SCISSOR: Scalable I/O for Small Scattered Objects Runtime

Kevin Assogba
Rochester Institute of Technology
Rochester, New York, USA

Nigel Tan
Los Alamos National Laboratory
Los Alamos, New Mexico, USA

M. Mustafa Rafique
Rochester Institute of Technology
Rochester, New York, USA

Michela Taufer
University of Tennessee Knoxville
Knoxville, Tennessee, USA

Bogdan Nicolae
Argonne National Laboratory
Lemont, Illinois, USA

Abstract

Modern data-intensive workloads across high-performance computing (HPC) and artificial intelligence (AI) exhibit I/O patterns dominated by small, non-contiguous reads from large files, incurring high system call overhead, inefficient batching, and underutilized storage bandwidth. We present *SCISSOR*, an I/O runtime for workloads dominated by fine-grained random access. First, *SCISSOR* applies spatially aware offset coalescing to merge nearby requests into contiguous segments, reducing kernel interactions. Second, a regression-based performance model selects batching and coalescing parameters that balance I/O and computation throughput. Third, *SCISSOR* leverages `io_uring` to issue large batches of asynchronous reads. On HPC reproducibility analytics over cosmology simulations (HACC), *SCISSOR* achieves 3.6× end-to-end throughput over `io_uring` and 42× over POSIX memory-mapped I/O on average, and 2.56× over `io_uring` on AI data-loading pipelines.

Keywords

Data loading, Offset coalescing, `io_uring`, Non-contiguous I/O

1 Introduction

Data-intensive workloads at the intersection of HPC and AI use I/O patterns dominated by concurrent, small, random, non-contiguous reads from large files, where the offsets to be read are largely deterministic, in the sense that they are known in advance or can be inferred indirectly. In reproducibility analysis, downstream comparison routines use sampling or fingerprinting [2] to issue many small reads at offsets derived from checkpoint metadata, and AI training pipelines traverse massive datasets of small samples where similar spatial clustering arises. This recurring signature exposes a coalescing opportunity that existing I/O interfaces fail to exploit.

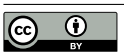
Traditional POSIX-compliant interfaces treat each I/O as an isolated synchronous request, so workloads of millions of small reads spend more cycles crossing the kernel-user boundary than moving data. Higher-level HPC libraries such as MPI-IO, e.g., ROMIO [3], introduce data sieving, which services a set of nearby requests by reading a bounding region into a sieve buffer and copying the relevant pieces to user space. Data sieving is the closest prior technique to *SCISSOR*'s gap-aware coalescing. *SCISSOR* adopts the same core idea for the per-process case and extends it along three axes: coalesced reads are issued asynchronously via `io_uring` with deep submission queues; the sieve region size (gap) is selected jointly with a batch size using a lightweight I/O-compute regression model; and *SCISSOR*'s coalescing is integrated into a producer-consumer pipeline, whereas data sieving operates as a synchronous primitive within a single MPI-IO call. Modern asynchronous submission interfaces such as `io_uring` [1] amortize kernel crossings via shared submission/completion ring buffers, yet remain pattern-agnostic: even when millions of requests are submitted efficiently, the storage stack still executes each one as a discrete operation. As a consequence, no existing approach simultaneously reduces both the *number* and the *randomness* of fine-grained reads.

This work introduces *SCISSOR*, an asynchronous I/O technique optimized for non-contiguous, random, fine-grained reads under concurrency. The observation we exploit is that many non-contiguous operations land close to each other, so clustered small reads can be coalesced into a single I/O operation, at the cost of reading more data than strictly necessary. *SCISSOR* studies scattered, random, and small read access patterns drawn from representative HPC and AI workloads, introduces gap-aware coalescing and co-optimized producer-consumer I/O scheduling that uses a regression-based model to select the optimal gap and batch size, and implements an asynchronous engine built on top of `io_uring`.

