OSNABRÜCK UNIVERSITY

Embedded Software Systems

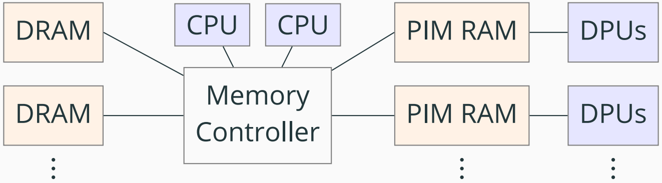
Overhead Prediction for PIM-Enabled Applications with Performance-Aware Behaviour Models

Birte Friesel, Olaf Spinczyk

October 9th, 2025

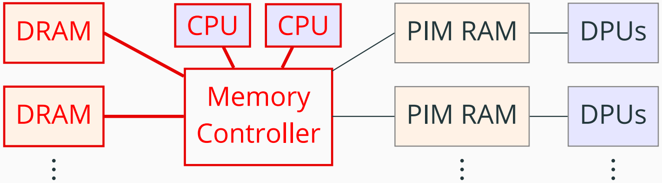
`ess.cs.uos.de/~bf` birte.friesel@uos.de

Processing in Memory



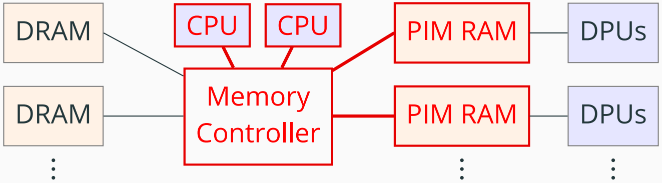
- Processing in Memory (PIM): embed processing into DRAM DIMMs

Processing in Memory



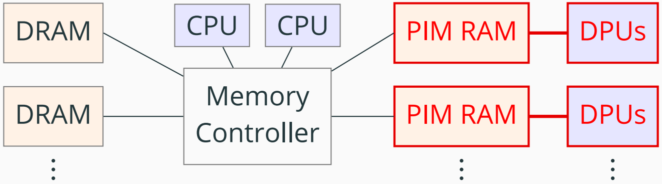
- Processing in Memory (PIM): embed processing into DRAM DIMMs

Processing in Memory



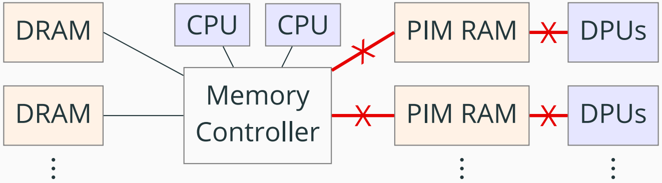
- Processing in Memory (PIM): embed processing into DRAM DIMMs

Processing in Memory



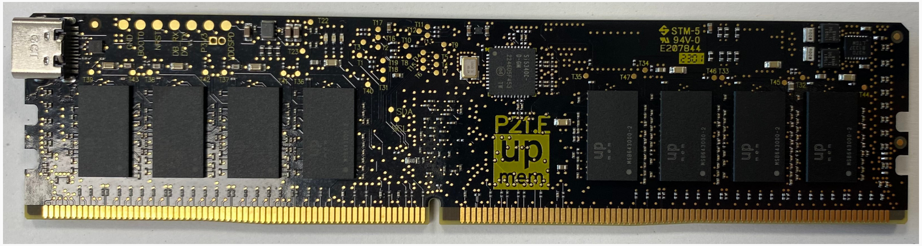
- Processing in Memory (PIM): embed processing into DRAM DIMMs
  - DRAM Processing Units (DPUs) have direct data access
  - Massive parallelism without contention: thousands of DPUs per system

Processing in Memory with UPMEM PIM



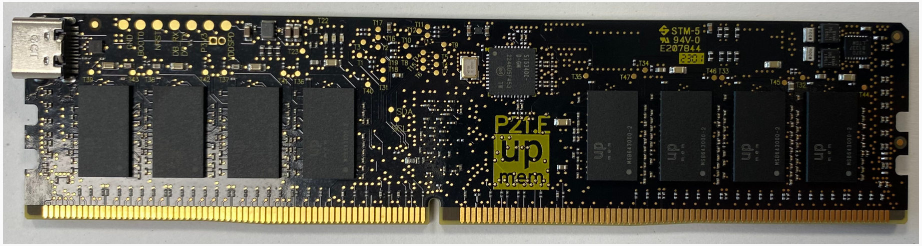
- Processing in Memory (PIM): embed processing into DRAM DIMMs
  - DRAM Processing Units (DPUs) have direct data access
  - Massive parallelism without contention: thousands of DPUs per system
- UPMEM PIM: only commercially available PIM hardware (up to 2560 DPUs)
- Builds upon DDR memory interface; challenging for true PIM usage

UPMEM PIM



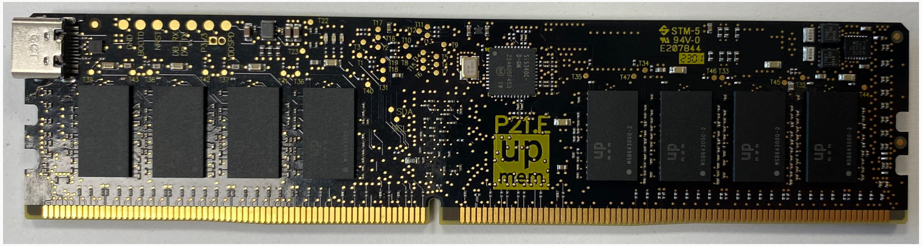
- First (and only) commercially available PIM platform from 2019 to 2025
  - 8 GiB DDR4 module (2 ranks × 4 GiB): 16× (4 Gbit + 8 DPUs) → 128 DPUs total

UPMEM PIM



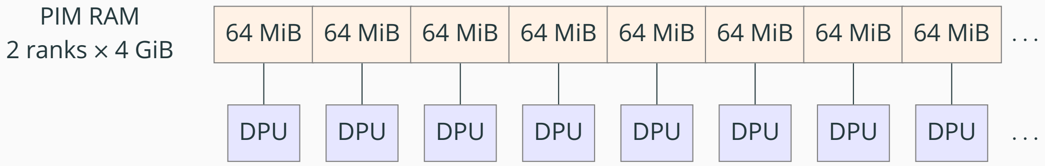
- First (and only) commercially available PIM platform from 2019 to 2025
  - 8 GiB DDR4 module (2 ranks × 4 GiB): 16× (4 Gbit + 8 DPUs) → 128 DPUs total
  - 32-bit RISC @ 267 . . . 450 MHz; one 64 MiB chunk of DRAM per DPU
  - DPU logic built with DRAM process (few, slow transistors)
  - Simple ISA: math limited to 32bit add/sub; no FP or mul/div support

