Implementation of a Two-Level Programmable Cache Emulation and Test System

Marcus Figorito
Rochester Institute of Technology
United States
mpf7573@rit.edu

Vincent Michelini
Rochester Institute of Technology
United States
vfm6849@rit.edu

Benjamin Reber
University of Rochester
United States
breber@cs.rochester.edu

Alexander Kneipp
Rochester Institute of Technology
United States
ahk8565@rit.edu

Matthew Gould
Rochester Institute of Technology
United States
mdg2838@rit.edu

Chen Ding
University of Rochester
United States
cding@cs.rochester.edu

Linlin Chen
Rochester Institute of Technology
United States
lxcsma@rit.edu

Dorin Patru
Rochester Institute of Technology
United States
dxpeee@rit.edu

Abstract

The processor-memory bottleneck is a well-documented problem in HPC. The authors have set out to create a programmable cache. Traditional caches are hardware-controlled and use automatic (built-in) replacement policies, such as Least-Recently-Used (LRU) or Pseudo-Least-Recently-Used (PLRU). A programmable cache uses input from software, more specifically from the compiler, about when to evict a block of data. Previous work has described several eviction algorithms/policies, which have been verified through simulation using memory traces and tested in a single-core with single-level programmable cache. In this work, we discuss in detail the design and architecture of the current emulation and test system, which instantiates a single-core RISCV and a two-level programmable cache in a Field-Programmable Gate Array (FPGA).

When using a set of scientific loops, the two-level lease cache system reduces the main memory access by 50% to 80% on average compared to the single-level lease cache in the prior work. Compared to a two-level cache system using PLRU, the programmable cache reduces the average miss count by 20% to 40%.

CCS Concepts

• **Software and its engineering** → **Compilers**; • **Hardware** → **Dynamic memory**.

ACM Reference Format:
Marcus Figorito, Vincent Michelini, Benjamin Reber, Alexander Kneipp, Matthew Gould, Chen Ding, Linlin Chen, and Dorin Patru. 2024. Implementation of a Two-Level Programmable Cache Emulation and Test System. In *The International Symposium on Memory Systems (MEMSYS '24), September 30–October 03, 2024, Washington, DC, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3695794.3695821

MEMSYS '24, September 30–October 03, 2024, Washington, DC, USA

ACM ISBN 979-8-4007-1091-9/24/09
https://doi.org/10.1145/3695794.3695821

1 Introduction

Optimal cache management requires eviction of the cache-block which will be used the furthest in the future. Traditional cache management methods follow simple heuristics, which seek to capture this behavior - the commonly used LRU or PLRU policies evict the least recently used cache-block under the assumption that stale data is less likely to be reused than new data. However, heuristic-based solutions may not perform well in all cases.

Prechtl et al. [7] introduced the *lease-based, programmable cache*. Rather than making eviction decisions when the cache is full, a lease-based, programmable cache prescribes a cache-block's lifetime with a lease. The block's lease is updated when it is stored or refreshed by a memory access. Leases are assigned at compile time for each load/store instruction, which we call *memory references*. A lease-based, programmable cache is said to be *programmable*, because it uses program information and so its behavior can be different from program to program.

The question which arises is how to assign leases to memory references. The key metric is the *reuse interval*, which denotes the number of memory accesses between a data's use and reuse. For a certain class of programs, which we call *compile-time enumerable*, reuse intervals can be determined statically. The reuse interval histogram (RIH) of each reference is used to assign a lease to each reference, which is used at run-time by the cache to predict the next reuse [7]. There are four lease-assignment policies, which use reuse interval histograms to produce lease assignments: CLAM - Compiler Lease of Cache Memory, which assigns a lease for each reference across the entire program[7], PRL - Phased Reference Leasing, which breaks up the program into fixed-width segments[7], SHEL - Scope-Hooked Eviction Leasing, and C-SHEL - Cross-Scope Hooked Eviction Leasing, which break up the program by its loop structure [8].

The host processor uses the RISC-V architecture. Its openness, reconfigurability, and the ecosystem of associated hardware and software tools enable research at the hardware-software interface using actual hardware emulation and test systems. Furthermore, from a practical point of view, it offers a cost-effective and reliable

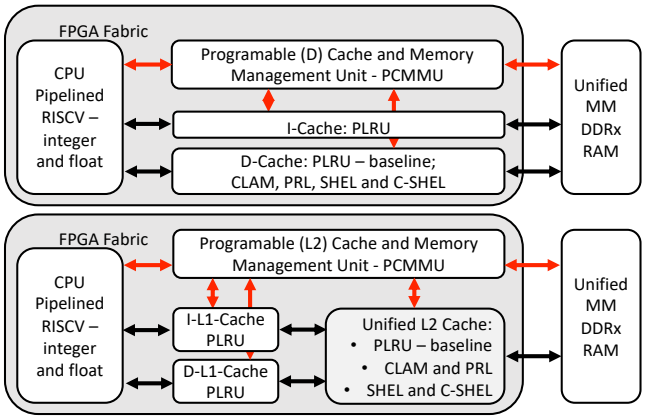


Figure 1: Generic Hardware Systems with Programmable Cache. Lease policies acronyms are explained in text. Black arrows represent data and red arrows control and status signals.

processor architecture, around which one can develop SoC based hardware accelerators.

The goal of this work as far as cache design is concerned is minimal miss count for the same program using the same amount of cache space. The study does not consider prefetching, which reduces the running time by reading from memory early, but does not remove the need for reading from memory. Prefetching hides the memory access latency, but does not reduce it. Locality is a sub-problem of memory performance. It is only concerned with reusing the data in the cache. We address the locality problem in this study. The system does not implement prefetching, and each memory access is a cache miss. By minimizing the miss count, the cache design minimizes the amount of data movement.

The remainder of this paper is organized as follows: Section 2 covers the design and operation of the hardware emulation and test system. Section 3 describes in detail its current use case: to emulate and test the operation of a single RISC-V core and associated two-level, lease-based, programmable cache. Section 4 describes the various lease policies and the lease generation process. Finally, Section 5 presents and discusses test results.

2 Design & Operation

Next we give the first full description of the emulation and test system. Two components, the hardware sampler (Section 2.4) and lease-based, programmable cache architecture (Section 2.5), have been described previously [8]. We include them in this section for completeness.

2.1 Overview

The systems able to be emulated and tested are comprised of a single RISC-V core and associated single- or two-level hardware reconfigurable and software programmable cache. All of these are instantiated in an FPGA, while the main memory is implemented externally using DDR3 or DDR4 memory, as shown in Figure 1.

As shown in Figure 2, the host PC runs an emulation and test control program. This sends down the binary machine code (program, benchmark) to be executed, the associated lease values for the selected lease policy, and any additional initialization and/or constant data. At the end of each program or benchmark run, it receives back performance metrics, such as miss ratio, total number of misses, and clock cycle count. The binary of the program or benchmark to be executed is initialized in the external DDRx main memory, while the lease values are uploaded in designated lease look-up tables (LLUTs). At run-time, the Programmable Cache and Memory Management Unit (PCMMU) controls the entire instruction and data flow/ movement between CPU and the entire memory hierarchy. The latter is comprised of: the L1 Instruction and Data Caches, the Unified and Shared L2 Cache, and the external DDRx Main Memory.

Our system uses a RISC-V 32-bit core with the M and F standard extensions [4]. We are also currently implementing a subset of the *Zicsr* standard extension, primarily to extend our system to a multi-core (dual-core) system. The pipelined core has 6 stages: Instruction Fetch, Instruction Decode, Operand Fetch, Execute, Memory Operation, and Writeback. It further supports a 2-bit branch prediction scheme with a 64-entry lookup table. The system is described in Verilog and implemented on an Intel Cyclone V FPGA. Specifically, the device is 5CGTFD9E5F35C7 and the resource utilization for the two configurations is captured in Table1.

While we have chosen to use the RISC-V architecture because of its openness and widespread use within the research community, a potential user of our emulation and test system could instantiate any other CPU core architecture. The interface between the CPU and L1 caches and PCMMU is quite generic.

Motivation and Prior Work The prior work presented a single-core processor with a single-level cache Prechtl et al. [7] and Reber et al. [8]. The size of the cache is limited in the prior work. This has motivated us to create a two-level cache system, in particular, a level-two lease cache with four times the size of the prior design. We will show in Section 5.2 that the amount of main memory access in the larger lease cache is two- to four-times smaller.

2.2 Reconfigurable Cache

We have developed and tested multiple existing algorithms for assigning leases to memory references, which we call *lease assignment policies*. Much of the difference between policies is opaque from a hardware point of view, and simply results in different values being filled into the lease lookup table (LLUT). However, some lease assignment policies use dynamic scoping rules, which require additional hardware support relative to the baseline lease-based, programmable cache implementation.

To serve as a test platform for the various lease cache policies, our system implements hardware reconfigurable and software programmable caches. The various hardware configurations are selected using swappable Verilog *define* statements. The lease values are initialized and/or uploaded in lease lookup tables (LLUTs). The programmable cache supports all previous algorithms and policies presented in [7, 8].

2.3 Inclusivity

In a *fully inclusive* cache system the caches closer to the CPU, L1 in our case, contain blocks of data which are also stored in caches farther from the CPU, L2 in our case. In a *fully exclusive* cache

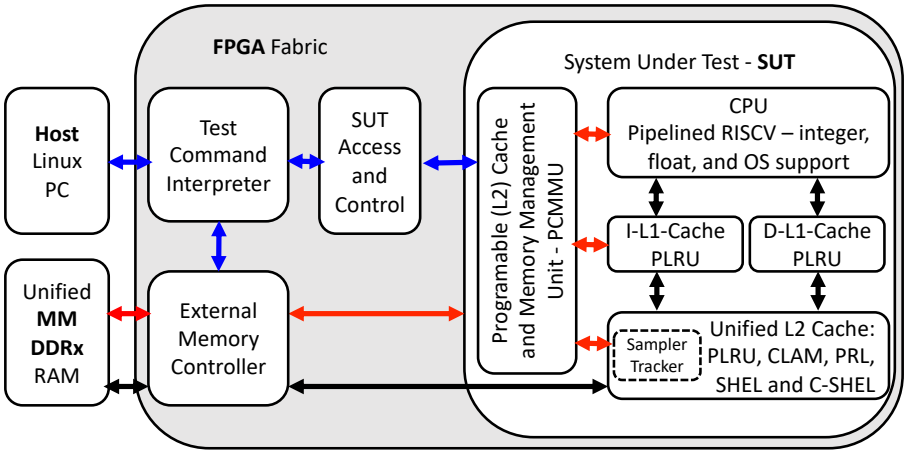


Figure 2: Block diagram of the two-level programmable cache emulation and test system. Black arrows represent SUT data, red arrows SUT control and status signals, and blue arrows host PC control program configuration and test results data.

