CLAM: Compiler Lease of Cache Memory

Ian Prechtl
Department of Electrical Engineering,
Rochester Institute of Technology
irp2474@rit.edu

Ben Reber
Department of Computer Science,
University of Rochester
breber@cs.rochester.edu

Chen Ding
Department of Computer Science,
University of Rochester
cding@cs.rochester.edu

Dorin Patru
Department of Electrical Engineering,
Rochester Institute of Technology
dxpeee@rit.edu

Dong Chen
Department of Computer Science,
NUDT
jameschennerd@gmail.com

ABSTRACT

Traditional caching is transparent to software but cannot utilize program information directly. With Moore's Law ending and general-purpose processor speed plateauing, there is increasing importance and interest in specialization including the interaction between the software and the cache.

This paper presents Compiler Lease of cAche Memory (CLAM) which augments the interface between software and hardware and lets a compiler control cache management. The new software control enables optimization beyond what is possible in traditional memory system designs. CLAM has been implemented on a CycloneV-GT FPGA card with a RISC-V processor and the new hardware cache, and the evaluation has shown performance improvements over existing techniques in all of the 7 programs tested from the Polybench suite.

1 INTRODUCTION

Memory technology is increasing in complexity, with the adoption of new material such as Intel Optane and new organization and interconnect such as high bandwidth memory (HBM).

Owing to its inherent costs in latency and energy, data movement has become the main target for memory optimization. The primary mitigation is caching. Its popularity comes from its autonomy — caching is automatic, which simplifies memory programming. In the best case, a program obtains good performance with no programming effort. In the common case, the caching policy provides robust performance, i.e. through the use of Least Recently Used (LRU) replacement policy and its variants.

Being automatic, however, caching does not allow direct program control. Hence in the worst case, the cache is blind, and the program impotent.

Recently, a new design called the *lease cache* (Section 2.1) makes direct program control possible. Theoretically, program control enables optimal cache management not possible with automatic caching. However, to realize this potential, program control must solve the following four problems to be practical:

- On the software side, programming should (1) be automatic and (2) do no worse than automatic caching.
- On the memory side, the cache management should handle (3) a fixed-size space and should be (4) as efficient as automatic caching.

This paper presents Compiler Lease of cAche Memory (CLAM). It uses a compiler to assign a lease for each reference in a program. When the access pattern is known at compile time, it assigns different leases to favor the data that has the most reuse. When the access pattern is unknown, it assigns the same lease to each access. The dynamic lease values provide an opportunity to visualize cache management. This paper uses *Cache Tenancy Spectrum* to view cache management down to each cache block and covering the entire length of execution.

The main contributions of the paper are:

- Lease programming by a compiler, including CARL, which optimizes for a variable size cache (Section 2.2),

and PRL, which optimizes for a fixed size cache (Section 2.4).
- Dual leases (Section 2.3) and uniform leases by FUL (Section 2.5). The former supports the optimization in both CARL and PRL, and the latter provides performance safety when optimization is infeasible.
- Lease management by hardware, including the actual design and implementation on a CycloneV-GT FPGA with a RISC-V processor and the lease cache in hardware. (Section 3)
- Experiments validating the performance optimization and safety (Section 4) and visualizing the dynamics of cache management (Section 5).

Lease caching is a new cache control paradigm, designed to replace automatic policies like LRU. It is possible to implement a lease cache at any level of the traditional memory hierarchy. Our goal is to explore the potential benefits over LRU at the policy level, rather than to analyze the different properties of a lease cache implementation in L1 vs. L3 cache, or in a fully associative vs. set associative cache. Such questions are outside the scope of this paper and may be the subject of future work. Our hardware implementation most closely matches a fully associative L1 lease cache implementation.

The paper focuses on loop-based scientific kernels in software and FPGA implementation in hardware. While a similar approach may be used for general-purpose applications and platforms, both programming and system design will be more complex and are beyond the scope of this paper.

2 CLAM ALGORITHMS

2.1 Background: Variable-size Lease Cache

Recently, Li et al. [17] proposed a new type of cache called the *lease cache*, where a program specifies a length of time called the lease at each data access. A lease from a program instructs the cache to store the data block in the cache for the duration of the lease. Cache management is prescriptive and in this fashion analogous to memory allocation — a lease is the prescribed lifetime of a data block in the cache. Lease cache enables a program to allocate the cache space by controlling the lease.

If the access is a miss, a new data block is loaded into the cache and given the lease. If the access is a hit, the lease of the data block is renewed. In either case, a lease is given at *every* data access. The lease is measured by the number of accesses rather than the physical time. A lease of 1000 means that the lease cache keeps the data block until 1000 accesses later. The lease is renewed if the data block is accessed before the end of the lease; otherwise, the block is evicted from the cache. At each access, the lease specifies the maximal lifetime of the data block in cache in the absence of a reuse.

It helps to draw an analogy with an automatic water faucet. When a faucet detects a user's hand, it discharges water for a period of time. This time can be viewed as a lease. If a hand is detected again, the lease is renewed. If not, the lease eventually expires and the water valve is closed. If a lease is too short, water stops while a user is still washing, but if it is too long, it wastes water. By controlling the lease, a program controls the cache allocation.

The lease cache developed by Li et al. [17] has two problems preventing a practical use. First, it assigns the distinct lease for each data (page), which is impractical at program level. Second, the size of the cache can temporarily grow or shrink, which is not suitable for a real hardware cache.

2.2 The CARL Algorithm

Compiler Assigned Reference Leasing (CARL) is a static solution to variable lease assignment, meaning it assigns one lease per memory *reference*, as opposed to dynamic solutions which assign one lease per memory *access*. A reference is an expression in code which refers to data. Consider the following five-point stencil program:

```
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];
```

There are six memory references in this example, and each reference invokes multiple memory accesses; one per iteration of the inner loop. Note that each reference is not limited to a singe piece of memory; a reference in a loop touches memory on multiple data blocks.[1] We call the set of all accesses invoked by a particular reference its *reference access group*.

We consider the *reuse intervals* (RI) of memory accesses. The reuse interval of an access is defined as the length of time before that memory is reused. By traversing a memory trace, we may generate the histogram of all reuse intervals for a given reference access group. Table 1 shows the set of RI histograms for the five-point stencil program. Notice that reference $a[i][j]$ has multiple RIs. Most of the time, the data in $a[i][j]$ will be reused in the next iteration of the inner loop for an RI of 7. However, for the last iteration of the inner loop where $j = 1022$, the data is not reused until another full iteration of the outer loop is complete, for an RI of 6135.

CARL assigns a lease to each reference. This means that each time a memory access is cached, it receives the lease of the reference by which it was invoked. All accesses within a single reference access group receive the same lease. Each lease has an associated *profit* and *cost*. Profit is defined as the number of cache hits granted by a lease assignment. Unit

[1]For the sake of simplicity, suppose that cache block granularity is equal to the size of one array element in this example.

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

Table 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.

cost is defined as the occupancy of one cache block for one unit of logical time.

The cost and profit of a lease $l \in \mathbb{N}$ can be determined using the RI histogram for that reference. Suppose for simplicity that RI histograms are represented as dense vectors, whose index refers to the reuse interval and whose value refers to number of accesses with that RI. The profit is simply the number of accesses in the histogram whose RI is less than or equal to l, so for a given lease l and RI vector H with maximum index RI_{max}, the profit is given by:

$$\text{Profit}(l, H) = \sum_{i=0}^{l} H[i] \tag{1}$$

For the cost, there are two terms to consider. Each access whose reuse interval $ri \in \mathbb{N}$ is less than l will occupy cache for ri units of time before it is refreshed, and each access whose ri is greater than l will occupy cache for l units of time. Thus the total cost is the sum of these two values over the full histogram. The cost is given by:

$$\text{Cost}(l, H) = \sum_{i=0}^{l-1} i * H[i] + \sum_{i=l}^{RI_{max}} l * H[i] \tag{2}$$

Note that under this definition, cost is not directly proportional to lease length. In the five point stencil program, reference $a[i][j]$ has two candidate leases, 7 and 6135. The cost of the short lease is about seven million allocation units, and the cost of the long lease is about fourteen million units.