UPMEM PIM



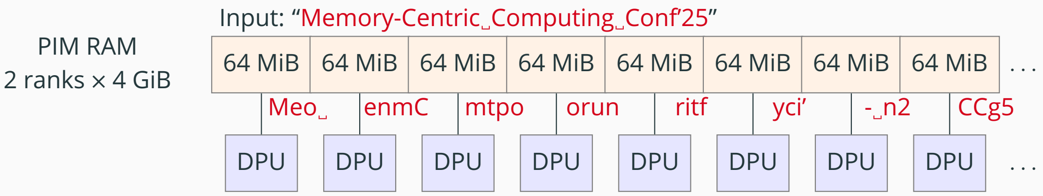
- First (and only) commercially available PIM platform from 2019 to 2025
  - 8 GiB DDR4 module (2 ranks × 4 GiB): 16× (4 Gbit + 8 DPUs) → 128 DPUs total
  - 32-bit RISC @ 267 . . . 450 MHz; one 64 MiB chunk of DRAM per DPU
  - DPU logic built with DRAM process (few, slow transistors)
  - Simple ISA: math limited to 32bit add/sub; no FP or mul/div support

UPMEM PIM: Limitations



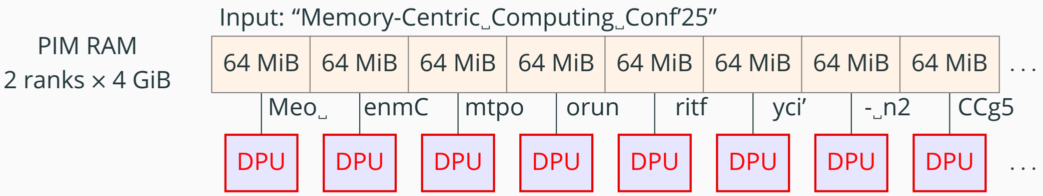
- 64 MiB chunk of DRAM per DPU, no shared memory → architecture ≠ CPU; algorithms need adjustments (unless “embarrassingly parallel”)

UPMEM PIM: Limitations



- 64 MiB chunk of DRAM per DPU, no shared memory → architecture ≠ CPU; algorithms need adjustments (unless "embarrassingly parallel")
- Interleaving → PIM RAM ≠ DRAM; costly scatter/gather required [Dev19]

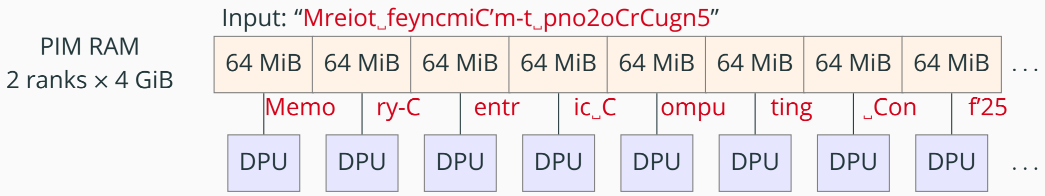
UPMEM PIM: Limitations



- 64 MiB chunk of DRAM per DPU, no shared memory → architecture ≠ CPU; algorithms need adjustments (unless "embarrassingly parallel")
- Interleaving → PIM RAM ≠ DRAM; costly scatter/gather required [Dev19]

→ Applications must use PIM software development kit (SDK): dpu_alloc; write; dpu_load; dpu_launch; read; . . .

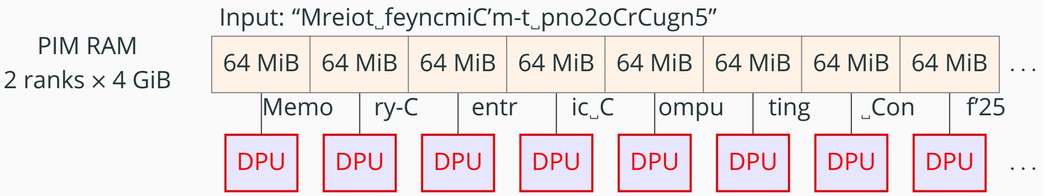
UPMEM PIM: Limitations



- 64 MiB chunk of DRAM per DPU, no shared memory → architecture ≠ CPU; algorithms need adjustments (unless “embarrassingly parallel”)
- Interleaving → PIM RAM ≠ DRAM; costly scatter/gather required [Dev19]

→ Applications must use PIM software development kit (SDK): dpu_alloc; write; dpu_load; dpu_launch; read; . . .

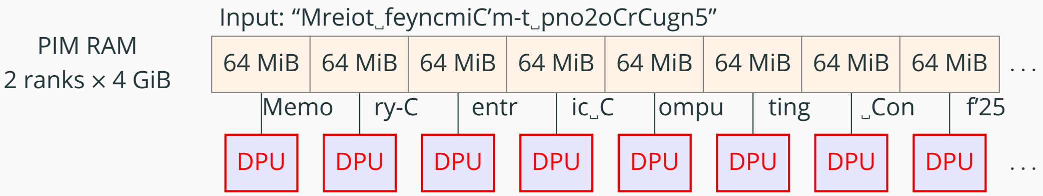
UPMEM PIM: Limitations



- 64 MiB chunk of DRAM per DPU, no shared memory → architecture ≠ CPU; algorithms need adjustments (unless “embarrassingly parallel”)
- Interleaving → PIM RAM ≠ DRAM; costly scatter/gather required [Dev19]

→ Applications must use PIM software development kit (SDK):
dpu_alloc; write; dpu_load; dpu_launch; read; . . .

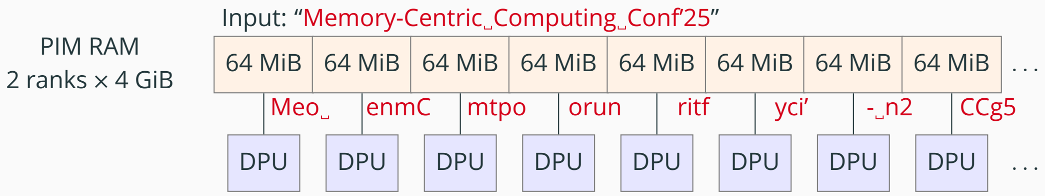
UPMEM PIM: Limitations



- 64 MiB chunk of DRAM per DPU, no shared memory → architecture ≠ CPU; algorithms need adjustments (unless “embarrassingly parallel”)
- Interleaving → PIM RAM ≠ DRAM; costly scatter/gather required [Dev19]

→ Applications must use PIM software development kit (SDK):
dpu_alloc; write; dpu_load; dpu_launch; read; . . .

