Benchmarking Cache Programming Against Optimal Caching

THE INTERNATIONAL SYMPOSIUM ON MEMORY SYSTEMS 2025

YANHUI WU, UR
CHEN DING, UR
VINCENT MICHELINI, RIT
DORIN PATRU, RIT

OUTLINE and Acknowledgements

- The Problem – What, Why, and How?
- Hardware Prototype
- Application and Goals
- Reference Leases – Definitions and Terms
- Lease Algorithms
- Experimental Setup
- Test Results and Discussions
- Conclusions

- The research is partly supported by the **National Science Foundation** (Contract No. **SHF-2217395, CCF-2114285, CCF-2114319**)
- Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

The Problem – What, Why, and How?

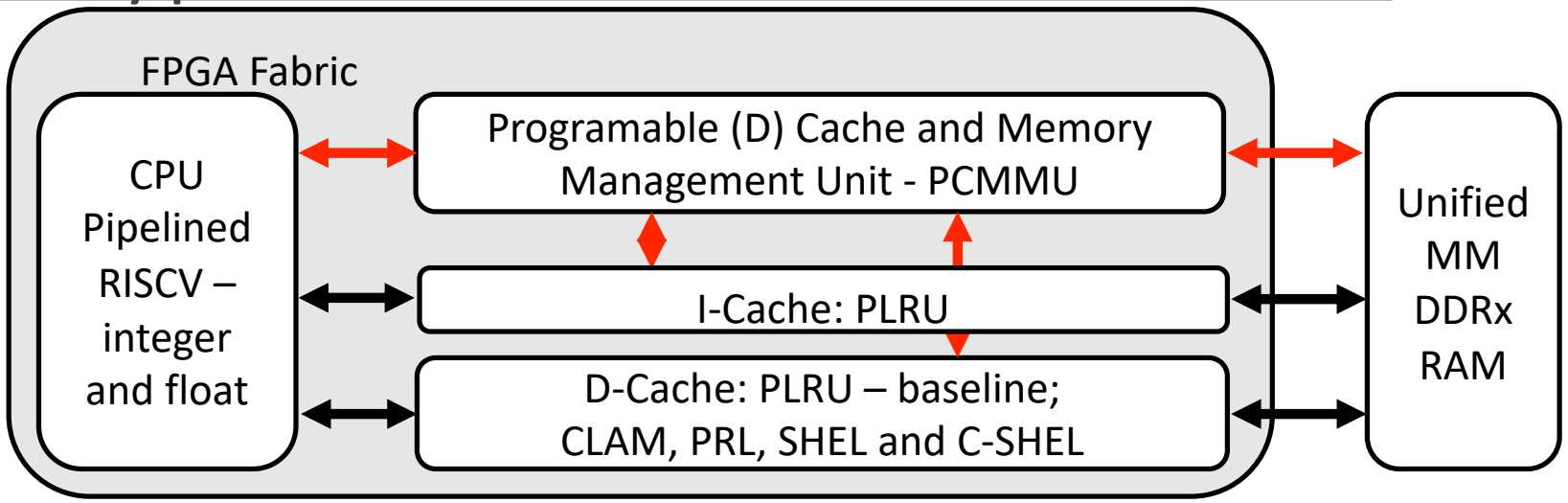
- **What is our focus**? Cache levels between the processing core(s) and main memory
- **Why is this important**?
  - Manual management is complex and not portable
  - Automatic management (HW-based) is sub-optimal

	Conventional Cache	Lease Cache
Primary (Eviction) Replacement Policy	Automatic	Reference Leases, Programmable
Information Used	Recent History at Runtime (Dynamic)	Program Analysis at Compile Time (Static)
Secondary Policy	N/A	Random Eviction

- **How are we proposing to solve it**?
  - Using a **Lease-Based Programmable Cache**
  - A **lease** protects items (data blocks or cache lines) in cache for a specified amount of time, i.e., these can only be evicted once the lease period elapses (**expires**)

