2020-01-01

Finalcache: Eviction Based On Implicit Entry Reachability

Adrian Veliz
University of Texas at El Paso

FINALCACHE: EVICTION BASED ON IMPLICIT ENTRY REACHABILITY

ADRIAN EDWARD VELIZ

Doctoral Program in Computer Science

APPROVED:

Eric Freudenthal, Ph.D., Chair

Omar Badreddin, Ph.D.

Yoonsik Cheon, Ph.D.

Art Duval, Ph.D.

Stephen L. Crites, Jr., Ph.D.
Dean of the Graduate School

Dedication

This dissertation is dedicated to my brother Oscar, mother Martha, and father Jesus. Your love and support mean the world to me. This would not have been possible without you. Love you.

FINALCAHCE: EVICTION BASED ON IMPLICIT ENTRY REACHABILITY

by

ADRIAN EDWARD VELIZ, B.S. & M.S. OF CS

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

August 2020

Acknowledgements

I would like to thank everyone who has helped me with research and writing.

First, thank you members of the Robust Autonomic Systems Group, namely David and Daniel, for always listening to my complaints and offering feedback. Special thanks to Aleksandr for writing the parser and simulation code that forms the basis of most of Chapter 3, and for being my first mentee.

Second, thank you to the UTEP faculty who left their doors open to listen to me include Drs. Gates, Roach, and Kiekintveld. Thank you to my committee members for your guidance and feedback. And thank you to Office of Student Conduct and Conflict Resolution for your interventions.

Last, thank you Sharon Kotarski, Catherine Tabor, and Dr. Eric Freudenthal. Without you Sharon and Cat, I would not have started my career in CS. And you Eric helped convince me to attend Graduate School.

Abstract

Software caches for garbage collected systems are unaware which cache entries are referenced by a client program. Lacking this key information, software caches cannot determine the cache's contribution to the program's memory footprint. Furthermore, their eviction heuristics must rely on access history as a proxy for usage. Divergence between usage and access history can undermine the intention of eviction thereby resulting in problematic cache behavior.

This dissertation proposes designs for a novel family of "usage-based" software cache informed of entry reachability by the automatic memory management system. Unlike extant software caches, usage-based caches can accurately determine their memory footprint because they are informed of memory usage. Usage-based software caches require information and controls that are not exposed by many garbage collected languages' runtime systems including Java and Python. C#'s finalization interface is sufficient to construct a prototype implementation of a usage-based software cache called "FinalCache".

Unfortunately, C#'s finalization interface creates high coupling between FinalCache and its entries. We are unaware of any language runtime systems that do not have this characteristic. These deficiencies and strategies for potentially remediating them are also described.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Software caches traditionally use entry access history as the basis for their eviction strategies. Unfortunately, this heuristic can lead to unnecessary evictions and even duplication of cached objects. FinalCache uses implicit entry reachability, rather than access history, as its basis for eviction. By only evicting entries that are not reachable by a client program, FinalCache avoids the problem of in-use eviction associated with access-based eviction.

FinalCache also exposes the flawed assumption that an access-based software cache's configured size equates to its total memory footprint. In contrast, because its configured size adjusts only unreachable cache entries, FinalCache knows precisely what its contribution is to total memory footprint is at any given moment. Preliminary results based off trace logs from Firefox show that FinalCache achieved the same number of hits as access-based software caches while requiring a smaller configured size.

1.1 Overview

Caches trade memory for potential future computation or latency savings. The most notable example caches are those built into the CPU, namely the L1 and L2 caches, because they save on trips to memory. Memory itself could be considered a cache because it saves on trips to disk. Yet not all caches reside in hardware.

Programs often create objects which may be needed again at a future point, for example a program that may need the inverse of some matrix M multiple times. Rather than re-computing M's inverse (M^{-1})on demand, , a program could keep M^{-1}in a temporary variable to facilitate future reuse. If the program needed to create multiple temporary inverted matrices, the program could use a data structure called a **software cache**.. A software cache is a memory-limited data structure

that maintains data a program is likely to need again. Example software caches include Firefox's DiskCache[18] and Memcached[19]. When the cache exceeds its capacity, it will evict one or more entries based on its eviction policy. Unfortunately, this eviction can cause problems because software caches are unaware of whether an entry is reachable from outside the cache unless it explicitly tracks reachability.

Consider an object *A* that has just been evicted from a software cache, for example, a google.com browser tab. The cache did not store the object itself but instead held a reference to *A*. The resulting impact of the eviction on the total memory footprint varies depending on whether there were any other references to *A*. If the cache was the last reference to *A*, then the memory associated with *A* is eligible to be recycled by the automatic memory manager. Otherwise, *A* will remain resident in memory, meaning the eviction resulted in negligible recoverable memory, considering the size of *A* versus the size of its cache entry mapping.

This behavior could result in inconsistencies, where two or more different versions of the same entry exist simultaneously. As depicted in Figure 1.1 below, if a request is made for an evicted, but still in-use, object *A*, then the cache will miss and signify the requested entry was not found. As a result, the program might re-compute an independent duplicate object *A′*. Should either object be modified while the other remains active, it could lead to undesired program behavior. In effect, the eviction of *A* resulted in an increased total memory footprint and possible bugs.

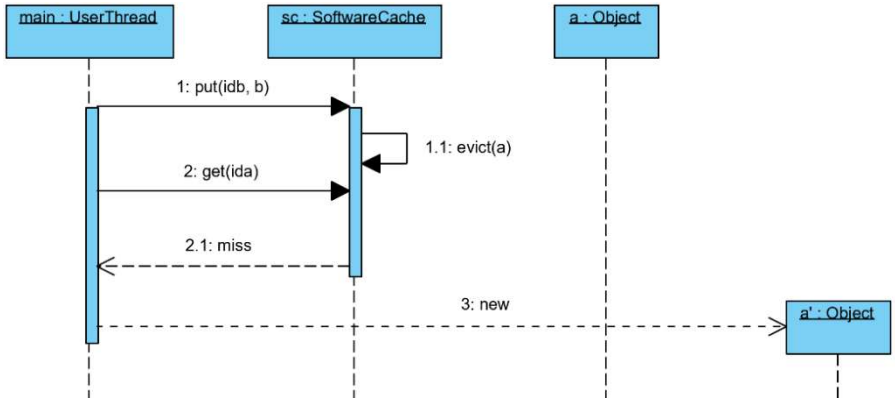

Figure 1.1: Sequence Diagram: Object Duplication[1]

To maintain consistency, the software cache could invalidate A on eviction, where it broadcasts that all references to A must be discarded. Or the cache could alter A itself to signify it is no longer valid. However, broadcasting would generate overhead and increase the complexity of the cache and client program. Additionally, a general-purpose software cache is not likely to directly modify its entries.

FinalCache avoids the consistency problem by implicitly tracking entry reachability, or usage, via the memory manager. Until an entry's Finalize method is invoked, FinalCache knows the object is still in-use by the client program and not eligible for eviction. Therefore, the addition of new FinalCache entries does not affect existing entries as shown in Figure 1.2 below. This is the same problematic scenario as before except since *A* is still in-use when *B* was inserted, *A* was not evicted.

