Benchmarking Cache Programming Against Optimal Caching

Vincent Michelini*
Rochester Institute of Technology
Rochester, New York, USA

Yanhui Wu*
University of Rochester
Rochester, New York, USA

Chen Ding
University of Rochester
Rochester, New York, USA

Dorin Patru
Rochester Institute of Technology
Rochester, New York, USA

ABSTRACT

The cost and performance of a modern computing system depend on its memory hierarchy. Manual management is complex and not portable. Automatic management is sub-optimal — it reacts to program behavior, but does not directly utilize program knowledge. Recent work has developed cache programming for a type of cache called Lease Cache.

This paper empirically evaluates cache programming using leases vs. optimal caching and shows that cache programming significantly narrows the performance gap between conventional and optimal caching. Multi-scope programs (with sequential loop nests) outperform single-scope cases in both consistency and efficiency from cache programming. In contrast, triangular loop nests present a greater challenge than rectangular ones. The experimental framework reliably quantifies how cache size, program input, and loop structures impact cache optimization.

1 INTRODUCTION

Current solutions for the memory hierarchy are largely bipolar: they are either completely programmatic, requiring program data and accesses known at compile time, or completely automatic as caches, which have no program information but the past accesses. It is difficult for program control to adapt to changing resources. It is not portable from system to system. Even on the same system, the available memory changes with the workload.

Lease cache is a new type of cache that is *prescriptive* rather than reactive. At each access, a lease is assigned to the data block being accessed, and the data block is cached when the lease is active and marked for eviction when the lease expires. Prescriptive caching manages space by allocation, while reactive caching by replacement. The shift from replacement-driven to allocation-driven caching enables simpler and more efficient architectures and, more importantly, the ability to program a cache system.

In cache programming, leases are assigned in the program code based on the knowledge of the programmer or compiler of the program behavior and the available cache space. At run-time, the cache content is managed by hardware based on leases. Hence, it combines program control and automatic caching. Past work has developed collaborative caching through hints [3, 4, 9–11, 18]. However, hints influence, but do not directly control cache management.

In lease programming, the basic technique is called Compiler Assignment of Reference Leases (CARL)[7]. It assigns a lease for each reference. For each reference and each lease value, CARL evaluates the number of cache hits per unit of lease and assigns leases in decreasing order of the benefit per cost. CARL is optimal for a variable-size cache, where the cache grows without bound or shrinks dynamically.

This paper is concerned with fixed-size caches and presents an empirical evaluation of cache programming using leases vs. optimal caching. Optimal caching is called Belady or MIN [1, 6]. Since the Belady method requires full future knowledge of data accesses, it has to be simulated (in two passes) and cannot be implemented for a real cache. We present a setup that allows for a direct comparison between the results of the lease cache system in hardware and those from software simulation. In particular, we run a program twice on actual hardware. The first run extracts information for lease programming and the trace for the Belady simulation. The second run tests the cache performance of a program programmed with leases.

We use a benchmark suite called PolyBench, which includes 30 kernel programs containing various compute-intensive loop nests common in scientific computing, image processing, and linear algebra. We test two inputs, small and medium, on the lease cache of two sizes, 64 blocks and 128 blocks.

To quantify the effectiveness of cache programming, we use a normalized score (0–100) comparing its miss ratio to the baseline (a policy called PLRU) and the theoretical optimum. A score of 100 indicates that lease cache matches Belady (optimal), while a positive score signifies improvement over PLRU. The formula is symmetric for hit/miss ratios, and the score can be visualized on a vertical ruler where PLRU is 0 and Belady is 100, showing how close lease caching gets to ideal performance.

Some of the findings are as follows:

- The average score for all tests is 67. Therefore, cache programming is closing the gap, achieving two-thirds of the performance improvement between conventional caching and the theoretical optimum.
- The average score is higher in the 128-block cache than in the 64-block cache (71 vs 63) and slightly higher in small input than in large input (68 vs 66).
- About half of the tests have mainly a single loop nest (single-scope), and the rest have two or more loop nests one after another (multi-scope). The mean score is higher, 84 vs 53, and the standard deviation is lower in multi-scope than in single-scope tests.
- Most challenging for cache programming are triangular loop nests. Compared to programs with rectangular loop nests, the mean score is much lower, 49 vs 82, and the scores are more varied, shown by the higher standard deviation, 59 vs 18.

*Both authors contributed equally to this research.

We have also tested a conservative variant called *lease cache rationing*, which assigns leases for a cache size smaller than the actual size. The results show that cache programming is robust under modest underallocation.

In this work, the goal of cache design is minimal miss count for the same program using the same amount of cache space, without considering prefetching. While prefetching improves performance by overlapping memory access with computation or other memory accesses, it does not eliminate misses or reduce actual data movement — our goal is to reduce the access in memory by maximizing reuse in cache. By isolating the locality problem (where data reuse dictates performance), we evaluate cache designs without prefetching, treating every miss as equally costly. Minimizing misses directly translates into minimizing unnecessary data transfers.

2 LEASE CACHE SYSTEM

The lease cache system consists of two key components: a hardware prototype that supports programmable lease-based cache policies, and a compiler-level mechanism for assigning leases to memory references. This section describes both components: we begin with the hardware design, including its support for static and multi-phase lease programming, and then explain the underlying theory and practical computation of reference leases.

2.1 Hardware Prototype

We have designed, implemented, and tested a lease cache emulator, whose architecture is illustrated in Figure 1. In this preliminary hardware prototype, a single RISC-V core executing integer and floating point manipulation instructions is connected to a single-level cache. This is controlled by a Programmable Cache and Memory Management Unit (PCMMU), which can be configured to apply a conventional Pseudo-Least Recently Used (PLRU), or use reference leases (CLAM or SHEL). The prototype is based on the test platform used in previous work [8, 15, 16]. The figure shows two lease cache techniques: CLAM and SHEL. CLAM is used for single-scope programs whose leases are loaded once at the start of an execution. SHEL is for multi-scope programs whose leases are loaded multiple times during an execution.

The following table compares cache programming with conventional caching. A conventional cache uses a replacement policy such as least-recently-reused (LRU) replacement. Replacement is reactive. In comparison, the lease cache is prescriptive. A lease is assigned every time a data block is accessed, and the block is marked for eviction once the lease expires. A conventional cache is automatic and uses only the history information, while leases can be constant, i.e., uniform leases [5, 16], or programmed based on program information [7, 8, 17]. In addition, a lease cache has a secondary policy. If a program requires more cache space than what is available, the second policy randomly evicts a data block at a cache miss.[1]

Since a lease is based on allocation, lease programming is similar to memory allocation through calls to *malloc* and *free*. They both aim to optimize resource utilization, but for different purposes. In heap management, the goal is to minimize the size of a heap, but there is no fixed upper bound on heap size. In cache leasing, the goal is to stay within a constant bound and obtain as many cache hits as possible. Operationally, the lifetime of an object in a heap is one allocation. In comparison, a lease is assigned every time a block is accessed.

	Conventional cache	Lease cache
primary policy	replacement, automatic	reference leases, programmable
info used	history only	constant, or program analysis
secondary policy	N/A	random eviction

Table 1: Comparing between conventional and lease caches

The prototype is implemented using Intel Quartus Prime Standard 22.1 on an Altera Cyclone-V GT FPGA (5CGTFD9E5F35C7). The FPGA contains 113,560 adaptive logic modules, 12.5 Mbits of block memory, 342 DSP blocks, and 20 PLLs. On this device, we synthesized a RISC-V core compliant with the unprivileged ISA specification [19]. The core has a classic 5-stage pipeline (instruction fetch/decode, operand fetch, execute, memory, writeback), supports 32 general-purpose registers and 32 floating-point registers, and runs at 40 MHz. Both integer and single-precision floating-point operations are supported using Quartus IP blocks.

The data cache is direct-mapped with 32-byte lines, tested at sizes of 64 and 128 lines (2–4 KB total). Instruction cache uses the same configuration with PLRU replacement. The data cache can be configured to run with either PLRU replacement or lease-based policies (CLAM or SHEL). Off-chip memory is DDR3, accessed through a lightweight controller.

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.

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 cache programming in a controlled hardware environment and to quantify how closely it can approach the theoretical optimum. 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.

2.2 Reference Leases

A reference is defined as the instruction that invokes a memory access, i.e., the program counter for the load/store instruction. Given

[1] In Appendix A in their MEMSYS 2020 paper, Prechtl et al. [16] compared random eviction (lease oblivious) and two other policies, shortest remaining lease and longest remaining lease. Through experiments, they found that "no one policy dominates another. Among them, random is the most space efficient and fastest to implement in hardware."

