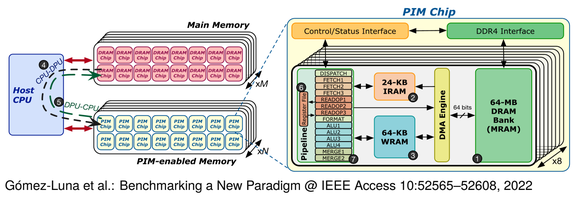
Overhead Prediction for PIM-Enabled Applications with Performance-Aware Behaviour Models

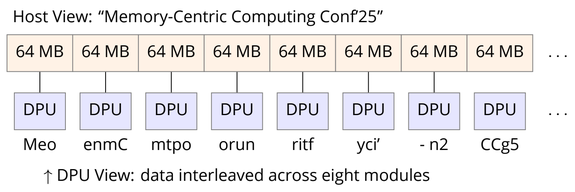
UNIVERSITÄT OSNABRÜCK

embedded software systems

Background: Processing in Memory with UPMEM PIM



Gómez-Luna et al.: Benchmarking a New Paradigm @ IEEE Access 10:52565–52608, 2022



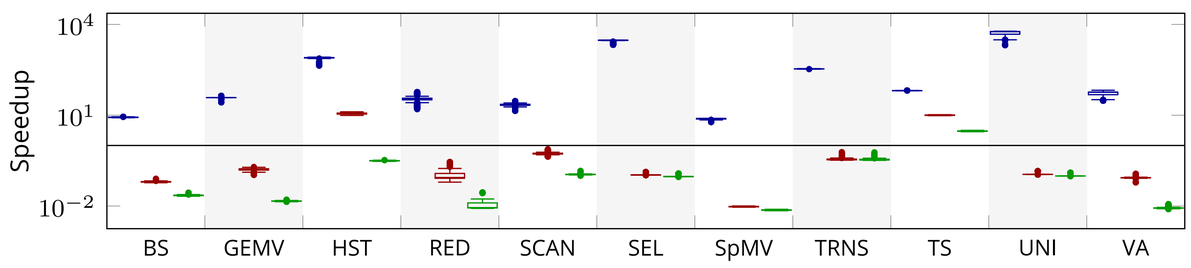
↑ DPU View: data interleaved across eight modules

- UPMEM PIM: DRAM modules with built-in DRAM Processing Units (DPUs): 32-bit RISC @ ≈ 400 MHz, SPMD in hardware, limited ALU (add/sub only)
- Exclusive 64 MiB partitions per DPU, **no sharing** (→ synchronization via CPU)
- Up to 2,560 DPUs (40 ranks) total; commercially available from 2021 to 2025

\+ Up to $10^3\times$ speedup of "embarrassingly parallel" compute kernels

− DDR **interleaving** mandates costly data transformation; jeopardizes speedup

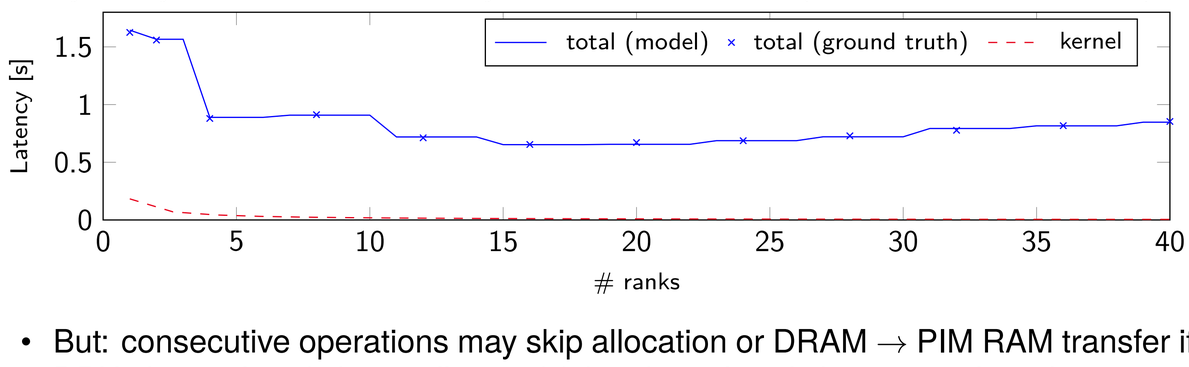
⇒ UPMEM PIM ≙ offloading engine, not true in-memory computing