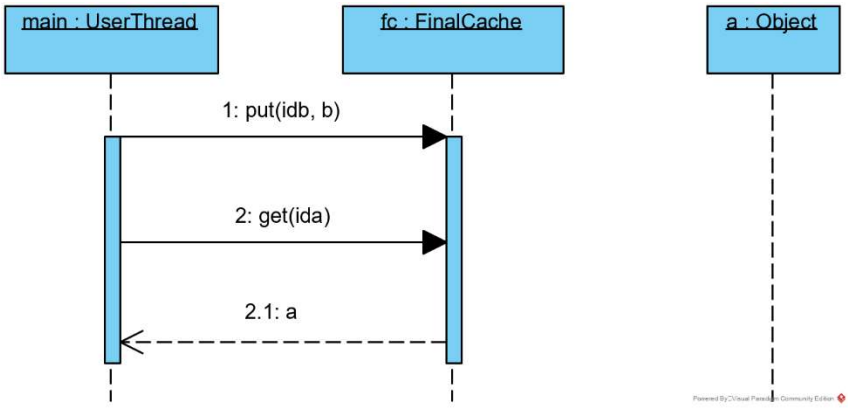

Figure 1.2: Sequence Diagram: FinalCache Avoids Object Duplication[1]

FinalCache is not the first software cache to be assisted by the memory manager[11,13,20], but as far as we are aware, it is the first that 1) uses purely usage-based eviction and 2) does not

[1] Object *A* does not have any messages sent to/from it. *A*'s lifeline shows reachability and is consistent with other Sequence Diagrams in this Chapter.

require the creation of a custom runtime.[2] FinalCache does not encounter the consistency problem described above because it cannot evict any entry that is still reachable by the program (excluding FinalCache itself). Preliminary results suggest that FinalCache's **usage-based** (reachability) heuristic is superior to **access-based** (order entries were last accessed in the cache) eviction.

1.2 CACHES

Application level software caches have a limited capacity and an eviction heuristic much like hardware caches. What differentiates the two is their level of abstraction and response times. Table 1.1 below compares/contrasts the two types of caches. Since software caches operate at the memory level, if each response takes a few extra clock cycles, it would not be noticeable; however, the same is not true of hardware. As such, software caches can employ more sophisticated eviction heuristics than is feasible in hardware.[6,7]

Table 1.1: Hardware v Software Cache Comparison

	Hardware Caches	Software Caches
Example Contents	Blocks of memory and frames	Data structures and files
Size	Fixed, < 10 GB	Dynamic, > 10 GB
Speed	Order of CPU cycles	Order of memory accesses
Example Mapping	Memory address -> Copy of memory value 0xfff ffff ffff 0000-> 0x0123 4567 89ab cdef	URL -> File Descriptor www.google.com -> 8
Eviction Heuristics	Simple and fast: Not Recently Used	Complex and slow: Least Recently Used, Frequency, Greedy

[2] More on this in Chapter 2 and Appendix C

While evictions make memory available for both cache types, software caches may not see any noticeable memory reduction. This is because evicted entries may still be reachable by the program, such as an open file, meaning the eviction may have only resulted in a handful of bytes saved. If the goal of eviction was to impactfully reduce total memory footprint, then in-use evictions by software caches fail at achieving this goal. Worse yet, should the program request an evicted in-use entry, then a duplicate object might be created which could result in inconsistent objects as seen in Sequence Diagram 1. Strong cache coherency schemes do exist to remedy this situation but applying them increases cache complexity and adds overhead (See Appendix A). The following section provides a detailed description of this problem, with respect to traditional Least Recently Used, and a simple, if imperfect, solution to it utilizing implicit reachability information.

1.3 LRA AND LRAWE

Least Recently Used (LRU) is an eviction heuristic, for both hardware and software caches, which applies the principle of temporal locality to identify cache entries unlikely to be accessed soon.[5] Unfortunately, this “used” label can be misleading because client programs contain references to seldom accessed cached objects. From the memory manager's perspective, an in-use or used object is one that is reachable from the **root set** (the set of all named objects). Those that are not reachable are called garbage, unused, or unreachable.[17] By this definition, traditional LRU tracks access history, not usage, and should be labeled **Least Recently Accessed** (LRA). This distinction is important because, as described below, eviction heuristics that utilize exact usage information are more deserving of the LRU mantle.

The reader may ask, “If access history approximates temporal locality, isn't it a good indicator of entry usage?" The answer is that programs may be actively referencing a cached object and may not access the cache to update this heuristic. Recall the scenario described at the start of this chapter. The client program was actively using an object which was evicted from the cache.

However, since the software cache did not know the entry was still being used, the cache needlessly evicted it thereby risking duplication and inconsistency.

There are several ways to address this problem. A naïve solution is to increase the size of the cache until in-use evictions do not occur, though selecting an appropriate size is difficult. Another solution is to utilize usage information. Unfortunately, explicit usage tracking has several drawbacks, principally its subversion of the automatic memory manager by requiring programmers be responsible for managing memory.[4] To ensure evictions, client programs would need to inform the cache when they are done using entries. Premature notification leads to the scenario described in Figure 1.1 above, and failure to notify means entries will remain in memory permanently.

Another solution would be to use memoization. Memoization is the process of automatically storing the results of a function call either using a lookup-table or software cache.[9,12] This process however is limited to function calls that do not result in visible side-effects. It also recursively solves the problem of caching by using a software cache which is not an adequate solution. Instead, caches could use Weak References to implicitly determine entry usage.

Weak references do not prevent their referent from being recycled by the memory manager when the referent is only reachable via Weak references. Should an object be reachable by at least one Strong (traditional) reference, then the object is considered live and cannot be recycled by the automatic memory manager. More information on the various reference types is available in Appendix C. [21]

If the software cache kept Weak References to evicted entries, then so long as the entry is still in-use by the program it will remain in the cache. This technique does not noticeably impact memory usage because the program would have still referenced the weakly referenced object anyway. We call this solution "Weak Eviction" and denote an LRA cache that employs it as LRAwe.[11]

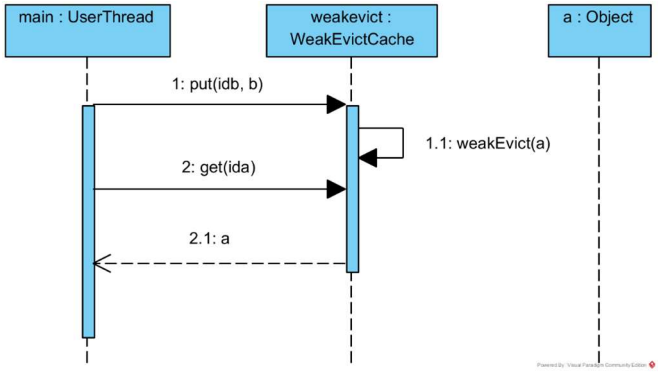

Figure 1.3: Sequence Diagram: WeakEvict Avoids Object Duplication

Figure 1.3 above illustrates the same problematic scenario for LRA described earlier except this time using LRAwe instead of LRA. A is weakly evicted from the cache when it is subsequently looked-up. This lookup is a hit because A was still reachable and the LRAwe cache will revert to Strongly referencing A. Although LRAwe solves this problem, it still relies on access history as a proxy for usage. The following drawbacks highlight why LRAwe is a less-than-ideal solution:

Uncertain total memory usage

There is no way to know the precise impact of LRA(we) on the Total Memory Footprint of a program. The footprint of an LRA cache is the number of entries it references that are not referenced by the program. Considering in-use entries as contributing to total memory usage is to count them twice. So, if an LRA(we) cache is configured to hold *n* entries then its impact on memory is in the interval from zero to *n*.

Entries have different lifetimes

LRA and LRAwe provide a safety net for entries that become unused before eviction. The cache will be the only reference keeping these entries resident in memory until they are either looked-up again or evicted. However, long-lived entries do not receive this benefit of being cached as they will persist long past their eviction. Even an LRAwe cache cannot ensure that these entries will briefly persist should the program request them shortly after becoming unused as can be seen in Figure 1.4 below.

