Accelerating Erasure Coding on Persistent Memory via Adaptive Prefetcher Scheduling

Guanglei Xu
Huazhong University of Science and Technology
Wuhan, China
grayxu@hust.edu.cn

Hai Zhou
Huazhong University of Science and Technology
Wuhan, China
haizhou@hust.edu.cn

Yuchong Hu
Huazhong University of Science and Technology
Wuhan, China
yuchonghu@hust.edu.cn

Dan Feng
Huazhong University of Science and Technology
Wuhan, China
dfeng@hust.edu.cn

Renzhi Xiao
Huazhong University of Science and Technology
Wuhan, China
renzhixiaohust@gmail.com

Abstract

Compared to DRAM, persistent memory (PM) offers higher density and persistence but encounters more severe reliability challenges. Erasure coding is widely adopted to enhance reliability with minimal space overhead. Unfortunately, applying erasure coding to PM introduces significant additional latency. Previous work to mitigate coding latency has primarily focused on optimizing computational efficiency. Instead, we reveal that the main performance bottleneck is high memory latency due to inefficient hardware prefetchers, rather than computation. We further observe that the prefetching inefficiency mainly results from: (i) too wide or narrow coding stripes, (ii) small block sizes, and (iii) high concurrency.

Based on these observations, we propose Dialga, an adaptive hardware/software prefetching scheduler aware of PM encoding characteristics. Dialga improves memory access efficiency and thereby enhances encoding performance. It first profiles the cache status and access patterns to adaptively switch prefetching strategies. Dialga then employs a lightweight operator to achieve fine-grained and low-overhead scheduling for both hardware and software prefetchers. Additionally, Dialga further optimizes the PM read buffer efficiency by leveraging the implicit data loading mechanism for prefetching. Compared with state-of-the-art erasure coding libraries, Dialga achieves up to 96.6% higher encoding throughput and up to 178.8% improvement in multi-thread scalability.

CCS Concepts

• **Computer systems organization** → **Redundancy**; Processors and memory architectures; • **Hardware** → *Non-volatile memory*.

Keywords

Erasure Coding, Persistent Memory, Prefetching

ACM Reference Format:
Guanglei Xu, Hai Zhou, Yuchong Hu, Dan Feng, and Renzhi Xiao. 2025. Accelerating Erasure Coding on Persistent Memory via Adaptive Prefetcher Scheduling. In *54th International Conference on Parallel Processing (ICPP '25), September 08–11, 2025, San Diego, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3754598.3754614

1 Introduction

Persistent memory (PM) is an emerging memory technology that provides memory-like performance and disk-like capacity with persistence [27, 30, 33]. System applications can access byte-addressable PM by `load/store` instructions, reducing the development effort required for migrating existing memory systems. Many studies [20, 33] focus on optimizations for migrating from DRAM to PM by targeting the unique characteristics of PM, such as high memory access latency and memory traffic contention.

Although PM provides new opportunities for memory system design, it has lower reliability compared to DRAM. In modern production systems, memory errors have been extensively studied and found to potentially cause serious problems. [15]. Errors in PM can originate from various sources, such as hardware-level bit flips, write disturbances, and system-level errors [26, 34]. These errors are difficult to detect and correct using only device-level redundancy.

To enhance the reliability of PM, erasure coding is used as system redundancy to provide the capability to repair erroneous data on PM [29, 34]. Reed-Solomon (RS) code [21] is a widely adopted erasure code that encodes k data blocks to generate m redundant parity blocks, forming a stripe. This enables the recovery of up to m lost data blocks. Unfortunately, existing studies [13, 28] show that the additional computation and `load/store` operations introduced by using erasure coding on PM to maintain reliability redundancy double the latency. Previous methods [18, 25, 35] have been proposed to accelerate encoding on DRAM, including optimizing the encoding matrix and reusing intermediate results to reduce computational overhead. But these methods focus primarily on improving computational efficiency. On high latency PM, the performance bottleneck may shift from computation to memory access.

Unlike prior work focusing on computation, we identify the primary performance bottleneck for erasure coding on PM as *high memory access latency due to inefficient hardware prefetchers*. The main process of encoding can be divided into multiple sets of memory load and computation operations. High-latency synchronous memory loads on PM can stall subsequent computation instructions, degrading encoding performance. We further observe three main causes of hardware prefetching inefficiency: (i) too wide or narrow encoding stripes, which can overload or underutilize the hardware prefetcher; (ii) small encoding block sizes, which lead to inaccurate prefetches; (iii) high concurrency, which causes PM read

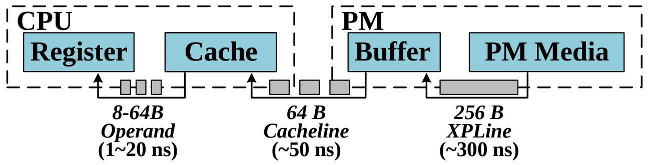


Figure 1: Data load granularity across different layers.

buffer thrashing and read amplification. Because of the limited ability of the hardware prefetchers, they cannot prefetch data blocks on PM timely and accurately to hide memory access latency in these short prefetching windows or under high concurrency pressure. What's worse, useless prefetching from hardware prefetchers increases memory traffic contention and can even lead to severe read amplification due to PM read buffer thrashing.

However, enabling efficient and flexible prefetching for erasure coding on PM remains challenging: *(i) Inflexible pattern tracking.* Hardware prefetchers are fixed and struggle to adapt to the memory access patterns of coding on PM, especially under varying coding parameters and block sizes [12, 31]. *(ii) Prefetch control overhead.* Hardware prefetchers lack lightweight and fine-grained control mechanisms [7, 22]. Meanwhile, introducing software prefetching may incur execution and branch misprediction overheads. *(iii) Inefficient buffer utilization.* Excessive prefetching not only fails to effectively utilize the PM read buffer to reduce latency, but even leading to read amplification due to buffer thrashing.

To this end, we propose DIALGA, an adaptive hardware/software prefetching scheduler that is aware of the characteristics of PM encoding, to improve memory access efficiency, thereby enhancing encoding performance. First, DIALGA's adaptive coordinator collects memory access patterns and samples hardware events to assess the status of CPU caches and hardware prefetchers, thus adaptively switching between different prefetching strategies and thresholds. Second, DIALGA's lightweight operator employs static shuffle mapping to achieve fine-grained control over hardware prefetchers, and embeds branchless pipelined software prefetch instructions. Additionally, it further employs a PM read buffer-friendly prefetching scheme to take advantage of implicit data loads. To the best of our knowledge, DIALGA is the *first* work to mitigate the memory access bottleneck of erasure coding on PM.

In summary, this paper makes the following contributions:

- We analyze the behavior of hardware prefetchers during encoding on PM and observe that the performance bottleneck is high memory access latency caused by inefficient hardware prefetchers. (§3)
- We design DIALGA, an adaptive hardware/software prefetcher scheduler that is aware of the characteristics of PM encoding. It utilizes both an adaptive coordinator and a lightweight operator for low-overhead prefetcher scheduling, while leveraging the PM read buffer, thus enhancing memory access efficiency for PM encoding. (§4)
- We implement DIALGA[1] within the widely-deployed acceleration library, ISA-L [1]. Results show that DIALGA significantly improves encoding throughput and multi-thread scalability compared to state-of-the-art libraries. (§5)

[1]The source code is available at https://github.com/Dialga-ICPP25.

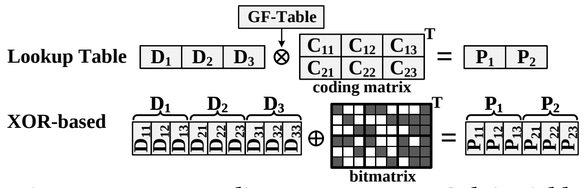


Figure 2: Erasure coding process over a Galois Field.