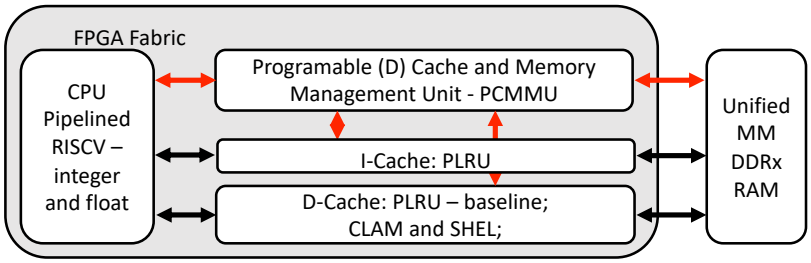


Figure 1: Lease Cache Hardware Prototype.

a program, a compiler assigns a lease to each reference. When the program executes, each access is given the lease of its reference. The lease of each reference is derived from two attributes for a reference: its *Reuse Interval Distribution (D)* and its *Access Ratio (AR)*. Next, we first define one basic term, Reuse Interval (RI), and then define these two per-reference attributions.

Definition 2.1 (Reuse Interval (RI)). 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 $RI = 5$.

Definition 2.2 (Reuse Interval Distribution (D)). 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.

Definition 2.3 (Access Ratio). The Access Ratio of a reference is the portion of all accesses in the trace that are invoked by that reference.

Ding et al. [7] present Compiler Assigned Reference Leasing (CARL), an optimal algorithm for assigning reference leases to a program such that the miss ratio is minimized, assuming a variable-sized cache which can store any number of leases at a time, so long as the average number of active leases throughout execution is equal to some target cache size. We call such a cache system a *virtual cache*, because its storage capacity is unbounded. Furthermore, we refer to the number of active leases in the virtual cache as the *virtual cache size (VCS)*. While virtual cache size may grow and shrink dynamically throughout execution, the physical cache size (PCS) remains constant.

We examine two CARL-based lease assignment techniques from previous work.

CLAM Compiler Lease of Cache Memory, which applies CARL, setting the average VCS of the whole program to be PCS, using random eviction at a cache miss when the cache is full and has no expired blocks [16].

SHEL Scope-Hooked Eviction Leasing, which applies CARL at each loop nest (scope), setting the VCS at each scope to be PCS. SHEL ignores cross-loop reuses [17].

SHEL assigns different leases to respond to variation in program behavior by dividing a program by its loop nests.

3 EXPERIMENTAL SETUP

To obtain experimental results, the small and medium input sizes of the PolyBench [14] suite is compiled, using `GCC -O3` optimization level without vectorization, to run on a RISC-V core configured on a Cyclone-V FPGA. The memory hierarchy consists of a single-level separate instruction and data caches, and a DDR3 off-chip main memory. Figure 1 shows both the RISC-V core and the memory hierarchy.

The data cache is direct-mapped and was evaluated at sizes of 64 and 128 lines. Each cache line is 64 bytes (16 words), giving total capacities of 4 KB and 8 KB, respectively. Instruction cache uses the same line size and replacement policy (PLRU). For lease-cache experiments, the data cache can be configured to apply either PLRU or lease-based policies (CLAM or SHEL).

3.1 Data Collection

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. Figorito et al. [8] describes the emulation and test controller (ETC) used to obtain cache results and sample data. At a high level, a program on a host computer will read and write data in DDR3 memory or memory-mapped registers (MMR). The data transfers occur via a universal asynchronous receiver transmitter (UART) communication channel that connects the host computer and the FPGA. In this way, the program running on the host computer controls benchmark execution and metric/results collection. MMR's are hardware registers that are accessed using memory space mapped addresses. They are primarily used for software-hardware interactions, allowing the core to use memory operations to interact with emulation and test support peripheral hardware units.

There are three main code sections during each benchmark execution that are relevant to data collection. These are the setup, kernel, and cleanup code.

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 will write 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.

3.1.1 PLRU Results Collection. PLRU benchmark results are obtained by first configuring the FPGA with both data and instruction caches using PLRU eviction policies. Then, the compiled benchmarks are written to the DDR3 memory by the host program. After this, the host program will write to an MMR on the FPGA, enabling the RISC-V core and starting benchmark execution. The host program begins to poll the test-complete MMR at this time. After the execution of the benchmark kernel, the core writes to the test-complete MMR signaling that the benchmark is finished. Once the host program observes that the test is complete, the core is disabled, and results are read from the metrics collection system MMRs.

3.1.2 Collecting Data Access Sample Information. The leases used by the lease cache are generated based on data access information collected by sampling the benchmark kernel code running on a PLRU cache. The hardware element that collects sample data is simply called *the sampler*.

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.

Sample Table stores memory references that have not seen a reuse yet. Once a reuse is seen, its forward reuse interval is calculated and written to the sample buffer.

Sample Buffer stores complete sample data after it is written from the sample table.

The sample buffer may become full during kernel code execution, stalling the processor and signaling to the host program to read (empty) the sample buffer. Completion of the kernel code will cause the host program to read the sample buffer, read the sample table, and re-read (re-empty) the sample buffer.

3.1.3 Collecting Trace Data. 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. The hardware unit that performs this collection is called *the tracer*. The tracer contains a trace buffer, in which results are collected until it is full. Once full, the processor core is stalled and the host program reads (and empties) the buffer. This process is repeated until the end of the benchmark's kernel code.

3.1.4 Lease Cache Results. Benchmarks that are run with one of the lease cache policies are collected in a similar manner to PLRU collection. However, after the host program enables the core the lease cache will stall the processor and read in the lease table from a predetermined section of memory. After the table is read in, the processor is un-stalled and execution proceeds normally. The lease cache metrics system now collects the number of expired and forced evictions. Once the kernel code is complete, the host program reads all metrics.

3.1.5 Accounting for the Initial Cache State. When miss counting begins at the start of the kernel code, the cache is already partially filled with program data. As a result, some cold-start misses are recorded as cache hits, causing the lease cache's miss count to occasionally fall below the theoretical optimum (Belady's algorithm, which simulates an idealized cache, accounts for all cold-start misses). While the exact impact of this discrepancy is difficult to quantify, we approximate it by adding a fixed number of misses equal to the cache size to each lease-cache run.

This adjustment ensures our miss count is a conservative estimate: it is always equal to or higher than the actual miss count. On the one hand, it cannot exceed the theoretical limit. On the other hand, when lease caching is optimal, its miss count can be higher than optimal due to overcounting.

3.2 Optimal Caching

To provide a theoretical lower bound on the cache miss ratio for each benchmark, we implemented an optimal offline caching simulator in Rust, modeling Belady's MIN algorithm (also called OPT) [1]. The MIN policy is not implementable in real hardware because it requires knowledge of the entire future access sequence. However, it serves as the gold standard against which all practical policies are compared and represents the theoretical lower bound for cache miss ratios in fixed-size caches.

Rust Simulator Implementation. Our Rust simulator takes a memory access trace as input and produces, for any fixed cache size, the precise sequence of cache hits and misses, as well as the miss ratio under optimal caching. The core logic follows the canonical Belady's MIN approach. For each access, the simulator determines which cached block will be needed farthest in the future (or not at all) and evicts that block if a miss occurs and the cache is full. This process guarantees the minimum possible number of cache misses for a given trace and cache size.

Trace Collection and Preprocessing. Full memory access traces are collected as described in Section 3.1.3, including the program counter, word address, and PLRU outcome for every memory access in the benchmark kernel. For the purposes of OPT simulation, only the memory addresses are used. Addresses are mapped to fixed-size cache block addresses (by dividing the word address by the number of words per cache block), yielding a block access trace as input to the simulator.

Belady's MIN Algorithm. The MIN algorithm operates by always evicting the cached block whose next reuse is farthest in the future (or, equivalently, that will not be used again for the longest time). To enable this, we first pre-process the access trace to compute, for each access, when the next occurrence of each block will be. This is also known as the forward reuse interval. Specifically:

(1) **Forward Pass (Preprocessing):** We scan the access trace backwards (from the last access to the first) and, for each block, record the index of its next occurrence. This produces a mapping from each access position to the next access of the same block.
(2) **Optimal Caching Simulation:** We simulate the cache of a given size as follows. For each memory access:
    - If the block is present in the cache (a hit), we proceed.
    - 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.

This process is repeated for the entire trace, and the miss ratio is calculated as the total number of misses divided by the total

number of accesses. This procedure is repeated for each cache size of interest to generate the complete OPT miss ratio curve.