Eviction of in-use entries

Caches have a maximum capacity to limit their total memory footprint. However, since evicting in-use entries does not reduce memory, why evict them? If software caches knew about entry usage, they could consider only unused entries as eligible for eviction, meaning evictions would guarantee reduced memory usage.

1.4 FINALCACHE

The main shortcoming of LRA and LRAwe is they are uninformed of entry usage at the time of eviction. We designed and prototyped a software cache called FinalCache, which only considers unused entries eligible for eviction. FinalCache accomplishes this without using explicit entry tracking, instead using an entry's Finalize method. Finalize is a mechanism provided by memory managed runtimes as an analog for object Destructors. Though Finalize is intended for pre-mortem cleanup, such as closing open file descriptors, it can also resurrect unused objects by making them reachable again by the root set.[3] [8]

FinalCache only considers resurrected entries eligible for eviction precisely because they became unreachable outside of FinalCache. Resurrected entries persist in FinalCache until n other entries become unused, causing an eviction (where n is the configured size, see Section 2.3). The addition of new used entries does not affect the set of evictable entries. Unused entries are evicted in the order they became unused i.e. true Least Recently Used (LRU). FinalCache addresses the three major flaws associated with LRA and LRAwe.

1. FinalCache knows precisely how much it is contributing to the client program's memory footprint; it is the total number of unused entries.
2. Unlike LRAwe, FinalCache' design ensures all unused entries persist for some time after becoming so. This is demonstrated in Sequence Diagram 4 below which is a problematic scenario for LRAwe but not FinalCache. After *A*'s Finalize executes,

[3] Implementations of Finalize vary by language and runtime. See Appendix C

FinalCache will create a Strong reference to *A*, allowing to it persist for a time after becoming unused.

3. FinalCache does not evict in-use entries because only unused entries are eligible for eviction.

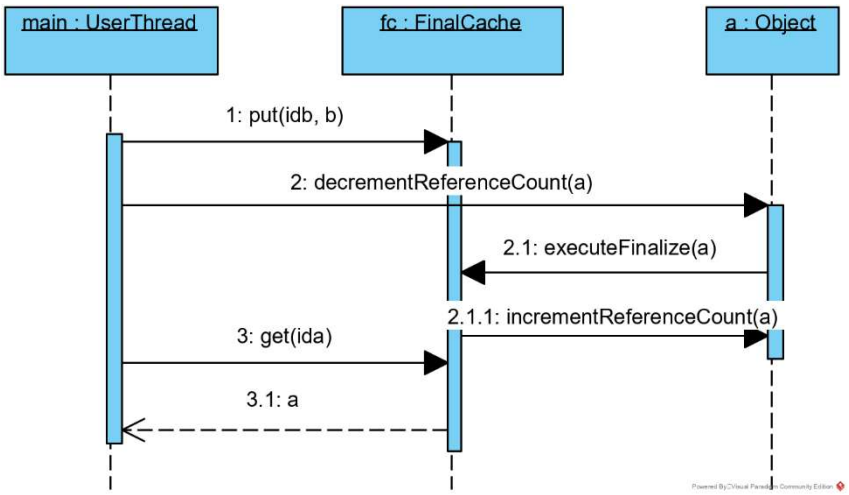

Figure 1.4: Sequence Diagram: All Unused FinalCache Entries Temporarily Persist

We wrote an implementation of FinalCache in C#. This language was chosen because it is the only language and runtime that the author is aware of that included all the features necessary for implementation, specifically the mechanisms available to interact with the memory manager. More information on this implementation is provided in Section 2.9.

The main shortcoming of this prototype FinalCache is that it cannot currently be used as a drop-in replacement for existing software caches because of its reliance on Finalize to provide usage information. Each FinalCache entry requires a custom Finalize method which strongly couples FinalCache to its entries. FinalCache will also behave differently depending on whether it runs with a Reference Counting or Tracing Garbage Collector and whether Finalize is Synchronous or Asynchronous. More information on this impact is in Chapters 2 and 3.

1.5 Document Preview

Chapter 2 is a complete description of FinalCache. This includes its design goals and a discussion on how cache size affects memory footprint. The simple FinalCache design assumes a Reference Counting memory manager with Synchronous Finalize because Asynchronous Finalize can lead to a race condition. Therefore, we also include designs which work with Asynchronous Finalize as well as a discussion on how Tracing collection affects FinalCache's behavior. The chapter concludes with an analysis of FinalCache's deficiencies and how they might be rectified.

Chapter 3 covers a simple empirical study of FinalCache compared against LRA and LRAwe. Because of the nature of its implementation, it was impossible for me to use FinalCache as a drop-in replacement for an existing software cache, requiring simulation instead. We generated cache access logs from Firefox's DiskCache[18], which includes entry usage information, to use as input. We then computed hits, misses, in-use evictions, in-use misses, and total memory footprint in terms of entries with many sizes per eviction policy. LRAwe and FinalCache do not have in-use evictions or in-use misses. The results show that not knowing usage was a problem for both LRA and LRAwe and that FinalCache, even configured with a relatively small csize, had a better hit rate than LRAwe.

Chapter 4 discusses possible applications of the techniques used to build FinalCache including object pools, passivation, and file closures. It also suggests modifications to existing memory management mechanisms which will enable widespread use of FinalCache in a production environment without the need for Finalize. The chapter concludes with possible directions for future work. Additional background information on software caching, garbage collection, and mechanisms which allow programs to interact with the runtime are included in the Appendices.

Chapter 2: FinalCache

This chapter describes the design and implementation of FinalCache as one solution to the unnecessary and potentially problematic eviction of in-use cache entries inherent to access-based eviction introduced in Chapter 1. First, we provide an overview of the properties of FinalCache, followed by an examination of how the definition of a cache's size changes depending on eviction policy. Then we examine the simplest possible design for FinalCache, assuming a Reference Counting memory manager and Synchronous Finalize. This is followed by a scenario using Asynchronous Finalize that causes said model to behave incorrectly. Last, we present an altered model that correctly deals with the problematic scenario and discuss the technologies needed to implement FinalCache.

2.1 FinalCache Goals, Requirements, and Benefits

FinalCache was designed to use object reachability, rather than access history, as its eviction heuristic. As such, the addition of a new **FinalCache Entry** (FCE) does not affect eviction decisions. Rather, FinalCache's heuristic is only updated when FinalCache is notified that an FCE has become unreachable by the client program.

This design ensures that:

1. Only unused entries will be considered for eviction from FinalCache
2. Entries will be evicted in the order in which FinalCache is informed they became unused[4]
3. Entries will persist for some time after becoming unused (see Section 2.2)

Recall LRA and LRAwe caches consider all entries as eligible for eviction as soon as they are inserted into the cache (see Section 1.3). This policy could result in LRA and LRAwe caches evicting an entry that is currently used by the program. On the contrary, FinalCache entries must first become unreachable by the client program before they can become eligible for eviction. FinalCache is notified that an entry has become unreachable when the FCE's Finalize method is

[4] A cache hit or update on an entry that has been identified as unused will become used again by the program, meaning the entry is no longer eligible for eviction

executed. At this point, finalized FCEs are eligible for eviction in FIFO order. Ultimately, FinalCache's design guarantees that, so long an FCE is reachable by FinalCache (in-use or unused), all requests for a given FCE will return the exact same reference.

FinalCache uses a combination of finalization and non-strong references to determine entry usage. An entry is considered in-use by the program until its Finalize method is executed. At this point the entry will be known to be unused because references from the client program to an FCE's referent (excluding FinalCache) prevent Finalize from being scheduled. As such, FinalCache will consider the entry to be in-use by the client program.

