Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

8-2020

CLAM: Compiler Lease of Cache Memory

Ian Prechtl
irp2474@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Prechtl, Ian, "CLAM: Compiler Lease of Cache Memory" (2020). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

CLAM: Compiler Lease of Cache Memory

by

Ian Prechtl

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Electrical Engineering

Supervised by

Dr. Dorin Patru
Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY

August 2020

Approved By:

Dr. Dorin Patru
Associate Professor - Rochester Institute of Technology Department of Electrical and Microelectronic Engineering

---

Dr. Chen Ding
Professor - University of Rochester Computer Science Department

---

Prof. Mark Indovina
Senior Lecturer - Rochester Institute of Technology Department of Electrical and Microelectronic Engineering

---

Prof. Carlos Barrios
Lecturer - Rochester Institute of Technology Department of Electrical and Microelectronic Engineering

---

Dr. Ferat Sahin
Professor & Department Head - Rochester Institute of Technology Department of Electrical and Microelectronic Engineering

CLAM: Compiler Lease of Cache Memory

by

Ian Prechtl

Submitted to the Department of Electrical and Microelectronic Engineering
August 2020, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

Abstract

Caching is a common solution to the data movement performance bottleneck of today's computational systems and networks. Traditional caching examines program behavior and cache optimization separately, limiting performance. Recently, a new cache policy called Compiler Lease of cAche Memory (CLAM), has been suggested for program-based cache management. CLAM manages cache memory by allowing the compiler to assign leases, or lifespans, to cached items over a hardware-software interface, known as lease cache. Lease cache affords new performance potential, by way of program-driven cache optimization. It is applicable to existing cache architecture optimizations, and can be used to emulate other cache policies.

This paper presents the first functional hardware implementation of lease cache for CLAM support. Lease cache hardware architecture is first presented, along with CLAM hardware support systems. The cache is emulated on an FPGA, and benchmarked using a collection of scientific kernels from the PolyBench/C suite, for three CLAM lease assignment policies: Compiler Assigned Reference Leasing (CARL), Phased Reference Leasing (PRL), and Fixed Uniform Leasing (FUL). CARL and PRL are able to achieve superior performance to Least Recently Used (LRU) replacement, while FUL is shown to serve as a safety mechanism for CLAM. Novel spectrum-based cache tenancy analysis verifies PRL's effectiveness in limiting cache utilization, and can identify changes in the working-set that cause the policy to perform adversely. This suggests that CLAM is extendable to more complex workloads if working-set transitions can elicit a similar change in lease policy. Being able to do so could yield appreciable performance improvements for large and highly iterative workloads like tensors.

Thesis Supervisor: Dr. Dorin Patru
Title: Associate Professor

Acknowledgments

Advisor

Dorin Patru

Thesis Committee

Mark Indovina

Carlos Barrios

CLAM Group

Chen Ding

Ben Reber

Dong Chen

Reviewers

Sreepathi Pai

Fangzhou Liu

Wesley Smith

Aaron Gindi

Donovan Snyder

Elias Neuman-Donihue

Joshua Radin

Katherine Seeman

This work was supported by a NSF grant sub-award from University of Rochester.

Contents

1 Introduction **10**

1.1 Data Locality 10

1.2 Cache 11

1.2.1 Cache Policy 12

1.2.2 Software-Driven Management 14

1.3 Objective 14

2 Background **16**

2.1 Reuse Interval 16

2.2 Lease Cache: CLAM 19

2.2.1 CARL: Compiler Assigned Reference Leasing 20

2.2.2 CARL Extensions 21

3 Lease Cache Hardware Design **25**

3.1 Lease Cache Implementation 25

3.1.1 Hardware 26

3.1.2 Software Support 32

3.2 Hardware Support for CLAM 34

3.2.1 Lease Tracking 37

4 Testing **39**

4.1 Test System 39

4.1.1 Processor Core 40

4.1.2 Communication and Control 41
4.1.3 Design Parameters 43
4.2 Cache Performance Metrics 45
4.2.1 Cache Tenancy Spectrum 46
4.3 Benchmark Applications and Policies 47

5 Results and Discussion **49**
5.1 Fixed Uniform Leasing 49
5.2 Variable Leasing 54
5.2.1 CARL vs. PRL 56
5.2.2 PRL Resolution 61
5.2.3 Preliminary Set Associativity 63

6 Future Work **66**
6.1 Scope Leasing 67
6.1.1 Multiple Dual Leases 68
6.2 Local Clocks 68
6.3 Parallel Assignment 69

7 Conclusion **70**

Glossary **73**

Acronyms **75**

A Tables **78**

List of Figures

1-1 Intel Haswell (i7-4770) cache memory hierarchy [1]. 12

2-1 Forward reuse interval calculation for an arbitrary access sequence. Red circles indicate the re-access of an item whose original access is circled in black. The first re-access includes directional arrows to show how the reuse interval calculation traces back to the reference that is associated with the interval. 17

2-2 CARL assignment procedure for a budget of one billion. Green items indicate a lease assignment made at the current step. Blue indicates a lease assignment previously made, which is contributing to the overall cost. Red indicates assignment termination due to the budget being exceeded. 24

3-1 Lease cache lookup stage hardware overview for fully and set associative cache. Component and signal sizings are given in Section 4.1.3. Note that *lease_multiplexer_bus* is driving a NOR reduction gate. . 28

3-2 Lease cache update stage for fully associative cache. Component and signal dimensions are given in Section 4.1.3. Note that *exp* bus drives an OR reduction gate. 30

3-3 Lease cache update stage for a generic four way set associative cache, with update circuitry shown for three sets. The *exp* bits of each set drive an OR reduction gate in the schematic. Component and signal dimensions are given in Section 4.1.3. Parameter *s* is set size. 32

3-4 Mapping of lease cache memory partitions in application binary. Addresses are byte addressable. 33
3-5 Hardware reuse interval sampler system overview. Delimited text file shown is output read from the sampler with same field order shown in the buffer. 36

4-1 Top level test system diagram. 40
4-2 Cache tenancy spectrum for the example stencil (Listing 2.1). 47

5-1 FUL policy performance over a 16-bit lease range with 7-bit resolution. FUL curves are normalized to baseline (LRU) performance, indicated by the dashed lines. 50
5-2 FUL policy vacancy of the trials shown in Figure 5-1. 52
5-3 Normalized miss ratios of trialed caching policies [27]. Dashed line indicates baseline (LRU) performance. Tabled results given in Figures A.1 and A.2. 54
5-4 Cache aggregate vacancy (left) and cache tenancy spectrum (right) for doitgen benchmark using CARL (top) and PRL (bottom). 57
5-5 Cache aggregate vacancy (left) and cache tenancy spectrum (right) for nussinov benchmark using CARL (top) and PRL (bottom). 58
5-6 Cache aggregate vacancy (left) and cache tenancy spectrum (right) for mvt benchmark using CARL (top) and PRL (bottom). 59
5-7 Cache aggregate vacancy (left) and cache tenancy spectrum (right) for 2mm benchmark using CARL (top) and PRL (bottom). 60
5-8 Cache aggregate vacancy (left) and cache tenancy spectrum (right) for 3mm benchmark using CARL (top) and PRL (bottom). 61

5-9 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 vacancy ratios as low as 36%. PRL eliminates over-allocation in the first three tests, shown by their near 100% vacancy ratios [27]. Tabled results given in Figure A.2. 62

5-10 How well set associative caches perform when using leases designed for the architecture and ignoring set constraints. Blue bars are lease assignments made by CARL, which assume a fully associative cache. Red bars are lease assignments made by spatial PRL, which groups RI distributions by set. The numbers at the top of each bar pair is the no vacancy ratio of each bar respectively. Tabled results given in Figure A.3. 64

List of Tables

2.1 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 [27]. 19

4.1 Compiler options for RISC-V toolchain. 41

4.2 Hardware design parameters for base cache, lease cache, reuse interval sampler, and lease tracker. 44

4.3 Benchmark programs and their baseline (LRU) performance. 48

5.1 Best FUL miss count reduction over LRU in four benchmarks. 51

A.1 Benchmark performance summary for reactive cache policies. 78

A.2 Benchmark performance summary for CLAM cache policies. Note: MV Rep (Replacments) are the number of evictions made when there are two or more expired cache lines - the numerator for MVR. 79

A.3 Benchmark performance summary for set associative CLAM cache policies. Note: Rep = Replacements. 80

Chapter 1

Introduction

For years, processor speeds have increased disproportionately to main memory speeds, limiting computational ability. Distributed networks and servers have seen similar effects in how they have scaled, especially with the rise in popularity of cloud computing. As applications and networks continue to increase in size and complexity, the impacts of the data bottleneck become that much more significant and hazardous to performance. Caching has been the long-standing solution to this issue. Using intermediate storage nodes, data can be more efficiently and effectively used, but only if the cache itself is appropriately managed. This work studies an alternative method of cache management, aimed at improving memory performance through program-driven analysis. This chapter serves as an introduction to cache theory, practicality, and management.

1.1 Data Locality

Locality is a highly exploitable, and equally dangerous, property of data. It is the tendency of data to be re-accessed, once initially accessed. It arises from program and data structuring, along with being an innate stochastic attribute of computer systems. There are two main categories of locality,

1. *Temporal locality* - if a data object is accessed, then it is likely that the same object will be re-accessed in the near future. Ex, an iterative control loop.

2. *Spatial locality* - if a data object is accessed, then it is likely that an adjacent object will be accessed in the near future. Ex, the elements of a data array.

In computing, locality is used to improve performance. If data is known to have temporal locality, then it would benefit the system to retain that data (as close to the core as possible in CPU). Similarly, if data has spatial locality, it would benefit the system to store data from the same memory region or node. Identifying these properties gives predictive insight about how a particular computational sequence will occur, which can consequently be used for performance gain.

1.2 Cache

Caching is an application of data locality. It is the practice of keeping often used data closest to where it is needed, to decrease the time required to access it. In the context of hardware, caches are intermediate storages between the processor and main memory that manage data and service requests from the core. Cache is robust in that it can be implemented in various configurations (size, architecture, etc.), but limited in the performance that can be gained from these different designs. Intel's Haswell (i7-4770) architecture, released in 2013, features several levels of cache in order to optimize the performance of the processor (Figure 1-1). Each level of increasing capacity is characterized by higher latency and associativity (cache freedom). These are common trade-offs; cache near the processor has more restrictive designs to retain low latency, while caches further away are larger and more complex (to accommodate more data, from multiple sources). The objective of the hierarchy is simple: keep data that will be accessed most immediately closest to the processor (L1 cache), and keep data that will be access later farther away (L2 or L3). If correctly managed, the cost of moving data between these levels is minimized.

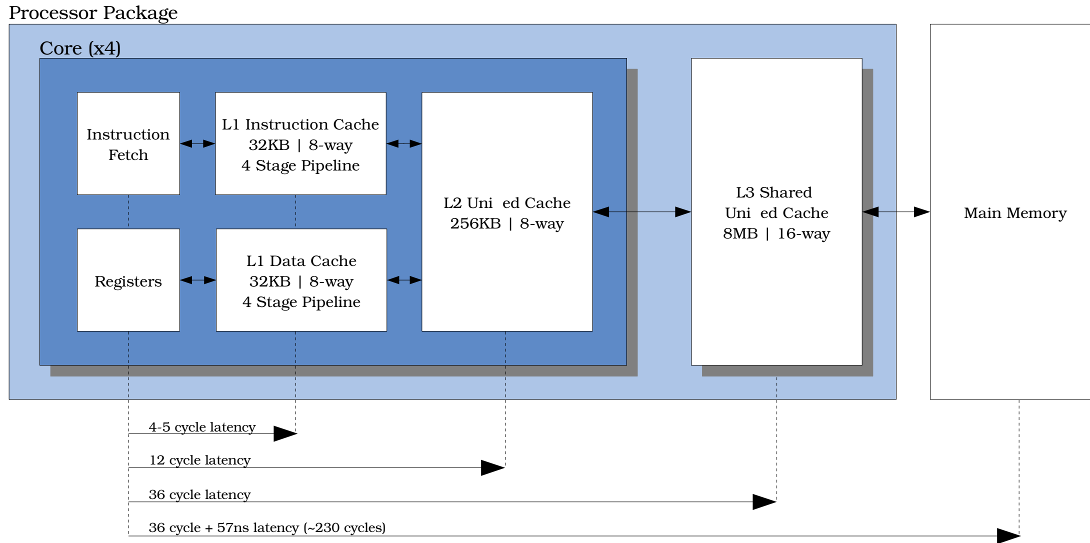

Figure 1-1: Intel Haswell (i7-4770) cache memory hierarchy [1].

1.2.1 Cache Policy

Because hardware cache is finite, there is a limit to how many items it can hold, or how many can be *allocated*. Conversely, a cache needs to remove, or *evict*, an item if it is full when it needs to allocate a new item. Cache must manage its content so that its performance is maximized (i.e. keeping any item that will be used soon, and removing any that will not be). How it manages itself and its content is known as the *cache policy*.

Cache policy is able to exploit data locality by targeting one or several attributes of data, and managing itself in accordance with this. For example, data with high temporal locality would be best managed by a policy that identifies which items have been accessed least recently, and prioritizes them for eviction. Policy effectiveness is generally discussed relative to *access patterns*, or patterns of locality. Examples of such patterns are stream, thrash, recency-friendly, cyclical and sawtooth. All have unique characteristics relating to recency, frequency, and size. General patterns of recency-friendly, thrash, and stream are given in Equations 1.1-1.3. For these patterns a is a data array of arbitrary length k, and N is any positive integer.

$$(a_1, a_2, ..., a_{k-1}, a_k, a_k, a_{k-1}, ..., a_2, a_1)^N \qquad \forall k \tag{1.1}$$

$$(a_1, a_2, ..., a_k)^N \qquad k > \textit{cache size} \tag{1.2}$$

$$a_1, a_2, ..., a_k \qquad k = \infty \tag{1.3}$$

Just as access patterns have unique attributes, there exist cache policies which perform well for some patterns, and poorly for others. For the patterns shown in Equations 1.1-1.3, a cache policy that manages based on preserving the most recently accessed items will perform well for 1.1 and poorly for the others. Conversely, a policy that preserves the least recently accessed items will perform well for 1.2 and poorly for the others. The memory content, in terms of size and pattern, accessed by the overarching algorithm directly affects the cache policy performance. This content is formally called the working-set [12].

Preserving the working-set is the general goal of the cache policy. Optimal caching is given by MIN (minimum) [8] and OPT (optimal) [22]. To achieve the best performance the items with the nearest future use should be cached. Although optimal, it is not practical because implementation requires future access information, i.e. a level of clairvoyance. Nevertheless, policies such as Hawkeye [17] still aim for OPT-like performance, by way of approximation.

OPT is viewed as *prescriptive*, in that cache management is externally prescribed. Conversely, *reactive* policies can self-manage. Basic policies like Least Recently Used (LRU) are implemented using simplistic stacks [22], while more modern policies use more complex architectures and theories for management. Enhancements of LRU such as ARC [23], Talas [6], and RRIP [18] decrease susceptibility to non-recency by cache partitioning and dynamic management. Such policies are significant improvements over earlier ones that require performance tuning, such as SLRU [24], or are particularly sensitive to parameter selection as with GD* [19]. In more niche applications like picture archiving, augmenting LRU with data mining and logistic regression has proven successful [34]. To this point, caching can simply accommo-

date performance hints to improve management, also known as *collaborative* caching [9, 16].

Caching is not restricted to be performed based purely on recency, frequency, etc. LHD [5], EHC [33], and LACS [20] use alternative metrics for evaluation and eviction (hit density, hit count, and caching cost respectively). Policies that use ranking functions such as EVA [7] and LHD [5] can outperform LRU and can avoid/reduce the impact of its common performance cliff access patterns (thrash). Additionally, these policies are feasible for both hardware and software caches (LHD and EVA respectively), making them highly applicable to general cache use.

1.2.2 Software-Driven Management

Prescriptive policies are realizations of software-driven cache management, while collaborative policies are software-augmented. Compiler management of the register file [11] is a prime example of how software-driven memory management is performed today. Caches have similarly been examined in this context [35], and there is an increasing desire to incorporate static analysis into cache policy [32]. OPT is achieved by retaining items with the most immediate need. Duong et al [13] formalize this into a concept known as *protection distance (PD)*. PD is essentially a lifespan for a cached item. An item will remain in cache until the end of the PD, and then it is removed. Their implementation of it, known as Dynamic Reuse Distances (DRD), evaluates and tunes PD at run-time. Optimal Steadystate Lease (OSL) [21] is essentially an offline implementation of PD. OSL analyzes memory access behavior at the page level, and assigns PDs, known as *leases*, accordingly. This requires trace level knowledge, which is not available at the program level, so OSL cannot be used by compilers, without back-annotating trace information.

1.3 Objective

Although optimal, MIN [8], cannot be practically realized. Techniques like Hawkeye [17] approximate this, but neglect software optimization. OSL [21] optimizes

cache performance via software, but is not hardware feasible and cannot be compiler-driven. Separate optimization of cache hardware and program structure ignores what can be an avenue for significant improvement, yet doing both is not traditionally possible. This work studies and implements a new prescriptive caching policy which aims to bridge the gap between OSL, static memory management, and cache hardware by using a software-hardware interface for program-driven cache control. This enables application-specific memory optimization, which ultimately improves execution throughput. This is particularly true for large and iterative workloads [15], where a slight improvement in memory management can elicit significant improvements in kernel performance. The significant contributions of this paper are,

- The first hardware implementation of this prescriptive caching policy, *lease cache*.
- Hardware support systems for lease assignment and lease cache performance evaluation.
- A novel spectrum-based tool for lease cache tenancy analysis.

Chapter 2

Background

The optimal replacement policy for fixed size caches is one where the the data object with the furthest future reuse is evicted, as given by MIN and OPT [8, 22]. While optimal, it is not feasible to dynamically forward predict reuses. OSL [21] introduces the concept of lease cache, where at every access the data item is assigned a lease, and remains in cache for the duration of that lease. Although OSL is shown to, at minimum, match OPT performance, it assigns leases per data page and for variable sized caches, neither of which make it practical for hardware implementation. In this chapter an extension of OSL is presented that targets fixed sized hardware caches using reference leases. A general theory on lease caching is presented along with algorithms for assigning leases.

