POSTER: NDS: Programmer-Free Offload of High-Performance Near Data Strands

Shreyas Singh
University of Utah
USA

Pratyush Nandi
University of Utah
USA

Lin Jia
University of Utah
USA

Shankar Balachandran
Intel
India

Rajeev Balasubramonian
University of Utah
USA

Abstract

Near-data processing (NDP) reduces data movement and exploits parallelism near memory, yet realizing those gains usually requires explicit data movement, offload interfaces, compiler support, or NDP-specific layouts. Near Data Strands (NDS) explores whether hardware can identify, validate, and offload eligible repeated regular loops from unmodified binaries without changes to the program, binary, or compiler. In simulation, NDS improves performance in the measured kernel regions of 12 kernels by 3.14× geomean versus one out-of-order host core and by 1.82× versus benchmark-provided 8-core host-parallel versions.

CCS Concepts

• **Computer systems organization** → **Parallel architectures**; *Heterogeneous (hybrid) systems.*

Keywords

near-data processing, processing-in-memory, transparent offload, hardware loop analysis, micro-operation scheduling

1 Introduction

Near-data processing (NDP) reduces data movement and exposes memory-side parallelism, yet most systems require applications to adopt offload interfaces, compiler support, or explicit data movement [2, 3]. NDS asks whether hardware can instead identify a constrained class of repeated regular loops in unmodified binaries and run them near memory with no program, binary, or compiler changes.

NDS observes repeated regular loops on an out-of-order host and uses their runtime structure and memory behavior to decide whether near-data execution is safe and worthwhile. It splits the iteration space into strands, contiguous portions each assigned to one simple near-data core (NDCore), and marshals their pages; candidates failing structure, version, page-mapping, or expected-benefit checks stay on the host.

2 NDS design

Figure 1 shows the two hardware actors, the SAE beside each host core and the LOU at each memory controller.

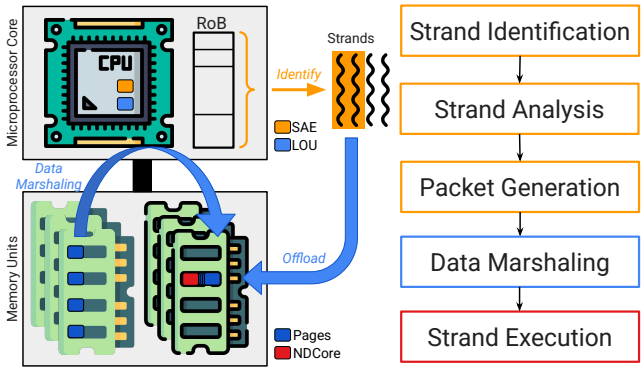


Figure 1: NDS workflow. The Strand Analysis Engine (SAE) records a template for an eligible loop after profiling, and the Loop Orchestration Unit (LOU) validates a later matching invocation, places pages, and dispatches strands to near-data cores (NDCores) beside memory banks.

Strand identification. Watching dispatched micro-operations, the SAE recognizes either a forward branch followed by an unconditional backward jump or a backward branch. After the closing branch commits, it promotes the loop to a candidate and profiles a bounded number of iterations off the critical path.

Strand analysis and packet generation. The SAE templates loops that fit resource limits, use supported operations and constant-stride references, and have no internal branches or system calls. A bounded dependence walk identifies a constant induction update and a loop-invariant limit whose inputs are constants or live-ins. The template records the host's out-of-order issue order and derives an in-order NDCore schedule that may interleave iterations. On a later invocation, an exact match between current architectural values and the stored version values makes the invocation eligible, and the LOU instantiates an offload packet with strands, live-ins, bounds, and bank assignments.

Data marshaling. NDCores access only their attached bank. The LOU at each memory controller proposes physical page and bank placement; a privileged OS/MMU path commits allocation or migration, updates translations, and performs TLB invalidations [5, 10]. A written page has one owning NDCore bank during an offload, while read-only pages may be replicated. At the loop boundary, the LOU acquires pages, flushes or invalidates host lines, and refreshes replicas; after ownership is released, host lines are refetched or revalidated [7].