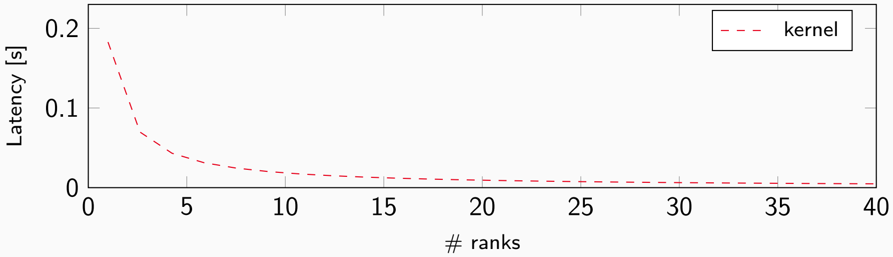
UPMEM PIM: Limitations



- 64 MiB chunk of DRAM per DPU, no shared memory → architecture ≠ CPU; algorithms need adjustments (unless "embarrassingly parallel")
- Interleaving → PIM RAM ≠ DRAM; costly scatter/gather required [Dev19]

→ Applications must use PIM software development kit (SDK):
dpu_alloc; write; dpu_load; dpu_launch; read; . . .

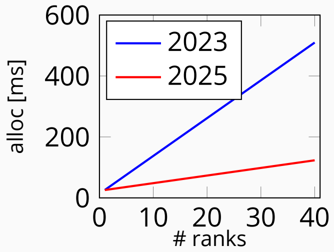
UPMEM PIM: SDK Overheads

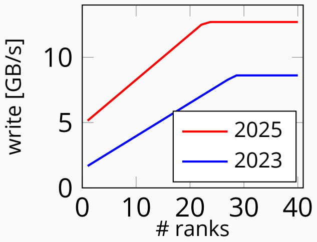


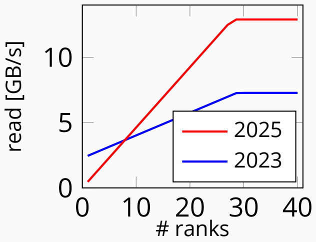
- (UPMEM) PIM is well-suited for “embarrassingly parallel” tasks

DBMS SELECT kernel latency: $237\,\mu s + 0.68\,ns \cdot \frac{\#rows}{\#ranks}$

UPMEM PIM: SDK Overheads

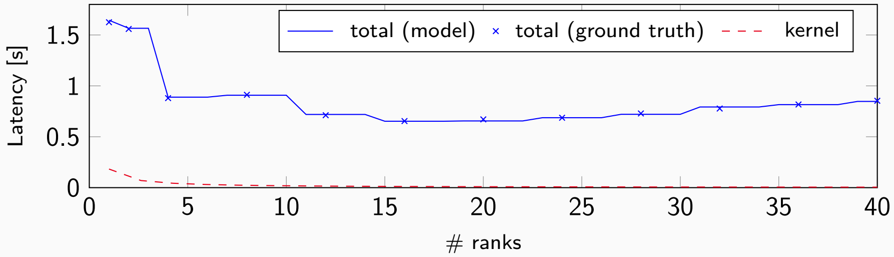






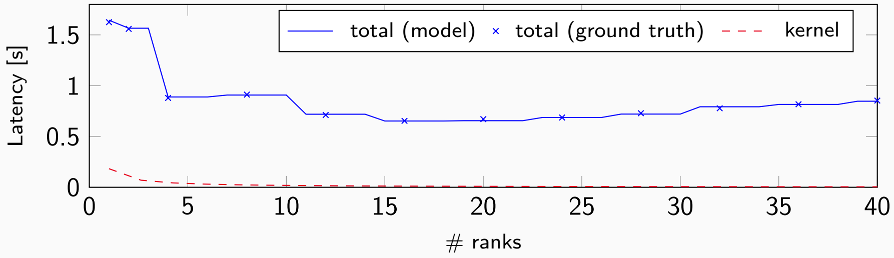
- (UPMEM) PIM is well-suited for “embarrassingly parallel” tasks
- Setup and data transfer costs

UPMEM PIM: SDK Overheads



- (UPMEM) PIM is well-suited for “embarrassingly parallel” tasks
- Setup and data transfer costs can jeopardize kernel speedup [FLS23; FS25]

UPMEM PIM: SDK Overheads

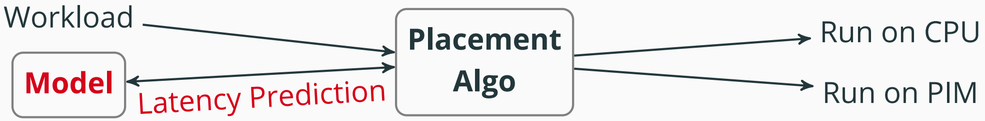


- (UPMEM) PIM is well-suited for “embarrassingly parallel” tasks
- Setup and data transfer costs can jeopardize kernel speedup [FLS23; FS25]

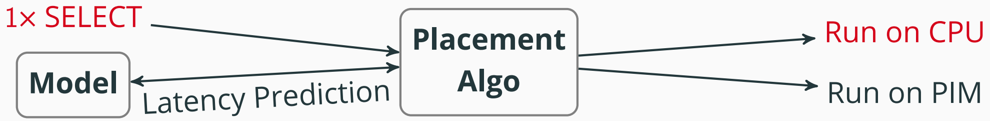
→ Placement algorithms must be aware of SDK overhead

→ Contribution: SDK overhead prediction for PIM-enabled applications

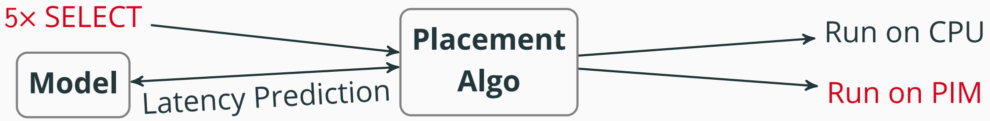
SDK Overhead Prediction: Motivation



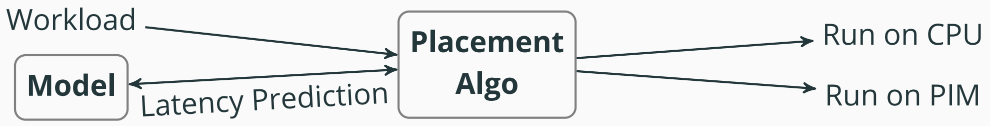
SDK Overhead Prediction: Motivation



SDK Overhead Prediction: Motivation

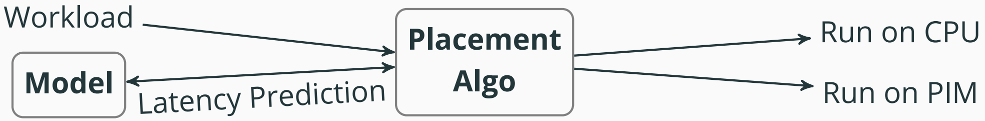


SDK Overhead Prediction: Challenges



- Traditional approach: application-specific performance models
  - For specific workload (e.g. fixed data placement / query sequences)
  - For specific software (e.g. UPMEM SDK 2023)
  - For specific hardware (e.g. Intel Xeon Silver 4215)