2.1 Reuse Interval

The reuse interval (RI) of a data object is the elapsed time between two accesses to the item. In the context of this paper, time is defined as *logical time*, or the memory access trace length. It is a deterministic measure of how immediately a data object will be reused (it carries no information about first access). If an object is accessed at trace lengths of 2 and 7, then the RI of that object is 5. Similarly, if an object is accessed at traces of 2, 7, 20, and 25, the RI distribution of the object is $\{5, 5, 13\}$. Distributions are useful for describing how RIs deviate over the trace (logical time),

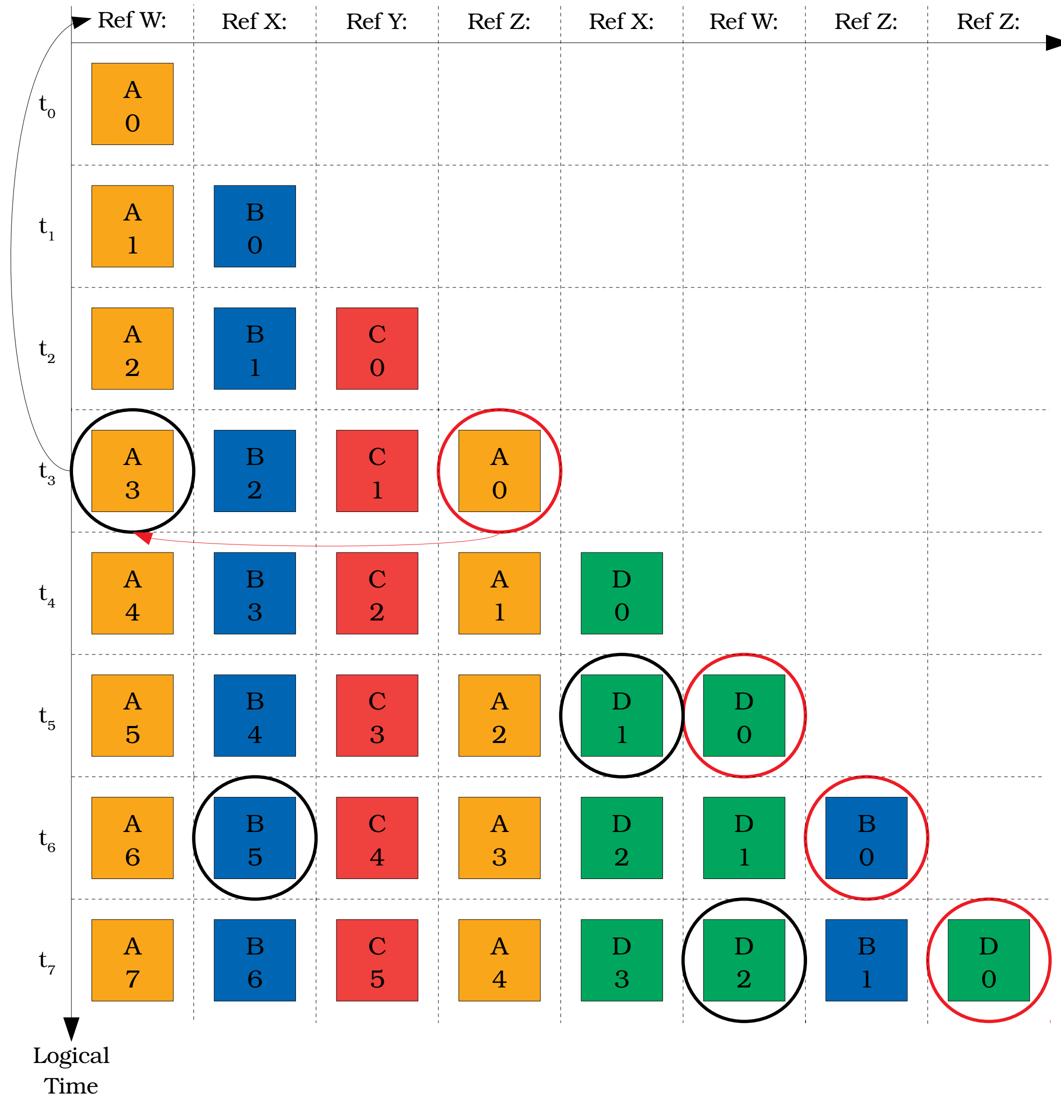

Figure 2-1: Forward reuse interval calculation for an arbitrary access sequence. Red circles indicate the re-access of an item whose original access is circled in black. The first re-access includes directional arrows to show how the reuse interval calculation traces back to the reference that is associated with the interval.

and provide information about the locality of the accesses [37].

Access based reuse intervals are introduced by Li et al [21]. These evaluations are made at data page level and are used in dynamic cache allocation (require trace length for assignments), which a compiler cannot use. Alternatively, RIs can be described statically or at the reference level of abstraction. While dynamic RIs are associated with a specific access, a static RI is associated with the reference instruction that results in the access and does not change over the length of the trace. An example of

this is shown in Figure 2-1.

The first and second accesses to data object A occur at t_0 and t_3, resulting in an RI of 3. This RI is associated with the reference that made the first access to the pair, reference W. The interpretation of this is as follows: after reference W accesses object A, it is known that the object will be accessed, or reused, either by reference W or some other reference in three accesses. Similar to access level RIs, reference level RIs can be defined by a distribution. Object D is first accessed at t_4 and reused at t_5, resulting in a reuse interval of 1 being associated with reference X. At t_7 object D is again re-accessed by a different reference, yielding a new RI of 2 (RIs are relative to the direct preceding access). This RI is associated with reference W, and does not affect the previous RI associated with reference X. RIs that project reuses ahead of time are called forward reuse intervals. For the purpose of this paper the term forward reuse interval is synonymous with RI.

A more practical example of a five point stencil (Listing 2.1) is used for subsequent discussion in this paper. In this example RIs are associated with high level programming language (HLPL) references, but there is no difference in concept between that and an instruction set architecture (ISA) level reference. The resulting RI distributions are given in Table 2.1. The combination of nested loop and index offsets in the stencil formula result in both immediate and long term reuses. There are also two references without an observed reuse. This example has characteristics of all three access patterns: recency-friendly (immediate reuses), scan (no reuses), and thrash (long term reuses) if the array dimension exceeds cache capacity.

