FBench: A Flexible Benchmark for CFG-Based What-If Exploration of HPC I/O Patterns

(Summary)

Zhaobin Zhu*, Chen Wang†, Kathryn Mohror‡, and Sarah Neuwirth*
*Johannes Gutenberg University Mainz, Germany †Nanyang Technological University, Singapore
‡Lawrence Livermore National Laboratory, USA {zhu,neuwirth}@uni-mainz.de

***Abstract*—The I/O performance of large-scale HPC applications depends on a complex interplay of access patterns, middleware optimizations, and file system configurations. To explore these effects without rerunning full applications, we introduce FBench, a flexible, code-transparent benchmark for what-if analysis. FBench leverages context-free grammars (CFGs) derived from Recorder traces to either generate a simplified global configuration file or replay I/O patterns on-the-fly without preprocessing. It supports POSIX and MPI-IO and injects optimization hints via JSON. FBench reproduces IOR's scaling within 6.9%, captures the irregular file-per-process pattern of HACC-IO, and reveals a $30\times$ collective/independent gap on Lustre that striping cannot close.**

***Index Terms*—HPC, I/O performance, performance modeling, I/O pattern, benchmark**

I. Problem and Motivation

In the exascale era, I/O time can quickly become a limiting bottleneck [1]. Performance depends sensitively on three factors: the access pattern, I/O library and middleware optimizations, and the file system configuration [2].

Tracing and profiling tools [3], [4] show where an application loses I/O time. DXTExplorer [2], Drishti [5], and IOSIG [6] identify bottlenecks and suggest optimizations, yet offer no way to evaluate those suggestions at the application level, and direct evaluation with real applications is expensive. Existing benchmarks are either too specific [7], [8] or too generic [9] to capture realistic patterns; developers therefore hand-write error-prone I/O kernels [10], and trace-based approaches cannot vary the pattern without re-tracing [11].

FBench models patterns as CFGs, replays them either from a global configuration file or directly from the per-rank traces, and applies optimization hints at the pattern level.

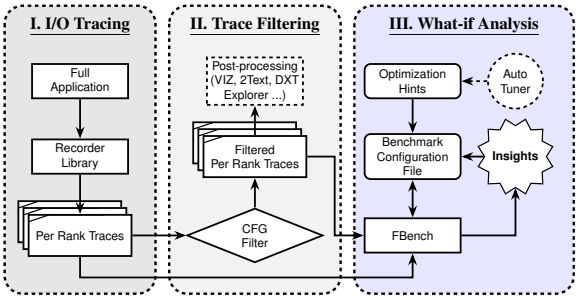


Fig. 1: FBench workflow: I/O tracing, trace filtering, and what-if analysis.

II. The Flexible Benchmark – FBench

CFGs [12] represent traces as structured languages with nesting and repetition: recurring patterns become grammar rules, compressing storage while preserving semantics, so modern tracing tools [13], [14] use CFG-based encodings. We use Recorder [15], which captures all calls across the targeted I/O layers with their complete parameter sets at only $\sim 3\,\%$ overhead.

Figure 1 shows the workflow. The application runs preloaded with Recorder, which emits a per-rank CFG preserving the structural relationships between operations. Since Recorder encodes each call including its arguments as a unique symbol, similar operations at different offsets become distinct; an optional filtering phase groups them, shrinking the CFG while keeping it valid for Recorder-Viz, Drishti, and DXT-Explorer.

Global view (Fig. 2). FBench derives a JSON configuration in four steps analogous to CFG reconstruction: group operations by transfer size, operation type, file, and rank; merge within an open–close session; group by access type (shared-file vs. file-per-process); and merge by session count. The volume per file operation is

$$D = R_{\text{total}} \cdot \sum_{c \in \text{chunks}} \left(\sum_{r \in \text{ranks}(c)} n_{r,c} \cdot s_c \right), \qquad (1)$$

with R_{total} open–close repetitions, $n_{r,c}$ repetitions of rank r for chunk c, and transfer size s_c. Preprocessing is $\mathcal{O}(R \cdot N + S)$ and execution $\mathcal{O}(S)$. The global view discards temporal structure by construction, which suffices for aggregate per-file throughput in checkpoint or dump phases.

On-the-fly replay. OTF instead replays directly from the per-rank traces, turning every event into an I/O operation with its original size, offset, and flags at $\mathcal{O}(R \cdot N)$ and no preprocessing. Recorder encodes MPI file handles as globally unique integers, so collective calls are matched across ranks without sub-communicators (Fig. 3). OTF also replays seeking and pointer updates, and can insert the recorded inter-event delays.