SDK Overhead Prediction: Challenges



- Traditional approach: application-specific performance models
  - For specific workload (e.g. fixed data placement / query sequences)
  - For specific software (e.g. UPMEM SDK 2023)
  - For specific hardware (e.g. Intel Xeon Silver 4215)

→ Model must be re-trained from scratch when any component changes

Proposal: Performance-Aware Behaviour Models

Decouple application behaviour from PIM / SDK performance

Proposal: Performance-Aware Behaviour Models

Decouple application behaviour from PIM / SDK performance

Behaviour Models

- Learnt from application traces
  - Coarse simulator sufficient
  - Independent of hardware

→ Predict SDK call sequences (including function arguments)

```
dpu_alloc(20 /* ranks */);
dpu_push_xfer(/* 4 GiB */);
dpu_push_xfer(/* 4 GiB */);
```

Proposal: Performance-Aware Behaviour Models

Decouple application behaviour from PIM / SDK performance

Behaviour Models

- Learnt from application traces
  - Coarse simulator sufficient
  - Independent of hardware

→ Predict SDK call sequences (including function arguments)

```
dpu_alloc(20 /* ranks */);
dpu_push_xfer(/* 4 GiB */);
dpu_push_xfer(/* 4 GiB */);
```

Hardware Models

- Learnt from microbenchmarks
  - One model per SDK API function
  - Independent of application

→ Predict latency of SDK calls (from workload and arguments)

$$T_{\text{alloc}} = 23.3 + 2.5 \cdot \#\mathit{ranks}$$

$$B_{\text{write}} = 4.80 + 0.35 \cdot \min(\#\mathit{ranks}, 22.7)$$

Proposal: Performance-Aware Behaviour Models

Decouple application behaviour from PIM / SDK performance

Behaviour Models

- Learnt from application traces
  - Coarse simulator sufficient
  - Independent of hardware

→ Predict SDK call sequences (including function arguments)

Hardware Models

- Learnt from microbenchmarks
  - One model per SDK API function
  - Independent of application

→ Predict latency of SDK calls (from workload and arguments)

→ Predict total SDK overhead for arbitrary workloads

→ Placement decision: run workload on CPU or PIM (out of scope)

Proposal: Performance-Aware Behaviour Models

Decouple application behaviour from PIM / SDK performance

Behaviour Models

- Learnt from application traces
  - Coarse simulator sufficient
  - Independent of hardware

→ Predict SDK call sequences (including function arguments)

Hardware Models

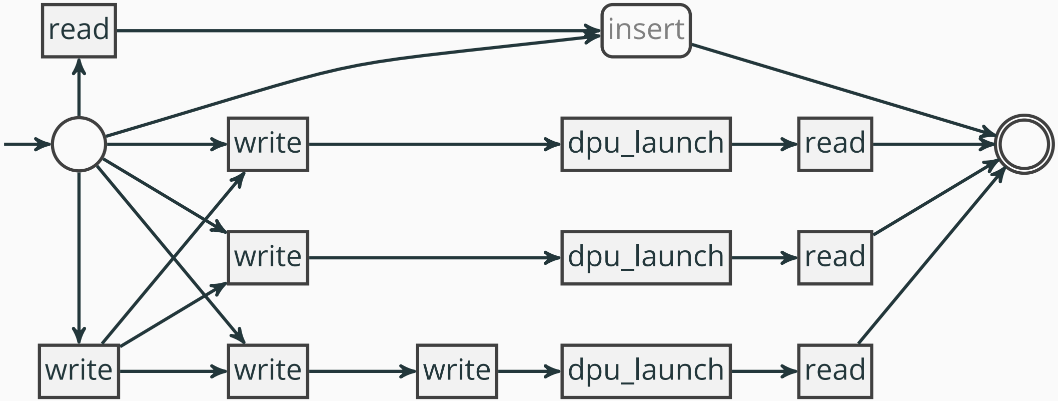
- Learnt from microbenchmarks
  - One model per SDK API function
  - Independent of application

→ Predict latency of SDK calls (from workload and arguments)

→ Predict total SDK overhead for arbitrary workloads

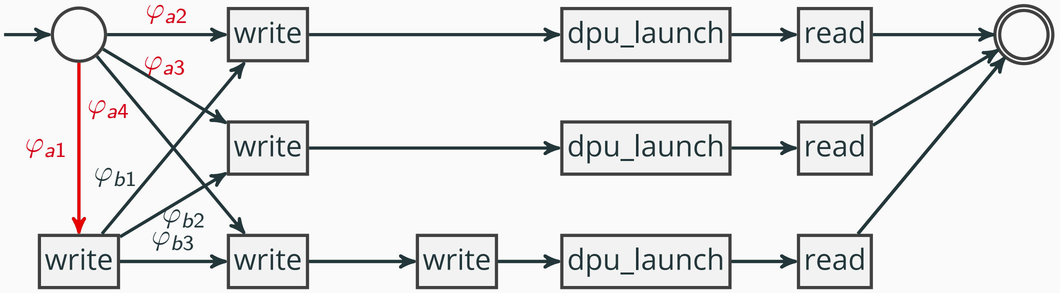
→ Placement decision: run workload on CPU or PIM (out of scope)

Behaviour Models



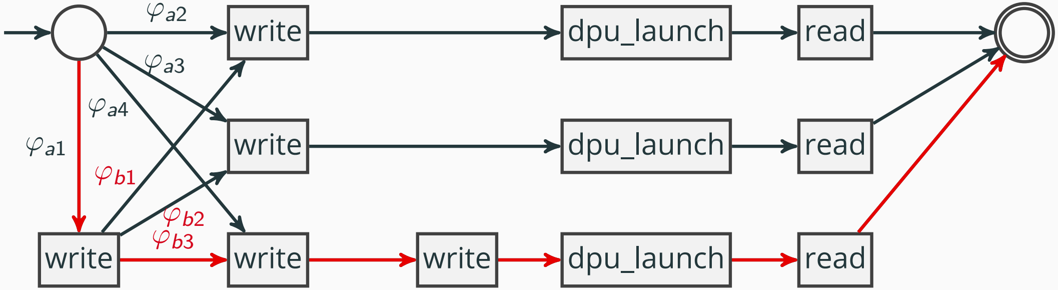
≙ Control flow graph: legal sequences of SDK calls (states ≙ callsites)