Listing 2.1: Example five-point stencil program.

```
for(i=1;i<1024;i++)
    for(j=1;j<1024;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

Table 2.1: 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 [27].

2.2 Lease Cache: CLAM

Lease cache is a new prescriptive cache interface where at every access a lease is assigned to the accessed object by the program. The lease is a protection mechanism for the object in that it acts like a lifespan. The object remains in cache for the duration of the lease, and is evicted at the end of the lease term (lease *expiration*). In this way lease cache is prescriptive - when accessed the program allocates a specific amount of cache for the object and when expired the space is de-allocated. When there is a cache miss, any expired cache line can be used for replacement.

This variant of lease cache differs from OSL by assigning leases to references (ISA level memory instructions), called *reference leases.* At every access, the reference lease of the instruction accessing the object is applied to the object (*lease assignment or lease renewal*). For example, if reference X and W of Figure 2-1 have reference leases of 1 and 2 respectively, then data object D is assigned a lease of 1 at t_5 and 2 at t_6. Assigning any smaller lease results in the object's lease expiring before its next reuse. Conversely, assigning any larger lease results in over-allocation of cache (using more cache resources than is required, limiting the possible resources other objects can use). The relationship between lease and RI is apparent; when the two are equivalent an object is minimally safe in cache until its next access and subsequent lease assignment. In deciding what reference leases to assign the compiler can control how cache memory resources are utilized. This is called *Compiler Lease cAche Management* or CLAM.

CLAM is an extension of OSL with improved characteristics for hardware caches.

Both make lease assignments based on reuse interval distributions, but CLAM does so statically at the reference level. Any reference can access any object, so basic properties extend to more complex programs and patterns (OSL reference groups are limited to one target). Because OSL requires a lease per data page it is potentially less scalable than CLAM (number of references can be many magnitudes less than the number of data pages accessed). Furthermore, CLAM considers accesses at cache block granularity, rather than at page level, which makes it applicable for low level hardware caches.

2.2.1 CARL: Compiler Assigned Reference Leasing

Reference leases are assigned by the compiler using an algorithm called Compiler Assigned Reference Leasing, or CARL [10]. CARL assigns reference leases with the highest benefit until a target cache utilization budget is reached. CARL is designed for variable sized caches (network, unified, or shared caches), meaning that it is not absolutely constrained by cache size. Instead, the optimization budget is the product of trace length (time quantity) and target cache size, an aggregate value. When averaged across the trace length the budget is the target cache size, but the instantaneous cache utilization is allowed to deviate from that, hence it is an algorithm for variable sized caches.

The cost and profit of a specific lease assignment, l, is determined by the RI distribution or histogram of the references. For simplicity assume RI distributions are represented as vectors, H, where the index and value refer to the RI and number of accesses with that RI (RI count) respectively. Lease profit is equivalent to the number of cache hits that result from a specific assignment. The profit (2.1) is then the number of accesses in H whose RI is less than or equal to the assigned lease, l.

$$\text{Profit}(l, H) = \sum_{i=0}^{l} H[i] \tag{2.1}$$

The cost associated with a specific lease assignment is the overhead resulting from the lease, i.e. how much cache space does the assignment allocate. Unlike lease

benefit, the cost of a lease is directly dependent on the value of the lease assignment. All RIs less than the lease will only incur cost until they expire or are renewed, while all RIs greater than the lease are limited by the selection of the lease. The first summation of Equation 2.2 is the cost associated with RIs less than the lease, while the second summation is the complementary condition.

$$\text{Cost}(l, H) = \sum_{i=0}^{l-1} i * H[i] + \sum_{i=l}^{RI_{max}} l * H[i] \tag{2.2}$$

CARL's objective is to maximize the profit per unit cost (PPUC) of assignment. CARL essentially looks at all references, and assigns the lease that provides the largest PPUC, regardless of past assignments. CARL is a greedy algorithm and hence ignores coverage - i.e. it will assign the highest value lease regardless if a lease has already been assigned to the reference. Lease assignment for the stencil (Listing 2.1, Table 2.1) is shown in Figure 2-2. In this example an arbitrary budget of one billion is used and PPUC is expressed as a marginal value to show the benefit in iteratively updating lease values from prior assignments.

The first observation from the assignment process is that not all references are assigned a lease. The target budget is exceeded at iteration 3 causing all remaining references to be assigned a zero lease. Since the assignment at iteration 3 exceeds the budget it also is not assigned; however, there is still residual budget. In order to utilize this an extension to CARL, *dual leases*, is introduced in Section 2.2.2. The second noticeable outcome is that the large PPUC of $a[i+1]b[j]$ does not result in an assignment of that lease. In order to consider this lease the preceding assignment (of a smaller lease) must first be made. The PPUC of this lease is not the largest of all possible candidates, so CARL does not make the assignment.

2.2.2 CARL Extensions

Dual Leases

Assignment by CARL may result in unused residual budget due to further assignment incurring more cost than there is remaining budget. In this case the cache is under-

allocated, which negatively impacts performance. To make use of this budget portion CARL assigns a *dual lease*. A dual lease is a lease assignment that can result in two possible values. Which of the two are assigned at an access is controlled by an assignment probability. The goal of these assignments is to interpolate the cost of assignment so that it equals the remaining budget. For example, if lease assignment $l_{overbudget}$ exceeds the remaining budget by a factor of two then $l_{overbudget}$ is only assigned for half of its accesses. The remaining half are assigned a lease of zero so that the budget is not exceeded. Dual leases are functions of the RI distribution while the assignment probability, p, is given by Equation 2.3. For the stencil (Figure 2-2) lease assignment would be terminated at step 3 because further assignment results in over-allocation by a factor of roughly 6.5. With dual leasing CARL assigns the red item with a probability of 15% so that the budget is not exceeded (lease of 4 is assigned 85% of all accesses, lease of 6128 is assigned the remaining 15%).

$$p = \frac{\text{remaining budget}}{\text{Cost}(l', H) - \text{Cost}(l, H)} \tag{2.3}$$

PRL: Phased Reference Lease

CARL by design is for variable sized caches. While not guaranteed, it is likely that for specific types of programs and patterns CARL may under-allocate or over-allocate cache at specific points in time. An over-allocated cache, from a lease definition, is one where no lease is expired when an eviction is needed. There is no clear eviction choice in this situation, and arbitrarily evicting a line would most likely incur some performance penalty, as all lines have non-zero leases (will be re-referenced). This type of over-allocation stems from CARL allowing the lease budget to deviate from the average along the entire trace length. The solution to this is to examine the trace in phases, evaluating budget utilization in each. This is called phased reference leasing, or PRL.

PRL splits the trace into n phases and associates a local allocation budget of $global_budget/n$ and local RI distribution to each phase. Leases are assigned based on the maximum global PPUC across phases, like CARL. When assigned, each phase

incurs a cost based on its respective RI distribution. Leases are assigned in the same manner as CARL until a phase's budget is fully utilized, whether by dual lease or otherwise. From this point on (to algorithm termination) no other assignments can be made that would increase the cost of the fully utilized phase. Although PRL is described in terms of phases it also extends to set associativity. Instead of splitting a trace into temporal phases it is split into access groups or spatial phases, and leases are assigned in the same manner.

FUL: Fixed Uniform Lease

A program may not be amenable to performance optimization by CARL or PRL. This can be due to compiler restrictions (lack of ISA or hardware clairvoyance) or program structure (small or very complex programs would not elicit significant benefit). *Fixed uniform lease*, FUL[26], is a safety feature extension of CLAM for these cases. As the name suggests FUL assigns one static lease for the entire set of program references. If the appropriate lease is chosen, then LRU equivalent performance is achieved. Thus, FUL can be seen as a safety feature of CLAM in that LRU performance, at minimum, is guaranteed. FUL assignments can be generated in the same manner as variable leases; however, for this work is empirically studied (by parameter sweeping the uniform lease value). FUL may also be used to emulate other cache policies through alternative lease selection/s.

Iteration	Reference	RI	RI Count	PPUC	Accumulated Budget Utilization
0	a[i][j]	7	1,043,462	0.1427175	0.73%
		6135	1021	1.63185(10^{-4})	n/a
	a[i][j+1]	4	1,043,462	0.2500000	0.42%
	a[i][j-1]	6128	1,043,462	1.63185(10^{-4})	639.43%
	a[i+1][j]	6124	1,043,462	1.63185(10^{-4})	639.64%
		6128	1021	0.1427	n/a
1	a[i][j]	7	1,043,462	0.1427175	0.73%
		6135	1021	1.63185(10^{-4})	n/a
	a[i][j+1]	4	1,043,462	0.2500000	n/a
	a[i][j-1]	6128	1,043,462	1.63185(10^{-4})	642.11%
	a[i+1][j]	6124	1,043,462	1.63185(10^{-4})	642.32%
		6128	1021	0.1427	n/a
2	a[i][j]	7	1,043,462	0.1427175	n/a
		6135	1021	1.63185(10^{-4})	.74%
	a[i][j+1]	4	1,043,462	0.2500000	n/a
	a[i][j-1]	6128	1,043,462	1.63185(10^{-4})	646.86%
	a[i+1][j]	6124	1,043,462	1.63185(10^{-4})	647.07%
		6128	1021	0.1427	n/a
3	a[i][j]	7	1,043,462	0.1427175	n/a
		6135	1021	1.63185(10^{-4})	n/a
	a[i][j+1]	4	1,043,462	0.2500000	n/a
	a[i][j-1]	6128	1,043,462	1.63185(10^{-4})	646.86%
	a[i+1][j]	6124	1,043,462	1.63185(10^{-4})	647.07%
		6128	1021	0.1427	n/a

Figure 2-2: CARL assignment procedure for a budget of one billion. Green items indicate a lease assignment made at the current step. Blue indicates a lease assignment previously made, which is contributing to the overall cost. Red indicates assignment termination due to the budget being exceeded.

Chapter 3

Lease Cache Hardware Design

This section presents the first functional hardware implementation of lease cache that supports all CLAM assignment policies. First, fully associative lease cache is considered, followed by an extension for set associative designs. The necessary hardware-software interface and software level support for the cache is then detailed. In addition to the cache hardware, two CLAM hardware support designs are also presented. One a reuse interval sampling front-end to CLAM, and the other a lease cache dynamics tracker. All three designs are considered the significant contributions of this work. Note that the objective of this work is functionality, not optimal hardware design. The designs outlined in this chapter are presented with generic parameters that are later defined in Section 4.1.3 as they relate to performance testing.

3.1 Lease Cache Implementation

A practical hardware implementation of CLAM must run in real-time without limiting the memory system, and its hardware-software interface must be simple enough to integrate into existing technologies. In theory, the reference assignments generated by CLAM are unbounded in quantity and value, yet they must be used effective enough such that CLAM is competitive with latencies seen in other caching policies. Similarly, any overhead from leasing must be justified by the resulting cache performance. As with traditional caching, lease case must be transparent to the processor

and user, otherwise one of the main benefits of caching is eliminated. The hardware and software considerations for CLAM are discussed in sections 3.1.1 and 3.1.2 respectively.

3.1.1 Hardware

Lease cache hardware is divided into two stages,

1. Lease lookup - hardware necessary to translate memory access information to lease assignment information; performed in parallel with cache access index/address translation.

2. Lease update - hardware necessary to modulate/control active leases and generate an eviction victim; occurs after lease information is generated for an access.

In this way the lease cache can be viewed as a general two stage pipeline. The dedicated lease components are built around existing cache infrastructure (traditional communication buses, request-service sequencing/control, pipeline stages, etc.) so that comparisons between lease and other policies are consistent. They in fact can be viewed as system augmentations; the cache can still function under an auxiliary replacement policy (discussed in Section 3.1.1) if leasing is disabled. CARL, PRL, and FUL all utilize the same component structure detailed hereafter.

Lease Lookup

Lease lookup is the stage of lease cache responsible for generating lease information and control signals based on a memory access request. It is implemented in parallel with the cache target address translation (Figure 3-1), in a similar manner. When an access is requested, the core provides an additional field, the reference address (instruction address) of the access. The reference address is searched for in a lookup table called the lease lookup table (LLT), which contains four fields,

1. *Reference Address* - the search parameter/input of the table. It is the address of the instruction (load/store) responsible for the memory request. This field

results in a match status bit output.

2. *Long Length Lease* - if assignment is a dual lease, this field is the longer lease assignment of the two. Otherwise, it is the sole lease assignment generated by CLAM. This field is a direct output.

3. *Short Length Lease* - if assignment is a dual lease, this field is the shorter lease assignment of the two. Otherwise, this value is redundant. This field is a direct output.

4. *Long Lease Probability* - probability of assigning the long length lease upon an access to the table entry. If the table entry is not a dual lease this field stored in the table as a quantity equivalent to 100%. This field is a direct output.

LLT entries are consistently aligned such that finding a reference address match produces all associated fields of that reference without additional hashing (all have the same index). The direct outputs of the table are used to multiplex dual leases while the match bit controls lease assignment.

Dual lease output selection is controlled by a linear feedback shift register (LFSR) circuit. The LFSR output is compared against the long lease probability generated by the LLT at an access. If the LFSR output is less than or equal to the table probability, the long lease is multiplexed; otherwise the short lease is multiplexed. In cases of non-dual leases the long lease is guaranteed to be assigned by fixing the probability field to 100% so that the LFSR output never exceeds the probability field. While leases are integer values (can only have integer reuse intervals), the lease assignment probability is a floating point number. In order to store and use this in hardware the value is discretized according to the bit width of the LFSR (with adjustments for the unachievable space of the LFSR sequence - i.e. zero). The size required to sufficiently discretize the probability values for the LFSR design is dependent on the program being examined; however, this parameter can be constrained to a specified uniform resolution and evaluated by CLAM (in theory - not a current feature of CLAM). The correct adaptation of this is to floor the decimal probability value to the nearest discrete value and incur this as a performance reduction due to under-allocation.