Table 1: FPGA Resource Utilization

Resource	Single-Core, Single-Level Cache Project Utilization on Cyclone-V GT	Single-Core, Two-Level Cache Project Utilization on Cyclone-V GT
ALM Count	28,711 / 113,560 (25 %)	50,082 / 113,560 (44 %)
Register Count	34,370	58,060
Pin Count	127 / 616 (21 %)	127 / 616 (21 %)
Block Memory Bit Count	425,056 / 12,492,800 (3 %)	687,200 / 12,492,800 (6 %)
DSP Block Count	10 / 342 (3 %)	10 / 342 (3 %)
PLL Count	2 / 20 (10 %)	2 / 20 (10 %)

system, L1 caches contain only blocks not included in the L2 cache, and vice-versa. *Partially inclusive* L1 caches are allowed to duplicate blocks stored in the L2 cache, but are not required to.

In the context of the lease-based, programmable cache, each inclusivity policy has different implications. The *partially inclusive* cache presents a "shadowing" problem when blocks are written back from L1 to L2. Since the evicted block being written-back is not guaranteed to be in L2, it may need to allocate new space for the written-back block. Since the written-back block is not the one corresponding to the current CPU instruction, the reference for the block is unavailable, and the correct lease cannot be assigned without additional hardware support in L1, to store the reference of the last access to each block. A *fully inclusive* cache resolves this issue, since a write-back will always hit in the L2 cache, and no new lease will need to be assigned.

A *fully exclusive* cache presents an interesting alternative. It shares the shadowing problem with the partially inclusive cache (in fact, all accesses to L2 will be a miss), so L1 must still store the reference of the last access for each block. However, an exclusive cache carries additional benefits: higher total cache capacity (due to no data duplication), and other potentially beneficial interactions between a traditional L1 cache and a L2 lease cache.

We call our current two-level cache implementation simply *inclusive*, i.e., neither fully nor partially inclusive. To mitigate the shadowing issue discussed above, a fully inclusive cache is presently work-in-progress. Our inclusive implementation uses a "protection" method, where cache blocks currently in L1 are prevented from being evicted from L2. In order to achieve this, L2 tracks which blocks are requested and used by L1, and L1 reports the address of each evicted block. L2 then tracks which of its blocks are protected using a bit-field, where each bit corresponds to a block in the cache. Each protected block is protected from eviction utilizing a strategy specific to each supported replacement policy. We are also considering implementing an exclusive cache, with the L2 cache serving as a victim cache for items evicted from the L1 cache.

2.4 Hardware Program Sampler

Leases are generated based on reuse-interval (RI) histograms for each reference in an arbitrary program. For a certain class of programs we designate as *compile-time enumerable*, RI histograms can be computed at compile time [2]. However, as compile-time enumerable programs represent a limited subset of all possible programs, we use a more general hardware sampling mechanism to collect RI statistics. These are extracted by custom hardware that samples the access stream of the program at the input of the cache-level that uses the lease caching policy.

The hardware sampler "snoops" the communication bus between the lease cache and the source of its accesses [8]. The sampler contains a Linear-Feedback Shift-Register (LFSR), which bit-length and seed is determined at run-time. The LFSR is used to populate a down-counter, which decrements at every memory access. When the counter reaches 0, a new sample acquisition is started and a new value is loaded into the counter. Once a sample is started, the

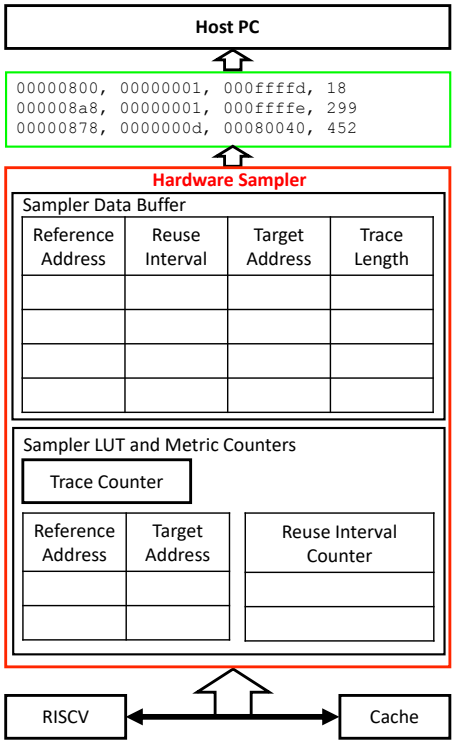


Figure 3: Structure of the hardware sampler (reproduced in part from Reber et al. [8])

reference address of the access is stored in a lookup table (LUT). In-turn, the table contains a counter for each active sample, i.e., already started, which counts up on every memory access. When a reuse for a block in any active sample is detected, the reference address, reuse interval, block address, and the logical time of the reuse is stored in a sample buffer. Finally, when the sample buffer is full, the samples are transferred to the host PC under the control of the emulation and test control program, (further discussed in Section 2.8). Figure 3 shows the structure of the hardware sampler. If a new sample acquisition is started when the lookup table of active samples is full, then the active sample with the largest reuse interval counter is stored in the sample buffer with the two's complement of its current reuse interval to indicate that no reuses for that access were detected.

For the purposes of this work, a profiling pass is used to collect RI statistics prior to execution. A dynamic lease assignment framework, which collects statistics and adjusts or modifies leases at runtime, is the target of future work.

2.5 Lease Cache Architecture

The lease cache hardware is able to support the application of lease policies from program compilation through its execution. Its architecture is shown in Figure 4 (significantly modified and updated from Reber et al. [8]).

Lease Assignment. The hardware that implements a lease policy complements an existing cache memory infrastructure through the addition of a lease policy controller (Figure 4). In support of the latter, the request bus to the cache is augmented with the address of the reference invoking the access. Both target and reference addressess propagate through lookup tables and provide concurrently cache location and lease policy information to the controller. A combination of four 128 entry lookup tables comprise the Lease Lookup Tables (LLUTs) and resolve the following signals:

(1) Lease Valid [1 bit] - flag indicating a lookup table hit.
(2) Primary Lease [n bit] - lease associated with the higher probability assignment.

There is a single secondary lease associated with a single reference per program phase. Its assignment is based on an associated probability value, specifically the probability that this lease will not be assigned. These are provided in the header that pre-appends each phase and are stored in software accessible registers.

The primary and secondary leases are multiplexed by probability evaluation. An LFSR generates a random number that is compared against the probability value output by the lease probability lookup table. If the random value is greater than the one output by the LLUT, the secondary lease is passed through, else the primary lease is passed through. A second multiplexer makes the final selection. If the access results in an LLUT hit, the current lease assignment is validated and passed. Else if the reference is not found in the table, a default lease assignment, stored in a software accessible register, is instead passed through. References without an associated lease assignment are assumed to have no near future re-reference and provide little benefit to cache performance regardless of cache utilization. We elect to assign a default lease of one to these references, so that after their immediate use these are eligible for eviction.

Line Vacancy. Each cache line (or block location) has an associated lease register with two control ports and a multi-bit output bus. The output bus of each register drives a NOR reduction operator, essentially a comparator with zero, which produces an expired bit per lease register. A priority encoder examines all expired bits and identifies the first occurrence (lowest address) of an expired lease. A pointer to this address is produced and transferred to the controller to be used in case an eviction is necessary. The pointer is validated by a reduction OR (inequality with zero comparison), of all expired bits. If at the time an eviction is necessary and the pointer is invalid (no lease has expired) the replacement follows the *auxiliary policy i.e., random replacement.*

The auxiliary replacement policy is also employed if there are a large number of default lease (of one) assignments in a row. This is done to mitigate the possibility that the assumption that references without an associated lease assignment have no near future re-reference is not valid. If this is the case, the lease cache would perform poorly, because it would evict the highest expired line in the set and completely ignore any type of data locality. Hence the lease cache is designed such that if there are more than x (for this work 1024 was chosen) default leases assigned consecutively, the lease cache will exclusively use the auxiliary policy until an LLUT hit occurs, whereupon normal operation resumes.

Application of a Lease Policy. The flowchart in Figure 5 illustrates the application of a lease policy. At every cache access, all non-expired lease registers are decremented. If the access resolves as a cache hit (not a lease lookup table hit) the lease register at the translated address is load enabled, regardless of lease assignment. If the access is a miss, then the item is cached in the location generated

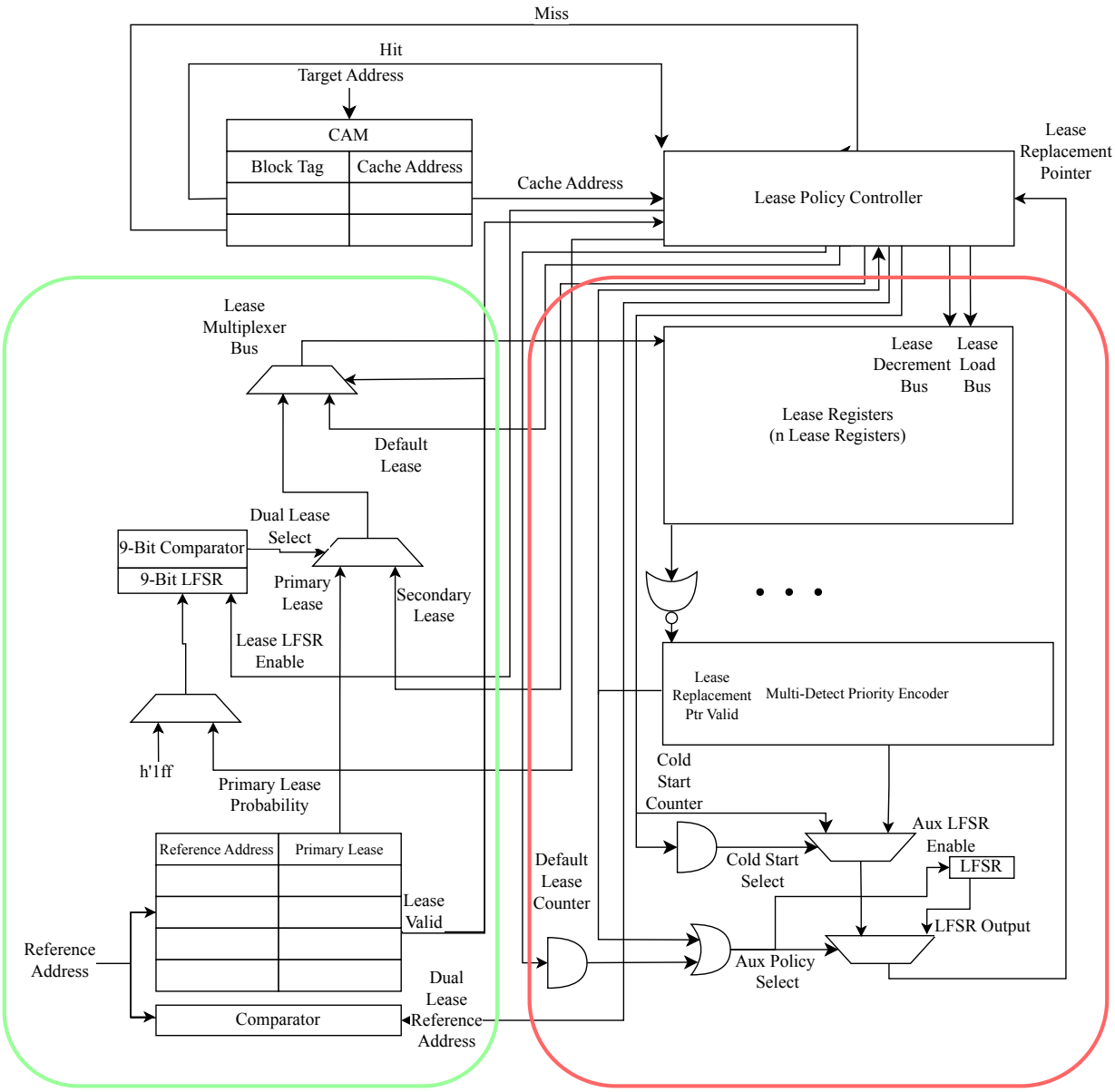


Figure 4: Lease cache hardware architecture for a cache of *n* blocks and lease register size of *m* bits.The components in the green box are the lease look-up circuitry. The components in the red box are the replacement logic and lease update circuitry. Significantly modified and updated from Reber et al. [8].

by the relevant policy (either lease or the auxiliary policy) and then assigned a lease value as described above.