There are three possibilities for mapping an FCE's id to value: a strong mapping, non-strong[5] mapping, or no mapping. Strong references are unsuitable as an initial mapping because they would prevent the FCE's referent value from being finalized. Alternatively, FinalCache could simply not map in-use entries to allow Finalize to occur when the referent became unreachable. However, not mapping would mean that some cache accesses would miss until the FCE's referent Finalize method executed. Non-strong entry mappings would permit referents to become garbage collected while also allowing for successful entry lookups when they are still in-use by the program. More information on language features necessary for FinalCache are provided in Section 2.9.

When an FCE's Finalize method executes, it replaces the FCE's non-strong reference mapping with a strong mapping. This prevents the memory manager from recycling the referent as it is now only reachable via FinalCache. Should an unused entry be looked-up, its entry mapping will be reverted to a non-strong mapping to begin the process anew.

Unlike LRAwe, whose long-lived entries are removed on becoming unreachable, FinalCache's long-lived entries persist until sufficient other FCEs' Finalize methods are run. This feature of FinalCache can provide a window of opportunity to quickly recover a long-lived but

[5] Weak references are preferable to Soft references as they allow for Finalize to be scheduled much sooner after becoming unreachable outside FinalCache. Phantom references are not suitable for this application. For more information on non-strong references see Appendix C.

seldom accessed browser tab, such as Gmail, should a user accidentally close it. Another feature of FinalCache is its ability to safely transfer unused entries from memory to secondary storage because unused entries are exclusively reachable by FinalCache (though the cost of this transfer may be undesirable).

2.2 Size, Maximum Capacity, and Policy

This section examines how the definition of a cache's size changes depending on the type of cache. For hardware caches, size refers to the number of entries that can be mapped. In contrast, a software cache's size is not fixed and can change overtime, requiring a more nuanced definition of size. Additionally, software cache entries that are reachable by the client program outside of the cache do not truly contribute to the program's total memory footprint; meaning that an increase in size does not necessarily result in increased memory usage.

We define **configured size** (csize) to be the number of entries, or total bytes, a software cache can map before the addition of new entries requires eviction. Specifically, csize is the upper limit of the contribution of an LRA cache to a program's total memory footprint. If all entries in an LRA cache are reachable by the client program, then the effective footprint of the cache is only the overhead of csize mappings. Conversely, if all entries are unreachable then the footprint contribution of an LRA cache is equal to csize. Essentially, only **unreachable cache entries** (usize) contribute to the program's memory footprint and **reachable entries** (rsize) do not. Both rsize and usize are, to some extent, implicit parameters controlled by the client program and not the software cache.

Clearly, the **total size** of a cache is the sum of usize and rsize; however exact value usize and rsize are unknown to caches that do not track entry reachability. For example, LRA's total size is equal to csize. LRAwe’s size changes with the number of **weakly evicted reachable entries** (wsize). As such, LRAwe's total size is the sum of csize and wsize. In FinalCache, rsize is the number of FCEs that are ineligible for eviction because they are still reachable by the client

program; and usize is equal to csize or the entries that are eligible for eviction because they are unreachable by the client program. Therefore, the total size of FinalCache is always the sum of csize and rsize. Table 2.1 provides example combinations of csize and rsize and their effect on total size.

Table 2.1 Contributions of csize and rsize on total size of LRA, LRAwe, and FinalCache

csize	rsize	LRA	LRAwe	FinalCache
0	50	0	50	50
50	0	50	50	50
50	50	50	50 … 100	100
50	100	50	100 … 150	150

A csize of zero is a degenerative configuration for LRA because an LRA cache will evict every entry as soon as it is added; however, LRAwe and FinalCache will continue mapping entries so long as they are reachable by the client program. For csize zero, LRAwe and FinalCache will contain the exact same set of rsize entries. A side effect of csize zero is that should all cache entries become unreachable, both LRAwe and FinalCache will shrink to a total size of zero, due to LRAwe's and FinalCache's ability to dynamically resize.

The second row of Table 2.1 is an example where all entries become unreachable as soon as they are added to a cache with a nonzero csize. In this scenario, all three caches will contain the exact same entries because access history will be equivalent to usage history. As such, all will have identical hits and misses for any csize greater than zero.

The final two rows illustrate the effects of ambiguity caused by not knowing usize. For LRAwe, the true total size exists in the interval between rsize and the sum of rsize and csize. FinalCache's total size, in all scenarios, is the sum of csize and rsize.

Lastly, while LRA has the lowest total size, it is only because it did not grow with wsize as did LRAwe. However, because some of LRA's evicted entries may still have been resident in memory, the client program's memory footprint size would be same when using an LRA cache as

it would be with an LRAwe cache. In fact, the LRA client program's memory footprint may surpass LRAwe's due potential object duplication.

To ensure correct behavior, both LRAwe and FinalCache must be allowed to grow to accommodate still reachable entries. Limiting this growth could result in the eviction of reachable entries, violating the promise of both caches. Alternatively, the caches might refuse new entries but this policy is less than ideal. Or the caches could adjust the limit, increasing the caches' complexity and having the same net effect as not having a limit.

2.3 SYNCHRONOUS MODEL

This section describes the state machine for a basic FinalCacheEntry. There may well exist alternative models for FinalCache that accomplish the same goals; however, this design is straightforward and demonstrably correct for the simple case of a Reference Counting garbage collector and Synchronous Finalize. A FinalCacheEntry can exist in one of three states: Empty, Used, and Unused. The possible transition events include: put, get, finalize, updateLRU, and delete. Put can take two forms based on whether there already exists a mapping of the put entry in FinalCache. Duplicate puts are idempotent, meaning only the first put will meaningfully change the state of an FCE. Later sections in this chapter explore the ramifications of delayed reachability identification and execution of Finalize. Figure 2.1 below shows the Synchronous Finalize FCE state machine followed by a table that elaborates on what different states mean and the key transitions between them.

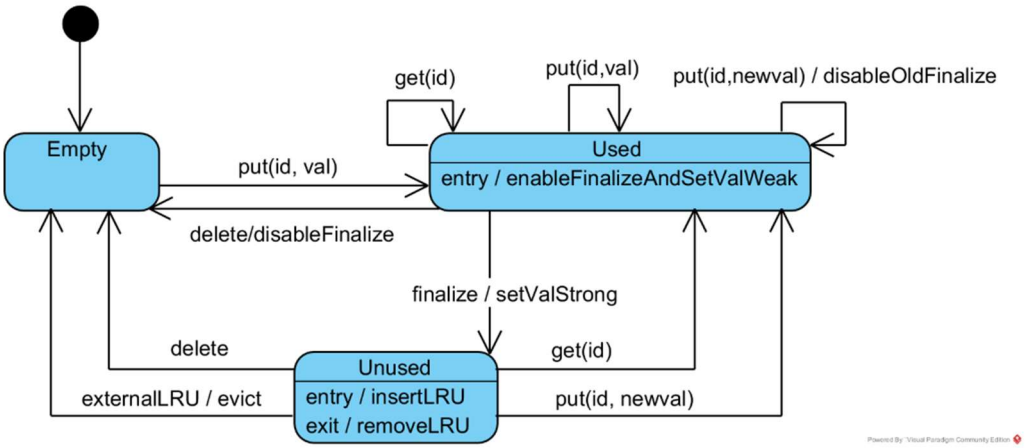

Figure 2.1: State Transition Diagram: Synchronous Model

Table 2.2 Synchronous state transition descriptions

State	Description	Key Event/Transitions
Empty	The FCE does not currently map an id to a value. Awaiting put.	The only path out of Empty is to the State. Used on a put of a new entry mapping. Aside from start, an FCE transitions to Empty when it is either deleted or evicted. An Empty FCE cannot be gotten, deleted, finalized, or evicted.
Used	The FCE value is reachable outside of FinalCache. Weakly mapped, finalize enabled. Awaiting finalize.	put and get of this entry cause no change in state as the FCE is still reachable by client. put(id, newval) is an update to the mapping. Remaining Used is a convenient optimization. Update or delete requires disabling old finalize. Otherwise, old finalize will occur for an invalid entry mapping. FCE transitions to Unused on finalize. Used FCEs cannot be evicted.
Unused	FCE value unreachable outside of FinalCache. Strongly mapped, finalize disabled. Awaiting eviction.	FinalCache's eviction heuristic is updated whenever any FCE enters or leaves this state. Update and get transition back to Used because the value is now reachable by the client. It is not possible to put an unreachable value.