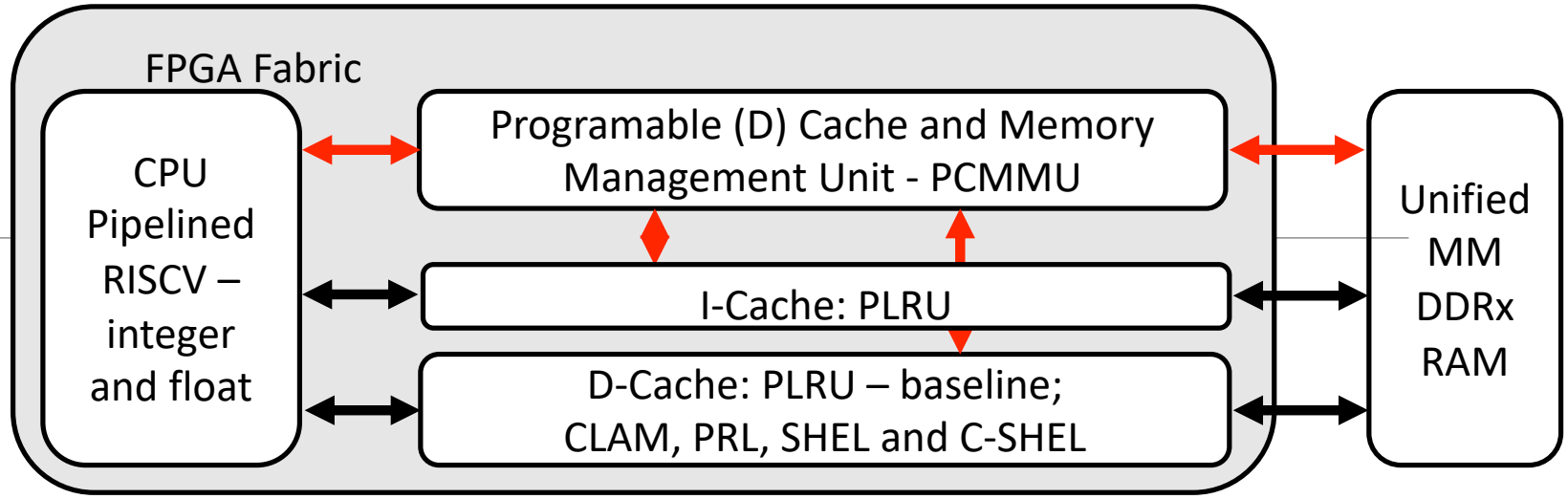
Lease Cache – Emulation and Test System or Hardware Prototype

- **FPGA:** Altera Cyclone-V
- **RISC-V Core:** Unprivileged Instruction Set
- **I-Cache:** 64-Byte lines; 64 and 128 lines (2KB and 4KB); PLRU
- **D-Cache:** 64-Byte lines; 64 and 128 lines (2KB and 4KB); PLRU, CLAM or SHELL



- **Benchmark** execution and data collection are controlled remotely from a host computer via a JTAG connection.
- The **host** program, written in C and compiled with GCC, provides commands to load binaries, run benchmarks, and collect metrics (misses, hits, cycle counts, samples, and traces).
- ***This setup allows for reproducible evaluation across a range of benchmarks while isolating the impact of cache policy from other architectural factors.***

Application and Goals



- This work is **not tied to a specific application domain** such as embedded, client, or HPC systems
- Rather, our goal is to evaluate lease-based programable cache in a **controlled hardware environment** and to quantify how closely it can approach the **theoretical optimum (OPT)**
- The RISC-V FPGA prototype provides a **simple, reproducible platform** that exposes cache behavior without interference from complex multicore or out-of-order features
- This makes it useful for controlled evaluation, but the techniques and findings are **not restricted** to RISC-V or FPGA platforms
- Lease caches are a **generic mechanism** that can be applied across system classes wherever fixed-size caches are deployed

Reference Leases – Definitions and Terms

- **A Reference** is defined as the instruction that invokes a memory access, i.e., the program counter for the load/store instruction.
- **Reuse Interval (RI)** is defined as the change in logical time between a data block's use and its reuse.
  - Suppose we have a trace ***abccba***, the reuse interval of the datum ***a*** is ***RI=5***.
- **The RI distribution of a reference (RID)** is the distribution of RI's among all of its accesses.
  - Using the same trace ***abccba*** given in previous definition and assuming there is only one reference, the RI distribution of this reference would contain **3 different reuses *1, 3, 5*** caused by the access to datum ***c, b and a*** respectively, each accounting for **1/3**.
- **The Access Ratio (AR)** of a reference is the portion of all accesses in the trace that are invoked by that reference.
- **Leases are derived from RIs.**

Lease Algorithms/ Techniques/ Policies – From Previous Work

Leases are **generated** by assignment algorithms based on program analysis:

- Compiler Assigned Reference Leases (CARL)
  - Variable-sized or virtual cache;
- Compiler Lease of Cache Memory (CLAM)
  - Fixed-sized cache;
- Scope-Hooked Eviction Leases (SHEL)
  - Fixed-sized cache;

Cost-based greedy algorithms, variable lease assignments, using **Profit Per Unit Cost (PPUC)** benefit of lease assignment normalized to its cost