Hardware Support for Scoped Lease Policies. is illustrated in Fig. 6. After reset, the Lease Lookup Table (LLUT) is populated with the leases of the first phase and the lease cache configuration information: secondary lease value, secondary lease probability, the number of references in the phase, default lease value, and the address of the reference assigned the secondary lease. During benchmark kernel execution, if a phase marker for a phase different than the current one is encountered in the software, the CPU adjusts the value of the current phase register to that of the marker. The lease cache detects this change and sets a flag that stalls the CPU. The lease cache then requests the leases for this new phase from main memory, writes them to the LLUT, and additionally updates the lease cache configuration information with the values from the new phase header. The lease cache then clears the flag it sets to take the CPU out of stall and to resume benchmark execution. This process repeats at every phase marker denoting a new phase until the benchmark kernel execution finishes.

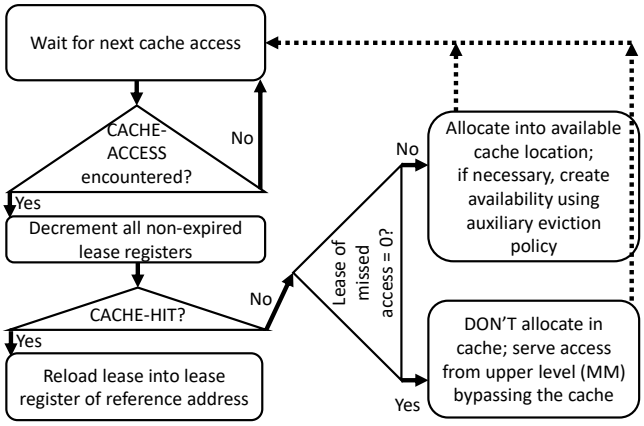


Figure 5: Lease cache operation flowchart - Fully described in the text.(reproduced from Reber et al. [8]).

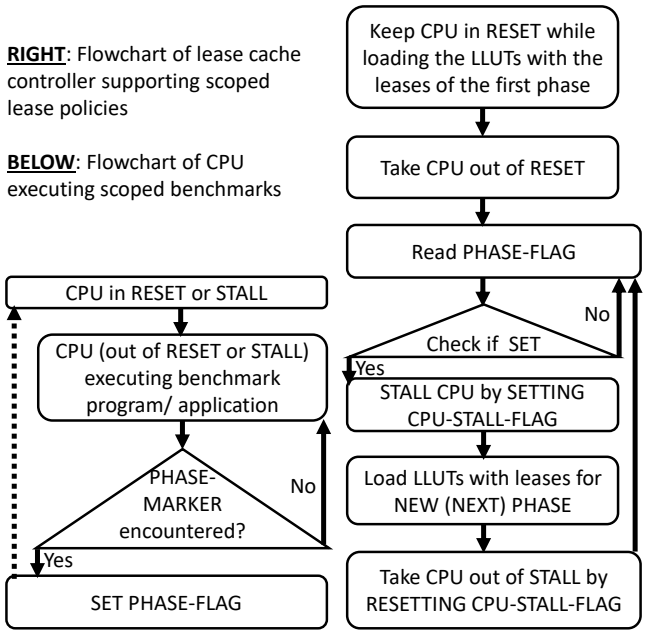


Figure 6: Hardware Support for Scoped Lease Policies: SHEL and C-SHEL(reproduced from Reber et al. [8]).

2.6 Hardware Cache Tracker

To determine the effect of the different lease generation policies on cache occupancy, a hardware cache tracker is implemented to visualize the utilization of the lease cache at each point in program execution. The tracker "snoops" on the lease registers of each cache line (block location) and separates them into categories of "long lease" (bits 31-16), "medium lease" (bits 15-8), and "short lease" (bits 7-0). If a lease is in none of these categories, then the lease is expired. The tracker collects these values for each cache line at fixed intervals, and stores them into a buffer. Once the buffer is full, the processor is stalled and the content of the buffer is transmitted to the host PC. Cache occupancy spectra are shown and discussed in Section 5.

2.7 External Memory Controller

The external memory controller presented in Fig. 2 provides the means by which the programmable cache and memory management unit (PCMMU) and the emulation and test control program running on the host PC, further discussed in Section 2.8, transfer program instructions and data to and from the external DDR3 or DDR4 main memory. The emulation and test control program can request data of any size, i.e., from a single word of data to multiple blocks of data to read or write, while the PCMMU will always request a block of data at a time, and that when a cache miss occurs. It should be noted that since a proxy memory request does not go through the internal processing system, the cache will be unaware of any writes that occur to the main memory in this way and the data there may become invalid.

2.8 Emulation and Test Control Program

The emulation and test control program is written in C++ and runs on the host PC, Figure 2. Before a test run it loads: the benchmark program machine code, test configuration information, and lease values. After a test run it collects: cache performance statistics (misses and miss rates), tracker information, and optionally result data stored in main memory.

At present, all communication between the control program (host PC) and the emulation and test system uses a UART-JTAG interface. On the system side this is embedded in a *test command interpreter* module, Figure 2. Depending on the objective of the command, this module gives the control program (host PC) either direct access to the main memory, or to the configuration and test registers of the the System Under Test (SUT) via an intermediate *SUT access and control* module. The latter registers are capturing cache performance statistics after a Polybench benchmark is run, and do not influence at all the operation of the SUT. The SUT access and control module provides a "bridge" between the control program (host PC) and SUT.

3 Use Cases

3.1 SoC Based Accelerator

The system developed and presented in this paper can be used in emulation mode as a system-on-chip (SoC) based accelerator. Toward this end, it instantiates in a FPGA, a RISC-V central processing unit, a reconfigurable and programmable "traditional" (PLRU) or lease-based cache, and registers for I/O peripherals. The main memory is external and optionally implemented in DDR3 or DDR4 SDRAM. Unlike in a traditional computing system, its performance is bolstered by the reconfigurable and programmable cache. Because the use in emulation mode employs a subset of commands and steps relative to the test mode, we will further focus on the latter.

3.2 Test System Workflow

So far, the system has primarily been used as a test system for novel cache memory implementations. This is possible because the system

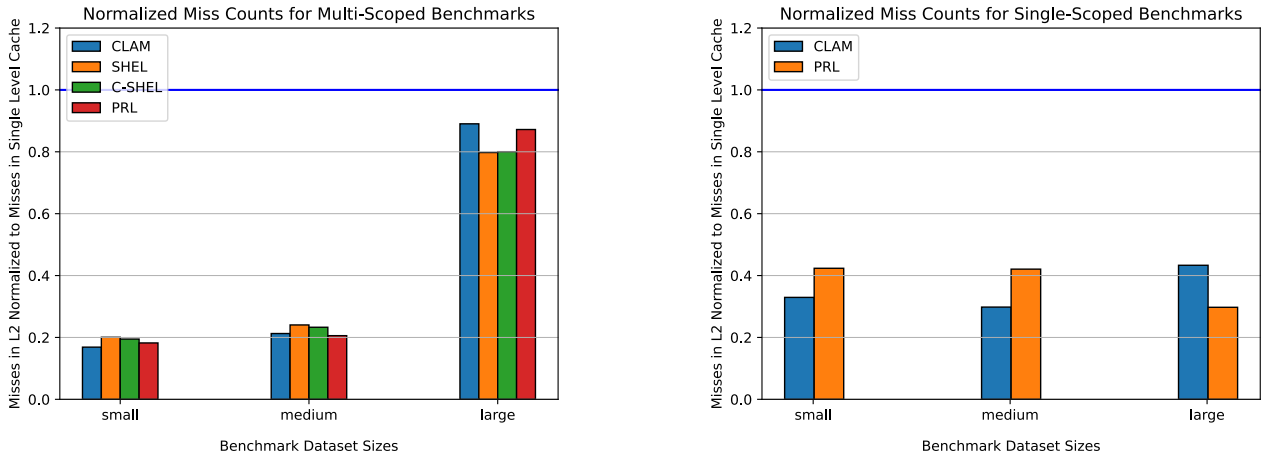


Figure 7: The geomeans of L2 miss counts normalized to the single-level lease cache for multi-scoped (left) and single-scoped (right) benchmarks for all three input sizes and without Pluto optimization.

is hardware reconfigurable and the cache software programmable. The Polybench suite [5], a set of polyhedral benchmarks written in C, have been used to evaluate the performance of different cache memory implementations, and by implication the effectiveness of different lease policies.

In *test mode* a benchmark run starts with the configuration of the FPGA with the SUT and test support hardware. Then, the user control program (host PC) initializes and launches the benchmark run and collects performance data. The latter is captured in a text file, in which each line contains: the number of cache hits, misses, write-backs, cycles, expired evictions, default assignments, default misses, and random evictions for each of the caches in use. These steps assume that benchmarks have been run before in *sampling mode* using PLRU, RIs have been collected and leases have been generated, as described previously in Section2.4.

3.3 Reference Array Checking

The control program (host PC) is able to directly access the main memory of the emulation and test system to retrieve the output results, also called *reference arrays*, generated by the benchmark program that just finished execution. These are then compared with the results generated by the same benchmark program run on the host PC, i.e., good expected output values.

This process is carried out for every benchmark of the Polybench suite individually. This functionality and feature is indispensable to validate the correctness of the performance results. So far, the operation of twenty-eight of the thirty benchmarks has been confirmed in this way. The process of reference array checking also provides an additional tool and means to debug the hardware and software of the emulation and test system.

4 Software Support

We first compile a program into RISC-V binary. Then we profile its execution once on the RISC-V machine. Profiling uses the hardware sampler (Section 2.4) to collect the RIH for each load and store instructions in the binary code. The profile result is then used to assign leases using one of the policies. The simplest policy is Uniform Lease (UL), which assigns a single lease value. This was used in the first lease-cache prototype [6]. Since then, we have been using reference leases, where each load and store instruction may be assigned a different lease.