Friesel et al.: A Full-System Perspective on UPMEM Performance @ 1st Workshop on Disruptive Memory Systems (DIMES), 2023

Motivation: Model-Guided Offloading Decisions for PIM

- Workflow: allocate DPUs, upload DPU binary, write data to PIM RAM, run kernel, read data from PIM RAM; repeat data transfers and kernel invocations as needed
- Functions provided by UPMEM SDK, updated ≈ every half year
- Typically > 99 % of PIM latency from SDK overhead (allocation, data transfer)



- But: consecutive operations may skip allocation or DRAM → PIM RAM transfer if DPUs have already been allocated / data has already been transferred
- How to find out which workloads / problem sizes actually benefit from PIM?

→ **Predict SDK calls and per-call latency for arbitrary workloads**

Contributions

① **Tracing with AspectC++**: obtain workload-specific SDK call traces
  - Supports arbitrary C/C++ software; similar methods exist for other languages

② **Behaviour Models**: learn to predict SDK calls for arbitrary application runs
  - Simulator traces are sufficient; (cycle-accurate) timing not required at this step

③ **Interpretable Performance Models**: predict and understand SDK performance
  - Based on microbenchmarks or traces on real hardware

- All contributions are also applicable to non-UPMEM / non-PIM SDKs

① Tracing and Benchmark Data Acquisition

AspectC++: Source code ↦ annotated execution traces for behaviour model learning

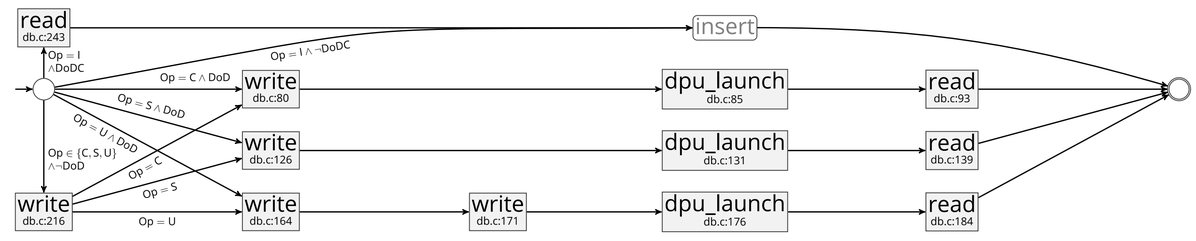
```
aspect GenericUpmemSdkTracing {
  // Corresponding application code:
  // dpu_prepare_xfer(dpu, host_pointer);
  // dpu_push_xfer(dpu_set, DPU_XFER_DIRECTION, dpu_addr, host_offset, size);
  advice call("% dpu_push_xfer(...)") : around() {
    size_t payload_size = *(tjp->arg<4>());
    gettimeofday(&start, NULL); tjp->proceed(); gettimeofday(&stop, NULL); // run dpu_push_xfer
    double time_us = (stop.tv_sec - start.tv_sec) * 1000000.0 + (stop.tv_usec - start.tv_usec);
    if (*(tjp->arg<1>()) == DPU_XFER_TO_DPU) {
      printf("[::] push_to_dpu @ %s:%d | n_dpus=%u n_ranks=%u payload_B=%lu"
        " | latency_us=%f throughput_MBps=%f\n",
        tjp->filename(), tjp->line(), n_dpus, n_ranks, payload_size * n_dpus
        time_us, payload_size * n_dpus / time_us
      );
    } /* else if DPU_XFER_FROM_DPU ... */
  }
};
```

