# HostPIMSim: A High-Productivity Framework for Host-Side Processing-in-Memory Simulation

Jihong Min[1] [0000−0001−9832−9573] and Bernhard Egger ✉[1,2] [0000−0002−6645−6161]

[1] Seoul National University, Seoul, Republic of Korea
[2] Lucerne University of Applied Sciences and Arts, Lucerne, Switzerland
{jihong,bernhard}@csap.snu.ac.kr

**Abstract.** Processing-in-Memory (PIM) reduces data movement in memory-bound computing infrastructures, promising significant energy and cost savings for data-intensive workloads. Before PIM hardware is widely available, realizing these savings requires accurate simulators for prototyping and evaluating PIM software stacks. Building such simulators, however, incurs a recurring engineering cost: every new simulator must independently reconstruct the same host-side memory interface, device discovery, and control-dispatch infrastructure before any device-specific behaviour can be studied. We present HostPIMSim, a userspace framework that eliminates this recurring cost by providing a reusable substrate for exposing PIM simulators to the host stack through standard interfaces. To validate both the functional and economic claims, we implement a HostPIMSim-based functional simulator for the UPMEM architecture and evaluate it against the official UPMEM functional simulator on the 16 Processing-In-Memory (PrIM) benchmarks. The target-specific frontend requires only around 3,300 lines of host-interface glue on top of the reusable substrate. The simulator matches or surpasses the official UPMEM functional simulator in execution time for most benchmarks, while reducing host CPU utilization and package energy by up to 50% on transfer-intensive workloads. Together, these results show that standardizing the host-interface layer reduces both the engineering cost of PIM simulator development and the energy overhead of simulation-driven workload evaluation.

**Keywords:** Processing-in-Memory · Hardware-Software Co-Design · Energy-Aware Computing · System Simulation · Device Virtualization · Memory-Mapped I/O · Ecosystem Economics

## 1 Introduction

PIM reduces data movement by placing computation near the data it consumes, offering significant potential for energy savings and reduced infrastructure cost in workloads bottlenecked by processor-centric memory hierarchies [11,4]. As data-intensive computing scales across cloud and edge infrastructures, the energy and operational cost of memory bandwidth is an increasingly important economic concern. In practice, evaluating and optimizing PIM software stacks requires

simulation: devices are not yet widely available or cost-effective to deploy at scale. Commercial and research PIM systems expose this capability through host runtimes and simulators that let application code target either a simulator or, where available, physical hardware. UPMEM is a representative case with its DRAM-integrated Data Processing Units (DPUs), host Software Development Kit (SDK), and PrIM workloads providing concrete reference points for host-to-device data movement and DPU execution [4].

Simulation is therefore the primary vehicle for evaluating PIM software stacks, but building a useful simulator carries its own engineering cost. A host-memory-type PIM simulator must reproduce not only DPU execution, but also device discovery, memory mapping, control writes, and completion observation. On Linux, this boundary is commonly expressed through device files, `mmap()`, and shared mappings [8]. Developing a PIM simulator therefore requires re-implementing this plumbing from scratch, consuming significant engineering effort before any device-specific behaviour can be studied. This recurring cost raises the barrier to entry for PIM ecosystem development, slowing the research and industry adoption that would allow PIM's energy efficiency advantages to be realized in production infrastructures.

HostPIMSim addresses this fixed engineering cost. It is a framework for building host-side PIM simulators, not a simulator for one target. As shown in Fig. 1, it provides reusable blocks that cover the full host-side surface: mapping device memory into the host address space, partitioning it into RAM and control regions, exposing both through standard host interfaces, detecting writes to Memory-Mapped Input/Output (MMIO)-style controls, and dispatching DPU handlers. The device author supplies the target memory layout, control protocol, and execution model; HostPIMSim supplies the host-visible memory and virtualization substrate. By eliminating redundant re-implementation of host-interface infrastructure, HostPIMSim lowers the economic cost of PIM simulator development and, as the evaluation demonstrates, does so without sacrificing – and often while improving – the energy efficiency of simulation-driven workload evaluation.

**Related Work** Prior work on PIM simulation spans general-purpose architecture simulators, full-system and modular PIM frameworks, trace-driven tools, and hardware emulation platforms. We survey the most relevant examples and show that, despite this breadth, no simulator framework addresses the recurring cost of rebuilding the host-side device interface that any host-memory-type PIM simulator requires.

