries into super blocks (eg. 4KB blocks), and set its expiration to be the max of the `ValidTill` field of the member entries.

Crash handling procedure: Recovering from a power failure, the OS first reads a memory location in NVM for the last valid and persisted checkpoint (`PersitedEID`). Like other undo logging schemes, the OS then scans the log backward from the latest entries for undo entries with `ValidFrom` and `ValidTill` range that covers this EID and applies them to memory. It is important to scan and apply the undo entries from the tail of the log backward, as, like in other undo logging designs [23], there could be multiple undo entries for the same address of the previous epoch but only the oldest one is valid. A full log scan is not necessary: the procedure can stop when the `ValidTill` tag of the next super block goes below the `PersitedEID` value.

C. Discussion

Multi-core: Data writes from different cores and threads share the same epoch ID (i.e., `SystemEID`), thus recovery applies system-wide. It is possible for some applications and memory spaces (such as memory-mapped I/O) to be exempted from EID tracking, but shared system structures like the page table and memory allocation tables must be protected at all time.

I/O Consistency: Prior work in crash recovery has studied the consistency of I/O side effects extensively [23, 32, 46]. In general, I/O reads can occur immediately, but I/O writes must be buffered and delayed until the epochs that these I/O writes happened in have been fully persisted.

Checkpoint Persist Latency: Because ACS delays epoch persist operations, the effective latency is the epoch length multiplied by the ACS-gap value. I/O and I/O side effects also have to be delayed correspondingly, which might be detrimental for certain workloads. If I/O is on the critical path, the system can forcefully end the current epoch, conduct a *bulk* ACS to write-back all outstanding undo entries, and release any pending I/O write. *Bulk* ACS is an extension where a range of EIDs is checked in a single pass and not just a single EID.

I/O Latency: While delaying I/O activities may affect I/O-sensitive workloads, it has shown that (1) throughput is not

affected by I/O delays, and (2) the performance of latency-bound workloads remains tolerable for epochs of up to 50ms in length [32]. Moreover, as noted in prior work [32], I/O writes do not have to be delayed for *unreliable* I/O interfaces such as TCP/IP as they have built-in fault tolerance at the application level, or with storage media that have idempotent operations. Consumer applications like video playback can similarly tolerate some degree of faults [47].

Recovery Latency: A thorough study of the recovery procedure for an undo-based logging system [24] finds that given a checkpoint period of 100ms, the worst-case recovery latency is around 620ms. Due to ACS and the co-mingling of undo entries, the worst-case recovery latency might be lengthened by a few multiples when PiCL is used. Even so, the decrease in runtime overhead is well worth the increase in recovery latency. For instance, supposing recovery latency increases to 4400ms, system availability is still 99.995% assuming a mean time between failures (MTBF) of one day. In contrast, a 25% runtime overhead amounts to 21600 seconds of compute time lost per day, or 25% fewer transactions per second.

DRAM Buffer Extensions: To accommodate NVMs with low IOPS, some systems include a layer of DRAM memory-side caching to cache hot memory regions. This DRAM layer typically caches data at page-size granularity (4KBytes) to capture spatial locality. PiCL functions well with both write-through and write-back DRAM. With *write-through* DRAM caches, no modifications are needed. The intuition is that the semantics of writes for NVM and for PiCL remain equivalent with and without the DRAM cache. Otherwise, if working in *write-back* mode, assuming the DRAM is an inclusive cache but with page granularity, we can apply PiCL to the DRAM cache and treat the LLC as a private cache. PiCL would still track changes at the cache block granularity.

V. Prototyping

We have fully implemented PiCL as a hardware prototype with OpenPiton [31] to demonstrate the feasibility and correctness of multi-undo and cache-driven logging. The design is implemented in Verilog, and the system is fully recoverable when running micro-benchmarks in simulation as well as in hardware. The prototype runs at OpenPiton's nominal FPGA frequency of 66MHz [31].

A. OpenPiton Implementation

There are three levels of cache in OpenPiton: private L1 and L2 per tile, and a shared-distributed LLC [2]. The L1 is write-through, so only L2 and LLC caches are modified to support cache-driven logging. For these caches, we precede each store operation with a cross-EID store check, and make an undo entry if necessary. To simplify the design somewhat, L2 undo logs are always sent to the LLC, and LLC logs are sent to the off-chip interface. The off-chip interface implements a buffer and a bloom filter as described in §III-B.

