2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

Lazy Persistency: a High-Performing and Write-Efficient Software Persistency Technique

Mohammad Alshboul, James Tuck, and Yan Solihin
Department of Electrical and Computer Engineering
North Carolina State University
{*maalshbo, jtuck, solihin*}*@ncsu.edu*

***Abstract*—Emerging Non-Volatile Memories (NVMs) are expected to be included in future main memory, providing the opportunity to host important data persistently in main memory. However, achieving persistency requires that programs be written with failure-safety in mind. Many persistency models and techniques have been proposed to help the programmer reason about failure-safety. They require that the programmer *eagerly* flush data out of caches to make it persistent. Eager persistency comes with a large overhead because it adds many instructions to the program for flushing cache lines and incurs costly stalls at barriers to wait for data to become durable.**

To reduce these overheads, we propose *Lazy Persistency* (LP), a software persistency technique that allows caches to slowly send dirty blocks to the NVMM through natural evictions. With LP, there are no additional writes to NVMM, no decrease in write endurance, and no performance degradation from cache line flushes and barriers. Persistency failures are discovered using software error detection (checksum), and the system recovers from them by recomputing inconsistent results. We describe the properties and design of LP and demonstrate how it can be applied to loop-based kernels popularly used in scientific computing. We evaluate LP and compare it to the state-of-the-art *Eager Persistency* technique from prior work. Compared to it, LP reduces the execution time and write amplification overheads from 9% and 21% to only 1% and 3%, respectively.

***Keywords*-Emerging Memory Technology; Memory Systems; Multi-core and Parallel Architectures;**

I. Introduction

Non-volatile memories (NVM) are increasingly considered as a fabric for future main memory, augmenting or replacing DRAM. For example, Intel's 3D-XPoint memory [1] drive product was launched recently, while a DIMM version is coming soon. These NVMs promise byte-addressability and access time that is not much slower than DRAM, but they are constrained by slow and high-power writes as well as limited write endurance.

Having a *Non-Volatile Main Memory* (NVMM) in the system provides an opportunity to host important data persistently in the main memory. However, achieving persistency requires programs to be written with failure-safety in mind, so that persistent data is not corrupted upon a failure. Typically, the system provides a *persistency model* consisting of (1) *durability ordering*, which specifies at which point (or order) stores reach the non-volatile domain in the NVMM, and (2) *atomic durability*, which specifies which group of stores are to be made durable atomically. Without durability ordering, stores are made durable following the order in which cache blocks are written back from the last level cache (LLC), making it hard to reason about failure safety. Thus, persistency models were proposed in the literature [2], [3], [4], [5], [6] that define primitives, such as a persist barrier, that the programmer can use to force stores to be (eagerly) flushed out of the last level cache (LLC) to NVMM. We refer to them as *Eager Persistency*. In the case of Intel's PMEM model [5], a sequence of store, cache line flush, and store fence is required to define the durability ordering of a store instruction, while atomic durability is left to the programmer to address, e.g. by implementing redo logging in software.

Eager persistency is costly for several reasons. First, the extra instructions added to the program to define durability ordering increase the instruction count substantially. Second, these instructions are long latency since they deal with the entire cache hierarchy, including the LLC and NVMM, hence they frequently cause pipeline stalls [7]. Third, by forcing writes to go to NVMM, the NVMM write endurance is reduced, not to mention the high power overheads that come with more frequent writes. Fourth, it goes against the principle of *make the common case fast*, where normal (failure-free) execution is common and persistency failures are rare.

In this paper, we are rethinking the relationship between the software view of failure-safety and the persistency model. We propose *Lazy Persistency* (LP), a high-performing and write-efficient softwre persistency technique. Instead of forcing caches to send targeted blocks to the NVMM, we let caches slowly send written blocks to the NVMM through natural evictions. Thus, there are no additional writes to NVMM, no decreased write endurance, and no performance problems associated with cache line flushes and durable barriers. A persistency failure, which occurs when a computation's result has not been made durable, is discovered using software error detection (checksum). In such a case, the persistent checksum in memory is inconsistent with the persistent data it protects. The system recovers by recomputing the inconsistent results. The recovery is more complex than Eager Persistency, but the common case, execution without persistency failure, is fast. In order to achieve Lazy Persistency, a few things are required. First, the programmer needs to define the persistency region granularity, which is the unit of recovery. Second, a persistency failure detection mechanism needs to be embedded into the code. Third, the recovery code corresponding to the persistency granularity is needed. Therefore, Lazy Persistency trades off longer recovery time and complexity for faster and cheaper normal execution.

To illustrate how Lazy Persistency works, consider a simple code example shown in Figure 1. The left column shows the original code that operates on array A and B and stores the results in arrays C and D. With Eager Persistency, each iteration may be wrapped in a durable transaction, augmented with cache line flushes and store fence appropriately. If the durable transaction is implemented in software, there will be code to create log entries and more durable barriers separating log creation and modifications to C and D. With Lazy Persistency, we compute a checksum based on the computation results of C and D. No log is created, and no cache line flushes or durable barriers are needed. The checksum acts as an error detection code. If a failure is detected (e.g. power failure, software crash), the recovery code is triggered. The recovery code visits computation results and compares them against the checksum. If some computation values of C or D were not persisted, the checksum mismatches the stored checksum, and the recovery performs recomputation. In the example, the recovery code uses a more conservative Eager Persistency approach to guarantee forward progress after a failure.

Original Code:
```
for (i=0; i<N; i++) {
  C[i] = foo(A[i],B[i]);
  D[i] = bar(A[i],B[i]);
}
```

Eager Persistency Code:
```
for (i=0; i<N; i++) {
  DurableTX {
    C[i] = foo(A[i],B[i]);
    CLFLUSH(&C[i]);
    D[i] = bar(A[i],B[i]);
    CLFLUSH(&D[i]);
    SFENCE;
  }
  last_i = i;
  CLFLUSH(&last_i);
  SFENCE;
}
```

Lazy Persistency code:
```
for (i=0; i<N; i++) {
  C[i] = foo(A[i],B[i]);
  D[i] = bar(A[i],B[i]);
  CkSum(i,C[i],D[i]);
}
```

Recovery code:
```
for (i=0; i<N; i++) {
  if (!validCkSum(i,C[i],D[i])) {
    C[i] = foo(A[i],B[i]);
    CLFLUSH(&C[i]);
    D[i] = bar(A[i],B[i]);
    CLFLUSH(&D[i]);
    SFENCE;
  }
  last_i = i;
  CLFLUSH(&last_i);
  SFENCE;
}
```

Figure 1: High level illustration of Lazy Persistency code.

Our contributions in this paper are:

1) We propose Lazy Persistency, a novel software persistency technique that exploits the normal cache eviction mechanism to provide low overhead failure-safety for persistent data in NVMM. The basic version of Lazy Persistency does not depend on any hardware persistency support, hence it is platform independent.
2) We evaluate the proposed technique and compare it to the state-of-the-art Eager Persistency technique EagerRecompute [8]. Our results show that compared to EagerRecompute, Lazy Persistency reduces the execution time and write amplification overheads from 9% and 21% to nearly nothing, 1% and 3%, respectively.
3) We propose a simple hardware support for Lazy Persistency that provides an upper bound for the recovery time, and show that it incurs only a small increase in the number of writes.