*General-purpose architecture and memory simulators.* Computer-architecture simulators such as gem5 provide broad full-system and microarchitectural modelling support [3], while memory-system simulators such as Ramulator focus on fast DRAM timing models [9]. They are useful for design-space exploration, but a host-side PIM simulator built on top of them still needs a Linux-visible device-interface layer, and neither addresses the recurring engineering cost of

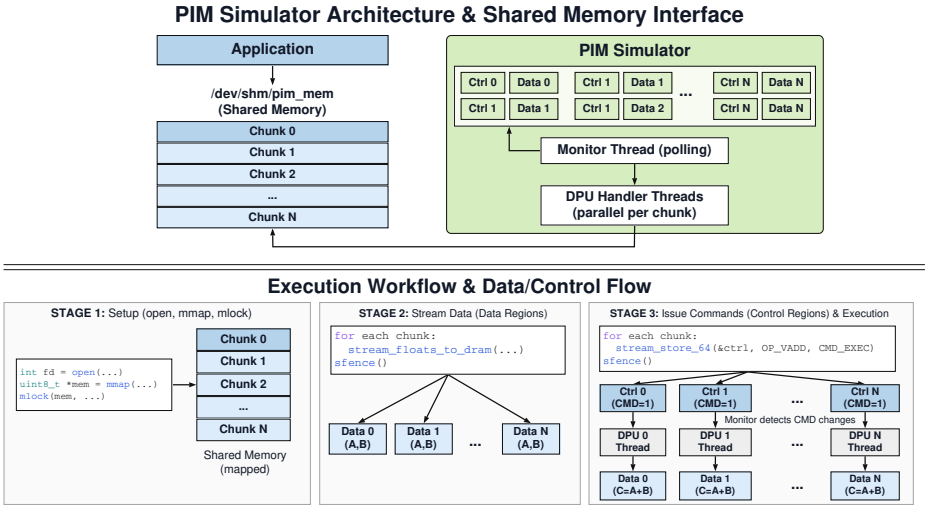


**Fig. 1.** HostPIMSim building blocks: device memory window, RAM and control region partitioning, standard host interface, write detection, and DPU-handler dispatch.

rebuilding that layer for each new target. NVMain [12] is an architectural-level cycle-accurate memory simulator targeting emerging non-volatile technologies (ReRAM, STT-RAM, PCM) as well as DRAM. It models channels, ranks, banks, and subarrays with configurable timing parameters and has become a foundational backend for several downstream PIM frameworks, but it provides no host-device interface substrate.

*Full-system and modular PIM simulators.* Several simulators embed PIM modelling inside gem5 or a gem5-derived full-system stack. SMCSim [2] models near-memory computation inside a Smart Memory Cube by augmenting gem5 with serial-link modelling, DMA engines, and SMC-specific device logic. NDPmulator [16] extends gem5 with a Programming Interface and a Load/Store Unit to attach Near-Data Accelerators at any cache or DRAM level, supporting both syscall-emulation and full-system modes. PIMSim [17] is a more comprehensive framework that integrates multiple timing backends (DRAMSim2, HMC-Sim, NVMain) and offers three operational modes: a fast trace-driven mode, an instrumentation-driven mode using Intel Pin, and a full gem5-based system mode with MESI coherence. In all three cases the host-visible boundary — device discovery, memory mapping, control dispatch — must be reconstructed inside each framework from scratch. MultiPIM [18] similarly combines a ZSim frontend with a Ramulator backend to model large multi-stack PIM systems with directory-based coherence and virtual memory, again building its own host-interface layer independently. These frameworks tackle important architectural questions, but every new project in this family pays the same fixed engineering cost before any device-specific behaviour can be studied.

*Trace-driven and modular simulators.* ZSim+Ramulator [5] extends the Ramulator DRAM simulator with cycle-accurate near-memory logic cores driven by traces produced by a modified ZSim frontend. M$^2$NDP [6] is built on Ramulator and models Near-Data Processing inside CXL memory expanders in a trace-driven fashion. PIMeval [14], together with its PIMbench suite, provides a lightweight functional and analytical simulator for digital DRAM-based PIM architectures, abstracting three design styles (bit-serial, Fulcrum-style bit-parallel, and bank-level) through a unified Application Programming Interface (API). PIMSimulator [13] models in-DRAM execution for HBM2-class systems at bank granularity using DRAMSim2 as a timing backend. None of these frameworks expose a reusable, standard host interface; each defines its own application-facing API that forces benchmark code onto a simulator-specific path.

*UPMEM-specific simulators.* The UPMEM SDK functional simulator runs programs without installed hardware, but the documentation states that it is not cycle accurate and should not be used for reliable cycle counts [15]. uPIMulator [7] is a cycle-level performance simulator for UPMEM-style PIM; its flow compiles UPMEM-PIM source into machine-level instructions consumed by the simulator. This is valuable for architecture characterization, but the PIM binary must be rebuilt through a dedicated toolchain and run through a dedicated runtime pipeline.