2 Background

2.1 Persistent Memory

Persistent memory (PM) provides larger density than DRAM [27, 30]. Data-intensive memory systems can be migrated to PM with the same `load/store` interface. The PM media (e.g. PCM) is slower than DRAM, leading to higher read latency for PM. Many studies [30, 36] aim to optimize systems by overcoming PM characteristics, including 2x higher load latency and memory traffic contention. Although the first commercially available PM device, Intel Optane DCPMM, has been discontinued, industrial research in PM continues, such as Samsung CXL-based CMM-H [24].

The CPU and PM operate at mismatched access granularities. PM can be accessed via 64 B memory interfaces like DDR-T or CXL, because PM devices integrate an on-chip buffer to bridge the speed and access granularity gap between internal persistent media and DRAM. Consequently, data loading granularity varies across layers, as shown in Figure 1. Registers of different sizes process operands at 8–64 B granularities. The CPU cache retrieves at a 64 B cacheline granularity, while the PM buffer accesses PM media at a 256 B (XPLine) granularity. Granularity increases at lower layers. We refer to the load amplification caused by this granularity mismatch *implicit data loads*. For example, a single 8B read request may trigger a larger 256B read from the underlying PM media[16]. Samsung CMM-H also exhibits similar characteristics due to its internal DRAM and flash [24].

Recent research [26, 29, 34] indicates that the reliability of PM is lower than that of DRAM. Uncorrectable memory errors may lead to serious errors in production [15]. First, the PM media may occur random errors, such as bit flips or write disturbances, some of which are beyond the capabilities of on-chip ECC. Second, errors from levels above hardware, like system kernel errors or software scrubbing, can lead to incorrect write requests to PM media. Even with ECC modules, existing PM products [11] only achieve a low, disk-level Mean Time Between Failures (MTBF).

2.2 Erasure Coding

For fault-tolerant memory systems, erasure coding can enhance system reliability with low space overhead [8]. Reed-Solomon codes [21] are the most commonly used erasure codes. RS($k + m, k$) encodes k data blocks to generate m parity blocks, forming a stripe of $k + m$ blocks. This allows for decoding any k blocks to recover from up to m erroneous data blocks. Due to additional computation and `load/store` operations, erasure coding inevitably introduces extra performance overhead.

Many studies focus on reducing the overhead of erasure coding. The encoding computation in RS codes is performed in the Galois Field (GF) of size 2^w, which is not natively supported by CPUs. Figure 2 shows two different implementations of GF encoding. The lookup table approach pre-computes the results of GF multiplication and retrieves them via table lookups. For example, ISA-L [1] is a

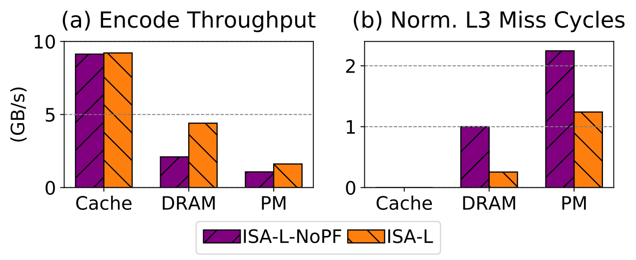


Figure 3: Encoding performance of RS(12, 8) with different load sources.

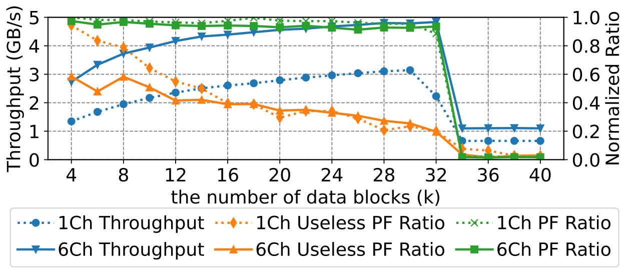


Figure 5: The impact of different stripe sizes for encoding on PM ($m = 4$), including encode throughput, useless prefetch ratio, and L2 prefetch ratio.

Single Instruction Multiple Data (SIMD) accelerated library widely used in both research and production systems [3, 8]. ISA-L only needs to load each data block once during encoding. In contrast, XOR-based approach converts GF multiplication into multiple XOR computation groups. Encoding one parity block requires several different groups of XOR operations. Researchers optimize the encoding bitmatrix to reduce memory accesses and computations [17]. Zerasure [35] systematically considers various factors impacting encoding performance and adopts heuristic optimization procedures to integrate multiple optimization strategies. Cerasure [18] further employs a greedy optimization approach for encoding bitmatrix search and decomposes wide stripe encoding. However, XOR-based approach requires repeatedly reading data blocks from different locations, leading to a larger memory footprint.

Existing research [18, 35] on erasure coding acceleration mainly focuses on computational efficiency, but memory access efficiency constitutes a more prominent bottleneck for PM with higher latency [27, 30]. Studies [13, 28, 29, 34] have shown that coding on PM introduces significant extra overhead, exceeding 50 %. The overhead stems from the computation and data updates required to maintain parity redundancy upon writes or updates. This makes optimizing the performance of erasure coding on PM an key problem.

3 Observations

3.1 Bottleneck Analysis

To investigate the performance bottlenecks related to high-latency PM, we designed an encoding test and used `Perf` [6] to sample PMU events for profiling. This benchmark encodes random 1 KB stripes using ISA-L with RS(12, 8) to evaluate the impact of different data load sources on encoding throughput and CPU L3 cache miss cycles. The encoding throughput refers to the rate of processing data blocks. We leverage AVX512 for load, store, and compute

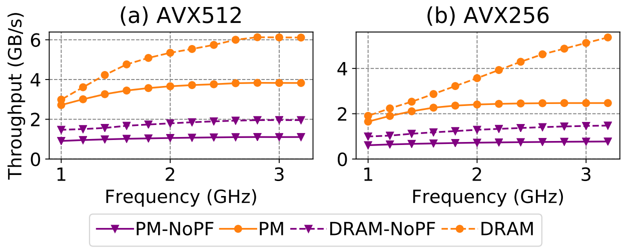


Figure 4: Encoding performance of RS(12, 8) with different CPU frequencies.

operations, where all writes to PM are non-temporal and a final memory fence is applied. Hardware configuration is shown in §5.1.

#Observation 1. Memory access efficiency is key for PM. Figure 3 shows that sourcing encoding data from DRAM achieves 195–272% higher throughput compared to sourcing from high-latency PM. With the hardware prefetcher disabled, the change in throughput corresponds proportionally to the change in L3 cache miss cycles. With the hardware prefetcher enabled, the reduction in L3 cache miss cycles is greater for DRAM than for PM, resulting in a 109% throughput increase for DRAM, compared to a modest 50% increase for PM. This result illustrates: (i) memory access efficiency is the key performance bottleneck, rather than computation. (ii) while the hardware prefetcher can effectively enhance DRAM encoding performance, its efficiency on PM is smaller.

#Observation 2. Computation efficiency is relatively less important for PM. Figure 4 shows the encoding performance with different CPU frequencies. On PM, we observed minimal additional performance gains beyond 2 GHz, whereas modern server CPU frequency can exceed 3 GHz. It indicates that this portion of CPU cycles is wasted for waiting memory loading. Conversely, DRAM with lower latency maintains a more consistent throughput improvement curve, better utilizing CPU cycles for encoding computations. This observed trend is more pronounced under half-width AVX256. Existing XOR-based erasure coding strategies [18, 35] introduce repeated `load/store` to the same memory address. This access pattern increases latency and amplify memory traffic. Therefore, we focus our analysis on ISA-L, which has a simpler memory access pattern. These XOR-based methods will be compared in §5.