- Requires source code access; similar approaches available for other languages
- Trace output: workload attributes ↦ location and arguments of SDK calls
- Simple simulator suffices; no PIM hardware or cycle-accurate simulation needed

```
[::] dpu_alloc_ranks @ host/app.c:292 | n_ranks=20 | latency_us=76533
[::] dpu_load @ host/app.c:296 | n_ranks=20 | latency_us=2893
[>>] DBMS | n_ranks=20 n_elements=1048576 e_op=count b_data_on_dpus=0 b_data_on_dpus_changed=0
[::] push_to_dpu @ host/app.c:216 | n_dpus=1274 n_ranks=20 ... | latency_us=2153 throughput_MiBps=3756
[::] push_to_dpu @ host/app.c:80 | n_dpus=1274 n_ranks=20 ... | latency_us=481 throughput_MiBps=60.622
[::] dpu_launch @ host/app.c:85 | n_dpus=1274 n_ranks=20 e_kernel=count | latency_us=398
[::] push_from_dpu @ host/app.c:93 | n_dpus=1274 n_ranks=20 ... | latency_us=221 throughput_MiBps=43.9
[<<] DBMS | n_ranks=20 n_elements=1048576 e_op=count b_data_on_dpus=0 b_data_on_dpus_changed=0
```

② Application Models for Runtime Behaviour

Simplified example: DBMS, INSERT on CPU, COUNT/SELECT/UPDATE on PIM
Workload parameters: **Op**eration ∈ {**C**OUNT, **I**NSERT, **S**ELECT, **U**PDATE};
Data on DPUs: **DoD** ∈ $\{0,1\}$; Data on DPUs changed: **DoDC** ∈ $\{0,1\}$



$\mathcal{A} = (Q, q_0, q_f, \Delta, P, \gamma, \rho, \lambda)$ — States (≙ API call sites) Q, transitions $\Delta \subseteq Q^2$, workload domain P, transition guards $\gamma : \Delta \rightharpoonup (P \to \{\bot, \top\})$, loop count predictors $\rho : Q \to (P \to \mathbb{N})$, argument predictors $\lambda : Q \to (P \to \mathbb{R}^{n_q})$ ≈̂ control flow graph

Learning: $\mathcal{A}$ determined automatically from (simulation-based) application traces

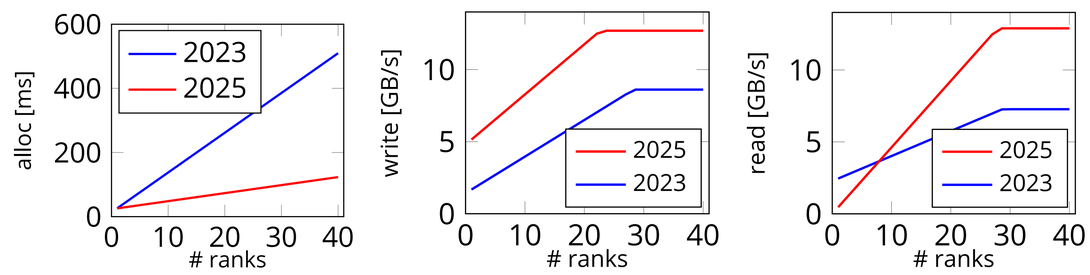
- Guards γ: learn predictor $P \to Q$, annotate each outgoing transition accordingly
- Loop counts ρ: arbitrary ML method, e.g. regression model trees (RMTs)
- Argument predictors λ: arbitrary ML method, e.g. regression model trees

Prediction: Workload specification $\vec{x} \in P$ as input

↦ API call sequence (run through $\mathcal{A}$) with callsite-specific API call args

③ PIM Overhead Prediction

- Annotate $\mathcal{A}$ with one RMT per API call (identical for all call sites): args (predicted by λ) ↦ latency
- Learnt from microbenchmarks or traces on real hardware
- **Interpretable** models → understand runtime performance (e.g. SDK changes) and bottlenecks