Using equations 1 and 2, we may define the change in *profit per unit cost* (PPUC) when increasing a lease from l to l' as:

$$\triangle\text{PPUC}(l, l', H) = \frac{\text{Profit}(l', H) - \text{Profit}(l, H)}{\text{Cost}(l', H) - \text{Cost}(l, H)} \tag{3}$$

The details of CARL are presented in algorithm 1. CARL assigns leases in a greedy manner. It first initializes a lease of zero for each reference, in line 2. Then, CARL iteratively increases the leases to higher values in their RI histogram. At each step it selects the reference-lease pair which maximizes PPUC, as seen in line 5.

The CARL algorithm continues increasing leases with maximal PPUC until one of the following conditions has been met:

(1) All references have been assigned the maximum possible lease from their reuse interval histogram, as seen in line 7.
(2) The total cost of the lease assignments has reached some target value, the condition in line 4.

It is helpful to think of the target cost as the overall allocation budget. The budget is selected such that the average cache occupancy for the duration of program execution is equal to some target value. For example, given a cache size of 128 blocks and a trace length of 1,000,000 memory events, the target cost will be 128,000,000 allocation units, since this will result in an average cache size of 128. The target cache size is an input of the CARL algorithm, and thus it must be known at compile time.

Because generating and storing a full memory trace requires a prohibitively large amount of data, we instead use sampled memory traces as detailed in section 3.2. We assume this sampled trace is representative, so the only change to the algorithm is to scale the cost function up by the sampling rate.

2.3 Dual-lease Assignment

CARL produces leases which are optimal assuming that each reference is allowed only one lease for the duration of program execution. This is a coarse lease assignment strategy. We introduce dual-leases, which allow for more fine grained control of reference leases.

In the case where a new lease assignment would cause the allocation budget to be exceeded in CARL, a dual lease is assigned. A dual lease is a pair of leases for a reference, one short and one long, and a probability value p. Each memory access invoked by that reference is assigned its long lease with probability p, and its short lease with probability $1 - p$.

Algorithm 1: CARL main loop.

```
Input  : The number of references $R$ and reuse
         interval histograms $H_{1...R}[1...RI_{max}]$
Input  : Allocation budget $B$
Output : Reference leases $L[1...R]$
1  Function Main():
2      $L[1...R] \leftarrow 0$;
3      $totalAlloc \leftarrow 0$;
4      while $totalAlloc < B$ do
5          $(ref, l_{new}) \leftarrow \underset{r\in 1...R,\ l\in L[r]...RI_{max}}{\operatorname{argmax}} \{\triangle PPUC(L[r], l, H_r)\}$;
6          $l_{old} \leftarrow L[ref]$;
7          if $l_{new} = l_{old}$ then
8              /* No more lease to assign      */
9              break;
10         else
11             $L[ref] = l_{new}$;
12             $totalAlloc += Cost(l_{new}, H_{ref}) - Cost(l_{old}, H_{ref})$;
13         end if
14     end while
15 End
```

A dual lease percent is assigned as follows: Suppose a reference has a lease l last assigned by CARL, and the cost of increasing its lease to l' exceeds the remaining budget. This reference will be given a dual lease (l', l, p) where p is given by:

$$p = \frac{\text{remaining budget}}{\text{Cost}(l', H) - \text{Cost}(l, H)} \tag{4}$$

For example, if there is a remaining budget of 100,000 allocation units and the cost of increasing a lease is 200,000 units, it is assigned the long lease with a 50% probability. At this point, the allocation budget has been exactly met, and so CARL terminates.

2.4 Phased Reference Leasing

CARL is designed to perform optimally on a variable-sized cache. The target allocation cost is a global average value, so there is no guarantee that cache occupancy is evenly distributed throughout program execution. It is possible to have some time periods where cache is over-allocated, and other times where it is under-allocated. This is acceptable with a variable-sized cache, since more cache space can be allocated in oversaturated time periods. In a fixed-size cache, over-allocated time periods lead to the eviction of active leases, which can harm performance.

In order to mitigate overallocation, we introduce Phased Reference Leasing (PRL). PRL is an extension of CARL which seeks to reduce overallocation of cache by partitioning the trace into equal length *phases*, and keeping track of an allocation budget for each phase. The allocation budget for each phase is equal to the global allocation budget for the program divided by the number of phases.

The details of PRL are presented in algorithm 2. Leases are assigned in the same order as CARL, based on the global maximum profit-per-unit-cost, as shown in line 5. However, separate RI histograms and allocation budgets are tracked for each phase in line 19. Whenever a lease is increased, the allocation cost in each phase is increased based on the RI histogram for the reference in that particular phase, as shown in line 20. Leases are continually increased until the budget for a single phase has been met using a dual lease, as seen in lines 11 - 13. This differs from CARL's dual-lease assignment in Eq. 4. CARL terminates after assigning a dual lease, since its budget is met. However, in PRL it is possible to continue increasing leases after the assignment of a dual-lease, since the budget has only been met for one phase. PRL will only accept a lease increase if the cost of the assignment is zero in the saturated phase. Because of this, PRL has only one condition for termination; once all maximal leases have been checked in line 7.

2.5 Uniform Leases

Not all programs are amenable to optimization. A compiler may not know the RI distribution of a reference, either because it is too difficult to analyze, or because it depends on the input and may change with execution. For such references, a compiler assigns a default lease. We call this a *Fixed-size Uniform Lease (FUL)*. FUL can be assigned by the compiler without any knowledge about program access patterns.

Essentially, FUL provides a working-set cache. The default lease corresponds to the working-set parameter defined by Denning [9] (more on this in Sections 4 and 6). The FUL cache is not a single policy. Every lease and lease-based eviction method represents a caching policy. Through the default lease, FUL provides tens of thousands of caching policies for a program to choose from.

The FUL cache is *performance safe* in that the program control can always match the performance of the fully-associative LRU cache for any application. FUL is a special lease assignment condition of lease cache and can be implemented in the same hardware as CARL or PRL, making it a robust alternative to variable lease assignment.

[2]This value is only considered among phases where the cost of the assignment is present, so the denominator is always nonzero.

Algorithm 2: PRL main loop.

```
Input  : The number of references $R$
Input  : Number of phases $P$
Input  : Global reuse interval histograms
         $H_{1...R}[1...RI_{max}]$
Input  : Per-Phase RI histograms
         $H_{1...P,1...R}[1...RI_{max}]$
Input  : Global Allocation Budget $B$
Output : Reference leases $L[1...R]$
Function Main():
    $L[1...R] \leftarrow 0$;
    $alloc[1...P] \leftarrow 0$;
    while $True$ do
        $(ref, l_{new}) \leftarrow \underset{r \in 1..R,\ l \in L[r]...RI_{max}}{\text{argmax}} \{\triangle PPUC(L[r], l, H_r)\}$ ;
        $l_{old} \leftarrow L[ref]$;
        if $l_{old} = l_{new}$ then
            $break$;
        else
            /* Equation 4 */
            $dual\_ratio \leftarrow \min_{p \in 1...P} \{ \frac{\frac{B}{P} - alloc[p]}{Cost(l_{new}, H_{p,ref}) - Cost(l_{old}, H_{p,ref})} 2 \}$ ;
            $dual\_ratio \leftarrow min\{dual\_ratio, 1\}$;
            if $0 < dual\_ratio < 1$ then
                $L[ref] \leftarrow (l_{new}, l_{old}, dual\_ratio)$
            end if
            if $dual\_ratio = 1$ then
                $L[ref] = l_{new}$;
            end if
            for $p$ in $1...P$ do
                $\triangle cost = (Cost(l_{new}, H_{p,ref}) - Cost(l_{old}, H_{p,ref})) * dual\_ratio$;
                $alloc[p] += \triangle cost$;
            end for
        end if
    end while
End
```

3 LEASE CACHE HARDWARE IMPLEMENTATION

Hardware is able to support CLAM from compilation to implementation. In this section we present two solutions: a hardware based reuse interval sampler and a supporting lease cache architecture. The objective of the sampler is to profile a program and provide CLAM with the RI distribution necessary to generate leases. This allows CLAM to operate independent of instruction set architectures (ISAs). The cache hardware architecture is designed to support lease management for all CLAM 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 an auxiliary policy as there is no line eligible for eviction according to CLAM. Multiple vacancies are handled by prioritizing the eviction of low index cache lines. The proposed cache architecture handles all eviction cases, stochastically assigns dual leases to accesses, and monitors cache utilization/vacancy.