Leases are **derived** from ***reuse intervals (Ris).***

Experimental Setup

- **Benchmark Suite**: **PolyBench** compiled using GCC -O3} optimization level without vectorization
- **Target**: **RISC-V** core configured on an Altera Cyclone-V FPGA
- The **Memory Hierarchy**: consists of a single-level, separate instruction and data caches, and a **DDR3 off-chip** main memory
- **I-Cache**: 64-Byte lines; 64 and 128 lines (2KB and 4KB); PLRU
- **D-Cache**: 64-Byte lines; 64 and 128 lines (2KB and 4KB); CLAM or SHELL

- A **metric collection system** is embedded inside both data and instruction caches and the RISC-V core
  - These collect relevant metrics such as: ***cache hits, misses, and total accesses*** by snooping the internal cache signals
- A program on a **host** computer reads and writes data in DDR3 memory or **memory-mapped registers (MMR)**
  - It controls benchmark execution and metric/results collection
- The **MMRs** are hardware registers that are accessed using memory space mapped addresses

Experimental Setup – Data Collection

- There are three main **code sections** during each benchmark execution that are relevant to data collection:
  - **Setup** Code: allocates benchmark arrays in the heap and initializes their default values
  - **Kernel** Code: executes the benchmark algorithm; it is during this code segment that metrics are collected
  - **Cleanup** Code: frees the heap allocated memory
- At the end of the setup code, the core writes to the metrics-control MMR ***enabling*** collection
- At the end of the kernel code, the core will again write to the metrics-control MMR ***disabling*** collection.

- **Data Collected:**
  - **PLRU** Run: cache hits, misses, and total accesses
  - **Lease** Runs (CLAM or SHEL): cache hits, misses, and total accesses
  - Data Access **Sample** Information – Reuse Interval Distributions (**RIDs**)
    - The sampler is embedded in the data cache and snoops each access to ultimately generate the forward reuse interval of memory blocks
    - The sampler consists of a ***sample table*** and a ***sample buffer***
  - **Trace Data** is the record of every request to the data cache during the execution of the kernel code of the benchmark
    - A trace data entry consists of the core program counter (PC), word address, and hit/miss information

Experimental Setup – Optimal Caching

- **OPT Simulator**:
  - Optimal offline caching simulator in Rust
  - Provides a theoretical lower bound for the cache miss ratio for each benchmark
  - Models Belady's MIN algorithm (also called OPT)
  - Operation:
    - Input: Trace information collected by the emulation and test system
    - Output: The precise sequence of cache hits and misses, as well as the miss ratio under optimal caching
- **OPT Rationale**:
  - Provides the absolute best miss ratio possible for any cache replacement strategy in a fixed available cache size setup, assuming perfect knowledge of the future
- **Belady's MIN** Algorithm:
  - Always evicts the cached block whose next reuse is farthest in the future (or, equivalently, that will not be used again for the longest time)
  - Operation:
    - **Pre-process** the access trace to compute, for each access, when the next occurrence of each block will be – Forward Reuse Interval
    - **Simulate**:
      - If the block is present in the cache (a hit) – proceed
      - Else if not (a miss), and the cache is full, we evict the block in the cache whose next use is farthest in the future (or never used again)
      - The accessed block is then inserted into the cache.

Test Results and Discussions
PolyBench Suite – Performance Scores

- The **PolyBench**/C 4.2.1 suite:
  - Contains 30 scientific computing workloads
  - The kernels span a range of domains, including linear algebra, image processing, physics simulation, statistics, and dynamic programming
  - This wide variety of kernel choices provides a good analysis for cache performance across multiple caching policies.
- **29/30** were chosen for analysis
  - ***jacobi-1d*** is excluded due to the small data size and almost zero miss ratio
- The 29 benchmarks can be categorized into single- and multi-scope
  - Each of the 16 ***single-scope*** benchmarks consists of kernel code that has a ***single loop nest***
  - The 13 ***multi-scope*** benchmarks contain ***two or more distinct loop nests back-to-back***

- To evaluate the lease cache performance, we compute a **score** to quantify how much closer it is to OPT compared to PLRU:

$$\text{Score} = \frac{100(mr_{PLRU}(c) - mr_{Lease}(c))}{mr_{PLRU}(c) - mr_{Opt}(c)}$$

- ***The score is at most 100***, which means that the performance has reached the theoretical optimum
- A ***positive score*** means an ***improvement*** over PLRU
- The formula is **symmetric** with respect to hit or miss ratios; substituting hit ratios yields the same value

Score Arithmetic means for two inputs and two cache sizes – How close are we to optimum (OPT)

performance scores	64 cache lines	128 cache lines	mean
small input	61.43	75.48	68.46
medium input	64.96	66.32	65.64
mean	63.20	70.90	67.05