The binary code is augmented with a lease table. Each row of the table contains the address of a load or store instruction and the lease. The lease may be a single value or a dual lease. A dual lease has two lease values and is used when the short lease itself would not fully use the cache and the long lease would use more than what is available. A dual lease contains three fields: the two leases, and a percentage value (for choosing the short lease).

When a program is run, the lease table is loaded by the processor. At a load or store, the lease is retrieved and given to the data block being accessed.

In an ideal lease cache, the size may vary arbitrarily. We call the number of active data blocks, i.e., those with a positive lease, the *Virtual Cache Size (VCS)*. In this case, leases can be optimally assigned by the CARL (compiler assignment of reference lease) algorithm, which takes a target average VCS and set of reference RI distributions and assigns leases to maximize the number of hits per unit of a lease. CARL is optimal in that no other lease assignment can further reduce the number of cache misses [3]. In a hardware cache, however, the size is fixed. We call it the *Physical Cache Size (PCS)*. It requires a secondary policy to evict a cache block when VCS exceeds PCS. When the cache is full at a miss, we use the auxiliary policy i.e., random replacement.

The following policies were used to program the single-level lease cache [8]. In the new system, the second-level lease cache is programmed in the same way except that the RI histograms come from sampling L1 misses (instead of loads and stores when programming the single-level lease cache). The following describes four policies used in these experiments.

- *CLAM* Compiler Lease of Cache Memory, which uses CARL without change. It lets CARL assign leases for the average VCS equal to PCS and uses the same leases.

- *PRL* Phased Reference Leasing, which divides the execution into equal-length intervals, and uses CARL to assign leases within each interval, setting the target average VCS at each interval to be PCS.
- *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.
- *C-SHEL* Cross-loop SHEL, which considers cross-loop reuses when assigning leases in adjacent scopes (loop nests). SHEL and C-SHEL are applicable in only multiple scope tests.

5 Test Results & Discussions

The test results for a single-core with a single-level cache were presented and discussed in Reber et al. [8]. In this section, we present for the first time test results for a single-core with two-level cache.

5.1 Implementation and Test Setup

We use GCC (RISC-V embedded GCC 8.3.0-2.3.1) to compile a program into RISC-V binary and the hardware sampler (Section 2.4) to generate RI samples for each load and store instruction. Given the sample trace, a lease assignment program is used to assign reference leases which are then added as the lease table to the program binary. When a program is run, the lease table is loaded by the processor. At the first occurrence of a load or store, the lease is retrieved and assigned to the data block being accessed. RI processing and lease assignment are implemented in Rust and support the policies described in Section 4.

We use PolyBench/C 4.2.1, which contains 30 numerical kernels [5]. The benchmark suite is relatively easy to port through the FPGA toolchain to allow testing on a real system. The kernels are extracted from linear algebra, image processing, physics simulation, dynamic programming, and statistics, which are all common workloads in scientific computing. We compile each program with the `GCC -O3` optimization level without vectorization, which our CPU does not currently support. We use small (128 KB), medium (1 MB), and large (25 MB) data set sizes.

Experiments were performed without and with PLUTO optimizations. Version 0.11.4 of the PLUTO optimizing compiler was used to optimize the PolyBench benchmarks [1]. The compiler was invoked using only the `-tile` option, in order to enable cache tiling and polyhedral optimizations. Furthermore, it was configured to produce 16x16x16 tiles in order to effectively fit in our cache size.

The Polybench suite can be divided into two benchmark (program) categories: single- and multi-scope, with 17 and 13 kernels respectively. The latter group is further divided into two subgroups. *Acyclic* programs have one phase per scope, whereas *cyclic* programs have multiple phases per scope. Multi-scope programs are run with the SHEL and C-SHEL policies, where the scopes were manually annotated as described in Reber et al. [8]. Further statistical properties of these programs were also discussed and captured in Table 1 of the same paper, Reber et al. [8].

5.2 Comparison with Single-level Lease Cache

Figure 7 shows the reduction in memory traffic by the two-level lease cache compared with the prior design of a single-level lease cache. The L2 lease cache is four times the size of the single-level lease cache. The two lease caches use the same policy, which are CLAM and PRL for single-scoped benchmarks (right graph) and all four policies for multi-scoped ones (left graph). The figure shows the geomeans of the L2 cache miss count normalized to that of the single-level cache.

Except for multi-scoped tests using the large input size, the L2 lease cache reduces the average miss count by two thirds to over a half for single-scoped tests and around 80% for multi-scoped tests. The reductions are similar across different lease policies. These large improvements show the benefits of moving from a single-level lease cache to the two-level design.

5.3 Two-level Lease Cache Performance

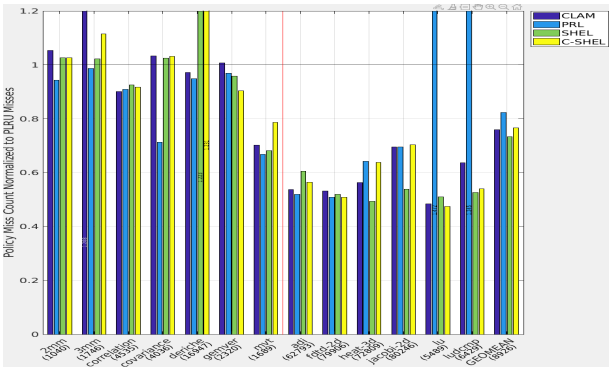
Figure 8 shows misses normalized to PLRU without Pluto optimization, for multiple- (left column) and single-scope (right column) benchmarks, and three data sizes. Without Pluto optimization, the only applied improvement relative to traditional caching is one of the lease policies. As can be inferred from these graphs, the application of lease policies is effective for a majority of benchmarks, i.e., it reduces the miss rate compared to PLRU. As one would expect, the effectiveness diminishes with an increase in the size of the data set, i.e., when this significantly exceeds the capacity of the fixed, real cache. The same test results are shown in Figure 9, which plots the absolute miss ratio values for PLRU and all four lease policies.

Figure 10 shows misses normalized to PLRU with Pluto optimization, for multiple- (left column) and single-scope (right column) benchmarks, and three data sizes. These tests and results combine the powerful Pluto optimizations with the application of lease policies. As can be inferred from these graphs, the application of lease policies is still effective for a majority of benchmarks, i.e., it reduces the miss rate compared to PLRU even after the application of Pluto optimizations. Again, the effectiveness diminishes with an increase in the size of the data set, i.e., when this significantly exceeds the capacity of the fixed, real cache. The same test results are shown in Figure 11, which plots the absolute miss ratio values for PLRU and all four lease policies.

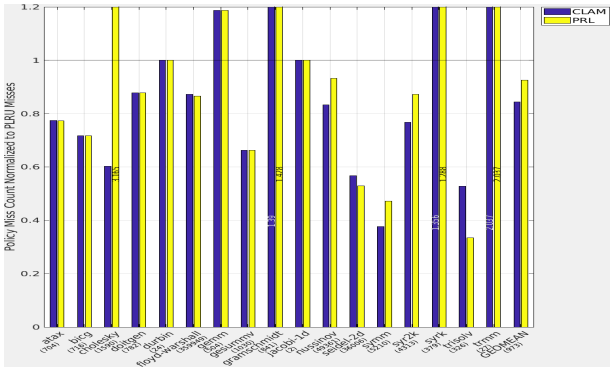
Figure 12 and Figure 13 show an interesting comparison: misses with the application of ONLY the lease policies, but without the application of the Pluto optimization, versus PLRU with Pluto optimization. The latter is what is available to users at the present time in "off-the-shelf" computing systems. While the results are not as good as combining Pluto optimizations and lease policies, as shown previously in Figures 10 and 11, the fact that lease policies alone still lower the miss ratio for almost half the benchmarks is further proof that these are effective.

5.4 Cache Performance Visualization

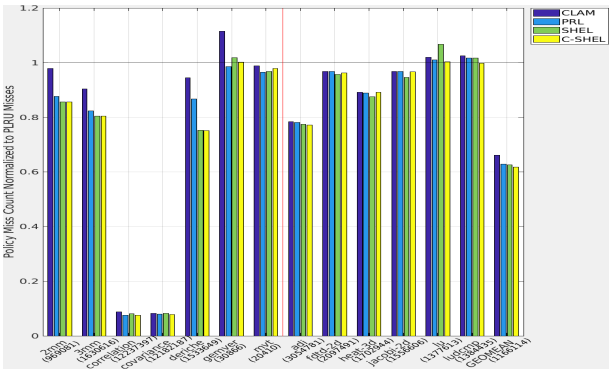
Cache Occupancy Spectra have been introduced by Prechtl et al. [7] to visualize the state of the lease cache. While we have collected data and generated graphs for all benchmarks, for brevity we only show the spectra graphs for four benchmarks. Figure 14 shows the occupancy spectra of two *acyclic* benchmarks: 2mm and 3mm, for the medium data set, and CLAM and SHEL lease policies. Figure 15 shows the occupancy spectra of two *cyclic* benchmarks: adi and



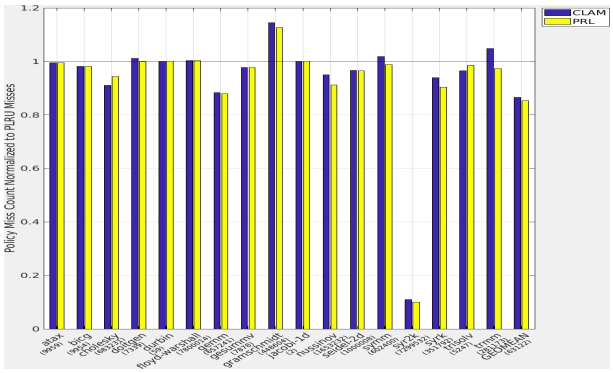
(a) Multiple scope, SMALL_DATASET, Misses normalized to PLRU without Pluto optimization.



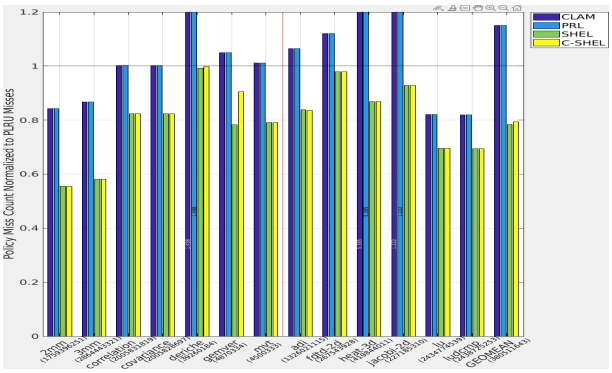
(b) Single scope, SMALL_DATASET, Misses normalized to PLRU without Pluto optimization.