3.1 Lease Cache Architecture

Lease Assignment. Lease policy hardware is implemented in parallel with existing cache infrastructure (Figure 1). To support this, the request bus to the cache is augmented with the address of the reference invoking the access. Both target and reference addresses are propagated through translation tables - yielding cache location and lease policy information to the controller concurrently. The collection of tables that drive lease policy logic is the lease lookup table (LLT). The LLT contains four 128 entry tables, enough to sufficiently store a complete lease set for an appropriately sized program (Table 2). At every access the LLT resolves to 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.
(3) Secondary Lease [n bit]- lease associated with the lower probability assignment (for non-dual leases this is zero).
(4) Lease Probability [9 bit] - the probability of the primary lease being used in assignment (for non-dual leases this is 100%). Probability bit width is the digitized resolution of CARL's dual lease assignment.

The primary and secondary leases are multiplexed by probability evaluation. An LFSR generates a random number that is compared against the probability bus of the LLT. If the random value is greater than that of the LLT the secondary lease is multiplexed through, otherwise the primary lease passes. A secondary multiplexer then drives the final assignment. If the access results in an LLT hit the current lease assignment is validated and passes. However, if the reference is not found in the table a default lease assignment, stored as a software configurable register, is instead multiplexed. In this way lease selection is not transparent to the policy controller and strictly abides by CARL/PRL. References without an associated lease assignment are assumed to have no near

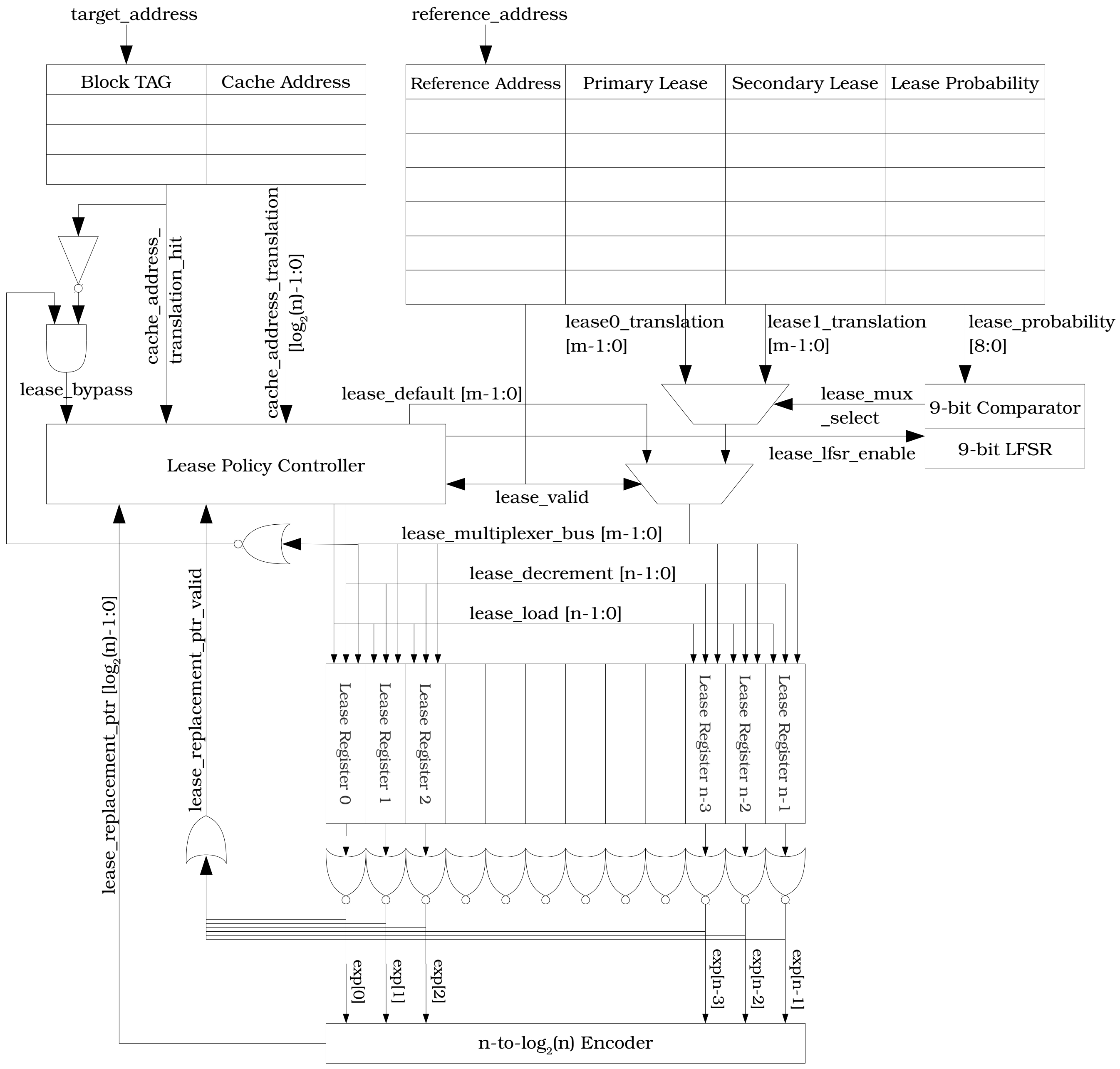

Figure 1: Lease cache policy specific hardware architecture for a cache of *n* blocks and lease register size of *m* bits. Note that NOR gate array is used in an input reduction configuration.

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 they 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 bused to the cache controller to be used for eviction. The pointer is validated by a reduction OR (inequality with zero comparison), of all expired bits. If at eviction the pointer is invalid (no lease is expired) replacement follows the auxiliary policy, random replacement.

Lease Policy Application. At every cache access all non-expired lease registers are decremented. If the access resolves as a cache hit (not LLT hit) the lease register at the translated address is load enabled, regardless of lease assignment. The miss policy, however, depends on whether the item is cacheable. If the lease assignment of the missed access is non-zero then the item is allocated into cache, either to an expired line or by auxiliary policy. A lease assignment of zero to an access miss elicits a *zero-lease cache bypass* wherein the access is serviced but not cached. This prevents early eviction of non-expired items and preserves the cache partitioning that CARL/PRL intend in their assignments. Partitioning is intended to minimize the misses to cache and so the cost of making additional data transfers due to these uncached items is less than if elected to cache them at increased misses.

3.2 Reuse Interval Sampling

Again consider the five point stencil; from its inspection there are 6 memory references. The resulting reuse interval distribution of the program is straightforward as given in Table 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 [25] 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 manage data that would otherwise be managed in the register file. The clairvoyance breadth of the compiler required is a practical issue when considering CLAM as a cache solution.