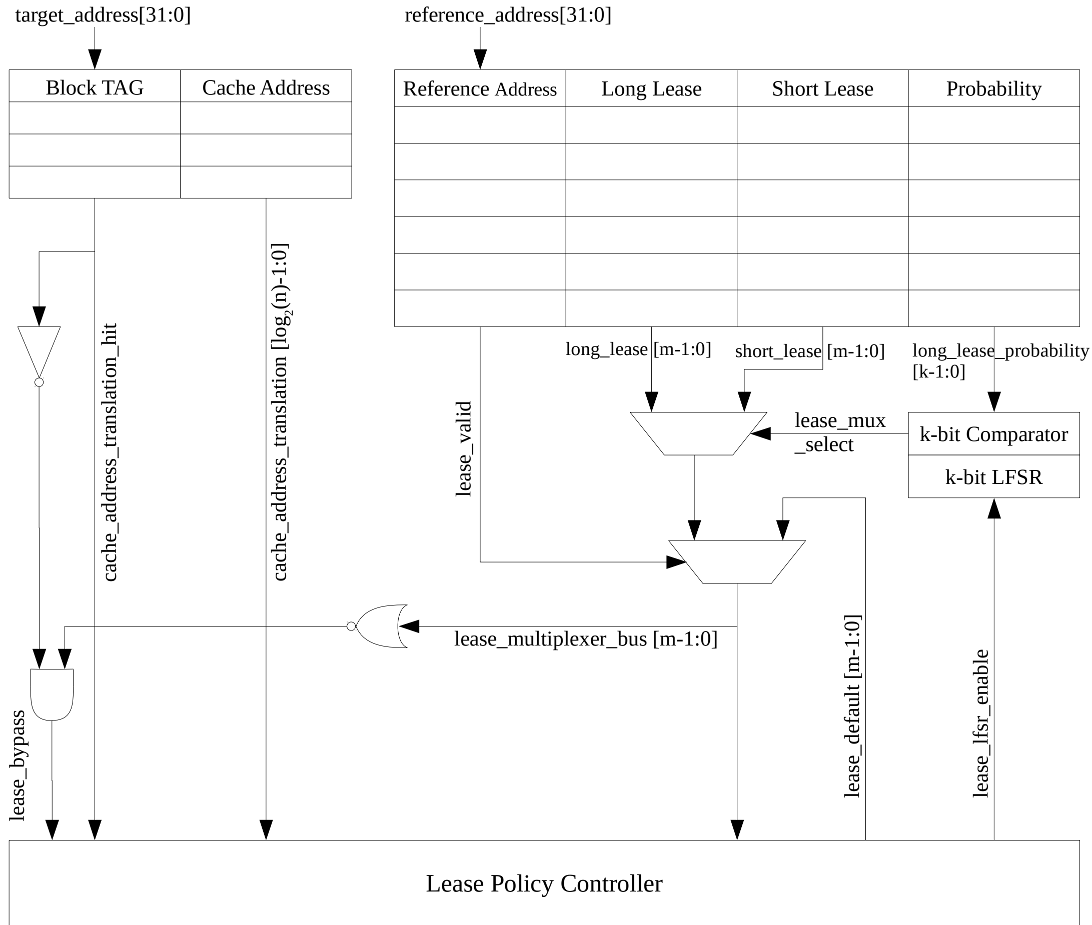

Figure 3-1: Lease cache lookup stage hardware overview for fully and set associative cache. Component and signal sizings are given in Section 4.1.3. Note that *lease_multiplexer_bus* is driving a NOR reduction gate.

A secondary multiplexer after the LFSR validates the lease assignment. If the reference address is not found in the LLT during translation (LLT miss) the lease assignment is overwritten by a default value, which is stored as a software configurable register. The default lease serves the primary purpose of accounting for unmatched references; however, this may also be used to reduce the number of LLT entries needed by grouping together similar leases, a hardware resource specific detail. Given an infinitely sized/scalable table and CLAM clairvoyance of program execution the default lease is unnecessary. For an embedded system the LLT can be sized according to the workload; however, CLAM clairvoyance requires that the application sequence is deterministic enough for the compiler to analyze. This issue of program clairvoyance

is addressed in Section 3.2.

Section 2.2.1 describes how a cache is partitioned by CLAM. In cases where no lease can be assigned due to the limited budget, all remaining references are assigned a zero lease so that they do not impact assignment cost. Dual lease assignment is similar in that either the primary or secondary lease can be a zero lease. The hardware implementation of this is known as a *zero lease bypass*. It is initiated when a request results in a cache miss, and the item is to be assigned a lease of zero. When this occurs the request is serviced, but not cached so that the working set is preserved; otherwise, this results in the forcible eviction of a cached item in order to allocate an item of no benefit, i.e. zero lease, reducing performance. The occurrence of forcibly evicting a cache line, regardless of its current lease value, is discussed in Section 3.1.1.

Lease Update

The lease update stage is where all active leases are maintained and updated. Each cache line is associated with a lease register (Figure 3-2) that holds the line's current lease. At every access, the cache line targeted by the request loads its lease register with the lease assignment generated by the lookup stage, while the lease registers of all other cache lines are decremented, as long as they are non-zero. A cache (not to be confused with LLT) miss necessitates that the stage generates an eviction victim based on the state of the active leases. There are three possible eviction conditions to consider,

1. 1 expired lease - the trivial eviction case; the sole expired cache line is selected for eviction.

2. 2+ expired leases - the lowest address expired cache line is selected for eviction.

3. 0 expired leases - there is no eviction candidate according to lease policy, replacement follows an auxiliary policy.

A priority encoder (Figure 3-2) is used for victim selection when there is at least one expired cache line. Each lease register drives a logical NOR reduction gate to

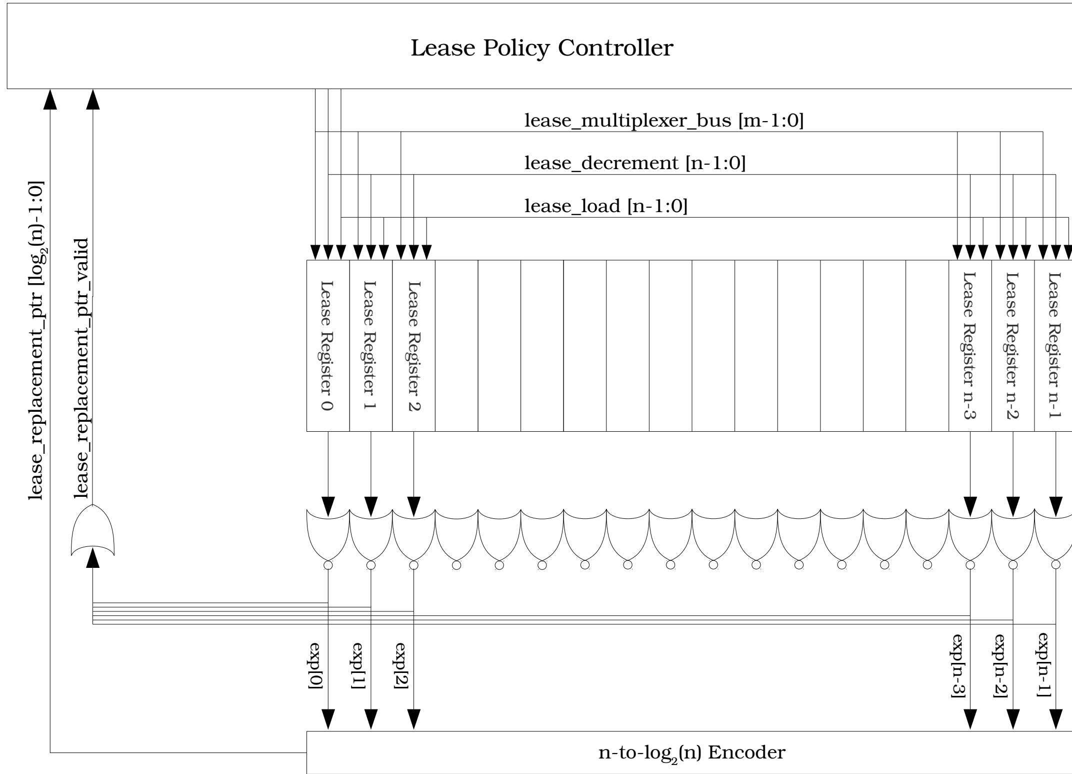

Figure 3-2: Lease cache update stage for fully associative cache. Component and signal dimensions are given in Section 4.1.3. Note that *exp* bus drives an OR reduction gate.

create an expired flag (logic high if every bit of the lease register is logic low). The bus of expired flags subsequently drive a priority encoder which generates a pointer to the cache address of the first (least significant bit priority) expired lease register. The pointer is verified (necessary due to default encoder output) by taking the logical OR reduction of the expired flag bus. If at least one flag is logic high the replacement valid flag is driven high. In this way the lease encoding scheme can be efficiently used to determine the trivial case victim, the multi-expired lease victim, and can identify when there is no expired lease. Additionally, there is no theory to suggest that an alternative method of victim selection, for the multiple expired lease case, would result in cache performance improvement. It is shown in Section 2.2.1 that it is possible to assign a reference lease that does not cover the entire RI distribution of a reference due to budget limitations. In this scenario it would be beneficial to replace the "most

expired lease", or the cache line that has been expired for the longest amount of time. Tracking this is unrealistic from a hardware view (additional counter per cache line and comparison/stack circuit). Additionally, there is no guarantee that this is the case for any lease, nor is there a way to predict this in hardware (without additional information encoded in the LLT).

If there is no expired lease (replacement valid flag seen as logic low), the cache is over-allocated and the lease policy cannot identify a victim. An auxiliary replacement policy, random replacement, is then used to choose the victim. Random replacement is used for several reasons,

- The LFSR from the lookup stage is efficiently reused.
- Regardless of the selected victim, a penalty is incurred due to early eviction.
- This eviction scenario is thought to be irregular or uncommon. It is assumed that there will be an expired lease when an eviction is required, according to CLAM theory. CARL budget averages to the target cache size so this may be proved incorrect for certain types of program/patterns.
- Due to a finite budget leases may not be indicative of the actual reuse interval.

MIN [8] states that optimal replacement is when the furthest reuse item is evicted. From this viewpoint evicting the largest active lease is best. CLAM however, can irregularly assign leases based on PPUC so there is no guarantee that the largest active lease in fact correlates to the longest reuse interval. Again, from a hardware view the circuit required to identify the largest lease comes at unreasonable cost, especially when a performance penalty will be absorbed regardless of choice. Section 5.1 further supports use of random policy.

Figure 3-2 depicts the update stage for a fully associative cache. The expired flag of every lease register drives the same encoder. A set associative cache can use the same general concept to determine local (set) replacements (Figure 3-3). Each set drives a unique encoder and reduction gate. The output buses of each component are multiplexed by group (sub field of target address) which yields the appropriate

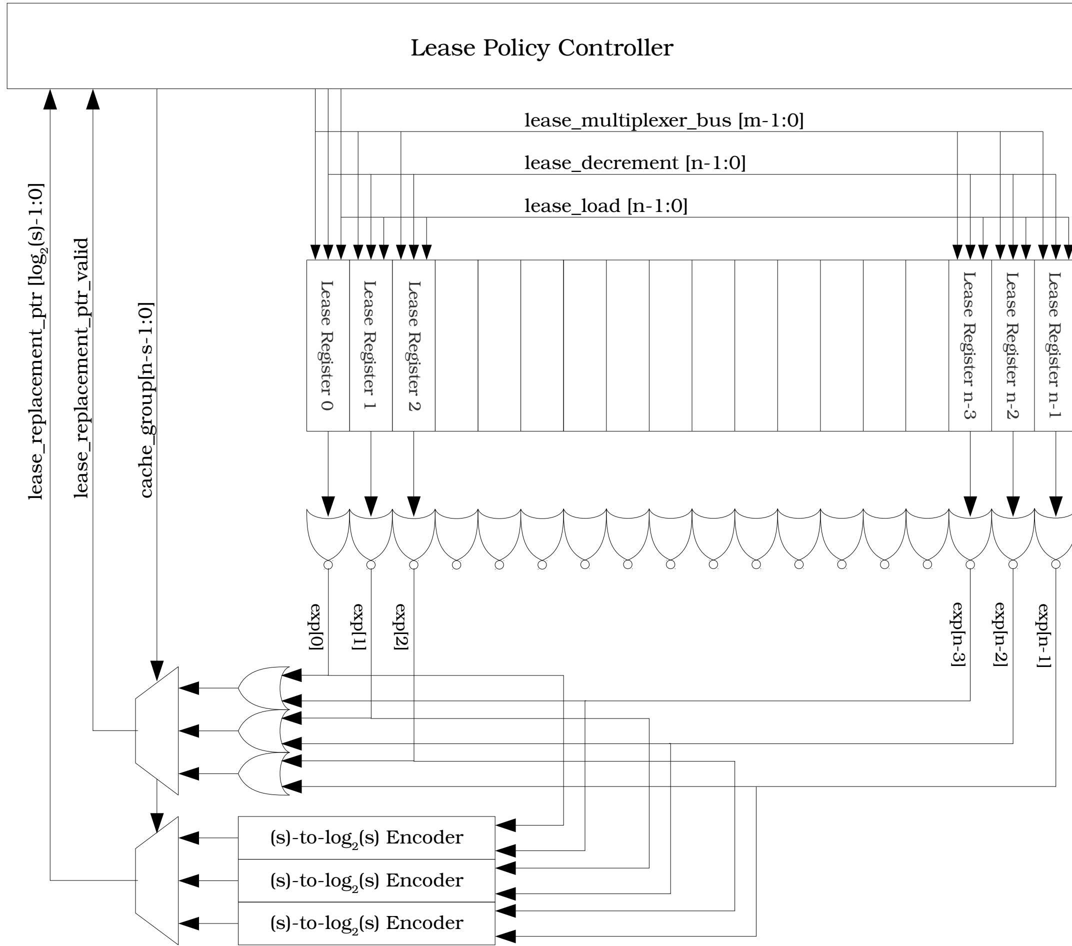

Figure 3-3: Lease cache update stage for a generic four way set associative cache, with update circuitry shown for three sets. The *exp* bits of each set drive an OR reduction gate in the schematic. Component and signal dimensions are given in Section 4.1.3. Parameter s is set size.

set replacement pointer and validation flag. The lookup stage structure for set lease caching is identical to the fully associative hardware shown in Figure 3-1.

3.1.2 Software Support

The architecture presented in Section 3.1.1 assumes that the LLT is populated with all run-time lease information, and provides no mechanism for managing it. One of the objectives of the lease cache implementation is to be transparent to the user, and not require extraneous interaction. In support of this, lease information is embedded in the application binary. Other than the cost of managing and storing more data,

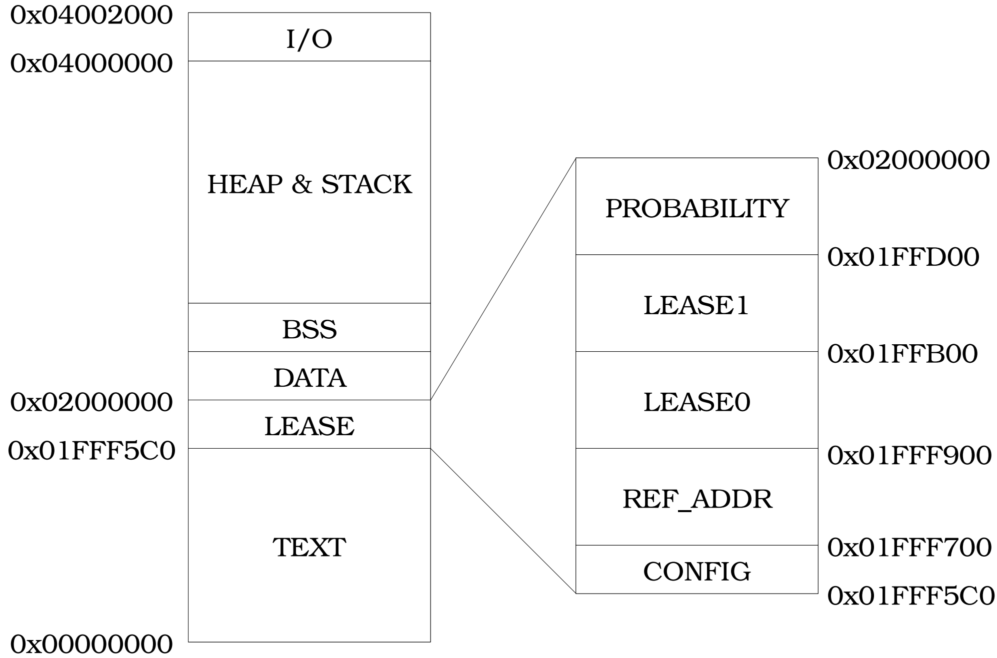

Figure 3-4: Mapping of lease cache memory partitions in application binary. Addresses are byte addressable.

this is a robust way of communicating lease cache information invisibly from software to hardware.

Programs are compiled to executable and linkable formats (ELF), a common Unix format. Figure 3-4 shows how lease cache information is linked in this format. LLT data along with cache configuration information is placed at the end of read only memory. It is sized such that the the benchmarks discussed in Section 4.3 can be allocated in the partitions along with configuration data used in lease cache and the hardware discussed in Section 3.2. The lease cache controller contains an internal read only register initialized with the starting address of the configuration partition. Out of reset the cache controller requests the first block of lease configuration data. The content of this block provides LLT initialization information to the cache such as the number of entries to write into the table. The cache controller then imports the respective data from main memory to the LLT. This is referred to as *static lease caching*.

LLT population by static caching is not considered to be performed at run-time,

since it occurs as a preprocess out of reset. It is however, directly extendable, in terms of overhead. For an LLT with 128 entries, 32 block transfers are needed to fully populate the table (1 LLT entry is 4 fields which is one-fourth of a block). Current versions of CARL and PRL only produce one dual lease entry so the minimum number of required transfers to populate the table is reduced to 17 blocks. For a given workload, if leasing decreases the cache miss count (as compared to another policy) by the block transfer overhead associated with filling the table, use of lease cache is justified. Static lease caching incurs no run-time overhead to maintain the LLT.

3.2 Hardware Support for CLAM

Compiler driven analysis of reuse intervals is the basis for CLAM. In Section 2.1 the five point stencil (Listing 2.1) is used to illustrate this. Compilers have the ability to examine the HLPL of an application and analytically determine the resulting RI distribution, as shown in Table 2.1 by the symbolic references. The issue with this representation of references and RIs is that hardware lease cache requires binary mappings (memory addresses) for lease lookup/translation. CLAM must have both the ISA support for the target environment and the ability to track how the intermediate symbols translate to binary instructions (which first requires ISA support) in order to be of practical use.

The current version of CLAM, which is discussed in this paper, does not have ISA specific support. The proposed solution to this is a hardware front-end that generates binary mapped RI distributions by profiling the target application. CLAM then evaluates these statistical RIs, and generates leases through a CARL/PRL implementation. This is referred to as *hardware reuse interval sampling.*

The hardware sampler is essentially a snooping agent placed on the communication bus between the cache and processor core (Figure 3-5). The sampler continuously monitors bus signals, and periodically samples/records transaction data, specifically the reference address and target address. After sampling an item it looks for a re-access of the same item (to calculate a reuse interval), and when one is seen stores

all reuse information about the item. The sampler has three main components,

1. Sample table - small capacity lookup table that holds all sampled data without an observed reuse.

2. Reuse interval counters - maintain the active/running reuse interval of each entry in the sample table. The counter increment at every observed memory access.

3. Reuse interval buffer - large capacity memory that stores all sampled data with an observed reuse. The buffer can be accessed by external hardware.

The communication bus is sampled at random variable sized intervals using an LFSR to generate the sequence of intervals. Random sampling is chosen over fixed interval sampling in order to improve sampling coverage (fixed interval sampling runs the risk of continuously sampling the same reference and ignoring others). When sampled, the communication bus's current values are written to the next open location in the table. A sample remains in the table until either: a reuse of the target address is seen (reuse interval found), or the table reaches capacity (no unused table addresses). Unused table locations are kept track of by an address stack (when writing to the stack is popped, when evicting from the table the address is pushed).

When a table entry reuse is seen the entry is evicted from the table and written to the buffer along with its associated reuse interval counter value and the current trace length. The objective of the eviction is to minimize table utilization due to the table's finite size. Ideally, the sampler would observe at least one reuse for every possible reference, for the application being profiled. It is not possible to track all memory accesses (in most cases), and so once a reuse is identified the entry is removed. This action frees up the table location for a new sample, which is likely to be different from the sample that was just removed, presumably increasing the coverage rate of references with an associated reuse interval.

There are two causes for the table reaching capacity: the table is not sized proportionally to the working-set (specifically with respect to array sizes) and/or the table

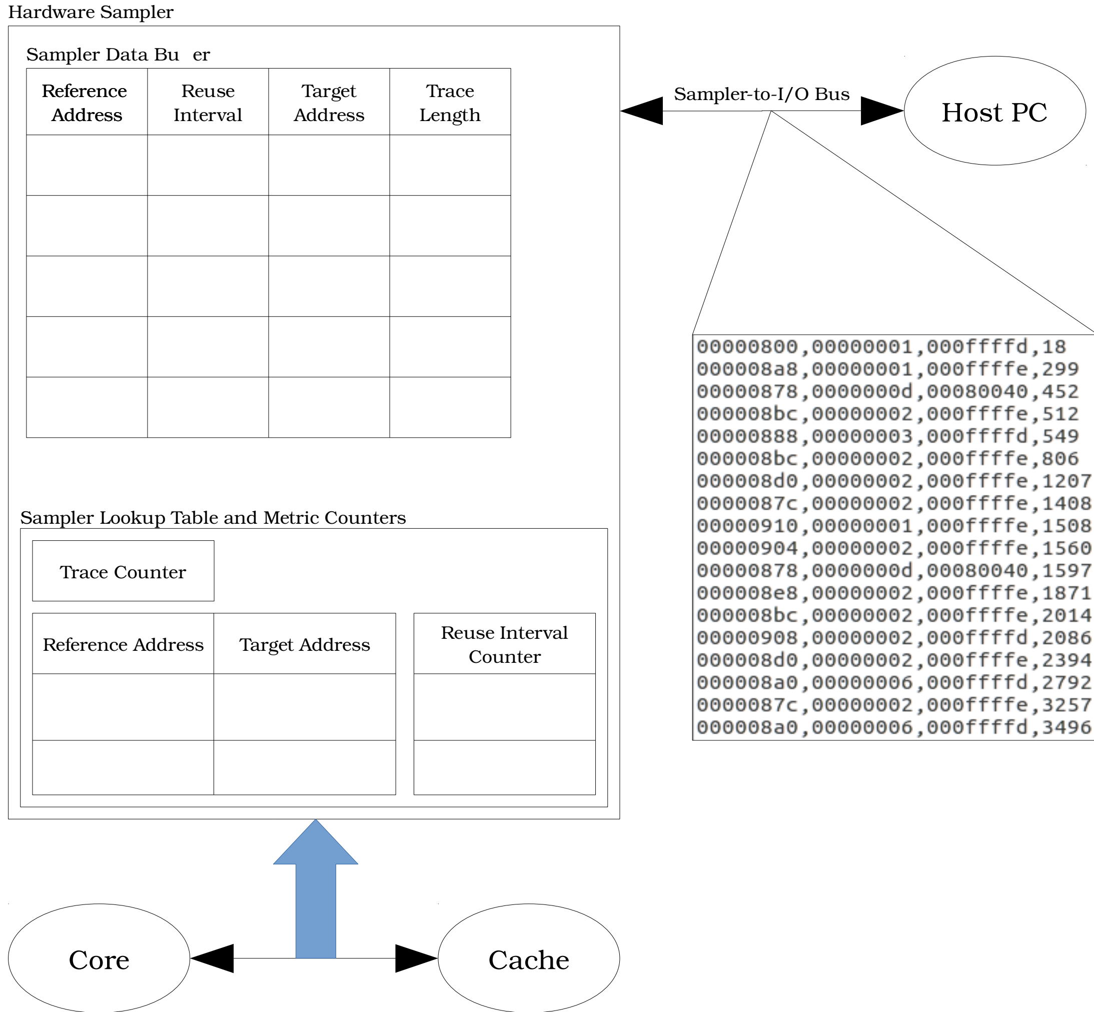

Figure 3-5: Hardware reuse interval sampler system overview. Delimited text file shown is output read from the sampler with same field order shown in the buffer.

contains non-reuse entries (entries that do not have a future reuse - stream pattern characteristics). In these cases a table entry is forcibly evicted so that a new sample can be allocated. The sampler finds the entry with the largest active counter value, and removes it from the table. Although this is not a reuse, it is written to the buffer as a negative RI value to flag this occurrence for CLAM. When sampling is complete, all active table entries are likewise written to the buffer in the same format.

Comprehensive reference coverage is not guaranteed by sampling. There is no analytical method for determining a minimum sampling rate or any similar control parameter, nor is there a proof for what adequate coverage is considered. Moreover,

there is a direct trade-off between the sampling rate and range of reuse interval magnitudes that can be captured. The higher the sampling rate, the faster the sampling table reaches capacity (assuming program size > table size). Because the oldest entry is forcibly evicted when this occurs, only references with short reuses (RI < table size) can be measured. Due to this, sampling is heuristic and highly dependent on the program being profiled. Regardless, it is hypothesized that if sampled sufficiently, all unsampled references are of low benefit due to irregularly large reuse intervals or are part of a stream access pattern (no reuse). Being of low benefit, these references can be associated with any arbitrarily small default lease, such that they do not significantly impact CLAM assignment cost.

3.2.1 Lease Tracking

The state (current active values) of the lease register array is a direct indicator of cache utilization and performance, a topic discussed in Section 4.2. If many leases are expired the cache is under-utilized. Conversely, if no lease is expired the cache is over-utilized. In reactive and collaborative caches, even ones with assumed re-reference intervals [18], there is no concept of the cache state because there is no knowledge of when an item will be reused. Lease caching (being a prescriptive policy) however, enables one to evaluate the cache's effectiveness by examining the lease register array state. Real-time tracking of this can be used to quantify policy effectiveness and theoretically enables performance adjustments to be made concurrent to execution, such as repartitioning/sharing a network cache for other tasks.

Lease state tracking is possible using the same general infrastructure as the sampler (Figure 3-5). Instead of monitoring reuses the tracker simply records the lease register array state at fixed intervals. Comprehensive tracking of every lease register, down to one bit resolution, is costly to perform, manage, and store, so the state is octal approximated by Equations 3.1-3.4 as follows,