The rest of the paper is organized as follows. Section II discusses relevant background. Section III describes the design of Lazy Persistency. Section IV describes implementation details of Lazy Persistency using a tiled matrix multiplication example. Section V describes the evaluation methodology, while Section VI discusses the results of the experiments. Section VII discusses related work. Finally, Section VIII concludes the paper.

II. Background

Due to the non-volatile property, NVMM provides an attractive substrate for providing crash recovery. Crash recovery is useful for reducing the reliance on the file system for hosting data. More recently, NVMM crash recovery was shown to provide an alternative to traditional checkpointing [8].

However, achieving crash recovery requires data in NVMM to be in crash-consistent state at all time. In recent years, persistency programming models have been proposed to allow programmers to provide crash consistency in their code [2], [3], [4], [5], [6].

A. Intel Persistency Programming Support

We choose Intel PMEM [5] as a representative Eager Persistency model to illustrate the difference between Eager Persistency with Lazy Persistency. PMEM provides several instructions, such as *clwb* and *clflushopt* in addition to existing x86 instructions such as *clflush* and *sfence*. With PMEM, programmers must explicitly flush a cache block written to by a store instruction in order to force the store to become durable. A store fence is needed afterward to avoid stores that follow it (younger stores) from becoming durable before stores that precede it (older stores). Thus, a store fence has two roles; it acts as memory barrier that ensures older stores are visible to other threads, and it acts as a durable barrier that ensures older stores (including cache line flush or write back instructions) are durable prior to younger stores. The latter role is made possible by a recent *Asynchronous DRAM Refresh* (ADR) platform specification which requires the write buffer in the memory controller (MC) to be in the non-volatile domain. Thus, when a dirty block is flushed out of the cache hierarchy into the MC, it can be considered durable. Prior to ADR, the MC write buffer is not in the non-volatile domain, hence another instruction (pcommit) is needed to flush the MC write buffer to NVMM before a store is durable. Note that PMEM does not provide a primitive to specify a durable atomic region. Hence, programmers need to create their own atomic durable region in software, e.g. through software write-ahead logging or other logging mechanisms.

To illustrate the high cost of Eager Persistency, consider the code shown in Figure 1 (left column). The programmer may implement the durable transaction as shown in Figure 2.

```
1   for (i=0; i<N; i++) {
2       logC = createLog(&C[i], C[i]);
3       logD = createLog(&D[i], D[i]);
4       logLast_i = last_i;
5       CLFLUSHOPT(logC);
6       CLFLUSHOPT(logD);
7       CLFLUSHOPT(logLast_i);
8       SFENCE;
9       logStatus = 1;
10      CLFLUSHOPT(&logStatus);
11      SFENCE;
12      C[i] = foo(A[i], B[i]);
13      D[i] = bar(A[i], B[i]);
14      last_i = i;
15      CLFLUSHOPT(&C[i]);
16      CLFLUSHOPT(&D[i]);
17      CLFLUSHOPT(&last_i);
18      SFENCE;
19      logStatus = 0;
20      CLFLUSHOPT(&logStatus);
21      SFENCE;
22      }
```

Figure 2: Illustration of a durable transaction with Eager Persistency using PMEM.

The figure shows a simple example of a PMEM implementation of a durable transaction. The example assumes that each loop iteration body forms a durable transaction[1]. Lines 2-7 show the creation of log entries for C[i], D[i], and last_i, their cache line flushes, and a durable barrier that ensures the log creation is complete prior to the next step of setting logStatus to indicate the completion of log creation (Lines 9). logStatus must also be made durable prior to modifications to the actual data (Lines 12-14). Lines 15-18 show results being written to C[i], D[i], and the loop index last_i are persisted. When that is completed, the log is no longer needed and hence can be marked appropriately (Lines 19-21). Note that four sets of cache line flushes and durable barriers are needed to implement one durable transaction.

[1]While loop iteration body provides a natural durable transaction granularity and is used for ease of illustration, other granularities are also possible.

B. Tiled Matrix Multiplication

While we use several benchmarks for evaluating our technique (Section V-C), we will use matrix multiplication to illustrate it without loss of generality. Suppose that we want to multiply matrix a with matrix b and store the result into matrix c. All matrices are square and have $n \times n$ size.

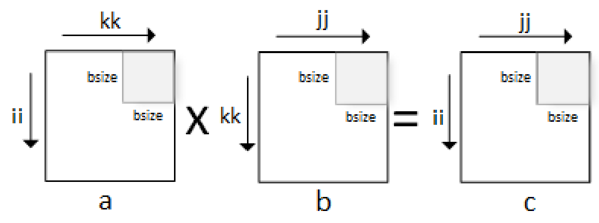

Figure 3: Tiled matrix multiplication illustration.

Figure 3 and Figure 4 illustrate a tiled matrix multiplication and show its code, respectively. Tiling is a common cache optimization for improving temporal locality of matrix-based kernels. For the matrix multiplication, assuming that *bsize* is the tile size, we assume a standard 6-loop tiling [9] which splits all the three matrices into tiles of $bsize \times bsize$ elements.

```
for (kk=0; kk<n; kk+=bsize)
    for (ii=0; ii<n; ii+=bsize)
        for (jj=0; jj<n; jj+=bsize)
            for (i=ii; i<(ii+bsize); i++)
                for(j=jj; j<(jj+bsize); j++)
                {
                    sum = c[i][j];
                    for(k=kk; k<(kk+bsize); k++)
                        sum += a[i][k]*b[k][j];
                    c[i][j] = sum;
                } //end of the j for loop
```

Figure 4: Tiled matrix multiplication (tmm) code.

The tiled matrix multiplication (*tmm*) code consists of 6 loops (from outer to inner): *kk*, *ii*, *jj*, *i*, *j*, and *k*. The outermost *kk* loop splits matrices *a* and *c* into vertical groups, each consisting of *bsize* columns. *kk* loop splits the *b* matrix into several *bsize* rows-width horizontal groups. The *ii* loop splits *a* horizontally. The intersection between the split caused by *ii* and *kk* results in squares of $bsize \times bsize$ tiles (Figure 3). Similarly, the *jj* loop splits each of matrix *b*'s horizontal *kk* groups into square $bsize \times bsize$ tiles. The innermost loops *i*, *j*, and *k*, perform partial matrix multiplication for a tile and accumulate the partial result into the c matrix elements within the tile.

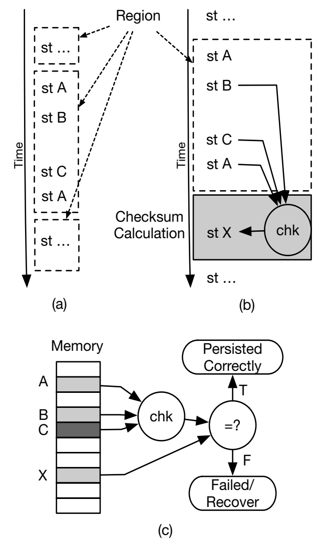

Figure 5: Lazy Persistency involves formation of regions (a), calculation of a checksum per region (b), and detection and recovery mechanism (c).

III. Lazy Persistency

A. *Definition*

Lazy Persistency requires that a programmer organize their algorithm into *LP regions*, where each region is a unit of recovery. Since Lazy Persistency relies on natural cache evictions to lazily write back data rather than eagerly flushing data to NVMM, we cannot be absolutely sure that all data becomes durable before a failure. To determine if all data in a region became durable prior to a failure, we leverage an error checking code in software. The error checking code is meta-data that the program will compute and maintain, and for convenience, we refer to this meta-data as a checksum for the rest of this section.

