Cache Programming for Scientific Loops Using Leases

BENJAMIN REBER, University of Rochester
MATTHEW GOULD and ALEXANDER H. KNEIPP, Rochester Institute of Technology
FANGZHOU LIU, University of Rochester
IAN PRECHTL, Rochester Institute of Technology
CHEN DING, University of Rochester
LINLIN CHEN and DORIN PATRU, Rochester Institute of Technology

Cache management is important in exploiting locality and reducing data movement. This article studies a new type of programmable cache called the lease cache. By assigning leases, software exerts the primary control on when and how long data stays in the cache. Previous work has shown an optimal solution for an ideal lease cache.

This article develops and evaluates a set of practical solutions for a physical lease cache emulated in FPGA with the full suite of PolyBench benchmarks. Compared to automatic caching, lease programming can further reduce data movement by 10% to over 60% when the data size is 16 times to 3,000 times the cache size, and the techniques in this article realize over 80% of this potential. Moreover, lease programming can reduce data movement by another 0.8% to 20% after polyhedral locality optimization.

CCS Concepts: • **Software and its engineering → Compilers;** • **Hardware → Dynamic memory**;

Additional Key Words and Phrases: Cache management, reuse interval distribution, cache replacement policy, lease cache, phase marking, compiler transformations

ACM Reference format:
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 Trans. Arch. Code Optim.* 20, 3, Article 39 (July 2023), 25 pages.
https://doi.org/10.1145/3600090

1 INTRODUCTION

Today's computers, such as CPUs, GPUs, and accelerators, have complex memory systems that all use caches. This complexity is rapidly increasing with new technology, e.g., **high-bandwidth memory (HBM)**, new material, e.g., Intel Optane, and new architectures, e.g., heterogeneous systems. This complexity is too great for purely automatic solutions to be fully effective and robust.

The manuscript is new and not a revision of a previous conference paper.
This work was supported in part by the National Science Foundation (Contract No. SHF-2217395, SHF-2114319, SHF-2114285, CNS-1909099). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.
Authors' addresses: B. Reber, F. Liu, and C. Ding, University of Rochester, Rochester, NY; M. Gould, A. H. Kneipp, I. Prechtl, L. Chen, and D. Patru, Rochester Institute of Technology, Rochester, NY.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1544-3566/2023/07-ART39
https://doi.org/10.1145/3600090

A recent design called the *lease cache* lets a program control cache management using leases [22, 29]. Each time a data block is accessed, a lease is given to specify the time of eviction if the data block is not accessed again. Such leases can be assigned for each load and store instruction and communicated to hardware when a program is loaded.

The lease cache enables program control of the cache. We refer to such program control as *cache programming*. In conventional caching, all applications use the same generic caching policy such as LRU. In the lease cache, different programs can have program-specific cache management.

The annotation for the lease cache consists of a lease for each memory reference, i.e., each load or store instruction in the compiled code. We call them *reference-lease annotations*. The problem of lease assignment may be broken down into two distinct parts. The first is how to accurately gather per-reference statistics for use in lease assignment. The second is how to use these statistics to assign leases that optimize cache performance. This work focuses on the latter problem and uses a profiling pass to gather the per-reference statistics.

This article studies the problem of lease assignment, which accounts for the structure of a program, in particular, the loop structure in scientific code. A scientific application may compute in many steps that we call *phases*. Each phase may have a different data reuse pattern. Naturally, we want to assign appropriate leases based on the usage patterns. To do so, we need to solve three problems: (1) how to divide a program into phases, (2) how to assign leases in each phase, and (3) how to consider data reuses between phases.

In this article, we develop new compiler techniques for cache programming in scientific code using scoped leasing, support the new techniques with a hardware implementation on FPGA, and evaluate their performance on the complete PolyBench suite. The main contributions are as follows:

- We formulate the problem of balanced lease cache programming using leases. (Section 2.2)
- We present *scope hooked eviction leasing (SHEL)*, where each scope is a loop nest and is optimized separately (Section 2.3), and *cross-scope hooked eviction leasing (C-SHEL)*, which extends SHEL to consider cross-scope data reuses. (Section 2.4)
- We implement the system on a CycloneV-GT FPGA, including a RISC-V processor with single-precision floating-point and the ability to load scoped leases dynamically during an execution. (Section 3)
- We evaluate the system using the 30 programs from the PolyBench/C 4.2.1 benchmark suite and compare scoped leases with automatic caching and two previous leasing techniques. Furthermore, we examine the effects of polyhedral optimization and loop tiling on lease cache performance. These two evaluations of cache programming are conducted on three input sizes: small, medium, and large.

This study has several limitations. Owing to the hardware prototype, we consider sequential programs and use hardware support to collect data reuse information in program executables using a profiling pass (Section 4.1). We optimize static lease assignment given memory trace statistics. The problem of gathering reuse statistics at compile time is not explored in this article.

2 LEASE CACHE PROGRAMMING

2.1 Lease Cache

Lease Cache 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 core RISC-V executing integer and floating point manipulation instructions is connected to a single-level cache. This is controlled by a **Lease Cache Management Unit (LCMU)**, which can

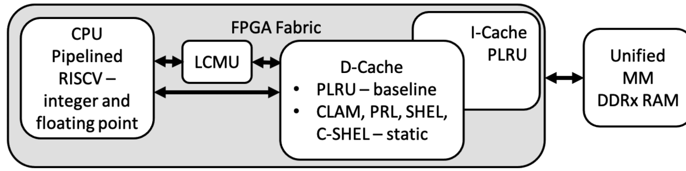


Fig. 1. Lease cache hardware prototype.

be configured to apply a conventional **Pseudo-Least Recently Used (PLRU)**, or use reference leases. The prototype is based on the test platform used in previous work [28, 29].

The following two tables compare cache programming with automatic caching and two other programming problems. First, compared to the LRU cache, the lease cache is programmable in that the eviction is determined first by a lease. Unlike the LRU cache whose actions are based on only the history information, the lease cache can be programmed based on program information. If a program requires more cache space than what is available, then the lease cache has a secondary policy, which randomly evicts a data block at a cache miss.[1] Second, a lease is an allocation, and lease programming is similar to malloc-free and register allocation. These techniques all 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 obtain as many cache hits as possible, but the use of the cache must be within a constant bound.

	LRU cache	Lease cache
primary policy	LRU, automatic	leases, programmed
info used	history only	loop analysis
secondary policy	N/A	random eviction

	Heap	Register Allocation	Lease cache
allocation	malloc/free per object	live range per data	a lease per access
granularity	object	variable	cache block
mem. size	unbounded	fixed	fixed
optimality	minimal liveness	fewest loads/stores	fewest misses

2.2 Lease Balancing

Ding et al. [17] present **Compiler Assigned Reference Leasing (CARL)**, an optimal algorithm for assigning leases to a program such that the miss ratio is minimized. However, their solution assumes a variable-sized cache that 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 value. 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.

Because virtual cache size grows and shrinks dynamically, it can exceed the physical cache size. When this happens, new data must be cached, but there is no block with an expired lease to select for replacement. In this case, a secondary policy must be used to force-evict a cacheline with an active lease. We call such an event a *forced eviction*. If the evicted data is reused before its remaining lease at the time of eviction, then its reuse is a hit in virtual cache, but it is a miss in the real cache. We call this event a *contention miss*, and it represents the degradation of idealized CARL performance on a real machine.

[1]In Appendix A in their MEMSYS 2020 paper, Prechtl et al. [29] 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.”

We denote a program region whose virtual cache size is above-average as *overallocated* and a region whose VCS is below-average as *underallocated*. CARL leases achieve optimal performance with an average allocation equal to the physical cache size. However, this may be obtained through a balance of overallocated and underallocated program regions, which require forced evictions and waste cache space, respectively. Since both of these effects degrade performance, CARL leases are not optimal in practice. We therefore seek to augment CARL lease assignments such that the variance in VCS is limited and the cache allocation is balanced.

We examine four CARL-based lease assignment techniques, which seek to balance cache allocation in different ways.

CLAM. **Compiler Lease of Cache Memory**, which applies CARL, setting the average VCS of the whole program to be PCS. This is the naive solution with no balancing that causes the most cache contention.

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. See Section 2.3.

C-SHEL. **Cross-Scope Hooked Eviction Leasing**, which augments SHEL by considering cross-loop reuses. See Section 2.4.

PRL. **Phased Reference Leasing**, which divides the execution into equal-length intervals and constrains its VCS at each interval. PRL is the first solution to reduce cache contention [29], which we discuss in Section 2.8.

Lease programming in practice requires solving two problems: program analysis and lease assignment. Program analysis may be based on profiling or loop analysis. To focus our study entirely on the second problem, we use profiling, in particular, hardware sampling analysis. Profiling shows data reuse at binary load and store instructions and includes the effect of all compiler optimization.

2.3 Scope Hooked Eviction Leasing

We first define the concepts of scopes and phases.

Reuse Intervals, Scopes, Phases, and Cross-scope Data Reuse. We assume scientific code has a regular structure: A program is a series of statements and loop nests. Each level of a loop nest contains a series of statements and loops. A *scope* is a textual region of a program in which a binding environment is active [30, Section 3.3]. It may contain a loop including its inner loops.

We manually select scopes for assigning leases. We call them *annotated scopes*. In the rest of the article, unless otherwise indicated, a scope refers to an annotated scope. A *phase* is a runtime instance of a scope. While a scope is a fragment of program code, a phase is a period of program execution.