The biggest challenge was that in OpenPiton, cache blocks are 16 bytes in the private caches, but 64 bytes in the LLC. If we were to track 64-byte blocks, the L2 would have to check the EIDs of not only the current sub-block, but also the other three blocks that form the quads for a 64-byte block for every store. As a trade-off between complexity and performance, we adopted a tracking granularity of 16 bytes instead of 64 bytes. The drawback is that the LLC now has four EID tags per cache entry, one per each sub-block.

Lastly, as part of the software interface, we have completed and tested the OS epoch boundary handler. The purpose of the handler is to save internal states that are not part of the memory space, such as the register files and arithmetic conditions at each checkpoint. These are cacheable stores to a fixed memory address per core visible only to the OS. The handler is implemented as a periodic (adjustable) timer-based interrupt that occurs transparently to userspace programs. The actual stores are done by the interrupt vector executed by the CPU. Note that this handler is a necessary ingredient to all epoch-based checkpointing schemes and not only to PiCL.

[2]OpenPiton named them L1, L1.5, and distributed L2 instead.

TABLE III
PiCL Hardware Overheads When Implemented on FPGA

Logic	**LUTs**	**%**	**BRAM**	**Slices**	**%**
L2	232	0.12%	L2	1	1.04%
LLC	1400	0.70%	LLC	3	3.13%
Controller	198	0.10%			
Total	**1830**	**0.92%**		**4**	**4.17%**

TABLE IV
System configuration

Core	2.0GHz, in-order x86 1 CPI non-memory instructions
L1	32KB per-core, private, single-cycle 4-way set associative
L2	256KB per-core, private 8-way set associative, 4-cycle
LLC	2MB per-core 8-way set associative, 30-cycle
Memory Link	64-bit wide @ 1.6GHz (12.8GB/s)
NVM Timing	FCFS controller, closed-page 128ns row read, 368ns row write

B. Hardware Overheads

Table III summarizes the hardware overhead when the design is implemented on a Xilinx Genesys2 FPGA. Total logic overhead (LUTs and registers) are less than 1%, with the LLC modifications being more than 75% of the overhead due to the need for more buffering than at the L2. The EID arrays in the L2 and LLC account for 4.17% of Block RAM (BRAM) allocation. BRAM overhead is a little more than expected because the LLC maintains four EID values per 64-byte cache as mentioned earlier, but still reasonable.

VI. Evaluation

A. Methodology

As PiCL is designed to provide crash consistency for legacy software, we used the SPEC2k6 benchmark suite to profile performance. Specifically, we use a modified Pin-based trace-driven PRIME simulator [48, 49] to simulate each benchmark

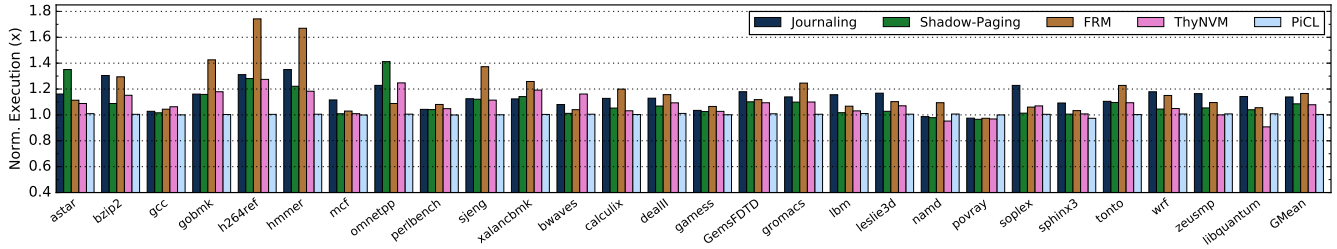


Fig. 9. Single-core total execution time, normalized to ideal NVM (lower is better).

with the most representative 1-billion cycles traces using publicly available SimPoint regions [50, 51]. Multi-program studies profile each trace for 250M instructions, and they are kept running until all traces have finished 250M instructions to sustain loads on other programs.