- **Observations:**
  - On average, scores are **over 60**
  - For **small input**, the average score improves from 61 to 76 as the cache size increases
  - For the **medium input**, the score is effectively unchanged, 65 for the 64-block cache and 66 for the 128-block cache

$$\text{Score} = \frac{100(mr_{PLRU}(c) - mr_{Lease}(c))}{mr_{PLRU}(c) - mr_{Opt}(c)}$$

Impact of Loop Nest Shape on Lease Cache Effectiveness

Rectangular	Triangular
2mm	cholesky
3mm	correlation
adi	covariance
atax	durbin
bicg	gramschmidt
deriche	lu
doitgen	ludcmp
fdtd-2d	nussinov
floyd-warshall	symm
gemm	syr2k
gemver	syrk
gesummv	trisolv
heat-3d	trmm
jacobi-2d	
mvt	
seidel-2d	

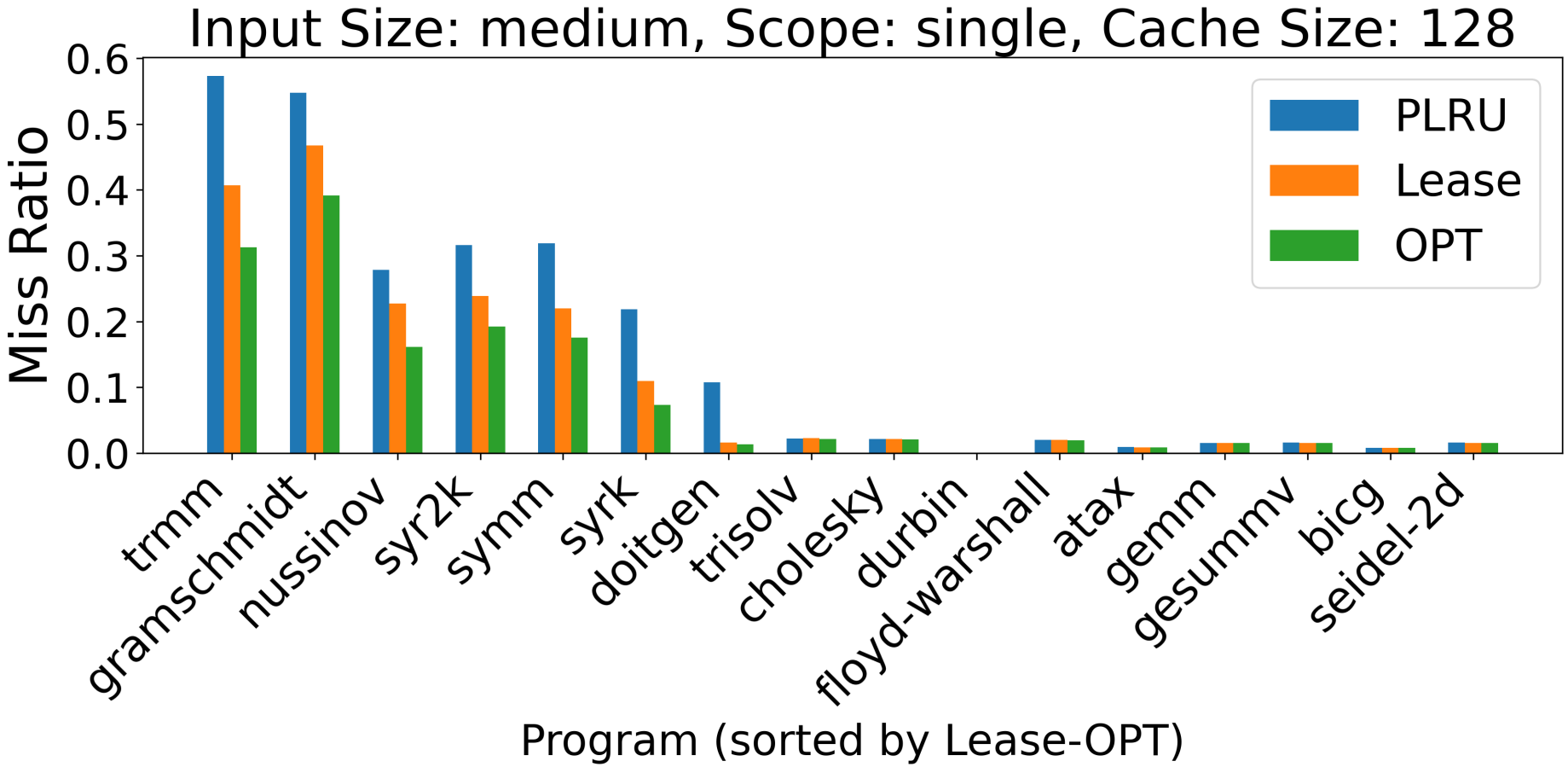
Inputs	Cache Size	Iteration Space	mean	std	min	25%	50%	75%	max
Both Inputs	Both Sizes	Triangular	48.52	59.45	-181.61	45.96	67.74	79.53	97.43
small	64	Triangular	42.80	65.65	-163.16	43.07	71.42	74.39	79.11
small	128	Triangular	73.02	27.43	0.00	76.46	83.29	87.72	97.43
medium	64	Triangular	36.10	63.41	-103.99	43.47	58.70	66.70	84.64
medium	128	Triangular	42.16	70.98	-181.61	47.99	63.74	70.75	88.30
Both Inputs	Both Sizes	Rectangular	82.10	17.54	22.92	77.20	87.81	93.20	99.99
small	64	Rectangular	76.56	20.28	22.92	70.81	82.33	89.84	99.72
small	128	Rectangular	77.48	20.75	25.00	65.68	85.29	90.40	98.54
medium	64	Rectangular	88.40	11.85	60.71	82.44	92.82	96.82	99.99
medium	128	Rectangular	85.95	14.01	39.05	85.79	90.08	92.16	99.47