*The unaddressed host-interface concern.* A recent survey of PIM simulation frameworks shows a broad landscape of independently developed tools spanning full-system, modular, trace-driven, functional, and emulation-oriented approaches [1]. Across this landscape, we observe that host-side device-interface infrastructure remains largely treated as target-specific implementation work: surveyed tools do not factor out the device memory window, region metadata, write detection, virtual-device nodes, and handler dispatch that host-memory-type simulators must provide before modelling the device itself. HostPIMSim is complementary to these simulators: it factors out exactly this host-side plumbing so that simulator authors can focus on the command protocol and execution model of their target device, reducing the recurring engineering cost across the PIM ecosystem.

**Contributions** HostPIMSim exposes the common substrate of PIM simulation as reusable blocks, lowering the amortized cost per simulator: the host-interface substrate is written once and reused, while each target contributes only its device-specific execution model. We use UPMEM as a realistic exercise because its public software stack, functional simulator, and PrIM workloads provide a concrete comparison point [4,15]. This case study is not the central claim: it demonstrates that the reusable substrate can support a nontrivial runtime and that doing so yields measurable energy efficiency benefits, while other host-memory-type PIM targets can reuse the same plumbing with their own memory maps and execution models. Other commercial PIM products, such as Samsung's

**Table 1.** Core `libhostpimsim` memory region types.

| Region Type | Description | DPU Trigger |
|---|---|---|
| `PIM_REGION_RAM` | Regular host-visible memory bank. Reads and writes proceed without triggering handlers. | No |
| `PIM_REGION_CTRL_MMIO` | MMIO-style control registers. Host writes to these offsets trigger DPU handlers. | Yes |
| `PIM_REGION_CTRL_RW` | Bidirectional control registers. Simulator code publishes status or response words for host-side reads. | No |

HBM-PIM [10], expose similar host-side boundaries and would benefit from the same reusable substrate.

This paper makes the following contributions.

- We identify host-side memory and device-interface plumbing as a reusable part of host-memory-type PIM simulators, and quantify the engineering cost it represents for new simulator projects.
- We present HostPIMSim, a userspace framework with arbitrary device regions, standard memory-mapped access, MMIO handlers, adaptive polling, page-protection write detection, and bounded dispatch.
- We provide virtual device and sysfs support so existing runtimes can reach a simulator through Linux-facing interfaces instead of a simulator-specific application API.
- We validate the framework with a HostPIMSim-based UPMEM functional simulator and compare it with the official UPMEM functional simulator on PrIM workloads, showing that preserving a standard host interface does not require sacrificing performance or energy efficiency, and often improves both.

## 2 Design and Implementation

HostPIMSim is a userspace substrate for host-memory-type PIM simulators. It owns the host-visible aperture, region metadata, write-detection frontends, virtual file interface, and handler scheduling; the target simulator owns the memory map, control protocol, and DPU execution or timing model.

### 2.1 Building Blocks

The central object is a `pim_device_t`: one contiguous aperture created as POSIX shared memory, attached from an existing file or device mapping, or wrapped around already-mapped memory. Regions are views into this aperture, so HostPIMSim imposes no rank, bank, chip, or page layout.

Table 1 lists the core region types and whether each can trigger DPU handlers.

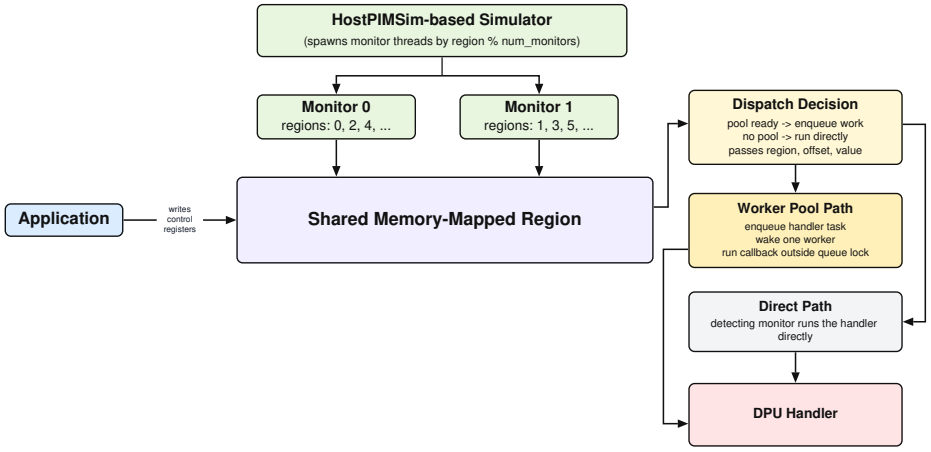


**Fig. 2.** Shared memory and polling monitor model.

Handler registration records an offset, a 1-, 2-, 4-, or 8-byte register width, an optional depth for explicit nested dispatch, and a simulator callback. The same dispatch path is used for polling, page-protection faults, and virtual-device callbacks. Handlers can run synchronously in the detecting context or through a per-device worker pool; device and region barriers wait for submitted work. Region topology and handler-table updates are rejected while monitoring is active, keeping hot-path lifetime rules explicit.