The default system configuration is as detailed in Table IV. NVM row buffer miss latency is set at 128ns for reads and 368ns for writes–similar to the other studies of byte-addressable NVMs [10, 27]–[29]. Epoch length is set to 30-million instructions by default to be consistent to prior work [23]–[26], though PiCL can support and would benefit from longer epoch lengths. To evaluate multi-core performance, we run multiprogram workload mixes of eight randomly chosen benchmarks each as listed in Table V.

We compare PiCL against four representative software-transparent crash consistency solutions: Journaling, Shadow-Paging, FRM, and ThyNVM. **Journaling** is based on redo logging [36], as described in §II-B. **Shadow-Paging** is largely similar to Journaling, but increases the tracking granularity to be page size (4KBytes). Page copy-on-write (CoW) is done on a translation write miss, and page write-back is done on a commit. We made two further optimizations to make its performance acceptable: (1) CoW is done locally within the memory module to decrease memory bandwidth utilization, and (2) even though the page is written back, the entry is retained to avoid misses to the same memory page in the next epoch step. **FRM** is a representative undo logging scheme frequently used in "high-frequency" (10-100ms time period per epoch) checkpoint designs [22]–[25, 37]. The general workings are as described in §II-B. **ThyNVM** [26] is based on redo logging, with mixed 64B/4096KB logging granularity and single checkpoint-execution overlapping capability. We implemented ThyNVM as described, though we did not implement the DRAM cache layer and instead allocated the redo buffer in NVM. We also assumed that cache snooping is free, even though it is required for correctness but not described in detail in the original paper. **Ideal NVM** is a model that has no checkpoint nor crash consistency, given for performance comparison.

For Journaling, Shadow-Paging, and ThyNVM, the translation table is configured with 6144 entries total (2048 and 4096 entries for block and page respectively for ThyNVM) at 16-way set-associative to be consistent with prior studies [26].

B. Performance Overhead

We evaluate single-threaded performance across SPEC2k6. Simulations show that where prior work can slow down the system by as much as 1.7× for single-threaded workloads (Figure 9) and between 1.6× and 2.6× for multi-threaded (Figure 10), PiCL provides crash consistency with almost no overhead. Only for rare cases like `sphinx3` that PiCL loses 1-2% of performance due to the undo buffer flushing 2KB of contiguous data blocking other read or write requests.

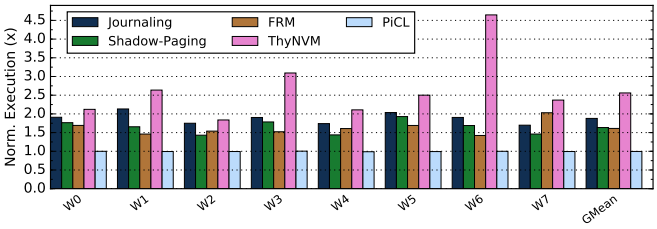


Fig. 10. Eight-thread multi-core performance, normalized to ideal NVM (lower is better).

TABLE V
MULTIPROGRAM WORKLOADS–RANDOMLY CHOSEN.

W0	h264ref soplex hmmer bzip2 gcc sjeng perlbench hmmer
W1	gcc gobmk gcc soplex bzip2 gamess tonto gcc
W2	bzip2 lbm gobmk perlbench cactusADM bzip2 h264ref mcf
W3	gcc bzip2 tonto cactusADM astar bzip2 namd zeusmp
W4	perlbench wrf gobmk gcc namd gobmk milc bzip2
W5	omnetpp bzip2 bzip2 gobmk sjeng perlbench bzip2 gobmk
W6	gcc tonto gamess cactusADM dealII gobmk omnetpp bzip2
W7	gcc wrf gcc bzip2 gamess gromacs gcc perlbench

As elaborated in §II, there are two primary causes: (1) the increase in cache flush frequency, and (2) the increase of IOPS because of non-sequential logging.