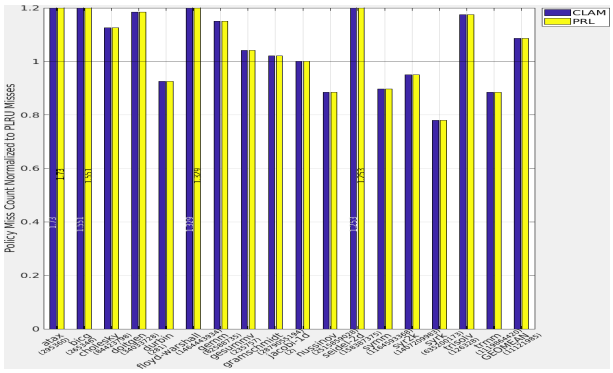
(c) Multiple scope, MEDIUM_DATASET, Misses normalized to PLRU without Pluto optimization.



(d) Single scope, MEDIUM_DATASET, Misses normalized to PLRU without Pluto optimization.



(e) Multiple scope, LARGE_DATASET, Misses normalized to PLRU without Pluto optimization.

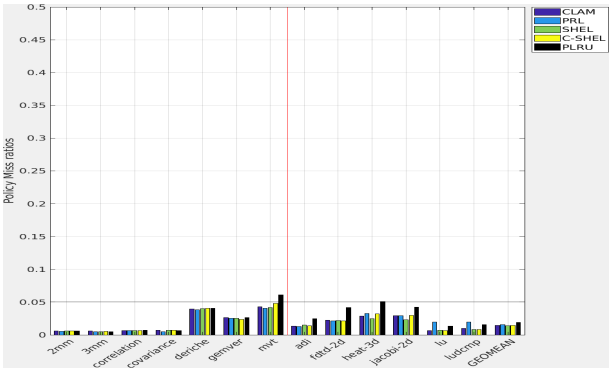


(f) Single scope, LARGE_DATASET, Misses normalized to PLRU without Pluto optimization.

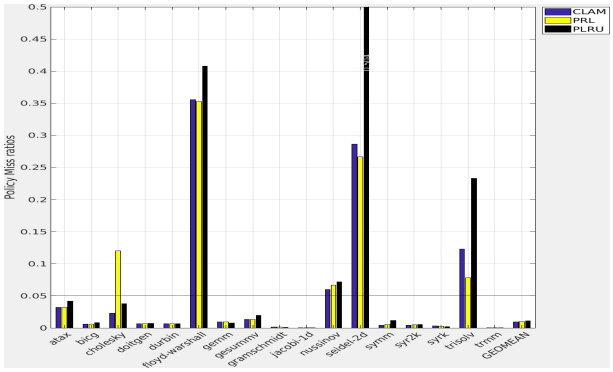
Figure 8: Misses normalized to PLRU without Pluto optimization for multiple- and single-scope benchmarks, and three data set sizes.

fdtd-2d, for the medium data set, and CLAM and SHEL lease policies. Program classification is explained in Section 5.1.

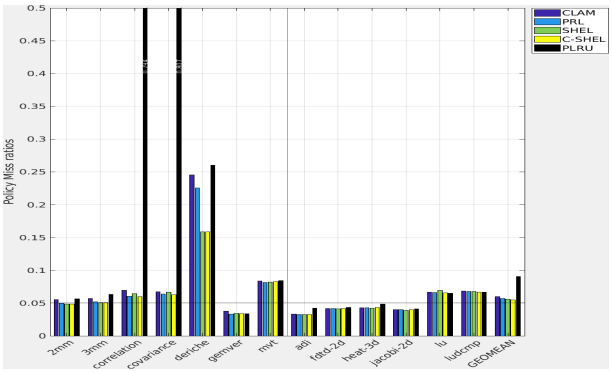
Each spectrogram (or spectra) shows cache state, sampled every 256 L2 accesses, during the entire execution of a program. Each vertical slice is a cache state, which is colored for each cache line for its lease state. The legend at the bottom shows how the lease is represented by a range of colors, from a long lease in dark blue to an expired lease in yellow. Our test programs need the full capacity of the cache. They use more data than the cache can hold, and they incur cache misses throughout the execution. A spectrogram shows the quality of a lease policy whether it fully utilizes the available cache space. In a spectrogram, too much yellow means the cache is under-utilized: while program data exceeds the cache capacity, not all cache blocks are "leased." Too much deep blue means that the cache is over allocated: they are no cache line with an expired or expiring lease, and a cache miss will cause a random eviction.



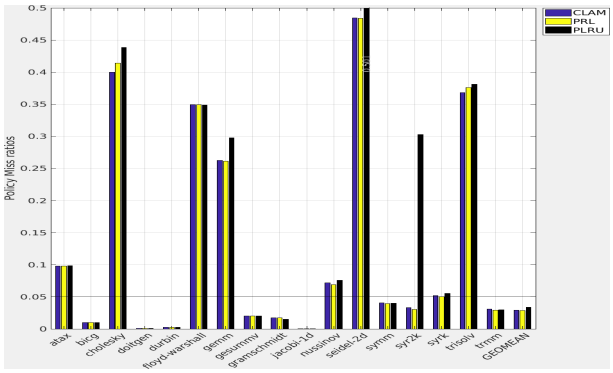
(a) Multiple scope, SMALL_DATASET, Miss ratios without Pluto optimization.



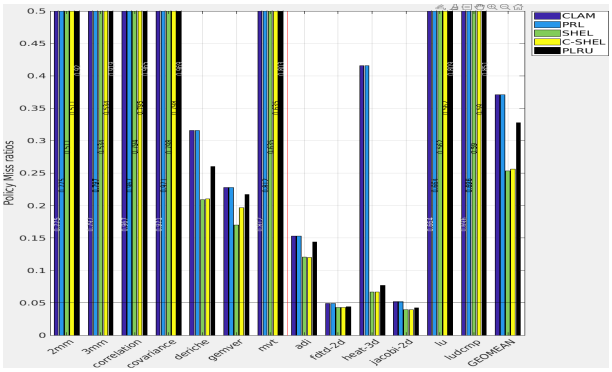
(b) Single scope, SMALL_DATASET, Miss ratios without Pluto optimization.



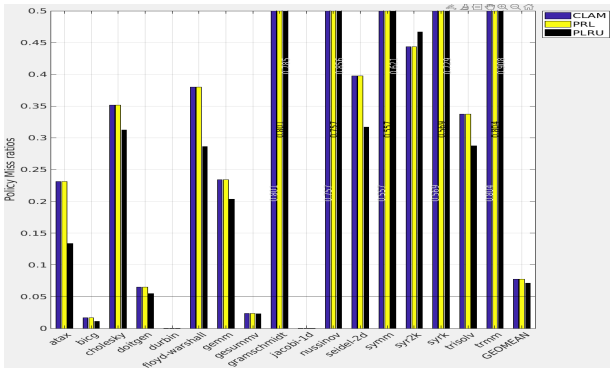
(c) Multiple scope, MEDIUM_DATASET, Miss ratios without Pluto optimization.



(d) Single scope, MEDIUM_DATASET, Miss ratios without Pluto optimization.



(e) Multiple scope, LARGE_DATASET, Miss ratios without Pluto optimization.



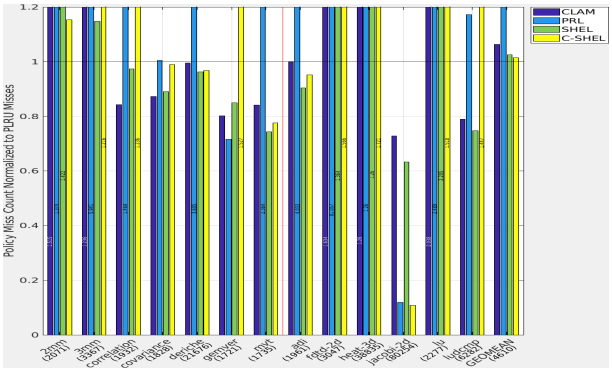
(f) Single scope, LARGE_DATASET, Miss ratios without Pluto optimization.

Figure 9: Miss ratios without Pluto optimization for multiple- and single-scope benchmarks

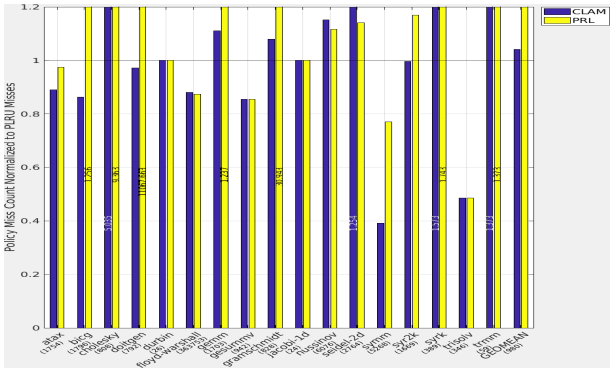
Both of these situations are undesirable, and therefore "ideally" the graphs show an even mix of colors with one cache line colored yellow or nearly yellow.

In each of these two Figures, every four graphs are for a particular benchmark. The effect of lease policy improvement is shown by the change from top-left to top-right, the effect of program optimization (Pluto) from top-left to bottom-left, and the combined effect by the bottom-right. In other words, cache programming causes the changes from left to right, and program optimization from top to bottom.

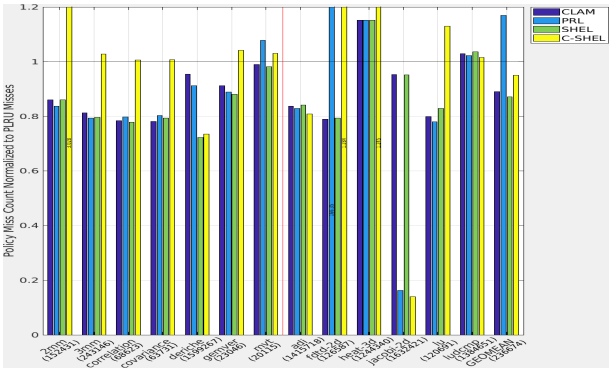
In an ideal lease cache, the size varies arbitrarily. Hardware cache has a fixed size. In multi-scope programs, leases are assigned for the program as a whole may over allocate in one phase and under allocate in another. This is seen in the spectrogram from CLAM leases, with large blocks of yellow. The uneven allocation is reduced by SHEL, which assigns leases within each scope and by program optimization, which reduces the distance between data



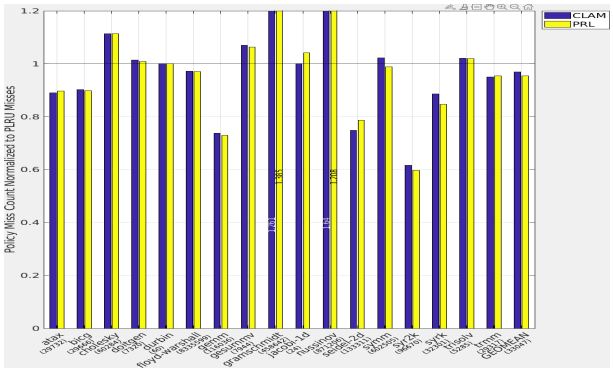
(a) Multiple scope, SMALL_DATASET, Misses normalized to PLRU with Pluto optimization.