$$Approximated\ Lease[3] = |(Lease[31:24])2^{24} \quad (3.1)$$

$$Approximated\ Lease[2] = |(Lease[23:16])2^{16} \quad (3.2)$$

$$Approximated\ Lease[1] = |(Lease[15:8])2^{8} \quad (3.3)$$

$$Approximated\ Lease[0] = |(Lease[7:0]) \quad (3.4)$$

The lease register is subdivided into eight bit groups, each driving a logic OR reduction gate (similar to the NOR reduction array of the lease update stage shown in Figure 3-2). Each bit of the approximation is associated with a reuse quantification: future, near-future, near-immediate, and immediate (from MSb to LSb). If all bits of the approximation are logic low the lease it is derived from is expired. This approximation is a measure of how cache is allocated by the lease assignment. If tracked over a period of time cache effectiveness for a given application or pattern can be visualized. These quantities are further used in Section 4.2.

Chapter 4

Testing

While cache is an integral hardware of computer systems, it is not a standalone component. The stencil (Listing 2.1) program demonstrates how leases can be used for caching; however, a processing system is still required to execute the program, and generate the related memory accesses. Similarly, the manner by which the program is compiled, in terms of ISA and options, can significantly alter the execution sequence. In this chapter the test architecture for the described lease cache is provided, along with formal metric definitions for lease cache performance. A new visual tool for evaluating lease cache utilization is also presented.

4.1 Test System

The hardware test system (Figure 4-1) is implemented on a CycloneV-GT FPGA development board [4]. The system is comprised of a processor core, internal and external memory subsystems, and a communication and control subsystem which is interfaced to an external host computer. The lease cache, reuse interval sampler, and lease tracker systems all use the same bare-metal architecture and components. The FPGA hardware is responsible for processing and generating all data, in the case of the sampler and tracker, while the host PC is used for system control and data management. This section outlines the hardware subsystems that support the lease cache.

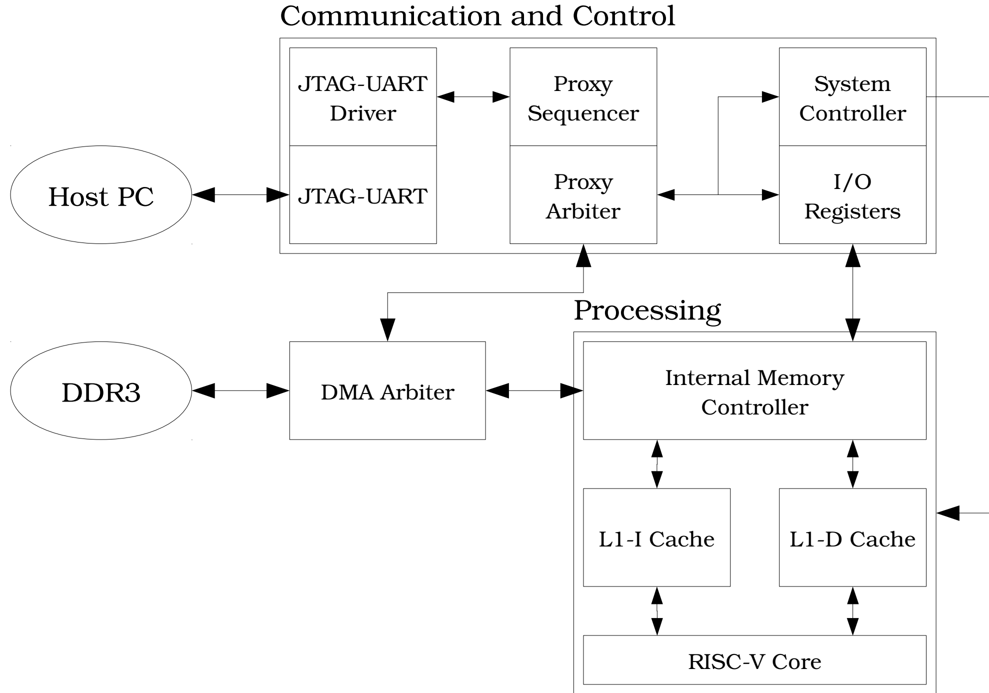

Figure 4-1: Top level test system diagram.

4.1.1 Processor Core

The processor core is designed according to the RISC-V ISA [36], and supports 32-bit integer base instructions as well as the 32-bit integer multiplication extension (dedicated multiplication and division hardware). The core features a six stage pipeline that uses fall-through branch prediction, and can be run with memory access latencies of either one or two cycles, to support cache pipelines. The cache is integrated with instruction and data buffers to the pipeline memory ports to accommodate this functionality. Additionally, data request ordering is maintained by the data buffer (in the case of multiple cycle access latencies), as requests to I/O addresses have a fixed one cycle latency. When running in single cycle latency mode the buffers are bypassed.

Table 4.1: Compiler options for RISC-V toolchain.

Compiler Option	Option Value	Description
-march	rv32im	Hardware system supports 32-bit base instructions as well as 32-bit integer multiplication and division extension.
-mabi	ilp32	’int’, ’long’, and pointers are all 32-bits in width.
-specs	nosys.specs	Specifies bare-metal compilation (no support for system calls). Program linked against ’libnosys’.
-mno-div	n/a	Generate code without using dedicated division hardware instructions. Legacy option from December 2019 FUL system.

RISC-V GNU Toolchain

Applications are cross-compiled for the core using the RISC-V GNU toolchain [2], which supports C/C++ languages. Programs are compiled into generic ELFs for newlib standard libraries, in support of the bare-metal/embedded execution environment of the FPGA system. A custom linker script links the compiled objects according to the memory segments shown in Figure 3-4. The complete list of compiler options used is given in Table 4.1. Note that applications for this work are compiled using a now deprecated variant of the toolchain, *riscv-none-embed*. This is a dedicated multilib toolchain for bare-metal applications and has since been absorbed into the *riscv-unknown-elf* distribution (set *-march* flag to include *rv32e* for equivalent settings).

4.1.2 Communication and Control

The system runs in a bare-metal environment and requires a supporting infrastructure for testing, control, and data acquisition. Previous versions of the system were emulated on a CycloneV-DE0 [31] which limited the complexity of the hardware system (resource constraints), and how it could interface with other devices (aside from an archaic PS/2 connector, it has no standardized input connection hardware). The lack of I/O capability was circumvented by using an existing JTAG port connection, which is used for configuring the FPGA through Intel’s proprietary software. This

subsystem is likewise implemented on the CycloneV-GT as a legacy design.

The hardware communication interface is comprised of a JTAG-UART circuit for transmitting and receiving packets, and a communication proxy that sequences the raw packets into a protocol, and communicates with other hardware blocks. The proxy is interfaced to,

1. External memory DMA controller - used to directly write binaries (received from the host) into external memory, DDR3.
2. Test controller - controls the processing system test and execution sequence (resets, transfers, etc.).
3. I/O registers - allows for direct communication with the sampler and tracker data buffers, RISC-V core, and other subsystems.

Test Sequence

The lease cache and sampler/tracker have similar operational sequences. Application binaries are sent, from the host, over the UART-JTAG interface, packetized by memory segments. Those packets are then sequenced into write commands and issued by the proxy. When the transfer is complete, the host pulls all remaining hardware subsystems out of reset and into execution. Depending on the system being run (lease, sampler, or tracker) the host polls different I/O registers to check either the application execution status or for a buffer interrupt.

When the main of the program is exited an I/O register is written to, signaling the event to the host so that a subsequent system command can be issued. When the host acknowledges that the main has been exited, all cache performance data is read. Each cache has a dedicated performance monitor which tracks hits, misses, etc. This hardware (and the analogous sampler and tracker circuits) is enabled by special I/O writes executed before entering and after exiting the kernel of the program, limiting the performance evaluation and data acquisition to it (the kernel). The pre-kernel entrypoint/bootload sequence has little cache utilization impact and is consistent for

all benchmarks and cache policies/systems tested, so the cache is not flushed prior to entering the kernel.

When the hardware sampler or tracker is enabled there is another level of polling required. In addition to checking the execution status, the sampler/tracker buffer interrupt flag is also read. This interrupt signals to the host that the buffer is full, and the core has been temporarily stalled. After reading the data from the buffer, the host clears the interrupt, which subsequently re-enables the core.

FUL is tested using the same interface and methodology as CARL and PRL; however, the best performing lease is not known a-priori. In order to determine this parameter the system is empirically parameter swept with leases of increasing value. The range of trialed leases spans zero to 65536 in 7-bit increments, as leases exceeding the range did not show further improvement for the benchmarks trialed. Aside from altering the lease, the FUL replacement policy configuration is also controlled (FUL was the first policy trialed so it has more variants). In the event of no expired lease the auxiliary policy is either random, eviction of the Largest Remaining Lease (LRL), or eviction of the Smallest Remaining Lease (SRL). When evicting by LRL or SRL the cache lines are randomly pooled. The cache line associated with the largest (LRL) or smallest (SRL) active lease (of the pooled subset), is then evicted. Pool sizes of four and eight are trialed for both SRL and LRL. Pool sizes are non-zero and smaller than the cache size because a zero pool size results in random replacement. Conversely, if every line is evaluated LRL and SRL default to MRU and LRU respectively.

4.1.3 Design Parameters

Sizing of lease cache hardware components is application specific. The main focus of the hardware implementation is functionality, rather than speed or efficient resource utilization. Similarly, the design is not meant to be universally applicable for all workloads (in terms of lease register sizing, LLT size, etc.). Nonetheless, embedded system hardware can be designed according to its narrow workload, making it an appropriate candidate for functional lease cache implementation. Table 4.2 lists the lease cache, reuse interval sampler, and lease tracker design parameters that are used

Table 4.2: Hardware design parameters for base cache, lease cache, reuse interval sampler, and lease tracker.

Design Parameter	Value
Sampler table entries	64
Sampler table fields per entry	3
Sampler table field width	32-bit
Sampler RI counter width	32-bit
Sampler trace counter width	64-bit
Sampler LFSR width	9-bit
Sampler average sampling rate	1/256 [samples/access]
Sampler clock frequency	20 MHz
Sampler latency to replace oldest entry	64 cycles
Sampler buffer entries	8192
Sampler buffer fields per entry	4
Sampler buffer field width (excluding trace)	32-bit
Sampler buffer trace field width	64-bit
Tracker buffer entries	4096
Tracker buffer fields per entry	4
Tracker buffer field width	128-bit
Tracker sampling rate	1/256 [samples/access]
Tracker clock frequency	20 MHz
Cache clock frequency	20 MHz
Cache access latency	1 cycle
Cache write-out to main memory latency	16 cycles
Cache read-in from main memory latency	minimum 17 cycles
Cache transfer throughput	1 word per cycle
Cache write-out buffer entries	1
Cache word size	32-bit
Cache block size	16 words
Cache data bus width	32-bit
Cache levels	1
L1-I capacity	8kB (128 blocks)
L1-D capacity	8kB (128 blocks)
Lease cache clock frequency	20 MHz
Lease cache victim selection latency	1 cycle
Lease cache LLT entries	128
Lease cache LLT reference address width	32-bit
Lease cache LLT primary and secondary lease width, m	24-bit
Lease cache LLT lease probability width, k	9-bit
Lease cache LLT data bus width	32-bit
Lease cache default lease	1 [-]
Lease cache lease register count (same as cache block size), n	128
Lease cache lease register width	24-bit
Lease cache LFSR width	9-bit

to generate the results presented in Chapter 5.

4.2 Cache Performance Metrics

Cache performance is evaluated most commonly by metrics derived from the number of misses incurred by the replacement policy. Lease cache, being prescriptive, can also be evaluated by how efficiently the cache is allocated. CARL and PRL assign leases until a target budget is reached, on average. Ideally for a fixed size cache, both algorithms would make assignments such that the lease cost never exceeds the budget at any point in the trace (i.e. cache block capacity is never exceeded). For this to occur at least one lease in cache must be expired at every access miss, so that the requested/missed item can be safely allocated. If however there is no expired cache line, the resources required by CLAM to cache that item exceeds the average, and the cache is said to be over-allocated. Conversely, having an expired lease at every miss does not strictly equate to optimal performance. An expired lease is associated with an item that has no future benefit, meaning there is no benefit to letting the item remain in cache. If, at a miss, there are multiple expired items in cache, one will be used for replacement while the others continue to serve no future benefit. This is synonymous with cache under-allocation.

While over-allocation is directly related to performance, under-allocation is a relative metric. Assume a cache that is significantly larger than the working set. In this case the cache will be significantly under-allocated; however, regardless of the replacement policy, performance is optimal. Lease cache is similar in that under-allocation resulting from a 'small' program or pattern does not necessarily correlate to reduced performance. For this reason several formal metrics for aggregated lease cache performance are defined,

1. *No Vacancy Ratio* - the ratio of evictions by auxiliary policy to the total evictions (lease + auxiliary).
2. *Multiple Vacancy Ratio* - the ratio of expired evictions to total evictions (by lease expiration or auxiliary policy, i.e. random replacement) when there are at least two expired cache lines.

The no vacancy ratio is a measure of over-allocation; the higher this metric the more common it is to make an eviction by auxiliary policy due to no expired lease. Conversely, the multiple vacancy ratio measures under-allocation. The higher this number the greater the probability of there being multiple expired items in cache at any given eviction. Vacancy metrics reflect cache performance as an aggregate average over a given time interval, in this case the trace length. Ideal allocation, as defined above, would be defined as both vacancy metrics being zero. In such a case there would always be a sole eviction victim candidate by lease policy.

4.2.1 Cache Tenancy Spectrum

CARL assigns leases based on a variable cache size, an issue PRL attempts to mitigate through phase-based analysis. PRL limits the cost of each phase in order to reduce over-allocation. At the same time a cost reduction in one phase may lower the cost in adjacent phases, increasing overall under-allocation. To examine the temporal dynamics of CLAM a new visualization and evaluation graphic, the *cache tenancy spectrum* is used. The spectrum is a two dimensional performance plot of each cache line's lease over a time interval. Leases are octal discretized as described by Equations 3.1-3.4, and are accordingly periodically sampled every 256 accesses (minimum frequency for octal resolution).

An example spectrum is shown in Figure 4-2 for informative purposes only. The horizontal axis represents logical time while the vertical axis represents a specific cache line (8kB cache has 128 lines, thus the axis ranges from 0 to 127). The value of a lease is indicated by the color/intensity of a data point, which is referenced to the colorbar to the left of the graphic. In this spectrum, roughly, the top third of the graphic is yellow, while the bottom is predominantly blue. The interpretation of this is that cache lines at high addresses are commonly expired (lease update stage encoder prioritizes low addresses for replacement), and thus the cache is under-allocated. There are also instances where all cache lines are have non-zero leases, which is indicated by all pixels in a vertical slice being non-yellow. These are instances of either full allocation or over-allocation; there is no straightforward way to discern

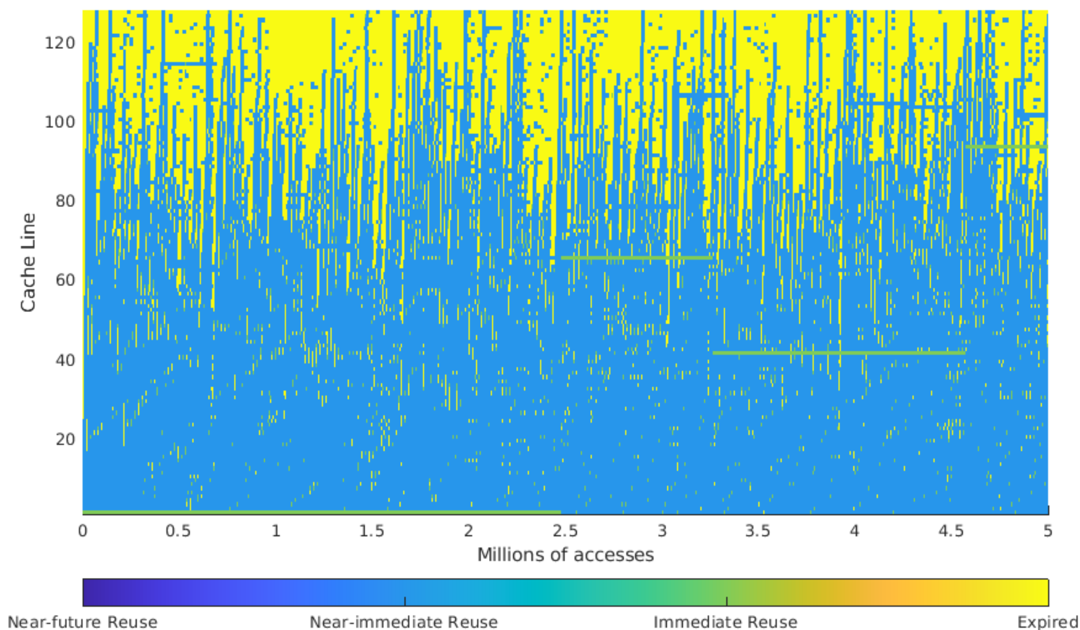

Figure 4-2: Cache tenancy spectrum for the example stencil (Listing 2.1).

between the two. This spectrum graph is heavily used in Chapter 5 to evaluate the performances of CARL and PRL.

4.3 Benchmark Applications and Policies

CLAM performance is evaluated using PolyBench/C 4.2.1, a benchmark suite which contains 30 numerical kernels [25]. Whereas the stencil (Listing 2.1) or any other arbitrary pattern demonstrates lease cache potential, this suite evaluates it. The kernels have real applications, being taken from linear algebra, image processing, physics simulation, dynamic programming, and statistics. Of this collection seven programs are selected for testing. These seven are chosen on the basis of having integer support, or can be directly ported for integer based data types. To make the benchmark programs completely compatible with the bare-metal FPGA environment POSIX APIs are removed and replaced where necessary. Table 4.3 highlights key parameters of the benchmarks along with baseline cache policy performance for reference.