The state Empty represents a FinalCacheEntry that contains no mapping between id and value. All FCEs transition to the Empty state from the Start state. FCEs can have their data cleared by a delete event or when an Unused FCE is evicted from FinalCache. The only valid transition from Empty is to Used on a put where the FCE is given a mapping of id to value. Naturally, at the time of the put, any value being added to FinalCache must be reachable by the program.

On entering the state Used, the FCE will map the id to a weak value. The weak mapping permits Finalize to execute when the referent becomes unreachable outside of FinalCache. The referent's Finalize may already have been disabled (either by the user program or having already been executed) requiring FinalCache to explicitly enable Finalize on entering the state Used. From Used, a put event of the same id to value mapping or a successful get will have no side effects and the FCE will remain in the state Used. FCEs may also have their mapping updated via a put of the same id but a new value. On an updating put event, the old value's Finalize method must be disabled. This ensures only one unique Finalize event is associated an FCE.

The execution of Finalize transitions the FCE to the state Unused; adding the FCE to FinalCache's LRU set and replacing the weak mapping with a strong mapping. If an insertion to FinalCache's LRU set causes FinalCache to exceed csize, then FinalCache will select an FCE for eviction. Otherwise the FCE will remain Unused unless it is either accessed via a put or there is a successful get. Both cases result in transitioning the FCE back to the Used state and replacing the FCE's strong mapping with a weak mapping. The eviction and delete events transition the FCE to the state Empty and the complete removal of the FCE's mapping. Additionally, the delete event requires disabling the FCE's scheduled Finalize.

2.4 Basic Synchronous Sequence

Figure 2.2 is a Sequence diagram depicting a complete lifecycle of an FCE where a new entry is added to FinalCache, is looked up, becomes unreachable, and is then evicted. For

simplicity, this example uses a csize of zero which allows for the entry to become evicted as soon as it becomes unreachable.

1. The client program makes a call to a constructor to create a new object obj. This implicitly invokes the memory manager to allocate and return the memory address to the client program The FCE is currently Empty.
2. The main thread adds obj to FinalCache. The FCE transitions to the state Used.
3. The main thread successfully gets obj from FinalCache. No change in state.
4. The main thread finishes using obj. This causes the FCE to transition to Unused via finalize. However, obj is evicted because csize is zero. Once evicted, obj will be freed by the memory manager. The sequence finishes with the FCE in the state Empty.

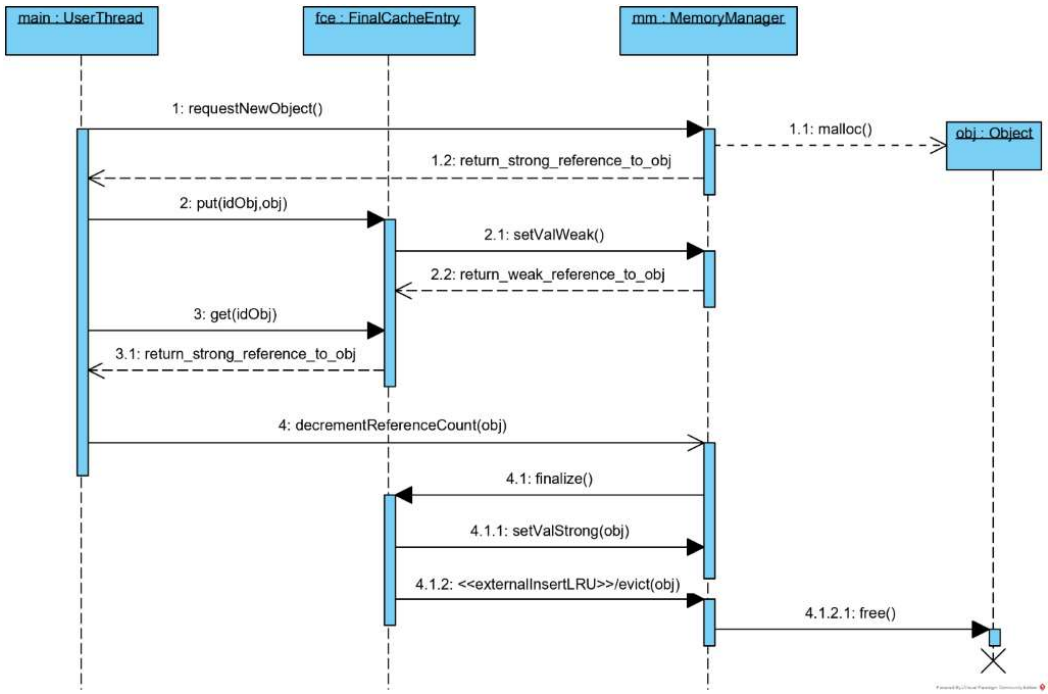

Figure 2.2: Sequence Diagram: Synchronous Model

2.5 LIMINAL MODEL

FinalCache depends on the execution of Finalize to be notified when an object has become unreachable by the program and is therefore Unused. Unfortunately, depending on the language or runtime, Finalize can either be Synchronous or Asynchronous. The type of Finalize a runtime uses can affect FinalCache's behavior.

Synchronous Finalize executes as soon as an object's reference count reaches zero. This behavior is desirable for FinalCache because it gives precise usage information; however Synchronous Finalize can have undesirable side effects as well. For example, the memory manager might pre-empt the main GUI thread to execute Finalize, causing a noticeable delay in program responsiveness. Consequently, **Asynchronous Finalize** is when memory managers choose to schedule Finalize to execute with low priority, queueing it to run when the program becomes idle. The execution of Asynchronous Finalize may be significantly delayed long past the time an object became unreachable, perhaps indefinitely, increasing FinalCache's total size.

We call this interval between the scheduling and execution of Finalize, Liminal. Liminal refers to the state between state transitions, where it is difficult to cleanly classify an entity as exclusively belonging to one state. A real-world example of Liminal is a hospital patient whose heart has just stopped beating. If not revived quickly (if appropriate) the patient will soon suffer organ failure and die. However, the patient is not officially dead until a doctor makes the pronouncement.

Accurately modeling Asynchronous Finalize requires the addition of a new Liminal state between Used and Unused. Additionally, the finalize event must also be split into scheduleFinalize and executeFinalize. Altogether, Liminal FinalCache's STD is quite similar to that of Synchronous FinalCache. These modifications are included in Figure 2.3 and Table 2.3 shown below.