Optimization hints (Fig. 4). The same configuration carries POSIX hints (`use_direct`, `disable_readahead`, Lustre stripe size and count) and MPI-IO hints (collective I/O, `cb_buffer_size`, `cb_nodes`, `romio_*`). Separating replay engine from tunables lets hints from Drishti or IOSIG be evaluated without rerunning the application.

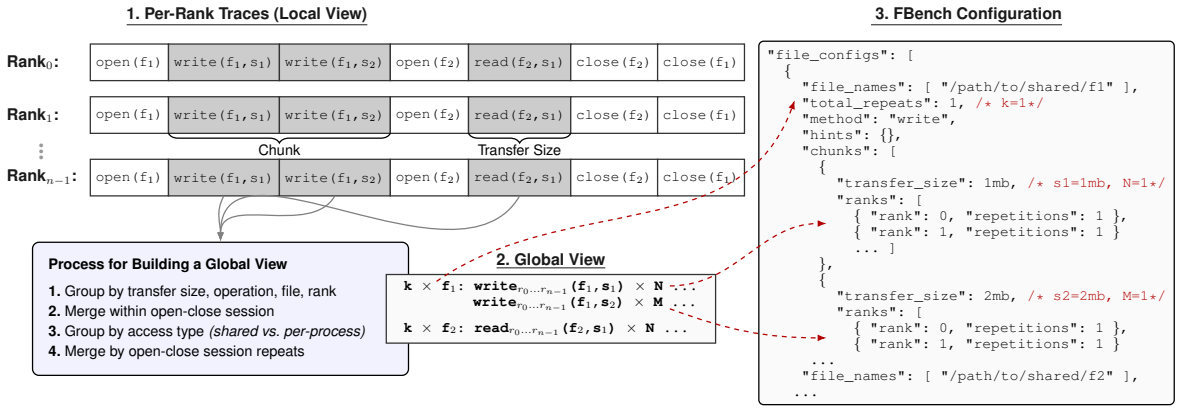


Fig. 2: Transformation of per-rank traces into a structured FBench configuration via a global view. Two shared files (`f1`, `f2`) yield two configuration entries involving all ranks. The write entry for `f1` holds two chunks with transfer sizes `s1` and `s2`, each aggregating how often the operation is executed across all ranks; the read entry for `f2` follows the same structure with a single chunk.

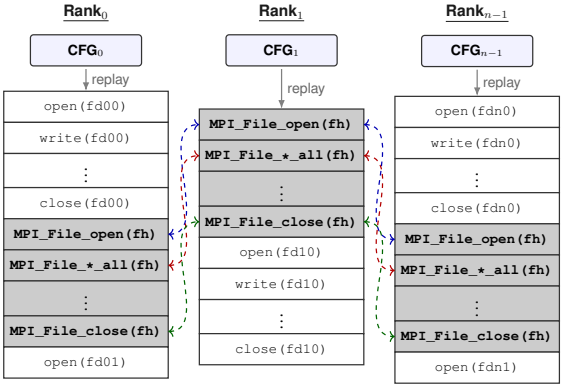


Fig. 3: On-the-fly (OTF) replay global synchronization using MPI-IO collective operations.

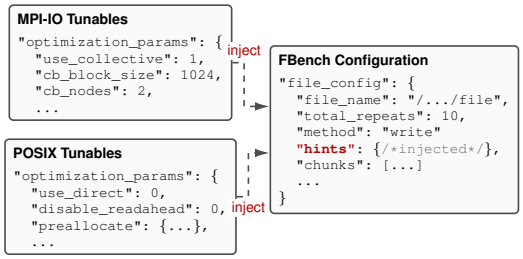


Fig. 4: Injection of MPI-IO and POSIX optimization parameters into the FBench configuration, enabling backend-specific tuning without code changes.

TABLE I: Evaluation workloads and what FBench reproduces or reveals.

Workload	Setup	Result
IOR	1,024 tasks, 32 nodes	Matches scaling within 0.6–6.9% across interfaces and access patterns
HACC-IO	480 ranks, FPP	Reproduces the irregular chunk sequence; stripe *count* dominates, not stripe size
FLASH	20 nodes, MPI-IO	30× collective/independent gap; 1.5× from UnifyFS, untuned
LAMMPS	124 GB dump	18% of the original runtime; up to 8× from simple tuning

III. Evaluation

All experiments ran on Corona (LLNL) [16]; Table I summarizes the workloads.

IOR ran with POSIX and MPI-IO across 1 to 32 nodes, file-per-process (FPP) and single-shared-file (SSF), collective and non-collective. FBench follows IOR's scaling throughout (Fig. 5): deviations stay below 1% for POSIX SSF and FPP writes and reach 6.86% only for caching-dominated MPI-IO independent reads. OTF stays within ∼5% of the configuration-based replay on three of four configurations.