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

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 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 latter subset of references are 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 populates a sampling counter which decrements at every access. When expired the table is populated with the current access fields, and the next number in the sequence is registered into the counter. 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. When this occurs the entry of the table that resulted in the match is evicted and its fields are stored into a separate buffer, along with the access trace, as the reuse interval of the memory reference. This is a pragmatic feature to reduce the number of active table entries that have already been associated with an RI. Eviction also forcibly occurs if all entries of the table are active when the sampling counter elapses. To allocate the new sample the oldest entry of the table is evicted. This entry, as it is not evicted due to a reuse, is written to the buffer with a negative RI to flag it as a non-reuse for CLAM.

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 bottlenecked limits the magnitude of reuse intervals that can be recorded by the table (capacity evictions become more frequent - removing entries with the largest active running intervals).

4 EVALUATION

We evaluate lease cache performance to foremost determine its benefit over alternative cache replacement policies. Variations in PRL phases are further trialed to examine how cache allocation is affected. The cache configurations are implemented on an FPGA running the same RISC-V core. We use seven benchmarks from the PolyBench suite to evaluate performance.

4.1 Implementation and Testing Setup

System Specification. The cache hardware variants are implemented on a CycloneV-GT FPGA [2]. The split, single level, fully associative cache memory consists of an 8KB instruction cache and an 8KB data cache. A block or cache line consists of 64 bytes (16 words). Transactions from cache to external memory (512MB of DDR3) are performed on four byte buses. Cache hits are serviceable same cycle, while misses have a 20-40 cycle latency depending on whether a write-back is necessary. Electing to design the cache in a single stage limits the base speed to 20 MHz on the FPGA, however, is easily scalable with additional pipeline stages. The base cache hardware is not designed with any functional optimizations other than a write-out buffer.

Software Support and Testing. A six stage pipeline RISC-V [25] core, with support for 32IM extensions, is used for benchmarking. The core is designed for a baremetal execution environment; however, limited I/O functionality is implemented using a hosted proxy for testing purposes. Through this we control the sampling system, test system, load binaries (by direct memory accesses), and collect run-time data. Programs are compiled to executable and linkable formats using the RISC-V GNU toolchain for embedded variants. Leases and lease cache configurations are allocated in static partitions of the binaries, which are mapped to hardware addresses known by the cache. These blocks are requested and populated into the LLT as a program preprocess.

Compiler Implementation. The CLAM 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. The table enumerates a list of load/store instruction addresses, and the lease or the dual leases for each instruction. In the case of a dual lease, the information includes two leases and a percentage number.

We have implemented source-level compiler analysis in LLVM based on Static Parallel Sampling (SPS) [7]. It analyzes and assigns leases for array references (as in the example program in Section 2.2). 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. In the first program execution, FPGA sampling provides the analysis, as described in Section 3.2. The code generator then computes and inserts reference leases based on the sampled traces. In the second execution, the generated code is tested for performance. The code generator implements all the algorithms described in Section 2, including CARL, PRL and FUL.

Performance Metrics. Cache performance metrics are only recorded when executing the benchmark kernel. For this paper we utilize, most significantly, access misses and vacancy statistics. Total accesses are not impacted by lease policy, so absolute miss counts accurately show improvement in cache performance. The system is cycle accurate so wall-clock time is not considered; however, LLT population equates to an overhead of roughly 16 block transfers (1KB of required LLT data / 64B per block) - a negligible cost. The memory overhead of lease cache is the product of the lease register size and number of cache lines (cost to store leases) plus the LLT cost. The LLT cost depends on the table field sizes (which are program dependent); however, assuming a uniform field size the cost can be approximated as twice the product of the number of entries and the field size.

Cache utilization shows the quality of the lease-based assignments. Under-assignment of leases results in an under-utilized cache (high quantities of expired lines), while over-assignment leads to over-utilization (no expired lines). Vacancy ratios are used to describe the achieved cache utilization of a particular lease policy. We define the following metrics for cache vacancy:

(1) No Vacancy Ratio - the ratio of auxiliary policy evictions (where no cache line is expired) to total evictions (by lease expiration and auxiliary policy).
(2) Multiple Vacancy Ratio - the ratio of expired evictions to total evictions when there are at least two expired cache lines.

The no vacancy ratio captures the degree of over-utilization. The higher the ratio, the greater the number of auxiliary (random) policy replacements due to non-expired leases. This suggests that the reference lease assignments are too large and prevent optimal cache management. The multiple vacancy ratio is a measure of under-allocation. A large ratio (≈ 1) indicates that the cache is saturated with expired lines. Alone the metrics identify degrees of poor cache performance; however, when used together describe ideal performance. Ideal utilization is one where at any access miss there is exactly one expired cache line. The equivalent metrics for this is a no vacancy ratio of 0 (there is always at least one expired line) and a multiple vacancy ratio of 0 (there is never more than 1 expired line).

Caching Policies. For benchmarking we try variable lease sets produced by both CARL and PRL. PRL is further run at varying phase counts. The best FUL cache performance for each benchmark is also achieved through experimental parameter sweep. We compare these policies against least recently used (LRU), pseudo least recently used (PLRU, using a single status bit in each cache line) [20], and static re-reference interval prediction (SRRIP) [15]. LRU is the traditional reference policy, PLRU is a practical employed cache solution, while SRRIP (2 bit scheme - empirically determined to be best for the elected program array) has the advantage over LRU of being able to mitigate thrashing.

Benchmarks. We use PolyBench/C 4.2.1, which contains 30 numerical kernels [19]. We use PolyBench for several reasons. First, its kernels are extracted from linear algebra, image processing, physics simulation, dynamic programming, and statistics. Second, the benchmark suite is relatively new and easier to port through the FPGA tool chain to allow testing on a real system. We selected 7 programs that have a version that has only integer operations. Polybench is part of a larger collection [4] which we may expanded into in future work.

Table 2: Test programs and baseline performance

Benchmark	Input Size (N)	Memory References	Memory Accesses	LRU Miss Count
Atax	120	60	491454	924
Doitgen	25	59	8885194	941
Floyd-Warshall	180	35	116868071	364160
2mm	60	86	6213792	19447
3mm	60	115	10892247	34990
Mvt	120	54	491302	15331
Nussinov	180	98	20051779	335369

The input size is shown in Table 2. We chose the input size to have a typical miss ratio for the 8KB data cache. The miss ratio ranges from 0.01% to 3.1%, as shown in Table 2. All programs use data stored in arrays. Each element in an array is 4 bytes. A cache block is 64 bytes and contains 16 array elements. The chosen LLT size of 128 entries allows the lease cache to statically load all references for any of the target applications. For applications where the reference set size exceeds the LLT capacity dynamic LLT content control can be used to reduce the number of active LLT entries; however, for this work this is not considered.

4.2 Overall Comparison

Figure 2 shows the results for all seven tests. For each program, it shows miss ratios as bars normalized to the LRU miss ratio. The first bar represents LRU, and so it is always 1.

The second bar in each group is PLRU, which approximates LRU using a single bit in each cache line. PLRU increases the miss ratio in most tests, but is within 5% margin of LRU performance. The third bar is SRRIP. It shows a mixed effect, reducing the miss ratio in one program (thrash mitigation) but increasing it in two other programs. The effect is negligible in the four remaining programs - being equivalent to PLRU.

The best uniform lease result is shown by the fourth bar. The performance of FUL varies depending on the uniform lease that is selected. We show the best performance across all tested lease values for each program. The complete results are included in the appendix of the paper. At its best, FUL has the same performance as LRU for the first three programs, reduces the miss ratio slightly in the next two, and yields large reductions in the last two.

The last two bars show the two algorithms of CLAM. Both have changed the miss ratio of all tests compared to LRU. This shows that for scientific kernels for a fixed size cache, PRL consistently out-performed CARL. We discuss them in detail in the next section.