Increased cache flushes. Forced cache flushes due to redo buffer overflows, as described in §II-B is one key reason for poor performance in redo-based schemes (Journaling, Shadow, and ThyNVM). To give more insight, Figure 11 plots the actual commit frequency when accounting for translation table overflows. Normally, there is one commit per 30M instructions, but we see that Journaling can commit as much as 16× to 64× more frequently than PiCL. Note that undo-based approaches (PiCL and FRM) do not suffer from this problem.

That said, write characteristics vary from workload to workload. The write set is small for compute intensive workloads and the translation table can track them quite consistently. Workloads with sequential write traffic (e.g., *mcf*) also favor Shadow-Paging since its page-sized entries can track up to 64 cache lines per entry. However, workloads with less spatial

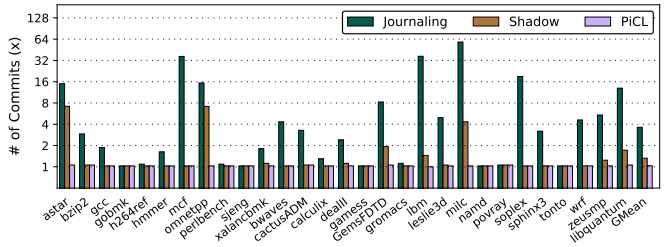


Fig. 11. Average number of commits per 30M instructions (single-threaded, lower is better). By default there is one commit per 30M, but hardware translation table overflow forces epochs to commit early.

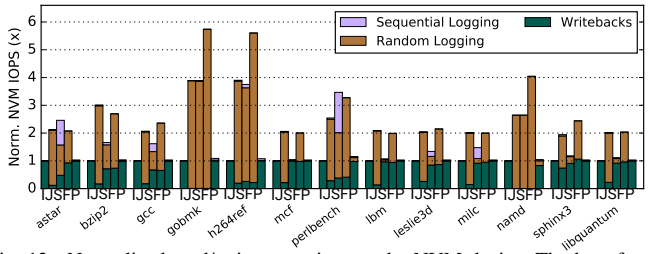


Fig. 12. Normalized read/write operations to the NVM device. The bars from left to right: [I]deal NVM, [J]ournal, [S]hadow-Paging, [F]RM, and [P]iCL.

locality like *astar* are neither suitable for Journal nor Shadow-Paging, resulting in many more commits and forced cache flushes.

Increased IOPS due to logging. Poor performance can also be explained by increase of I/O operations due to non-sequential logging operations, in light of the poor random access performance of NVMs. Figure 12 plots the IOPS at the memory of selected benchmarks, normalized to the write traffic of `Ideal NVM`[3]. Write activities are grouped into 3 categories: `sequential`, `random`, and `write-back`. `Sequential` denotes the accesses to NVM that fill up the row buffer. As an example, for Shadow-Paging, this indicates the copy on write (CoW) operations and page write-back. `Random` here denotes the number of extra cache line read/write and cache flushes for Journal and Shadow-Paging, and in-place write count for PiCL. Note that the plot does not indicate the raw size of these operations (i.e., reading a 4KB memory block counts as one operation).

In Figure 12, we see that the extra IOPS generated by different checkpoint mechanisms can be as much as 2× to 6× that of the original write-back traffic. Despite having less write-backs because of the cache flushes[4], the latency of these extra accesses can substantially degrade performance. FRM incurs the highest random IOPS among the alternatives because it has to perform the *read-log-modify* access sequence for each cache eviction. Meanwhile, PiCL exhibits little extra access count to the NVM due to the block write interface for undo logging. The amount of in-place writes done by ACS is also minimal, all of which lead to a low performance overhead.

[3]`Ideal NVM` does not log so it only has eviction (write-back) traffic

[4]When flushing the cache, dirty data are made clean, obviating the need for future evictions

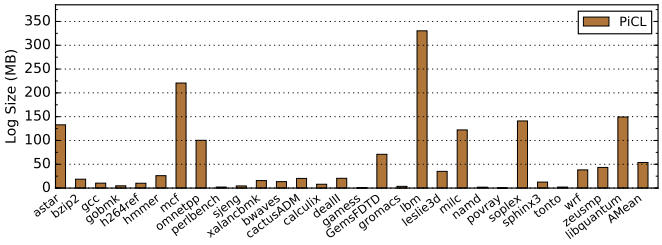


Fig. 13. Undo log size for eight epochs (240M instructions total) in `PiCL`.