HACC-IO (Fig. 6) writes blocks of 20, 24, 40, and 80 MB per rank; the read phase reuses these sizes in a different order. IOR cannot express this, yet FBench reproduces it from the configuration alone. Over Lustre striping, reads are page-cache dominated and the dip is tied to the stripe count, not the stripe size, while writes scale to over 50 GB/s at factor 8.

FLASH Sedov [17], [18] shows little headroom from collective buffering (340–400 MB/s) but a pronounced access-mode gap (Fig. 7a): 180 MB/s collective against 5.2 GB/s independent, so the access mode, not striping, dominates; UnifyFS [19] lifts independent writes to ∼8 GB/s untuned (Fig. 7b, c). **LAMMPS** [20] reproduces its 124 GB shared dump in 101.8 s instead of 566.5 s, and exposes that the aggregator count is coupled to the striping count, capping aggregation at 8 processes. With delays enabled, OTF reproduces all five bursts and matches every inter-burst gap within 1–4%.

IV. Conclusion

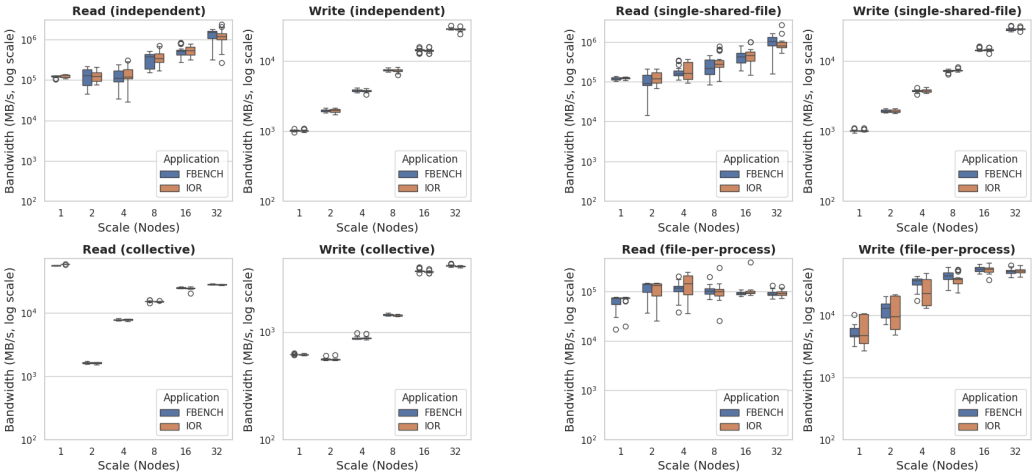
FBench bridges detailed I/O tracing and actionable tuning: CFG-based traces drive either a simplified global configuration or a full-fidelity on-the-fly replay, with hints injected without touching application code. Future work targets HDF5, netCDF, PnetCDF, and cross-rank pattern scaling.

Acknowledgment

The authors gratefully acknowledge funding from the Federal Ministry of Research, Technology and Space (BMFTR) and the state governments participating in the NHR (www.nhr-verein.de/unsere-partner). This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-POST-2024820).

References

[1] P. Carns, J. Kunkel, K. Mohror, and M. Schulz, "Understanding i/o behavior in scientific and data-intensive computing (dagstuhl seminar 21332)," *Dagstuhl reports*, vol. 11, no. 7, pp. 16–75, 2021.



(a) MPI-IO Interface **(b)** POSIX Interface

Fig. 5: Scaling comparison of FBench and IOR up to 1,024 tasks on 32 nodes. Bandwidth increases steadily with the number of nodes across all access patterns, and both tools follow nearly identical trends.

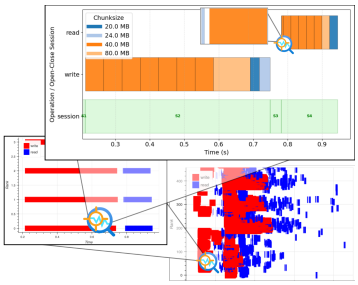


Fig. 6: I/O access pattern of HACC-IO on 480 ranks (10 nodes), with more than one million particles per rank.

[2] J. L. Bez, H. Tang, B. Xie, D. Williams-Young, R. Latham, R. Ross, S. Oral, and S. Byna, "I/o bottleneck detection and tuning: Connecting the dots using interactive log analysis," in *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*, 2021.

[3] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient parallel i/o tracing and analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 1–8.

[4] S. Neuwirth and A. K. Paul, "Parallel i/o evaluation techniques and emerging hpc workloads: A perspective," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 671–679.

[5] J. L. Bez, H. Ather, and S. Byna, "Drishti: Guiding end-users in the i/o optimization journey," in *2022 IEEE/ACM International Parallel Data Systems Workshop (PDSW)*. IEEE, 2022, pp. 1–6.