Behaviour Models



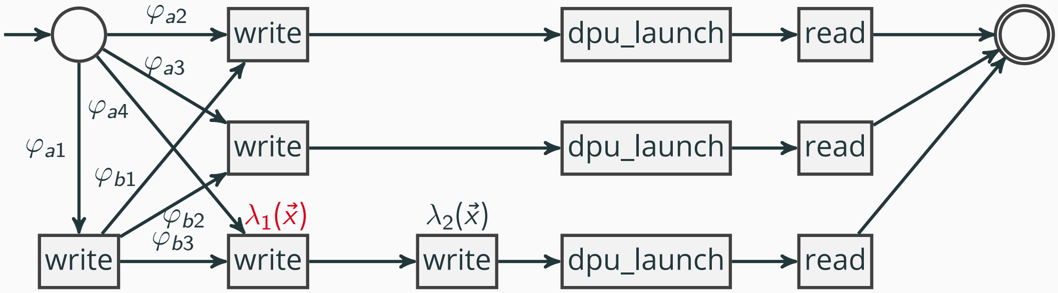
- Transitions guarded with workload-dependent conditions → deterministic
  - Example: $\vec{x}$ = (Op = UPDATE, DataOnDPUs = 0, $\#ranks$ = 20, $\#rows$ = 2^{30})
  - $\varphi_{a1} \equiv$ Op $\in$ {COUNT, SELECT, UPDATE} $\wedge$ ¬DataOnDPUs

Behaviour Models



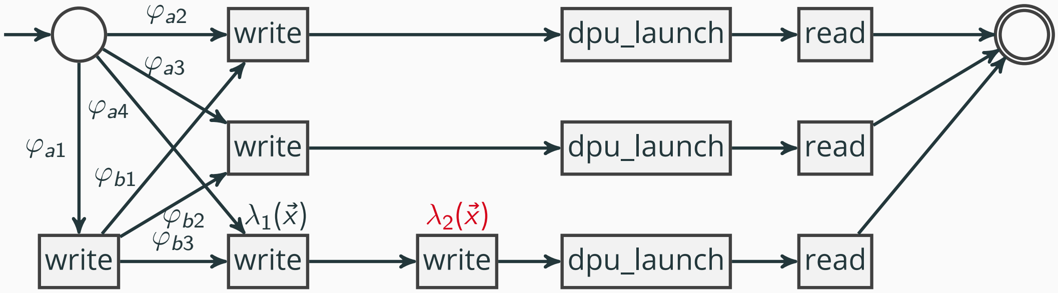
- Transitions guarded with workload-dependent conditions → deterministic
  - Example: $\vec{x}$ = (Op = UPDATE, DataOnDPUs = 0, $\#ranks = 20$, $\#rows = 2^{30}$)
  - $\varphi_{b1} \equiv$ Op = COUNT; $\varphi_{b2} \equiv$ Op = SELECT; $\varphi_{b3} \equiv$ Op = UPDATE

Behaviour Models



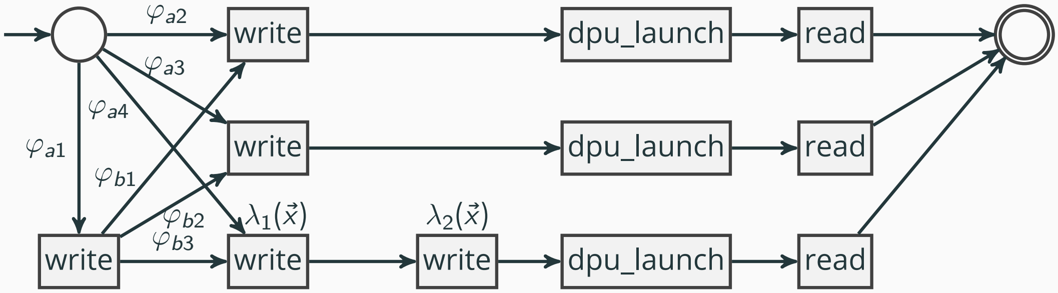
- Transitions guarded with workload-dependent conditions → deterministic
- States (callsites) predict SDK args and # iterations from workload config $\vec{x}$
  - $\lambda_1 = 46 + 1526 \cdot \#ranks$; $\lambda_2 = \frac{1}{8} \cdot \#rows + 368 \cdot \#ranks$; ...

Behaviour Models



- Transitions guarded with workload-dependent conditions → deterministic
- States (callsites) predict SDK args and # iterations from workload config $\vec{x}$
  - $\lambda_1 = 46 + 1526 \cdot \#ranks$; $\lambda_2 = \frac{1}{8} \cdot \#rows + 368 \cdot \#ranks$; ...

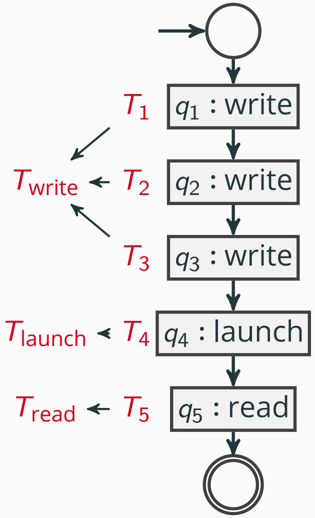
Behaviour Models



- Transitions guarded with workload-dependent conditions → deterministic
- States (callsites) predict SDK args and # iterations from workload config $\vec{x}$
- Learnt automatically; independent of SDK / hardware performance

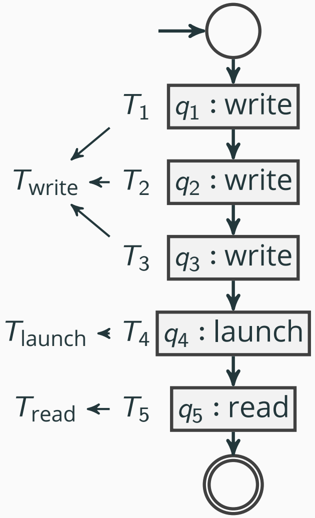
Performance-Aware Behaviour Models

- For each callsite q_i: hardware model T_i
  - Identical for all callsites of an SDK API function
  - Independent of application / behaviour model