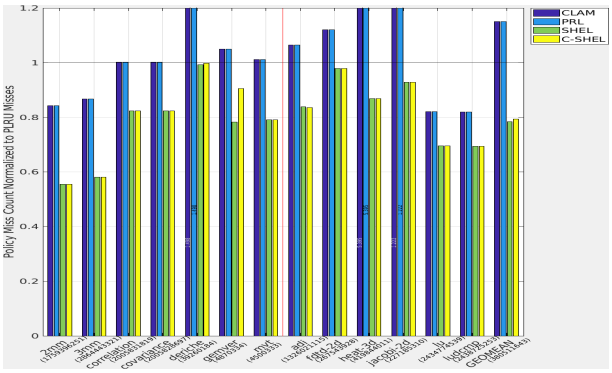
(b) Single scope, SMALL_DATASET, Misses normalized to PLRU with Pluto optimization.



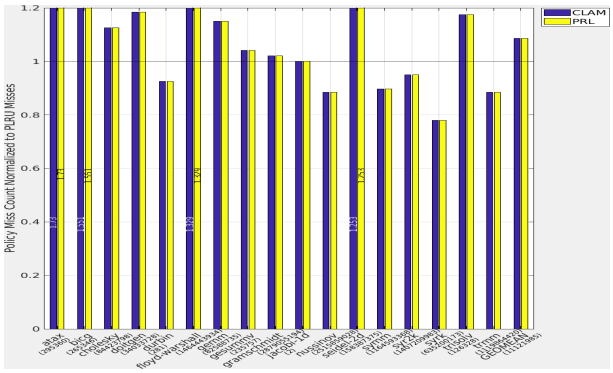
(c) Multiple scope, MEDIUM_DATASET, Misses normalized to PLRU with Pluto optimization.



(d) Single scope, MEDIUM_DATASET, Misses normalized to PLRU with Pluto optimization.



(e) Multiple scope, LARGE_DATASET



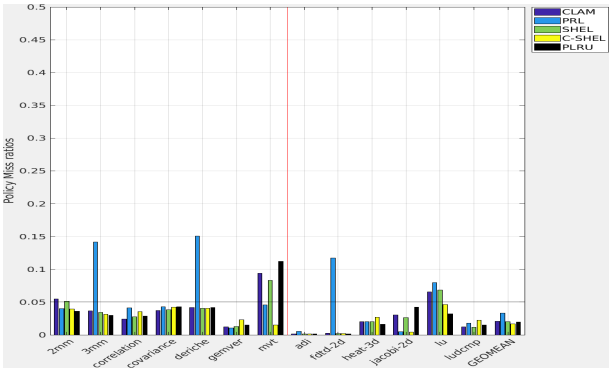
(f) Single scope, LARGE_DATASET

Figure 10: Misses normalized to PLRU with Pluto optimization for multiple- and single-scope benchmarks

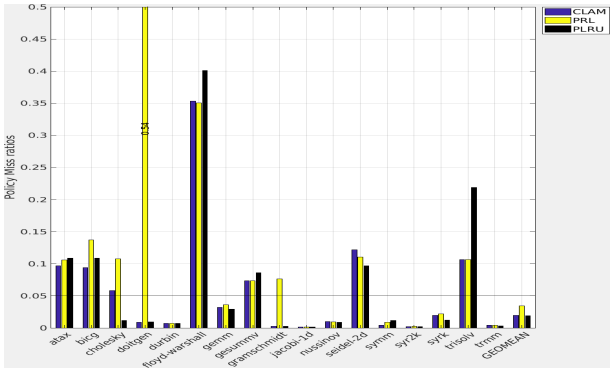
reuses and may fuse multiple loops together. When the two methods are combined, the evenness in cache allocation is retained.

6 Conclusions

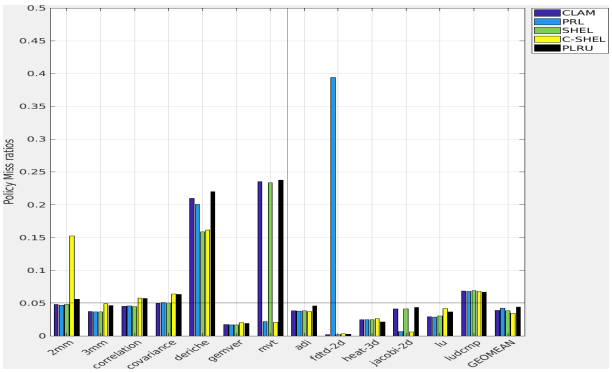
The programmable cache emulation and test system described in this paper instantiates a RISC-V processor core and associated L1 and L2 caches. L1 instruction and data caches use PLRU, while L2 is programmable, and can use PLRU or one of four lease policies. To date, the system has been used to collect performance data for various lease cache implementations. The Polybench suite test results show that lease policies are effective in a majority of cases, with or without Pluto optimization. The two-level lease cache system reduces the main memory access by 50% to 80% on average compared to the single-level lease cache in the prior work. Compared to a two-level cache system using PLRU, the programmable cache reduces the average miss count by 20% to 40%. The system can also



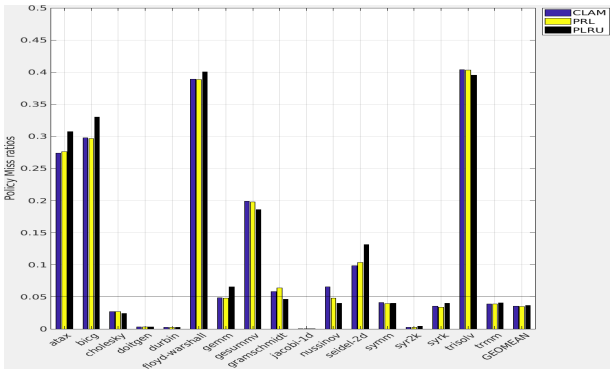
(a) Multiple scope, SMALL_DATASET, Miss ratios with Pluto optimization.



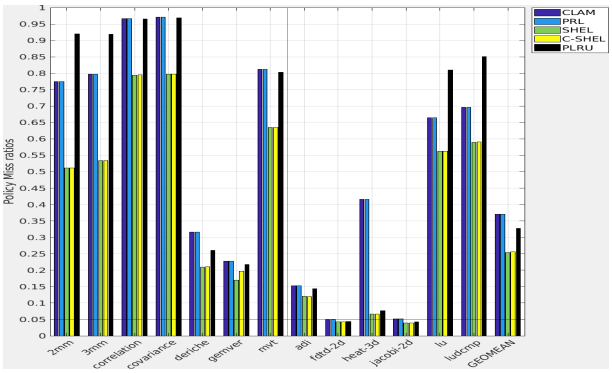
(b) Single scope, SMALL_DATASET, Miss ratios with Pluto optimization.



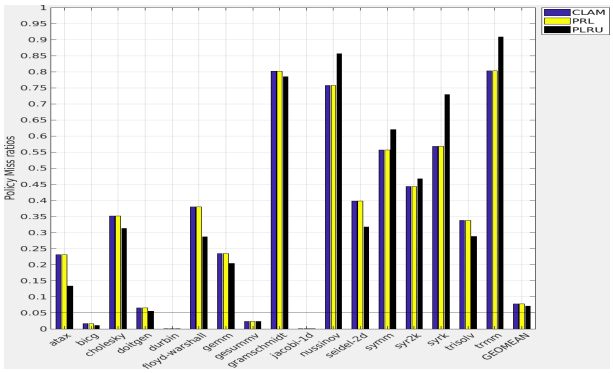
(c) Multiple scope, MEDIUM_DATASET, Miss ratios with Pluto optimization.



(d) Single scope, MEDIUM_DATASET, Miss ratios with Pluto optimization.



(e) Multiple scope, LARGE_DATASET, Miss ratios with Pluto optimization.



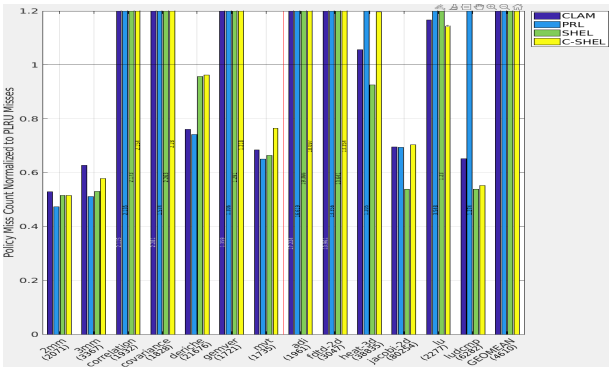
(f) Single scope, LARGE_DATASET, Miss ratios with Pluto optimization.

Figure 11: Miss ratios with Pluto optimization for multiple- and single-scope benchmarks

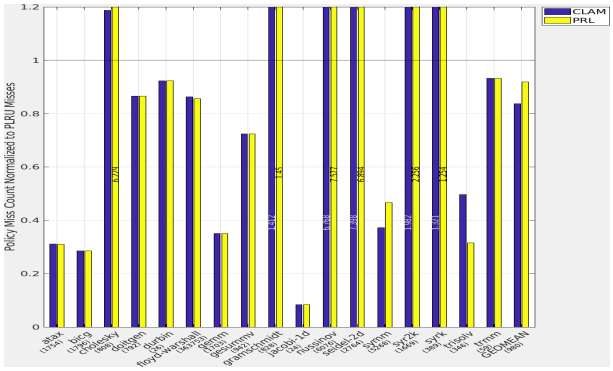
be used to emulate a processor with associated reconfigurable and programmable caches.

Acknowledgments

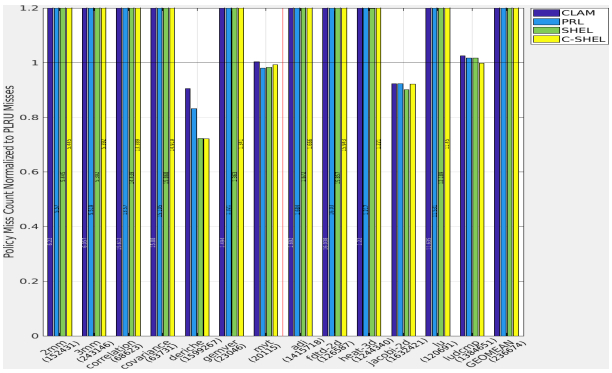
The authors wish to thank Woody Wu, Jack Cashman and Leo Sciortino for proof reading of the paper. This work was supported in part by the National Science Foundation (Contract No. CCF-2217395, CCF-2114285, CCF-2114319, CNS-1909099). Conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the funding organizations.



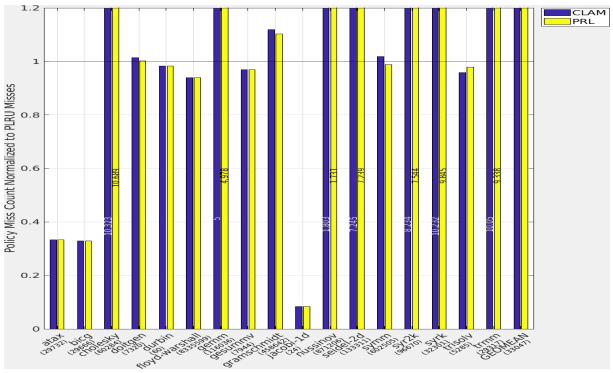
(a) Multiple scope, SMALL_DATASET, Misses without Pluto normalized to PLRU with Pluto.