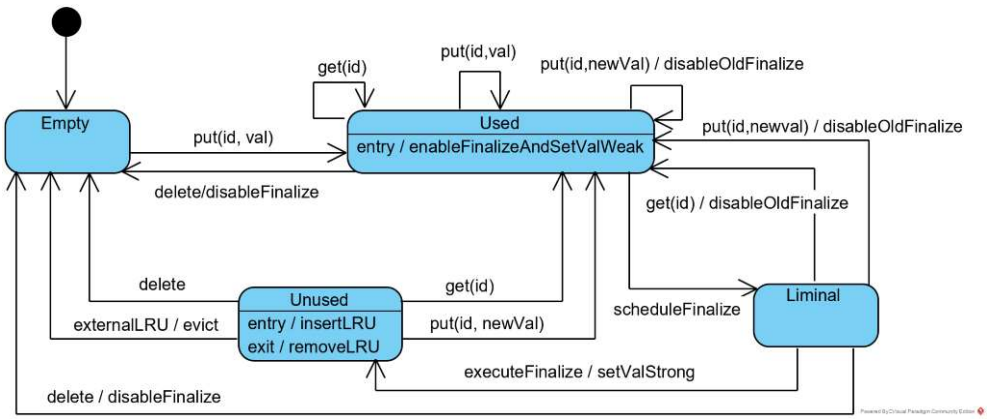

Figure 2.3: State Transition Diagram: Liminal Model

Table 2.3 Liminal addendum to Synchronous

State	Description	Key Event/Transitions
Liminal	The memory manager has identified the object as unreachable by the program. Awaiting executeFinalize.	scheduleFinalize transitions from Used to Liminal. get or put goes back to Used, disable old Finalize for correctness. executeFinalize transitions to Unused, update the mapping to a strong reference. delete transitions to Empty

The event scheduleFinalize transitions the FCE from Used to Liminal. From Liminal, the FCE can be made Used again via get or put(id, newval) events. A put(id,val) is not possible in this state because finalize could not have been scheduled if the program had at least one reference to val. Eventually, finalize will execute, transitioning the FCE to Unused. There does not exist a path from Unused to Liminal. These changes in state transitions due to Asynchronous Finalize mirror the flatlined hospital patient. The event scheduleFinalize is the flatline itself, get is a successful resuscitation, and executeFinalize is the official time of death.

2.6 Asynchronous Finalize Race Condition

Asynchronous Finalize requires an updated sequence diagram that splits Finalize into two events as shown in the previous section. Unfortunately, the memory manager does not expose the event scheduleFinalize to the user program. Consequently, FinalCache is unaware of when a FinalCacheEntry should transition to the state Liminal. Therefore, implementing the Liminal State Transition Diagram is not possible with current language APIs.

Since implementing Liminal is impossible, would FinalCache work correctly with Asynchronous Finalize using the Synchronous State Transition Diagram? Sadly, it would not. There exists at least one sequence of events, shown in Figure 2.4 below, that violates a core FinalCache design goal by incorrectly labeling an FCE's state. In a sequence that closely resembles the basic lifecycle of an FCE, a get event occurs between scheduleFinalize and executeFinalize. In so doing, the program will have a reference to an FCE that FinalCache labeled as Unused.

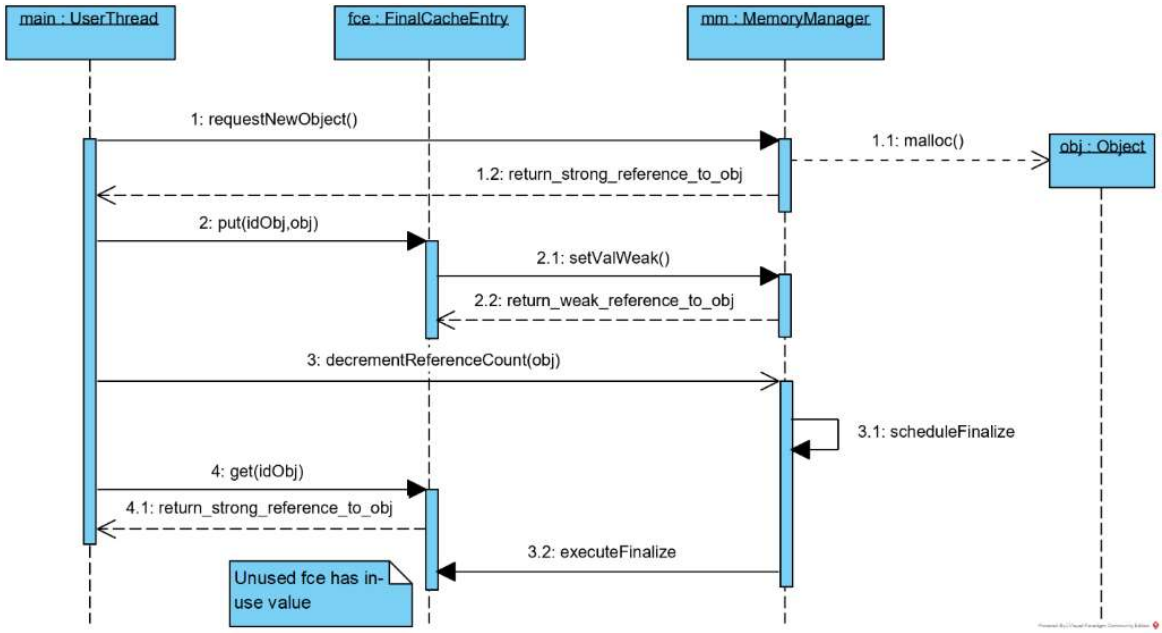

Figure 2.4: Sequence Diagram: Race Condition

Worse yet, if:

1. The FCE value remains reachable outside of FinalCache,

2. the FCE is not looked up again and,

3. csize other entries become Unused then FinalCache will evict this misidentified entry, resulting in the very situation that it was intended to avoid.

2.7 GOTTEN MODEL

As shown in the previous section, applying the Synchronous model with Asynchronous Finalize can result in a violation of FinalCache's design goals. The root of the problem is the ambiguity created due to FinalCache being unaware of when an FCE's Finalize is scheduled to be executed. If FinalCache knew when this event occurred, it could disable the scheduled Finalize. Doing so would prevent the FCE from erroneously transitioning away from the state Used.

To resolve the race condition, FinalCache should conservatively assume all get events during the state Used occur after a scheduleFinalize event. On a get, the FCE will transition to a different state that will absorb the next Finalize event. This modification is the basis for the Gotten State Machine shown below in Figure 2.5, where an FCE can only transition to the state Unused if the FCE is not gotten while in the state Used. Coincidentally this altered state machine strongly resembles the Second-chance approach to page replacement, where the head of the replacement queue can only be swapped out if its referenced bit is clear.

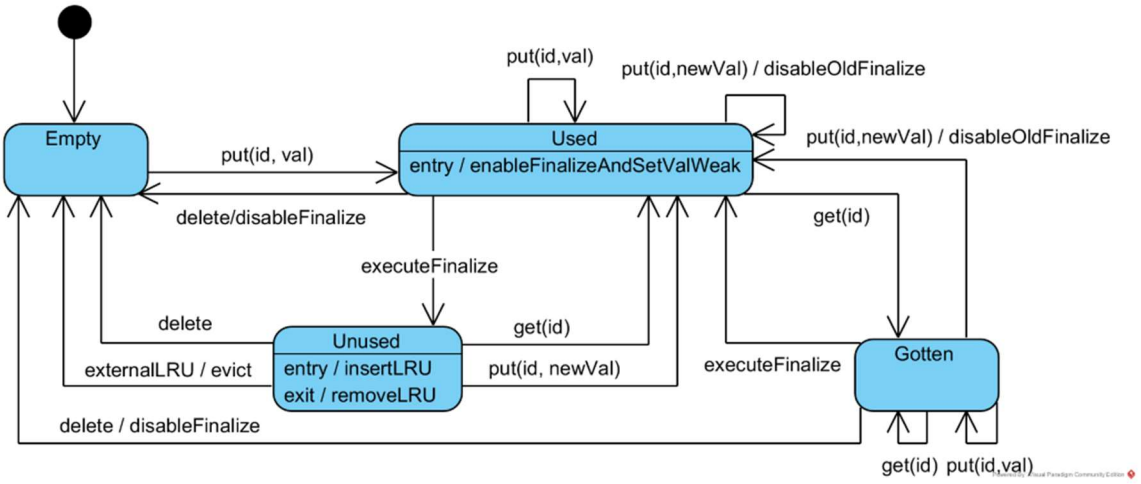

Figure 2.5: State Transition Diagram: Gotten Model

Table 2.4 Gotten addendum to Synchronous

State	Description	Key Event/Transitions
Gotten	A Used FCE was accessed. Since FinalCache does not know if Finalize was scheduled or not, this state will consume the next executeFinalize. Awaiting executeFinalize	get from Used transitions to Gotten get and non-updating put remain in Used Updating put and executeFinalize transition to Used. Delete transitions to Empty