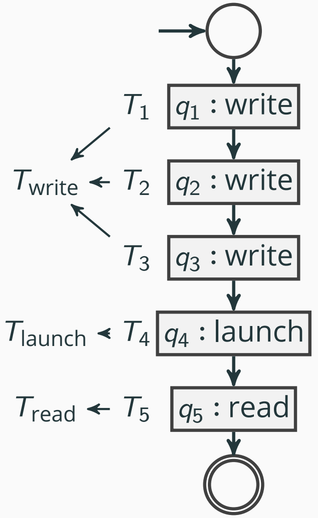
Performance-Aware Behaviour Models

- For each callsite q_i: hardware model T_i
  - Identical for all callsites of an SDK API function
  - Independent of application / behaviour model
- Proof-of-concept latency prediction workflow:
  - Input: Workload configuration $\vec{x}$
  - → Sequence $(q_1, \ldots, q_n)$ of SDK calls and args λ_i, iteration counts ρ_i via behaviour model
  - → Total latency = $\sum_{i=1}^{n} T_i(\lambda_i(\vec{x}_i)) \cdot \rho_i(\vec{x}_i)$



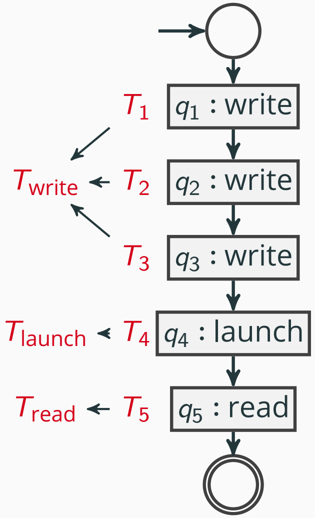
Performance-Aware Behaviour Models

- For each callsite q_i: hardware model T_i
  - Identical for all callsites of an SDK API function
  - Independent of application / behaviour model
- Proof-of-concept latency prediction workflow:
  - Input: Workload configuration $\vec{x}$
  - → Sequence $(q_1, \ldots, q_n)$ of SDK calls and args λ_i, iteration counts ρ_i via behaviour model
  - → Total latency = $\sum_{i=1}^{n} T_i(\lambda_i(\vec{x}_i)) \cdot \rho_i(\vec{x}_i)$



Performance-Aware Behaviour Models

- For each callsite q_i: hardware model T_i
  - Identical for all callsites of an SDK API function
  - Independent of application / behaviour model
- Proof-of-concept latency prediction workflow:
  - Input: Workload configuration $\vec{x}$
  - → Sequence $(q_1, \ldots, q_n)$ of SDK calls and args λ_i, iteration counts ρ_i via behaviour model
  - → Total latency = $\sum_{i=1}^{n} T_i(\lambda_i(\vec{x}_i)) \cdot \rho_i(\vec{x}_i)$

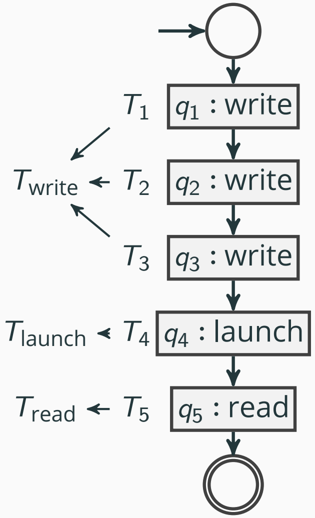


Performance-Aware Behaviour Models

- For each callsite q_i: hardware model T_i
  - Identical for all callsites of an SDK API function
  - Independent of application / behaviour model
- Proof-of-concept latency prediction workflow:
  - Input: Workload configuration $\vec{x}$

  → Sequence $(q_1, \dots, q_n)$ of SDK calls and args λ_i, iteration counts ρ_i via behaviour model

  → Total latency = $\sum_{i=1}^{n} T_i(\lambda_i(\vec{x}_i)) \cdot \rho_i(\vec{x}_i)$



Model Learning

- 100 % unattended proof-of-concept implementation

Model Learning

- 100 % unattended proof-of-concept implementation

① Obtain application traces for representative workloads $\vec{x}_1, \vec{x}_2, \ldots$

  - Fully automated via AspectC++ (similar methods for non C / C++ applications)
  - Simulator is sufficient; no cycle-accurate timings required

```
int main(/* ... */) {
  /* ... */
  n_dpus = part(&input);
  dpu_alloc(n_dpus);
  /* ... */
}
```

```
advice call("% dpu_alloc(...)") : around() {
  n_dpus = *(tjp->arg<0>());
  tjp->proceed();
  printf("[::] dpu_alloc @ %s:%d | n_dpus=%u\n",
    tjp->filename(), tjp->line(), n_dpus
  );
}
```

Model Learning

- 100 % unattended proof-of-concept implementation

① Obtain application traces for representative workloads $\vec{x}_1, \vec{x}_2, \ldots$
  - Fully automated via AspectC++ (similar methods for non C / C++ applications)
  - Simulator is sufficient; no cycle-accurate timings required

② Learn behaviour model from traces
  - Structure (Q, Δ) and annotations (guards φ_i, args λ_i, loops ρ_i)
  - Interpretable decision trees / regression model trees for φ_i, λ_i, ρ_i [FS22]

```
[>>] BS | n_dpus=1 n_elements=262144 n_queries=512
[::] dpu_alloc @ host/app.c:104 | n_dpus=64
[::] dpu_load @ host/app.c:108 | n_dpus=64
[::] dpu_push_to_dpu @ host/app.c:221 | n_dpus=64 total_payload_B=1536
...
```

Model Learning

- 100 % unattended proof-of-concept implementation

① Obtain application traces for representative workloads $\vec{x}_1, \vec{x}_2, \ldots$

- Fully automated via AspectC++ (similar methods for non C / C++ applications)
- Simulator is sufficient; no cycle-accurate timings required

② Learn behaviour model from traces

- Structure (Q, Δ) and annotations (guards φ_i, args λ_i, loops ρ_i)
- Interpretable decision trees / regression model trees for φ_i, λ_i, ρ_i [FS22]

③ Learn latency prediction models (hardware models)

- Either from microbenchmarks or from non-simulator traces
- Independent of application; must only be done once

Proof of Concept: Evaluation Setup

- 12 applications:
  - Custom PIM-enabled database kernels [FLS25]
  - PrIM suite: matrix ops, data analysis/lookups, neural networks [Góm+22]
  - 4× SDK calls within loops; 3× conditional SDK calls