4.3 PRL vs. CARL

The best performing technique overall is PRL. As described in Section 2.4, PRL first divides a program execution into phases and ensures no excessive allocation in any phase. It may help to consider an analogy between cache allocation and governmental spending. The budget demand may vary. CARL is analogous to balancing the budget at the end of the year but having no spending limit until the end. PRL, however, ensures there be no deficit at the end of each quarter.

Figure 2 shows that CARL reduces the miss ratio relative to LRU in all but *2mm*, for which it causes a significant increase. PRL, however, is able to reduce the miss ratio in all tests. Furthermore, PRL improvement is greater than the best FUL improvement in all tests except for *nussinov*.

Figure 3 compares CARL and PRL in more detail. Four of the seven benchmark programs are shown. Atax, Doitgen, and Floyd-Warshall are not displayed because their leases assigned by CARL are identical to those assigned by PRL. CARL is the same as a single-phase PRL. Each other PRL variant is labeled with the number of phases it uses. To continue the analogy with governmental spending, more phases in PRL is analogous to dividing a fiscal year into more periods with the spending cap at the end of each period.

As discussed in Section 4.1, two performance metrics, the cache no vacancy ratio and the multiple vacancy ratio, measure the degree of over- and under-allocation. These two ratios are measured and shown in Figure 3 for CARL and all PRL variants.

CARL over-allocates the cache with its leases, shown by its no vacancy ratios as high as 64%. PRL is able to eliminate over allocation, shown by the near 0% no vacancy ratios it has achieved in every test. Intuitively, the more phases in PRL, the more conservative it is. This is the case for three programs shown in the figure. When PRL uses 2 to 20 phases, the vacancy ratio decreases from 9.5% to 0% in *2mm*, 23.3% to 0.2% in *3mm*, and 60.8% to 39% in *nussinov*. In *mvt*, however, the no vacancy ratio initially increases from 21.9% with two phases to 5.5% with five phases, but it drops to 7.6% and then 34.6% with 10 and 20 phases respectively.

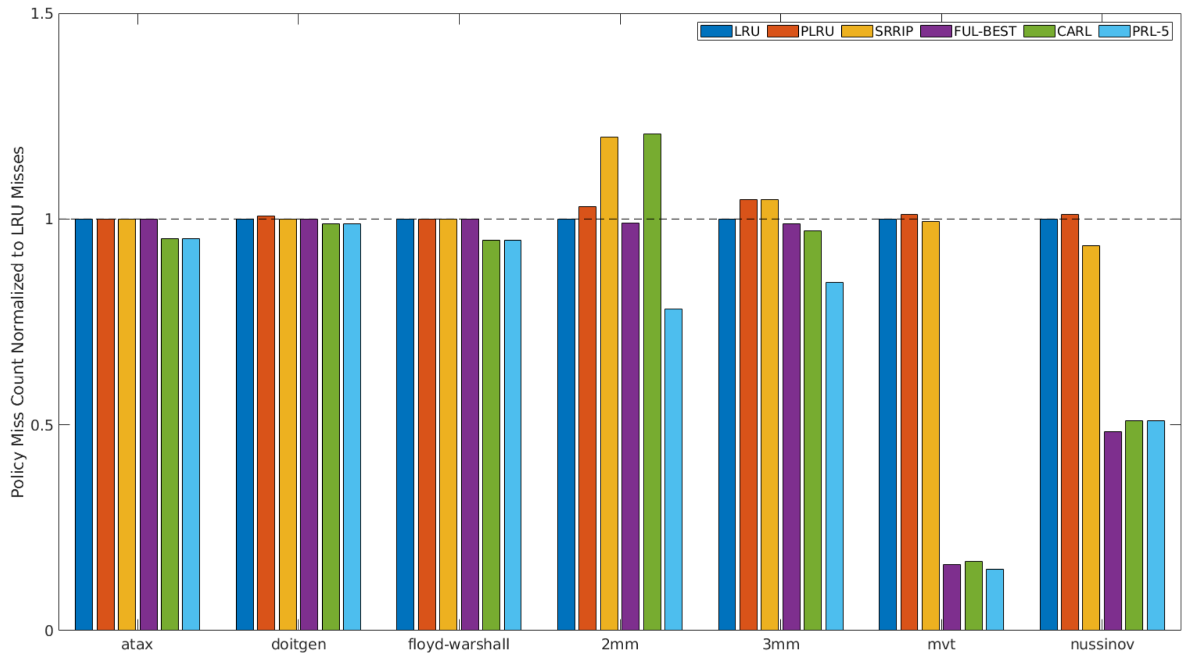

Figure 2: Normalized miss ratios of caching policies

Comparing the leases of *mvt* of PRL-10 and PRL-20, we see that when dividing it into more phases, reference leases either stay the same or become smaller. For the dual-lease assignment, the percentage of accesses with the longer lease is also reduced in PRL-20. In other words, PRL-20 allocates cache strictly less than PRL-10, so the cache has more vacancies. The opposite result shown by the vacancy ratio is an artifact of how it is measured.

Recall that the lease cache supports zero-lease cache bypass, as described in Section 3.1. At a miss, the cache is polled to see if there is a vacancy. At a bypass, the cache is not involved. PRL-20 assigns more zero leases than PRL-10, e.g. the greater percentage in the dual-lease reference that was assigned the shorter lease which was zero. Many of the vacancy counts in PRL-10 became bypasses and were not counted in PRL-20. As a result, the vacancy ratio dropped. The only other benchmark with zero lease assignments was *nussinov*, in which they are significantly less frequent than in *mvt*.

5 VISUALIZING CACHE DYNAMICS

With the implementation of CLAM and associated data collection in hardware, we have the ability to visualize the state of the cache. This section defines and uses two types of graphs as follows.

The first type of graph shows the *aggregate cache vacancy*. At each moment, the aggregate vacancy is the number of cache blocks with an expired lease. We then draw the aggregate vacancy as a time series across the entire execution.

The second type of graph shows the current lease in each cache block. At each moment, the entire cache space of c blocks is shown as a column of c cells, colored individually to show the current remaining lease in that block. From the starting time at the left boundary to the ending time at the right boundary, we plot the execution as a matrix of colored cells. We call it a *cache tenancy spectrum*.

We have implemented the FPGA to output the aggregate vacancy and the remaining lease values every k accesses during an execution. Figure 4 shows the results for *3mm*. The program has the most distinct phase behavior, and it is the only program where both PLRU and SRRIP performed worse than LRU.

The aggregate cache vacancy and the cache tenancy spectrum are shown for CARL in the top row and PRL in the bottom row. Comparing the vacancy graphs on the left, we see a division of three parts in both graphs. CARL and PRL differ mainly in the first phase, which includes the first 2.46 million accesses out of the 10.85 million total accesses. In this period, the aggregate vacancy is zero for CARL but between 1 and 20 for PRL. In other words, CARL over allocates the cache in this period, but PRL does not.