3.2 Hardware Prefetcher Analysis

We further examine hardware prefetcher behavior during encoding across different scenarios, including varying encoding parameters. Unless otherwise specified, the default configuration uses a 4 KB block size with hardware prefetchers enabled and AVX512.

#Observation 3. The number of data blocks influences the prefetching window and affects efficiency, while excessive numbers will disable hardware prefetchers. The number of data blocks per stripe differs across erasure coding systems, with Facebook's f4 using 12 and VAST [8] using 154. Figure 5 shows the changes in encoding throughput, useless hardware prefetch ratio (`PMU 0xf2`), and total L2 prefetch ratio as k increases. We divide the results into three stages: *(i)* $k < 16$*:* The throughput initially starts low but increases as k grows. A smaller k reduces the prefetch window, making effective prefetching more challenging. During this stage, the hardware prefetcher is less efficient, leading to higher traffic. *(ii)* $16 < k < 32$*:* The throughput moderately increases, with

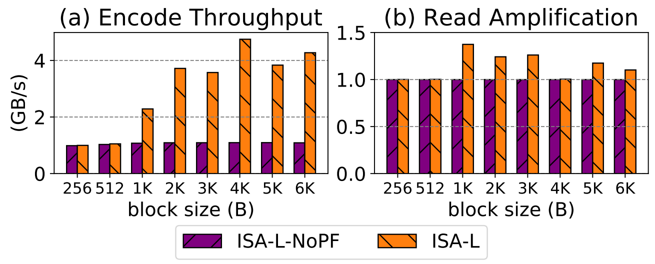


Figure 6: RS(28, 24) encoding throughput and read amplification on PM with different block sizes.

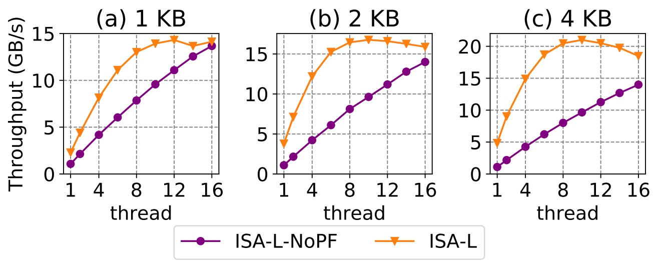


Figure 7: Multi-thread scalability of RS(28, 24) encoding on PM.

a gradual reduction in useless hardware prefetch count. During this stage, the hardware prefetcher operates more efficiently because of a larger prefetch window, although some useless prefetches remain. *(iii)* $k > 32$*:* The throughput is extremely low, because the number of data blocks surpasses the L2 stream prefetcher's tracking capability. While Intel [5] only states a tracking capability of 16 bidirectional streams, we show that the prefetcher can track up to 32 unidirectional streams. We believe exceeding this threshold overwhelms this stream prefetcher, reducing its prefetching confidence. Consequently, its aggressive prefetching activity for 4 KB pages drops to near zero, leading to a sharp throughput decline. Similar behavior is also observed across different Intel CPU generations. We find that beginning with the Intel 3rd Gen Xeon Scalable, the stream prefetcher can track up to 64 unidirectional streams. However, this capacity remains insufficient for wide stripe encoding [8].

#Observation 4. Small encoding block sizes lead to inaccurate hardware prefetching. The object sizes can vary, ranging from several hundred bytes to a few KBs [31]. Figure 6 shows the encoding throughput and media read traffic amplification for different block sizes, tested with RS(28, 24) as a hardware prefetcher friendly stripe size. Hardware prefetcher significantly improves encoding throughput for block sizes larger than 1 KB, but it has no effect on smaller blocks. The shorter stream lengths for each block impede effective hardware prefetching. This reduces the prefetcher's confidence and eventually stops prefetching, which also explains the absence of read amplification for small blocks. When the block size is between 1–3 KB, the hardware prefetcher improves performance but leads to 23–37% read amplification due to its aggressive prefetching. Since the hardware prefetcher does not prefetch across 4 KB pages, 4 KB blocks incur no read amplification and thus represent the most effective block size. When the block size exceeds 4 KB or is not 4 KB aligned, mixed patterns emerge, like moderate acceleration and read amplification on 5 KB.

#Observation 5. Hardware prefetching under high concurrency causes PM read buffer thrashing, leading to read amplification and limited scalability. We further explore the multi-thread scalability of RS(28, 24) encoding on PM. Figure 7 shows that throughput plateaus, and may even decrease, when the number of threads reaches 8 to 10. In contrast, disabling the hardware prefetcher results in linear scalability, although throughput decreases due to the loss of latency hiding provided by prefetching. We found that the rapid bottleneck in multi-threading is caused by the aggressive prefetch behavior of uncontrolled hardware prefetchers. Each prefetch retrieves a 64 B cacheline to the CPU cache. Correspondingly, on the PM device, a 256 B XPLine containing this cacheline is fetched to its on-chip read buffer to accelerate subsequent accesses. However, under high concurrency, the data prefetched to the read buffer is more likely to be evicted by other prefetch requests before being accessed. This results in significant PM media read bandwidth being wasted on transfers from PM media to the read buffer. Such PM read buffer thrashing not only prevents the read buffer from hiding load latency, but also amplifies PM media read traffic and limits multi-thread scalability.

3.3 Challenges

Our observations indicate that inefficient hardware prefetching is the primary performance bottleneck for erasure coding on PM, causing high memory access latency, especially with wide stripes, small blocks, and high concurrency. It is non-trivial to improve prefetching efficiency due to the following challenges: *(i) Inflexible pattern tracking.* Encoding parameters, access patterns in production are varying [12, 31], which makes it challenging for fixed hardware prefetcher strategies to track the access patterns of coding on PM. *(ii) Prefetch control overhead.* Hardware prefetchers function as black boxes, lacking lightweight and fine-grained control interfaces [7, 22]. Meanwhile, introducing software prefetching incurs additional overhead and potential branch misprediction. *(iii) Inefficient buffer utilization.* Despite the on-chip PM read buffer, unordered or excessive prefetch requests fail to utilize it effectively and may even cause severe read amplification due to buffer thrashing.

4 Design

We propose DIALGA to accelerate erasure coding on PM by adaptively scheduling hardware/software prefetchers, which coordinates prefetching based on cache events and I/O patterns, executes strategies via a lightweight operator, and leverages a PM read buffer-friendly scheme. Figure 8 shows the architecture of DIALGA.

4.1 Adaptive Coordinator

4.1.1 Challenge. Production systems have different characteristics of coding tasks [12, 31]. For example, a system may need to encode data with different parameters based on different fault tolerance requirements. In addition, system pressure may fluctuate with concurrency. Prefetcher scheduling strategies need to comprehensively evaluate factors to enable adaptive adjustments for coding on PM.