Proof of Concept: Evaluation Setup

- 12 applications:
  - Custom PIM-enabled database kernels [FLS25]
  - PrIM suite: matrix ops, data analysis/lookups, neural networks [Góm+22]
  - 4× SDK calls within loops; 3× conditional SDK calls
- Two behaviour models: traces from simulator / traces from real hardware

Proof of Concept: Evaluation Setup

- 12 applications:
  - Custom PIM-enabled database kernels [FLS25]
  - PrIM suite: matrix ops, data analysis/lookups, neural networks [Góm+22]
  - 4× SDK calls within loops; 3× conditional SDK calls
- Two behaviour models: traces from simulator / traces from real hardware
- Two sets of hardware models ($T_{\text{alloc}}, T_{\text{load}}, T_{\text{write}}, T_{\text{read}}$):
  - Learnt from microbenchmarks (4 . . . 19 % prediction error)
  - Learnt from timed traces on real hardware (5 . . . 23 % prediction error)

Proof of Concept: Evaluation Setup

- 12 applications:
  - Custom PIM-enabled database kernels [FLS25]
  - PrIM suite: matrix ops, data analysis/lookups, neural networks [Góm+22]
  - $4\times$ SDK calls within loops; $3\times$ conditional SDK calls
- Two behaviour models: traces from simulator / traces from real hardware
- Two sets of hardware models ($T_{\text{alloc}}, T_{\text{load}}, T_{\text{write}}, T_{\text{read}}$):
  - Learnt from microbenchmarks (4 … 19 % prediction error)
  - Learnt from timed traces on real hardware (5 … 23 % prediction error)

→ Behaviour model accuracy: predicted vs. observed API calls

Proof of Concept: Evaluation Setup

- 12 applications:
  - Custom PIM-enabled database kernels [FLS25]
  - PrIM suite: matrix ops, data analysis/lookups, neural networks [Góm+22]
  - 4× SDK calls within loops; 3× conditional SDK calls
- Two behaviour models: traces from simulator / traces from real hardware
- Two sets of hardware models ($T_{\text{alloc}}, T_{\text{load}}, T_{\text{write}}, T_{\text{read}}$):
  - Learnt from microbenchmarks (4 . . . 19 % prediction error)
  - Learnt from timed traces on real hardware (5 . . . 23 % prediction error)

→ Behaviour model accuracy: predicted vs. observed API calls

→ Hardware model accuracy: predicted vs. observed total latency

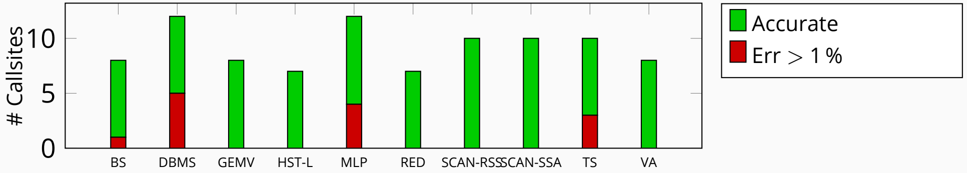
Evaluation Results: SDK Call Prediction

- Two applications: unable to learn behaviour model
  - Conditional API calls within loops not yet supported by proof-of-concept algo

Evaluation Results: SDK Call Prediction

- Two applications: unable to learn behaviour model
  - Conditional API calls within loops not yet supported by proof-of-concept algo
- Remaining 10 applications:
  - traces 100 % accurate; argument values (λ) accurate at > 85 % of callsites
  - Affected applications: BS, DBMS, MLP, TS

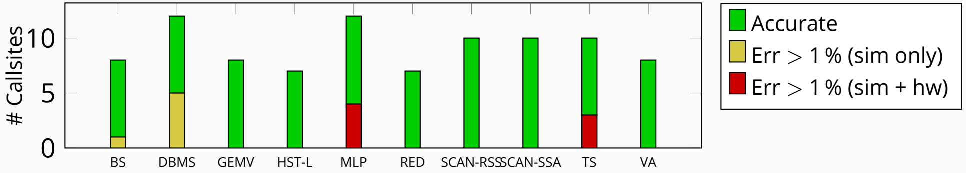
Predicted vs. observed SDK argument values: model learnt via simulator



Evaluation Results: SDK Call Prediction

- Two applications: unable to learn behaviour model
  - Conditional API calls within loops not yet supported by proof-of-concept algo
- Remaining 10 applications:
  - traces 100 % accurate; argument values (λ) accurate at > 85 % of callsites
  - Affected applications: BS, DBMS (when learnt via simulator), MLP, TS

Predicted vs. observed SDK argument values



Evaluation Results: SDK Call Prediction

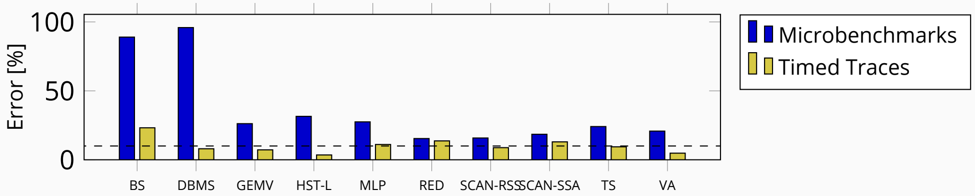
- Two applications: unable to learn behaviour model
  - Conditional API calls within loops not yet supported by proof-of-concept algo
- Remaining 10 applications:
  - traces 100 % accurate; argument values (λ) accurate at > 85 % of callsites
  - Affected applications: BS, DBMS (when learnt via simulator), MLP, TS

→ BS, DBMS: discrepancies between simulated and real hardware:
1 DPU per rank (UPMEM simulator) vs. 64 DPUs per rank (UPMEM PIM)

→ MLP, TS: observed argument values are not constant within loops
(limitation in proof-of-concept model and algorithm)

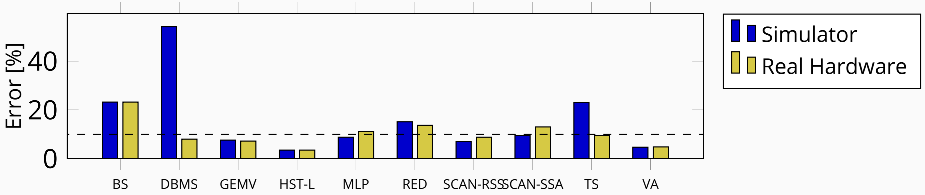
Evaluation Results: Latency Prediction

- Hardware model learning: microbenchmarks vs. timed traces
    - BS, DBMS: microbenchmarks must use appropriate SDK argument values