Figure 5(a) illustrates code divided into three LP regions. Each region contains a set of stores to persistent memory. For the stores in this region, a checksum calculation is added to the program that summarizes the content of all the written data. In the figure, the middle region has four stores (to A, B, C, and A). Figure 5(b) shows the addition of checksum calculation (X) based on all the last store values for each address, and a store that writes it to persistent memory. We assume that the programmer (either directly or through a library) adds such a checksum calculation for all regions as well as a location in memory to hold all of the checksums.

If a failure occurs during or after this region, some of the writes (A, B, or C) or the checksum (X) may fail to become durable. To detect this case, data from relevant memory locations (A, B, and C) is read; only store values that became durable prior to the failure are read out from the NVMM, other values were lost. A checksum is then calculated from them, and compared against the saved checksum. If they mismatch, we can be certain that some of the data or the checksum failed to persist before the failure. In this case, we detect inconsistent state and must recover. This checking procedure is illustrated in Figure 5(c). Note, the checksum must only include the final data written to a memory location in a Region, otherwise a recomputed checksum is certain to fail. That is why only the last write to A is shown as part of the checksum in parts (b) and (c) of Figure 5.

If a Region is found to be inconsistent with its checksum, a recovery action must be invoked. Recovery mechanisms are Region and workload dependent, and the programmer must implement a suitable recovery approach for each Region.

B. *Lazy vs. Eager Persistency*

Table I compares Lazy vs. Eager Persistency. Essentially, Lazy Persistency achieves fast normal (failure-free) execution at the expense of slower and more complex recovery. During normal execution, Lazy Persistency does not require explicit cache line flushes and durable barriers, because it relies on normal cache evictions. Furthermore, no atomic durable regions (or durable transactions) are needed in Lazy Persistency. However, to detect persistency failure, it needs to calculate and maintain a checksum in software. The checksum presents a small write amplification overhead since multiple values can rely on just one checksum. In contrast, Eager Persistency incurs much higher write amplification because there is more meta-data to write (i.e. the logs), and data is flushed early, preventing coalescing of multiple writes to different parts of the same cache block. Recovery is more complex with Lazy Persistency, because programmers need to construct recovery code.

C. *Persistency Region Choices*

What regions are suitable for Lazy Persistency? One requirement is that regions must be *associative* with respect to others. With Eager Persistency, they will be performed and persisted one after another. With Lazy Persistency, they may be persisted out of order, depending on which computation results and checksums are evicted from the cache first. Figure 6 illustrates this. Suppose there are five regions R1–R5. Each region stores four computation results and a checksum. For R1 and R5, all results and checksums are persisted. For R2, only a few results are persisted. For R3, only the results are persisted. For R4, only the checksum is persisted.

Suppose now a failure occurs, then R1 and R5 will not be recomputed because they were fully persisted. However, R2, R3, and R4 will be recomputed. After recomputation, the order of computation will be R1, R5, R2, R3, and R4. This is different than the original order of R1–R5. The overall computation result must be the same with both orders, requiring that each region is associative with respect to all others. A

Table I: Comparing Eager vs. Lazy persistency

Aspect	Eager Persistency	Lazy Persistency
CL Flushes	Needed	–
Durable barriers	Needed	–
Logging	Needed	–
Error detection	Checking Log's state	Software checksum
Write amplification	High (from logging and flushes)	Low (from checksum)
Execution time overheads	High (extra instructions and pipeline stalls)	Low (checksum computation)
Recovery overheads	Low (persistency region)	High (checksum validation and recomputation)

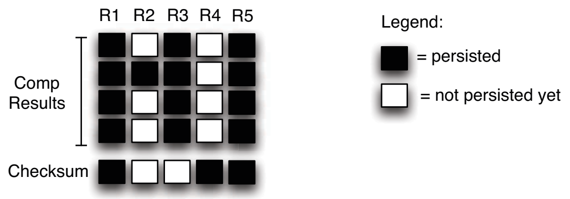

Figure 6: Illustrating the progress of Lazy Persistency regions.

necessary (but not sufficient) requirement for associativity is that there cannot be data dependences between regions, be they true or false (anti or output) dependences. However, the scope of Lazy Persistency can be expanded further. For example, false dependencies can be removed through code transformation. Furthermore, some false dependences may be permitted, depending on the recovery code complexity (Section IV shows an example).

With loop-based computations, loop bodies are typically good candidates for LP regions. It is often easy to find loop iterations that are associative and have no loop-carried dependences. With loop bodies, the granularity of parallelization and persistency regions may coincide. However, there are typically multiple loops that are associative, hence multiple choices of granularities for the LP regions.

The granularity of an LP region must be chosen carefully while weighing the overhead and reliability of the checksum calculation. Consider that each region has a corresponding checksum calculation. The cost of a single checksum is small, but it is incurred for all regions. Hence, smaller regions may aggregate a larger total cost for checksum computation than larger regions. Granularity also affects recovery after a failure. Larger regions imply more lost work that must be recomputed when the checksum does not match. In the extreme, a very large region may require computation time that approaches the mean time to failure (MTTF), making it hard to guarantee forward progress. At the same time, the checksum for a large region protects more data. For larger regions, the likelihood of false-negatives increases. To compensate, larger regions may need stronger error detection mechanisms that are costlier. A choice of good LP region granularity takes into account all of the above considerations.

D. Error Detection

There are several aspects to consider for error detection. One aspect is whether the checksum computation itself should adopt Eager or Lazy Persistency. Another important aspect is what error detection code should be employed. Let us first visit the persistency approach for the checksum. At each LP region, a checksum must be calculated. After it is calculated, with an Eager approach, it can be persisted immediately, e.g. using cache line flush and store fence. An advantage of this approach is that the checksum is always durable. However, one disadvantage is that we will be paying some of the costs of Eager Persistency, including cache line flushes, durable barriers, and write amplification.

If we choose Lazy Persistency for the checksum, the checksum calculation itself is not incurring any Eager Persistency costs. However, a false negative situation where computation in a region is persisted correctly but is misidentified as a persistency failure, is possible. Region R3 in Figure 6 illustrates this. With R3, the computation result is fully persisted, but the checksum is not persisted yet, so recovery will recompute this region unnecessarily. Considering that failures arise due to power failure or software crashes, the failure rate will be quite low, and therefore we choose the Lazy Persistency approach for the checksum, just as for computation results.

Another aspect of error detection design is what type of checksum to use. Choosing a checksum involves a trade-off between accuracy and execution time overhead. Accuracy refers to the likelihood that the checksum suffers a false-positive, the computation results were not fully persistent but the checksum matches anyway. While unlikely, it is possible for an error checking code to produce the same value for two different data sequences. Such cases must be exceedingly rare for Lazy Persistency to work properly. Hence, we need a low overhead technique that is also highly accurate. Fortunately, such inexpensive, memory efficient, and accurate techniques exist. We discuss a few of them.

1) **Parity Bit**: Each data in the region is XOR'd together. Our studies showed that Parity Bit incurs the least overhead but worse detection accuracy.
2) **Modular Checksum**: This is similar to Parity Bit, but it relies on summation rather than XOR'ing all the elements in the Region.

3) **Adler-32** [10], [11]: This checksum is used in a popular compression library [12].