Leases are assigned based on the **Reuse Interval (RI)**, which is 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 $RI = 5$. A *cross-scope RI* is a reuse interval that spans at least two phases of different scopes; otherwise, the RI is *scope local*. By this definition, an RI spanning two consecutive phases is still scope local if both phases are of the same scope. Scope local reuses are not a problem, because their leases allocate the cache only in the same scope, i.e., their effect is scope local.

Contention in a Fixed-size Cache. CLAM [29] assigns leases such that it targets an *average cache size* in the same way as CARL [17]. Leases are assigned based on global RI histograms and therefore blind to dynamic fluctuations in reuse behavior. CARL leases are optimal on a virtual cache, which can grow and shrink arbitrarily as long as the average size during execution matches the target [17].

When the cache has a bounded size, CARL leases may cause *cache over-allocation* or *contention*, when the number of active leases exceeds the cache size; and *cache under-allocation*, when the

number of active leases is smaller than the cache size. A lease assignment may have the correct average cache size because an overallocated portion of program execution is balanced out by an underallocated portion. Cache over-allocation will lead to contention misses, while under-allocation will result in fewer hits. CLAM is the naïve lease assignment policy and has no mechanism to mitigate these effects.

Scope Hooked Eviction Leases. Leases assigned to references based on global reuse interval histograms may target an average cache size. Because these histograms contain no information about *when* different RIs (and therefore leases) occur, cache allocation may not be balanced in the event that access patterns change significantly throughout execution.

If we assume RIs are uniformly distributed throughout execution, then cache usage variance during execution is low, and so contention misses are rare and lease assignments based on average cache size will perform well on a fixed-size cache. However, reuse behavior is not always uniform. Programs may be composed of multiple outer loops, or else alternate between multiple inner loops, each of which may have different reuse behavior.

This problem is solved by encoding time information in RI histograms, in a technique we call **Scope-Hooked Eviction Leasing (SHEL)**. In SHEL, the programmer annotates a set of program scopes. These scopes indicate program phases with possibly different reuse behavior. For simplicity, we assume each reference belongs to a single scope.[2] Thus, by including a scope field in reference RI histogram entries, lease assignment may be done on a per-scope, rather than global, granularity. This allows for leases that are less profitable globally to effectively bypass more profitable leases if they take up space during under-allocated phases. Hence, scope annotation allows for lease assignments that are more balanced throughout program execution, resulting in fewer contention misses.

It is possible for the allocation in one phase to spill over into the next phase. SHEL ignores such effects. As a result, SHEL may over-assign leases in a scope if the cache space available to the scope is reduced by the spill-over effect from the previous phase.

In programs with coarse-grain phases, cross-phase effects may be negligible, and scopes may be optimized independently. An example is a computation with two steps, and each step computes matrix multiplication. When executed, the second step runs long enough to nullify the lingering effect of any lease assigned in the first step.

Intuitively, the spill-over effects can be ignored for a program if all its phases are sufficiently longer than the longest lease. We state this property precisely, as follows:

PROPOSITION 1. *Let s_{min} be the minimal number of accesses in a phase, l_{max} the longest lease assigned, and c the cache size. If $s_{min} \gg l_{max}c$, cross-scope RIs can be ignored.*

Cross-scope RIs can be assumed to be in scope, and the resulting lease is the same. To see why this is true, consider what happens at the end of a phase. The number of remaining resident items in the cache is at most the cache size c, and they stay in the cache for at most l_{max}. The leases in a phase change the miss count in the next phase by at most $l_{max} \times c$. When the next phase is sufficiently long, the miss ratio is unaffected; hence, there is no need to consider cross-scope RIs.

In other cases, considering cross-scope RIs may lead to a different lease assignment and better cache utilization, which we show next.

2.4 Cross-scope Leasing

It may be the case that cross-scope reuses make up a significant portion of all reuses. This can occur when the program structure is composed of several alternating phases. When assigning leases in

[2]Our design allows for one reference to occur in multiple scopes due to branching or function calls. However, such behavior is not present in any of the benchmarks presented.

these phases, there is a natural question of where to attribute the cost of the allocation. The cost could naïvely be assigned fully in either the first phase, where the first use occurs, or the second phase, where the reuse occurs. However, both options miss the true program behavior, and thus may lead to incorrect allocation that harms performance.

To solve this problem, we introduce **Cross Scope-Hooked Eviction Leasing (C-SHEL)**. In C-SHEL, we record the cross-phase behavior of reuse samples and store this aggregate data along with RI histograms. We assume that sampled cross-phase RIs are representative of the full trace.

For a given reference lease, there are two components that together make up the total cost. The first, which we denote as the *head cost*, is the contribution of RIs, which are less than or equal to the lease. Accumulating the head cost of a reference lease to multiple phases is trivial; as each RI sample is processed, we simply divide the head cost among the phases it occupies.

The second cost component, which we denote as the *tail cost*, is the contribution of RIs, which are greater than the lease. Unlike head costs, accumulating the tail costs of all RIs cannot be done sample-by-sample. This is because the tail cost of a sample contributes to all RIs that are lesser. Thus, handling tail costs requires a second pass through the sampled RIs.

The result of sample processing is a set of RI histograms for each reference. For each RI, the cumulative head and tail costs in each phase are stored and can be used during lease assignment to more accurately allocate lease costs among program phases.

2.5 Instrumentation and Sampling

With our instrumentation, any scope may be annotated by the programmer. Any load or store instruction that occurs within an annotated scope is considered as part of a phase of that scope. In the case where no explicit phase marker is in scope, the last scope marker read in sequential order is used.

During trace sampling, the budget for each scope is simply determined based on the number of samples from phases associated with that scope ID. By using programmer-annotated scopes, we are able to assign leases that more accurately use cache resources.

Compiler Implementation. The compiler has two parts: analysis and code generation. The first collects the RI histograms, and the second inserts reference leases. The code generator inserts a table in a data segment of the binary code to store the lease annotations. We have implemented source-level compiler analysis in LLVM based on **Static Parallel Sampling (SPS)** [12]. It analyzes and assigns leases for array references. However, source-level analysis cannot determine the corresponding load and store instructions in the binary code, nor can it determine the machine code address. Therefore, we adopted an alternative solution and used profiling by running the program twice. The first execution samples RIs and outputs their reference by its binary instruction address. The code generator then computes and inserts reference leases based on the sampled RIs. In the second execution, the generated code is tested for performance. The code generator implements all leasing techniques, including CLAM and PRL from previous work and SHEL and C-SHEL in this article.

2.6 Scope Annotation

We statically place scope markers in program code. A phase is then the execution between any two consecutive scope markers. In loop-based code, markers are placed by the following two rules. First, each outermost loop is a separate scope, i.e., given its own marker.

Some scientific computing problems are solved by an iterative method. We describe such program structure as having *cyclic phases.* The second rule applies to these programs, where we insert markers for the outermost loops inside the time-step loop. The problem of cross-phase reuse is particularly important for cyclic programs.

The question of how to select scopes in general code, and how to automate this process in a compiler, is an interesting open problem, which is outside the scope of this work. For this reason and for now, we leave the problem of scope annotation to the programmer.

Table 1 shows the statistics about the scope markers for the 30 benchmarks in PolyBench. It shows the number of scope markers used in each program and the number of runtime phases for each input data size. Acyclic programs have one phase per scope, whereas cyclic programs have multiple phases per scope.

2.7 Lease Assignment Algorithms

Here, we describe in greater detail the two lease assignment algorithms that use annotated scopes. One is **Scoped Hooked Eviction Leasing (SHEL)** and the other is **Cross-Scope Hooked Eviction Leasing (C-SHEL)**. SHEL assigns leases independently for each phase without considering cross scope RIs and C-SHEL considers the cross-scope RIs.

Algorithm 1 presents the lease assignment process for SHEL and C-SHEL. The inputs for both algorithms are the same. s is the number of scopes we have for a program. For each scope, we record its set of **reuse interval histograms**, **RIHs**, and cumulative phase length p. The phase length is assumed to be proportional to the number of RI samples among histograms of a scope. For C-SHEL, these RIHs contain head and tail costs for each reference, as described in Section 2.4. With this information, each algorithm produces reference lease assignments for a specified cache size c. Both SHEL and C-SHEL rank all candidate lease values according to their **PPUC (profit per unit cost)**. PPUC calculates the number of hits divided by the cache use for an assignment. Higher PPUC for a lease assignment means the same time-space cost can produce more hits, i.e., it is a more efficient use of cache space. The PPUC of a lease can be calculated from the RI histogram of that reference. For each algorithm, the assignment procedure is a greedy process described in Algorithm 1. CLAM takes the reuse interval histogram *RIH* for each reference in the program and the total time-space budget. The *budget* is initialized by $c \times N$, which is the number of cache blocks times the number of accesses (denoted as N) in a program. One iteration in the while loop in line 2 will update the lease of a reference to a larger value. Line 3 chooses the new lease l and the reference *ref* to be assigned. Line 4 calculates the remaining budget after assignment in Line 3. This loop terminates when the budget is used up or all references have been assigned with the leases equal to the value of their maximum RIs in Lines 5–8.

SHEL simply applies CLAM independently for each scope in Lines 12–15. Instead of using a *budget* and an *RIH* for the entire execution, *RIH* and *budget* are collected and calculated for each scope. The scopes containing more accesses will be initialized with larger budgets based on scope lengths p in line 13.