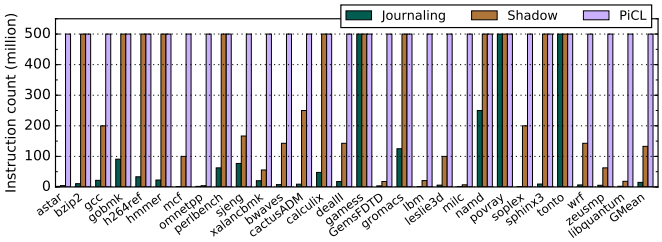


Fig. 14. Observed epoch lengths (higher is better) when the default length is set to 500M.

C. Log Storage Overhead

While PiCL incurs lower overhead than other schemes, having multiple epochs outstanding means that more storage is allocated. Nonetheless, Figure 13 shows that the majority of workloads consumes less than 50MB of log storage per eight epochs. For workloads that do produce the heaviest of logging, they remain within a few hundreds of megabytes, well within the capacity of NVM storages.

D. Very Long Epochs Support

If fine-grain recovery or I/O responsiveness are not needed, epoch length can be increased to decrease logging overheads. Unfortunately, redo-based schemes cannot handle longer epochs in many cases. Figure 14 quantifies this claim, where the default epoch length is set to 500M instructions. As we can see, 500M-instruction epochs are only possible with `Journaling` and `Shadow` for compute-bound workloads (e.g., *gamess* and *povray*). With other scheme, the effective epoch length hovers between 100M to 200M for `Shadow` and less than 50M for `Journaling`. PiCL is not limited by hardware resources but by memory storage for logging. In this study, a 1GB log storage is sufficient to maintain 500M instruction epochs for all tested workloads.

E. Sensitivity Studies

Cache sizes: As we discussed in §II-A, the larger the on-chip cache, the longer it takes to synchronously flush the dirty data at each checkpoint. Fig. 15 shows that PiCL generally has no performance overhead across cache sizes because it asynchronously and opportunistically scans dirty data. It is noteworthy that ThyNVM's overhead grows faster than other schemes: this is due to it using the redo-buffer across multiple epochs leading to greater pressure and shorter checkpoints.

NVM write latencies: To see how different byte-addressable NVMs with different write latencies would affect the results,

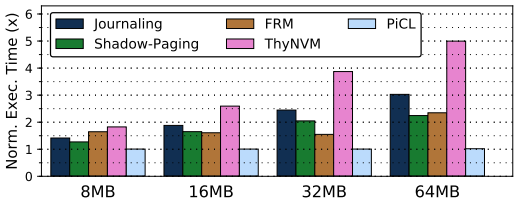


Fig. 15. Average eight-core execution time at various cache sizes, normalized to Ideal NVM (lower is better).

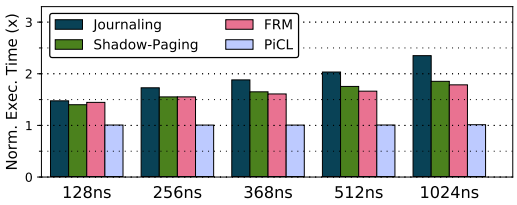


Fig. 16. Average eight-core performance at other NVM write latencies, normalized to Ideal NVM with the same write latencies (lower is better).

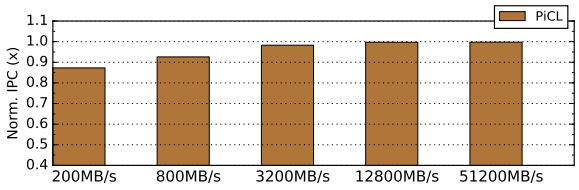


Fig. 17. Performance of PiCL under varying off-chip link bandwidth, normalized to Ideal NVM with the same bandwidth.