- **Triangular loop nest**: inner loop bounds depend on the index of an outer loop variable (e.g., for i=0 to N, for j=0 to i)
- **Rectangular or non-triangular**: loop bounds are independent

- **Mean** improvement **above 80% for non-triangular** loop nests, with **low variance**
- However, **for triangular** loop nests, not only is the mean lower (often **below 50%**), but the **variance** is much **higher**
- This trend is consistent across input and cache sizes

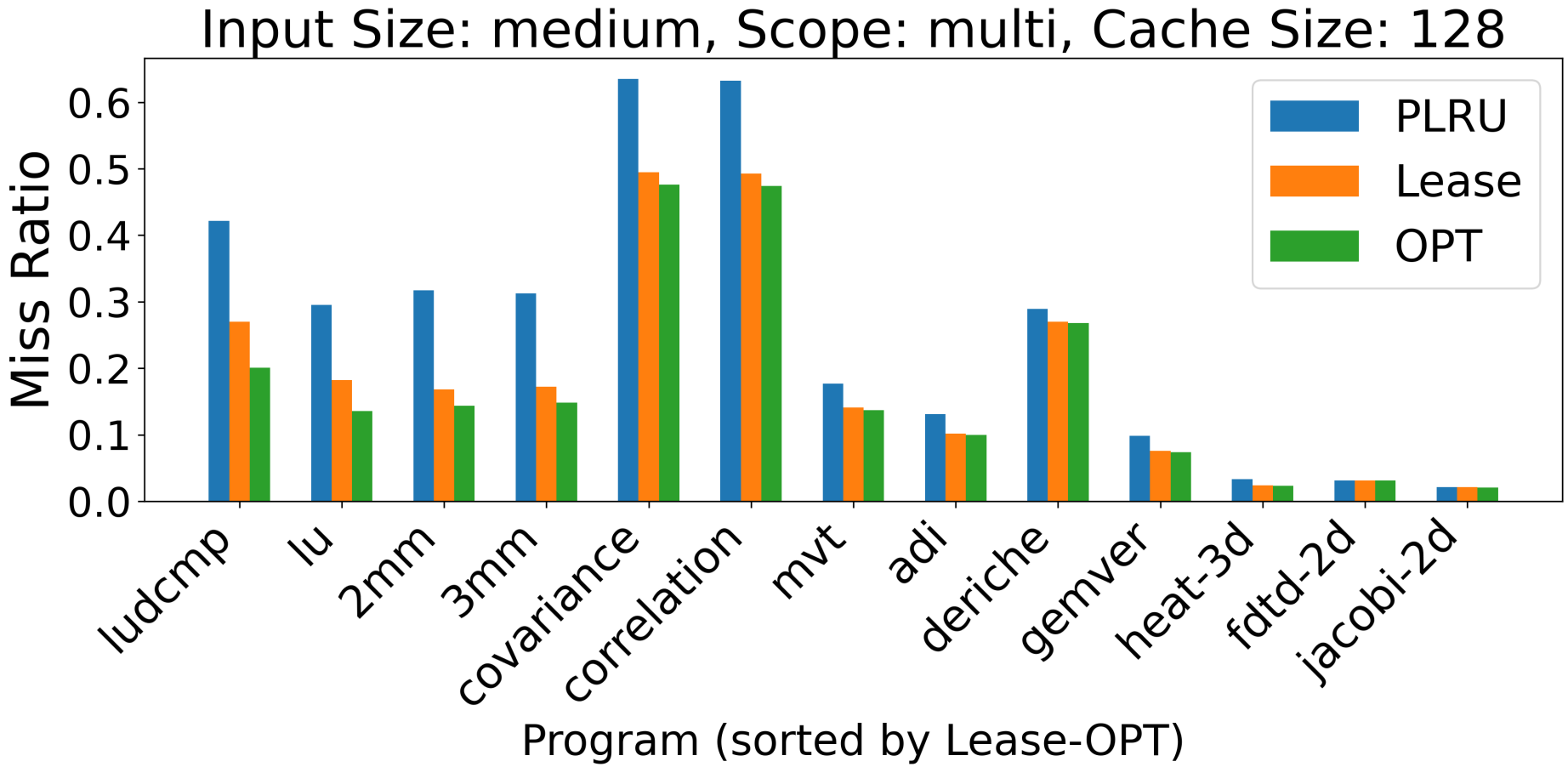
Miss Ratio Breakdown by Scope Assignment
Single-Scope Assignment – CLAM