OPT Caching Rationale. The OPT policy 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. Any practical cache management policy—whether hardware-based (PLRU), software-based (Lease Cache), or hybrid—will have a miss ratio that is at least as high as OPT for the same trace and cache size. By comparing Lease Cache and PLRU against the OPT baseline, we can quantify the gap between realistic, programmable, and theoretical caching strategies.

3.3 PolyBench Suite

The PolyBench/C 4.2.1 [14] 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.

Out of the 30 available benchmarks, 29 were chosen for analysis. `jacobi-1d` is excluded due to the small data size and zero miss ratio when run on the four input configurations (two input data sizes and two cache sizes). The input array contains 120 and 400 single-precision floating-point elements for small and medium, respectively. All input combinations resulted in a 0% miss ratio as the entire array fits inside the cache and remains present after initialization.

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.

4 RESULTS AND ANALYSIS

Prior work, such as CLAM [16], demonstrated the feasibility and correctness of using CARL-assigned leases in fixed-size caches, but did not systematically compare lease cache performance to the optimal policy (OPT). In this work, we close that gap by systematically evaluating lease cache performance under a range of practical fixed-size conditions, directly comparing it to both PLRU and the OPT baseline. We examine the effects of input size, cache size, and scope granularity in lease assignment. We want to clarify both the potential and the limits of lease-based cache management in practical scenarios. Throughout this section, we first quantify overall performance using a normalized score, then examine how loop structure and phase granularity impact effectiveness, and finally assess the robustness of lease policies under practical resource tightening.

4.1 Performance Scores

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

Let $mr_{policy}(c)$ be the miss ratio at cache size c for one of the three caching policies: lease programming, PLRU, and OPT. The score for lease cache performance is

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

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. It is intuitive to visualize the score of hit ratios. Imagine that a score marks the position on a vertical ruler where the PLRU hit ratio is the baseline or the zero mark, and the OPT hit ratio is the top line or the 100 mark.

We score the test programs on two input sizes (small, medium) and two cache sizes (64 lines and 128 lines). Table 4 shows the arithmetic average in these four cases.

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. This trend reflects the fact that smaller working sets fit better in larger caches, leaving less room for further improvement from better caching policies.

To further illuminate the sources of variation, we analyze the impact of program scope granularity. A test program is either single-scope or multi-scope. Table 2 shows a statistical summary of the scores for these two groups for the medium size input and for two cache sizes. Multi-scope programs have clearly higher and more stable scores: means of 85 or higher, with standard deviations around 10, compared to means near 50 and standard deviations around 64 for single-scope. In comparison, single-scope programs have extreme min/max values, from -181.61 to 99.99. In addition, 128-line cache sees higher multi-scope program scores, with a mean of 87 vs. 83 for 64-line.

Table 3 extends the comparison across input sizes. Multi-scope "superiority" holds for both small (mean 84 in multi vs 56 in single) and medium inputs (85 vs 50). Multi-scope "stability" also holds, as shown by the low std deviation. Combining the effect of both inputs, the medium input scores have greater variability than small input scores, with a higher std deviation, 51 vs 39. This is somewhat surprising, since small inputs are more sensitive to cold starts and other initialization or finalization effects.

Combining the results from both tables, multi-scope lease programming consistently scores 30–35% higher and is 5x more stable than single-scope, regardless of input or cache size. On both cache sizes, the median score is over 85 for medium input but 66 or lower for small input. The min score is 47.

We show individual scores in Appendix A and miss ratio results in Appendix B and may refer to them in the following discussion about individual tests.

4.2 Impact of Loop Nest Shape on Lease Cache Effectiveness

To better understand these variations, we next examine how loop nest shape influences lease cache performance. A key differentiator is the geometric structure of loop nests that also plays a direct and significant role in the effectiveness of lease cache optimization.

Triangular vs. Rectangular Loop Nests. A consistent pattern emerges across our benchmarks: lease cache provides less improvement for codes with triangular loop nests—those in which inner loop bounds depend on the index of an outer loop variable (e.g., for $i = 0$ to N,

Inputs	Cache Size	Scope	mean	std	min	25%	50%	75%	max
medium	Both Sizes	Single	49.85	63.88	-181.61	46.86	65.22	87.38	99.99
medium	64	Single	49.99	63.23	-103.99	48.87	63.82	93.89	99.99
medium	128	Single	49.71	66.61	-181.61	46.85	66.37	79.35	99.47
medium	Both Sizes	Multi	85.07	10.68	56.78	83.76	88.29	91.88	99.47
medium	64	Multi	83.37	12.98	56.78	79.09	84.64	92.45	99.47
medium	128	Multi	86.77	7.94	68.78	85.92	89.90	91.50	93.63

Table 2: Statistical summary of scores for single- and multi-scope tests for medium input

Inputs	Cache Size	Scope	mean	std	min	25%	50%	75%	max
small	Both Sizes	Both	68.46	38.52	-163.16	58.04	79.93	87.67	99.72
small	Both Sizes	Single	56.12	47.74	-163.16	45.52	66.45	81.43	99.72
small	Both Sizes	Multi	83.63	11.06	46.79	77.74	85.30	90.32	98.54
medium	Both Sizes	Both	65.64	50.81	-181.61	60.77	81.32	91.30	99.99
medium	Both Sizes	Single	49.85	63.88	-181.61	46.86	65.22	87.38	99.99
medium	Both Sizes	Multi	85.07	10.68	56.78	83.76	88.29	91.88	99.47

Table 3: Statistical summary of scores for small and medium inputs for both cache sizes

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

Table 4: Arithmetic means for two inputs and two cache sizes

for $j = 0$ to i). In contrast, programs with rectangular or other non-triangular loop nests (where loop bounds are independent) achieve much higher cache performance gains or high scores under lease-based management. Table 6 classifies the PolyBench programs used in our study according to whether they exhibit triangular or rectangular loop nests. Table 5 then summarizes the statistical distribution of lease cache performance scores for each group.

Lease cache achieves a 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.

Since the loop nest iteration space "shrinks" along one dimension in triangular loops, the number of memory accesses per inner loop iteration decreases as the outer loop index increases. This leads to a much wider and more uneven distribution of reuse intervals across the execution. Some references are reused very soon, while others experience much longer gaps between uses, especially near the end of the triangular region. As a result, the RI histogram becomes more spread out, and lease assignment based on average statistics may be less effective, leading to bigger performance variability compared to regular, rectangular iteration spaces.

4.3 Miss Ratio Breakdown by Scope Assignment

We now examine detailed miss ratio results for lease assignment under two scope granularities: single-scope (CLAM) and multi-scope (SHEL).

4.3.1 Single-Scope Assignment (CLAM). Figure 2 shows the miss ratios for the 16 single-scope PolyBench benchmarks (medium input, 128-line cache). Overall, CLAM narrows the gap between PLRU and OPT, but the degree of improvement is variable across benchmarks.

For each benchmark, we compare three policies:

- **PLRU**: pseudo-LRU hardware replacement policy,
- **Lease (CLAM)**: compiler-assigned single-scope leases,
- **OPT**: Belady's theoretical minimum miss policy.

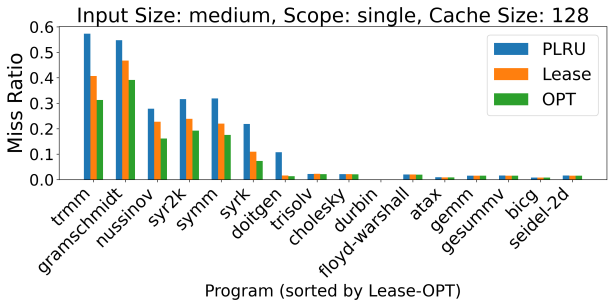


Figure 2: Miss ratio comparison for PolyBench single-scope benchmarks (medium input, 128-line cache). Programs are sorted by the Lease–OPT gap.

At the high end, programs such as `doitgen`, `syrk`, and `symm` show clear reductions in miss ratio: CLAM achieves up to 84% fewer misses compared to PLRU, consistently moving performance closer to OPT.

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

Table 5: Statistical summary of scores for rectangular and triangular loop nest programs

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	

Table 6: PolyBench benchmarks classified by loop nest shape.

A second category consists of benchmarks where PLRU, CLAM, and OPT curves nearly overlap. Here, the absolute miss ratios are very small, so even minute differences in miss counts can translate into disproportionately large variation in normalized scores. For example, several kernels show CLAM within < 0.005 miss ratio of PLRU, yet still receive volatile scores because of the normalization formula. Despite this, CLAM generally provides a small but consistent advantage in raw miss count.