Evaluation Results: Latency Prediction

- Hardware model learning: microbenchmarks vs. timed traces
  - BS, DBMS: microbenchmarks must use appropriate SDK argument values
- Behaviour model learning: simulator vs. real hardware
  - Little difference except for DBMS and TS (see previous slide)
  - MLP: inaccurate argument values $\not\Rightarrow$ inaccurate latency predictions



Evaluation Results: Latency Prediction

- Hardware model learning: microbenchmarks vs. timed traces
  - BS, DBMS: microbenchmarks must use appropriate SDK argument values
- Behaviour model learning: simulator vs. real hardware
  - Little difference except for DBMS and TS (see previous slide)
  - MLP: inaccurate argument values $\nRightarrow$ inaccurate latency predictions

→ Representative training data is crucial for hardware models

→ Suitable simulators are sufficient for behaviour model learning
  - 6 of 12 targets: $< 10\,\%$ latency error ($\approx$ underlying performance models)
  - Tracing time: minutes rather than hours

Evaluation Results: Latency Prediction

- Hardware model learning: microbenchmarks vs. timed traces
  - BS, DBMS: microbenchmarks must use appropriate SDK argument values
- Behaviour model learning: simulator vs. real hardware
  - Little difference except for DBMS and TS (see previous slide)
  - MLP: inaccurate argument values $\not\Rightarrow$ inaccurate latency predictions

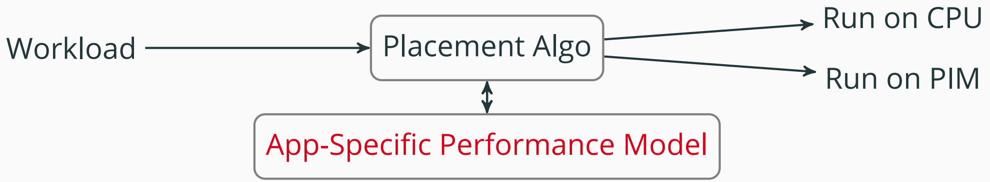
→ Representative training data is crucial for hardware models

→ Suitable simulators are sufficient for behaviour model learning
  - 6 of 12 targets: $< 10\,\%$ latency error ($\approx$ underlying performance models)
  - Tracing time: minutes rather than hours

- Artefacts at `ess.cs.uos.de/git/artifacts/ccmcc25-behaviour-models`

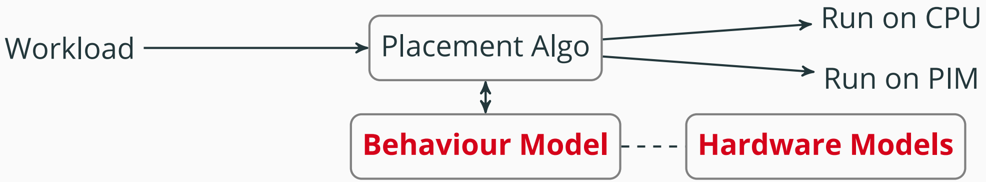
Conclusion

- (UPMEM) PIM: up to > 99 % of latency in management overhead (SDK)
  - Must be considered by latency prediction / placement decisions
  - Existing approaches: costly and inflexible application-specific models



Conclusion

- (UPMEM) PIM: up to > 99 % of latency in management overhead (SDK)
  - Must be considered by latency prediction / placement decisions
  - Existing approaches: costly and inflexible application-specific models
- Performance-aware behaviour models disentangle application / HW
  - Behaviour model: Learnt from simulation traces
  - Hardware models: learnt from appropriate microbenchmarks



Conclusion

- (UPMEM) PIM: up to $> 99\,\%$ of latency in management overhead (SDK)
  - Must be considered by latency prediction / placement decisions
  - Existing approaches: costly and inflexible application-specific models
- Performance-aware behaviour models disentangle application / HW
  - Behaviour model: Learnt from simulation traces
  - Hardware models: learnt from appropriate microbenchmarks
- Proof-of-concept implementation: promising results
  - Behaviour model ($\approx$ CFG) accurately captures control flow within application
  - 6/12 evaluation targets: $< 10\,\%$ latency prediction error
  - Reduced training time; improved flexibility and interpretability

References i

[Dev19] Fabrice Devaux. **“The true Processing In Memory accelerator”.** In: 2019 IEEE Hot Chips 31 Symposium (HCS). 2019, pp. 1–24. DOI: `10.1109/HOTCHIPS.2019.8875680`.

[FLS23] Birte Friesel, Marcel Lütke Dreimann, and Olaf Spinczyk. **“A Full-System Perspective on UPMEM Performance”.** In: Proceedings of the 1st Workshop on Disruptive Memory Systems. DIMES ’23. Koblenz, Germany: Association for Computing Machinery, Oct. 2023, pp. 1–7. ISBN: 979-8-4007-0300-3. DOI: `10.1145/3609308.3625266`. URL: `https://doi.org/10.1145/3609308.3625266`.

References ii

[FLS25] Birte Friesel, Marcel Lütke Dreimann, and Olaf Spinczyk. **“Lightning Talk: Feasibility Analysis of Semi-Permanent Database Offloading to UPMEM Near-Memory Computing Modules”.** In: *Datenbanksysteme für Business, Technologie und Web – Workshopband*. BTW ’25. Bonn, Germany: Gesellschaft für Informatik, Mar. 2025, pp. 355–366. DOI: `10.18420/BTW2025-140`. URL: `https://doi.org/10.18420/BTW2025-140`.

[FS22] Birte Friesel and Olaf Spinczyk. **“Regression Model Trees: Compact Energy Models for Complex IoT Devices”.** In: *Proceedings of the Workshop on Benchmarking Cyber-Physical Systems and Internet of Things*. CPS-IoTBench ’22. Milan, Italy: IEEE, May 2022, pp. 1–6. DOI: `10.1109/CPS-IoTBench56135.2022.00007`. URL: `https://doi.org/10.1109/CPS-IoTBench56135.2022.00007`.

References iii

[FS25] Birte Friesel and Olaf Spinczyk. **“Overhead Prediction for PIM-Enabled Applications with Performance-Aware Behaviour Models”.** In: Proceedings of the 1st IEEE Cross-disciplinary Conference on Memory-Centric Computing. CCMCC ’25. to appear. Dresden, Germany: IEEE, Oct. 2025.

[Góm+22] Juan Gómez-Luna et al. **“Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System”.** In: IEEE Access 10 (2022), pp. 52565–52608. DOI: `10.1109/ACCESS.2022.3174101`. URL: `https://doi.org/10.1109/ACCESS.2022.3174101`.