- Miss ratio comparison for PolyBench single-scope benchmarks (medium input, 128-line cache)
- Programs are sorted by the Lease--OPT gap



Miss Ratio Breakdown by Scope Assignment
Multi-Scope Assignment – SHEL

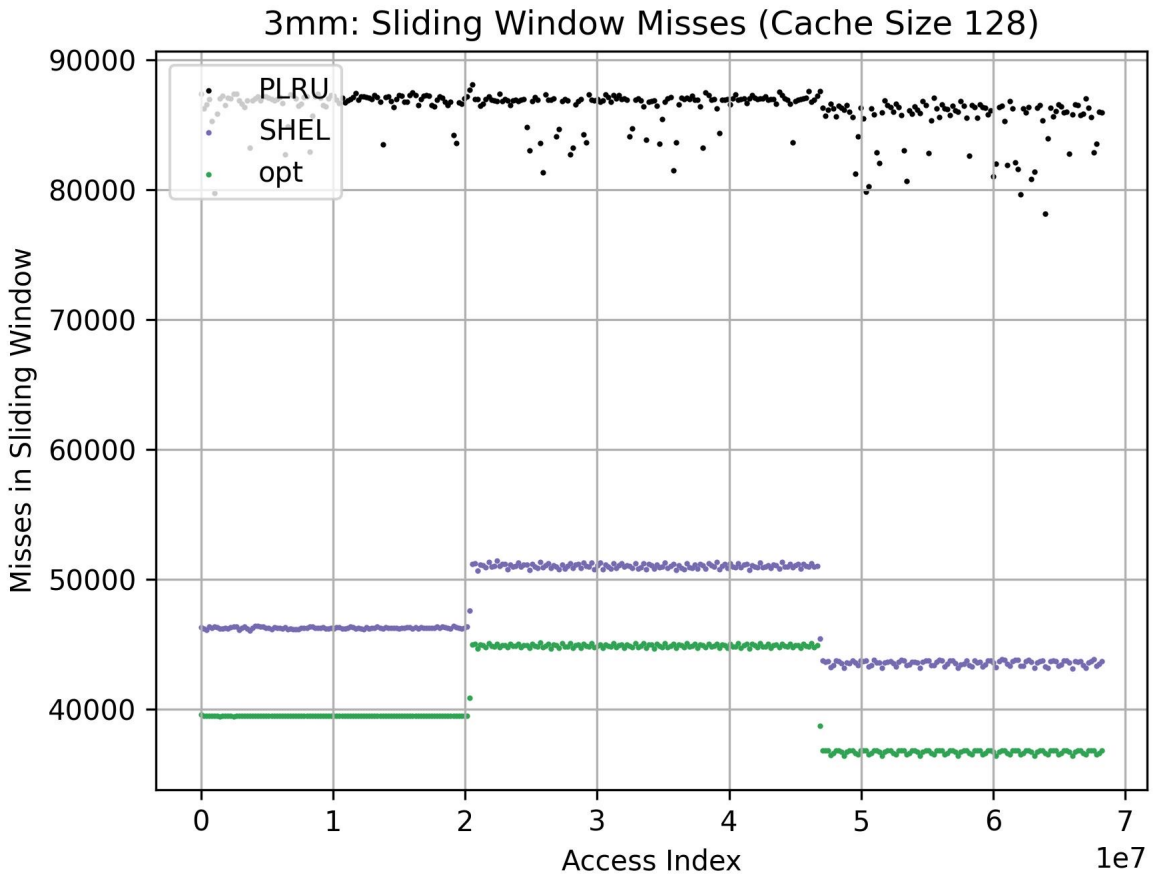
- Miss ratio comparison for PolyBench benchmarks (medium input, multi-scope lease, cache size 128)
- Programs are sorted by the Lease--OPT gap