The only benchmark where CLAM produces a negative score is `trisolv`. In this case, the PLRU miss ratio is 0.012813, CLAM's miss ratio is slightly higher at 0.013094, while OPT achieves 0.012536. Although this results in a poor normalized score, the raw difference is just 0.000281 in miss ratio. Given a total trace length of 419,035 memory accesses (medium input), this corresponds to only 118 additional misses. Thus, while CLAM technically underperforms PLRU in this isolated case, the absolute impact on execution is negligible.

4.3.2 Multi-Scope Assignment (SHEL). Figure 3 shows the corresponding results for 13 multi-scope benchmarks using SHEL instead of CLAM. Here, the lease table is reloaded at the start of each scope, allowing leases to adapt across different program phases. We again compare PLRU, SHEL, and OPT.

Benchmarks are sorted in descending order of the lease result–OPT gap, so the leftmost bars represent programs where lease caching leaves the most room for improvement.

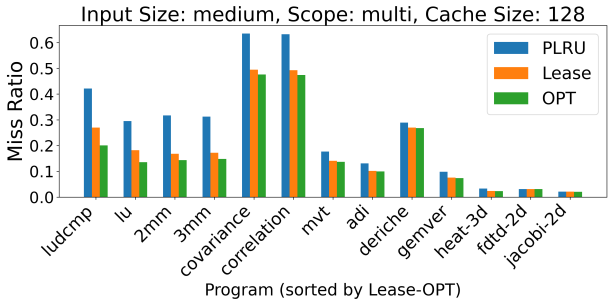


Figure 3: Miss ratio comparison for PolyBench benchmarks (medium input, multi-scope lease, cache size 128). Programs are sorted by the gap between the Lease result and OPT

Across multi-scope benchmarks, the lease cache policy consistently narrows the gap between PLRU and OPT, yielding substantial reductions in miss ratio. For instance, in `ludcmp`, `2mm`, and `covariance`. In contrast, for benchmarks with high temporal locality or highly predictable access patterns (e.g., `gemver`, `adi`), all policies perform similarly and closely track the OPT baseline, as the room for improvement is inherently limited. SHEL adapts more effectively to phase changes and yields a smoother miss profile than CLAM. Overall, multi-scope adaptation improves both the mean score and stability, with fewer extreme outliers than in the single-scope case.

For completeness, the full set of miss ratio results for all PolyBench benchmarks—across both input sizes and cache sizes—is provided in the appendix. In these four summary plots (Figure 9, Figure 10, Figure 11, Figure 12), both single-scope and multi-scope programs are displayed together, with different colors used to distinguish the scope assignment of each benchmark.

4.4 Strengths and Limitations of Lease Cache.

The preceding analyses highlight both the gains and limitations of lease assignment. Here, we summarize the key scenarios where lease cache is especially effective. 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).

4.4.1 Multi-Scope Lease Assignment: SHEL vs. CLAM. A key strength of lease cache is its flexibility in adapting to diverse and evolving reuse patterns. CLAM assigns a single set of leases for the entire program, implicitly assuming uniform reuse behavior across all phases. This works well for programs with steady, predictable locality, but falls short in real-world codes with changing or irregular working sets.

SHEL extends lease programming by dividing program execution into multiple logical phases—typically by loop nest or other code structure—and assigning leases independently within each phase. This allows the lease cache to adjust more precisely to each phase's locality characteristics, minimizing both under- and over-retention. This advantage is clearly illustrated in Figure 4 and Figure 5.

- In `3mm` (Fig.4), the miss count forms distinct steady plateaus, corresponding to phases with different but internally consistent reuse patterns. SHEL tracks these phases, resulting in stable, phase-aligned miss counts significantly lower than those achieved by PLRU.
- In contrast, programs with more complex or dynamically shifting locality, such as `ludcmp` (Fig.5), still benefit from SHEL's finer adaptation compared to CLAM.

SHEL's multi-scope assignment bridges much of the remaining gap to OPT in phase-structured codes and represents a robust, scalable improvement over single-scope approaches.

4.4.2 Limitations and Remaining Gaps. Despite substantial improvements, lease cache policies still fall short of achieving the theoretical lower bound set by OPT. The most pronounced gaps are observed in benchmarks such as `ludcmp`, `lu`, and `syr2k`, which are characterized by highly irregular or dynamically shifting reuse patterns. In these cases, 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. This "phase misalignment" limits how effectively lease assignment can match actual program behavior, leaving a persistent gap to OPT.

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. This allows OPT to evict a block at the exact moment of its last use, ensuring that no cache space is wasted on data with no future benefit. In contrast, lease cache must assign leases based on past or predicted behavior, without knowing the precise position of the last access. As a result, the lease cache will treat all accesses from a given reference similarly, often reserving space for blocks that are about to become dead, thus incurring avoidable misses and suboptimal utilization.

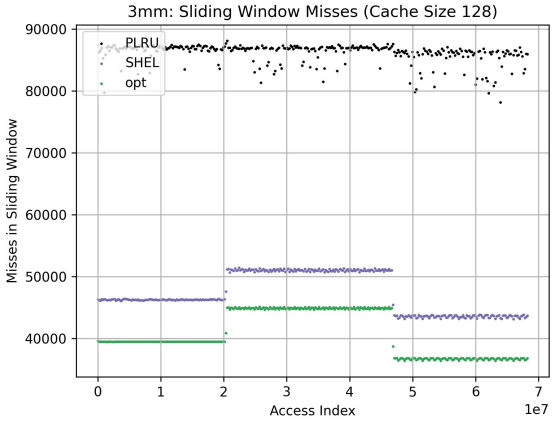


Figure 4: Sliding window miss counts for `3mm` (cache size 128, window overlap 25%). Distinct steady phases are visible, reflecting strong alignment between lease scopes and true locality phases.

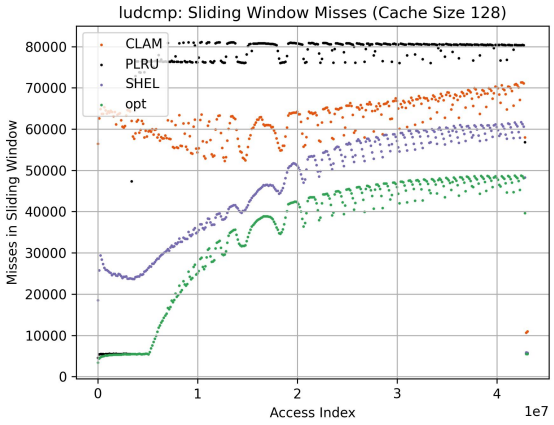


Figure 5: Sliding window miss counts for `ludcmp` (cache size 128, window overlap 25%). More variable and irregular locality limits the effectiveness of even multi-scope lease assignment, but SHEL still outperforms CLAM and PLRU.

4.5 Lease Cache Rationing

Finally, we consider the robustness of lease cache under resource under-allocation, a key practical concern in shared systems. In real environments, the amount of cache available to any single program can vary over time, especially in shared environments where multiple programs or processes compete for cache resources. This variability can result from dynamic allocation, context switches, or interference from other workloads. As a result, the actual cache

available to a program may fluctuate below its nominal or "advertised" capacity.

To evaluate the robustness of lease cache policies under such conditions, we systematically varied the target cache size used for lease assignment—rationing the number of blocks reported to the lease assignment algorithm—while keeping the actual hardware cache size fixed (128 blocks for this experiment). We compared the miss ratios obtained by targeting from as little as 50% of the real capacity (i.e., 64 blocks) up to the full size.

Figure 6 shows the geometric mean miss ratio across all PolyBench benchmarks as a function of the rationed block count. Several trends are immediately apparent:

- **Rationing is safe and may be beneficial.** The miss ratio steadily decreases as the rationed block count increases, reaching a broad, flat minimum when the target is set just below the actual cache size (e.g., around 100–110 blocks). This valley shows that cache programming is robust under modest underallocation.
- **Insensitivity near the optimum.** There is little penalty for choosing a rationed size slightly below the true cache size.
- **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.

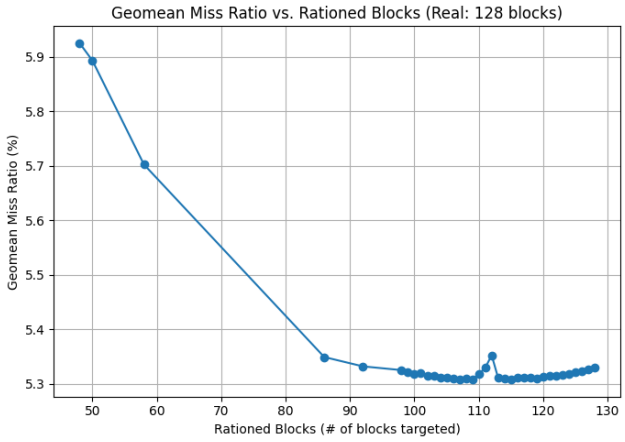


Figure 6: Geometric mean miss ratio across PolyBench benchmarks vs. rationed block count (actual cache: 128 blocks). Miss ratio is minimized with a slight under-allocation, and remains flat near the ground truth.