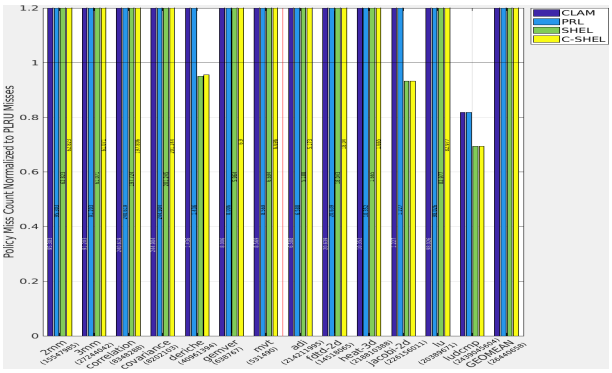
(b) Single scope, SMALL_DATASET, Misses without Pluto normalized to PLRU with Pluto.



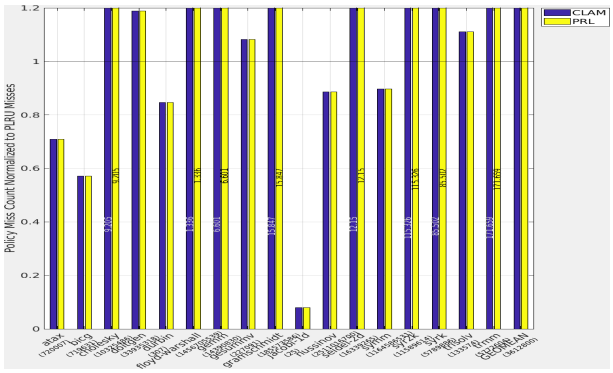
(c) Multiple scope, MEDIUM_DATASET, Misses without Pluto normalized to PLRU with Pluto.



(d) Single scope, MEDIUM_DATASET, Misses without Pluto normalized to PLRU with Pluto.

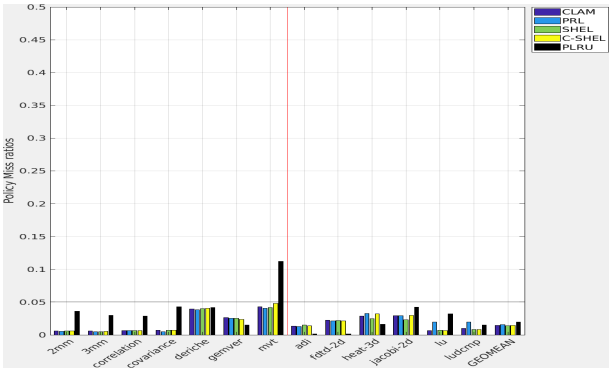


(e) Multiple scope, LARGE_DATASET, Misses without Pluto normalized to PLRU with Pluto.

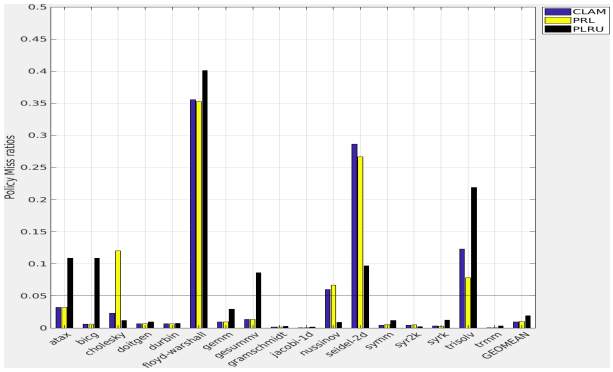


(f) Single scope, LARGE_DATASET, Misses without Pluto normalized to PLRU with Pluto.

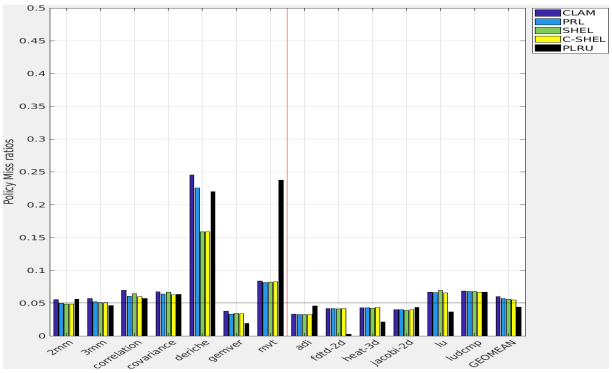
Figure 12: Misses without Pluto normalized to PLRU with Pluto for multiple- and single-scope benchmarks



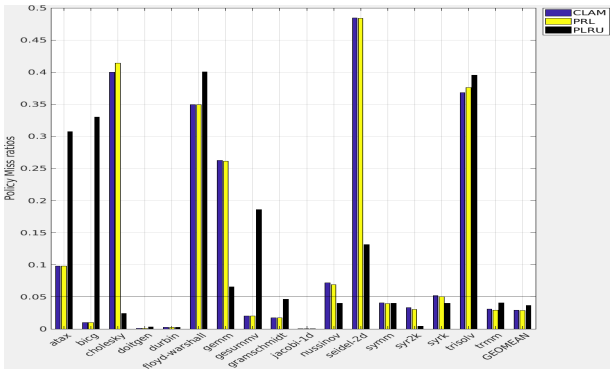
(a) Multiple scope, SMALL_DATASET, Miss ratios without Pluto versus PLRU with Pluto.



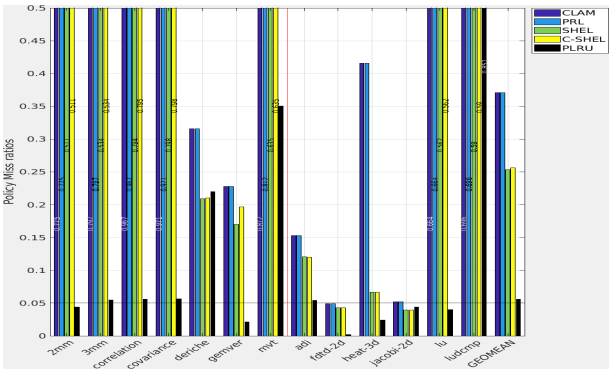
(b) Single scope, SMALL_DATASET, Miss ratios without Pluto versus PLRU with Pluto.



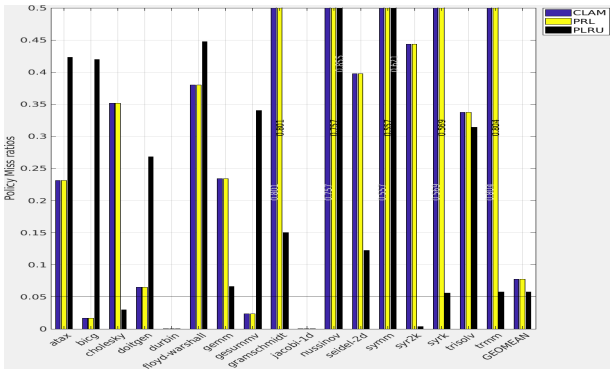
(c) Multiple scope, MEDIUM_DATASET, Miss ratios without Pluto versus PLRU with Pluto.



(d) Single scope, MEDIUM_DATASET, Miss ratios without Pluto versus PLRU with Pluto.

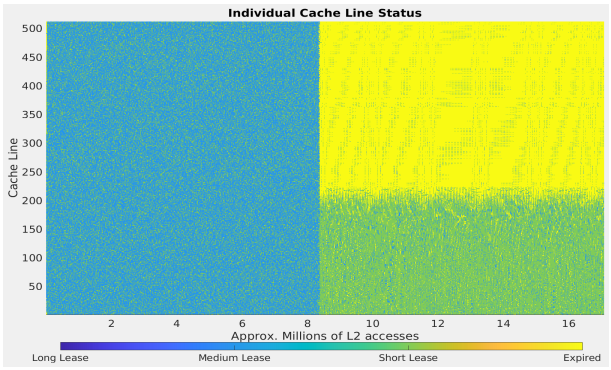


(e) Multiple scope, LARGE_DATASET, Miss ratios without Pluto versus PLRU with Pluto.

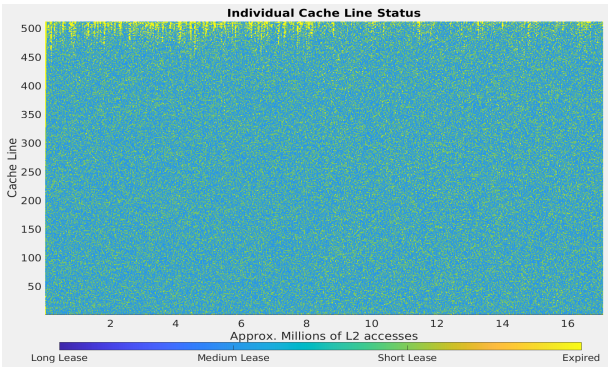


(f) Single scope, LARGE_DATASET, Miss ratios without Pluto versus PLRU with Pluto.

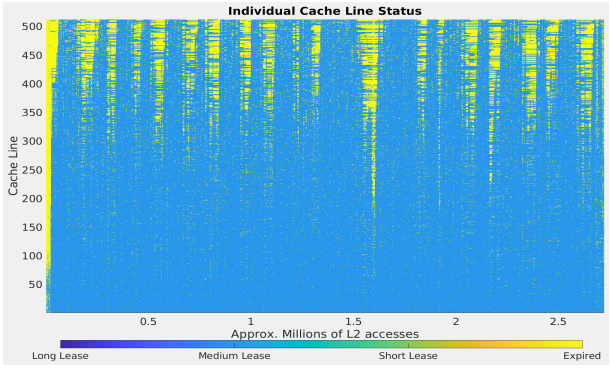
Figure 13: Miss ratios without Pluto versus PLRU with Pluto for multiple- and single-scope benchmarks



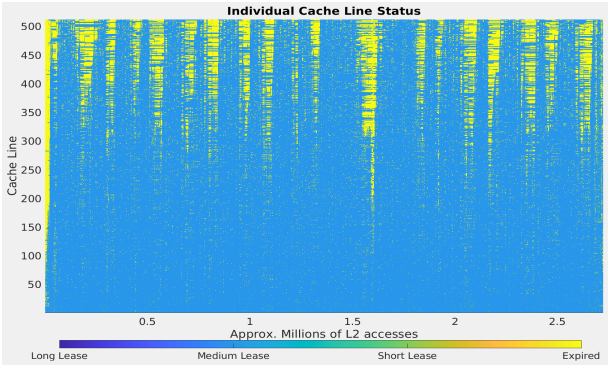
(a) 2mm, MEDIUM, CLAM lease without Pluto.