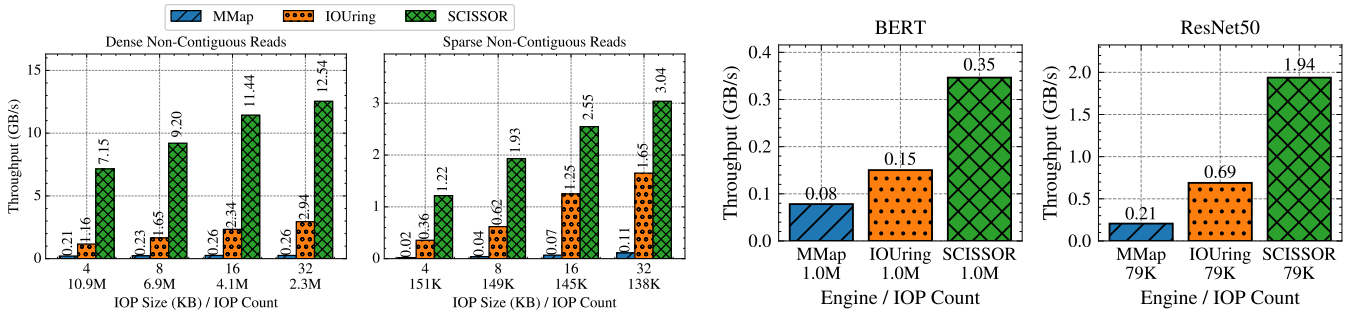
2 *SCISSOR* System Design

The design of *SCISSOR* is guided by three key principles, aiming to balance the trade-off between coalescing of IOPs and extra unnecessary data movement. First, *SCISSOR* introduces low-overhead gap-aware coalescing, merging adjacent read operations whose offsets in the original file fall within a tunable gap. This exploits the partial spatial locality observed in scattered workloads, where offsets cluster within limited ranges despite appearing globally scattered. We denote each read operation as $r_i = (o_i, s_i)$, where o_i is the file offset and s_i is the requested size. Two IOPs r_i and r_j can be coalesced into a single I/O request of size $s^*_{ij} = s_i + \delta_{ij} + s_j$, if the distance $\delta_{ij} = o_j - (o_i + s_i)$ is smaller than a tunable *gap size*. Because the gap size is fixed, coalescing reduces to a single linear scan over sorted offsets with $\Theta(n)$ complexity, enabling the *Aggregator* to process millions of IOPs with negligible overhead. Second, *SCISSOR* tracks pending and completed IOPs and adjusts the completion-polling interval based on their ratio. Third, *SCISSOR* uses a lightweight regression model to select batch and gap sizes that balance data-loading and processing completion times.

SCISSOR comprises four components: the *Aggregator*, *Regressor*, *Loader*, and *Reader*. The *Aggregator* receives raw read offsets and



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846884



(a) HPC reproducibility analysis (dense and sparse).

(b) AI training data loading (BERT and ResNet50).

Figure 1: End-to-end throughput for reproducibility analysis (a) and AI data loading with BERT (8 KB IOPs) and ResNet50 (160 KB IOPs) (b).

analyzes their spatial locality through a contiguity analysis that identifies adjacent or nearby accesses. Offset groups and I/O sizes are passed to the *Regressor*, which determines the optimal batch and coalescing gap sizes using I/O and compute models. Based on these parameters, the *Aggregator* generates coalesced read segments, the *Loader* batches them for asynchronous submission, and the *Reader* issues the batched reads via `io_uring`, after which the loaded data is staged out and made available to computation. The *Regressor* selects the gap size g and batch size C that yield the highest throughput, evaluating candidate configurations to identify (g^*, c^*) that reduce the imbalance between I/O and compute throughputs. It uses two lightweight models, based on micro-benchmarking, to predict I/O and computation time for a candidate (g, c) configuration, defined as $T_{\text{io}}(N, S) = K \cdot N^{\alpha} \cdot e^{-\beta N} \cdot S^{\gamma} \cdot e^{-\delta S}$ and $T_{\text{comp}}(B) = \kappa \cdot B^{\lambda}$, with configuration selection executing in under a millisecond. *SCISSOR* is implemented as a user-level I/O runtime library built on top of `io_uring`, and each *SCISSOR* runtime instance uses a single `io_uring` ring to serve one or multiple files concurrently. While computation proceeds on the elements of one batch, reads for subsequent segments are issued asynchronously.