4.5.1 Why Rationing Works. One of the key reasons why lease cache rationing is effective—and robust to modest under-allocation—lies in the cache's eviction policy. In a lease cache, eviction is not triggered immediately when a lease expires. Instead, eviction only occurs when space is needed for an upcoming access. The system always prefers to evict blocks whose leases have expired (expired eviction) before resorting to the more costly forced eviction of still-active blocks.

When the lease generator is told a cache size smaller than the true hardware capacity, it compensates by assigning generally shorter leases to all references. This results in more blocks whose leases expire earlier, and therefore, a greater number of blocks residing in the cache with expired leases. However, these expired blocks are not immediately evicted. Instead, they remain in the cache and can still be accessed (hit) as long as there is spare capacity. Eviction only takes place if the cache becomes full and a new block must be inserted, at which point expired blocks are evicted first.

This decoupling of lease expiration and actual eviction allows the cache to absorb small inaccuracies in reported cache size without a significant performance penalty. If the cache is under-allocated (due to rationing), expired eviction provides a buffer: the cache can opportunistically retain blocks with expired leases, extracting additional hits, until memory pressure demands their removal. Only if all expired blocks have been evicted and more space is still required does the system perform forced eviction of active leases. The overall design of the eviction policy allows lease cache to handle modest discrepancies between the target (rationed) and actual cache size.

4.6 Analysis Conclusion

In summary, 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 for irregular or triangular access patterns. Additionally, lease cache rationing is robust to resource under-allocation, making it practical in shared environments.

5 RELATED WORK

Variable-size Caching. Ding et al. [7] studied the theoretical properties of lease programming and showed that it is optimal under certain statistical assumptions, along with miss curve convexity and sub-partitioning monotonicity. They evaluated the potential using PolyBench and showed similar or better cache utilization (in variable size cache) than the optimal fixed-size caching policy. They used compiler analysis to collect RI distributions for lease programming. Their technique does not require running a program. However, they did not measure the effect of lease programming in a fixed-size cache.

Collaborative Caching. Collaborative caching adds a hint at a memory instruction. A number of hardware systems have been built or proposed to provide an interface for software to influence cache management. Examples include cache hints on Intel Itanium [3], bypassing access on IBM Power series, and evict-me bit [18]. Wang et al. [18] called a combined software-hardware solution collaborative caching. Gu and Ding [10] developed the bipartite LRU-MRU cache model where software can dynamically choose between LRU and MRU policies for individual memory accesses. The paper proves that the bipartite cache maintains the inclusion property, introduces the LRU-MRU Stack Distance, and gives a one-pass algorithm to compute stack distances for mixed LRU-MRU workloads. As a programming technique, it gives a compiler technique called PACMAN which uses profiling of OPT training to tag each load and store

instruction as either LRU or MRU. Through simulation, they found that OPT reduces the miss ratio by 24% compared to LRU, and PACMAN achieves 50% of the reduction. This corresponds to a score of 50 in our study.

PACMAN hints and lease programming differ in two other aspects. First, PACMAN requires OPT simulation which is costly. Second, the LRU-MRU hints are specific for a given execution and must be re-programmed if the input or the cache size changes. In comparison, lease programming uses program analysis and does not simulate optimal caching. Therefore, lease programming exerts greater control over cache management than cache hints while maintaining superior programmability. Based on reported data, lease programming is more effective than PACMAN (67 vs. 50). Finally, we note that the hardware-simulation framework in this paper enables a direct comparison with the reported data from prior work.

For loop code, a compiler can analyze the data access patterns and insert cache hints accordingly. Beyls and D'Hollander [3] developed a technique called reuse distance equations, and Brock et al. [4] a technique based on the linear patterns of OPT stack distances. In both cases, they use code replication by branching [2] or splitting loops [4]. There is at most one hint at each instruction, so code replication increases the resolution of "collaboration." Ding et al. [7] defined a property called sub-partitioning monotonicity which means that cache programming never loses performance by this type of code replication.

Optimal Caching Algorithms. The theoretical basis for cache replacement is rooted in Belady's MIN/OPT algorithm [1], which provides the lowest possible miss ratio for any access trace under a fixed-size cache size constraint by always evicting the block whose next use is farthest in the future. Mattson et al. [13] introduced the inclusion property and showed that caching solutions that have the property can be evaluated using a stack algorithm to obtain the miss ratio for all cache sizes in a single pass over the trace. OPT requires an additional pass for pre-processing. Our implementation of OPT caching uses a stack algorithm.

OPT-inspired Practical Replacement Policies. While Belady's MIN/OPT provides a theoretical optimum, it is not directly implementable. Recent work has explored how hardware policies can approximate OPT by learning from past behavior. Jain and Lin's Hawkeye policy [12] is a good example: Hawkeye reconstructs Belady's OPT solution for past cache accesses and uses this to guide future replacement decisions. Their OPTgen component efficiently simulates OPT's decisions for a long window of accesses, using sampling and liveness intervals, and then uses a PC value to classify lines as cache-friendly or cache-averse. Hawkeye achieves higher performance than prior RRIP-based and heuristic policies, especially on memory-intensive workloads. Notably, their results confirm that no heuristic policy can match OPT across all workloads, reinforcing the value of program-guided or OPT-inspired approaches. Our work is complementary: rather than learning heuristics, lease cache aims for prescriptive cache management using program analysis and leasing, and we provide a direct, empirical comparison of lease-based programming against both PLRU and OPT for a range of scientific codes.

6 SUMMARY

This paper presents an empirical evaluation of cache programming using leases vs. optimal caching. It first describes an experimental framework that reliably quantifies how cache size, program input, and loop structures impact cache optimization. By testing the full suite of PolyBench, it shows that cache programming achieves an average score of 67, closing two-thirds of the gap between conventional and optimal caching, with better performance in 128-block caches (71 vs. 63 for 64-block) and small inputs (68 vs. 66). Multi-scope tests (multiple loop nests) show higher and more consistent scores (mean 84 and std dev 11 multi-scope vs. 53 and 64 for single-scope), while triangular loop nests prove challenging, scoring lower (49 vs. 82) with greater variability (std dev 59 vs. 18). In addition, we have developed and tested lease cache rationing and found that cache programming is robust under modest underallocation.

ACKNOWLEDGMENTS

Ben Reber participated in the early part of the project. The authors also wish to thank the MEMSYS reviewers for their questions and suggestions, which have improved the presentation of the paper. 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.

REFERENCES

[1] L. A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 5, 2 (1966), 78–101.
[2] K. Beyls. 2004. *Software methods to improve data locality and cache behavior.* Ph. D. Dissertation. Ghent University.
[3] Kristof Beyls and Erik H. D'Hollander. 2005. Generating cache hints for improved program efficiency. *Journal of Systems Architecture* 51, 4 (2005), 223–250.
[4] Jacob Brock, Xiaoming Gu, Bin Bao, and Chen Ding. 2013. Pacman: Program-Assisted Cache Management. In *Proceedings of the International Symposium on Memory Management.*
[5] Dong Chen, Chen Ding, Fangzhou Liu, Benjamin Reber, Wesley Smith, and Pengcheng Li. 2021. Uniform lease vs. LRU cache: analysis and evaluation. In *ISMM '21: 2021 ACM SIGPLAN International Symposium on Memory Management, Virtual Event, Canada, 22 June 2021*, Zhenlin Wang and Tobias Wrigstad (Eds.). ACM, 15–27.
[6] Edward G. Coffman Jr. and Peter J. Denning. 1973. *Operating Systems Theory.* Prentice-Hall.
[7] Chen Ding, Dong Chen, Fangzhou Liu, Benjamin Reber, and Wesley Smith. 2022. CARL: Compiler Assigned Reference Leasing. *ACM Transactions on Architecture and Code Optimization* 19, 1 (2022), 15:1–15:28.
[8] Marcus Figorito, Vincent Michelini, Ben Reber Alexander H. Kneipp, , Matthew Gould, Chen Ding, Linlin Chen, and Dorin Patru. 2024. Implementation of a Two-Level Programmable Cache Emulation and Test System. In *Proceedings of the International Symposium on Memory Systems (MEMSYS).*
[9] Xiaoming Gu, Tongxin Bai, Yaoqing Gao, Chengliang Zhang, Roch Archambault, and Chen Ding. 2008. P-OPT: Program-Directed Optimal Cache Management. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing.* 217–231.
[10] Xiaoming Gu and Chen Ding. 2011. On the theory and potential of LRU-MRU collaborative cache management. In *Proceedings of the International Symposium on Memory Management.* 43–54.
[11] Xiaoming Gu and Chen Ding. 2012. A generalized theory of collaborative caching. In *Proceedings of the International Symposium on Memory Management.* 109–120.
[12] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *Proceedings of the International Symposium on Computer Architecture.* 78–89. https://doi.org/10.1109/ISCA.2016.17
[13] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM System Journal* 9, 2 (1970), 78–117.
[14] Louis-Noël Pouchet. [n. d.]. PolyBench/C 4.0. http://polybench.sourceforge.net.