To consider cross-scope RIs, C-SHEL distributes the budget to all phases and updates them simultaneously. Any assignment of a lease that results in cross-scope reuses increases the use of all scopes it reaches in Line 22. If a new lease has any impact in a scope whose budget is already used up, then the lease is rejected and the next-most-profitable lease is checked. C-SHEL terminates when all budgets for all scopes are used up or all leases are assigned to the maximum values.

2.8 Phased Reference Leasing (PRL)

The first solution to reduce the amount of cache contention is **Phased Reference Leasing (PRL)**, developed by Prechtl et al. [29]. In PRL, the overall cache capacity for the program is naïvely split into equal-width intervals. Leases are assigned step-by-step as in CARL, except that an assignment step is canceled if it causes the VCS in some interval to exceed PCS. By skipping assignment steps, it is a constrained version of CARL.

ALGORITHM 1: Lease assignment procedure

```
Input: s: number of annotated scopes
       p[0..s-1]: cumulative phase length per scope
       RIH[0..s-1]: reuse interval histograms per scope
       c: cache size
1  Function clam(RIH, budget):
2      while True do
3          ref, l = max_ppuc (RIH);
4          leases,budget = update(ref, l);
5          if (∀ leases assigned to maximum RIs
6             or budget is used up) then
7              return leases;
8          end
9      end
10 End
11 Function shel(s, p, RIH, c):
12     for i ∈ 1..s do
13         budget[i] = p[i] * c;
14         leases[i] = clam (RIH[i], budget[i]);
15     end
16     return leases;
17 End
18 Function c-shel(s, p, RIH, c):
19     budget[i] = p[i] * c, ∀i ∈ 1..s;
20     while True do
21         ref, l = max_ppuc (RIH);
22         leases[i], budget[i] = update(ref, l), ∀i ∈ 1..s;
23         if (∀ leases assigned to maximum RIs
24            or ∀ budget[i] is used up, i ∈ 1..s) then
25             return leases;
26         end
27     end
28 End
```

There are several drawbacks to this approach. First, it only accounts for coarse-grained contiguous intervals. In programs with cyclic phases where each phase has a short length, e.g., a single outer loop containing multiple inner loops exhibiting phase behavior, an interval includes phases of different behavior. The interval-based constraint cannot remove imbalanced allocation across phases.

Compared to SHEL, PRL does not have fine-grained control to set interval boundaries to match program phases. This may lead to inaccurate phase marking that can harm performance as follows: We define a *boundary interval* as one that contains the boundary between two program phases. A boundary interval contains pieces from different phases. We call others *interior intervals*. Now consider a simple case where there are two loops, L_1, L_2. PRL sees interior intervals for each loop and a boundary interval. When assigning leases for L_1, PRL considers the behavior of its interior intervals and the boundary interval. Since the boundary interval contains pieces of L_2, its behavior differs and may prevent PRL from assigning the best lease for interior intervals. Since the less-than-the-best lease is used on all interior intervals, the missing opportunity of optimization can be significant if L_1 is long running.

3 HARDWARE EMULATION SYSTEM DESIGN

The Hardware Emulation System, Figure 2, has two objectives: first, to be able to instantiate a single-core CPU (RISC-V) and associated programmable and re-configurable cache memory, and second, to assess the impact of adding the former cache memory features to actual hardware. The first objective supports the testing and comparison of various lease cache policies, while the second objective offers a cost estimate of implementing the former in actual hardware.

The lease cache hardware is able to support the application of lease policies from compilation to program execution. Its architecture is based on that described in Prechtl et al. [29], to which the current work makes two major additions: RISC-V 32F (floating point) instruction set extensions and

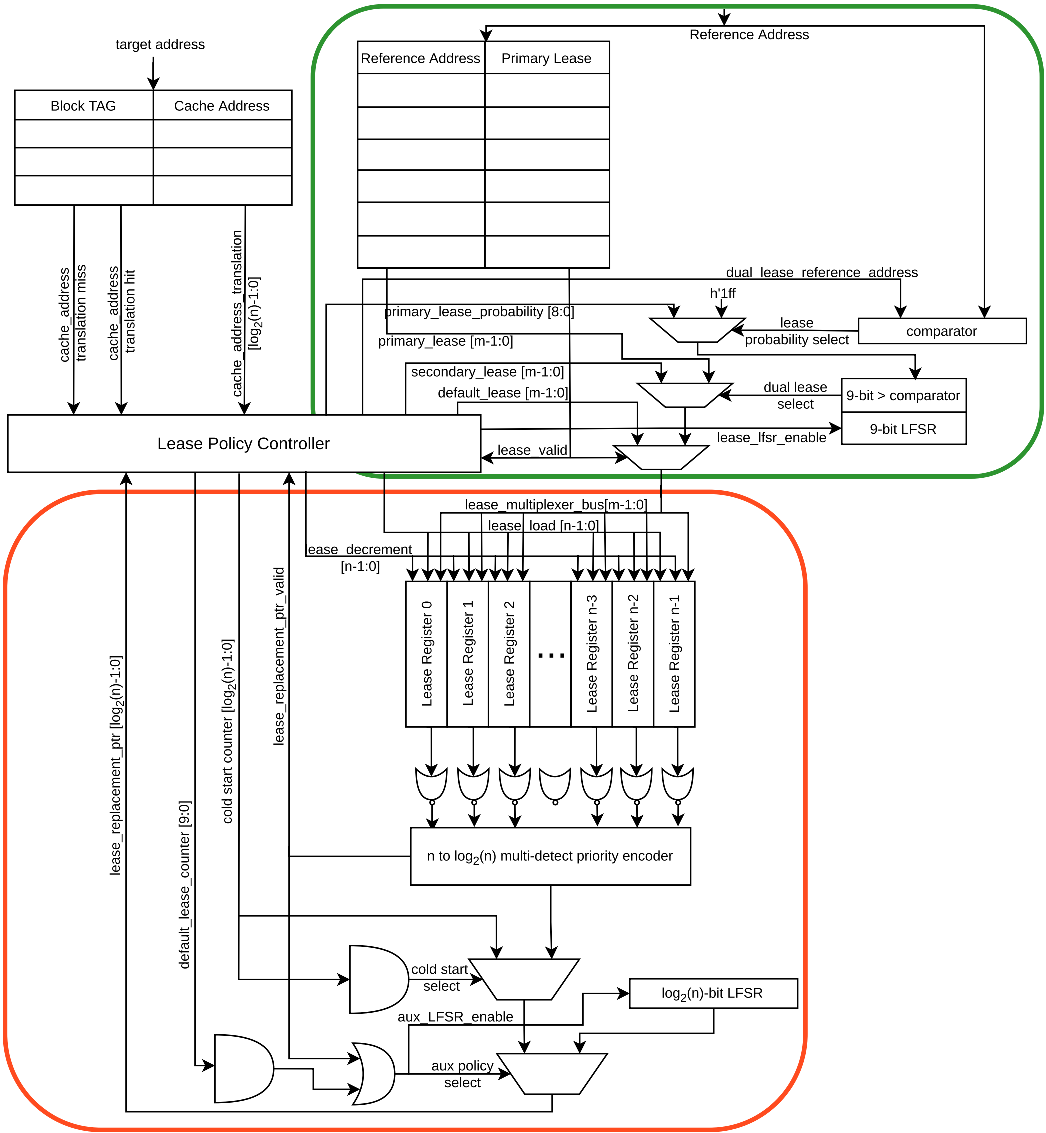


Fig. 2. Lease cache hardware architecture for a cache of *n* blocks and lease register size of *m* bits. The components in the green box are the lease look-up circuitry. The components in the red box are the replacement logic and lease update circuitry.

hardware support for scope-leasing. The prototype runs all 30 programs of the PolyBench suite, while previous work ran just 7 of the 30 [29]. In the interest of reading coherence, we discuss below the detailed lease cache architecture.

Lease Assignment. The hardware that implements a lease policy complements an existing cache memory infrastructure through the addition of a lease policy controller (Figure 2). 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 addresses 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 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, then 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. In this way, lease selection is not transparent to the policy controller and strictly abides by CLAM/PRL/SHEL/C-SHEL. 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 after their immediate use these are eligible for eviction.

Line Vacancy. Each cache line 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), then 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 leases assignments in a row. This is to handle the possibility that the assumption that references without an associated lease assignment have no near future re-reference is invalid. If that is the case, then the lease cache would perform poorly, as it is just evicting the highest expired line in the set and completely ignoring any type of data locality. Hence, the lease cache is designed such that if there are more than x (for this work 1,024 was chosen) default leases assigned consecutively, the lease cache will exclusively use the auxiliary policy until an LLUT hit occurs, whereupon normal operation resumes.

The Application of a Lease Policy is illustrated in Figure 3. 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), then 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 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 Figure 4. 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

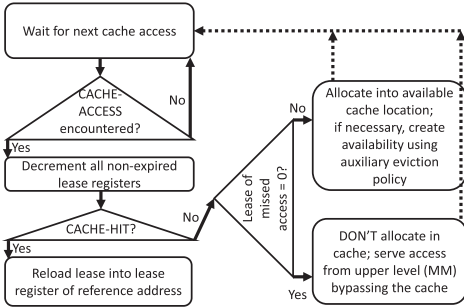


Fig. 3. Lease cache operation flowchart.