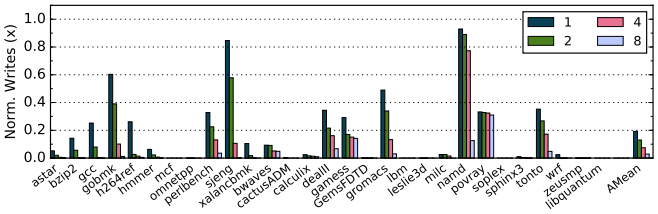


Fig. 18. In-place logging write traffic when the ACS-gap is varied, normalized to an ACS-gap value of zero.

Fig. 16 plots the multi-core performance with write latencies varied from 128ns to 1024ns (read latency is kept at 128ns). As expected, the write IOPS overhead of other schemes is more taxing at higher write latencies, leading to worsening performance. Even with a relatively optimistic write latency (128ns), the average performance overhead is 50% for previous work.

Off-chip bandwidth: PiCL makes a trade-off between off-chip bandwidth and device bandwidth: to save the additional reads and writes at the NVM, it sources undo data from on-chip cache and uses more off-chip bandwidth to transfer these data. In this experiment, we measure the smallest amount of off-chip bandwidth that does not incur performance loss. Figure 17 suggests that around 3.2GB/s of off-chip bandwidth is needed (the default DDR3 baseline is 12.8GB/s).

ACS-gap: We suggested in §III-C that deferring asynchronous cache scan (ACS) can result in less bandwidth usage. To see how effective ACS-gap is at reducing the amount of in-place logging by the ACS, in Figure 18 we plot the in-place write traffic of various ACS-gap values in comparison to an ACS-gap value of zero. We see that delaying the cache scan by even only one epoch step can greatly reduce the amount of in-place logging by 80% on average; increasing ACS-gap to eight results in a 97% reduction.

VII. Related work

There has been a substantial body of work to support crash consistency with persistent memory, though most proposed solutions are not software transparent.

Epoch-based persistency. Write-ahead logging is not the only way to provide data persistency based on epochs. Epoch-based persistency [52] lowers the performance overhead of write-ordering by separating writes into epochs, often referred to as *persist barriers*. Essentially, these related works track the EpochID with hardware meta-data per cache line, such that when a cache line tagged with EpochID *n* is evicted or overwritten, all cache lines tagged with older EIDs are first evicted to preserve write ordering at the NVM. Work in this space deals with lowering tracking overheads [16], further relaxing write ordering [17], reducing false-dependency and exposing more concurrency [12, 18, 19, 45, 53], and reducing the overhead of thread synchronizations [44]. Generally, the differences with PiCL are as follows: (1) they are not software-transparent, (2) they track writes per-thread, which can have complex write synchronization between threads, (3) persist barriers are typically very short, often tracking only a handful of stores, and (4) they can be tightly coupled with eviction policies and cache coherence protocols. Cumulatively, these differences lead to a solution space very different from PiCL.

Recent prior work has been exploring isolating and decoupling the execution and persistence phases to achieve better performance. DudeTM [54] maintains both the redo page buffer and the redo log to allow asynchronous data update in the background. Redo+Undo Logging [20] writes both undo and redo data in order to support a write-back cache hierarchy without overly constraint the cache eviction policy. HOPS [13, 55] proposes a persist buffer at the L1 to separate logging operations from the cache hierarchy; in fact, dirty cache evicts are discarded in this design. Compared with PiCL, these schemes are not software transparent. Second, they are based on redo logging, which is challenging to scale to workloads with big working sets due to the limited capacity of the redo buffer/remap-table at the memory controller. Last, they perform *log-on-store* instead of *log-on-cache-evict* as used in PiCL and other software-transparent schemes. Essentially, *log-on-store* means that every store (even repeated stores to the same address) is sent to the NVM, bypassing the cache hierarchy. While okay for small transactions, *log-on-store* uses orders of magnitude more bandwidth than *log-on-cache-evict*, resulting in unacceptable performance if applied as software-transparent consistency.

Others. Being software-transparent, PiCL can be used with legacy software, unlike proposals that augment and optimize

the software persistent API [6, 13, 14, 16, 44, 56, 57]. Secondly, PiCL is designed to have a large reordering window to optimize for write traffic. In contrast, much of the prior work in crash persistency [14, 16, 20] are built on memory fences that write data through the cache hierarchy which incurs significant bandwidth penalties.