[15] Ian Prechtl, Chen Ding, and Dorin Patru. 2020. Design and Evaluation of a Fixed-size Programmable Working-set Cache on FPGAs. preprint online at https://dx.doi.org/10.13140/RG.2.2.24423.60320.

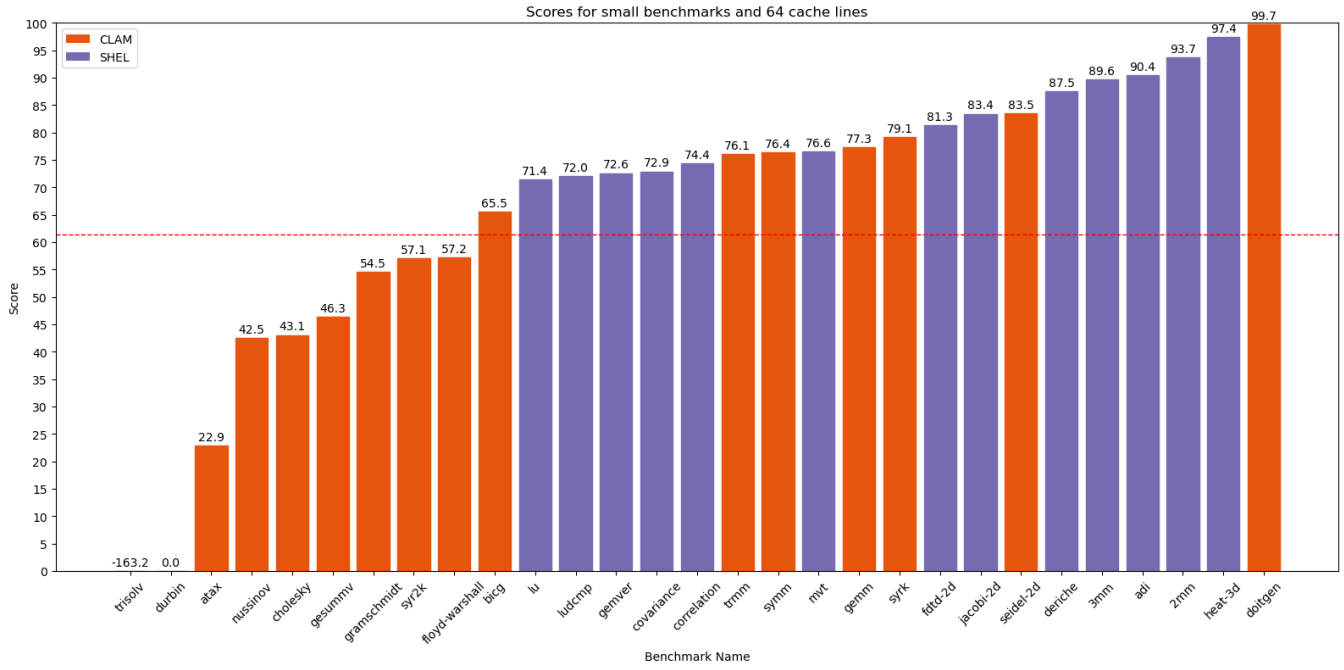
[16] Ian Prechtl, Ben Reber, Chen Ding, Dorin Patru, and Dong Chen. 2020. CLAM: Compiler Lease of Cache Memory. In *MEMSYS 2020: The International Symposium on Memory Systems, Washington, DC, USA, September, 2020*. ACM, 281–296.

[17] Benjamin Reber, Matthew Gould, Alexander H. Kneipp, Fangzhou Liu, Ian Prechtl, Chen Ding, Linlin Chen, and Dorin Patru. 2023. Cache Programming for Scientific Loops Using Leases. *ACM Transactions on Architecture and Code Optimization* 20, 3, Article 39 (jul 2023), 25 pages.

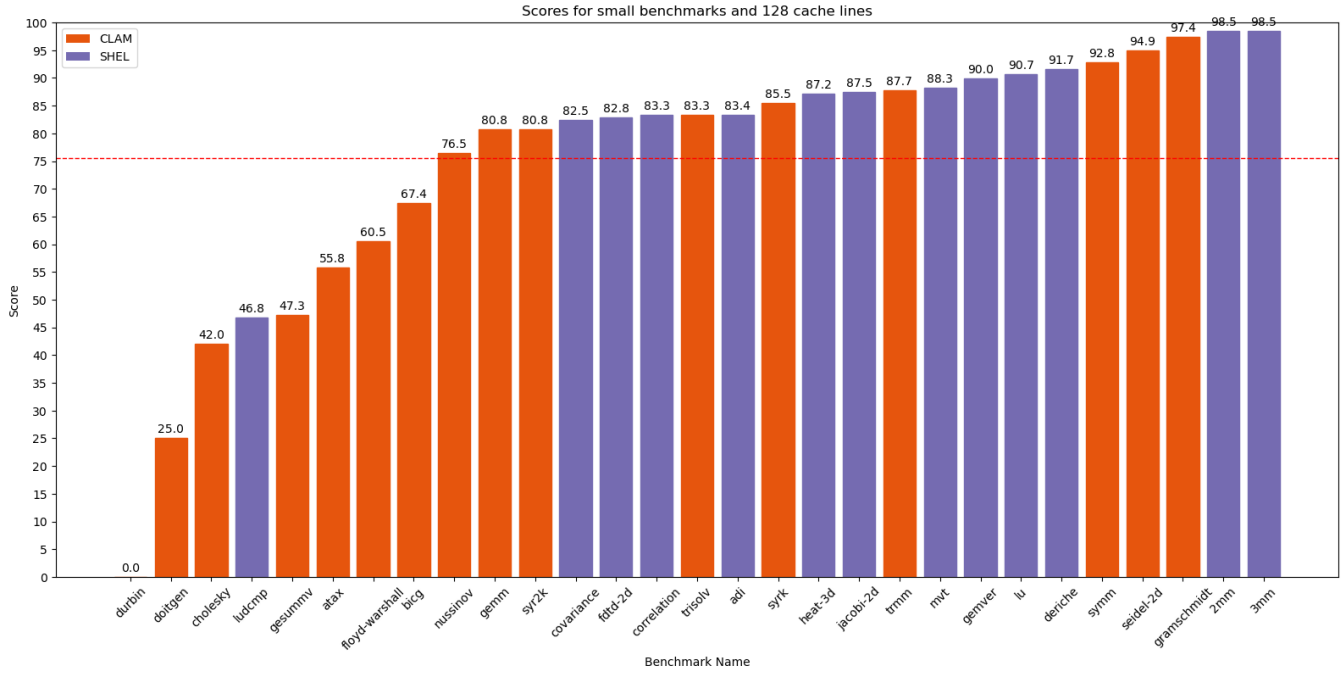
[18] Z. Wang, K. S. McKinley, A. L.Rosenberg, and C. C. Weems. 2002. Using the compiler to improve cache replacement decisions. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. Charlottesville, Virginia.

[19] Andrew Waterman and Krste Asanović. 2024. *The RISC-V Instruction Set Manual: Volume II: Privileged Architecture*. Version 2.2.