The difference is shown by the two spectra graphs on the right. Three colors are used: one for leases greater than 255, under 255, and after expiration. In the first phase under

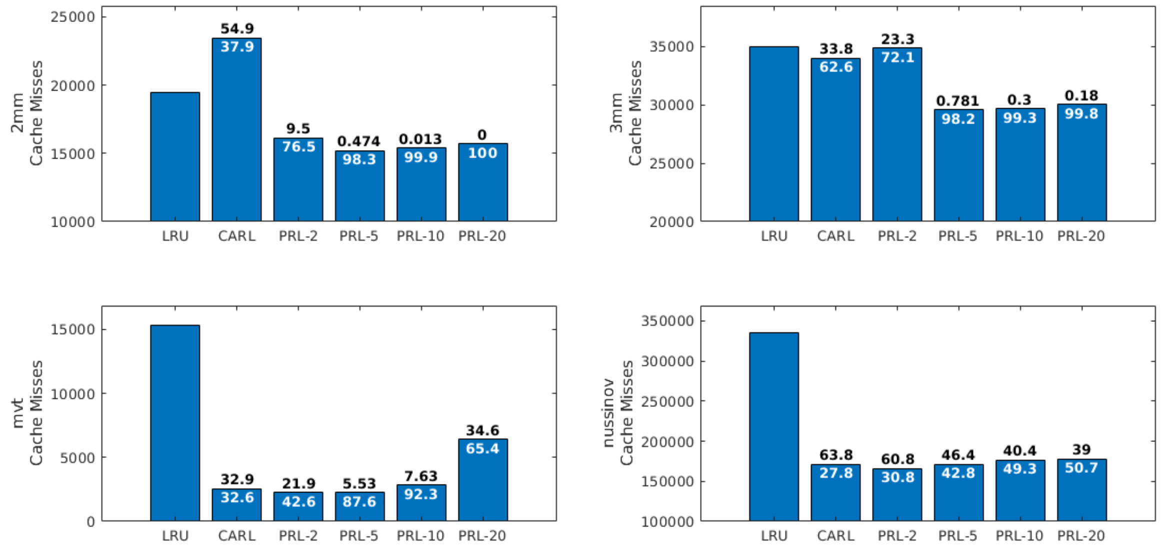

Figure 3: Comparing CARL and PRL for the four tests. Each PRL variant is labeled with the phase count. The numbers at each bar show the no vacancy ratio (above) and the multiple vacancy ratio (below the bar top). CARL over-allocates the cache with its leases, shown by its no vacancy ratios as high as 64%. PRL eliminates over-allocation in the first three tests, shown by their near 0% no vacancy ratios.

CARL, most cache blocks are assigned long leases, and there is no vacancy. Under PRL, however, many leases are shorter, and there is some vacancy at most times.

Visualization helps to examine and analyze the effect of a caching policy. This example shows that the two new types of plots are useful in identifying the length of time and intensity of over allocation of cache by CARL.

6 RELATED WORK

Lease Cache. The term *cache leases* was initially used in distributed file caching [13]. Such uses continue today in most Web caches, e.g. Memcached [12], and recently in TLB [3]. A lease specifies the lifetime of data in cache to reduce the cost of maintaining consistency. In 2019, Li et al. defined the lease cache [17]. Similar to prior work, a data block is evicted when the lease expires. However, it differs in that the lease is re-assigned *each time* the data block is accessed. The implementation is more difficult, because it must manage the lease at every access. As far as we know, this paper is the first hardware design and implementation of the lease cache.

Li et al. gave the algorithm called *Optimal Steady-state Lease (OSL)* [17]. The basic CARL algorithm in Section 2.2 is a direct copy of OSL. While OSL assigns a lease for each data page, CARL assigns it for each reference. This paper, however, extends OSL in significant ways. First, the number of references can be many orders of magnitude less than the number of pages. Reference leases are practical for hardware caches, while per page leases are not. Second, a reference may access an arbitrary amount of data, which requires dual leases. Third and most importantly, while OSL and CARL optimize for a variable-size cache, PRL provides a novel extension to avoid over-allocation in a fixed-size cache.

Software Managed On-chip Memories. The management of on-chip memories has been a long standing interest in compiler research. Alam and Horspool [1] surveyed the literature up to 2015. Among these, Udayakumaran et al. [21] showed that the performance of their compiler solution (called the Data-Program Relationship Graph or DPRG) enables dynamic memory allocation and as a result could significantly outperform optimal static allocation. Li et al. [16] extended the static solution of graph coloring to enable dynamic allocation through live-range splitting and interval coloring. Udayakumaran et al. [21] reported a performance comparable to a direct-map cache. Our study uses the fully-associative cache, which is a more robust baseline.

It is non-trivial to implement dynamic allocation by a compiler. The lease cache de-couples allocation from its implementation. A compiler simply makes a *request* of allocation by specifying a lease. There is a broad range of analysis that can potentially be used, including not just DPRG and

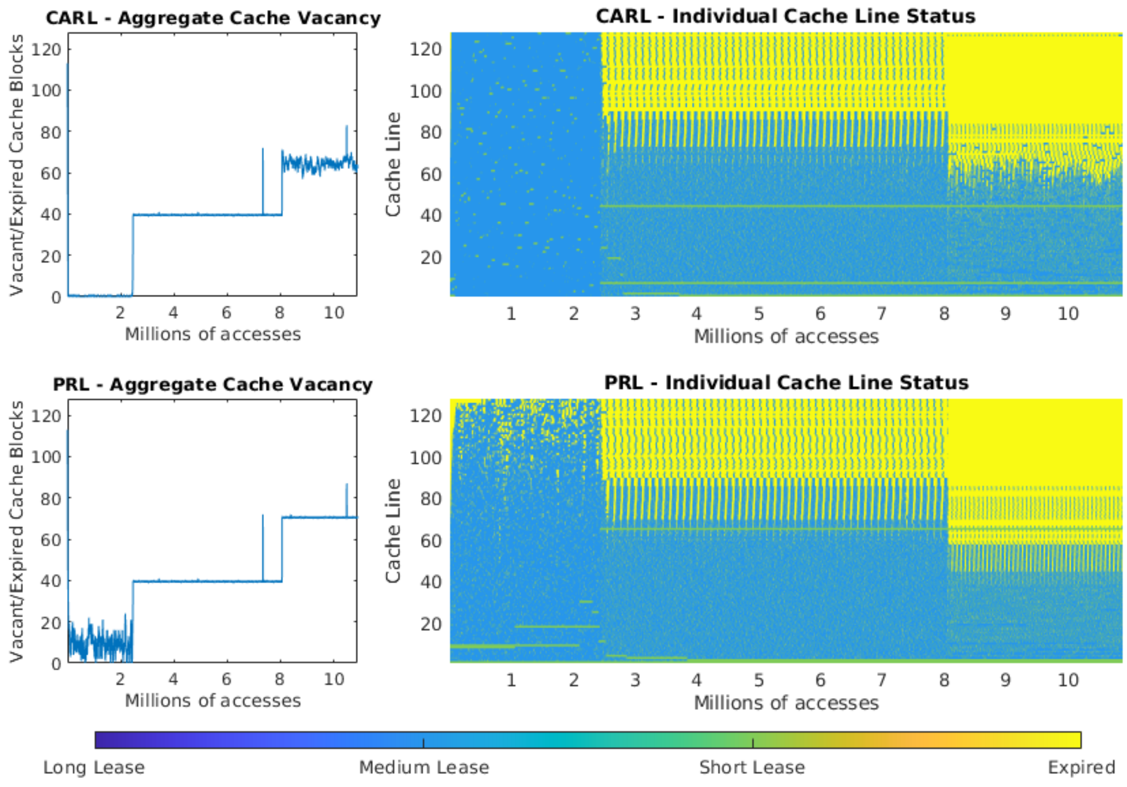

Figure 4: The aggregate cache vacancy and the cache tenancy spectrum for *3mm* using CARL (top row) and PRL (bottom row).

graph coloring but also cutting-edge locality models in other areas [4, 23]. Although such compiler designs are outside the scope of the paper. Aside from its leasing algorithms, CLAM provides a programming interface for use by other algorithms and a fall-back policy, i.e. uniform leases, that is performance safe.

Working Sets and Stack Algorithms. The uniform lease corresponds to the working-set parameter τ. Denning defined the working set as the data touched in the last τ accesses [8–10]. Virtual memory management is based on working sets. It can be viewed as a variable-size cache.