Strands are not speculative: dependence and placement checks complete before launch, and a loop whose placement cannot be

An extended 11-page version of this paper is available on ArXiv.

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/10.1145/3838684.3846876

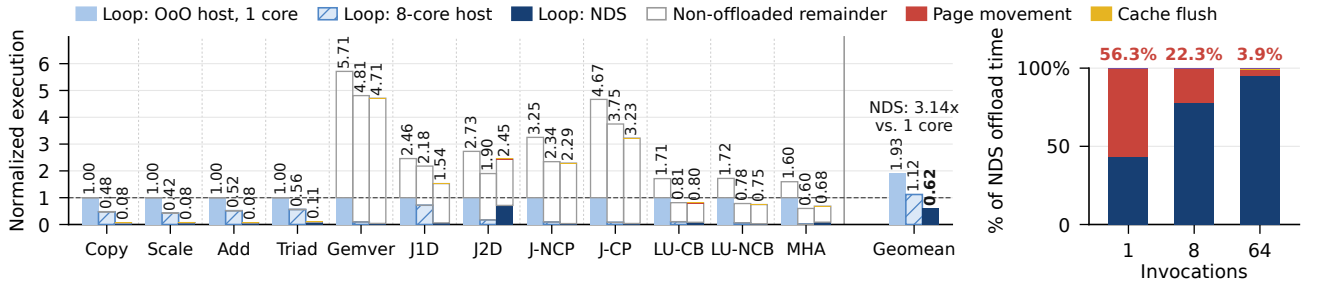


Figure 2: Left: normalized execution in measured kernel regions for one host core, benchmark-provided 8-core host-parallel versions (hatched), and NDS. Bars stack selected-loop time, outlined remainder, and page-movement and cache-flush costs. Right: arithmetic-mean NDS offload-time composition over 1, 8, and 64 invocations, separating loop execution, initial movement, and flushes.

committed safely runs on the host. In-place writes are supported only when page-level ownership represents the required read-after-write and write-after-write dependences: iterations that touch a written page run at its owning bank, independent write regions proceed on different banks, and a dependence the page-level model cannot represent leaves the loop on the host.

Strand execution. x86 instructions expand into dependent micro-operations on which an in-order core stalls. NDS replays the issue order observed on the out-of-order host as the NDCore schedule, preserving register dependences; ambiguous memory aliases retain program order or leave the invocation on the host.

Execution contract. Each accepted offload is blocking and runs to completion; the host waits at the loop boundary. The model assumes stable physical mappings and no concurrent host access to acquired pages; NDCores have no TLB and use physical pages resolved by the host MMU, and host fallback occurs only before launch. Interrupts and context switches during an offload, recovery after launch, and arbitrary architectural live-outs are outside the modeled design.

3 Methodology

Simulation model. Our evaluation combines measured out-of-order host traces with simulated SAE, LOU, and NDCore behavior in Sniper [1], measuring full kernel runs and isolated inner loops; learned micro-operation schedules run on simple in-order NDCores. The Coffee Lake-based model has an 8-core 3.6 GHz host and 256 bank-local NDCores at 350 MHz. It estimates interface costs rather than implementing OS migration, blocking waits, or recovery.

Workloads, costs, and baselines. We use 12 kernels at large inputs, evaluating measured regions: STREAM Copy, Scale, Add, and Triad [6], Ocean CP/NCP and LU CB/NCB from Splash-3 [9], Gemver and Jacobi 1D/2D from PolyBench/C [8], and llama2.c multi-head attention (MHA) on TinyLlama-1.1B, timed to its 2048th token [4]. Page movement uses reported UPMEM CPU-to-DPU bandwidth as a conservative proxy [3], and every offload is charged a worst-case 308 μs cache flush. Baselines are the out-of-order host and benchmark-provided 8-core host-parallel kernels, with the same inputs, regions, and timestamp method.