Strengths of Lease Cache

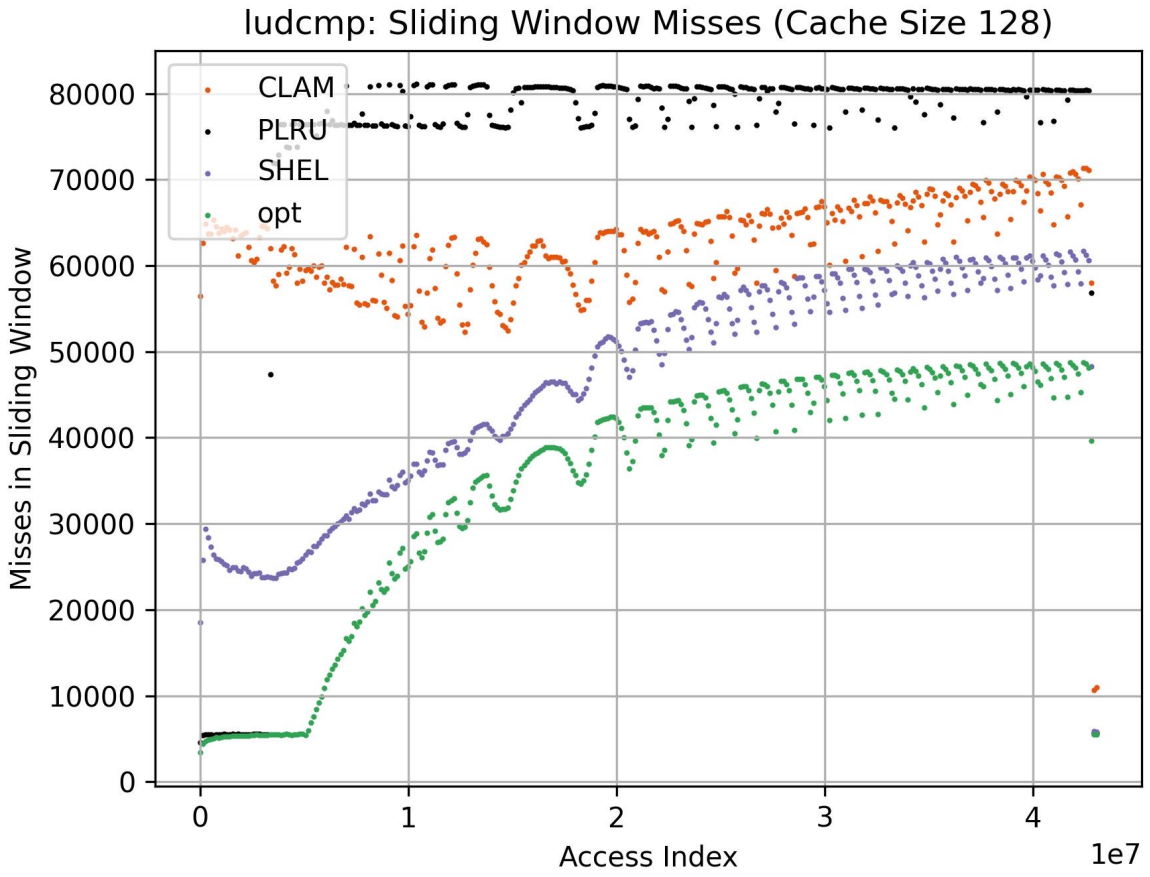
- Lease programming demonstrates its greatest benefit on ***programs with***:
  - **Frequent, regular reuse**: Where compiler or profile-based analysis can assign leases that tightly match reuse intervals (e.g., matrix multiplication kernels)
  - **Multi-phase locality**: Multi-scope lease assignment (SHEL) adapts leases across different program phases, overcoming limitations seen with both hardware policies (PLRU) and single-scope lease assignment (CLAM)

- **Example**: Sliding window miss counts for ***3mm*** (cache size 128, window overlap 25%)
  - Distinct steady phases are visible, reflecting strong alignment between lease scopes and true locality phases



Limitations of Lease Cache

- **Example**: Sliding window miss counts for ***ludcmp*** (cache size 128, window overlap 25%)
  - More ***variable and irregular locality limits the effectiveness*** of even multi-scope lease assignment, but SHEL still outperforms CLAM and PLRU
  - Even advanced techniques like SHEL, which assigns leases per program phase, can struggle because ***static scope boundaries*** may not align with the true, fine-grained changes in locality
- A key fundamental limitation is that **OPT leverages perfect knowledge** of the entire future access trace
  - For example, OPT knows precisely when each cache block will be accessed next, or if it will never be accessed again
  - In contrast, lease cache must assign leases based on past or predicted behavior, without knowing the precise position of the last access