Fixed-size caches are managed by a replacement policy. Mattson et al. [18] showed that a set of policies including LRU, MRU, LFU, and OPT can be modeled as a stack. In essence, these policies maintain a total order among the cache blocks, and this order is charted by the position of each block in the stack. CLAM is a hybrid design — it uses program control but manages a fixed-size cache with a replacement policy. Unlike stack algorithms, CLAM does not use a total ranking. Each block is assigned a lease, and the lease is maintained individually. One benefit is the ability to visualize a caching policy as shown in Section 5.

Stack algorithms impose global management, which has two shortcomings. First, stack algorithms do not permit software control. An exception is collaborative caching. The second problem of global management is cost, which practical designs have solved using pseudo LRU, which we have evaluated in Section 4. We next discuss these two aspects.

Hardware Cache Design. Hardware cache is set associative, so each eviction considers just the data blocks in the same set. Sampling by our lease cache has the effect of associativity. We plan to add associativity in our cache design and evaluate its effects.

Duong et al. developed the *Protecting Distance-based Policy (PDP)* [11]. PDP "prevents replacing a cache line until a certain number of accesses to its cache set." It is the same as the uniform lease in our design. To choose the best protecting distance, PDP included additional hardware to sample the reuse interval (called reuse distance in the paper) and adaptively found the protecting distance that maximized the hit ratio. In comparison, CLAM algorithms assign different leases. In performance, the best FUL result is equivalent to the best off-line PDP result. As shown in Section 4, PRL improvement is greater than the best PDP improvement in all our tests except for one.

It is a widely recognized problem that LRU does not perform well for the streaming pattern, e.g. when an array larger than the cache size is repeatedly traversed. Many techniques have been developed, including page coloring, cache replacement techniques such as SRRIP (see Section 4), cache bypassing, and cache hints. Two recent techniques are Talus [5] and SLIDE [22]. Talus partitions the access stream to have the effect of dividing the working set, and SLIDE, with scaled-down simulation, achieves this effect for stack or non-stack cache policies. The working-set portioning by the uniform lease (discussed in Appendix) has a similar effect, but using program rather than cache control.

Collaborative Caching. Wang et al. [24] coined to term collaborative caching where a program uses cache hints to communicate program knowledge to the cache as hints. A series of techniques have been developed to insert cache hints in software [6, 14, 24]. Collaborative caching is performance safe, because a program can always choose not to speculate. Collaborative cache can be used to cache any program without program information. By using uniform leases, CLAM shares these qualities of the collaborative cache.

Unlike the lease cache, however, the collaborative cache does not implement complete program control. In fact, it requires the fully automatic cache control as the baseline. Cache hints make the cache more complex to implement and model. In comparison, the lease cache are controlled by reference and uniform leases, which may be optimized using CLAM algorithms.

In collaborative cache, program control is secondary. In fact, it requires the fully automatic cache control as the baseline. This makes it trivial to establish performance safety. However, cache hints make the cache system strictly more complex. Developing software techniques is challenging since they must model the global management in the baseline cache control. The collaborative cache is not just more complex and also more complex to model. The lease cache, in contrast, implements program control. It can be controlled as simply as assigning a single uniform lease for the entire execution. Cache management becomes localized in that its lease tracking is de-coupled among data blocks. On the other hand, collaborative cache can achieve optimal cache performance with more complex program control, i.e. by inserting a hint individually for every access (and changing it when the cache size changes) [14].

7 SUMMARY

This paper has presented CLAM for compiler optimization of cache management. The CLAM algorithms include CARL, which optimizes for a variable size cache, PRL, which optimizes for a fixed size cache. Both use dual leases to more fully utilize the cache. By using variable-length leases, CLAM achieves performance beyond what is possible with traditional automatic caching. For programs that are not amenable to static or profiling analysis, the compiler inserts uniform leases which perform at least as good as LRU.

CLAM has been implemented on a CycloneV-GT FPGA card with a RISC-V processor and the new hardware cache. The evaluation has shown performance improvements over existing techniques in all of the 7 programs tested from the Polybench suite. This is the first hardware design and implementation of the lease cache.

ACKNOWLEDGMENTS

The authors would like to acknowledge and thank Sreepathi Pai, Fangzhou Liu, Wesley Smith, Aaron Gindi, Donovan Snyder, Elias Neuman-Donihue, Joshua Radin, and Katherine Seeman for their help and participation in this work.

The research is supported in part by the National Science Foundation (Contract No. CNS-1909099, CCF-1717877). 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.

REFERENCES

[1] Shahid Alam and R. Nigel Horspool. A survey: Software-managed on-chip memories. *Comput. Informatics*, 34(5):1168–1200, 2015.

[2] Altera. *Cyclone V GT FPGA Development Board Reference Manual*. Altera.

[3] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. Avoiding TLB shootdowns through self-invalidating TLB entries. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 273–287, 2017. doi: 10.1109/PACT.2017.38. URL https://doi.org/10.1109/PACT.2017.38.

[4] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, and P. Sadayappan. Analytical modeling of cache behavior for affine programs. *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2(POPL):32:1–32:26, 2018.

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

[6] Kristof Beyls and Erik H. D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 51(4): 223–250, 2005.

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

[8] Edward G. Coffman Jr. and Peter J. Denning. *Operating Systems Theory*. Prentice-Hall, 1973.
[9] Peter J. Denning. The working set model for program behaviour. *Communications of the ACM*, 11(5):323–333, 1968.
[10] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1), January 1980.
[11] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, pages 389–400, 2012. doi: 10.1109/MICRO.2012.43. URL https://doi.org/10.1109/MICRO.2012.43.
[12] Brad Fitzpatrick. Distributed caching with Memcached. *Linux Journal*, 2004(124):5, 2004.
[13] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 202–210, 1989. doi: 10.1145/74850.74870. URL https://doi.org/10.1145/74850.74870.
[14] Xiaoming Gu and Chen Ding. A generalized theory of collaborative caching. In *Proceedings of the International Symposium on Memory Management*, pages 109–120, 2012.
[15] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010.
[16] Lian Li, Jingling Xue, and Jens Knoop. Scratchpad memory allocation for data aggregates via interval coloring in superperfect graphs. *ACM Trans. Embedded Comput. Syst.*, 10(2):28:1–28:42, 2010.
[17] Pengcheng Li, Colin Pronovost, William Wilson, Benjamin Tait, Jie Zhou, Chen Ding, and John Criswell. Beating OPT with statistical clairvoyance and variable size caching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, 2019.
[18] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.
[19] Louis-Noel Pouchet and Tomofumi Yuki. Polybench/c 4.2.1. http://https://sourceforge.net/projects/polybench/files/, 2016.
[20] Kimming So and Rudolph N. Rechtschaffen. Cache operations by MRU change. *IEEE Transactions on Computers*, 37(6):700–709, 1988. doi: 10.1109/12.2208. URL https://doi.org/10.1109/12.2208.
[21] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Transactions on Embedded Computer Systems*, 5(2):472–511, 2006.
[22] Carl A. Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *Proceedings of USENIX Annual Technical Conference*, pages 487–498, 2017. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger.
[23] Qingsen Wang, Xu Liu, and Milind Chabbi. Featherlight reuse-distance measurement. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 440–453. IEEE, 2019.
[24] Z. Wang, K. S. McKinley, A. L.Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, Charlottesville, Virginia, 2002.
[25] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The risc-v instruction set manual, volume i: Unpriviledged isa. Technical report, EECS Department, University of California, Berkeley, June 2019.

A FUL PERFORMANCE AND ANALYSIS

When there is no vacancy, CLAM uses random eviction as mentioned in Section 3.1. The choice is made after examining three eviction methods, as listed next:

(1) *Random*: a randomly chosen block regardless of the remaining lease
(2) *SRL*: shortest remaining lease
(3) *LRL*: longest remaining lease

Unlike Random, which is lease oblivious, the other two choose a victim based on the remaining lease, shortest for SRL and longest for LRL respectively. To implement SRL and LRL efficiently, we sample a few cache blocks and check their leases. The number of cache blocks sampled is the *pool size*.