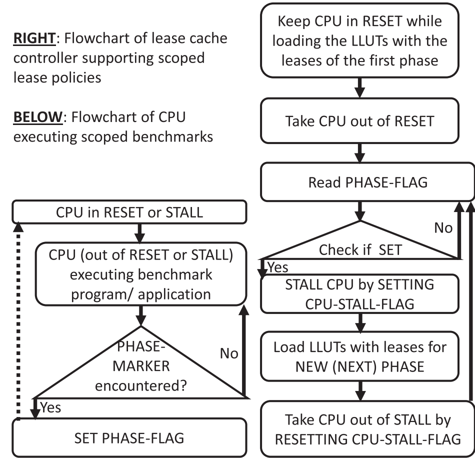


Fig. 4. Scoped leases flowchart.

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.

Expanded RISC-V Instruction Set Support. The RISC-V core used by Prechtl et al. [29] supported only the RISC-V 32IM extensions, which limited the testing to only 7 out of the 30 benchmarks in the PolyBench suite. The current RISC-V core has been redesigned to support the 32F extension (single-precision floating-point), which allows the execution of the entire PolyBench suite.

4 EVALUATION

4.1 Experimental Setup

Implementation The cache size is 8 KB with 128 cache blocks. The baseline is automatic caching, for which we use the **Pseudo Least Recently Used (PLRU)** eviction policy, a commonly used approximation of LRU that is more time- and space-efficient to implement (using a single status bit in each cache line) [31]. We compare four cache programming techniques: CLAM, which does not consider phase variation; PRL, which divides a program execution into a fixed number of phases (Section 2.8); SHEL, which uses scope local leases (Section 2.3), and C-SHEL, which assigns inter-scope leases (Section 2.4). For PRL, we divide executions into five equal-length phases, as was done in Reference [29]. In addition, we have implemented the FPGA to output the aggregate vacancy and the remaining lease values for visualization (Section 4.2).

Benchmarks. We use PolyBench/C 4.2.1, which contains 30 numerical kernels [26]. We use PolyBench for several reasons. First, the benchmark suite is relatively easy to port through the FPGA tool chain to allow testing on a real system. Second, the current emulation system is not yet equipped to execute other benchmark suites due to limited RISCV instruction set coverage. Despite the latter limitations, PolyBench kernels are extracted from linear algebra, image processing, physics simulation, dynamic programming, and statistics, which are all common workloads in scientific computing and have been extensively used in studying performance analysis [1, 25] and optimizations [2, 19]. We compile each program with the `GCC -O3` optimization level without vectorization, which our CPU does not currently support, and report the results for small, medium,

Table 1. Phase Markers Inserted for PolyBench

benchmark	# of scope markers	# of phases for different datasets		
		small	medium	large
2mm	2	2	2	2
3mm	3	3	3	3
adi	3	80	200	1,000
correlation	4	4	4	4
covariance	3	3	3	3
deriche	6	6	6	6
fdtd-2d	3	120	300	1,500
gemver	4	4	4	4
heat-3d	2	80	200	1,000
jacobi-2d	2	80	200	1,000
lu	2	240	800	4,000
ludcmp	4	242	802	4,002
mvt	2	2	2	2
atax, bicg, cholesky, doitgen, durbin, floyd-warshall, gemm, gesummv, gramschmidt, jacobi-1d, nussinov, seidel-2d, symm, syr2k, syrk, trisolv, trmm	1	1	1	1

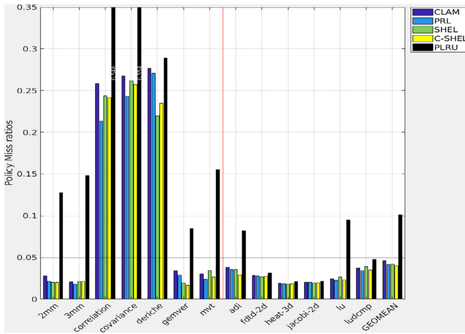
17/30 of them only have single scope/phase across all three datasets.

and large dataset sizes. Approximately, the amount of program data in these sizes are 128 KB, 1 MB, and 25 MB, respectively.

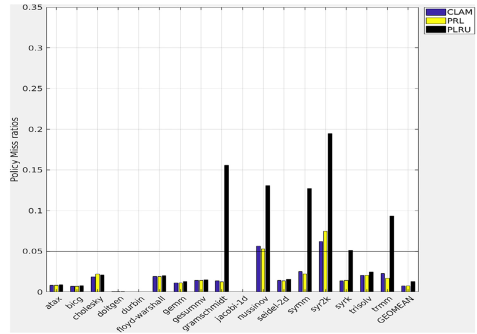
PLUTO Compiler. Version 0.11.4 of the PLUTO optimizing compiler was additionally used to optimize the PolyBench benchmarks [9]. The compiler was invoked using only the `--tile` option, to enable cache tiling and polyhedral optimizations. The compiler was configured to produce $16 \times 16 \times 16$ tiles to effectively fit in our cache size. For the code run with the CLAM and PRL leasing policies, the PLUTO output was unchanged. For codes run with the SHEL and C-SHEL policies, the PLUTO output needed to be manually annotated to work with the current lease generator application with scope markings. For benchmarks that produced large optimized output, we separate the non-optimized source into several subregions and let the PLUTO compiler be invoked in each subregion. The code, after the PLUTO optimization, was then compiled and run exactly as described previously. All of the benchmarks were able to be optimized with the PLUTO compiler, excluding `heat-3d`, which we were not able to run and collect data for after optimization. We present the results of lease cache on programs compiled without PLUTO optimization in Section 4.2, and we explore the combined effects of lease cache and PLUTO optimization in Section 4.3.

4.2 Performance of Cache Programming in Unoptimized Loops

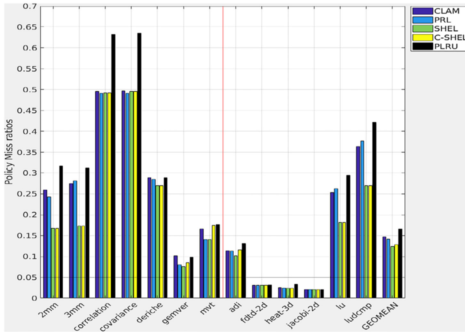
We divide the benchmarks into two groups based on the number of scopes in them (shown in Figure 5). The first group consists of 13 benchmarks that have two or more scopes, and the second group has the remaining 17. Figure 5 shows the two groups side-by-side in three rows, each showing a different data size from small (top row) to large (bottom). The x-axis shows program names, and the y-axis shows the miss ratio of the cache. We discuss the multi-scope group in this section and single-scope group later in Section B. For the multi-scope group, the red vertical line in each graph separates those with non-cyclic phases (Left) and those with cyclic phases (Right). Their phase counts are shown in Table 1.



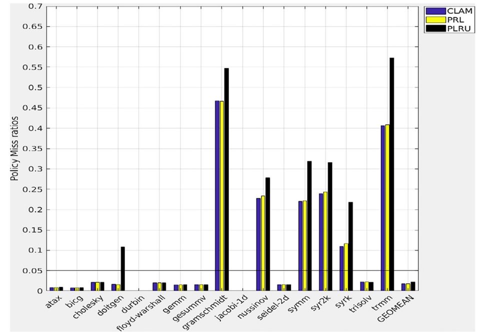
(a) Multiple scope, SMALL_DATASET



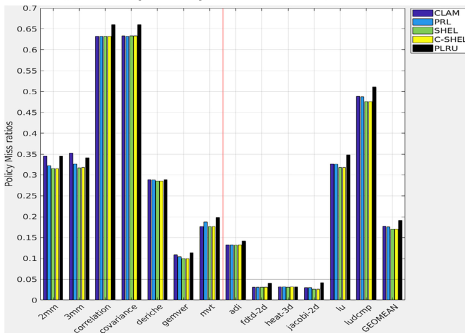
(b) Single scope, SMALL_DATASET



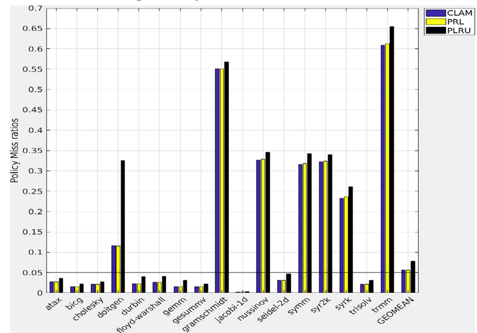
(c) Multiple scope, MEDIUM_DATASET



(d) Single scope, MEDIUM_DATASET



(e) Multiple scope, LARGE_DATASET



(f) Single scope, LARGE_DATASET

Fig. 5. Miss ratios for 13 multi-scope benchmarks (left) and 17 single-scope benchmarks (right) as well as their geometric means. Lower is better. Values are reported for small (top) medium (middle), and large (bottom) inputs. The red vertical line separates multi-scope benchmarks into those with non-cyclic (left) and cyclic phases (right).

Overall Comparison. Overall, cache programming improves significantly over automatic caching. The four techniques are used to provide leases for 13 multi-scope tests on each input, for a total of 156 ($4 \times 13 \times 3$) lease solutions. When compared with PLRU as shown in Figure 5, in 153 out of 156 cases, lease cache matches or performs better than PLRU, reducing the number of misses by over 15% in 75 (about half) solutions, by over 75% in 16 (over 10%) solutions, and by as much as 87% in the best case. Hence, we have the first finding:

Finding.

(1) *Caching programming using leases is overwhelmingly better than automatic caching (using PLRU), reducing the miss count by over 15% in one of every two cases and over 75% in every ten.*

This shows the benefit of using program information when managing the cache. Note that, in this study, we use profiling, and the test run is the same as the training run. Hence, this improvement is an ideal case.