### 2.2 Shared Memory and Polling Monitor Model

As shown in Fig. 2, the shared-memory model creates or attaches the aperture, defines RAM/control regions, registers handlers, and lets a host process access the same bytes through ordinary `open()`, `mmap()`, `mlock()`, `munmap()`, and `close()` operations [8]. Data movement becomes stores to RAM regions, and command issue becomes stores to MMIO-style controls.

The monitor snapshots and shards registered `PIM_REGION_CTRL_MMIO` offsets, then scans declared-width control words against cached values and dispatches changed entries after each scan. Adaptive polling spins after activity and backs off during idle periods; the next detected write restores the hot budget.

Polling is portable across processes that map the aperture, but because it samples at monitor granularity, repeated writes to one offset between scans collapse to the final transition; strict per-store registers should use the fault-based or virtual-device paths.

### 2.3 LD_PRELOAD-Based SIGSEGV Handler Model

For lower idle overhead and faster in-process detection, HostPIMSim can use page protection as shown in Fig. 3: a preload library installs the global `SIGSEGV`

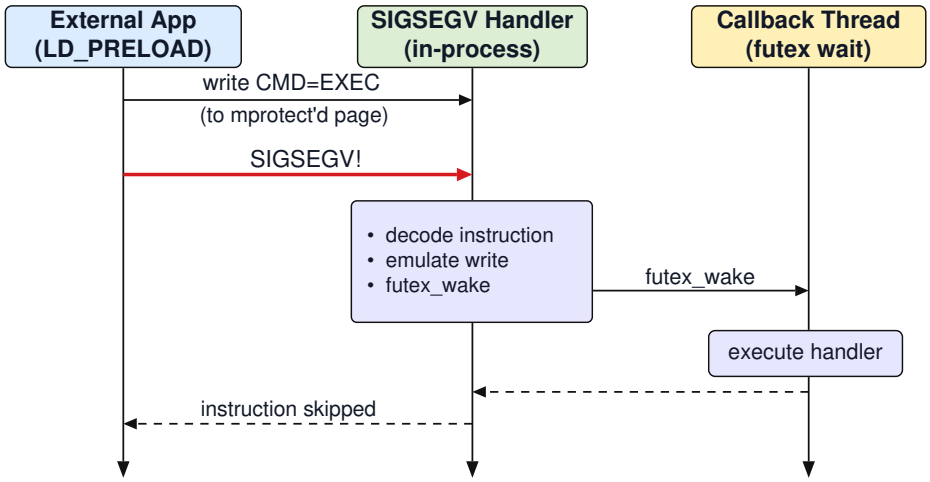


**Fig. 3.** LD_PRELOAD-based SIGSEGV handler model.

helper, enables a device, creates callback workers, and marks pages covering `PIM_REGION_CTRL_MMIO` regions read-only with `mprotect()`. Because protection is page-granular, control regions should be aligned to the target layout.

A write to a protected control word raises `SIGSEGV`; the handler validates the fault, decodes supported stores, extracts value and width, advances the program counter, and chains misses or unsupported instructions to the previous signal path. It performs only async-signal-safe work by recording region, offset, and value, then waking a futex-backed worker that invokes the common DPU dispatch path, optionally through the worker pool.

This path intentionally does not commit writes to `PIM_REGION_CTRL_MMIO` memory; the value is passed directly to the handler, avoiding duplicate detection if polling is also active elsewhere. Writes to `PIM_REGION_CTRL_RW` can be replayed for host-visible status. The cost is dependence on `LD_PRELOAD`, signal installation, supported instruction decoding, and controlled enable/disable phases.

### 2.4 LD_PRELOAD-Based Virtual PIM Device Model

Some runtimes reach PIM through `/dev`, `/sys`, `ioctl()`, character-device mappings, fd duplication, and sysfs reads before any DPU work starts. They may also encode control data in ways that are hard to reconstruct from final memory bytes. The virtual-device model intercepts this boundary before the kernel driver path, as shown in Fig. 4.