A SCORES OF INDIVIDUAL PROGRAMS

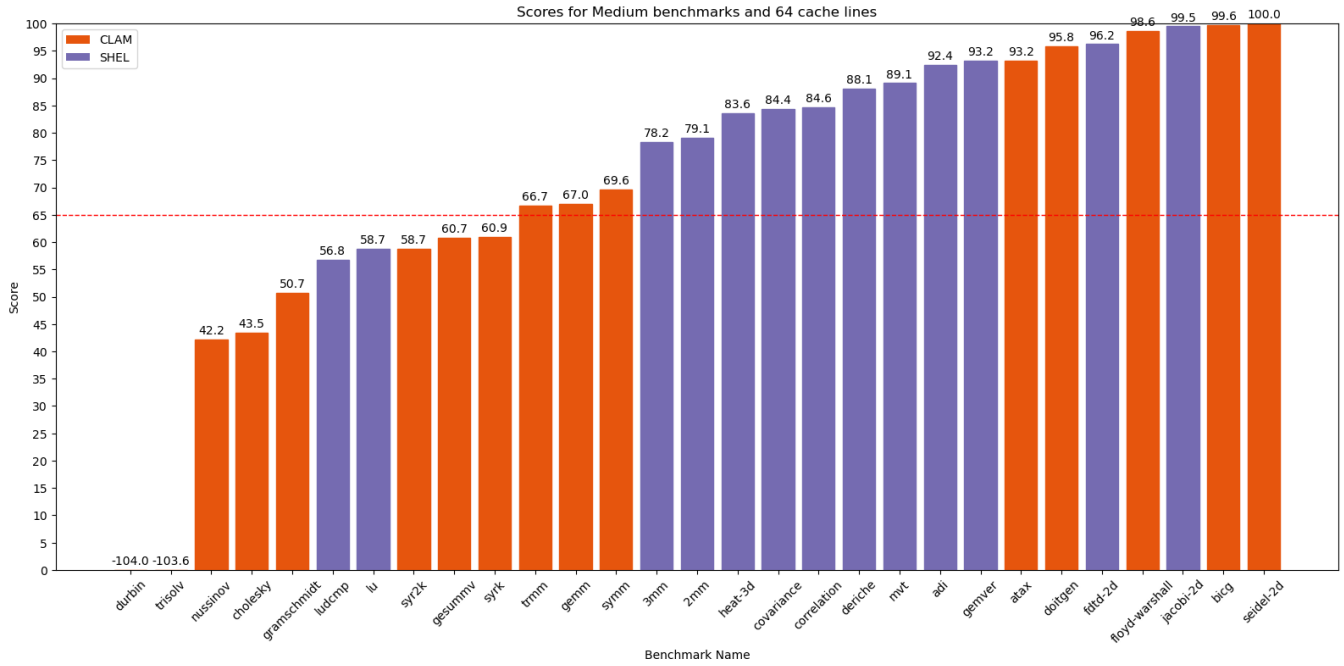


(a) Small input, 64-block cache

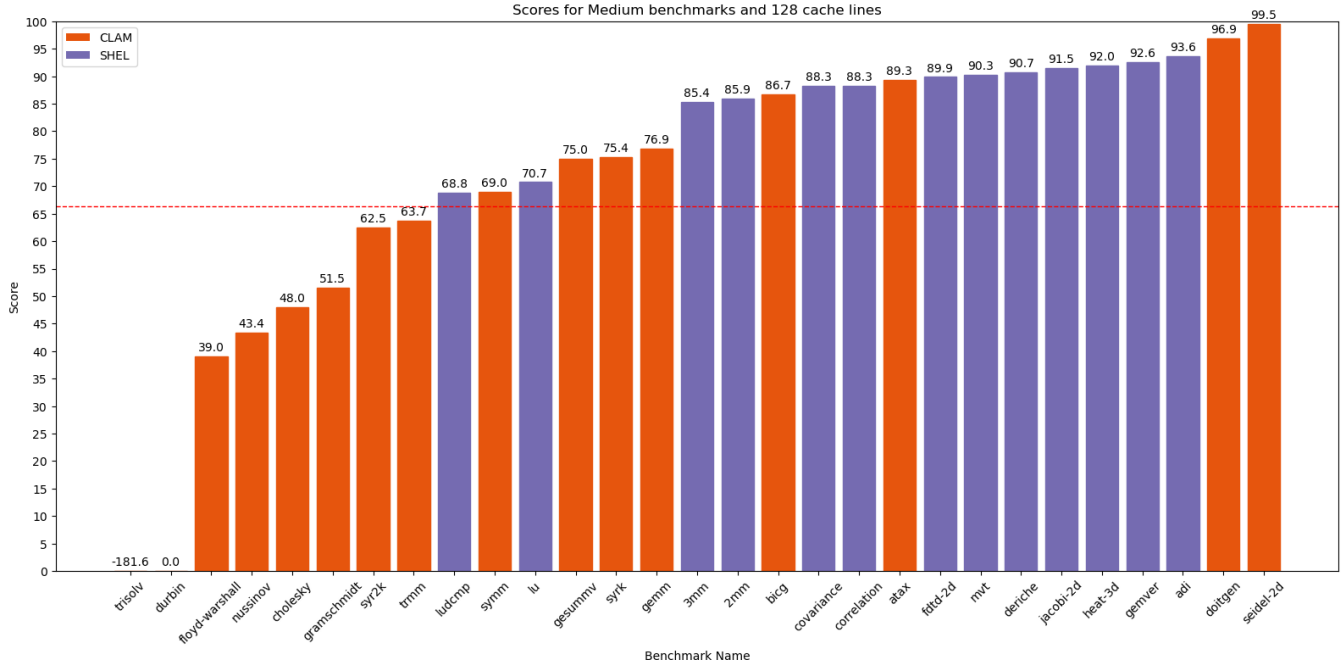


(b) Small input, 128-block cache

Figure 7: Score comparison for small sized benchmarks across both cache sizes, where the red line is the arithmetic mean of the scores in the figure. Scores are sorted in ascending order.



(a) Medium input, 64-block cache



(b) Medium input, 128-block cache

Figure 8: Score comparison for medium sized benchmarks across both cache sizes, where the red line is the arithmetic mean of the scores in the figure. Scores are sorted in ascending order.

B MISS RATIOS OF INDIVIDUAL PROGRAMS

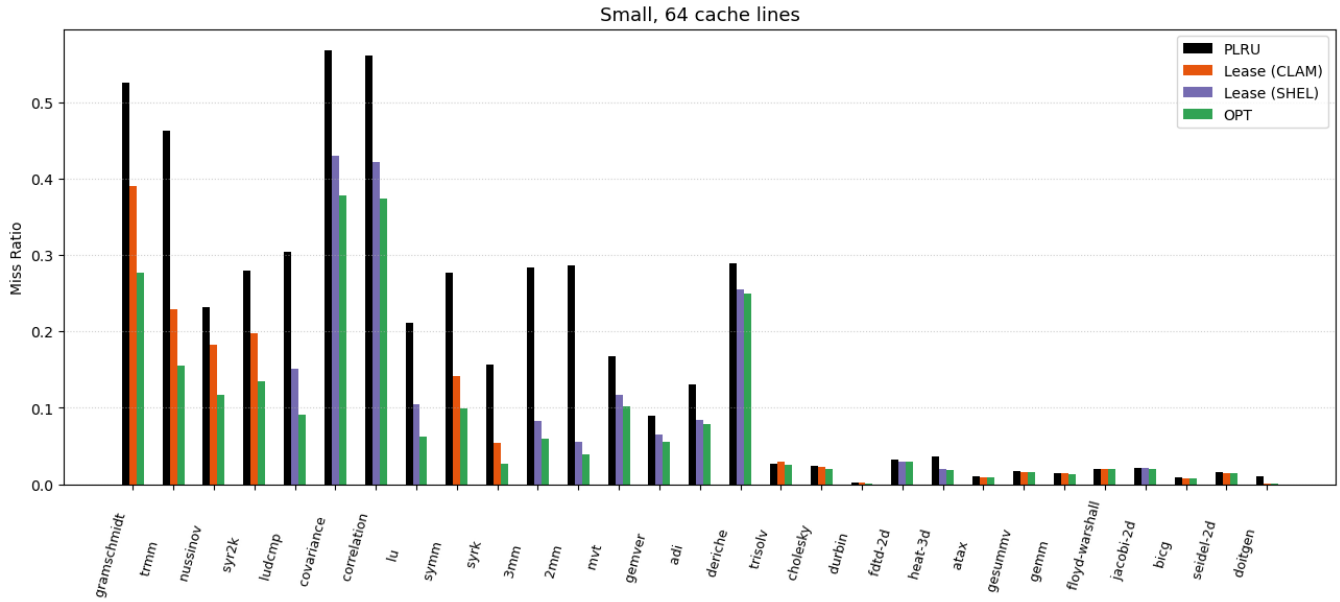


Figure 9: Miss ratio comparison across all PolyBench benchmarks using a small input size and a 64-line cache. Each benchmark is evaluated under four policies: PLRU, Lease (single scope, CLAM), Lease (multi scope, SHEL), and OPT. Benchmarks are sorted by Lease–OPT gap in descending order.