4.1.2 Design. DIALGA builds a coordinator to identify patterns of different coding tasks to design and dispatch prefetcher scheduling strategies to improve memory access efficiency. First, to identify cache pressure status, DIALGA utilizes a lightweight sampling

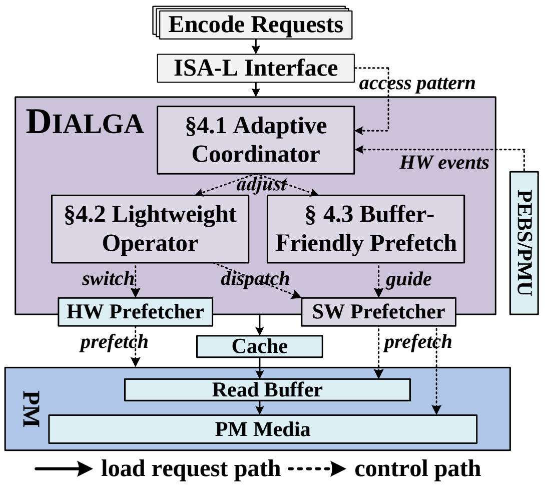


Figure 8: Overview of the Dialga architecture. Purple boxes denote Dialga components, blue boxes denote hardware components. Solid arrows indicate load request paths, dashed arrows indicate control paths.

method to read hardware counters, including PEBS and PMU, for collecting memory and cache events. Second, Dialga collects access requests via the ISA-L library interface to identify and track application-level access patterns. Finally, based on the collected data, Dialga dynamically adjusts the scheduling policy for hardware and software prefetchers.

Cache Events. Dialga's coordinator samples memory access latency via PMU hardware counters, using 110% of the average latency under low pressure as a threshold to determine if read traffic contention occurs. The threshold setting referenced previous work [33]. Since useless prefetching by the hardware prefetcher exacerbates read traffic contention, we also collect the number of useless L2 hardware prefetches. We quantify the hardware prefetcher efficiency by calculating the difference in the number of useless prefetches. A threshold of 150% is set to determine if the current hardware prefetcher is operating inefficiently. If significant traffic contention and inefficient hardware prefetcher are both detected (e.g., under high concurrency pressure), Dialga disables the hardware prefetcher to alleviate contention. The counter sampling frequency is set to 1 kHz to maintain low overhead [32].

I/O Access Pattern. Dialga's coordinator collects stripe size, block size and the number of concurrent threads via the ISA-L library interface as metrics of the I/O access pattern. When encoding blocks larger than 4 KB, Dialga keeps the hardware prefetcher enabled for the 4 KB aligned portions. For non-4 KB aligned portions or small blocks, Dialga determines whether to enable the hardware prefetcher based on cache events. For wide-stripe coding, there is no need to manage hardware prefetchers, because their hardware limitations prevent them from tracking too many streams, causing them to quickly lose confidence in prefetching and shut down automatically. The threshold for the number of concurrent threads in Dialga is set to 12 based on the observations. When the number of concurrent threads exceeds a threshold, Dialga attempts to disable the hardware prefetcher to prevent contention and buffer thrashing caused by excessive prefetching. Although hardware prefetchers perform inefficiently with narrow stripe and small blocks, they are not directly disabled because the amplified traffic under low pressure does not degrade performance.

Pipelined software prefetch. To address the limitations of hardware prefetcher strategies, Dialga's coordinator further employs a pipelined software prefetcher. Dialga inserts asynchronous software prefetch instructions during ISA-L encoding tasks to hide memory access latency and improve encoding performance. By adaptively determining an appropriate prefetch distance d, the pipelined software prefetcher prefetches the $N + d$-th cacheline while accessing the N-th cacheline. The trade-off in selecting the prefetch distance is that a shorter distance may prevent data from being loaded into the cache before the associated load, whereas a longer distance may increase CPU cache and PM buffer usage, leading to the prefetched data being evicted before it is needed. Regardless of whether hardware prefetching is enabled, Dialga attempts software prefetching, which can trade increased read traffic for improved performance under low pressure.

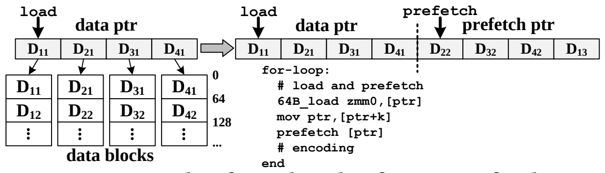
Dialga employs hill climbing [23] to determine the software prefetch distance d. It initiates this search upon startup or when the encoding performance fluctuates by more than 10%. The search begins by setting $d = k$. To reduce search time, Dialga uses the latency of 128B sub-tasks as the optimization target, as the benefits of longer prefetch distances may manifest in the subsequent cacheline row within the stripe. It then iteratively explores a neighborhood of size 16 around the current distance to find a local optimum.

In conclusion, Dialga employs a threshold-based heuristic to switch the hardware prefetcher and hill climbing to select the software prefetch distance. Instead of dynamic instrumentation, Dialga statically extends ISA-L's existing `ec_encode_data(len, k, m, gf_table, data, parity)` assembly entry points (originally for different instruction sets). The coordinator selects a corresponding entry point according to the policy, where each entry point corresponds to a distinct strategy, while the prefetch distance is adjusted via parameters. Given that accessing PM is the primary bottleneck, the I-cache impact of larger static code is acceptable.

Other Coding Tasks. Dialga is also applicable for other erasure coding tasks, including decoding and Locally Repairable Code (LRC) [10]. In ISA-L, encoding and decoding tasks share the same memory load pattern: reading k data blocks, while decoding reads k correct blocks. A (k, m, l) LRC code builds on RS codes by dividing data blocks into l groups and adding a local XOR parity to each group. LRC encoding still requires reading k data blocks. As a result, the bottleneck also stems from slow memory loads. Therefore, Dialga's adaptive prefetching can also be applied to improve their performance. We discuss the generality for other PM devices in §6.

4.2 lightweight Operator

4.2.1 Challenge. While Dialga's coordinator optimizes prefetcher scheduling strategies, mitigating the scheduling overhead remains challenging: (i) Hardware prefetchers lack mechanisms for fine-grained and lightweight switching. Prior approaches only permit switching at the machine level via BIOS [27] or at the core level using `msr-tools` which requires privileged kernel mode with mode switch overhead [5]. (ii) Implementing schedulable software prefetching interfaces within compact assembly libraries may introduce additional overhead, such as branch misprediction penalties.


load
data ptr
D_{11} D_{21} D_{31} D_{41}
D_{11} D_{21} D_{31} D_{41}
D_{12} D_{22} D_{32} D_{42}
0
64
128
...
data blocks
load
data ptr
prefetch
prefetch ptr
D_{11} D_{21} D_{31} D_{41} D_{22} D_{32} D_{42} D_{13}

```
for-loop:
  # load and prefetch
  64B_load zmm0,[ptr]
  mov ptr,[ptr+k]
  prefetch [ptr]
  # encoding
end
```

Figure 9: An example of pipelined software prefetching in DIALGA when $k = 4$ and the prefetch distance $d = 5$.

4.2.2 Fine-Grained Hardware Prefetcher. DIALGA implements a lightweight, fine-grained hardware prefetcher switch interface, enabling low-overhead, function-level switching of hardware prefetchers. Inspired by studies on reverse engineering of CPU hardware prefetchers [7, 22], we observed that hardware prefetchers are triggered by specific access patterns. We find that the majority of hardware prefetch requests stem from the L2 stream hardware prefetcher, which is activated by detecting patterns such as sequential memory accesses or accesses crossing 4 KB boundaries. Each accurate prefetch increases the confidence of the L2 stream hardware prefetcher, thus enhancing its aggressiveness. Therefore, DIALGA fools the L2 stream prefetcher by *shuffling* consecutive memory access instructions, which reduces prefetch confidence and decreases the number of hardware prefetches. We carefully designed a static shuffle mapping to reorder encoding tasks at the 64 B cacheline granularity, thereby avoiding identified patterns. Conversely, deactivating this shuffle mapping re-enables the hardware prefetcher by rebuilding its confidence. In evaluation, applying vectorized offsets to data pointers incurred only marginal overhead.