4 Evaluation and scope

Across the measured kernel regions of 12 kernels (Figure 2), NDS improves performance by 3.14× geomean (1.11 to 13.23×) versus one out-of-order host core and by 1.82× versus benchmark-provided 8-core host-parallel versions; NDS is faster on 10 of 12 kernels. Initial page movement is 56.3% of one-invocation offload time and 3.9% on average at 64 invocations.

Coverage and profitability. In the instrumented regions, the SAE detects all 33 loops matching the two supported patterns; 15 pass both checks (12 inner and 3 outer), 14 fail parallelizability, and 4 fail offloadability (unsafe page mappings, conflicting dependences, unsupported bodies, or too little work).

Gains depend on the offloaded fraction and the initial page layout. The selected loop is 17.5% of measured Gemver time and 36.7% of Jacobi 2D, and effective NDCores (those doing useful work) range from 11.7 to about 254, averaging 202, because a transparent design cannot remap data before the first offload. Jacobi 2D and MHA trail the 8-core host because page mapping exposes only 11.7 and 64 effective NDCores; MHA's host-parallel version is also effective.

Sensitivity. Alone, a host-class out-of-order core next to memory is 1.16× slower than the host; a reduced-capability out-of-order NDCore is 8.84× slower, and an unscheduled in-order NDCore is 15.31× slower. Schedule replay with program-order fallback (SS+, used here) is 1.39× faster than unscheduled in-order execution, and issue width 2 (used here) gives 1.45× over width 1.

5 Conclusion

In simulation, hardware can identify and validate constrained regular loops in unmodified binaries and run them as near-data strands profitably when reuse and bank-level parallelism justify modeled movement and flush costs. Future work includes vectorized and unrolled host baselines (which also change the micro-operation stream NDS observes), broader loop classes, and post-launch recovery.

Acknowledgments

We thank the reviewers. This work was supported in part by NSF grant CCF-2217154 and Intel.

References

[1] Trevor E Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 3 (2014), 1–25. https://doi.org/10.1145/2629677

[2] Fabrice Devaux. 2019. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE Computer Society, IEEE, Piscataway, NJ, USA, 1–24. https://doi.org/10.1109/HOTCHIPS.2019.8875680

[3] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. 2021. Benchmarking memory-centric computing systems: Analysis of real processing-in-memory hardware. In *2021 12th International Green and Sustainable Computing Conference (IGSC)*. IEEE, Pullman, WA, USA, 1–7. https://doi.org/10.1109/IGSC54211.2021.9651614

[4] Andrej Karpathy. 2023. llama2.c. https://github.com/karpathy/llama2.c. Accessed: 2026-09-15.

[5] Dongjae Lee, Bongjoon Hyun, Taehun Kim, and Minsoo Rhu. 2024. PIM-MMU: A Memory Management Unit for Accelerating Data Transfers in Commercial PIM Systems. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, IEEE, Los Alamitos, CA, USA, 627–642. https://doi.org/10.1109/MICRO61859.2024.00053

[6] John D McCalpin. 1995. Stream benchmark. https://www.cs.virginia.edu/stream/. Accessed: 2026-09-15.

[7] Ben Perach, Ronny Ronen, and Shahar Kvatinsky. 2023. On Consistency for Bulk-Bitwise Processing-in-Memory. In *2023 IEEE International Symposium on High-Performance Computer Architecture*. IEEE, Montreal, QC, Canada, 705–717. https://doi.org/10.1109/HPCA56546.2023.10071007

[8] Louis-Noël Pouchet and Tomofumi Yuki. 2016. PolyBench/C 4.2.1: The Polyhedral Benchmark Suite. https://sourceforge.net/projects/polybench/. Accessed: 2026-09-15.

[9] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Uppsala, Sweden, 101–111. https://doi.org/10.1109/ISPASS.2016.7482078

[10] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 331–345. https://doi.org/10.1145/3297858.3304024