- Input: Trace $(q_0, q_1, \ldots, q_n, q_f)$ predicted from workload specification $\vec{x}$ by ②
- Each state q_i is linked to a performance model f_i (note: $f_i = f_j$ if q_i, q_j call the same function)

→ Total latency of SDK function calls: $\sum_{i=1}^{n} f_i(\lambda(q_i)(\vec{x})) \cdot \rho(q_i)(\vec{x})$

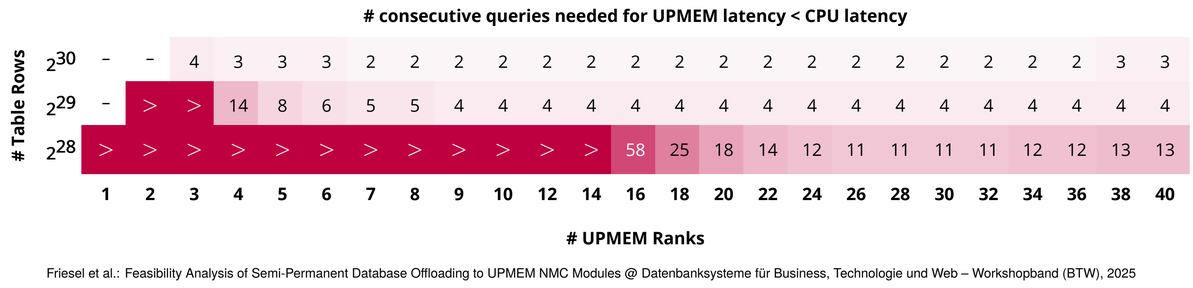
→ **Predict PIM SDK latency for arbitrary workloads** or workload sequences $\vec{x} \in P$

Evaluation

- Proof-of-concept algorithm & implementation evaluated on UPMEM PIM
- DBMS + 11 applications from PrIM suite (vector/matrix ops, data lookups, . . .)
- Behaviour prediction: learn $\mathcal{A}$ via simulator, compare with traces on real hardware
  - states Q, transitions Δ, guards γ, loops ρ: perfectly accurate on 10/12 targets
  - No $\mathcal{A}$ for 2 targets: conditional transitions within loops not supported yet
  - Argument predictors λ: > 85 % of call sites with < 1 % error
  - Issue: simulator uses 1 DPU per rank; real hardware has 64 DPUs per rank
- Predicted latency vs. SDK-specific latency observed on real hardware
  - < 25 % error on 10/12 targets (i.e., all targets where $\mathcal{A}$ is available)
  - < 10 % error on 6/12 targets (≙ error of underlying SDK performance models)

→ Proof of concept successful; next: improve loop support & evaluate on other HW

Application Example: Simulation-Based Placement Decisions



Friesel et al.: Feasibility Analysis of Semi-Permanent Database Offloading to UPMEM NMC Modules @ Datenbanksysteme für Business, Technologie und Web – Workshopband (BTW), 2025

- DBMS on UPMEM PIM: faster than single-core CPU only for multi-GiB columns
- Multi-core CPU execution: PIM only sensible for ≥ 2 **consecutive operations**
- Behaviour model allows for systematic (exhaustive) config space exploration
- Model-driven decisions verified with benchmarks on real hardware

Summary

- Performance-Aware Behaviour Models disentangle workload-dependent application behaviour from SDK / hardware performance
- SDK / hardware changes → update ③, keep behaviour model ② as-is
- Interpretable models explain performance bottlenecks and changes over time (e.g. UPMEM: high allocation and data transfer overhead, SDK 2025 is better)
- Prediction of PIM kernel latency: out of scope; covered by related works

Funded by DFG Deutsche Forschungsgemeinschaft German Research Foundation

Birte Friesel and **Olaf Spinczyk** · {birte.friesel,olaf}@uos.de · ess.cs.uos.de
SMAUG: System-Level Modelling and Optimized Use of Disruptive Memory Technologies @ DFG SPP 2377