The syscall hook calls `_pim_vsyscall_dispatch()` for candidate syscalls. The dispatcher resolves virtual device, sysfs, and module paths; binds fds for virtual `/dev` sessions; routes later fd-owned syscalls; and synthesizes sysfs/module entries, attributes, stat results, reads, writes, and seeks. This preserves existing runtime file and descriptor lifecycles without requiring a kernel module.

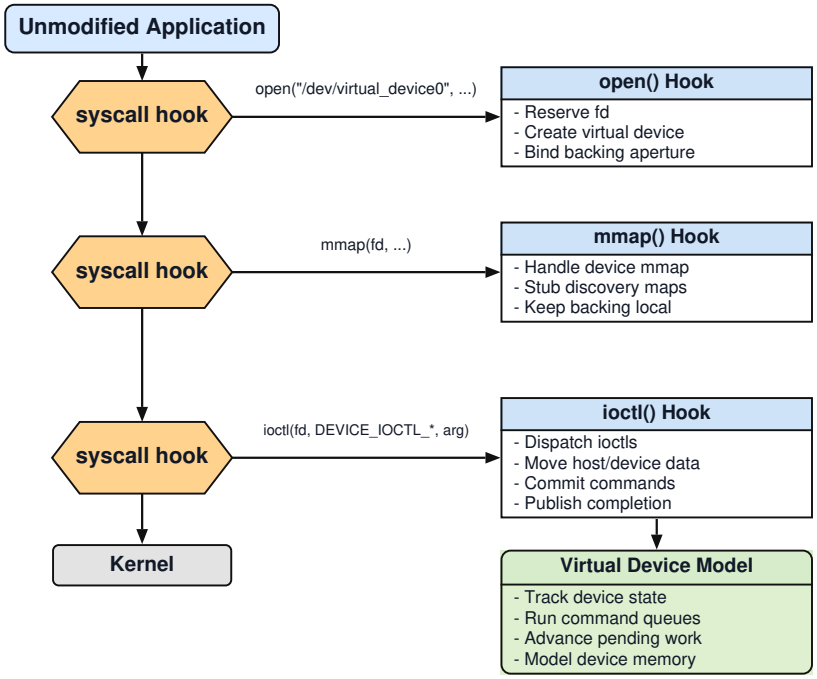


**Fig. 4.** LD_PRELOAD-based virtual PIM device model.

The target frontend implements virtual open, `ioctl()`, `mmap()`, and close callbacks that create or find a `pim_device_t`, expose an aperture, translate driver commands, or directly invoke decoded handlers. Our UPMEM frontend uses this path for virtual nodes, sysfs metadata, rank apertures, and SDK ioctls. Target code keeps UPMEM Control Interface (CI), MRAM, IRAM, WRAM, and DPU semantics. A different host-side PIM target can reuse the same layer with different device names, memory maps, or command protocols.

## 3 Experimental Evaluation

### 3.1 Experiment Environment and Methodology

We compare a preloadable HostPIMSim-based UPMEM functional simulator with `libdpufsim.so`, the official UPMEM SDK functional simulator, on the 16 PrIM programs: BFS, BS, GEMV, HST-L, HST-S, MLP, NW, RED, SCAN-RSS, SCAN-SSA, SEL, SpMV, TRNS, TS, UNI, and VA. These cover graph traversal, search, dense and sparse linear algebra, histograms, prefix scans, sequence alignment, selection, compaction, and vector arithmetic.

The testbed is one physical node with an AMD Ryzen 9 9950X3D, 128 GB of DDR5-5600 ECC memory, and a 2 TB WD_BLACK SN8100 NVMe SSD,

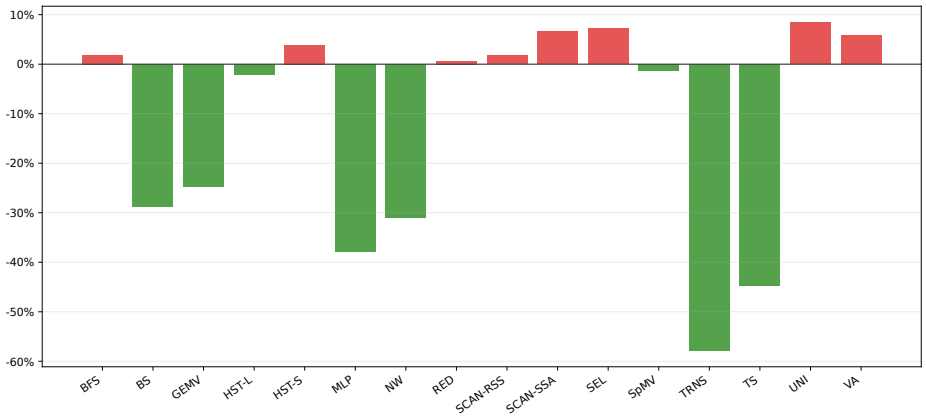


**Fig. 5.** Execution-time difference of HostPIMSim relative to `libdpufsim.so` on PrIM.

running Debian GNU/Linux testing (forky/sid) with Linux 7.0.8. All runs execute inside an Ubuntu 22.04 LTS Podman container, matching the distribution supported by the official UPMEM 2025.1.0 SDK.