Should a get event occur while the FCE is in the state Used, it will transition to the Gotten state rather than remaining in Used. Thereafter, any additional get or non-updating put event will not transition the FCE away from the Gotten state. There does not exist a transition between Gotten and Unused. Only an executeFinalize or updating put event can transition a Gotten FCE to the Used state (much like moving a referenced page to the tail of the queue and clearing its reference bit). In so doing, the Gotten model avoids the race condition that exists in the Liminal model.

One drawback of the Gotten state is that if a get event occurs before an object's finalize is scheduled, one additional execution of Finalize is required to transition to Unused. In fact, another valid design would be to transition directly from Empty or Unused to Gotten, requiring all FCEs to have at least two executeFinalize events before becoming Unused. However, said design is

inferior to the one pictured above because of the interaction between get and finalize. If a Used FCE is never gotten, then the race condition could not have occurred. Therefore, it is more efficient to transition to Unused if never gotten while Used.

2.8 Reference Counting Versus Tracing

In addition to whether a language runtime uses Synchronous or Asynchronous Finalize, how and when garbage collection occurs can also have an impact on FinalCache. There are two major approaches to managing memory: **Reference Counting** (RC) and **Tracing** collection. RC collectors allocate additional memory per object to record the number incoming references to an object. This count is updated every time a reference is changed and objects are recycled (or finalize is scheduled) when their count reaches zero. Tracing collectors visit all reachable objects from the set of named variables. Those objects that could not be reached are recycled. More information on memory management is available in Appendix B. [17]

Thankfully, FinalCache under the Gotten model, will work correctly regardless whether the runtime uses a RC or Tracing garbage collector. Therefore, no redesign is needed. However, the runtime may influence FinalCache's hit rate depending on the ordering of get, scheduleFinalize, and executeFinalize events.

Consider the sequence of events shown in Figure 2.6 below, where three FCEs become unreachable, one after the other. With an RC system, each finalize would be scheduled to run in the order with which they became unused. If Tracing collection were run very frequently, it would also identify the three entries in the order they became unused. However, this fine of a granularity is unlikely to occur in practice due to the expense of running three garbage collection cycles to only identify three unused objects. Moreover, such frequent garbage collection would negatively impact program responsiveness.

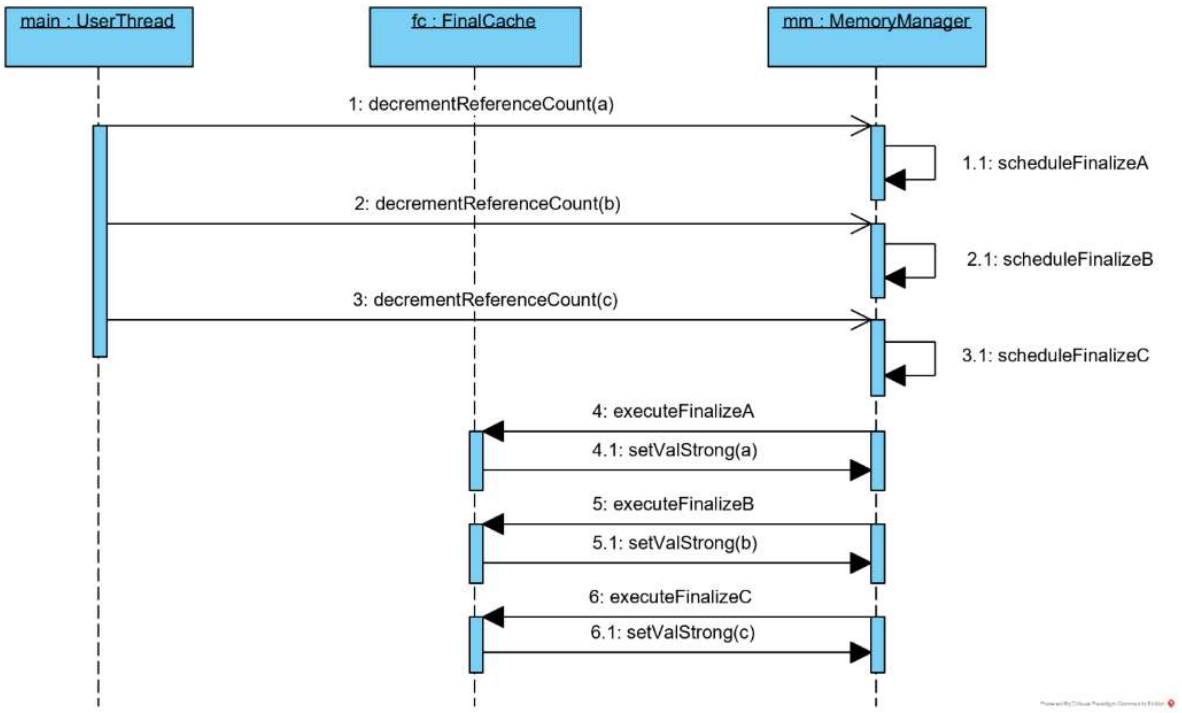

Figure 2.6: Sequence Diagram: Three unreachable entries with RC and Asynchronous Finalize

The more probable scenario, shown in Figure 2.7 below, is all three entries would be identified as unreachable during the same garbage collection cycle. However, the order with which the runtime schedules their Finalize methods is essentially random. The implication being that Tracing FinalCache's eviction ordering will not be the same as an RC FinalCache.