We ran experiments with several scientific kernel workloads (Section V), injecting random errors to matrix elements. We tested if any of the injected errors yielded the same checksum value as error-free execution. We found that Modular Checksum and Adler-32 provide very good accuracy. The probability of failing to detect an error is less than 2×10^{-9} for both of them. Fewer than one failure out of two billion errors is quite acceptable, considering that the likelihood of errors is *decreasing* over time, instead of increasing. A traditional hard/soft error rate increases over time due to longer exposure to error-causing factors. In contrast, we are dealing with errors that are caused by data not being persisted yet. As time goes by, the data and checksum are more likely to be evicted from the cache hierarchy, hence the probability of error goes down. However, anyone concerned with false negatives can employ a stronger checksum or even employ multiple checksums simultaneously.

In terms of execution time overheads, Adler-32 is significantly more expensive than Modular Checksum. Therefore, we select the Modular Checksum as our default error detection technique, but we also evaluate Modular and Parity checksums applied in parallel for greater reliability.

Another important design aspect is the checksum organization. Checksums can be embedded into the data structure or stored in a standalone structure. Figure 7 illustrates both approaches. Consider a tiled matrix multiplication where the output is an $N \times N$ matrix. Figure 7(a) shows few columns added to the matrix. Assuming that each LP region is a kk loop body, there are $\frac{N}{bsize}$ kk regions per thread, where $bsize$ is the blocking factor, with each region and each thread needing their own checksum. Thus, with P threads, the number of checksum columns would be $\frac{N}{bsize} \times P$. With N rows, the total space overhead is high, at $\frac{N^2P}{bsize}$. Furthermore, since the data structure is directly modified, it results in high programming complexity, and complicates some compiler optimizations that deal with matrix layout. Thus, we adopt an alternative approach where the checksum is stored in a standalone structure, a hash table (Figure 7(b)). Programmers can call the cksum(key, val1, val2, ...) function, supplying a unique key that identifies the LP region being hashed, and all values to be protected by the checksum. For tiled matrix multiplication, the key is a combination of ii, kk, and thread ID. The hash function and hash table size are adjustable depending on the space target and tolerance for hash collisions. In our case, the table size is chosen to be $\frac{N}{bsize} \times \frac{N}{bsize} \times P = \frac{N^2P}{bsize^2}$, and ii, kk, and thread ID form the key. Our design eliminates hash collisions. Since each checksum is 32 bits wide, the total space overhead for the hash table is 1% of the size of the matrices.

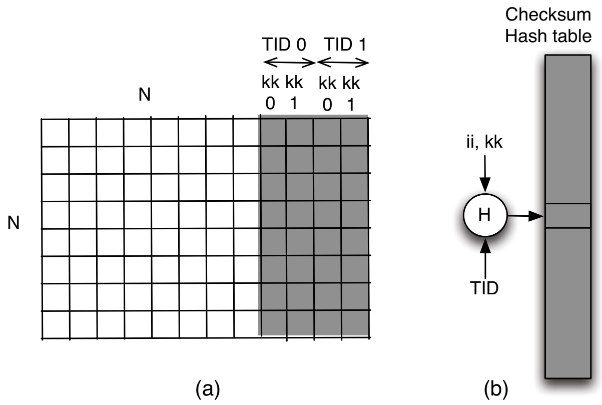

Figure 7: Checksum organization embedded into the data structure (a) vs. in a standalone structure (b).

E. Recovery

After a failure and upon detection of a mismatch between a checksum and a region, a recovery action is needed to restore the region's data to a consistent state. Recovery is code dependent, but there are special cases which make its construction trivial. One special case is when an LP region is idempotent. An idempotent region [13] is a code region that can be executed multiple times without changing the output of the program. Idempotent regions can be identified through compiler analysis [14]. If the regions coincide with LP regions, the recovery code can be trivially constructed since it is identical to the region code itself. For non-idempotent regions, recovery is program specific (Section IV discusses an example).

Two recovery styles are possible: Eager or Lazy. Lazy Persistency is possible for the recovery code. However, one concern is that persistency failure can occur during recovery, triggering recovery of the recovery code. Thus, we choose Eager Persistency for the recovery code, to ensure forward progress. Even though Eager Persistency is expensive, we only apply it for the recovery, which is the rare case.

1) Periodic Flushing to Limit the Recovery Time: It is possible that a dirty block containing either a result or a checksum from a very old region stays in the cache for a long time, especially if the cache size is very large. This can possibly make the recovery time unbounded with Lazy Persistency. To provide an upperbound on recovery time, we can employ a periodic cache cleanup; where after N regions are executed or after T amount of time elapsed, all dirty blocks are written back (but not evicted). The cache cleanup can be controlled by software (through executing a cache cleanup instruction), or performed automatically in hardware. With the latter, the hardware cache cleanup logic can space out write backs to avoid bursty writeback traffic, similar to how DRAM-refreshes to different rows are spaced apart in time. More elaborate hardware schemes are possible.

We evaluate this technique in Section VI-A.

IV. Matrix Multiplication Example

In this section, we will describe Lazy Persistency in more details using a specific example of tiled matrix multiplication. Figure 8 shows tiled matrix multiplication code from Figure 4, modified for Lazy Persistency. The LP region is an *ii* iteration. A larger or smaller region granularity is possible. However, the largest granularity (*kk* iteration) is too large; failure may trigger nearly the entire matrix multiplication to be repeated. Smaller granularities (*jj*, *i*, or *j* iterations) are also possible, but we choose a larger region granularity to keep the checksum overheads low.

```
1  for (kk=starting_kk; kk<n; kk+=bsize) {
2      for ( ii = starting_ii ;  ii <n; ii +=bsize) {
3          ResetCheckSum();
4          for  ( jj =0;  jj <n; jj +=bsize) {
5              for  ( i=ii ;  i<(ii+bsize);  i++) {
6                  for ( j=jj ;  j<(jj+bsize);  j++) {
7                      sum = c[i ][ j ];
8                      for (k=kk; k<(kk+bsize); k++)
9                          sum += a[i][k]*b[k][ j ];
10                     c[ i ][ j ]  = sum;
11                     UpdateCheckSum(c[i][j]);
12                 }  // end of  for  j
13             }  // end of  for  i
14         }  // end of  for  jj
15         hashIndex = GetHashIndex(ii,kk);
16         HashTable[hashIndex] = GetCheckSum();
17     }  // end of  for  ii
18 }  // end of  for  kk
```

Figure 8: Tiled matrix multiplication code with Lazy Persistency.

As shown in the figure, supporting Lazy Persistency can be achieved by adding only a few lines of code to the original code. The additional code includes resetting the checksum when we enter a new LP region, i.e. a new *ii* tile (Line 3). Then, the local checksum is updated to include the newly generated data in $c[i][j]$. If the checksum uses modular (or parity) checksum, the update is as simple as adding (or XORing) the new value $c[i][j]$ with the running checksum. After finishing all the operations for the current LP region, and before exiting the *ii* loop, the final running checksum value is stored into the hash table (line 15-16). Due to the simplicity of code transformation, Lazy Persistency can be added through a compiler pass.

The code in Figure 8 shows a sequential mode with one thread. For a parallel mode with multiple threads, the only change needed is to declare the checksum variable as thread private. For the hash table, since a key includes the threadID, different threads will access different parts of the hash table, hence no critical section (and locks) are needed to protect the hash table entries. If a smaller hash table is used where threads may collide on a single hash table entry, locks will be needed to protect hash table entries. Considering that collision-free hash table design only occupies 1% the space of the matrices, while not requiring locks, we chose it.