Each benchmark runs once under each simulator with 16 DPUs and 16 tasklets per DPU; the DPU count matches the host's 16 physical cores because larger counts caused `libdpufsim.so` faults on NW. The official run uses `UPMEM_PROFILE="backend=simulator"`, while HostPIMSim uses `LD_PRELOAD` with `libhostpimsim-upmem.so` and `UPMEM_PROFILE="backend=hw,regionMode=safe"`. We measure execution time, CPU utilization, and system power with turbostat and Tapo P110M, omitting memory use because it showed no meaningful difference; energy is included because repeated simulation runs add non-trivial overhead to adoption studies.

### 3.2 Experimental Results

Fig. 5 reports HostPIMSim's execution-time difference relative to `libdpufsim.so`, where negative values mean HostPIMSim is faster. HostPIMSim is faster on BS, GEMV, HST-L, MLP, NW, SpMV, TRNS, and TS, especially TRNS, TS, MLP, NW, BS, and GEMV; the remaining benchmarks stay within about 9% of `libdpufsim.so`. In the faster cases, lower DPU-kernel execution cost outweighs host-side virtualization overhead.

Fig. 6 shows generally lower host CPU use from shorter HostPIMSim runs, except that TS uses slightly higher average CPU while still finishing substantially faster.

Fig. 7 reports package and wall energy: package energy follows execution-time reductions on TRNS, NW, TS, MLP, BS, GEMV, and HST-L, while wall energy follows the same direction for longer workloads but is noisier for short benchmarks due to one-second sampling and fixed measurement overhead.

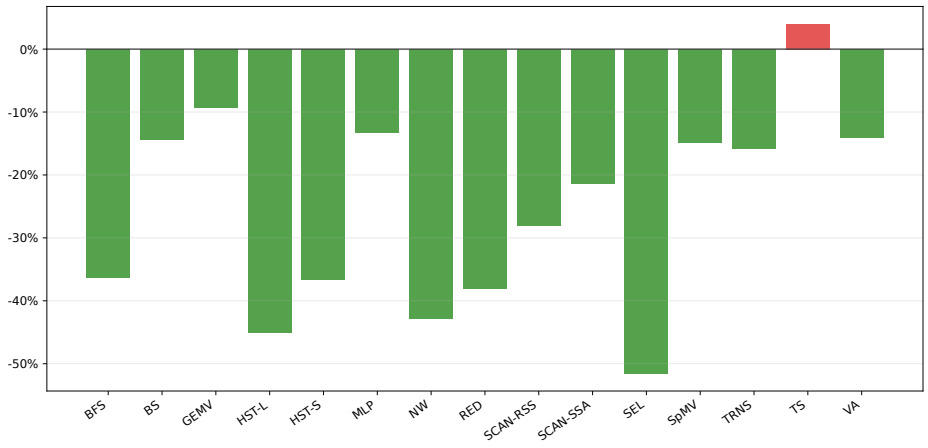


**Fig. 6.** Host CPU utilization of HostPIMSim relative to `libdpufsim.so` on PrIM.

**Table 2.** Source responsibility breakdown of the UPMEM case study.

| Component | Files | LOC | Role |
|---|---|---|---|
| HostPIMSim substrate | 13 | 12.6K | Reusable host-interface substrate |
| UPMEM virtual-device glue | 5 | 0.4K | Virtual `/dev` and sysfs nodes |
| UPMEM rank/ioctl frontend | 2 | 2.0K | UPMEM SDK calls to regions and handlers |
| UPMEM CI protocol | 2 | 0.9K | Target-specific command-interface protocol |
| UPMEM DPU model | 7 | 16.9K | Target-specific ISA, memory, and execution |

The UPMEM case study also marks the framework/target boundary: `libhostpimsim-upmem.so` builds on `libhostpimsim`, which supplies the aperture, region metadata, virtual device/sysfs exposure, write-to-handler dispatch, worker pool, and barriers. The UPMEM library supplies the command protocol and DPU model; Table 2 shows that about 84% of `libhostpimsim-upmem.so` is the DPU model, while the host-facing frontend is much smaller and mostly calls HostPIMSim APIs.