4.2.3 Branchless Pipelined Software Prefetcher. DIALGA efficiently embeds a low-overhead, branchless software prefetching interface within compact assembly functions by reorganizing task pointers with vectorization. To mitigate overheads introduced by the software prefetcher scheduling interface, such as potential branch mispredictions and increased register usage, DIALGA performs external, vectorized pre-processing in advance. As shown in Figure 9, this pre-process creates a prefetch pointer array of length k alongside the data pointers used for encoding tasks. When accessing data for encoding, the address to be prefetched can be derived by applying the fixed offset k to the data pointer. If the prefetch distance d is not a multiple of k, the vectorized construction is performed in two groups with different offsets to ensure the correctness of the prefetch pointer. By constructing prefetch pointers with specified distances through external vectorization, additional logic branches and register usage are avoided. To prevent extra read traffic, we revert to the standard encoding interface for tail-end tasks. Moreover, externally constructed prefetch pointers retain their order even after shuffling, ensuring compatibility with fine-grained hardware prefetcher switching strategies.

4.3 PM Read Buffer-Friendly Prefetch

4.3.1 Challenge. Unordered or excessive prefetching fails to effectively utilize the PM read buffer for latency reduction and may cause read buffer thrashing under high concurrency, leading to severe read amplification and limited scalability.

4.3.2 PM Read Buffer Implicit Loads. The on-chip read buffer of PM exhibits implicit loads due to the granularity mismatch. Specifically, a 64 B data loading will implicitly load the associated 256 B (XPLine size) into the read buffer. This results in varying latency for loads depending on whether they hit the read buffer. If the access interval between different 64 B cachelines within an XPLine is greater than the implicit load time, only the latency of the first load reflects the true PM media latency, while the others reflect the buffer latency.

DIALGA further specifies distinct prefetch distances based on different memory load latencies. The prefetch distance should align the load latency to achieve a trade-off between cache footprint and timeliness. Therefore, non-uniform load latency reduces the efficiency of uniform prefetch distance. Although the hardware prefetch distance cannot be modified, we can further optimize pipelined software prefetching. Instead of uniformly applying the same prefetch distance to all cachelines, we increase the prefetch distance for the first cacheline in each accessed XPLine while reducing distance for the remaining cachelines. But this strategy is applied only under low pressure because the increased number of simultaneously accessed XPLines leads to higher PM read buffer usage. The initial prefetch distance for the first cacheline of each XPLine is set to $k + 4$, and is then adjusted by the adaptive coordinator.

4.3.3 Reduce Read Amplification. The primary reason why excessive hardware prefetching leads to read buffer thrashing and amplification is the mismatch between the size of the read buffer and the working set size at high concurrency. Data loaded into the read buffer is prematurely evicted due to capacity without being utilized, resulting in wasted PM media bandwidth. Thus, to improve the efficiency of the on-chip read buffer under high pressure, we should limit the memory access ranges, including prefetching.

To this end, DIALGA expands the granularity of a single assembly encoding loop task to access the associated data in the XPLine earlier. The task granularity in the assembly loop is extended to XPLine by reusing existing general-purpose and AVX512 registers. This expansion ensures that each iteration achieves a 256 B read and compute granularity, increasing the likelihood that data implicitly prefetched to the PM read buffer is loaded rather than prematurely evicted. Note that increasing the granularity of a single loop process does not adversely affect cache efficiency. In ISA-L's memory access pattern, each data is read once for computation without repetition. However, expanding granularity reduces baseline encoding performance. This decline occurs due to decreased efficiency in the implicit prefetching of the PM on-chip read buffer. After each 64 B load, the read buffer loads its associated 256 B XPline. Processing larger granularity increases read latency. Therefore, we adopt this strategy only under high pressure. Because the PM read buffer size is fixed, we can estimate the utilization rate of the read buffer based on the configurations, ensuring the max prefetch distance satisfies:

$$nthread \times k \times 256B \times \left\lceil \frac{max(distance)}{k+m} \right\rceil \leq buffersize, \qquad (1)$$

where $m = 0$ for non-temporal stores. For example, on our 6 channel system with a total 96 KB read buffer, thrashing occurs when the number of threads exceeds 12 under hardware prefetching enabled.

5 Evaluation

5.1 Methodology

Hardware Configurations. Our evaluation testbed is a Linux 4.15.0 server with Intel Xeon Gold 6240 @ 3.30 GHz with 18 cores.

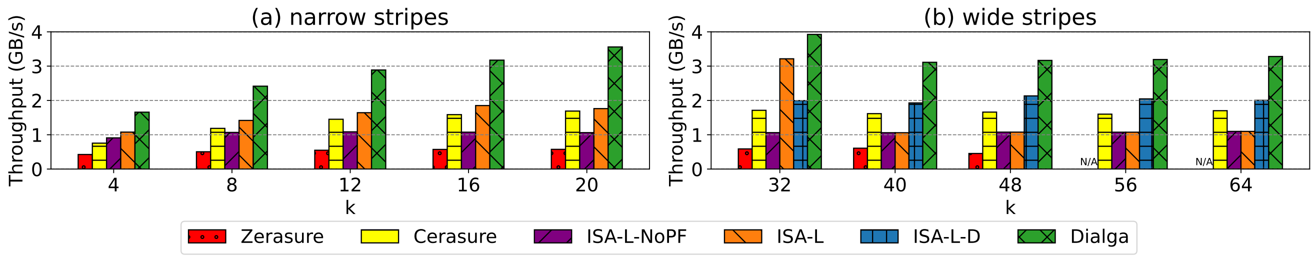


Figure 10: Encoding throughput on with different numbers of data blocks.

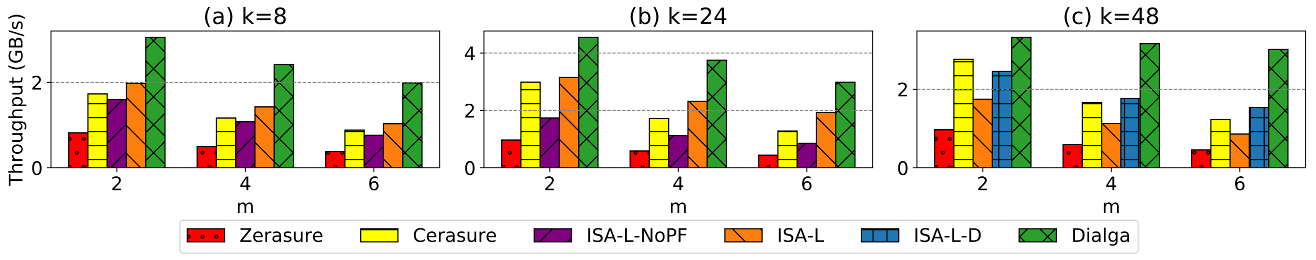


Figure 11: Encoding throughput with different numbers of parity blocks.

Gold 6240 has a 32 KB L1 data cache, a 1 MB L2 cache, and the LLC is 24.75 MB. Each CPU is equipped with 6 memory channels, each having 16 GB of DDR4 2666 MHz DRAM and 128 GB of Intel Optane DCPMM 100 series.

Compared systems. We select three state-of-the-art libraries as our baselines: (1) **Zerasure** [35] is an XOR-based encoding library that accelerates encoding through optimized algorithms combined with various techniques, such as bitmatrix normalization and scheduling optimization. (2) **Cerasure** [18] further designs XOR encoding matrices with fewer computations and optimizes cache efficiency during the encoding process. We report Cerasure's best performance, as its decomposing did not consistently improve PM coding performance in our tests. Note that Zerasure and Cerasure only support AVX256. (3) **ISA-L** [1] is a widely deployed erasure coding library, configured here with non-temporal stores. **ISA-L-D** enhances ISA-L via decomposing (using the same size as Cerasure) to improve wide stripe encoding. **Dialga** is built upon ISA-L, which internally uses multiple, automatically switched variant assembly entry points within the standard ISA-L encoding interface.