By testing all three eviction methods, we determine whether there is a benefit in using lease-based eviction.

The performance is shown in Figure 6 by the total number of misses of FUL normalized to that of LRU.

FUL is programmable. We first consider the best performance, i.e. lowest miss count, and comparison with LRU. Judging by the best performance, FUL is significantly better than LRU for two programs, slightly better for another two, and the same as LRU for the remaining 4 programs.

Table 3 shows the best reduction over LRU and the FUL parameters which give the best FUL performance for four of the benchmarks used. For *Nussinov, Mvt*, FUL can be programmed to reduce the number of misses by 60% and 85% respectively over LRU. For *2mm, 3mm*, the improvement is about 4% and 6% respectively. Except for *Mvt*, the improvements by the best FUL are surpassed by

Table 3: Best FUL miss count reduction over LRU in four benchmarks.

Benchmark	FUL Eviction	Pool Size	Miss Count Reduction [%]	Best FUL Lease
Nussinov	LRL	8	60.28	3683
Mvt	LRL	8	85.15	2286
2mm	All	All	3.92	1778
3mm	All	All	5.62	1905

Case Study Mvt. The greatest improvement happens for *Mvt*, which we analyze in more detail. The program computes a matrix-vector multiplication twice. In the second time, the matrix is traversed column by column, shown by the following code:

```
for (i=0; i < N; i++)
  for (j=0; j< N; j++)
    x2[i] = x2[i] + a[j][i] * y2[j];
```

Checking the source code, we found that $N = 120$. The inner loop traverses one column of the matrix, which consists of 120 data blocks; 1 data block for vector $x2$; and 8 for $y2$. The total working set is just larger than the cache size, which is 128 blocks. As a result, all matrix accesses miss in the

LRU cache. In the FUL cache, when the lease covers only a portion of the matrix column, the remaining portion is evicted randomly. As a result, the remaining working set fits in the cache, i.e. most matrix accesses become cache hits, and hence the improvement.

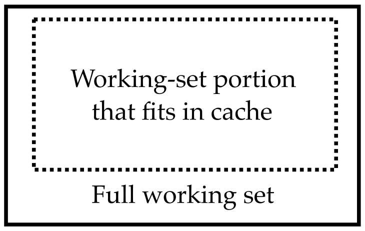

Figure 5: Illustration of working-set portioning

Working-set Portioning. In effect, the FUL lease and eviction method divide the working set of *Mvt* and store a portion of the working set that fits in the cache. We call this effect *Working-set Portioning*.

The effect is better explained using the simple illustration in Figure 5. The best performance happens when the lease captures the full working set. When the working set exceeds the cache size, the FUL cache evicts part of the working set so it fits in the available cache. The effect of working-set portioning is seen in three other programs.

For streaming accesses, the best policy is most recently used (MRU) eviction, which is why we see LRL performs the best in *Mvt* and *Nussinov*. It may appear puzzling that SRL can improve over LRU. The reason is the set sampling used by FUL when selecting a block for eviction. It does not evict deterministically as the fully associative LRU cache does. This is also the reason that for *2mm,3mm*, all three eviction methods perform better.

When the lease is too short to capture the working set, the FUL cache uses random replacement. In the extreme case when the lease is 0, the miss ratio is that of random replacement, which is worse than LRU in six of the test programs. When the lease is too long, it “covers” data not in the working set. SRL behaves the same as LRU. However, in this case both Random and LRL perform worse than LRU.

If we compare the three eviction policies overall, we see that no one policy dominates another. Among them, random is the most space efficient and fastest to implement in hardware. We therefore use random in CLAM.

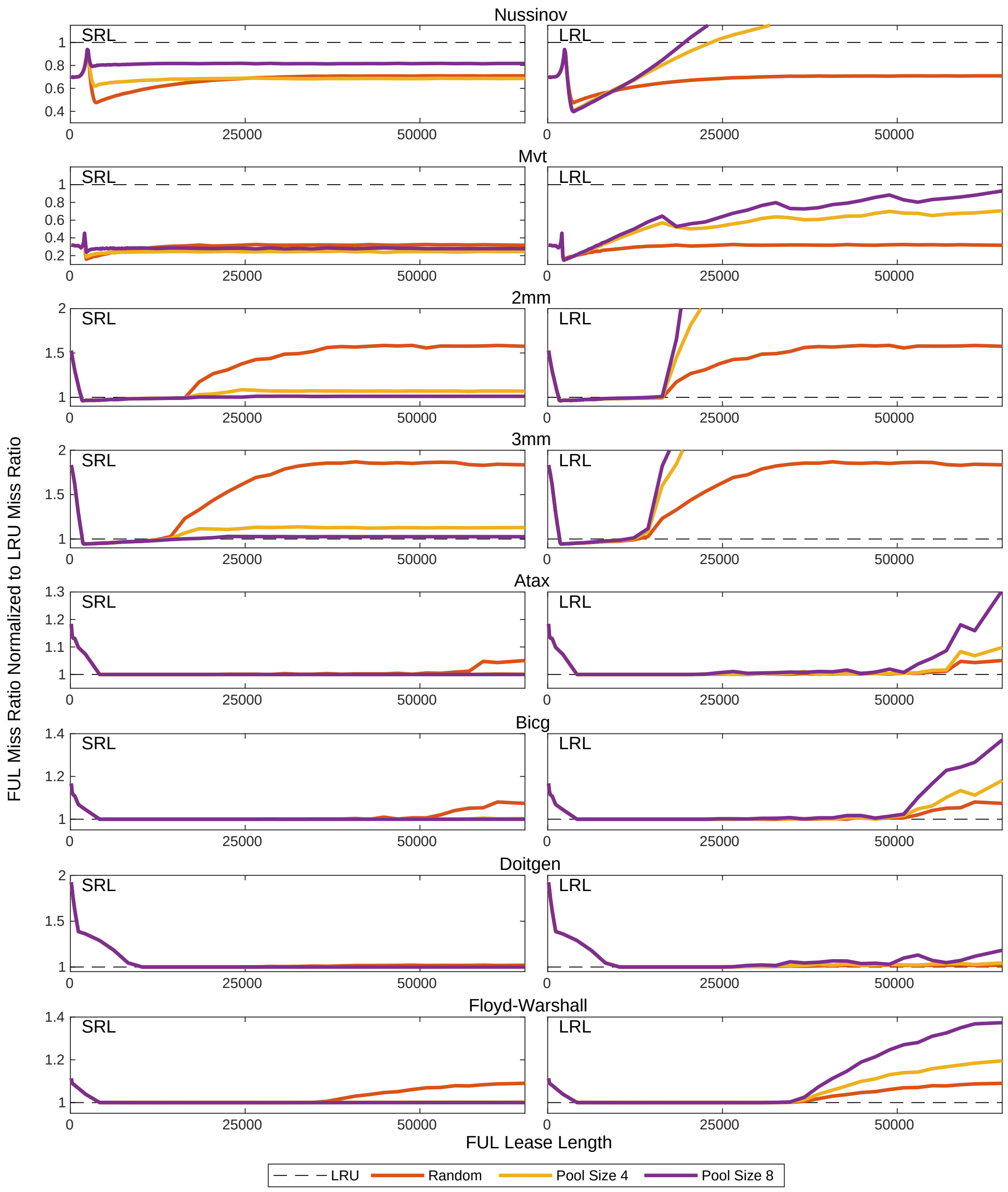

Figure 6: The miss ratio of FUL cache normalized to LRU for each test program with eviction methods SRL (left graph), LRL (right graph), Random (replicated in both graphs), and the FUL lease as the x-axis in each graph.