Lease cache performance is compared against three reactive cache policies, LRU,

pseudo-LRU (PLRU), and static re-reference interval prediction (SRRIP). LRU acts as the baseline policy as many cache architectures use approximations of it as their policy, and it is used in Intel's Itanium [14] and AMD's Zen [3] L1 instruction caches. PLRU is a practical and commonly used policy [14, 29]. In this work, the policy is implemented using a single status bit in each cache line [30]. SRRIP [18], proposed as a more contemporary policy for insertion [28] with scan resistance potential, is a more general application of Not Recently Used (NRU), which is used in Itanium as well [14], so it is trialed as well.

Table 4.3: Benchmark programs and their baseline (LRU) performance.

Benchmark	**Input Size (N)**	**Kernel Memory References**	**Memory Accesses**	**LRU Miss Count**	**Benchmark Description**
Atax	120	60	491454	924	Matrix transpose and vector multiplication
Doitgen	25	59	8885194	941	Multiresolution analysis kernel (MADNESS)
Floyd-Warshall	180	35	116868071	364160	Dynamic programming: path search
2mm	60	86	6213792	19447	2 matrix multiplications (D=A.B; E=C.D)
3mm	60	115	10892247	34990	3 matrix multiplications (E=A.B; F=C.D; G=E.F)
Mvt	120	54	491302	15331	Matrix vector product and transpose
Nussinov	180	98	20051779	335369	Dynamic programming: table sort

Chapter 5

Results and Discussion

In this chapter, the performance of all three CLAM algorithms (FUL, CARL, and PRL) are evaluated. FUL is first presented as a lease cache safety feature, in that it can guarantee LRU equivalent performance if variable leases cannot be assigned by CARL or PRL. After showing CLAM safety, variable lease performance is compared against baseline cache policies and the best FUL assignment. Following this, PRL temporal sensitivity is evaluated for its impacts on cache performance. The analysis is concluded by examining preliminary results for CLAM's extension to set associative lease cache architectures.

5.1 Fixed Uniform Leasing

Across all seven programs FUL is able to, at minimum, perform as well as LRU, and in some cases perform significantly better (Figure 5-1). Atax, doitgen, and floyd-warshall all exhibit FUL-LRU equivalency. Regardless of the configuration, each FUL curve is coincident with the LRU performance over an extended range of leases. In the case of atax, the program has a relatively low amount of memory accesses (Table 4.3) making it more difficult for policy effects to manifest and be prevalent. Doitgen has a similar LRU miss count to atax, but at 20 times the number of accesses. This indicates that LRU already performs fairly well for doitgen's program structure, so FUL can only match its performance. On average, each data array in doitgen is 25

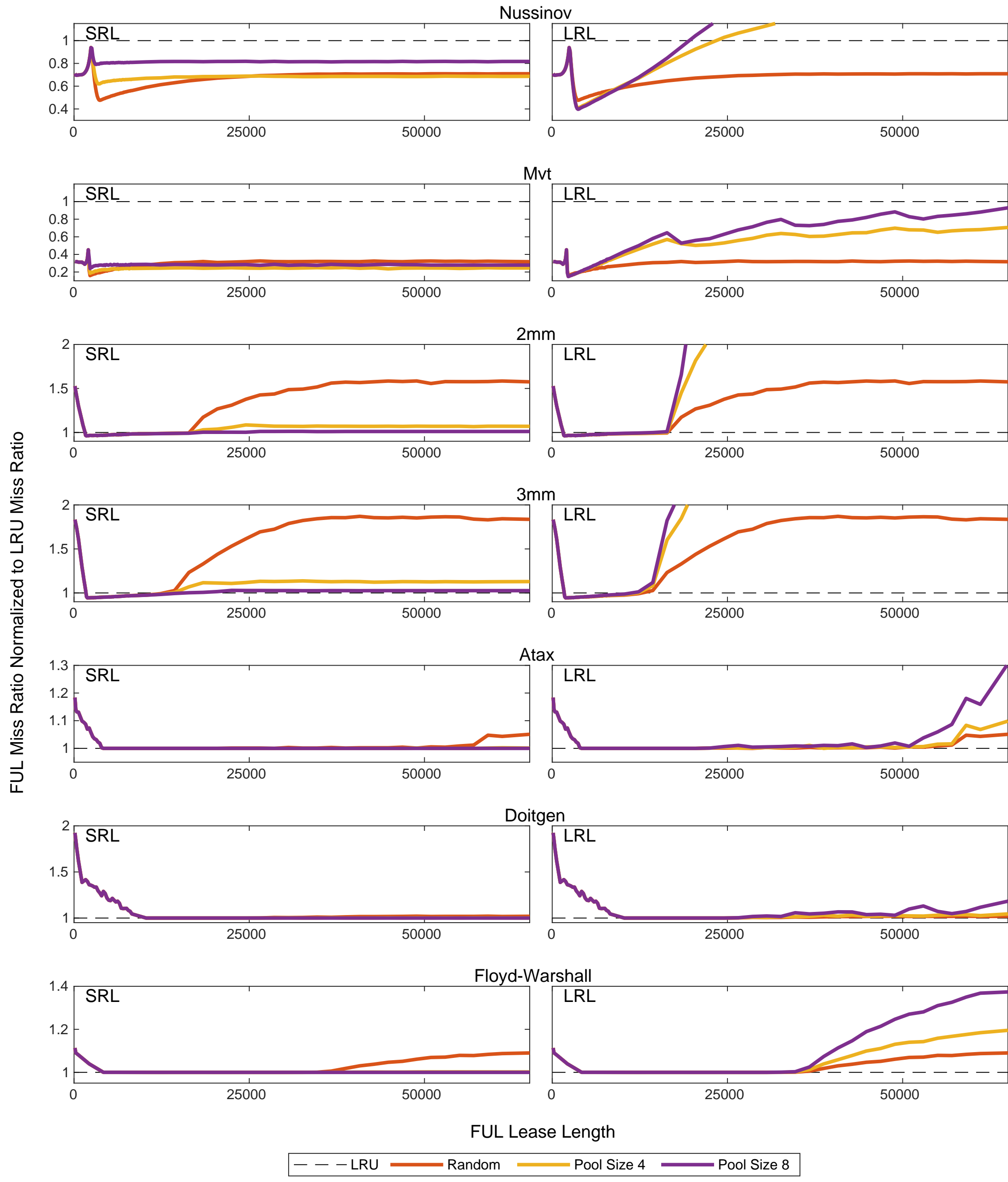

Figure 5-1: FUL policy performance over a 16-bit lease range with 7-bit resolution. FUL curves are normalized to baseline (LRU) performance, indicated by the dashed lines.

elements, so even when nesting loops the cache will retain the working-set. Floyd-warshall has similar attributes to doitgen, having a large number of accesses and a low LRU miss ratio. FUL likewise matches LRU performance for this benchmark.

Table 5.1: 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

For all three programs, the FUL configuration (auxiliary policy and pool size) does not impact the best achievable performance, only the range over which it is equivalent. Increasing the aggression (larger pool size) of SRL leads to a larger equivalence range (better approximation of LRU), while the opposite is true for LRL. A similar consistent characteristic is seen when examining the vacancy curves of these policies (Figure 5-2). The vacancy rate is static or decreases very slightly over the range of LRU equivalence. When exiting that range, the curve begins to decrease more rapidly, indicating that the uniform lease is now over-allocating cache, and the working-set is not being properly managed.

Similar LRU equivalence characteristics are seen in 2mm and 3mm, but with a narrower range of leases. There is a point at which FUL outperforms LRU (Table 5.1), regardless of the eviction configuration. The previous three programs displayed characteristics of having a minimum equivalency range that is not affected by the pooling, eviction, etc; only when leaving that range do those factors elicit different performance. 2mm and 3mm corroborate this observation, while also showing that performance improvements within this range can be similarly independent of the configuration.

FUL elicits the most unique characteristics in the remaining two programs, nussinov and mvt. These programs are the only instances of the zero lease assignment (equivalent to MRU) outperforming LRU, indicating that LRU is not optimal for these applications, and that the programs are more amenable to improvement by an alternative policy. Contrary to the equivalence ranges seen in the other benchmarks, neither benchmark has such a region, but rather a more defined optimal lease operating point. Additionally, cache performance at this point is now dependent on the

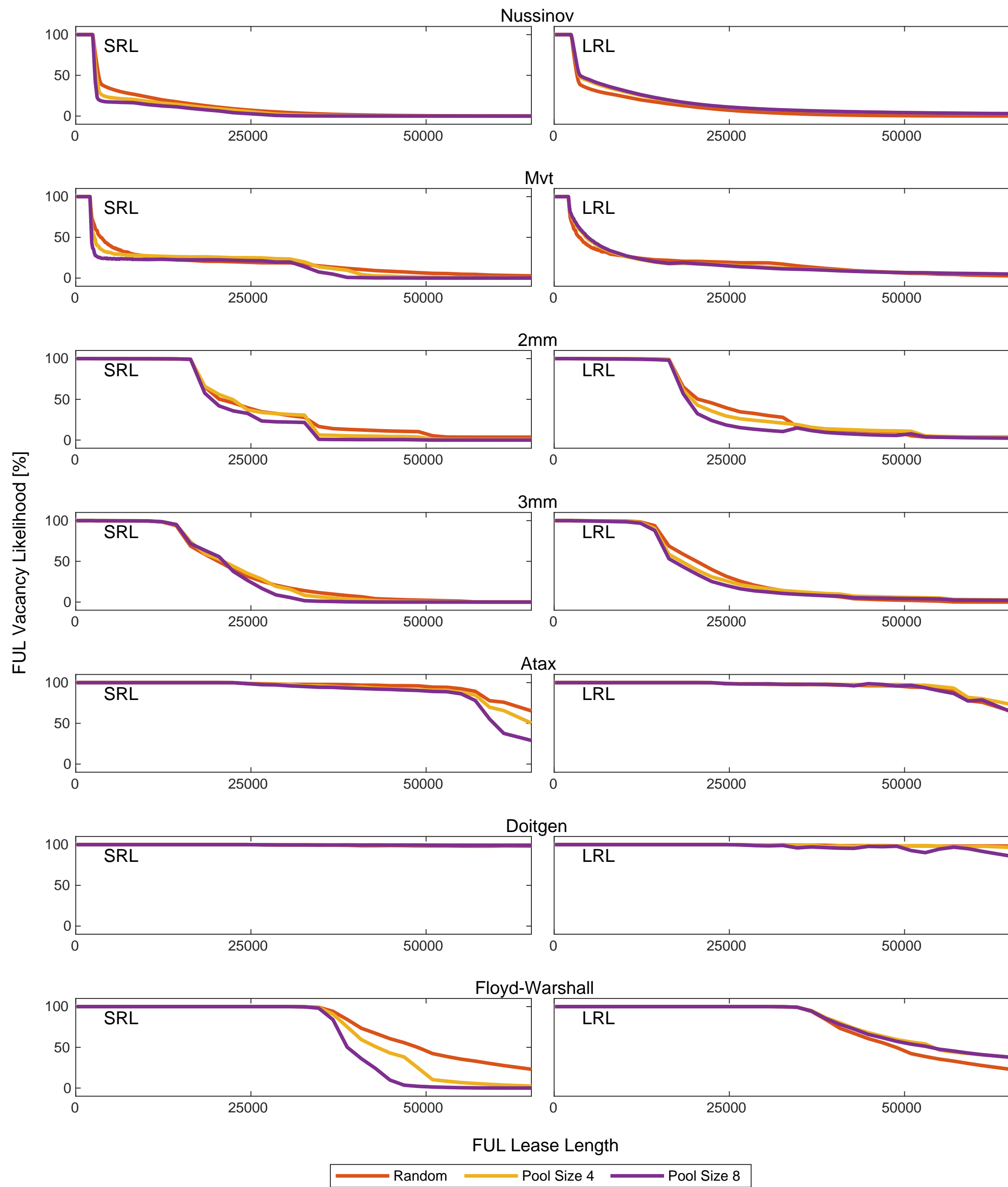

Figure 5-2: FUL policy vacancy of the trials shown in Figure 5-1.

configuration. When using SRL the best performing lease is negatively impacted by more aggressive evictions because this more accurately emulates LRU. Conversely, LRL performs best with this configuration; however, the window over which this is true is short and well approximated by random replacement. Because of this, random auxiliary eviction is considered the best policy for these two applications.

The most significant outcome of the FUL trials is the robustness and effectiveness of the random auxiliary policy. Random eviction is the most simple to implement and resource friendly of all policies, making it desirable to use if possible. Furthermore, the FUL results suggest that there is no benefit to using a policy other than random when there is no expired lease. In cases where FUL can significantly outperform FUL random is the best auxiliary policy outright. When significantly better-than-LRU performance is not possible, the choice of policy does not impact the lease range over which equivalence holds. Rather, the policy affects how quickly the vacancy curve decreases from a focal point, which in turn can effectively extend that range. Essentially, the lease and policy are orthogonal. The uniform lease portions the working-set, while the policy affects the remaining cache content. This is why the focal point exists in the vacancy curves. Hence, if the working-set is sufficiently portioned, the policy choice is arbitrary. This is directly extendable to variable leasing because both CARL and PRL evaluate analytically, instead of empirically. When using variable leases, it is assumed that the working-set is sufficiently partitioned by the assignment algorithm, and the cache is operating within this equivalence or better-than-LRU range, by nature of the assignment. Thus, the auxiliary policy has little impact, and can be chosen arbitrarily. This is the premise for choosing a random auxiliary policy in Section 3.1.1.

The results of each benchmark support the use of a singular lease as a safety measure; however, there is a nuance to this (which is most noticeable in 2mm). The range of equivalence is roughly the lease subset, $[1000, 15000]$. After exceeding 15000, the lease results in over-allocation, which is verified by the decreasing vacancy curve. Shortly after the focal point, the curves disjoint and appear to have piecewise characteristics (specifically referencing LRL), something that is not seen in other programs. This is theorized to be the result of the structure of 2mm. 2mm is comprised of two nested loop computational sequences, each with unique memory references and sizes. Because each loop (and consequently each working-set) is unique, each has different equivalence conditions (the lease range). When a lease of around 35000 is reached, the curves change behavior and cease to be monotonic. The working-set for

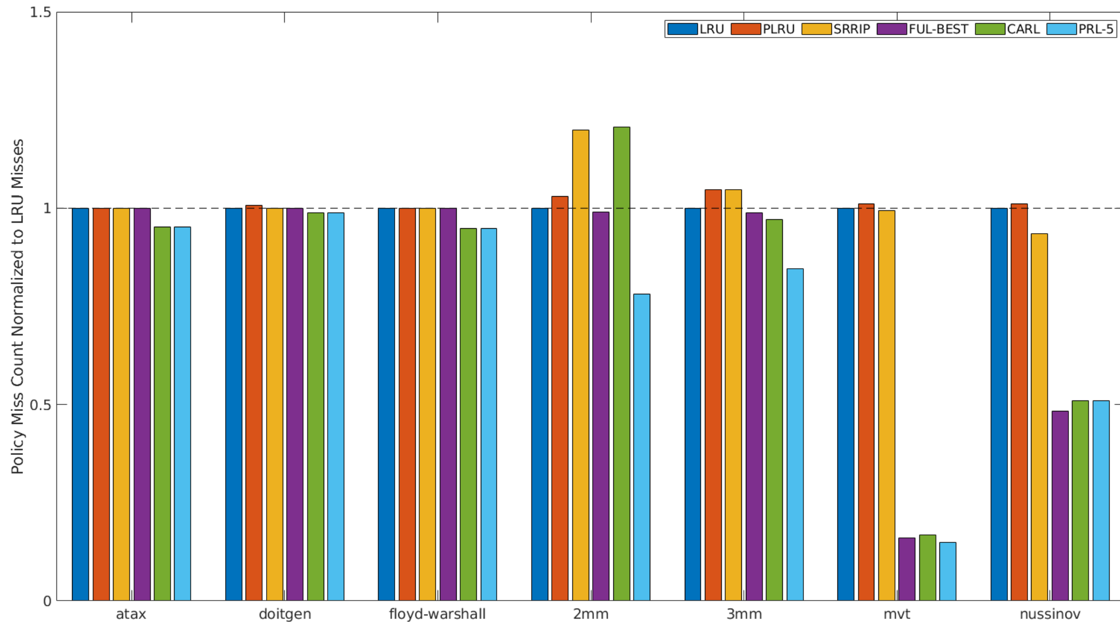

Figure 5-3: Normalized miss ratios of trialed caching policies [27]. Dashed line indicates baseline (LRU) performance. Tabled results given in Figures A.1 and A.2.

one loop is now disproportionately portioned compared to the other, which results in the irregular behavior. If two program segments are different enough or if there are many unique patterns LRU equivalence may not hold true, so it is important to recognize that this attribute is dependent on the evaluation scope. This concept is further examined using spectra in Section 5.2.

5.2 Variable Leasing

For six of the seven benchmarks CARL outperforms the LRU baseline (Figure 5-3). Atax, doitgen, and floyd-warshall are shown to be immune to improvement by uniform lease (Figure 5-1); yet, variable leasing elicits considerable positive change. Atax is difficult to meaningfully optimize due to its small number of accesses; even so an 8% improvement is achieved with CARL. A similar improvement is seen in floyd-warshall, which shows that leasing is scalable with access quantity. Of the three programs to which FUL fails to achieve better-than-LRU performance, doitgen is confirmed to be the least amenable to leasing (a modest miss reduction is still seen).

FUL performs best on nussinov and mvt, which suggests that a similar performance improvement should be possible when using variable leases. While CARL does outperform LRU to the same general degree that FUL does, it fails to do better than or equivalently to FUL. FUL shows that the kernels perform better when using MRU (as compared to LRU), indicating that the programs have higher degrees of non-recency based patterns. CARL partitions cache to the working-set, which explains the significant improvement over LRU. The inability to outperform FUL however, requires a more in-depth evaluation of how the cache resources are being utilized. This is discussed in Section 5.2.1.

A similar irregularity is exhibited by 2mm, but to a much more severe degree. This is the only instance of a lease assignment algorithm performing worse (20% increase in misses) than LRU. Unlike the previous two benchmarks, this has a straightforward explanation. As previously discussed with the FUL results, the benchmark is comprised of two separate loop scopes, each with distinct reuse interval distributions. These distributions are different, and yet they use the same set of leases, which yields different costs for each. CARL evaluates budget utilization as one aggregate term by projecting the statistical reuse interval distributions to the trace length size. When multiple distinct distributions exist two possible situations can occur: CARL over-allocates one program segment and under-allocates another so that the average cost meets the target budget, or CARL mis-projects the distributions causing the lease assignment cost to be valued incorrectly. Discerning between the two based solely on cache miss performance is not possible, so cache spectra are used in Section 5.2.1 to identify the issue.

3mm is an extension of 2mm (same algorithm, but with 3 sequential nested loops), so a similar cache performance is expected. Interestingly, CARL is able to perform better than both LRU and FUL. Miss performance is measured relative to the LRU baseline so it is possible that LRU innately performs worse on 3mm, a statement which the FUL results support based on the improvement percentages provided in Table 5.1. FUL is able to more significantly improve 3mm performance, as compared to 2mm, so CARL is expected to produce a similar effect. Although, with CARL the

improvement is much more significant, suggesting a fundamental difference in how the lease cache performs under the assignment policy.

5.2.1 CARL vs. PRL

In every benchmark PRL shows improvement over LRU, and performs equivalently or better than CARL. PRL and CARL are roughly equivalent in the first three programs (atax, doitgen, floyd-warshall). For atax and doitgen, this is a result of the cache being under-utilized, due to program structure. The CARL and PRL spectra for doitgen (Figure 5-4) are shown to prove this. The spectrums of each algorithm are exactly the same, which is confirmed by the aggregate vacancy curve of each. Even when forcing the cache to fill up from cold start before enabling lease-based evictions, no additional re-accesses occur. Thus, for the reuse intervals sampled, the leases generated for atax and doitgen, by either CARL or PRL, optimally allocate the working-set. The cache utilization for optimal doitgen performance is 50%. When using a reactive policy, such as LRU, the entire cache is blindly allocated; however, only half that is actually needed. This realization has obvious extensions to network caches. If occurrences such as this can be recognized, network resources can be moved and re-allocated to maximize overall utilization and performance. The same can be said about higher level processor caches. Shared levels of caches can be allocated to cores or hardware threads based on this predictive need, which can improve data movement efficiency.

PRL performs similar to CARL in nussinov as well, but not due to under-allocation. Nussinov is a dynamic programming application, essentially a sorting algorithm. Its reuses are not entirely dependent on the structure of the program, but rather the arrangement of items to sort, which can result in a varied reuse distribution. The complication of this is that it creates a more complex hybrid access pattern; this is why LRU performs poorly for this benchmark. Assignments from both CARL and PRL result in large tenancy oscillations (Figure 5-5) forcing the auxiliary replacement policy. PRL fails to prevent over-allocation, but does show signs of attempting to limit it. Zero/no cache vacancy is reached later on in PRL (5 million accesses) than in CARL (3 million accesses), which shows the policy somewhat limiting the phase costs.

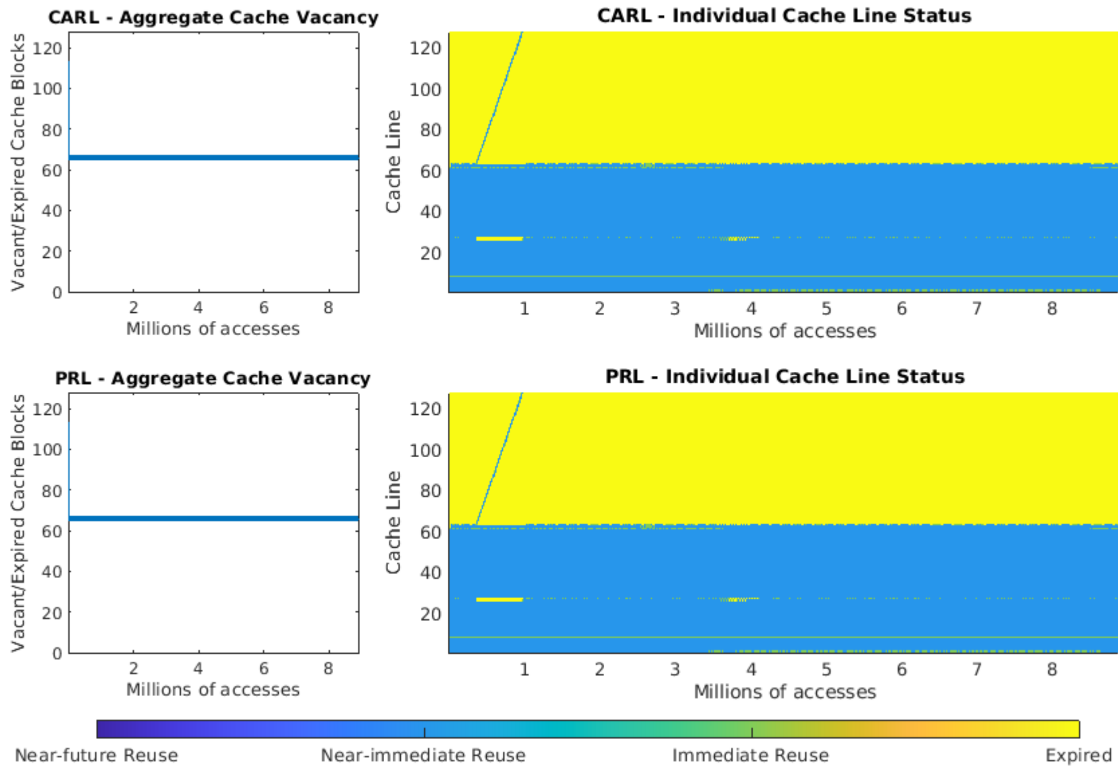

Figure 5-4: Cache aggregate vacancy (left) and cache tenancy spectrum (right) for doitgen benchmark using CARL (top) and PRL (bottom).

Additionally, the top of the PRL curve is bounded higher than CARL, indicating that on average there are more vacancies resulting from PRL than CARL (a byproduct of PRL more strictly limiting allocation). Based on cache performance (Figure 5-3) the benefit of limiting the over-allocation is negated by the induced under-allocation.

Mvt is the first instance of PRL outperforming all trialed cache policies, a direct result of its phase-based analysis. Similar to how 2mm is described to be a 2 scope program, mvt is comprised of a vector product and transpose. When CARL generates the leases for mvt, it under-allocates the first scope and over-allocates during the second scope (Figure 5-6). PRL similarly under-allocates the first scope; however, does not over-allocate in the second. At face value the aggregate vacancy curves indicate that CARL allocates more effectively in the first half, and as a result might perform better overall than PRL. Only by examining the spectrum is this proven wrong. The increase in tenancy during the first scope is due to large leases, that

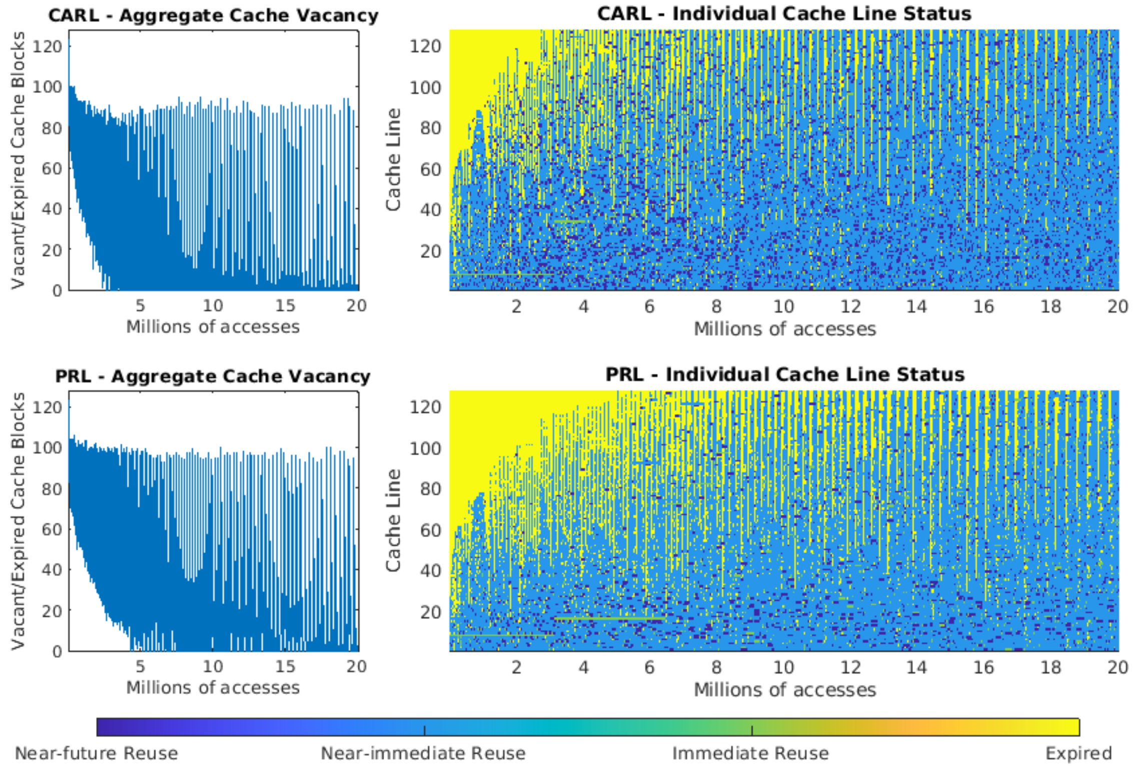

Figure 5-5: Cache aggregate vacancy (left) and cache tenancy spectrum (right) for nussinov benchmark using CARL (top) and PRL (bottom).

presumably have small PPUC (due to their large cost). These cache lines are not re-accessed during this interval, and as such are not providing any real benefit. This is something that PRL recognizes, and accordingly re-assigns leases (this is why there is no triangular dark blue region). CARL and PRL then, essentially contribute the same benefit during the first scope, or phase, of the program. Then, during the second program phase PRL more effectively manages the cache, resulting in improved overall performance. The aggregate curves confirm that PRL vacancy does not reach zero during this phase, something that is corroborated by the 'yellow marbling' seen in the latter interval of the spectrum.