Methodology. Following the methodology of prior work on erasure coding acceleration [18, 25, 35], we focus on encoding throughput as the key performance metric. We analyze throughput across different coding parameters, workloads and hardware configurations. For the experiments, we measure the performance of $RS(k, m)$ random encoding in GF(2^8) on 1 GB of pre-filled data, averaging results across 10 runs. Unless otherwise specified, the default settings use $m = 4$ parity blocks and a 1 KB block size.

5.2 Encode Throughput Analysis

5.2.1 Impact of the number of data blocks. Figure 10 presents the encoding throughput of 1 KB blocks across various stripe sizes. For encoding with narrow stripes ($k < 20$) with 1 KB blocks, Zerasure exhibits the lowest performance due to its suboptimal encoding matrix. Cerasure shows improvement as k increases, but achieves lower throughput than ISA-L due to additional `load/store` operations and lower memory access efficiency. In contrast, Dialga employs pipelined software prefetching together with enabled hardware prefetchers to effectively mitigate memory access latency, even with a short prefetch window. Dialga achieves a 53.9–102.0% throughput improvement compared to the best of other strategies.

For wide stripe encoding ($k > 32$) with 1 KB blocks, Zerasure's encoding matrix search space is too large for its search algorithm to converge, resulting in some missing results. Meanwhile, ISA-L's performance declines sharply due to the limitations of the hardware prefetcher. The decompose strategy in Cerasure mitigates this by splitting the encoding into narrower stripes, reactivating the hardware prefetcher. But compared to ISA-L without prefetching, Cerasure only achieves a 48.9-53.2% improvement. ISA-L-D, employing the same decomposing strategy, achieved an 81.9–98.0% improvement, suggesting that a simpler memory access pattern yielded superior results. Dialga, even without decomposing, attained a 193.6–198.9% improvement through independent software prefetching strategy. At the threshold $k = 32$, as the hardware prefetcher reaches its peak efficiency, Dialga achieves only a 22.1% performance improvement over ISA-L.

5.2.2 Impact of the number of parity blocks. Figure 11 shows the impact of different numbers of parity blocks. When $m = 2$, the performance of Cerasure is close to ISA-L. As m increases, the complexity of the encoding increases. Unlike ISA-L, which relies on table lookups for computation, XOR-based libraries such as Cerasure exhibit computational complexity that grows both faster and non-linearly with increasing m. As a result, Cerasure experiences more pronounced performance degradation as m increases. On narrow stripes, all methods except Dialga exhibit limited performance regardless of the value of m, due to low hardware prefetching efficiency. Dialga outperforms other methods significantly under different parameters, achieving improvements of 20.1–96.6% over the best alternative strategies.

For wide stripes such as RS(52, 48), Dialga maintains a performance advantage with minimal degradation as m varies. This stability arises because a larger k on wide stripes increases the proportion of load operations, concentrating the bottleneck on memory

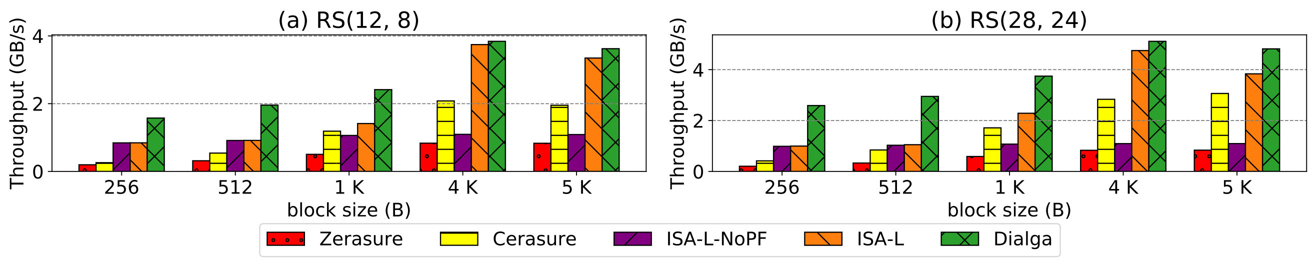


Figure 12: Encoding throughput with different block sizes.

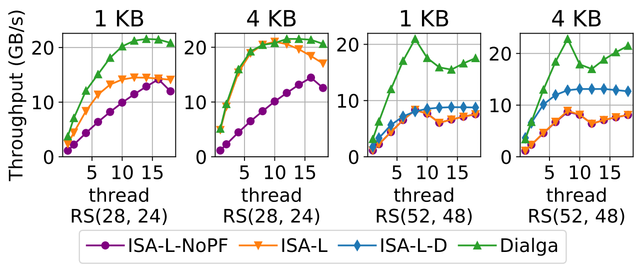


Figure 13: Multi-thread scalability of encoding throughput.

load, which DIALGA's effective software prefetching mitigates by improving memory access efficiency. In contrast, after applying the decompose strategy, both Cerasure and ISA-L-D face the problem of reduced memory access efficiency due to smaller stripes after decomposing, while also introducing amplified write traffic.

5.2.3 Impact of block size. Figure 12 shows the encoding throughput for different encoding block sizes. For small block sizes of 256 B and 512 B, where the hardware prefetcher has low confidence, enabling the hardware prefetcher yields no improvement for ISA-L. Zerasure and Cerasure suffer from exacerbated memory access inefficiency due to excessively small packet sizes. For 1 KB blocks, hardware prefetching improves ISA-L's performance by 33–112%. RS(28, 24) benefits from a longer prefetch window, leading to better performance. DIALGA achieves significant performance improvements for block sizes of 1 KB and smaller, outperforming the best alternative strategy by 63.82–180.5%. However, at 4 KB, the improvement from DIALGA is limited because the hardware prefetcher operates at peak efficiency. It aggressively prefetches all cachelines within each 4 KB blocks but does not prefetch across 4 KB boundaries, avoiding unnecessary bandwidth consumption. Similarly, at 5 KB, the improvement is limited to only 8.2–25.6% because 4 KB blocks still dominate the workload.

5.3 Multi-Thread Scalability Analysis

Figure 13 shows the multi-thread encoding scalability of DIALGA for different stripe sizes and block sizes. For RS(28, 24) 1 KB block encoding, unlike ISA-L which rapidly bottlenecks at 8 threads, DIALGA scales faster and bottlenecks later, achieving 50.0% higher peak performance. But for 4 KB blocks, DIALGA shows only marginal performance improvements, due to efficient hardware prefetching at 4 KB. This suggests that DIALGA's performance improvements stem from the hardware prefetcher's inefficiency. Only after ISA-L degrades due to excessive concurrency does DIALGA demonstrate up to a 21.0% improvement because of the stable performance maintained by adaptive coordination.

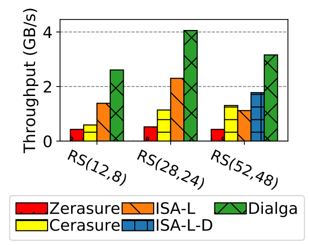


Figure 14: Decoding throughput with different stripe sizes.

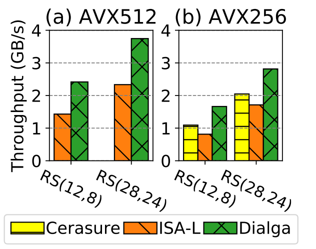


Figure 15: Encoding throughput with different SIMD instructions.