Data-layout translation can dominate transfer-heavy PIM workloads because practical DIMM-attached devices distribute data across chips and banks rather than exposing each DRAM chip as a linear stream. Reproducing this interleaving in software on every transfer is expensive, so future host-memory-type PIM hardware should hide placement details, for example with LR-DIMM-like module-side logic that accepts contiguous transfers and performs chip/bank interleaving internally. This avoids a recurring simulation and deployment cost that erodes the energy-efficiency advantage motivating PIM adoption.

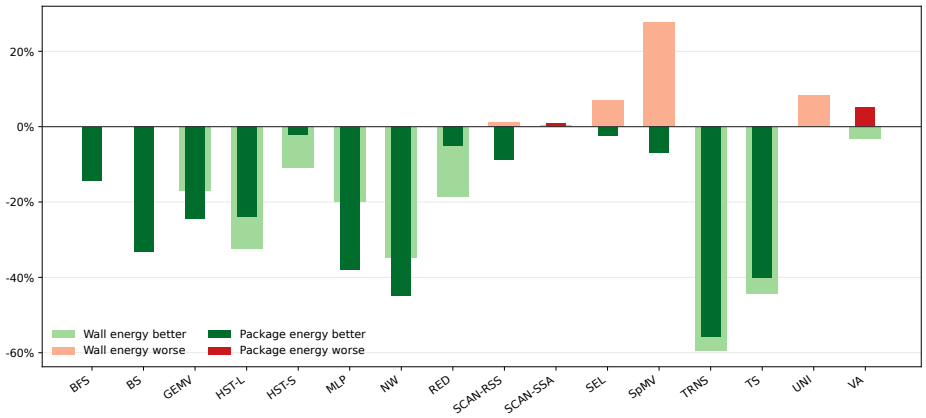


**Fig. 7.** Package and wall energy of HostPIMSim relative to `libdpufsim.so` on PrIM.

**Economic Cost Estimate** Nominal COCOMO II post-architecture effort is $2.94 \times \text{KSLOC}^{E}$ person-months with $E \approx 1.10$, giving about **48 person-months** for the 12.6-KSLOC host-interface substrate in Table 2. The recent PIM simulation survey identifies over twenty simulator and emulator projects for varied PIM and near-data targets [1]; if projects in this landscape independently rebuild an equivalent host-interface layer, the aggregate cost exceeds **950 person-months**, or about 80 engineer-years. These approximate figures illustrate the economic case for reusable simulation infrastructure rather than precise project measurements.

## 4 Conclusion

This paper presented HostPIMSim, a framework that separates reusable host-side plumbing from target-specific DPU behaviour, so that the 12,600-line substrate need not be rebuilt for each new simulator. This directly lowers the engineering barrier to PIM ecosystem entry. Standardizing the host-interface layer does not require sacrificing performance or energy efficiency, and often improves both. The UPMEM case study confirms this: the HostPIMSim-based simulator matches or surpasses the official UPMEM functional simulator on the PrIM benchmarks, with host CPU utilization and package energy reductions reaching up to 50% on transfer-intensive workloads. The results also surface a broader design concern: software chip/bank interleaving can dominate transfer-heavy workloads, offsetting part of PIM's energy advantage. Future devices should hide such placement details from the host interface, for example, with module-side logic that accepts contiguous transfers and performs interleaving internally. By making host-interface costs explicit and measurable, HostPIMSim provides a foundation for evaluating such trade-offs and lowering the overall economic cost of PIM adoption.

**Acknowledgments.** We thank the anonymous reviewers for their helpful feedback. This work was funded, in part, by the Korean National Research Foundation (grant no. 21A20151113068 – BK21 Plus for Pioneers in Innovative Computing - Dept. of Computer Science & Engineering, SNU), by the Ministry of Trade, Industry and Energy (MOTIE) and the Korea Evaluation Institute of Industrial Technology (KEIT) (grant no. 10077609), by the Ministry of Science and ICT (MSIT) (grant no. RS-2023-00302083), and by the EU Horizon program within the f-inference project (grant no. 101298393). ICT at Seoul National University provided research facilities for this study.

**Disclosure of Interests.** The authors have no competing interests to declare that are relevant to the content of this article.