3 Performance Comparison

We evaluate *SCISSOR* on two workload classes using the Polaris supercomputer at the Argonne Leadership Computing Facility: HPC reproducibility analytics with HACC checkpoints, the most demanding regime for *SCISSOR* with very small IOPs (4–32 KB), millions of reads per batch, and two access densities (Dense and Sparse, corresponding to 10^{-7} and 10^{-3} reproducibility error margins), and AI training data loading, a complementary regime with medium-to-large IOPs (8 KB for BERT and 160 KB for ResNet50) over class-clustered datasets. We compare *SCISSOR* against `mmap` (MMap) and `io_uring` (IOUring) baselines, where all engines start from a cold cache, and IOUring and *SCISSOR* share an identical `io_uring` configuration so that throughput differences are attributable to request structure rather than submission-engine efficiency.

We report the end-to-end throughput of *SCISSOR* against MMap and IOUring baselines in Figure 1. Across all configurations, *SCISSOR* outperforms both baselines. On HACC reproducibility analysis, *SCISSOR* achieves an average speedup of 3.6× over IOUring (1.75×–7.1×) and 42× over MMap (26×–66×), with 5.4× on average across `dense` patterns, where coalescing has abundant spatial locality to exploit, and 2.4× on average across `sparse` patterns, where fewer offsets fall within a coalescing gap. On AI workloads, *SCISSOR* delivers a 2.56× average improvement over IOUring, with 2.33× on BERT and 2.81× on ResNet50. MMap performs worst because each missing page triggers a synchronous page fault and kernel intervention, leading to frequent context switches and page-cache handling. IOUring improves over MMap by submitting requests asynchronously and amortizing system-call overhead, but still executes IOPs independently at the application-defined granularity. *SCISSOR* removes this overhead by coalescing adjacent accesses into larger contiguous reads, reducing IOP count issued to the storage.

4 Lessons Learned

Our evaluation of *SCISSOR* on HPC reproducibility analytics and AI training data loading yields three lessons. First, gap-aware coalescing is most effective where offsets cluster tightly: *SCISSOR* gains 5.4× on average across dense patterns and 2.4× across sparse patterns. Second, reducing the number of I/O operations matters more than the submission engine alone: IOUring amortizes system-call overhead through batched asynchronous submission but still executes IOPs independently at the application-defined granularity, whereas *SCISSOR* coalesces adjacent accesses into larger contiguous reads, cutting the 1-billion particle configuration with 4 KB IOPs from 10.9M to ≈253K at a 32 KB gap. Third, the same coalescing opportunity carries across workload classes: *SCISSOR* delivers 3.6× over IOUring and 42× over MMap on HACC reproducibility analysis, and 2.56× over IOUring on AI data loading.

Acknowledgments

This material is based upon work supported by: the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357; the National Science Foundation under Grants #1900888, #1900765, #2223704, #2331152, #2411386, #2411387, #2106635.

References

[1] Jens Axboe. 2020. Efficient IO with io_uring. *Linux Weekly News* (2020). https://lwn.net/Articles/810414/

[2] Nigel Tan, Kevin Assogba, Walter J. Ashworth, Befikir Bogale, Franck Cappello, M. Mustafa Rafique, Michela Taufer, and Bogdan Nicolae. 2024. Towards Affordable Reproducibility Using Scalable Capture and Comparison of Intermediate Multi-Run Results. In *The 25th International Middleware Conference*. 392–403. doi:10.1145/3652892.3700780

[3] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. Data sieving and collective I/O in ROMIO. In *7th IEEE Symposium on the Frontiers of Massively Parallel Computation*. 182–189. Available: 10.1109/FMPC.1999.750599.