NVM as caches (NV-LLC) has been proposed to provide multi-versioning transaction capability [9]. In contrast, PiCL is designed specifically for volatile SRAMs which lowers the implementation barrier and cost. Additionally, NV-LLC designs need to modify the eviction replacement policy to prevent transactions from being evicted out of order, and require the OS to handle overflows at the NV-LLC.

Narayanan et. al [15] also proposed a transparent persistence scheme, where on a failure, the processor flushes its volatile cache to *DRAM-based NVRAM* using residual energy from the system power supply. While this technique only protects against power loss, PiCL enables recovery for a wider class of errors. Furthermore, it is unclear if the residual energy is enough to flush big on-chip caches.

VIII. Conclusion

To capitalize on emerging byte-addressable NVM technology and decrease the adoption barrier, we propose PiCL, a software-transparent crash-consistent technique for NVMM. PiCL implements innovative solutions to overcome the scalability challenge of cache flushes, which are mandatory in prior work, and has logging patterns better matched with NVM technologies where random IOPS is low. Compared to the state-of-the-art, PiCL provides crash consistency at virtually no performance loss. Lastly, we fully implemented PiCL as an FPGA prototype in Verilog to demonstrate the feasibility of software-transparent crash consistency.

Acknowledgments

This material is based on research sponsored by the NSF under Grants No. CNS-1823222 and CCF-1438980, AFOSR under Grant No. FA9550-14-1-0148, Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement No. FA8650-18-2-7846 and FA8650-18-2-7852. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA), the NSF, AFOSR, DARPA, or the U.S. Government.

References

[1] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE Micro*, vol. 30, Jan 2010.

[2] (2015) Intel and micron produce breakthrough memory technology. [Online]. Available: https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/

[3] H. H. Li, Y. Chen, C. Liu, J. P. Strachan, and N. Davila, "Looking ahead for resistive memory technology: A broad perspective on reram technology for future storage and computing," *IEEE Consumer Electronics Magazine*, vol. 6, 2017.

[4] JEDEC, "Ddr4 nvdimm-n design standard (revision 1.0)," *Web Copy: https://www.jedec.org/standards-documents/docs/jesd248*, 2016.

[5] T. Hardware, "Intel optane dimms coming second half of 2018," *Web Copy: http://www.tomshardware.com/news/intel-optane-dimms-timing-2018,35928.html*, 2017.

[6] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011.

[7] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," *ACM Sigplan Notices*, vol. 46, 2011.

[8] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell *et al.*, "Consistent and durable data structures for non-volatile byte-addressable memory." in *FAST*, vol. 11, 2011.

[9] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013.

[10] J. Huang, K. Schwan, and M. K. Qureshi, "Nvram-aware logging in transaction systems," *Proceedings of the VLDB Endowment*, vol. 8, 2014.

[11] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage management in the nvram era," *Proceedings of the VLDB Endowment*, vol. 7, 2013.

[12] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014.

[13] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *ASPLOS'17*. ACM, 2017.

[14] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016.

[15] D. Narayanan and O. Hodson, "Whole-system persistence," *ACM SIGARCH Computer Architecture News*, vol. 40, 2012.

[16] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging."

[17] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*. IEEE, 2014.

[18] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080229

[19] S. Shin, J. Tuck, and Y. Solihin, "Hiding the long latency of persist barriers using speculative execution," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080240

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

[21] J. Ren, Q. Hu, S. Khan, and T. Moscibroda, "Programming for non-volatile main memory is hard," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, ser. APSys '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3124680.3124729

[22] P. A. Lee, N. Ghani, and K. Heron, "A recovery cache for the pdp-11," in *Reliable Computer Systems*. Springer, 1985, pp. 115–125.

[23] Y. Masubuchi, S. Hoshina, T. Shimada, B. Hirayama, and N. Kato, "Fault recovery mechanism for multiprocessor servers," in *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*. IEEE, 1997.

[24] M. Prvulovic, Z. Zhang, and J. Torrellas, "Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 30, no. 2. IEEE Computer Society, 2002.