## References

1. Aghaei, M., Ebrahimi, S., Arafati, M.S., Cheshmikhani, E., Rahmati, D., Gorgin, S., Kim, J.: Modeling and Simulation Frameworks for Processing-in-Memory Architectures. CoRR **abs/2512.00096** (2025), https://doi.org/10.48550/arXiv.2512.00096
2. Azarkhish, E., Rossi, D., Loi, I., Benini, L.: Design and Evaluation of a Processing-in-Memory Architecture for the Smart Memory Cube. In: Proceedings of the 29th International Conference on Architecture of Computing Systems – ARCS 2016 - Volume 9637. p. 19–31. Springer-Verlag, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-319-30695-7_2, https://doi.org/10.1007/978-3-319-30695-7_2
3. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (Aug 2011). https://doi.org/10.1145/2024716.2024718, https://doi.org/10.1145/2024716.2024718
4. Gómez-Luna, J., Hajj, I.E., Fernandez, I., Giannoula, C., Oliveira, G.F., Mutlu, O.: Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. IEEE Access **10**, 52565–52608 (2022). https://doi.org/10.1109/ACCESS.2022.3174101, https://doi.org/10.1109/ACCESS.2022.3174101
5. Gómez-Luna, J., Oliveira, G.F.: ZSim+Ramulator - A Processing-in-Memory Simulation Framework. https://github.com/CMU-SAFARI/ramulator-pim (2020), GitHub repository, accessed: 2026-06-15
6. Ham, H., Hong, J., Park, G., Shin, Y., Woo, O., Yang, W., Bae, J., Park, E., Sung, H., Lim, E., Kim, G.: Low-Overhead General-Purpose Near-Data Processing in CXL Memory Expanders. In: 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 594–611 (2024). https://doi.org/10.1109/MICRO61859.2024.00051
7. Hyun, B., Kim, T., Lee, D., Rhu, M.: Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology. In: 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA). pp. 263–279 (2024). https://doi.org/10.1109/HPCA57654.2024.00029
8. Kerrisk, M.: mmap(2) - Linux Manual Page (2025), https://man7.org/linux/man-pages/man2/mmap.2.html, linux man-pages 6.16; accessed: 2026-06-01

9. Kim, Y., Yang, W., Mutlu, O.: Ramulator: A Fast and Extensible DRAM Simulator. IEEE Computer Architecture Letters **15**(1), 45–49 (2016). https://doi.org/10.1109/LCA.2015.2414456
10. Lee, S., Kang, S.h., Lee, J., Kim, H., Lee, E., Seo, S., Yoon, H., Lee, S., Lim, K., Shin, H., Kim, J., Seongil, O., Iyer, A., Wang, D., Sohn, K., Kim, N.S.: Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product. In: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). pp. 43–56 (2021). https://doi.org/10.1109/ISCA52012.2021.00013
11. Mutlu, O., Ghose, S., Gómez-Luna, J., Ausavarungnirun, R.: A Modern Primer on Processing in Memory. CoRR **abs/2012.03112** (2020), https://arxiv.org/abs/2012.03112
12. Poremba, M., Xie, Y.: NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories. In: Proceedings of the 2012 IEEE Computer Society Annual Symposium on VLSI. p. 392–397. ISVLSI '12, IEEE Computer Society, USA (2012). https://doi.org/10.1109/ISVLSI.2012.82, https://doi.org/10.1109/ISVLSI.2012.82
13. Samsung Advanced Institute of Technology: PIMSimulator. https://github.com/SAITPublic/PIMSimulator (2025), GitHub repository, accessed: 2026-06-15
14. Siddique, F.A., Guo, D., Fan, Z., Gholamrezaei, M., Baradaran, M., Ahmed, A., Abbot, H., Durrer, K., Nandagopal, K., Ermovick, E., Kiyawat, K., Gul, B., Mughrabi, A., Venkat, A., Skadron, K.: Architectural Modeling and Benchmarking for Digital DRAM PIM. In: 2024 IEEE International Symposium on Workload Characterization (IISWC). pp. 247–261 (2024). https://doi.org/10.1109/IISWC63097.2024.00030
15. UPMEM SAS: Installing the UPMEM DPU Toolchain: Functional Simulator (2025), https://sdk.upmem.com/2025.1.0/01_Install.html#functional-simulator, accessed: 2026-06-01
16. Vieira, J., Roma, N., Falcao, G., Tomás, P.: NDPmulator: Enabling Full-System Simulation for Near-Data Accelerators From Caches to DRAM. IEEE Access **12**, 10349–10365 (2024). https://doi.org/10.1109/ACCESS.2024.3352924
17. Xu, S., Chen, X., Wang, Y., Han, Y., Qian, X., Li, X.: PIMSim: A Flexible and Detailed Processing-in-Memory Simulator. IEEE Computer Architecture Letters **18**(1), 6–9 (2019). https://doi.org/10.1109/LCA.2018.2885752
18. Yu, C., Liu, S., Khan, S.: MultiPIM: A Detailed and Configurable Multi-Stack Processing-In-Memory Simulator. IEEE Computer Architecture Letters **20**(1), 54–57 (2021). https://doi.org/10.1109/LCA.2021.3061905