For the multi-scope group for the small dataset, the baseline technique CLAM and PRL reduce the miss count by 55% and 58%, on average. As a reminder, PRL considers phases but not the program structure. SHEL increases the average reduction to 60%, and C-SHEL to 61%, by considering the program structure, and in the case of C-SHEL, considering cross-scope reuses.

The data sizes in medium and large inputs are 1 MB and 25 MB and much greater than the cache size 8 KB. The improvement in caching has a smaller effect. For medium, the average reduction is 12%, 15%, 26%, and 21% for CLAM, PRL, SHEL, and C-SHEL, respectively. For large, these are 8%, 9%, 11%, and 11%.

Effect on Multi-scope Programs. Lease optimization is most important for multi-scope programs. We focus on the three methods that treat scopes differently, while using CLAM, which does not distinguish scopes, as a baseline. Indeed, except a few cases, they all outperform CLAM by considering each scope and assigning scope-local leases. SHEL, for example, outperforms CLAM by 6%, 15%, 4% for the three inputs, respectively. Within each scope, they use the same algorithm and differ only in how they treat the boundary effect. Among the three, no single method dominates in all cases. On average, the lowest miss count is obtained by C-SHEL on the small dataset, SHEL on the medium, and both SHEL and C-SHEL on the large. SHEL performs well except on the small input, where it performs the worst. While all three methods assign different leases for different program phases, SHEL ignores cross-loop reuses when assigning leases. On the small input, however, phases are the shortest, and cross-loop reuses are important to cache management.

By considering these reuses, both C-SHEL and PRL outperform SHEL on small inputs. Between the two, C-SHEL performs better than PRL on average on all three input sizes, for mainly two reasons. First, C-SHEL phases are marked according to program structure, while PRL intervals are even divisions. PRL optimization is less effective in boundary intervals where the reuse behavior is mixed. Second, as discussed in Section 2.8, PRL considers cross-loop reuses only in a boundary interval but uses the same leases on all interior intervals. The lease assigned based on a boundary interval may be sub-optimal for interior intervals. The first weakness can be ameliorated by using more intervals, but the second cannot. In comparison, C-SHEL considers the boundary effect proportionally. For our tests, however, the results show that it is better to ignore these effects on medium and large inputs. C-SHEL gives no significant advantage over SHEL on any of these tests but is significantly worse on some of them. In summary, we find the following:

Finding. *On multi-scope tests, the three lease-optimization methods show that:*

(2) *It is best for lease optimization to ignore cross-loop reuses, that is, SHEL is the best or close to the best in all cases.*
(3) *C-SHEL, by considering boundary reuses, has significant benefits on small inputs, is counter productive on medium inputs, and has no effect on large.*
(4) *PRL, by constraining CARL using intervals, is always beneficial compared to CLAM, but less effective than SHEL on medium and large inputs*

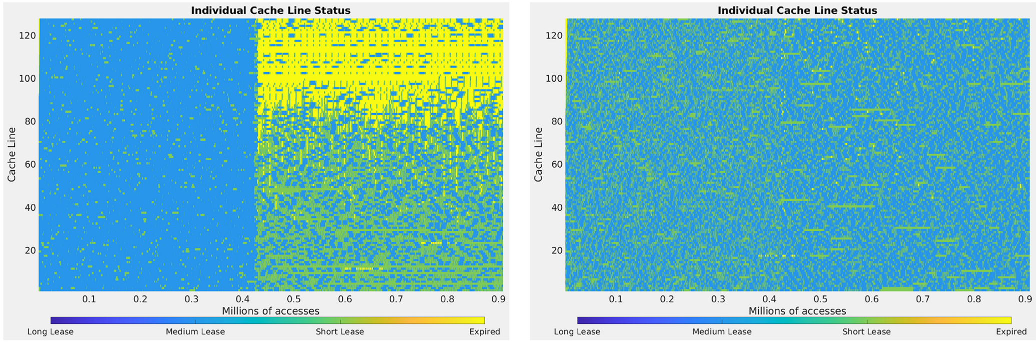


Fig. 6. Cache occupancy spectra for CLAM (left) vs. SHEL (right) for 2 mm with the small dataset. Over-allocation is visible as large chunks of dark blue and under-allocation as chunks of yellow.

It is worth noting that the weakness of PRL, i.e., leases are constrained by the behavior of all intervals, is also a strength in robustness in that PRL never increases cache contention compared to CLAM. This is shown in the comparison where in all but one test, PRL performs the same as or better than CLAM.

Limit Analysis. CARL computes the miss ratios for an ideal cache, which, as proved by Ding et al. [17], is the best for the same average virtual cache size for all lease solutions. No lease optimization can perform better than CARL. Note that CARL bounds may not be tight, i.e., what is included in the potential may not be reachable. What is valuable, however, is that they show what is excluded, which is definitely not realizable. No lease optimization can perform better than CARL.

Compared to PLRU, the potential for cache programming depends on the input size. On our system, the cache size is 8 KB, and the amount of program data is approximately 128 KB, 1 MB, and 25 MB for the three input sizes, respectively. The average potential is 62%, 36%, and 11% reduction over PLRU for the three input sizes. Given these potentials, the average realized by our three techniques is 84% for small by SHEL, 46% for medium by C-SHEL, and 82% for large by C-SHEL.

The CARL bound is not tight, because they require variable cache sizes. The realizable bound lies in the gap between the best actual result and the CARL bound. Among the three input sizes, this gap is narrowest in the large input, so the CARL bound is closest to the actual potential, so is the realized portion of the CARL bound to the realized potential. Hence, we have two additional findings based on the limit analysis:

Finding. *The limit analysis shows that:*

(5) *Lease-based cache programming may improve cache management over LRU by over 60% when data size is 16 times the cache size and may still improve by over 10% when the data size 3,000 times larger.*
(6) *The techniques in this article have realized over 80% of the potential of lease cache programming.*

The limit of cache programming is also bounded by optimal fixed-size caching, i.e., the OPT or MIN method [13, 23]. Optimal caching is automatic but unrealizable, because it requires precise future knowledge. Two earlier studies have shown that ideal lease cache performs as well as or better than OPT for both storage traces [22] and PolyBench programs [17].

Lease Visualization. Prechtl et al. [29] showed that leases give a user the ability to visualize the state of the cache at each moment, and by taking samples and showing them in a sequence, cache dynamics over an execution. We use visualization in Figure 6 to show the effect of lease

optimization on one of our test programs. The graphs plot the *individual cache line status*, which shows the current lease for each cache block. At each moment, the entire cache space of 128 blocks is shown as a column of 128 cells, colored individually to show the current remaining lease in that block. Previous work [29] calls this a cache tenancy spectrum. From the execution start time at the left boundary to the execution end time at the right boundary, we plot the execution as a matrix of colored cells. Yellow means no lease (empty block) and blue means occupied. The darker the blue is, the longer the lease. Thus, over-allocation is visible as large chunks of blue and under-allocation as chunks of yellow.

Figure 6 clearly shows the benefit of phase-aware lease assignment strategies. This program is composed of two top-level nested loops, whose effect can be clearly seen in the cache occupancy spectra. CLAM allocation is blind to this structure; it simply assigns all the most profitable leases until the budget has been met. Unfortunately, a disproportionate number of the most profitable lease assignments lie in the first loop, and fewer of them lie in the second loop. The result is over-allocation during the first loop and under-allocation during the second loop. SHEL, however, is able to optimize each loop individually, resulting in a balanced cache use, with fewer contention misses in the first phase and more hits in the second phase.

Effect on Single-scope Programs. Figure 5 shows the normalized miss count for the 17 single-scope tests. Because these programs have only a single scope, SHEL and C-SHEl lease assignments are identical to those of CLAM. On average for the small dataset, PRL reduction, 54%, is slightly better than CLAM's 52%. For medium and large, the two results are effectively the same, with reductions of 27% and 26%, respectively. This can be explained by behavior variations in single-scope tests. PRL considers them separately using intervals, but CLAM does not. The effect, however, is significant only in small inputs. For the other two inputs, PRL sometimes performs worse than CLAM, likely because optimal leases assigned some intervals are sub-optimal overall.

4.3 Performance of Cache Programming in Optimized Loops

Data locality can be greatly improved by the compiler using the polyhedral abstraction [7, 20]. We consider compiler transformations and lease caching to be complementary solutions for improving cache performance. Lease caching seeks to improve the hit ratio via optimization of the cache replacement policy given a memory access stream. Compiler transformations seek to improve the hit ratio by reordering the memory access stream itself, such that locality is improved. We examine the combined effect of polyhedral loop optimization done by the PLUTO compiler [9] and our method of lease caching. We evaluate all four caching techniques (CLAM, PRL, SHEL, and C-SHEL) on 12 multi-scope benchmarks and CLAM and PRL on 17 single-scope benchmarks, each run with three different data sizes.