When recovering from a failure, the system needs to run the recovery code shown in Figure 9. Note that for matrix multiplication, LP regions within a *kk* are associative, but across different *kk*'s are not due to output dependences. A straightforward implementation requires Lazy Persistency to be applied only within one *kk* iteration. However, in the figure, we show how to relax the associativity requirement by designing recovery code that systematically loops over all the checksums in *reverse* program order, indexed by the *kk* and *ii*. Reverse order is necessary because each *kk* modifies the C matrix, hence there are output dependences across *kk* iterations.

```
1  for  (kk=last_kk; kk>=0; kk-=bsize) {
2      // decide the correct  kk loop
3      for (good_ii=0;good_ii<N;good_ii+=bsize){
4          if (IsMatchingChecksum(good_ii,kk)){
5              for  ( ii =0;  ii <n; ii +=bsize) {
6                  if (! IsMatchingChecksum(ii,kk)){
7                      Repair( ii ,kk);
8                  }
9              }  // end of  for  ii
10             // start  from the beginning of  next kk loop
11             starting_kk=kk+bsize;
12              starting_ii =0;
13             goto NormalExecutionMode;
14         }// end of  if  statement
15     }// end of  for  good_ii
16 }  // end of  for  kk
```

Figure 9: Recovery code for tiled matrix multiplication with this.

Once we find a computed checksum that matches the stored checksum (line 4), it indicates that at least one of the blocks within this *kk* tile persisted successfully prior to the failure, hence repair is triggered (line 7) for all *ii* at the given *kk*.

To accomplish the repair, we take advantage of the fact that we can recompute the tile using the input matrices. We zero out all intermediate values that were stored in the inconsistent tile(s) and fill them with the computation result from matrix multiplication for this tile, from the beginning until the *kk* preceding the one that starts normal execution.

Note, we can also optimize the *Repair* function to make recovery quicker. Instead of assuming that we must recover from the beginning, we can look for a prior *kk* iteration for the same *ii* block that does match its checksum. If one exists, we can recompute the difference rather than recomputing from the beginning.

Another case we need to handle is when the checksum is never assigned an initial value during the normal execution phase, i.e. because the program did not reach the corresponding iteration before the failure. Detecting the case can be achieved by initializing each checksum to an invalid value, such as NaN (Not-A-Number), or a value that programmers know that matrix values will not use, such as "-1" for matrices that only have positive numbers and zeroes.

V. Methodology

A. Simulation Configuration

For evaluation, we rely on a simulator that is built on top of the *gem5* simulator [15]. *Gem5* is an open source simulator which provides cycle-accurate full system simulations. Table II shows configuration parameters used in the evaluation. We use the x86-64 instruction set architecture (ISA). Our simulator models a processor with 2–17 cores. Each core has a 4-way out-of-order pipeline. It builds a memory hierarchy model on top of Ruby [16]. Each core has a private L1 cache, and an L2 cache shared by all cores. While cache size tends to go up with number of cores in a balanced system, we keep it fixed so that we can isolate the effect of increasing the number of threads independently from the total L2 cache size. Coherence among these caches is maintained through a MESI two-level coherence protocol. The default access latencies for the NVMM are 150ns for read and 300ns for write. In addition, in Section VI-C, we will investigate varying the NVMM latencies.

Table II: Machine configuration used in evaluation.

Component	Configuration
Processor	2-17 cores (default 9), each OoO, 2GHz, 4-wide issue/retire ROB: 196, fetchQ/issueQ/LSQ: 48/48/48
L1I and L1D	64KB, 8-way, 64B block, 2 cycles
L2	512KB, 8-way, 64B block, 11 cycles
MC	ReadQ/WriteQ: 32/64, ADR
NVMM	Latencies: 60-150ns read (default 150ns), 150-300ns write (default 300ns)

To support Eager Persistency, we implement the *clflushopt* instruction from Intel PMEM consistent with Intel's manual [17].

B. Real Machine Configuration

In addition to the evaluation on gem5 simulator, we evaluate Lazy Persistency on a real system (shown in Table III). That Lazy Persistency does not require any hardware support made it possible for us to evaluate it on any real systems, in contrast to Eager Persistency techniques. The system is DRAM-based since no NVMM-based systems are available commercially yet. Thus, we only measured execution time overheads and ignored the persistency aspects.

Table III: Real system used in evaluation.

Component	Configuration
Processor	32 CPUs, AMD Opteron(TM) Processor 6272, 2.099 GHz
L1I and L1D	16KB and 64KB, respectively. 64B block
L2	2048KB, 64B block
L3	6144KB, 64B block
Main Memory	32GB DRAM

C. Benchmarks

For evaluation, we compare several schemes shown in Table IV. We focus on tiled matrix multiplication for all of our evaluation, but we also evaluate other benchmarks using a subset of the schemes and configurations. These kernels are heavily used in HPC and machine learning. For example, the convolution layer of deep neural network contributes to about 90% of the execution time [18], [19]. In the table, *base (tmm)* represents a standard tiled matrix multiplication without failure safety. *tmm+LP* represents our proposed Lazy Persistency

Table IV: Various approaches tested for tiled matrix multiplication. The matrix dimension used is 1024×1024, and the tile size is 16, which allows one stride to be persisted using only one *clflushopt*.

Variant	Description
base (tmm)	Tiled matrix multiplication without failure safety
tmm+LP	Lazy Persistency (ii granularity)
tmm+EP	Eager Persistency (ii granularity)
tmm+WAL	Transaction with Logging (ii granularity)

tmm+EP represents the state-of-the art Eager Persistency scheme called *Recompute* [8]. EagerRecompute is an application level in-place checkpointing that was shown to perform substantially better in terms of performance and write amplification for scientific applications, when compared to other well-known schemes such as Checkpoint and Restart (C/R) and write-ahead logging with PMEM [8]. With EagerRecompute, programs are allowed to be in an inconsistent state during a transaction. A transaction covers a single tile in a tiled matrix multiplication. It persists computation as it goes in a semi-consistent manner. There is no guarantee of precisely consistent state at any given time during execution. When failure occurs, computation is rolled back to the last known state, any state for which its consistency status is unknown is discarded, and recomputation is triggered. EagerRecompute reduces the execution time by relaxing the order of persists within a transaction. EagerRecompute relies on Eager Persistency by forcing the program to wait after finishing each tile until all data modified in the transaction is persistent.

tmm+WAL represents a failure-safe version achieved using durable transactions with write-ahead logging implemented using PMEM instructions. For fairness, in all of *tmm+EP*, *tmm+WAL*, and *tmm+LP*, we choose the same persistency region granularity, which is a single *ii* loop iteration. Unless otherwise indicated, by default we run our experiments on nine cores, with each benchmark running with eight threads plus one master thread.

In addition to the Tiled Matrix Multiplication (TMM), we evaluated several other benchmarks, including Cholesky Decomposition (Cholesky), 2-dimensional convolution (2D-conv), Gaussian elimination (Gauss), and Fast Fourier transform (FFT). Table V describes these benchmarks and the input used. These benchmarks are popular kernels in scientific computation. Our implementation of these benchmarks is based on the SPLASH-2 suite [20], in addition to other resources such as [21], [22], [9].