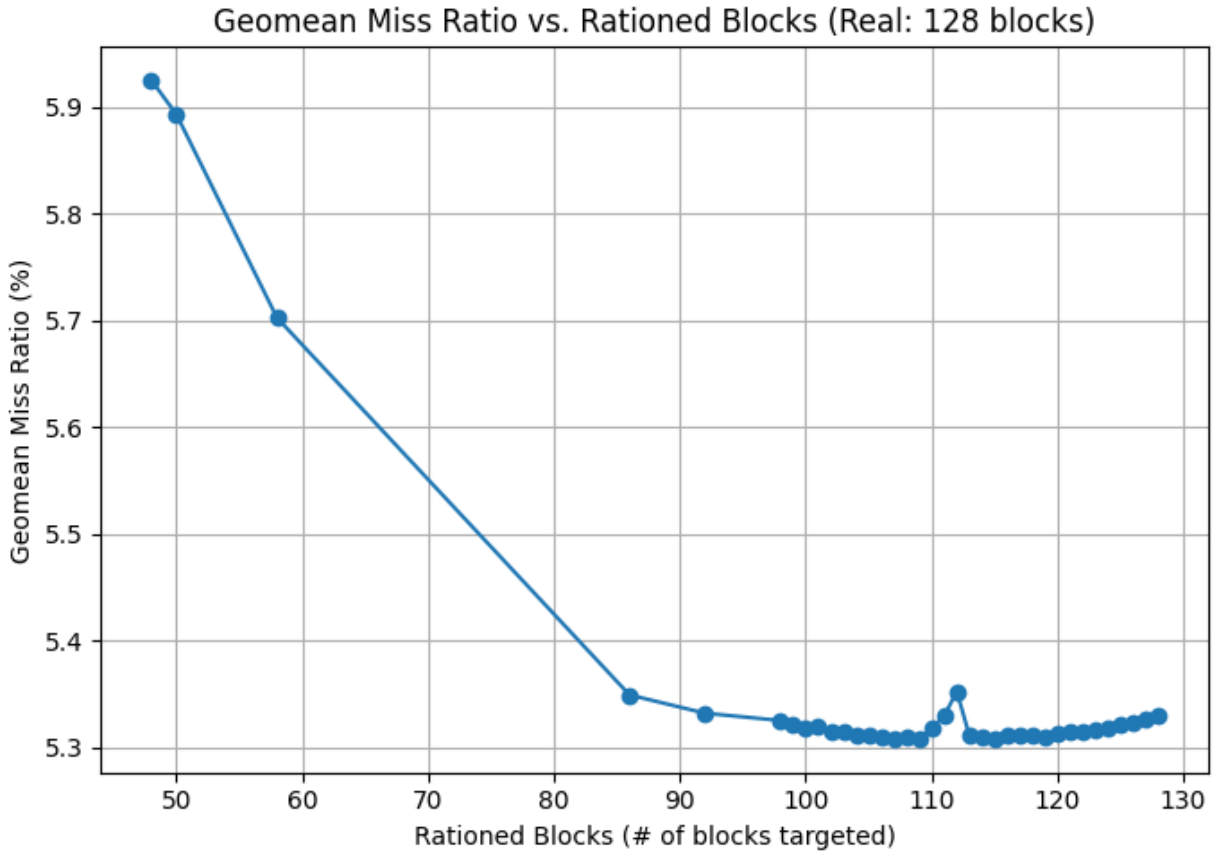


Lease Cache Rationing

- Considers the robustness of lease cache under resource under-allocation, a key practical concern in ***shared systems***

- **Observations**:
  - ***Rationing is safe and may be beneficial***
  - Insensitivity near the optimum
  - Implications for multi-programmed environments
    - These findings support the strategy of ***conservative lease assignment in environments with uncertain or fluctuating cache availability***
    - By under-reporting the cache size during lease table generation, programs can guard against transient cache pressure due to other workloads or system events, without significant risk of performance loss

- **Graph**: Geometric mean miss ratio across PolyBench benchmarks vs. rationed block count (actual cache: 128 blocks); Miss ratio is minimized with a slight under-allocation, and remains flat near the ground truth.



Conclusions and Acknowledgements

- **Lease**-based cache programming **closes** much of the **gap to OPT**, especially for programs with **regular structure and multi-phase locality**
- However, more space for **improvement** remains **necessary** for irregular or triangular access patterns
- Additionally, lease **cache rationing is robust** to resource under-allocation, making it practical in shared environments

- The research is partly supported by the **National Science Foundation** (Contract No. **SHF-2217395, CCF-2114285, CCF-2114319**)
- Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

RIT | Rochester Institute of Technology

Questions and Notes