For RS(52, 48) wide stripe encoding, DIALGA shows significant improvements, achieving up to 182.8% better performance compared to ISA-L and up to 140.3% compared to the decompose strategy. This improvement arises because ISA-L faces limitations caused by the automatic disabling of the hardware prefetcher, whereas the decompose strategy is constrained by narrow stripes encoding and increased additional write traffic. Under wide stripes, performance degradation appears after 10 threads, even with the hardware prefetcher disabled. This is due to the 96 KB PM read buffer, which can sustain at most 8×48 access streams (load + prefetch). Consequently, when the number of thread exceeds 8, read buffer thrashing ensues, leading to performance degradation. Although DIALGA alleviates this issue by increasing the prefetch distance, it still reduces memory load efficiency.

5.4 Decode Throughput Analysis

Figure 14 presents the decoding throughput with different stripe sizes. The performance of XOR-based methods degrades significantly. This occurs because XOR-based methods are optimized for encoding matrix, while the complexity of the corresponding decoding matrix remains unconstrained. This is because XOR-based methods can only optimize the complexity of the encoding matrix. But decoding matrix is directly derived from the encoding matrix through a transformation and cannot be further optimized for complexity. Therefore, in contrast to the performance degradation observed with Zerasure and Cerasure, lookup table-based methods such as ISA-L and DIALGA maintain stable performance and thus show a more pronounced advantage. DIALGA achieves decoding throughput that is 142.1–340.7% higher than Cerasure, and 76.1–88.1% higher than ISA-L.

5.5 SIMD Instruction Analysis

Figure 15 presents encoding performance with two different SIMD instruction sets, AVX256 and AVX512. The operational width of

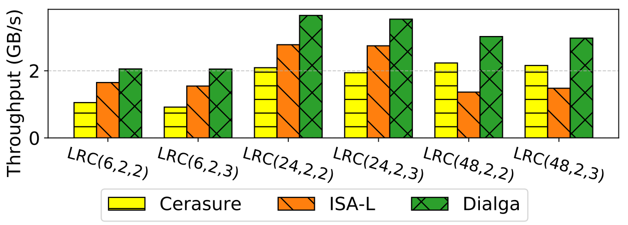


Figure 16: LRC encoding throughput with different parameters.

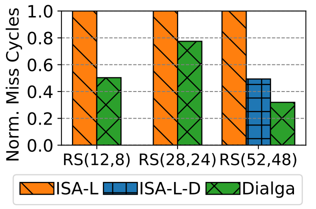


Figure 17: Cache miss cycles of different stripe sizes.

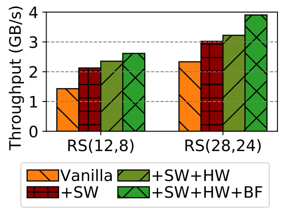


Figure 18: Breakdown of 1 KB encoding throughput.

AVX256 is half that of AVX512, which significantly reduces CPU computational efficiency. Although a smaller width affects memory access efficiency to some extent, the memory access bottleneck remains primarily tied to memory latency. After transitioning the instruction set to AVX256, varying degrees of performance degradation were observed. The baseline, ISA-L, exhibited only a moderate decline of 12.3–23.6%, as it is significantly memory latency-bound. Correspondingly, Dialga showed a greater deterioration of 24.9–31.1%. However, it still outperformed Cerasure and ISA-L with an improvement of 37.5–104.4%.

5.6 LRC Encode Throughput Analysis

Figure 16 shows LRC(k, m, l) encoding throughput with different parameters for 1 KB blocks. Because LRC codes require additional computation and store operations for extra local XOR parity blocks, all strategies experience varying degrees of throughput reduction compared to RS codes. Compared to the best results of other strategies, Dialga improves throughput by 24.3–32.7% with non-wide stripes, and by 35.2–37.8% with wide stripes. The higher proportion of store instructions in LRC codes results in less significant performance improvement for Dialga.

5.7 Cache Efficiency Analysis

Figure 17 presents the CPU cache miss cycles during encoding, normalized by the total number of loads. For RS(12, 8), ISA-L demonstrates approximately double the miss cycles compared to Dialga, consistent with the nearly 2x throughput improvement observed in prior experiments. This consistency supports our observation of the memory access bottleneck and indicates that Dialga's primary performance advantage arises from enhanced memory access efficiency, effectively reducing CPU stall cycles. For RS(28, 24), the hardware prefetcher exhibits relatively high efficiency, thereby limiting the reduction in cycles achieved by Dialga. For RS(52, 48), compared to the decomposition strategy, Dialga not only employs a more effective prefetching strategy but also eliminates the need for parity reloading, thereby achieving a greater reduction in cache miss cycles, amounting to 35.3%.

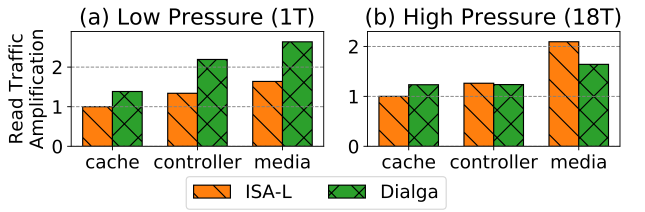


Figure 19: Read traffic on different layers of RS(28, 24) 1 KB encoding under various pressure conditions.

5.8 Breakdown Analysis

Figure 18 breaks down the performance of different Dialga variants to assess the contribution of individual optimizations. We examine four variants: `Vanilla`, which disables all optimizations and serves as a baseline; `+SW`, which adds pipelined software prefetching; `+HW`, which enables hardware prefetching; and `+BF`, which introduces buffer-friendly prefetching. Pipelined software prefetching improves performance by 29.4–48.6% by reducing memory access bottlenecks. Hardware prefetching further increases performance by 8.6–15.9% due to the low memory pressure of single-threaded encoding. Buffer-friendly prefetching provides an additional 18.3–29.3% performance boost with more precise prefetch timing. The improvement from buffer-friendly prefetching is less significant for narrow stripes, potentially because data block loads in narrow stripes have better spatial locality, limiting prefetching benefits.

5.9 Read Traffic Analysis

Figure 19 presents the memory read traffic observed for 1 KB block stripe encoding using `libipmctl` [2] at different levels: encoding, memory controller, and PM media. The traffic is normalized based on ISA-L encoding throughput. Figure 19(a) shows that in single-threaded low pressure tests, inaccurate hardware prefetching leads to cascading amplification of read traffic. Dialga introduces more software prefetch instructions, which lead to greater read traffic due to their training effect on the hardware prefetcher [7]. But this trade-off is reasonable in low pressure scenarios, as spare read bandwidth is still available. Figure 19(b) shows that under high pressure with 18 threads, ISA-L's read traffic amplification at the PM media layer increases significantly from 22.3% to 65.8%, due to PM read buffer thrashing caused by excessive prefetching. Under high pressure, this waste of read bandwidth becomes the most critical bottleneck limiting its multi-threaded scalability. In contrast, Dialga disables the hardware prefetcher to avoid amplification at the memory controller layer, increases task loop granularity to reduce amplification at the PM media layer, ultimately reducing amplification by 76.7% compared to ISA-L.

6 Generality

Because Dialga's optimization strategies target the general characteristics of PM, such as high access latency and internal hierarchies, it remains applicable to future PM devices, which also exhibit similar characteristics, including Samsung CMM-H[24] and Kioxia XL-FLASH[4]. From the perspective of storage hierarchy, PM media generally serves as a larger-capacity tier with higher access latency and requires a larger access granularity compared to DRAM. For example, Samsung CMM-H integrates a DRAM buffer to hide the large access granularity and latency of its underlying flash layer.