Table V: Summary of the benchmarks we evaluated.

Benchmark	Description
TMM	1k-square input matrix multiplication
Cholesky	1k-square input matrix cholesky factorization
2D-conv	1k-square input matrix 2D convolution
Gauss	4k-square input matrix gauss elimination
FFT	100k nodes vector FFT

For the real system evaluation, all workloads were run from start to completion. For simulation-based evaluation, we simulate over a fixed number of outer-loop iterations to ensure that each of our designs performs the same amount of work during simulation. For Tiled Matrix Multiplication, our simulation window was two iterations over the outer-loop (*kk*). This is equal to $\frac{1}{32}$ of the run-time of the program. For Cholesky, the simulation time was feasible so we ran the kernel until completion, performing the 1k-square matrix. In 2D-Convolution, each of the schemes ran for 5 iterations of the outer loop, which is about 4% of the running-time of the program. The simulation window for the Gauss benchmark is 4 iterations of the outer loop, which will pass over 4 columns. Finally, simulation window for the FFT benchmark is about 5% of the running-time of the program. We selected these parameters to ensure that we simulate and report the timing for 300 million instructions, on average. Note, we also warm-up the simulator for 250 million instructions, on average, in advance of these simulation windows.

VI. Evaluation

We implement and compare various failure safety techniques described in Section V, against Lazy Persistency. Figure 10 shows the execution time and number of writes (write amplification), normalized to the base tiled matrix multiplication that is not failure safe. The number of writes includes the number of L2 writebacks due to regular cache evictions, plus any cache line flushes.

Scheme	Exe Time	Num Writes
base (tmm)	1.00	1.00
tmm+LP	1.002	1.003
tmm+EP	1.12	1.36
tmm+WAL	5.97	3.83

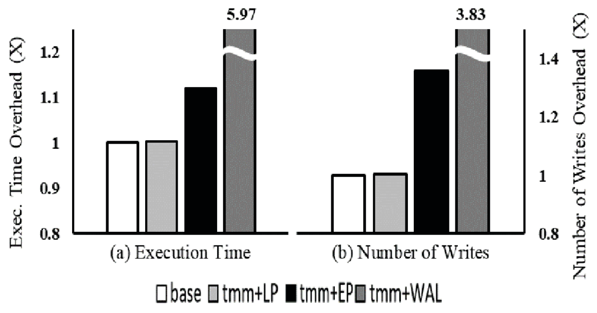

Figure 10: Execution Time and Number of Writes Comparison

As can be seen in the figure, using write-ahead logging (tmm+WAL) is the most expensive option, with execution time and write amplification of $5.97\times$ and $3.83\times$, respectively. The state-of-the-art Eager Persistency technique, EagerRecompute [8], yield a considerable execution time and write amplification reduction compared to tmm+WAL. EagerRecompute's execution time and write amplification are $1.12\times$ and $1.36\times$, respectively. However, these overheads are still high, especially the write amplification. Our new approach, Lazy Persistency, gives superior results, bringing both the execution time and write amplification overheads to below 1%, more specifically $1.002\times$ and $1.003\times$, respectively. The reason is that Lazy Persistency does not suffer from any cache line flushes, durable barriers, or logging. The code is changed very little, with only the checksum calculation being added to the normal execution.

Note that tmm+WAL is an Eager Persistency scheme that is relatively more straightforward for the programmer to work on, but it incurs the highest performance and write amplification overheads. EagerRecompute is an Eager Persistency scheme that requires significant changes to the source code and requires recovery code, which incurs similar programming complexity compared to our Lazy Persistency. Thus, the small execution time and write amplification overheads achieved by Lazy Persistency only trades off longer recovery time in comparison to EagerRecompute.

To dig deeper, we measure the number of occurrences of instructions that cannot be issued due to a structural hazard. We consider the following hazards: MSHRs full, integer functional units full (FUI), and load and store queue full (FUR and FUW, respectively). Finally, we also collect L2 miss rate (L2MR). These results are shown in Table VI. In the table, in the first three numeric columns, all numbers are normalized to base (tmm). The next numeric column (FUW) contains very small numbers in some schemes so they are not normalized. The final numeric column (L2MR) is also not normalized for obvious reasons.

Table VI: Various instances of pipeline hazards and L2 miss rate for different schemes.

	Normalized to base			Non-normalized	
Scheme	MSHR	FUI	FUR	FUW	L2MR
base (tmm)	1.00	1.00	1.00	1	0.01
tmm+EP	1.84	21.57	22.4	31,109	0.05
tmm+LP	0.95	1.11	1.2	2	0.02

As shown in the table, *tmm+EP* incurs significantly more structural hazards than base. This is primarily due to the back-pressure in the pipeline that resulted from pipeline stalls due to waiting for cache line flushes and store fences to complete. For Lazy Persistency, there are much fewer instances of structural hazards. Finally, the L2 cache miss rate is also higher with Eager Persistency due to cache line flushes.

Since Lazy Persistency allows caches to lazily evict blocks to NVMM as opposed to flushing them as in Eager Persistency, blocks may stay in the cache for a longer time. This higher cache residency time exposes a block to losing its value if a failure occurs, which will trigger recomputation on recovery. To measure such impact, we define *volatility duration* as the period, in clock cycles, between the time a block becomes dirty in the LLC until it is evicted and written back to NVMM. We measured the maximum value of volatility duration (maxvdur) attained by Lazy Persistency vs. EagerRecompute. To save space, we do not show it in a figure. The maxvdur for EagerRecompute is 20% of the base system, indicating that eager flushing significantly shortens the volatility duration of cache blocks. In contrast, Lazy Persistency's maxvdur is 101% of base, because it relies on natural cache evictions just like base.

A. Limiting the Recovery Time

As discussed in Section III-E, the maximum recovery time for Lazy Persistency may be arbitrarily large for a large cache, because the maximum volatility duration may increase with the cache size. To handle this issue, we add a simple hardware support that periodically cleans all dirty blocks in the caches. The cleanups are spaced in space (across cache sets) and in time, and happen in the background. Thus, the performance impact of periodic flushes is negligible. However, they incur write amplification, the degree of which depends on the frequency of flushes.

Figure 11 shows the relationship between the time between flushes and the number of writes to NVMM for Tiled Matrix Multiplication, normalized to the base case. As expected, the figure shows that increasing the time between flushes reduces the number of writes. The figure gives

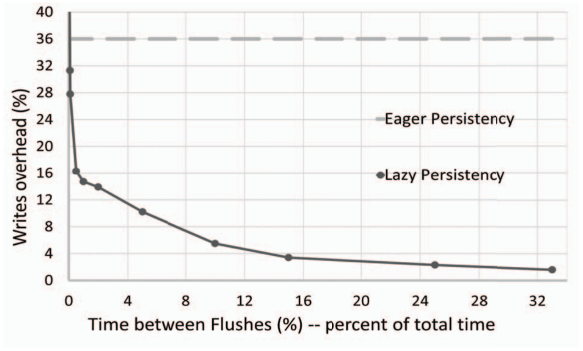

Figure 11: Number of additional writes (vs. base tmm) as affected by the time between flushes as a fraction of the total execution time, for Eager vs. Lazy Persistency.

two additional insights. First, even with a tiny 0.08% time between flushes, the write overhead of 32% is still smaller than that of EagerRecompute (36%). Second, extending the time between flushes rapidly reduces write overheads to negligible amounts, for example less than 2% write overhead with 33% execution time between flushes.