PRL is able to achieve the most significant improvement in performance, compared to CARL, with the 2mm benchmark. CARL's spectrum (Figure 5-7) shows that the algorithm is inefficiently assigning leases (consistently under-allocating). Even though this is true, the target cost of optimization is met, which verifies that CARL

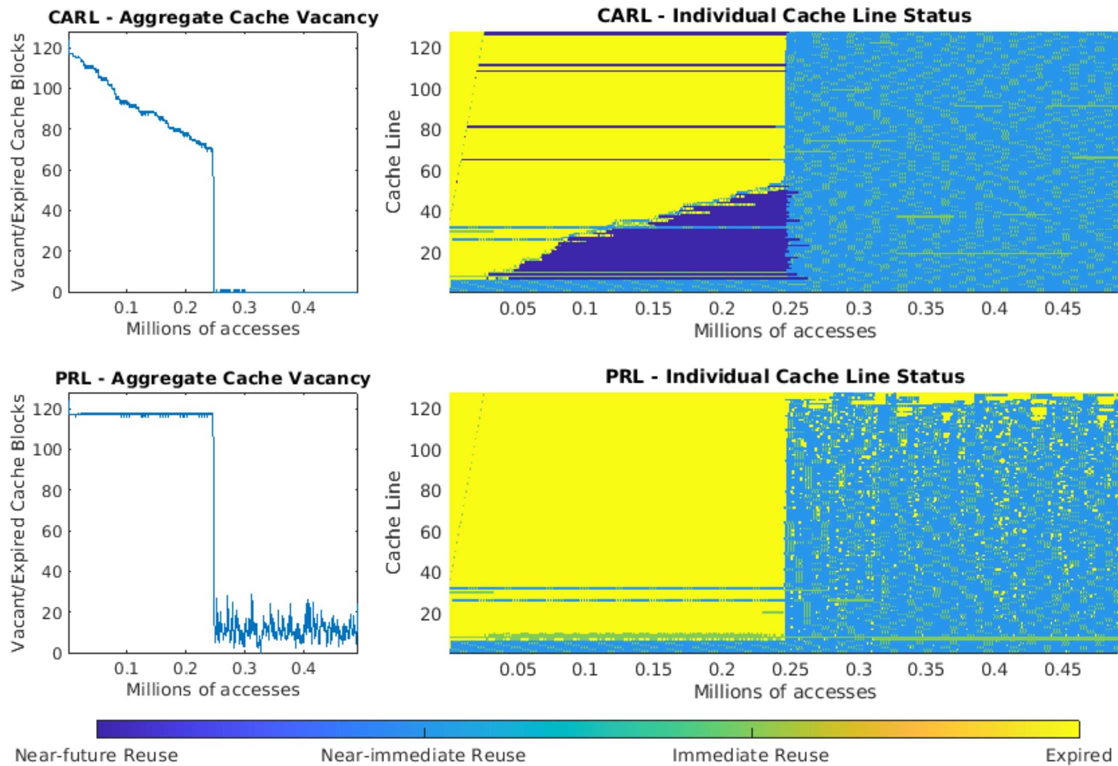

Figure 5-6: Cache aggregate vacancy (left) and cache tenancy spectrum (right) for mvt benchmark using CARL (top) and PRL (bottom).

mis-projects the reuse interval distributions. Essentially, sampling by a factor of x, reduces the allottable budget by x. This assumes that the trace has the same reuse characteristics over its length (equivalent to uniformly scaling the distribution counts), which for 2mm is known to be untrue. As a result, CARL overvalues its assignments. PRL is less susceptible to this because of how it partitions RI distributions by phase. It projects distributions to smaller subsets which limits how error scales. Using this PRL improves first phase utilization by increasing the dual lease allocation rate from 6% to 23%. When the second phase is entered, this increased dual lease results in larger allocation oscillations, as seen in nussinov, but is offset by reduced single lease assignment costs.

3mm is an extension of 2mm, yet lease cache performance does not follow the same trend as seen in 2mm. Both assignment algorithms outperform LRU (and FUL), and there is less relative improvement when using PRL over CARL. 3mm is a series of

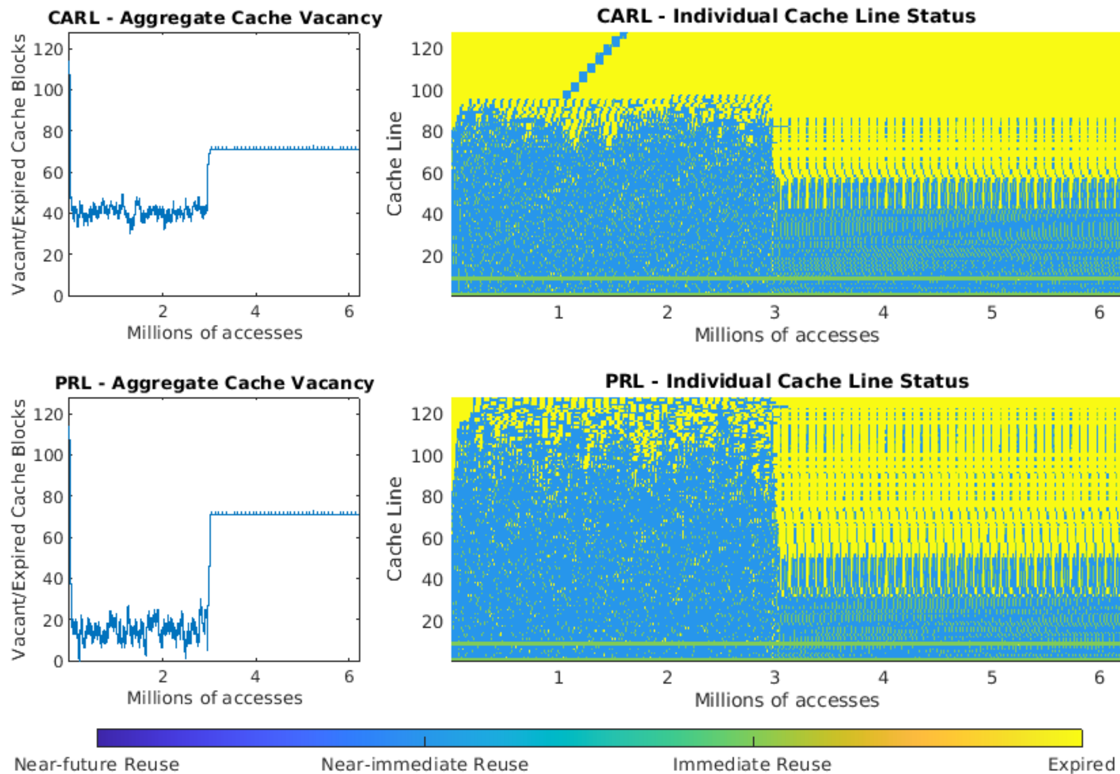

Figure 5-7: Cache aggregate vacancy (left) and cache tenancy spectrum (right) for 2mm benchmark using CARL (top) and PRL (bottom).

matrix multiplications, where the third matrix product is the result of the first two. Because of this dependency, changes in allocation to either of the first two phases elicit changes in the third, and vice versa. PRL limits the first phase allocation to prevent over-utilization, which results in increased under-allocation of the third phase, compared to CARL. The independence of the first two matrix operations would theoretically allow the second and third phase allocations to be improved through a shared reference; however, the compiler does not generate a reference mutually exclusive of the first phase (i.e. any additional assignment increases the cost of the first phase). With 2mm, RI projection error results in general under-allocation; yet, 3mm is able to avoid this, and is shown to be over-allocated in its first phase. The exact cause of this is unknown.

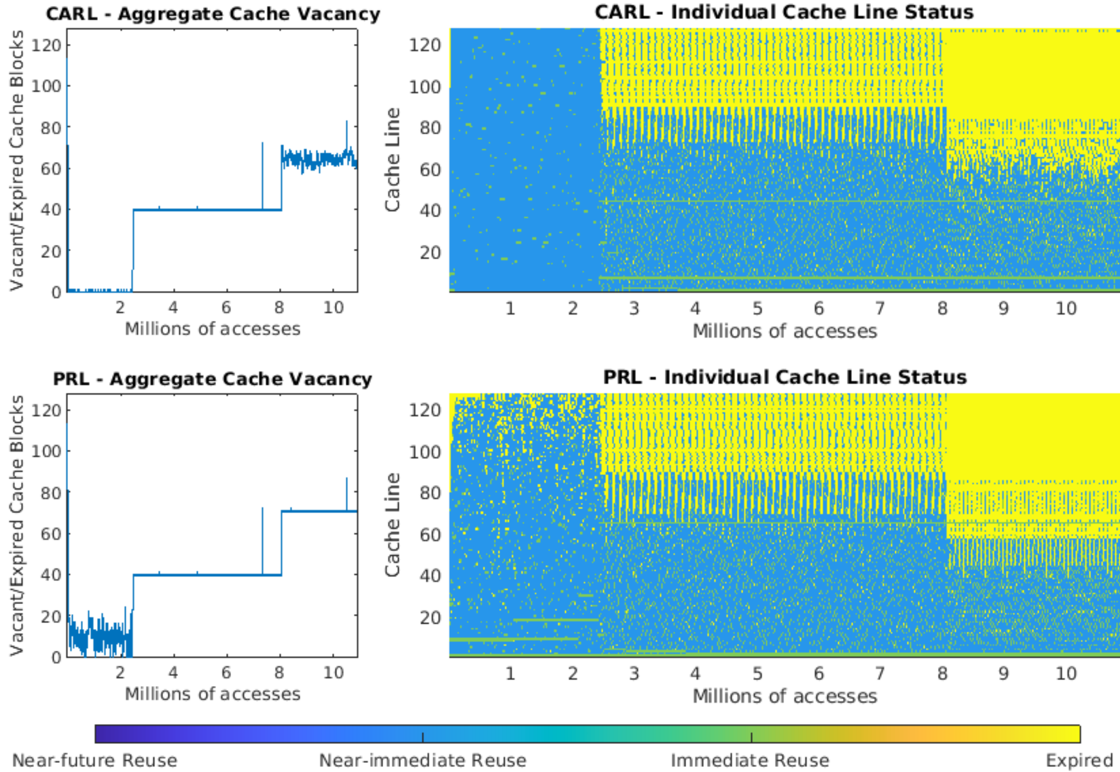

Figure 5-8: Cache aggregate vacancy (left) and cache tenancy spectrum (right) for 3mm benchmark using CARL (top) and PRL (bottom).

5.2.2 PRL Resolution

Benchmarks with multiple scopes such as 2mm and mvt showcase how a phase-based analysis can improve lease cache performance. Figure 5-9 presents the sensitivity of PRL phase resolution using the metrics defined in Section 4.2. Atax, doitgen, and floyd-warshall are omitted due to their similarity in how CARL and PRL perform.

Mvt exhibits the most apparent characteristics of the four benchmarks. As the no vacancy ratio (NVR) decreases the cache performance improves. When the phase count is increased past five, the complementary effect is seen, showing how increasing over-allocation (increasing NVR) directly reduces performance. Evaluating multiple vacancy ratio (MVR) is less straightforward; it is previously explained in Section 4.2 that this metric does not necessarily related to performance. As the phase number grows, MVR increases disproportionately to NVR, or rather it converges to the inverse of NVR ($1 - NVR$) at an increasing rate. PRL is more restrictive on the cost of

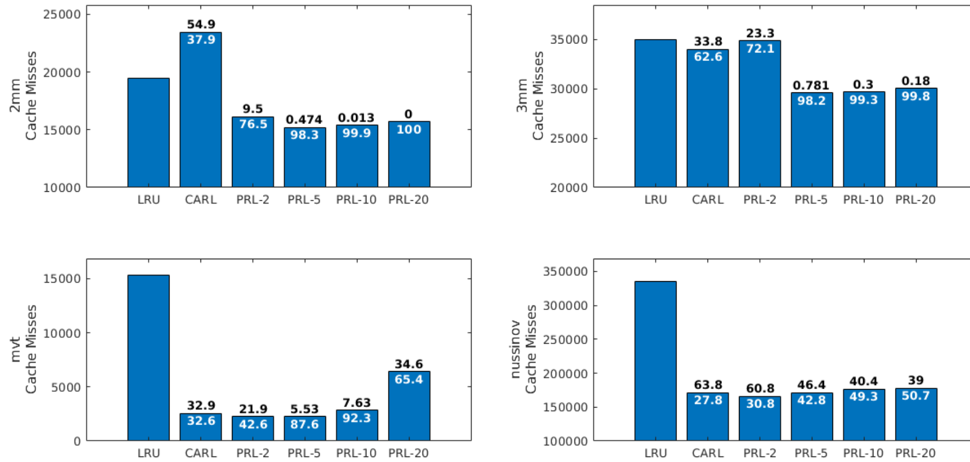

Figure 5-9: Comparing CARL and PRL for the four tests. Each PRL variant is labeled with the phase count. The numbers at each bar show the no vacancy ratio (above) and the multiple vacancy ratio (below the bar top). CARL over-allocates the cache with its leases, shown by its vacancy ratios as low as 36%. PRL eliminates over-allocation in the first three tests, shown by their near 100% vacancy ratios [27]. Tabled results given in Figure A.2.

each phase, hence the eventual increase in NVR. The accompanying impact of this is that the lease restrictions imposed by PRL similarly contribute to under-allocation. If cache performance were predictable based entirely on NVR, PRL-10 would outperform CARL and PRL-2, which it fails to do. PRL-10's MVR is very similar to its NVR inverse which shows that performance is instead being lost due to cached items of no future benefit.

2mm and 3mm phase results are similar to mvt. In both programs cache performance apexes at PRL-5, and become progressively worse with increasing phases. The programs however, exhibit continuously decreasing NVR and increasing MVR with phase. 2mm, the smaller of the two programs, results in 12 times more accesses than mvt. Phasing splits the number of accesses into bins, reducing RI distribution density. 2mm and 3mm, having a larger number of total accesses than mvt, have more statistically significant sample distributions per phase, which may impact how accurately leases can be assigned. Additionally, PRL-2 negatively impacts performance in 3mm, even though NVR decreases. In this instance the phase split does

not equal or exceed the number of program scopes (2 splits $<$ 3 scopes), a potential reason as to why PRL-2 is unable to improve over CARL. It has since been verified that increasing the PRL phase split to 3 improves cache performance to roughly a PRL-5 equivalent assignment. 3mm is the only three scope program examined so there is no benchmark to verify this theory against.

Nussinov is the exception to almost every observation made in mvt, 3mm, and 2mm. NVR decreases and MVR increases with the number of phase splits, but there is no correlation to cache performance. The previous three programs trended towards a NVR of 0% fairly quickly; however, nussinov is resistant to this, and its MVR does not converge to the NVR inverse. Figure 5-5 shows that there are no significant differences between CARL and PRL-5. This may suggest a more fundamental issue, that nussinov is not amenable to predictable lease-based optimization (although not predictable, but can still be improved using leases as compared to baseline policies).

The results suggest that the best number of phases is dependent on the number of program scopes, and by the sensitivity of the vacancy metrics. 3mm is resistant to improvement until PRL-5, presumably because the program is three phased. 2mm is similar in behavior to this; yet, mvt isn't shown to be as resistant (still benefits from increasing the phases to five). Conversely, the phases can be overly discretized, as evident by the decreased performance and vacancy metrics at high phase counts. In most cases the optimal phase number is where NVR is closest to zero and MVR is 'furthest from the NVR inverse' (a generalization of ideal allocation).

5.2.3 Preliminary Set Associativity

As of this work the impact of set associativity on lease cache performance is not yet conclusively studied; however, the initial work is presented. Using a spatial version of PRL (instead of splitting by time intervals, cache lines are grouped by set) the effects of cache freedom on lease potential is shown in Figure 5-10. For this analysis, CARL lease assignments, which assumes a fully associative cache, are compared against spatial PRL lease assignments (designed for set associative cache).

Without exception, the higher the associativity the better the cache performance.

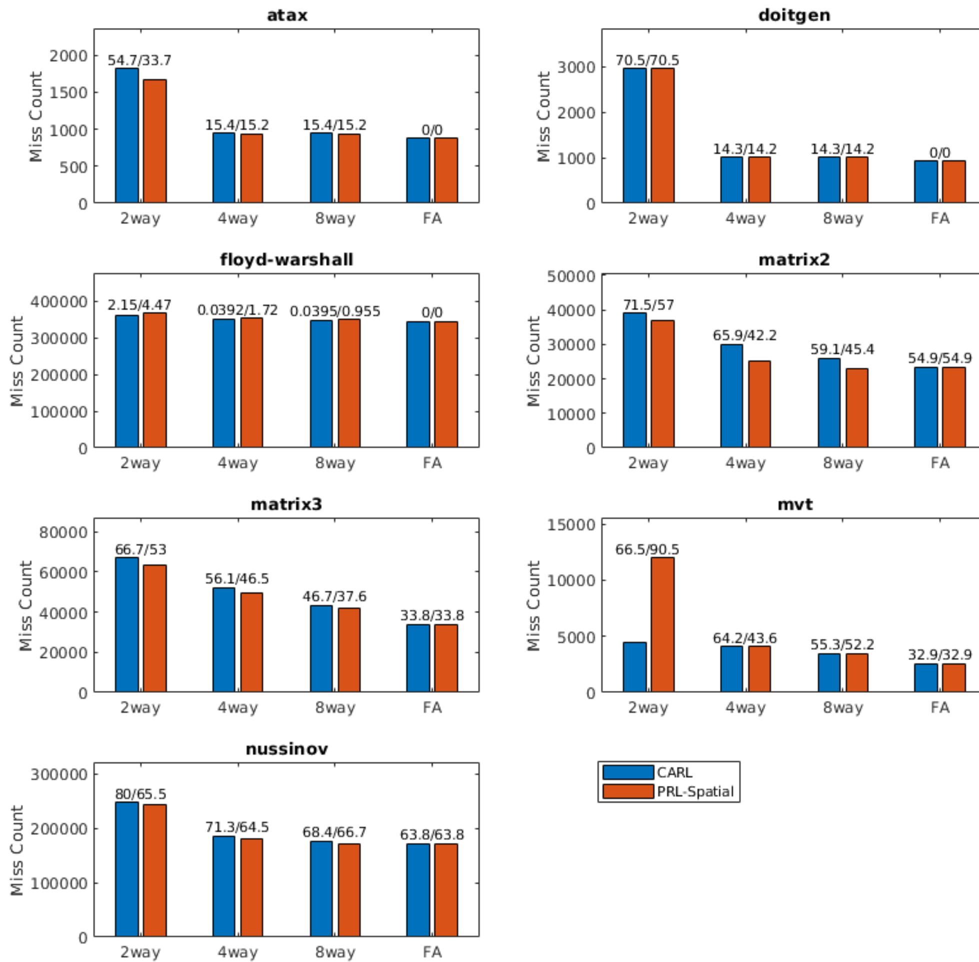

Figure 5-10: How well set associative caches perform when using leases designed for the architecture and ignoring set constraints. Blue bars are lease assignments made by CARL, which assume a fully associative cache. Red bars are lease assignments made by spatial PRL, which groups RI distributions by set. The numbers at the top of each bar pair is the no vacancy ratio of each bar respectively. Tabled results given in Figure A.3.

Higher degrees of associativity afford the assignment algorithm more freedom in making assignments, due to smaller numbers of phase splits (less chance of fully allocating a set, causing PRL to terminate early). This is expected of PRL due to its set-aware allocation, yet CARL is competitive, and even outperforms PRL in floyd-warshall and mvt (two-way). Ignoring the two exceptions, PRL results in a greater VR than CARL. The root cause of the floyd-warshall exception is not yet identified; the mvt exception is attributed to how PRL allocates the sets. It has been identified that

PRL heavily allocates about 15% of the total sets in the two-way architecture when evaluating mvt. Figure 5-6 shows that the second phase utilization of the program is high for a fully associative cache, so by limiting cache utilization to 15% PRL is inducing this significant performance hit, which CARL avoids.

The significant outcome of this preliminary study is that it confirms the theory that reduced cache freedom reduces lease cache potential. Eight-way performance is fairly similar to fully associative, and even four-way is able to avoid the performance cliff seen with two-way cache. How these results compare to baseline policies has not been studied yet; however, the fact that four-way and eight-way are within the same performance range as fully associative is a promising find to spur subsequent study.

Chapter 6

Future Work

This work focuses on embedded workloads for bare-metal execution environments. Although these programs are scientific applications that would be realistically computed in such an environment, they are predictable. Because these applications are predictable and repeatable, they can be profiled, as is done with the hardware sampler. The ultimate objective of lease cache is to be able to run operating system workloads. The issue with this is twofold. In this work the LLT is sized according to the benchmarks executed, and statically populated. Larger workloads however, will have more reference assignments than can be stored in a hardware lookup table which requires a mechanism for managing lease information - specifically when multitasking. The second issue relates to the practicality of storing and managing leases in hardware. In this work the lease registers are 24-bit because floyd-warshall produces a lease of that width, the largest of all trialed programs. In larger scale applications and caches the leases will continue to grow because CLAM does not bound parameters. In order to accommodate this all of the lease cache circuitry, specifically the update stage, will exponentially increase in size and power. The immediate first steps in solving each issue are,

1. *Scope Leasing* - the LLT is written to at run-time, allowing the table content to be altered either by an operating system, or directly at the machine level.

2. *Local Clocks* - lease registers, or groupings of, are decremented by local clocks

instead of global logical time.

3. *Parallel Assignment* - the executing program is sampled, and reference leases are assigned to lease cache in parallel.

6.1 Scope Leasing

Scope leasing at the machine level is the most immediate development step. It requires no changes to the lease cache circuitry, only changes to the controller logic involved with populating the LLT. The cache is able to utilize multiple lease sets, allowing CARL or PRL to optimize each set independent of the other, resulting in greater cache utilizations (no phase-cost dependencies of independent sets). This is directly extendable to set-associative caches, as increasing spatial phase allocation (sets) will have the same benefit. Once the LLT population mechanism switches from static to scope the overhead associated with this data movement becomes a more important performance factor to consider.

For a bare-metal system the cost of determining when to switch is absorbed at the machine level, so it doesn't attribute to latency cost. Two simplistic mechanisms have been evaluated: switching by trace length and by call (for bare-metal is a memory mapped load instruction). Given a predictable program sequence the trace lengths at which the LLT should be re/populated can be determined statically. Those lengths are then stored as configurations in the binary, just like lease information is currently. When the cache controller sees that number of accesses, it requests updated LLT data and writes in the new content. This method for leasing is not seen as scalable due to the degree of execution clairvoyance required.

A more realistic approach is to use the concept of phase markers. A compiler has the ability to place special instructions preceding what it determines to be a phase. This instruction can act like a system call, where it prompts the cache to fetch the phase encoded by the instruction. This way regardless of the execution sequence, the LLT will always be populated pre-phase. Implementing this to be tolerant of flow control instructions requires some degrees of redundancy in how the cache determines

what phase content the LLT currently has, and what is needed. Hashing the call write value or using a lookup table would accomplish this.

6.1.1 Multiple Dual Leases

CLAM is limited to generating one dual lease per trace evaluated. For programs with multiple scopes/phases, this can cause under-allocation as seen with mvt and 3mm (Figures 5-6 and 5-8 respectively). Incorporating additional phase-specific lease assignments can be used to locally improve allocation without increasing the cost in adjacent phases. To implement this, each lease can be associated with a phase ID. Similar to the effect of process IDs in memory managed cache, the phase IDs would be used on lease lookup to apply leases that only corroborate the current phase. Associating each lease with an ID would also improve LLT population efficiency. This enables several phases worth of information to be written to the table in one burst, which would theoretically optimize the memory transaction.

6.2 Local Clocks

Local clocks are a way of decreasing the resource overhead of the lease registers. Lease update stage circuitry scales with lease size, so minimizing the largest reference lease that can be applied is critical for practical applications. Set associativities provide an opportunity to modulate each group of lease registers independent of the other groups. The size of the reference leases decreases as a result because now they only depend on a subset of all accesses, only those inclusive to the set. Each cache group is associated with a local clock/counter, that increment uniquely based on global logical time. The lease registers of the set then decrement relative to this local counter. This is simple to implement in hardware; however, the software support for this is difficult (a comment made by the software collaborators of this work). Implementing this would most likely require a secondary register that implements some sort of modulo quantity to extend reference leases to any set, at any time.

6.3 Parallel Assignment

To use lease cache with operating systems the workload needs to be sampled, a statistical distribution of reuse interval distribution accumulated, and CLAM needs to assign leases, all in parallel with workload execution. Sampling and accumulation are tasks that can be performed easily at the machine level; however, implementing a variable lease CLAM algorithm in real-time is difficult. It would most likely require dedicated hardware, and even then the latency associated with generating leases may be too great to practically use. An alternative algorithm to use is FUL. Since FUL only generates a single lease, the algorithm can be implemented with simpler hardware, and presumably at greater speeds. The key obstacle to implementing this is how to do so when running parallel tasks, so that performance is not bottlenecked by CLAM.