7 Related Works

Erasure Coding Acceleration. Accelerating erasure coding has consistently been a significant focus in systems research. One approach concentrates on exploiting computer architectural resources to enhance performance. This includes designing cache-friendly memory access patterns to decrease memory accesses [17] and leveraging SIMD instructions for vectorizing finite field computations [1, 19]. Another approach transforms finite field computations into XOR operations and optimizes them by reducing XOR operations. For example, some studies propose more efficient encoding matrices to reduce the number of XOR computations [9, 18], such as searching using simulated annealing [35] or optimizing with SLP techniques [25]. But unlike Dialga, none of them focus on accelerating erasure coding for high-latency memory devices.

Coding on PM. Previous studies [29, 34] have employed coded redundancy in systems to enhance PM reliability and have identified consequent performance challenges. To mitigate overhead, TVARAK [13] proposes employing an additional hardware controller to offload parity computation and writing. Vilamb [14] delays and amortizes the parity updates over multiple data writes by re-using the page table dirty bits. CodePM [28] proposes pipeline encoding and persistent writing, which hide latency through overlapping instruction execution, thereby improving the update performance of parity redundancy. However, TVARAK requires hardware modifications, complicating practical implementations, while Vilamb sacrifices reliability guarantees. Furthermore, CodePM's optimization is limited to updating tasks and does not address the load bottleneck issue. In contrast, Dialga can be integrated into existing erasure coding libraries to accelerate memory loads.

8 Conclusion

This paper first identifies that the performance bottleneck of erasure coding on PM is the high memory access latency caused by inefficient hardware prefetchers. To address this, we introduce Dialga to adaptively analyze and schedule hardware/software prefetchers for accelerating coding on PM. Results show that Dialga significantly improves the encoding performance on PM.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. 62272185, No. 61821003) and Basic Research Program for Young Students (Doctoral Students) of the National Natural Science Foundation of China (No. 623B2037). The corresponding authors are Yuchong Hu and Dan Feng.

References

[1] 2024. Intel(R) Intelligent Storage Acceleration Library. Intel Corporation.
[2] 2024. IPMCTL User Guide. https://docs.pmem.io/ipmctl-user-guide/
[3] 2024. ISA Erasure Code Plugin – Ceph Documentation. https://docs.ceph.com/en/latest/rados/operations/erasure-code-isa/
[4] 2024. XL-FLASH | Storage Class Memory (SCM) | KIOXIA. https://americas.kioxia.com/en-us/business/memory/xlflash.html
[5] Intel Corporation. 2018. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf
[6] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. In *Slides from Linux Kongress*.
[7] Guillaume Didier, Clémentine Maurice, Antoine Geimer, and Walid J. Ghandour. 2022. Characterizing Prefetchers Using CacheObserver. In *Proc. of SBAC-PAD*.
[8] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick P. C. Lee, Weichun Wang, and Wei Chen. 2021. Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage. In *Proc. of USENIX FAST*.
[9] Cheng Huang, Jin Li, and Minghua Chen. 2007. On Optimizing XOR-Based Codes for Fault-Tolerant Storage Applications. In *Proc. of IEEE ITW*.
[10] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure Coding in Windows Azure Storage. In *Proc. of USENIX ATC*.
[11] Intel. 2019. Intel Optane Persistent Memory 100 Series 512GB. https://ark.intel.com/content/www/us/en/ark/products/190351/intel-optane-dc-persistent-memory-512gb-module.html
[12] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, K. V. Rashmi, and Gregory R. Ganger. 2022. Tiger: Disk-Adaptive Redundancy Without Placement Restrictions. In *Proc. of USENIX OSDI*.
[13] Rajat Kateja, Nathan Beckmann, and Gregory R. Ganger. 2020. TVARAK: Software-Managed Hardware Offload for Redundancy in Direct-Access NVM Storage. In *Proc. of ISCA*.
[14] Rajat Kateja, Andy Pavlo, and Gregory R. Ganger. 2020. Vilamb: Low Overhead Asynchronous Redundancy for Direct Access NVM. *arXiv preprint* (2020).
[15] Cong Li, Yu Zhang, Jialei Wang, Hang Chen, Xian Liu, Tai Huang, Liang Peng, Shen Zhou, Lixin Wang, and Shijian Ge. 2022. From Correctable Memory Errors to Uncorrectable Memory Errors: What Error Bits Tell. In *Proc. of SC*.
[16] Zhenxin Li, Shuibing He, Zheng Dang, Peiyi Hong, Xuechen Zhang, Rui Wang, and Fei Wu. 2024. CCL-BTree: A Crash-Consistent Locality-Aware B+-Tree for Reducing XPBuffer-Induced Write Amplification in Persistent Memory. In *Proc. of EuroSys*.
[17] Jianqiang Luo, Mochan Shrestha, Lihao Xu, and James S. Plank. 2014. Efficient Encoding Schedules for XOR-Based Erasure Codes. *IEEE TC* (2014).
[18] Tianyang Niu, Min Lyu, Wei Wang, Qiliang Li, and Yinlong Xu. 2023. Cerasure: Fast Acceleration Strategies For XOR-Based Erasure Codes. In *Proc. of ICCD*.
[19] James S Plank, Kevin M Greenan, and Ethan L Miller. 2013. Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions. In *Proc. of USENIX FAST*.
[20] Jiansheng Qiu, Yanqi Pan, Wen Xia, Xiaojia Huang, Wenjun Wu, Xiangyu Zou, Shiyi Li, and Yu Hua. 2023. Light-Dedup: A Light-Weight Inline Deduplication Framework for Non-Volatile Memory File Systems. In *Proc. of USENIX ATC*.
[21] I. S. Reed and G Solomon. 1960. Polynomial Codes Over Certain Finite Fields. *J. Soc. Ind. Appl. Math.* (1960).
[22] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. 2020. Reverse Engineering the Stream Prefetcher for Profit. In *Proc. of IEEE EuroS&P Workshops*.
[23] Stuart J Russell and Peter Norvig. 2016. *Artificial intelligence: a modern approach*. pearson.
[24] Samsung. 2022. Memory-Semantic SSD. https://samsungmsl.com/ms-ssd/
[25] Yuya Uezato. 2021. Accelerating XOR-Based Erasure Coding Using Program Optimization Techniques. In *Proc. of SC*.
[26] Ronglong Wu, Zhirong Shen, Zhiwei Yang, and Jiwu Shu. 2024. Mitigating Write Disturbance in Non-Volatile Memory via Coupling Machine Learning with Out-of-Place Updates. In *Proc. of HPCA*.
[27] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the Performance of Intel Optane Persistent Memory: A Close Look at Its on-DIMM Buffering. In *Proc. of EuroSys*.
[28] Guanglei Xu, Yuchong Hu, Dan Feng, Wenpeng He, and Junyuan Huang. 2024. CodePM: Parity-Based Crash Consistency for Log-Free Persistent Transactional Memory. *IEEE TCAD* (2024).
[29] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proc. of SOSP*.
[30] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proc. of USENIX FAST*.
[31] Juncheng Yang, Yao Yue, and KV Rashmi. 2020. A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *Proc. of USENIX OSDI*.
[32] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2015. Computer Performance Microscopy with Shim. In *Proc. of ISCA*.
[33] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. 2022. MT2: Memory Bandwidth Regulation on Hybrid NVM/DRAM Platforms. In *Proc. of USENIX FAST*.
[34] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *Proc. of USENIX ATC*.
[35] Tianli Zhou and Chao Tian. 2019. Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques. In *Proc. of USENIX FAST*.
[36] Bohong Zhu, Youmin Chen, and Jiwu Shu. 2024. Exploring the Asynchrony of Slow Memory Filesystem with EasyIO. In *Proc. of EuroSys*.