Overall Comparison. Comparing the PLRU results (black bar) in Figure 5 and Figure 7, the cache performance can be greatly improved by the PLUTO compiler. Cache programming still improves performance over automatic caching for locality-optimized code. As shown in Figure 7, in 168 out of 246 cases, lease cache matches or performs better than PLRU. In 146/168 tests, the programmable cache improves the cache performance by 25%. Another 14 solutions reduce the cache misses by over 50%, and the best reduction can reach as high as 82%. It is worth noting that PLUTO triples the cache misses in `ludcmp` on PLRU, but such a negative impact does not happen in a programmable cache: Both PLUTO and non-PLUTO codes have the same cache performance in `ludcmp`. For the remaining 78 cases where the lease cache does not perform well, more than 90% of them add less than 50% misses, and none of them makes the cache perform worse than the non-PLUTO performance. In addition, their cache miss ratio is relatively low, and the degradation is smaller in larger data inputs. The geometric mean miss ratios in the programmable cache are 1.00% (0.75% in PLRU) for small data size, 0.84% (0.63% in PLRU) for medium, and 0.34% (0.26% in PLRU) for large.

(a) Multiple scope, SMALL_DATASET

(b) Single scope, SMALL_DATASET

(c) Multiple scope, MEDIUM_DATASET

(d) Single scope, MEDIUM_DATASET

(e) Multiple scope, LARGE_DATASET

(f) Single scope, LARGE_DATASET

Fig. 7. Miss counts for 12 multi-scope benchmarks (left) and 17 single-scope benchmarks (right) as well as their geometric means after optimized by PLUTO compiler. Lower is better. Values are reported for small (top), medium (middle), and large (bottom) inputs.

Table 2 summarizes the average (geo-mean) cache miss reduction by the lease cache under two groups of benchmarks with three data inputs. Negative number means lease cache performs worse than automatic caching (PLRU). For multi-scope tests, except for one anomaly `lu`, the same discovery discussed in the previous section still applies to PLUTO-optimized code: C-SHEL performs the best on small dataset, SHEL on medium dataset, and SHEL and C-SHEL on the large. The same observation also applies to single-scope. Excluding one anomaly `gramschmidt`, the best performance is obtained by PRL on small dataset but later switched to CLAM on medium and large datasets. These two anomalies significantly impact the lease cache performance. For PLUTO results, Table 2 shows the effect without these anomalies in parentheses. Next, we discuss why such anomalies happen in lease cache.

Table 2. Cache Miss Reduction (in Geomean) by Lease Cache in PolyBench

	Input Size	Multi-scope				Single-scope	
		CLAM	PRL	SHEL	C-SHEL	CLAM	PRL
No PLUTO	SMALL	56.9%	61.5%	60.5%	62.9%	43.0%	43.7%
	MEDIUM	10.4%	13.4%	25.3%	22.3%	21.3%	21.0%
	LARGE	7.7%	8.7%	11.4%	11.4%	28.0%	27.9%
PLUTO	SMALL	10.9% (12.1%)	9.1% (14.1%)	7.0% (14.4%)	8.0% (15.4%)	20.8%	16.4%
	MEDIUM	−4.4% (−2.1%)	−4.2% (−2.3%)	−20.9% (8.5%)	−3.0% (8.0%)	5.0%	4.7%
	LARGE	−4.2%	−4.3%	4.5%	4.5%	0.8%	0.4%

Lease Cache Anomalies. There are two anomalies, `lu` from multi-scope programs and `gramschmidt` from single-scope programs. Lease cache has 2.3× and 35× more misses on small and medium dataset in `lu` and 2.5× more on small dataset in `gramschmidt`. PLUTO transforms a loop and increases the number of references in it. This poses two problems for the lease cache. The first is information loss. The effect of sampling is "diluted" in that the RI distribution for each reference may have fewer RIs. This is likely the problem in `gramschmidt`. The anomaly happens only for PRL and not for other methods, because PRL divides a program into five phases. The second problem is lease-table truncation. Our current hardware has 128 entries. If a program has more than 128 references, then the top 128 most profitable leases are loaded, and the remaining references are assigned the default lease (which is 1). This is likely the case of the two `lu` anomalies.

In summary, we have the following discovery on the effect of programmable cache on polyhedral-optimized code:

Finding. *The three programmable caching schemes on PLUTO-optimized code show that:*

(7) *Programs after polyhedral optimization can still benefit from the lease cache optimization, and their best performance can be achieved by considering the scope.*

Interaction between Cache Programming and Compiler Optimization. The PLUTO compiler will transform the original loop structures to remove dependencies to improve parallelization and locality. These transformations include loop distribution (fission) and loop fusion. Such transformations have two impacts on the lease cache: (1) The program scope can be altered by these transformations, making the scope marker we set on the original program less optimal. (2) More array references are inserted to handle boundary conditions when performing loop blocking or tiling. As the number of references increases, lease caching is negatively affected by two problems: information loss and lease-table truncation. The two problems happen on 3 of the 246 cases, 2 of them for the small input size, all by only SHEL and PRL (not CLAM or C-SHEL), and only for PLUTO optimized loops. Other tests are not significantly affected by these problems.

Finally, we observe that lease cache is beneficial for compiler optimized code. Polyhedral optimizations may not always provide perfect locality. In some cases, a compiler cannot eliminate all cache misses because either a program cannot be further optimized, the compiler fails to apply the full optimization possible, or a user fails to configure the compiler properly. Providing an additional layer of optimization can be beneficial for cases where there is still room for improvement in the cache replacement policy when applied to a transformed program.

Our results show improvement in geomean performance in all cases with SHEL and C-SHEL after PLUTO optimization, excluding the anomalous results of a single outlier program (lu). For `deriche` and `ludcmp`, PLUTO makes no locality improvement among all three input sizes. For `ludcmp` on the small size, PLUTO optimization is counter-productive. It more than triples the miss ratio in the PLRU cache. The lease cache has no such problem. It performs the same

with and without PLUTO on all three sizes. PLUTO is slightly worse in the medium size and the same in the large size. This shows that compiler optimization may falter in rare cases, and the programmable cache provides another line of defense, and in this case, largely removes the degradation.

In this work, we show the results of lease cache on benchmarks whose loops are **Static Control Parts (SCoP)** [9], because the reuse interval information required by the lease assignment can be gathered statically. For parts of programs that are not SCoPs, a dynamic lease cache policy, which gathers statistics and assigns leases at runtime, could be applied to more parts beyond those that are amenable to such compiler transformations. However, the development of such a system is beyond the scope of this article, in which we seek to evaluate the performance of our design, which uses static leases.

5 RELATED WORK

Programmable Cache. Prechtl et al. [29] presented the first study of lease-cache prototype and lease-based programming using CLAM and PRL (Section 2.2). Ding et al. [17] extended the study to show formally that the greedy algorithm, called **Compiler Assignment of Reference Leases (CARL)**, is optimal in that no other leases based on the same information can perform better. The optimality is for virtual cache only. This article formulates and solves the problem of a fixed-size cache.

CLAM is the first design and implementation of the lease cache in hardware and with a fixed size [29]. The current work adds hardware support for scope-based leasing. In addition, by adding RISC-V 32F (floating point) instruction set extensions, our prototype runs all 30 programs of the PolyBench suite, while previous work ran just 7 of the 30.

Prechtl et al. [29] tested for only CLAM and PRL. Ding et al. [17] included all PolyBench tests but only for the virtual cache. The past work did not consider the over- and under-allocation problem of lease programming, except in PRL. Based on intervals, PRL forgoes a lease in all intervals if it causes over-allocation in any interval (Section 2.8). SHEL and C-SHEL in this article are based on scopes. SHEL performs better than PRL on medium and large inputs (Finding 4 in Section 4).

The greedy algorithm was first used by Li et al. [22] to assign **Optimal Steady-state Lease (OSL)** for storage caches. Like CARL, OSL targeted the virtual cache. Unlike CARL, which assigned reference leases, OSL assigned a lease for each data page. The CARL optimality for reference leases implies OSL optimality for page-based leases [17]. Since the number of references in a program can be many orders of magnitude less than the size of data, reference leases are more practical for hardware caches.

A hardware technique, **Protecting Distance-based Policy (PDP)** "prevents replacing a cache line until a certain number of accesses to its cache set" [18]. Its hardware support is similar to our design, including the RI sampler and cache tags. Unlike our work, PDP assigns the protecting distance at runtime, so it stores an RI histogram in hardware. The technique is shown to improve the performance of single-core caches and the throughput and fairness of multi-core caches. Using a protecting distance is the same as assigning the same lease to every data access. Chen et al. [11] called such a policy the **uniform lease (UL)** and showed the theoretical conditions when UL is equivalent to LRU and the common cases in practice when one is better than the other. In lease programming, a program may assign a different lease for each reference. Lease programming performs better than LRU (Finding 1).

Lease programming uses dual leases, supported by our hardware prototype (Section 3). In cache management, Talus [6] and SLIDE [33] partitioned the access stream to have the effect of dividing the working set for LRU and other cache policies. The dual lease creates a similar effect through cache programming, which localizes their use for a single reference rather than for all.

Working-set Caching. As cache allocation, lease programming is related to the working-set theory. Denning defined the working set as the data touched in the last τ accesses [13–15]. In virtual memory management, physical memory is allocated based on the dynamic working set. In a recent survey paper, Denning summarized key properties including the optimality of the working-set policy [16]. The optimality has two conditions: the first is steady-state behavior within each phase; the second is variable size cache memory.

Similar to the working-set policy, PRL divided an execution into fixed length intervals but targeted a fixed-size cache. PRL intervals may be too coarse, so it misses phase variation inside a PRL interval, or too fine, so it misses the effect of cross-scope RIs. This article presents scope-based solutions and solves the programming problem using the structure of loops. Each scope is a loop nest that has a uniform data usage pattern. In addition, C-SHEL accommodates cross-loop reuses with cross-scope leases. PRL requires a profiling step, while SHEL and C-SHEL can use profiling (as in this study) or compiler analysis, for example, static sampling [12], which has been demonstrated for (virtual-cache) lease programming by Ding et al. [17].