[6] Y. Yin, S. Byna, H. Song, X.-H. Sun, and R. Thakur, "Boosting application-specific parallel i/o optimization using iosig," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE, 2012, pp. 196–203.

[7] M. Zingale, "FLASH I/O Benchmark Routine," https://www.ucolick.org/~zingale/flash_benchmark_io/, accessed: 2025-11-26.

[8] G. K. Lockwood, "HACC-IO Repository," https://github.com/glennklockwood/hacc-io, 2012, accessed: 2025-11-17.

[9] hpc/ior, "Ior and mdtest parallel i/o benchmark repository," https://github.com/hpc/ior, 2024, accessed: 2025-11-17.

[10] J. Logan, S. Klasky, J. Lofstead, H. Abbasi, S. Ethier, R. Grout, S.-H. Ku, Q. Liu, X. Ma, M. Parashar *et al.*, "Skel: generative software for producing skeletal i/o applications," in *2011 IEEE Seventh International Conference on e-Science Workshops*. IEEE, 2011, pp. 191–198.

[11] S. Snyder, P. Carns, R. Latham, M. Mubarak, R. Ross, C. Carothers, B. Behzad, H. V. T. Luu, S. Byna, and Prabhat, "Techniques for modeling large-scale hpc i/o workloads," in *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, 2015, pp. 1–11.

[12] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *Acm Sigact News*, vol. 32, no. 1, pp. 60–65, 2001.

[13] J. Luo, T. Yan, Q. Xu, J. Sun, and G. Sun, "Siesta: Synthesizing proxy applications for mpi programs," in *2024 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2024, pp. 14–26.

[14] C. Wang, P. Balaji, and M. Snir, "Pilgrim: scalable and (near) lossless mpi tracing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[15] C. Wang, I. Yildirim, H. Devarajan, K. Mohror, and M. Snir, "Recorder: Comprehensive parallel i/o tracing and analysis," *arXiv preprint arXiv:2501.04654*, 2025.

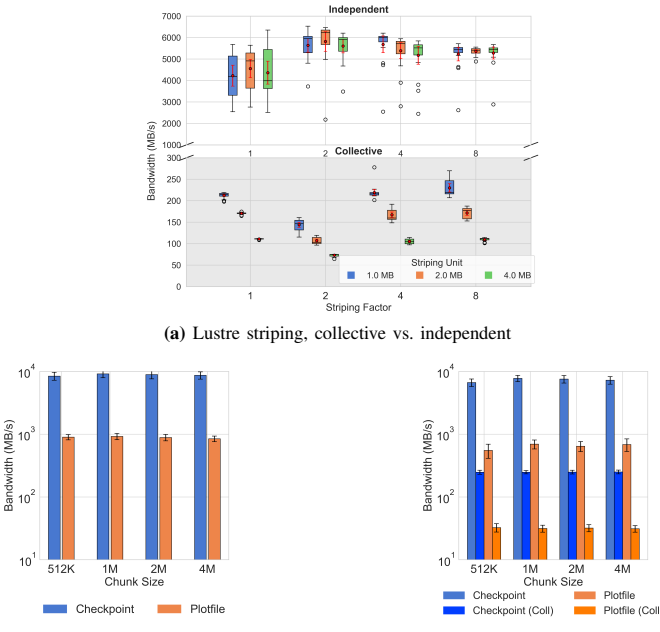
[16] LLNL, "Corona Compute Platform," https://hpc.llnl.gov/hardware/compute-platforms/corona, 2025, accessed: 2025-07-03.

[17] flash x.org, "Flash-x: A multiphysics scientific software system," https://flash-x.org, 2024, accessed: 2025-11-17.

[18] L. I. Sedov, *Similarity and dimensional methods in mechanics*. CRC press, 2018.

[19] M. J. Brim, A. T. Moody, S.-H. Lim, R. Miller, S. Boehm, C. Stanavige, K. M. Mohror, and S. Oral, "Unifyfs: A user-level shared file system for unified access to distributed local storage," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023.

[20] S. J. Plimpton, A. Kohlmeyer, A. P. Thompson, S. G. Moore, and R. Berger, "Lammps: large-scale atomic/molecular massively parallel simulator," *Zenodo*, 2007.



(a) Lustre striping, collective vs. independent

(b) FBench on UnifyFS **(c)** FLASH on UnifyFS

Fig. 7: FLASH Sedov what-if results. (a) Bandwidth distribution (FBench) by striping factor for write checkpoint files (mean and ±95% confidence interval in red); independent I/O consistently achieves higher bandwidth than collective I/O (gray background). (b, c) Write bandwidth on UnifyFS: FBench and FLASH exhibit similar performance for independent I/O, and FLASH confirms that collective I/O performs significantly worse.