Along the same lines of assigning lease in real-time, lease cache can be used as a collaborative policy instead of strictly prescriptive. For set associative caches in particular, under-allocation is a potential issue as discussed in Section 5.2.3. If lease cache is able to monitor each set's utilization over time, it can adjust lease assignments to achieve better performance. A simple implementation of this would be to apply a unique dual lease percentage correction factor for each set, which increases the effective percentage used to multiplex LLT leases. Sets with lower utilizations would have larger correction factors applied in order to improve the local allocation. Such a hybrid approach may also extend to improved OS support because it has reactive elements.

Chapter 7

Conclusion

CLAM gives the compiler complete control of cache memory, unlike traditional caching techniques. This allows software to emulate other policies or to target optimal cache performance by means of the lease assignment policy. Using FUL, LRU equivalent performance is matched in every benchmark, at minimum. Conversely, setting the uniform lease to zero achieves MRU performance, while making the lease arbitrarily large emulates random eviction. In this way FUL can easily imitate other policies, and affords lease cache a safety mechanism in the event that variable lease assignment cannot be accomplished.

The assignments made by CARL are optimal for variable sized caches; however, the cache tenancy spectrum developed for this work shows that practical CARL performance is reliant on program structure. Programs with multiple levels of complexities are less amenable to appropriate allocation by CARL. Using PRL, the deficiencies of CARL are mitigated, and clear improvements over all baseline cache policies are achieved. For recency-friendly applications CLAM is able to marginally improve performance ($<$15%) over LRU, which is viewed as an efficient policy for this pattern. Conversely, when LRU fails (stream or thrash patterns) CLAM is able to achieve between 50% to 70% miss reduction (as compared to LRU). CLAM has only been comprehensively studied for fully associative cache architectures; however, PRL is shown to have applications in spatial phase-based allocation as well, demonstrating improvement in set associative cache allocation and performance (relative to the

benchmarks trialed).

The immediate next development steps are identified for residual issues. Scope leasing will solve program scalability, while set-locality will support hardware scalability. Based on the allocations of the multi-phased programs examined in this work, there is a high probability that using non-static leasing methods to control lease lookup table content will improve local phase allocation and lease cache performance, a significant first step towards developing lease cache for general workloads.

Glossary

Aggregate Vacancy the total number of expired cache lines at any given time.

Cache Tenancy Spectrum time-space visualization of lease cache tenancy.

Dual Lease a two value lease probabilistic assignment generated by CLAM to allocated cache to the working budget, without exceeding it.

Lease a value used to protect items in cache for the duration of the value.

Lease Auxiliary Policy the replacement policy implemented on a cache miss when there is no expired lease.

Lease Update the stage of lease cache where lease register content is managed and eviction victims are identified.

Lease Lookup the stage of lease cache where memory access information is decoded into lease assignment information.

Lease Renewal a cache line's lease register content being updated with a new value.

Lease Expiration a cache line's lease register reaching zero.

Long Length Probability probability of assigning the long length lease upon a memory access.

Long Length Lease if assignment is a dual lease, this field is the longer lease assignment of the two. Otherwise, it is the sole lease assignment generated by CLAM.

Multiple Vacancy Ratio the ratio of expired evictions to total evictions (by lease expiration or auxiliary policy) when there are at least two expired cache lines.

No Vacancy Ratio the ratio of evictions by auxiliary policy to the total evictions (lease + auxiliary).

Reuse Interval the time between two accesses to the same data item.

Scope Leasing run-time management of the lease lookup table prior to entering a new program scope.

Short Length Lease if assignment is a dual lease, this field is the shorter lease assignment of the two. Otherwise, this value is redundant

Static Leasing static management (pre-execution) of the lease lookup table.

Working-set the collection of information referenced by the process during the process time interval.

Zero Lease Bypass the action of servicing a cache miss without caching the requested item due to a zero lease assignment.

Acronyms

AMD Advanced Micro Devices

ARC Adaptive Replacement Cache

CARL Compiler Assigned Reference Lease

CLAM Compiler Lease of Cache Memory

DDR3 Double Data Rate 3 (Synchronous Dynamic Random-Access Memory)

DMA Direct Memory Access

DRD Dynamic Reuse Distance

EHC Estimated Hit Cost

ELF Executable and Linkable Format

EVA Economic Value Added

FPGA Field Programmable Gate Array

FUL Fixed Uniform Lease

GD* Greedydual*

HLPL High Level Programming Language

I/O Input/Output

ISA Instruction Set Architecture

JTAG Joint Test Action Group

LACS Locality Aware Cost Sensitive

LFSR Linear Feedback Shift Register

LHD Least Hit Density

LLT Lease Lookup Table

LRL Longest Remaining Lease

LRU Least Recently Used

MIN Minimum

MVR Multiple Vacancy Ratio

NRU Not Recently Used

NVR No Vacancy Ratio

OPT Optimal

OSL Optimal Steady-state Lease

PD Protection Distance

PLRU Pseudo Least Recently Used

PPUC Profit Per Unit Cost

PRL Phased Reference Lease

RI Reuse Interval

RISC-V Reduced Instruction Set Computer V

RRIP Re-reference Interval Prediction

SLRU Segmented Least Recently Used

SRL Shortest Remaining Lease

SRRIP Static Re-reference Interval Prediction

UART Universal Asynchronous Receiver/Transmitter

VR Vacancy Ratio

Appendix A

Tables

Table A.1: Benchmark performance summary for reactive cache policies.

Benchmark	**Policy**	**Time [cycles]**	**Hits**	**Misses**	**Writebacks**
atax	LRU	1444374	491454	924	133
doitgen	LRU	28411706	8885194	941	940
floyd-warshall	LRU	358136746	116868071	364160	364160
matrix2	LRU	20241099	6213792	19447	448
matrix3	LRU	33675508	10892247	34990	643
mvt	LRU	1865464	491302	15331	142
nussinov	LRU	72231152	20051779	335369	1270
atax	PLRU	1444356	491454	924	133
doitgen	PLRU	28411906	8885194	947	943
floyd-warshall	PLRU	358141360	116868071	364311	364185
matrix2	PLRU	20257713	6213792	20023	448
matrix3	PLRU	33724264	10892247	36643	692
mvt	PLRU	1869798	491302	15502	142
nussinov	PLRU	72346314	20051779	338935	1280
atax	SRRIP	1444358	491454	924	133
doitgen	SRRIP	28411732	8885194	941	940
floyd-warshall	SRRIP	358148562	116868071	364357	364188
matrix2	SRRIP	20352806	6213792	23303	448
matrix3	SRRIP	33725494	10892247	36650	642
mvt	SRRIP	1863048	491302	15251	142
nussinov	SRRIP	71626283	20051779	313439	1259

Table A.2: Benchmark performance summary for CLAM cache policies. Note: MV Rep (Replacments) are the number of evictions made when there are two or more expired cache lines - the numerator for MVR.

Benchmark	**Policy**	**Time (cycles)**	**Hits**	**Misses**	**Writebacks**	**Auxiliary Rep.**	**MV Rep.**
atax	CARL	1442574	491454	879	36	879	879
doitgen	CARL	28411302	8885194	930	927	930	930
floyd-warshall	CARL	357477938	116868071	345342	345313	345342	345342
matrix2	CARL	20363587	6213792	23460	628	10578	8890
matrix3	CARL	33644658	10892247	33983	802	22493	21265
mvt	CARL	1492866	491302	2563	151	1721	835
nussinov	CARL	67469191	20051779	171194	2962	62049	47603
atax	PRL-2	1442548	491454	879	36	879	879
doitgen	PRL-2	28411309	8885194	930	927	930	930
floyd-warshall	PRL-2	357477938	116868071	345342	345313	345342	345342
matrix2	PRL-2	20143225	6213792	16140	472	14607	12348
matrix3	PRL-2	33671330	10892247	34851	750	26724	25121
mvt	PRL-2	1484539	491302	2276	145	1777	970
nussinov	PRL-2	67302055	20051779	165632	2818	64914	51082
atax	PRL-5	1442560	491454	879	36	879	879
doitgen	PRL-5	28411306	8885194	930	927	930	930
floyd-warshall	PRL-5	357477940	116868071	345342	345313	345342	345342
matrix2	PRL-5	20114329	6213792	15191	448	15119	14926
matrix3	PRL-5	33514016	10892247	29579	643	29348	29045
mvt	PRL-5	1483006	491225	2277	137	2151	1994
nussinov	PRL-5	67315025	20043758	171003	2343	91659	73249
atax	PRL-10	1442544	491454	879	36	879	879
doitgen	PRL-10	28411306	8885194	930	927	930	930
floyd-warshall	PRL-10	357477938	116868071	345342	345313	345342	345342
matrix2	PRL-10	20120610	6213792	15403	447	15401	15394
matrix3	PRL-10	33516020	10892247	29665	642	29576	29459
mvt	PRL-10	1497170	491085	2844	136	2627	2626
nussinov	PRL-10	67397521	20039191	176549	2159	105258	87012
atax	PRL-20	1442574	491454	879	36	879	879
doitgen	PRL-20	28411302	8885194	930	927	930	930
floyd-warshall	PRL-20	357477938	116868071	345342	345313	345342	345342
matrix2	PRL-20	20130695	6213792	15721	439	15721	15721
matrix3	PRL-20	33527650	10892247	30029	642	29975	29958
mvt	PRL-20	1566204	489073	6435	98	4206	4206
nussinov	PRL-20	67410189	20038184	177563	2139	108275	90064

Table A.3: Benchmark performance summary for set associative CLAM cache policies. Note: Rep = Replacements.

Benchmark	Cache Arch.	Lease Policy	Time (cycles)	Hits	Misses	Writebacks	Auxiliary Rep.
atax	2-way	CARL	1468887	491439	1718	317	1113
doitgen	2-way	CARL	28475717	8885178	2953	1529	870
floyd-warshall	2-way	CARL	358232176	116868062	366967	360128	350626
matrix2	2-way	CARL	20786267	6213774	37040	2712	15815
matrix3	2-way	CARL	34557699	10892221	63603	4652	29685
mvt	2-way	CARL	1576061	481112	11714	56	1270
nussinov	2-way	CARL	68352849	19981581	242564	4446	83338
atax	4-way	CARL	1443869	491439	939	7	796
doitgen	4-way	CARL	28413301	8885178	1014	870	870
floyd-warshall	4-way	CARL	357756580	116868062	353370	352557	347231
matrix2	4-way	CARL	20418343	6213774	25304	759	14303
matrix3	4-way	CARL	34122911	10892221	49828	1806	26387
mvt	4-way	CARL	1520059	490233	4140	71	2341
nussinov	4-way	CARL	67685489	20045798	181546	4057	62086
atax	8-way	CARL	1443877	491439	939	6	796
doitgen	8-way	CARL	28413303	8885178	1014	870	870
floyd-warshall	8-way	CARL	357623080	116868062	349525	349147	346166
matrix2	8-way	CARL	20357345	6213774	23257	576	12419
matrix3	8-way	CARL	33881299	10892221	41888	1038	25953
mvt	8-way	CARL	1517545	491286	3405	95	1628
nussinov	8-way	CARL	67472880	20051767	171103	3313	56959
atax	2-way	PRL	1467329	491439	1672	292	1108
doitgen	2-way	PRL	28475717	8885178	2953	1529	870
floyd-warshall	2-way	PRL	358233704	116868062	366964	360190	350549
matrix2	2-way	PRL	20782487	6213774	36911	2688	15881
matrix3	2-way	PRL	34552813	10892221	63427	4659	29801
mvt	2-way	PRL	1576609	480697	11993	53	1145
nussinov	2-way	PRL	68327984	19978906	243407	4367	83979
atax	4-way	PRL	1443869	491439	939	7	796
doitgen	4-way	PRL	28413301	8885178	1014	870	870
floyd-warshall	4-way	PRL	357757713	116868062	353404	352564	347339
matrix2	4-way	PRL	20410207	6213774	25026	744	14454
matrix3	4-way	PRL	34117929	10892221	49644	1824	26539
mvt	4-way	PRL	1519163	490258	4097	64	2312
nussinov	4-way	PRL	67631709	20043416	181285	3893	64415
atax	8-way	PRL	1443877	491439	939	6	796
doitgen	8-way	PRL	28413303	8885178	1014	870	870
floyd-warshall	8-way	PRL	357622978	116868062	349525	349130	346186
matrix2	8-way	PRL	20348079	6213774	22967	558	12549
matrix3	8-way	PRL	33880163	10892221	41865	1031	26143
mvt	8-way	PRL	1517549	491286	3408	84	1630
nussinov	8-way	PRL	67480415	20051767	171332	3360	56976

Bibliography

[1] Intel haswell. `https://www.7-cpu.com/cpu/Haswell.html`.

[2] Risc-v gnu compiler toolchain. `https://github.com/riscv/riscv-gnu-toolchain`.

[3] Advanced Micro Devices. *Software optimization guide for AMD Family 17h Processors*, 4 2017. "Rev. 3".

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

[5] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, Renton, WA, April 2018. USENIX Association.

[6] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. pages 64–75, 2015.

[7] Nathan Beckmann and Daniel Sanchez. Modeling cache performance beyond LRU. pages 225–236. IEEE, 2016.

[8] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

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

[10] Dong Chen, Pengcheng Li, Chen Ding, Colin Pronovost, Fangzhou Liu, Wesley Smith, Mingyang Jiao, and Hannah Simons. Optimal program allocation of cache: Theory and potential. In *PLDI*, 2020.

[11] Keith Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2nd edition, 2010.

[12] Peter J. Denning. The working set model for program behaviour. *Communications of the ACM*, 11(5):323–333, 1968.

[13] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. Improving cache management policies using dynamic reuse distances. pages 389–400, 2012.

[14] James Evans and Gregory Trimper. *Itanium architecture for programmers.* Hewlett-Packard Books, 1st edition, 2003.

[15] Evangelos Georganas, Kunal Banerjee, Dhiraj Kalamkar, Sasikanth Avancha, Anand Venkat, Michael Anderson, Greg Henry, Hans Pabst, and Alexander Heinecke. High-performance deep learning via a single building block, 2019.

[16] Xiaoming Gu and Chen Ding. A generalized theory of collaborative caching. pages 109–120, 2012.

[17] Akanksha Jain and Calvin Lin. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *ISCA*, pages 78–89, 2016.

[18] 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.

[19] Shudong Jin and Azer Bestavros. Greedydual* web caching algorithm: Exploiting the two sources of temporal locality in web request streams, 2000.

[20] M. Kharbutli and R. Sheikh. Lacs: A locality-aware cost-sensitive cache replacement algorithm. *IEEE Transactions on Computers*, 63(8):1975–1987, 2014.

[21] 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 *ASPLOS*, pages 243–256, 2019.

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

[23] Nimrod Megiddo and Dharmendra Modha. Arc: A self-tuning, low overhead replacement cache. 03 2003.

[24] K. Morales and B. K. Lee. Fixed segmented lru cache replacement scheme with selective caching. In *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*, pages 199–200, 2012.

[25] Louis-Noel Pouchet and Tomofumi Yuki. Polybench/c 4.2.1. `http://https://sourceforge.net/projects/polybench/files/`, 2016.

[26] Ian Prechtl, Dorin Patru, and Chen Ding. Design and evaluation of a fixed-size programmable working-set cache on fpgas. Technical report, March 2020.

[27] Ian Prechtl, Ben Reber, Chen Ding, Dorin Patru, and Dong Chen. Clam: Compiler lease of cache memory. submitted, 2020.

[28] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *SIGARCH Comput. Archit. News*, 35(2):381–391, June 2007.

[29] K. Shoemaker. The i486 microprocessor integrated cache and bus interface. In *Digest of Papers Compcon Spring '90. Thirty-Fifth IEEE Computer Society International Conference on Intellectual Leverage*, pages 248–253, 1990.

[30] Kimming So and Rudolph N. Rechtschaffen. Cache operations by MRU change. *TOC*, 37(6):700–709, 1988.

[31] Terasic. *DEO-CV User Manual.* Terasic.

[32] Tian Xingyan and Du Hongyan. Static cache hint generation based on a profile of the opt cache replacement. In *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)*, volume 9, pages V9–84–V9–87, 2010.

[33] A. Vakil-Ghahani, S. Mahdizadeh-Shahri, M. Lotfi-Namin, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad. Cache replacement policy based on expected hit count. *IEEE Computer Architecture Letters*, 17(1):64–67, 2018.

[34] Y. Wang, Y. Yang, C. Han, L. Ye, Y. Ke, and Q. Wang. Lr-lru: A pacs-oriented intelligent cache replacement policy. *IEEE Access*, 7:58073–58084, 2019.

[35] Z. Wang, K. S. McKinley, A. L.Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. Charlottesville, Virginia, 2002.

[36] 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.

[37] Liang Yuan, Chen Ding, Wesley Smith, Peter Denning, and Yunquan Zhang. A relational theory of locality. *ACM Trans. Archit. Code Optim.*, 16(3), August 2019.