[25] D. J. Sorin, M. M. Martin, M. D. Hill, and D. A. Wood, "Safetynet: improving the availability of shared memory multiprocessors with global

checkpoint/recovery," in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE, 2002.

[26] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling software-transparent crash consistency in persistent memory systems," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015.

[27] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555758

[28] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montao, "Improving read performance of phase change memories via write cancellation and write pausing," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, Jan 2010.

[29] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014.

[30] C.-H. Lai, J. Zhao, and C.-L. Yang, "Leave the cache hierarchy operation as it is: A new persistent memory accelerating approach," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3061639.3062272

[31] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang *et al.*, "Openpiton: An open source manycore research framework," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2. ACM, 2016.

[32] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas, "Revivei/o: Efficient handling of i/o in highly-available rollback-recovery servers," in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*. IEEE, 2006.

[33] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar *et al.*, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, vol. 34, 2014.

[34] T. Singh, S. Rangarajan, D. John, C. Henrion, S. Southard, H. McIntyre, A. Novak, S. Kosonocky, R. Jotwani, A. Schaefer, E. Chang, J. Bell, and M. Co, "3.2 zen: A next-generation high-performance x86 core," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2017.

[35] O. Lascu, B. Xu, E. Ufacik, F. Packheiser, H. Kamga, J. Troy, M. Kordyzon, and W. G. White, "Ibm z14 technical guide," *Web Copy: http://www.redbooks.ibm.com/redpieces/pdfs/sg248451.pdf*, 2017.

[36] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems (TODS)*, vol. 17, 1992.

[37] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The recovery manager of the system r database manager," *ACM Computing Surveys (CSUR)*, vol. 13, 1981.

[38] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM Journal of Research and Development*, vol. 52, July 2008.

[39] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM Journal of Research and Development*, vol. 52, 2008.

[40] Intel, "Intel optane memory series 32gb m2 80mm," *Web Copy: https://www.intel.com/content/www/us/en/products/memory-storage/optane-memory/optane-32gb-m-2-80mm.html*, 2017.

[41] L. Tokar, "Intel optane memory review," *Web Copy: http://www.thessdreview.com/our-reviews/ngff-m-2/intel-optane-memory-module-review-32gb-every-pc-user-know/6/*, 2017.

[42] M. Stanisavljevic, H. Pozidis, A. Athmanathan, N. Papandreou, T. Mittelholzer, and E. Eleftheriou, "Demonstration of reliable triple-level-cell (tlc) phase-change memory," in *2016 IEEE 8th International Memory Workshop (IMW)*, May 2016.

[43] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems," *IEEE Computer Architecture Letters*, vol. 14, July 2015.

[44] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015.

[45] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872381

[46] P. Ramachandran, S. K. S. Hari, M. Li, and S. V. Adve, "Hardware fault recovery for i/o intensive applications," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, 2014.

[47] P. A. Lee and T. Anderson, *Fault tolerance: principles and practice*. Springer Science & Business Media, 2012, vol. 3.

[48] 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," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[49] Y. Fu and D. Wentzlaff, "PriME: A parallel and distributed simulator for thousand-core chips," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, 2014.

[50] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011.

[51] H. Patil and T. E. Carlson, "Pinballs: Portable and Shareable User-level Checkpoints for Reproducible Analysis and Simulation," in *Proceedings of the Workshop on Reproducible Research Methodologies (REPRODUCE)*, 2014.

[52] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629589

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

[54] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory." ACM, April 2017. [Online]. Available: https://www.microsoft.com/en-us/research/publication/dudetm-building-durable-transactions-decoupling-persistent-memory/

[55] S. Haria, S. Nalli, M. Swift, M. Hill, H. Volos, and K. Keeton, "Hands-off persistence system (hops)," in *Nonvolatile Memories Workshop*, 2017.

[56] S. Gao, J. Xu, T. Härder, B. He, B. Choi, and H. Hu, "Pcmlogging: Optimizing transaction logging and recovery performance with pcm," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, 2015.

[57] F. Nawab, D. R. Chakrabarti, T. Kelly, and C. B. Morrey III, "Procrastination beats prevention," Tech. Rep. HPL-2014-70, HP Labs, Tech. Rep., 2014.