Program Optimization. Loop-nest optimization has long been developed and is critical in improving the cache performance [3, 4, 36, 37]. Recent work includes polyhedral optimization [27] and sparse optimization [24] for computation and for data [32]. Program optimization targets LRU caches. This has led to a growing list of compiler and runtime techniques to model the LRU cache performance [5, 8, 10, 12, 21, 34]. While it is beyond the scope of this article, programmable cache has the potential to enable greater optimization than what is possible with automatic caches, as well as being a new target for static analysis.

6 CONCLUSION

We have designed, implemented, and tested a lease cache architecture prototype with support for RISC-V with floating point instructions. We have formulated the problem of optimal cache programming using leases, presented two novel algorithms, SHEL and C-SHEL. Furthermore, we have compared four solutions on the full suite of PolyBench programs with three input sizes. The results show that (1) cache programming is overwhelmingly better than automatic PLRU caching; (2) this improvement potential is significant even when program data size is far larger than the cache size; (3) the best strategy is SHEL, which ignores inter-scope reuses; (4) SHEL realizes most of the potential of cache programming; and (5) after polyhedral optimization, apart from a few outliers, programs still benefit from the lease cache, and their best performance is achieved by considering the scope as in SHEL or C-SHEL.

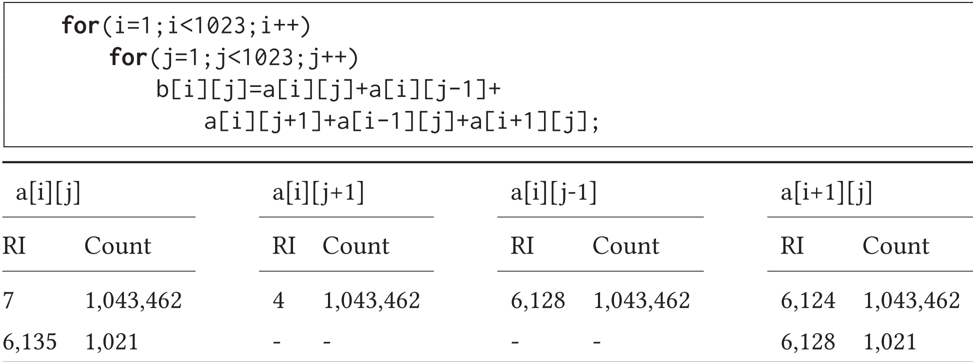
APPENDICES

A REUSE INTERVAL SAMPLER

The objective of the sampler is to profile a program and provide the lease compiler with the RI distribution necessary to generate leases. This allows the lease compiler to operate independently of **instruction set architectures (ISAs)**. The lease cache hardware is designed to support lease policy management for all eviction possibilities:

- *Zero vacancy*: no cache line has an expired lease.
- *Single vacancy*: exactly one cache line has expired.
- *Multiple vacancies*: more than one cache line has expired.

For a single vacancy, the eviction selection is obvious. The zero vacancy case requires the application of an auxiliary policy, because there is no line eligible for eviction according to the current lease values. Multiple vacancies are handled by prioritizing the eviction of low index cache lines.



```
for(i=1;i<1023;i++)
   for(j=1;j<1023;j++)
      b[i][j]=a[i][j]+a[i][j-1]+
         a[i][j+1]+a[i-1][j]+a[i+1][j];
```

a[i][j]		a[i][j+1]		a[i][j-1]		a[i+1][j]	
RI	Count	RI	Count	RI	Count	RI	Count
7	1,043,462	4	1,043,462	6,128	1,043,462	6,124	1,043,462
6,135	1,021	-	-	-	-	6,128	1,021

Fig. A.1. Reuse interval (RI) histograms for four of the references of the five-point stencil program. Each row represents a different reuse interval that is observed for each reference. References with no reuses (b[i][j] and a[i-1][j]) are omitted.

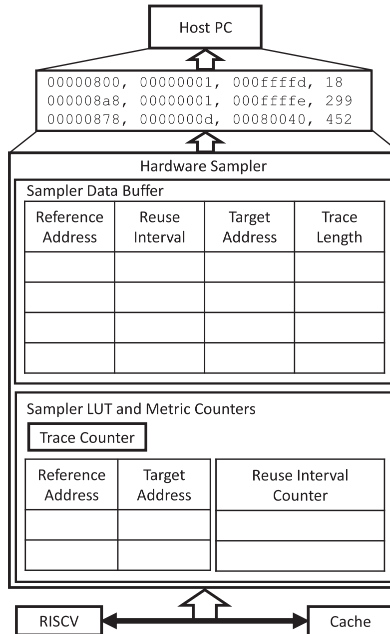


Fig. A.2. Hardware reuse interval sampler system overview. The text file snapshot shows the sampler output.

The current lease cache hardware handles all eviction cases, stochastically assigns dual leases to accesses, and monitors cache utilization/vacancy.

Consider the five-point stencil; from its inspection there are six memory references. The resulting reuse interval distribution of the program is straightforward, as given in Figure A.1. When assembled and linked, however, additional references are present in the form of stack manipulations and similar operations. The manner in which the binary is compiled has a direct impact on how the leases are to be practically applied. This is not limited to compiler nuance. Take, for example, the RISC-V ISA [35], which defines 32 general purpose registers. When compiled for the embedded variant of the ISA only 16 registers are used. This results in increased memory references to data that would otherwise be stored in the register file. The clairvoyance breadth of

the compiler required is a practical issue when considering lease policies as a solution towards a programmable cache.

The hybrid solution to this issue is a front-end lease cache hardware integrated with the lease compiler as the back-end. The hardware generates the reuse interval distribution for the compiler, which then generates leases based on it. In this way, the lease compiler requires no ISA/compile knowledge and can be applied to any system, given that this reuse interval sampler hardware can be integrated.

Operation of the Hardware Reuse Interval Sampler. The sampler is essentially a communication snooper. It is integrated within the request bus between the core and next level memory, which in this case is the internal cache. The sampler monitors the memory accesses between the RISC-V core and memory, periodically sampling bus transactions, and generates the resulting reuse intervals. The sampling period is an application heuristic—the objective is to gather reuse intervals for all memory references within a program. However, this is not a necessary condition. References without collected reuse intervals are assumed sparse and contribute minimally to program execution. This subset of references is instead associated with a default lease. A 64-entry hardware lookup table caches two access fields—the target address (search field) and address of the reference invoking the access. An additional counter is associated with each entry of the table to record the running reuse interval of the reference (incremented at every access). The table is populated at variable intervals using a nine-bit **linear feedback shift register (LFSR)**. The LFSR generates a pseudo-random sequence that seeds a sampling counter, which decrements at every access. When the counter expires, a new sample is started by adding the current access fields to the table. The sampling counter is re-seeded with the next number generated by the LFSR. Using a nine-bit LFSR results in an average sampling rate of 1 sample per 256 accesses.

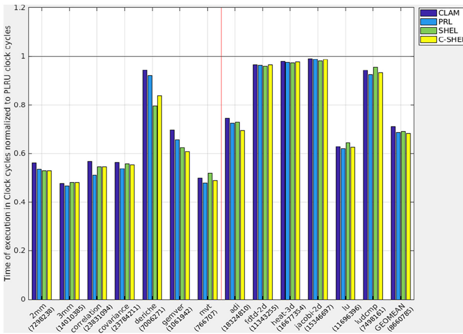
A block reuse is indicated by an access target address matching an entry of the table. At this point, the sample is complete, and the reuse interval for the memory reference should be recorded. The entry of the table that resulted in the match is evicted and its fields are stored into a sample buffer, along with the current time (current trace length). Eviction also forcibly occurs if all entries of the table are active when the sampling counter elapses. To allocate space for the new sample, the oldest entry of the table is evicted. Because this entry was not evicted due to a reuse, it is written to the buffer with a negative RI to flag it as a non-reuse for the lease assignment algorithm.

Reuse interval sampler parameter selection is heuristic and depends on the program being examined. Furthermore, there is a direct relationship between population and eviction rates. Increased sampling frequency results in more active table entries, bringing the table to full capacity more quickly. The rate at which the table becomes bottle-necked limits the magnitude of reuse intervals that can be recorded by the table. As the sample rate increases, capacity evictions become more frequent—removing entries with the largest active running intervals, and so long RIs have a smaller chance of being recorded.

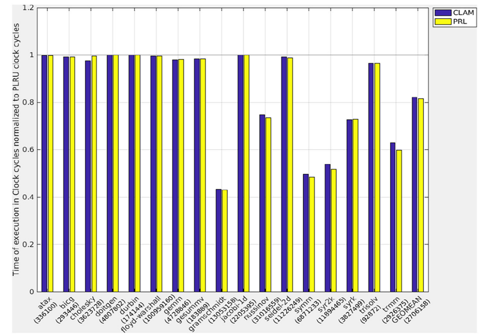
Hardware Sampler Extension to Support Scoped Lease Policies. The reuse interval sampler proposed in Prechtl et al. [29] is modified to record not only the instruction address of the memory reference, but also to record the specific phase of that access (Figure 2). This modification is what enables the SHEL and C-SHEL algorithms to determine which references belong to which scope during lease generation.