B. Other benchmarks

In addition to tiled matrix multiplication, we run other benchmarks with EagerRecompute representing the state-of-the-art Eager Persistency, and compare them against Lazy Persistency. Figure 12 and 13 show the execution time and write amplification overheads, respectively. All the numbers are normalized to unmodified versions of the benchmarks that are not failure safe.

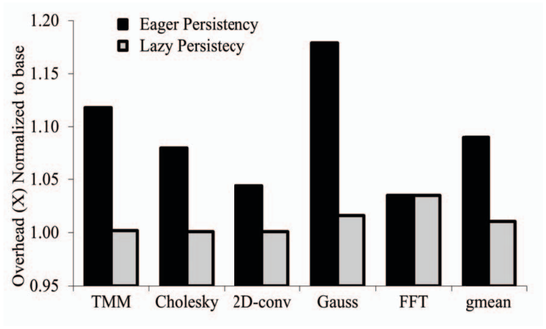

Figure 12: Normalized execution time for all benchmarks.

Figure 12 shows that Lazy Persistency incurs execution time overheads ranging from 0.1% to 3.5% (averaging 1.1%), compared to a range from 4.4% to 17.9% (averaging 9%) for EagerRecompute. Similarly, Figure 13 shows that Lazy Persistency incurs write amplification overheads ranging from 0.1% to 4.4% (averaging 3%). On the other hand, EagerRecompute incurs write amplification overheads ranging from 0.2% to 55% (averaging 20.6%). Both figures

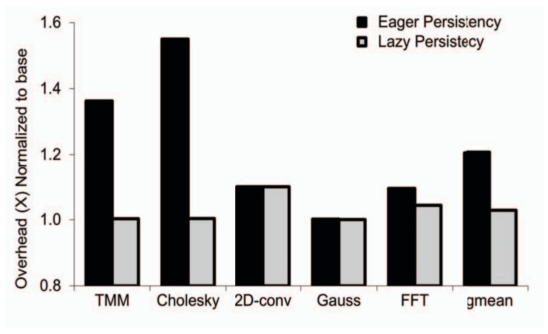

Figure 13: Normalized write amplification ratios for all benchmarks.

show that Lazy Persistency achieve superior execution time overheads and write amplifications compared to Eager-Recompute.

Write amplification overheads differ from one workload to another significantly, due to several factors. One factor is the duration between writes to the same block. For a workload that stores frequently to the same block, either to the same byte or different bytes in a block, EagerRecompute forbids coalescing the stores in the cache, due to explicitly flushing a block repeatedly after each store. On the other hand, Lazy Persistency relies on natural cache eviction, allowing stores to coalesce in the cache. Hence, applications with high temporal and spatial store locality suffers a higher write amplification overhead with EagerRecompute. A second important factor is the total memory footprint of the workload. A workload with a relatively small memory footprint will make the extra writes caused by writing the checksums in Lazy Persistency become more significant (relative to the total number of writes), which shows up as a larger write amplification overhead. This factor also explains why the write amplification overheads of Lazy Persistency is 55% less than EagerRecompute in some workloads, but negligible in other workloads (e.g. Gauss). However, the overall trend shows that Lazy Persistency achieves about 17.6% less write amplification overheads than EagerRecompute.

Evaluation on real hardware. We repeat the experiments shown in Figure 12 on a real hardware system described in Table III. Table VII reports the execution time overhead for Lazy Persistency. Apart from small variations, the overall magnitude of the execution time overheads is consistent between the two experiments.

Table VII: Execution time overhead (%) for Lazy Persistency on a real system, normalized to the non-persistent base case.

TMM	Cholesky	2D-conv	Gauss	FFT	gmean
0.8%	1.1%	0.9%	2.1%	1.1%	1.1%

C. Sensitivity Study

In this section, we report several results from varying the configuration parameters in order to measure the sensitivity to Lazy Persistency's performance. First, we vary the NVMM read and write latencies for both Lazy Persistency and EagerRecompute. Figure 14(a) shows the results for three different sets of latencies expressed in the following format: (read latency, write latency), normalized to the base case for each respective set of latencies. As the latencies increase, we can clearly see the execution time overheads trending differently for EagerRecompute vs. Lazy Persistency. With EagerRecompute, it trends higher because cache line flushes are more expensive, cache misses are also more expensive, and durable barriers also take longer time to complete. In contrast, with Lazy Persistency, the relative contribution of checksum computation on execution time decreases. Figure 14(b) shows the execution time for Lazy Persistency and base (tmm) when the number of threads is varied from 1 to 16. All numbers are normalized to base running with 1 thread. As illustrated in the figure, Lazy Persistency achieves similar scalability with base (tmm) as the number of threads increases.

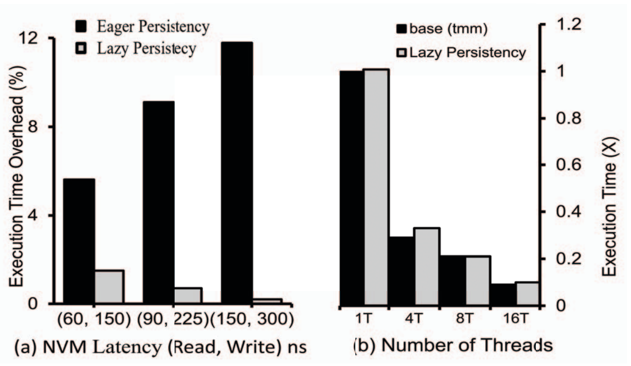

Figure 14: Sensitivity of execution time overheads of Lazy Persistency when NVMM latencies are varied (a), and the execution time when the number of threads varies from 1 to 16 (b), normalized to the base (tmm) with 1 thread.

Figure 15(a) shows the effect of L2 cache size on the execution time overhead over base (tmm). As the cache size increases, the overheads of Lazy Persistency decreases: a 256KB L2 incurs an overhead of 6.5%, which decreases to 0.2% and 0.1% for a 512KB and a 1MB L2 cache, respectively. This is because for a small cache, the working set and the checksums overflow the cache, resulting in increased L2 miss ratios to above 4%, compared to only 2% and 1.5% with a 512KB and a 1MB L2 cache, respectively. The overhead difference between 512KB and 1MB is very small, indicating that the total working set and checksums fit in the L2 cache. In addition, Lazy Persistency is more effective with larger caches as more dirty blocks can stay in the cache longer. With a small cache, the effectiveness of Lazy Persistency is lower because dirty cache blocks will be evicted quickly due to limited capacity. Figure 15(b) shows the execution time overheads from using different checksums over base (tmm). Modular checksums and parity bits achieve the lowest overheads, 0.2% and 0.1% respectively. Adler-32 incurs roughly 1% execution time overhead. A parallel combination of modular checksum and parity achieve lower a false negative rate but incurs a higher 3.4% execution time overhead, which is still lower than the 12% overhead from Eager Persistency.