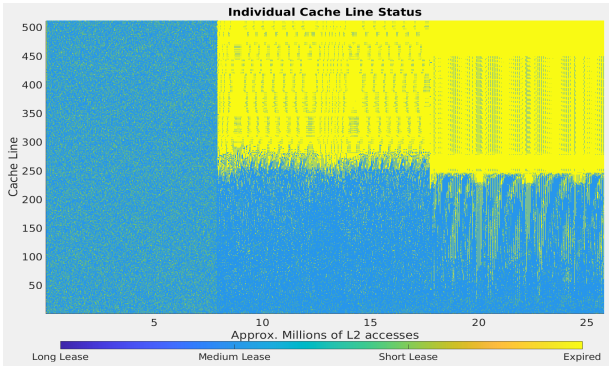
(b) 2mm, MEDIUM, SHEL lease without Pluto.



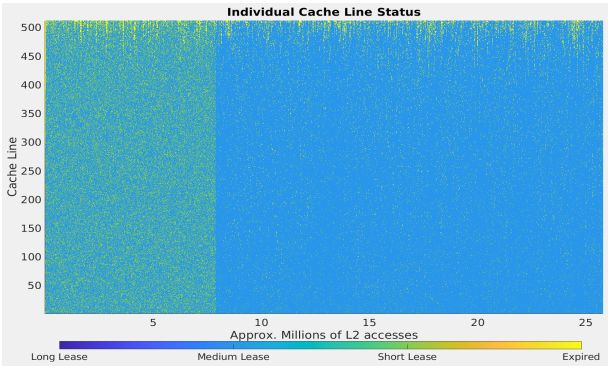
(c) 2mm, MEDIUM, CLAM lease with Pluto.



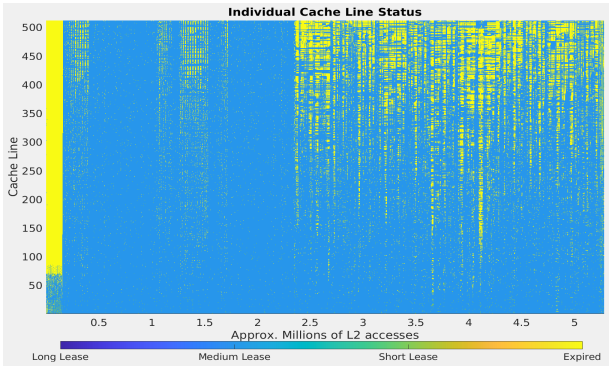
(d) 2mm, MEDIUM, SHEL lease with Pluto.



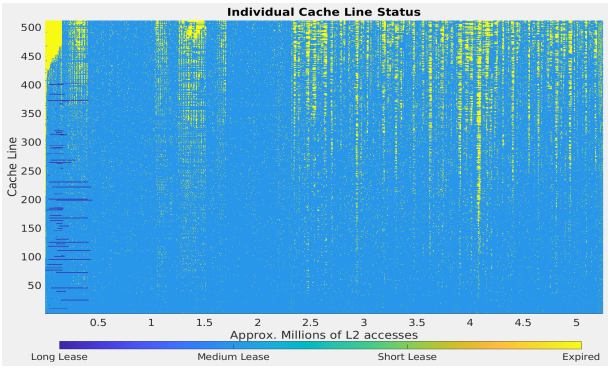
(e) 3mm, MEDIUM, CLAM lease without Pluto.



(f) 3mm, MEDIUM, SHEL lease without Pluto.

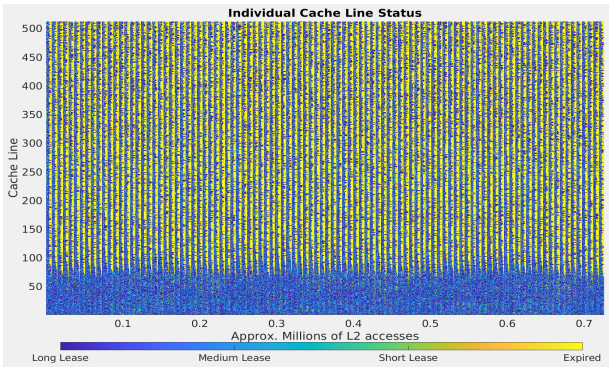


(g) 3mm, MEDIUM, CLAM lease with Pluto.

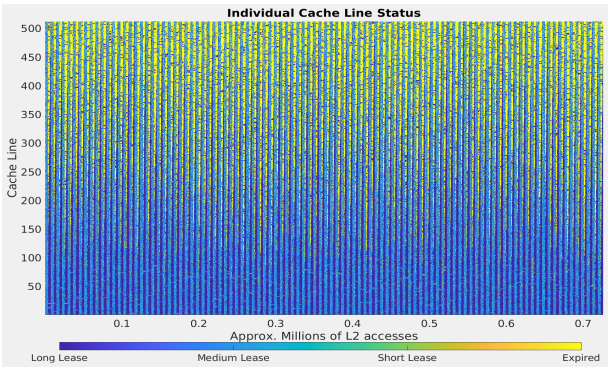


(h) 3mm, MEDIUM, SHEL lease with Pluto.

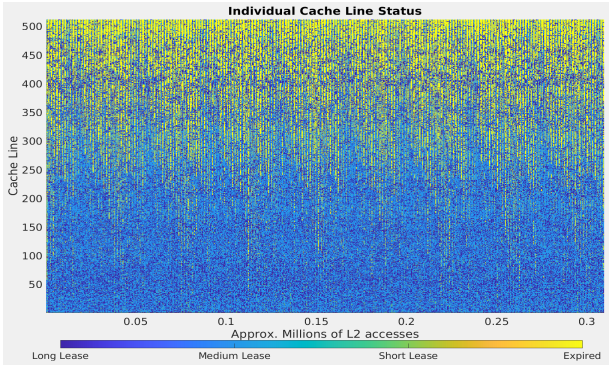
Figure 14: Occupancy Spectra of acyclic benchmarks 2mm and 3mm, for medium data set, CLAM and SHEL lease policies. Blue indicates lease remaining, and yellow indicates an expired lease in a cache line.



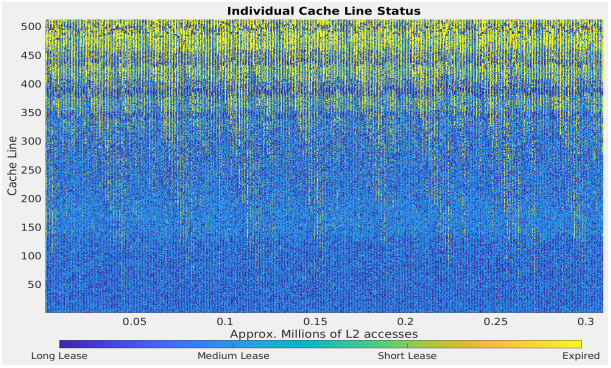
(a) adi, MEDIUM, CLAM lease without Pluto.



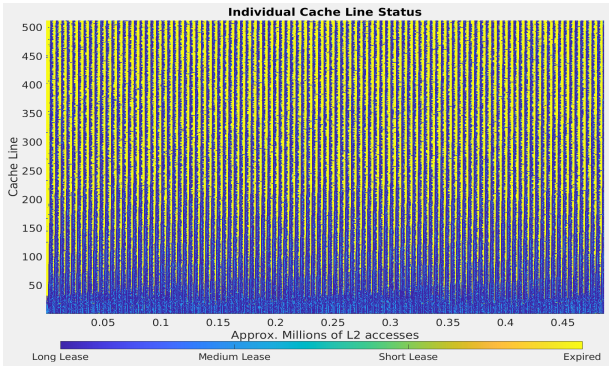
(b) adi, MEDIUM, SHEL lease without Pluto.



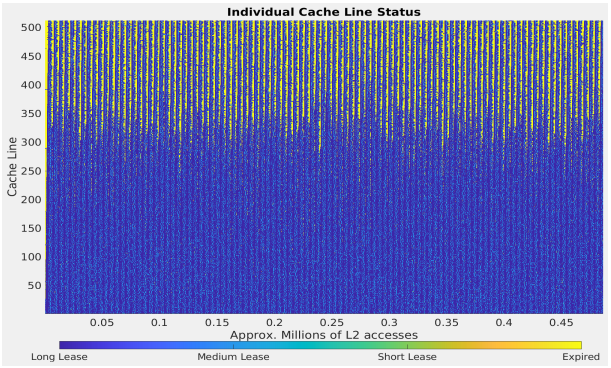
(c) adi, MEDIUM, CLAM lease with Pluto.



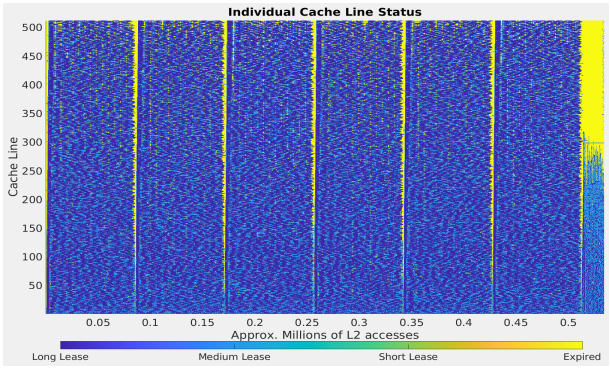
(d) adi, MEDIUM, SHEL lease with Pluto.



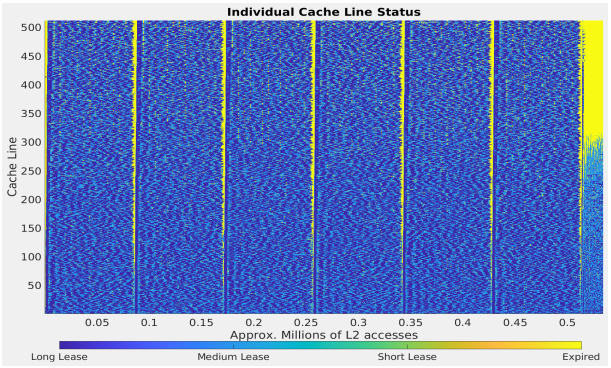
(e) fdtd-2d, MEDIUM, CLAM lease without Pluto.



(f) fdtd-2d, MEDIUM, SHEL lease without Pluto.



(g) fdtd-2d, MEDIUM, CLAM lease with Pluto.



(h) fdtd-2d, MEDIUM, SHEL lease with Pluto.

Figure 15: Occupancy Spectra of cyclic benchmarks adi and fdtd-2d, for medium data set, CLAM and SHEL lease policies. Blue indicates lease remaining, and yellow indicates an expired lease in a cache line.

References

[1] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 101–113.

[2] Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. 2018. Locality analysis through static parallel sampling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 557–570. https://doi.org/10.1145/3192366.3192402

[3] 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.

[4] RISC-V International. 2024. The RISC-V Instruction Set Manual Volume 2: Privileged Architecture.

[5] Louis-Noël Pouchet. [n. d.]. PolyBench/C 4.0. http://polybench.sourceforge.net.

[6] 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.

[7] 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.

[8] 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.