B RUNTIME RESULTS

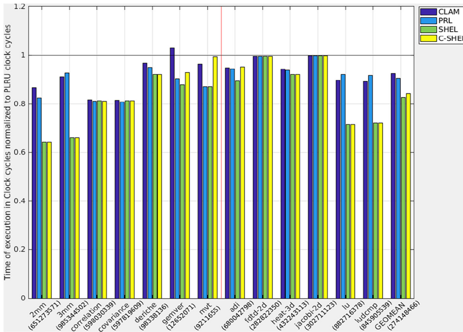
Figure B.1 shows the runtime of each benchmark in terms of clock cycles. The penalty of a cache miss in our system is 16 cycles. SHEL and C-SHEL require CPU stall cycles to repopulate the lease



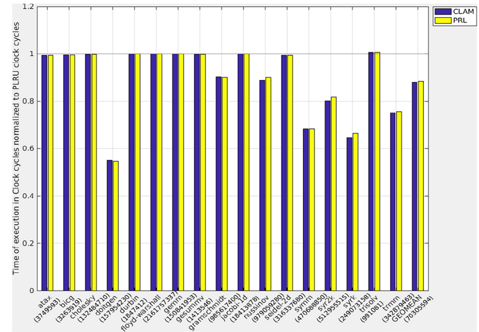
(a) Multiple scope, SMALL_DATASET



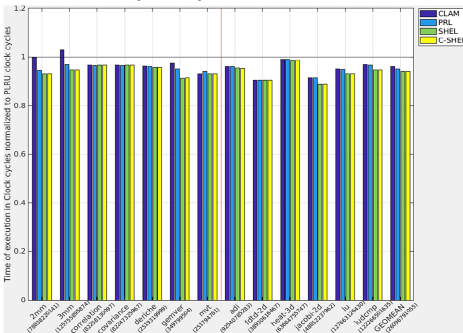
(b) Single scope, SMALL_DATASET



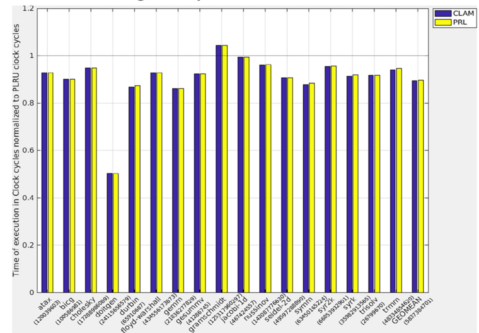
(c) Multiple scope, MEDIUM_DATASET



(d) Single scope, MEDIUM_DATASET



(e) Multiple scope, LARGE_DATASET



(f) Single scope, LARGE_DATASET

Fig. B.1. Clock cycles for execution of all benchmarks using each cache leasing technique, normalized to PLRU performance.

lookup table on phase change. This effect is greatest on cyclic benchmarks with small data sizes, e.g., `lu` and `ludcmp`. However, for medium and large data sizes, this effect is minimal. At each dataset size, SHEL and C-SHEL still outperform policies with no CPU stall.

ACKNOWLEDGMENTS

The authors would like to thank Dong Chen, Jonathan Waxman, Boyang Wang, Adam Bobok, Michael Scott, other members of the systems group, and the anonymous referees for their valuable comments and helpful suggestions.

REFERENCES

[1] Miguel Á. Abella-González, Pedro Carollo-Fernández, Louis-Noël Pouchet, Fabrice Rastello, and Gabriel Rodríguez. 2021. PolyBench/Python: Benchmarking Python environments with polyhedral optimizations. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. ACM, 59–70. DOI: https://doi.org/10.1145/3446804.3446842

[2] Aravind Acharya and Uday Bondhugula. 2015. PLUTO+: Near-complete modeling of affine transformations for parallelism and locality. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 54–64. DOI: https://doi.org/10.1145/2688500.2688512

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.

[4] Randy Allen and Ken Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers.

[5] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, and P. Sadayappan. 2018. Analytical modeling of cache behavior for affine programs. *PACMPL* 2, POPL (2018), 32:1–32:26. DOI: https://doi.org/10.1145/3158120

[6] Nathan Beckmann and Daniel Sanchez. 2015. Talus: A simple way to remove cliffs in cache performance. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 64–75. DOI: https://doi.org/10.1109/HPCA.2015.7056022

[7] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The polyhedral model is more widely applicable than you think. In *Proceedings of the International Conference on Compiler Construction*, Vol. 6011. Springer, 283–303.

[8] Kristof Beyls and Erik H. D'Hollander. 2005. Generating cache hints for improved program efficiency. *J. Syst. Archit.* 51, 4 (2005), 223–250.

[9] 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.

[10] Calin Cascaval and David A. Padua. 2003. Estimating cache misses and locality using stack distances. In *Proceedings of the International Conference on Supercomputing*. 150–159.

[11] Dong Chen, Chen Ding, Fangzhou Liu, Benjamin Reber, Wesley Smith, and Pengcheng Li. 2021. Uniform lease vs. LRU cache: Analysis and evaluation. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*. ACM, 15–27.

[12] 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. DOI: https://doi.org/10.1145/3192366.3192402

[13] Edward G. Coffman Jr. and Peter J. Denning. 1973. *Operating Systems Theory*. Prentice-Hall.

[14] Peter J. Denning. 1968. The working set model for program behaviour. *Commun. ACM* 11, 5 (1968), 323–333.

[15] Peter J. Denning. 1980. Working sets past and present. *IEEE Trans. Softw. Eng.* SE-6, 1 (Jan. 1980).

[16] Peter J. Denning. 2021. Working set analytics. *ACM Comput. Surv.* 53, 6 (2021), 113:1–113:36. DOI: https://doi.org/10.1145/3399709

[17] Chen Ding, Dong Chen, Fangzhou Liu, Benjamin Reber, and Wesley Smith. 2022. CARL: Compiler Assigned Reference Leasing. *ACM Trans. Archit. Code Optim.* 19, 1 (2022), 15:1–15:28.

[18] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving cache management policies using dynamic reuse distances. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 389–400. DOI: https://doi.org/10.1109/MICRO.2012.43

[19] Stefan Ganser, Armin Größlinger, Norbert Siegmund, Sven Apel, and Christian Lengauer. 2017. Iterative schedule optimization for parallelization in the polyhedron model. *ACM Trans. Archit. Code Optim.* 14, 3 (2017), 23:1–23:26. DOI: https://doi.org/10.1145/3109482

[20] Martin Griebl, Christian Lengauer, and Sabine Wetzel. 1998. Code generation in the polytope model. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 106–111. DOI: https://doi.org/10.1109/PACT.1998.727179

[21] Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. 2019. A fast analytical model of fully associative caches. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 816–829. DOI: https://doi.org/10.1145/3314221.3314606

[22] Pengcheng Li, Colin Pronovost, William Wilson, Benjamin Tait, Jie Zhou, Chen Ding, and John Criswell. 2019. Beating OPT with statistical clairvoyance and variable size caching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 243–256. DOI: https://doi.org/10.1145/3297858.3304067

[23] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J.* 9, 2 (1970), 78–117.

[24] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C. Davis, Mary W. Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. 2019. Sparse computation data dependence simplification for efficient compiler-generated inspectors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* 594–609.

[25] Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P. Sadayappan, and Fabrice Rastello. 2020. Automated derivation of parametric data movement lower bounds for affine programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation.* ACM, 808–822. DOI: https://doi.org/10.1145/3385412.3385989

[26] Louis-Noël Pouchet. 2018. PolyBench/C 4.0. Retrieved from http://polybench.sourceforge.net.

[27] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop transformations: Convexity, pruning and optimization. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 549–562.

[28] Ian Prechtl, Chen Ding, and Dorin Patru. 2020. Design and Evaluation of a fixed-size Programmable Working-set Cache on FPGAs. Retrieved from https://dx.doi.org/10.13140/RG.2.2.24423.60320

[29] Ian Prechtl, Ben Reber, Chen Ding, Dorin Patru, and Dong Chen. 2020. CLAM: Compiler lease of cache memory. In *Proceedings of the International Symposium on Memory Systems.* ACM, 281–296.

[30] Michael L. Scott. 2009. *Programming Language Pragmatics* (3rd ed.). Morgan Kaufmann Publishers.

[31] Kimming So and Rudolph N. Rechtschaffen. 1988. Cache operations by MRU change. *IEEE Trans. Comput.* 37, 6 (1988), 700–709. DOI: https://doi.org/10.1109/12.2208

[32] Anand Venkat, Mary W. Hall, and Michelle Strout. 2015. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 521–532. DOI: https://doi.org/10.1145/2737924.2738003

[33] Carl A. Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache modeling and optimization using miniature simulations. In *Proceedings of the USENIX Annual Technical Conference.* 487–498. Retrieved from https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger.

[34] Qingsen Wang, Xu Liu, and Milind Chabbi. 2019. Featherlight reuse-distance measurement. In *Proceedings of the International Symposium on High-Performance Computer Architecture.* IEEE, 440–453. DOI: https://doi.org/10.1109/HPCA.2019.00056

[35] Andrew Waterman and Krste Asanović. 2017. *The RISC-V Instruction Set Manual: Volume I: User-level ISA.* Version 2.2. https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[36] M. J. Wolfe. 1996. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Redwood City, CA.

[37] Jingling Xue. 2000. *Loop Tiling for Parallelism.* Kluwer International Series in Engineering and Computer Science, Vol. 575. Kluwer.

Received 20 September 2022; revised 2 May 2023; accepted 5 May 2023