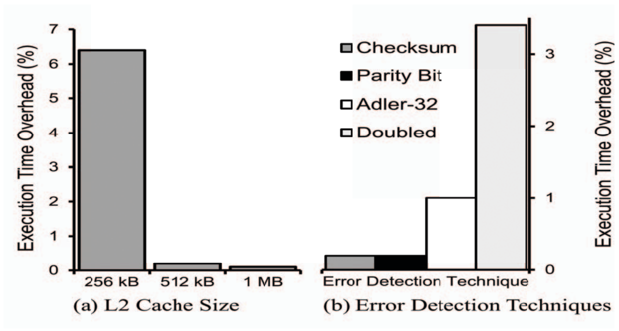

Figure 15: Sensitivity of execution time overheads of Lazy Persistency when the L2 cache is varied (a), and when different error detection techniques are used (b).

VII. Related Work

NVM has received significant research activities recently. Past work has examined various aspects of NVM, including memory organization (e.g., [23], [24]), security (e.g., [25], [26], [27], [28]), extending life time (e.g., [28], [29], [30], [26]), and persistency acceleration (e.g., [7], [31]). The above list is a small subset of examples of work in NVM research. From here on, we will expand on papers that are most immediately related to our work.

Persistency Models. Prior works have proposed various persistency models, including strict persistency [2], epoch persistency [3], buffered epoch persistency [3], [4], strand persistency [2], and transactional persistency [32], [33]. They specify when stores become durable, or in which order stores become durable. Transactional persistency defines an atomic durability boundary where all stores in a transaction either persist together or not at all, achieved through logging [31], [33], [34], [35]. Intel PMEM [5] is an example of persistency model for x86 system. PMEM provides several instructions, such as *clwb* and *clflushopt* in addition to existing x86 instructions such as *clflush* and *sfence*. All these persistency models force store values out of the caches early to the persistence domain which includes the NVMM. Thus, we refer to them as *Eager Persistency*. In contrast to those studies, Lazy Persistency does not force early writes to NVMM. Instead, it relies on the natural eviction of blocks from the cache hierarchy.

Using Meta-Data to Detect Failures. Relying on meta-data to detect an error is a general technique in fault tolerance. A related work to ours is application-based fault tolerance (ABFT) [36], [37], which relies on adding checksums to rows and columns of matrices to facilitate fault-tolerance without relying on checkpointing. ABFT [36], [37] does not deal with memory persistency and requires substantial changes in data structures. In contrast, Lazy Persistency deals with memory persistency and avoids changes to application data structures. Lu et al. [6] proposed adding meta-data to a group of cache blocks to remove persistence recording from the critical path of transaction commit. It requires complex hardware, whereas Lazy Persistency is a software technique, hence it is platform independent.

VIII. Conclusion

We proposed and discussed *Lazy Persistency*, a novel software persistency technique that achieves data persistency while relying on normal cache eviction mechanisms. Lazy Persistency code is free of cache flushes and persist barriers. Instead, code is split into associative regions that are protected by checksums that can be used to detect persistency failure, and recovery code that can restore the region on a failure is added. We evaluated Lazy Persistency and compared it to the state-of-the-art eager persistency technique EagerRecompute [8] for several workloads. Our results show that Lazy Persistency reduces the execution time and write amplification overheads, from 9% and 21%, to only 1% and 3%, respectively. Lazy Persistency opens the door for new hardware mechanisms that support persistency. We evaluate one such technique that periodically flushes dirty data. While this increases the number of writes modestly, it puts an upper bound on the recovery work needed after a failure.

Acknowledgment

This work is supported in part by NCSU and the NSF (CNS-171748). Authors would like to thank Keiji Kimura for giving valuable feedback and participating in the development of ideas in the paper.

References

[1] Intel and Micron, “Intel and micron produce breakthrough memory technology,” 2015.

[2] S. Pelley, P. M. Chen, and T. F. Wenisch, “Memory Persistency,” in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2014.

[3] 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 Symposium on Operating Systems Principles (SOSP)*, 2009.

[4] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, “Efficient Persist Barriers for Multicores,” in *Proceedings of International Symposium on Microarchitecture (Micro)*, 2015.

[5] “Persistent memory programming,” 2016, http://pmem.io.

[6] Y. Lu, J. Shu, L. Sun, and O. Mutlu, “Loose-Ordering Consistency for Persistent Memory,” in *Proceedings of the International Conference on Computer Design (ICCD)*, 2014.

[7] 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, 2017.

[8] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, “Efficient Checkpointing of Loop-Based Codes for Non-Volatile Main Memory,” in *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[9] M. E. Wolf and M. S. Lam, “A Data Locality Optimizing Algorithm,” in *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[10] P. Deutsch and J.-L. Gailly, “Zlib compressed data format specification version 3.3,” 1996.

[11] M. Adler, “Adler-32 — Wikipedia, the free encyclopedia,” 2010, [Online; accessed 10/25/2017]. [Online]. Available: https://en.wikipedia.org/wiki/Adler-32

[12] “zlib compression library,” 2017, [Online; accessed 10/25/2017]. [Online]. Available: https://zlib.net/

[13] S. A. Mahlke, W. Y. Chen, W.-m. W. Hwu, B. R. Rau, and M. S. Schlansker, “Sentinel scheduling for vliw and superscalar processors,” in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS V, 1992.

[14] M. A. de Kruijf, K. Sankaralingam, and S. Jha, “Static analysis and compiler design for idempotent processing,” in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[15] N. Binkert and et al., “The GEM5 simulator,” *ACM SIGARCH Computer Architecture News (CAN)*, 2011.

[16] “Ruby memory system,” 2016, http://gem5.org/Ruby.

[17] I. Corp., “Intel 64 and IA-32 Architectures Developer’s Manual: Vol. 3A,” 2016.

[18] K. Osawa, A. Sekiya, H. Naganuma, and R. Yokota, “Accelerating matrix multiplication in deep learning by using low-rank approximation,” in *2017 International Conference on High Performance Computing Simulation (HPCS)*, 2017.

[19] Y. Jia, “Learning semantic image representations at a large scale,” Ph.D. dissertation, 2014.

[20] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 1995.

[21] S. C. Woo, J. P. Singh, and J. L. Hennessy, “The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[22] S. Ahn, “Convolution,” 2005, http://www.songho.ca/dsp/convolution/convolution.html.

[23] A. Awad, S. Blagodurov, and Y. Solihin, “Write-aware management of nvm-based memory extensions,” in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS ’16, 2016.

[24] M. Saxena and M. M. Swift, “Flashvm: Virtual memory management on flash,” in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC’10, 2010.

[25] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, “Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’16, 2016.

[26] M. K. Qureshi, “Pay-as-you-go: Low-overhead hard-error correction for phase change memories,” in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[27] A. Awad, Y. Wang, D. Shands, and Y. Solihin, “ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[28] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, “Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling,” in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[29] J. Chen, G. Venkataramani, and H. H. Huang, “Repram: Re-cycling pram faulty blocks for extended lifetime,” in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 2012.

[30] J. Chen, Z. Winter, G. Venkataramani, and H. H. Huang, “rpram: Exploring redundancy techniques to improve lifetime of pcm-based main memory,” in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[31] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, “Proteus: A flexible and fast software supported hardware logging approach for nvm,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[32] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, “High-performance Transactions for Persistent Memories,” in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[33] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight Persistent Memory,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[34] 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,” in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[35] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, “Atom: Atomic durability in non-volatile memory through hardware logging,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[36] K.-H. Huang and J. A. Abraham, “Algorithm-based fault tolerance for matrix operations,” *IEEE Trans. Comput.*, 1984.

[37] Z. Chen and J. Dongarra, “Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources,” in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, ser. IPDPS’06, 2006.