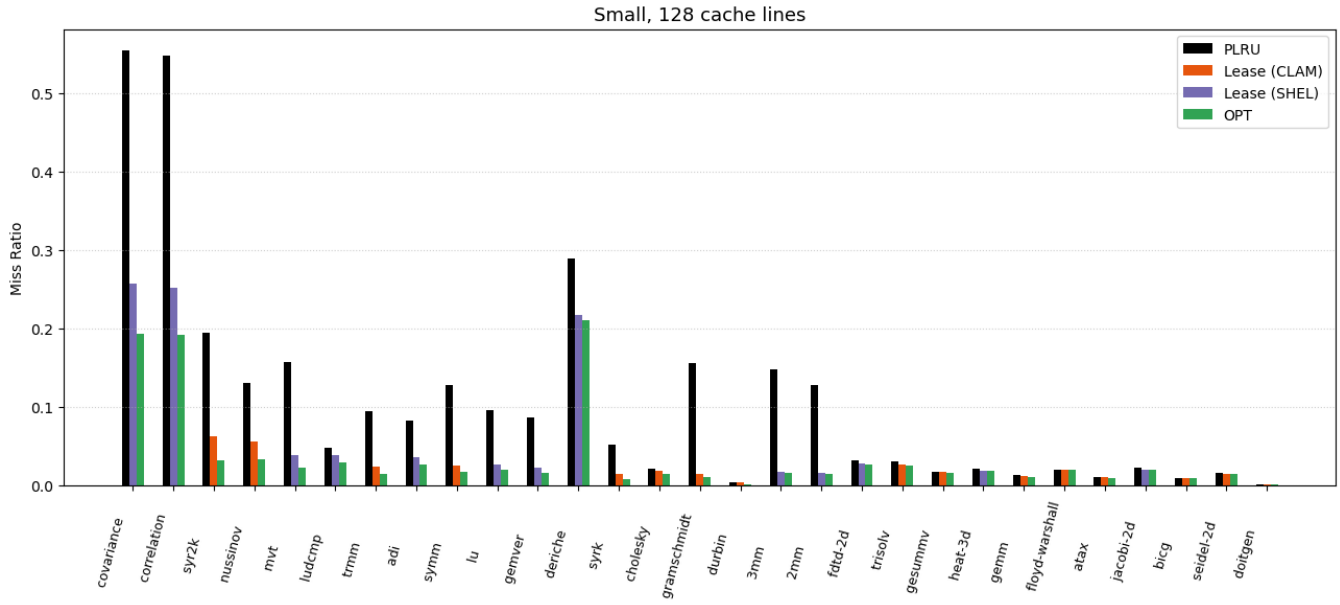


Figure 10: Miss ratio comparison across all PolyBench benchmarks using a small input size and a 128-line cache. Each benchmark is evaluated under four policies: PLRU, Lease (single scope, CLAM), Lease (multi scope, SHEL), and OPT. Benchmarks are sorted by Lease–OPT gap in descending order.

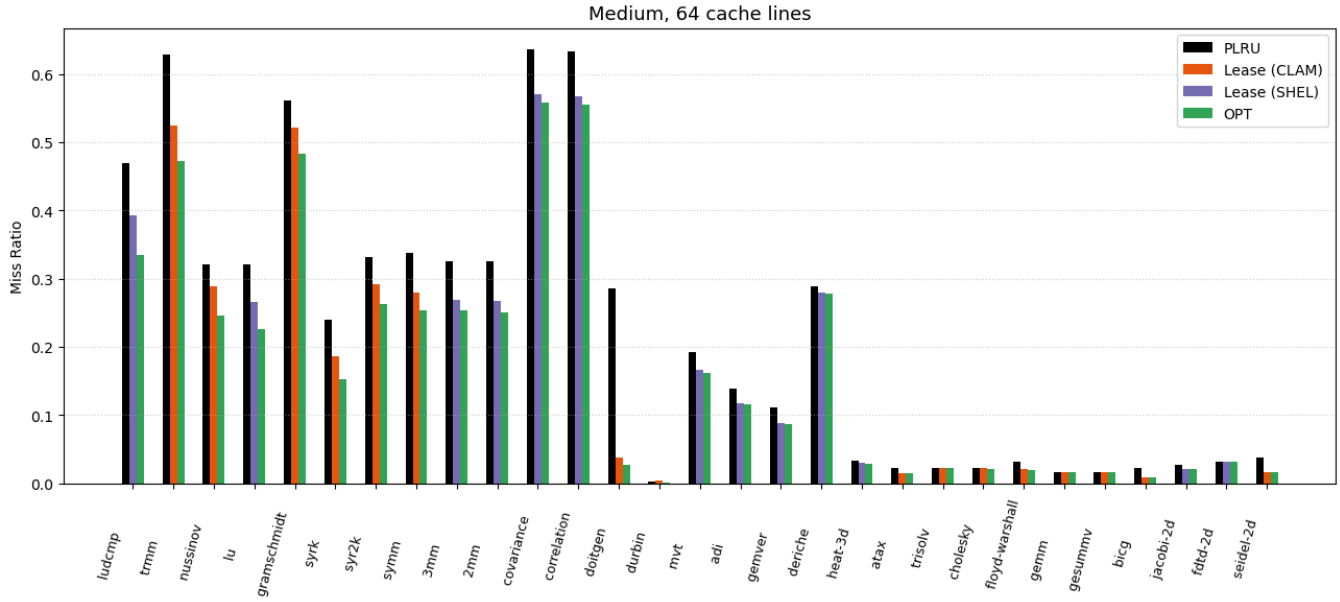


Figure 11: Miss ratio comparison across all PolyBench benchmarks using a medium input size and a 64-line cache. Each benchmark is evaluated under four policies: PLRU, Lease (single scope, CLAM), Lease (multi scope, SHEL), and OPT. Benchmarks are sorted by Lease–OPT gap in descending order.

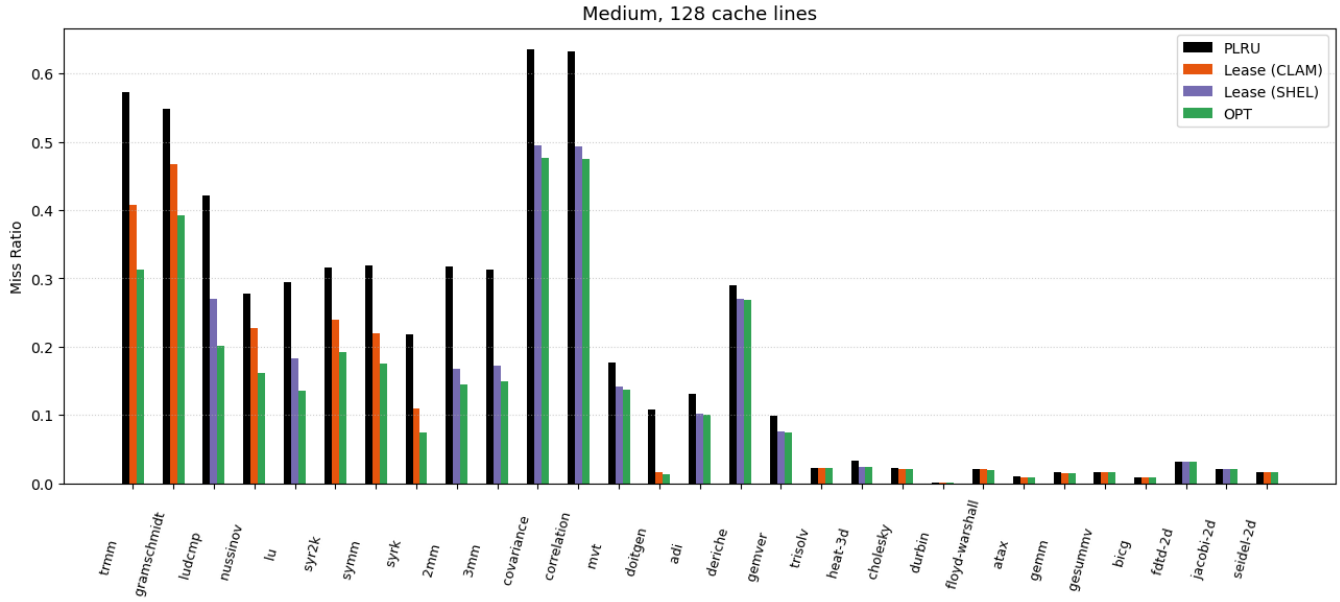


Figure 12: Miss ratio comparison across all PolyBench benchmarks using a medium input size and a 128-line cache. Each benchmark is evaluated under four policies: PLRU, Lease (single scope, CLAM), Lease (multi scope, SHEL), and OPT. Benchmarks are sorted by Lease–OPT gap in descending order.