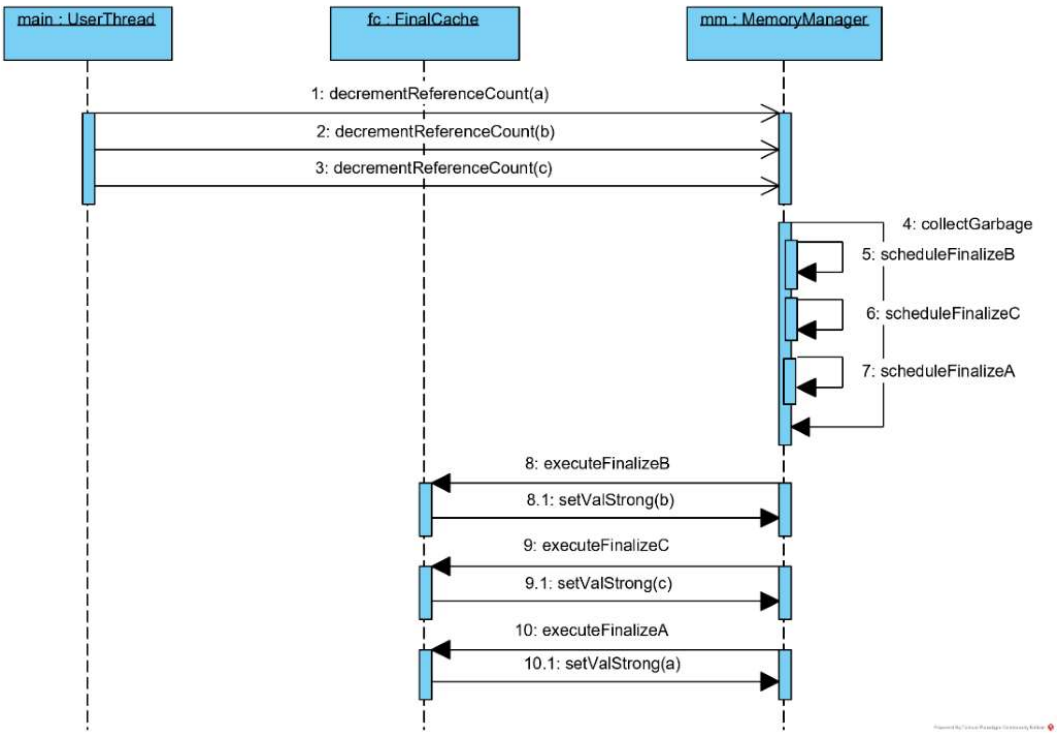

Figure 2.7: Sequence Diagram: Three unreachable entries with an infrequent Tracing collector and async finalize

The interval between garbage collection cycles means that FCEs will remain in the Used state far longer with a Tracing Collector than they would have with RC. Consequently, a Tracing FinalCache's total size may exceed that of a RC FinalCache for the same csize possibly leading to a better hit-rate, at least until the next garbage collection cycle. If too many FCEs are finalized, they could overflow csize, resulting in a mass eviction and a temporary reduction in hit rate.

In effect, the interval between garbage collections downgraded FinalCache's eviction heuristic from LRU to NRU. If csize was equal to 2 then, RC FinalCache would evict entry a whereas Tracing FinalCache will evict entry b even though entry a became reachable prior to b. Getting a more accurate reachability ordering with a Tracing collector requires a short interval between garbage collection cycles. Unfortunately, frequent garbage collections will negatively impact program response times and result in low yields of reclaimable memory per cycle. This topic is explored in greater depth in Chapter 4.

2.9 Notes on Implementation

My prototype FinalCache required many specific language features to perform as intended. Unfortunately, the implementations of these features varied by language and even runtime. We found only one language, C#, that contained all the following:

Weak References

To allow entries to become identified as unreachable, FinalCache could not maintain Strong References to them. While phantom references would have worked in this regard, their referent is not reachable by FinalCache making them unsuitable for this application. Many languages provide Weak References including Java, Python, and C#. [21–23]

Unsafe/Long Weak References

An entry can be looked up during the interim between when it becomes unreachable and when FinalCache is notified via finalize. Normally, Weak References are cleared by the memory manager when their referent becomes unreachable. This behavior could result in a cache miss if the entry is requested during this interval. Unsafe Weak References persist until the referent is recycled by the memory manager. Using these allows FinalCache to hit in this situation. It is crucial that no other Unsafe Weak References exist to FinalCache entries outside FinalCache. Violating this requirement could result in an object being in the state Unused but still reachable by the program outside FinalCache. C# was the only language we are aware of that provided this feature.[21]

Object Resurrection via Finalize

FinalCache uses finalize to become informed of when entries become unreachable. Traditionally, the next step after becoming so is being reclaimed by the memory manager. However, FinalCache resurrects the object by making it Strongly reachable again by FinalCache

itself. Reference Queues are usually the de-facto means of being notified an object has become unreachable however they cannot be used to resurrect the object. Many languages allow for object resurrection including Java, Python, and C#, while some languages, such as Objective-C, expressly forbit it. [22,24–26]

Infinite Resurrection

A FinalCache entry may be looked up again after becoming Unused, transitioning the entry back to the state Used. The FCE may then remain in FinalCache until it becomes unreachable again. Unless Finalize runs again, the entry would not be resurrected. Java and Python only execute Finalize once per object. C# can execute Finalize every time an object becomes unreachable. [25–27]

Enable/Disable Finalize

When an entry mapping is updated or deleted, the old object's Finalize is no longer necessary. We could have set a flag that indicated the Finalize should do noting when it eventually ran. However, it is much more efficient to disable the object's finalize execution entirely, a feature of C#. Doing so requires enabling Finalize for the current mapped entry when they transition to the state Used. It is also imperative that the client program does not enable/disable an FCE's finalize method. [25]

Explicit GC Invocation

In the case a program's runtime uses a Tracing collector, testing FinalCache can be unproductive as the programmer cannot know when the garbage collector ran (or even if it ran at all). Some runtimes include the ability to explicitly invoke a garbage collection cycle. However, this call may just be implemented as an advisory notice to the system that now might be an appropriate time to run the collector, though the runtime is free to ignore it as is the case with Java.

C# however, provides an explicit GC invocation which will initiate a background garbage collection. [28]

Wait for Finalizers to Run

Along the same vein, it does no good testing FinalCache if FCEs never have their finalize methods scheduled or executed. Programmers could put all program threads to sleep to facilitate the running of the thread responsible for managing the Finalize queue, however this approach seems excessive. Thankfully, C# includes a feature that blocks a thread until all Finalize methods have been run. By combining explicit GC and waiting for Finalize, it was possible to accurately test FinalCache. [28]

While FinalCache is possible to implement with current memory management mechanisms, there are several drawbacks that prevent it from being used in a production environment. We could not have built FinalCache without the exact combination of features mentioned above. For an in-depth discussion on these issues and viable solutions to them see Section 4.2. The code for the implementation in C# is available on GitHub and an in-depth discussion of its architecture is in Appendix D.[16]

Chapter 3: Empirical Study

. This dissertation's primary contribution is a novel model of usage-based caching and algorithms that implement it. The motivations for developing FinalCache was an intuition that the behavior of a usage-based cache would be superior, or at least as good as, access-based caches.

In particular, it was anticipated that

A. Unless cache capacity is very large, LRA would evict many in-use objects.
B. A significant portion of evicted in-use objects would be subsequently looked up, risking potential duplication.
C. The hit rates of LRAwe and FinalCache would be similar for similar maximum sizes.
D. FinalCache's csize parameter would be a more expressive capacity parameter than those exposed by access-based caches.

This chapter describes an initial empirical study of FinalCache's behavior to examine the validity of these expectations.

Unfortunately, FinalCache's unusual requirements preclude direct replacement and measurement strategies. As described in Section 3.2, FinalCache cannot be used as a drop-in replacement for exiting software caches. While a plethora of cache traces have been published, theses traces do not contain usage information. A purely analytic approach would require realistic models of cached object usage, and these are also unavailable.

Data supporting this empirical examination of FinalCache were derived from Firefox's logging framework, which was configured to generate traces with usage information suitable for simulating LRA, LRAwe, and FinalCache. Simulation results suggest FinalCache has better hit rate with a smaller csize than both LRA and LRAwe. This first section discusses some observations concerning entry usage and the research questions they inspire. The following two sections examine how these questions were evaluated and how trace data was generated. They are followed by three sections that present and describe results of simulations of LRA, LRAwe, and FinalCache driven by this trace.

3.1 Research Motivation and Questions

Below are the research questions and their motivating observations that will be examined in this chapter.

Observation 1: Eviction of In-Use entries does occur.

As described in Chapters 1 and 2, duplication can occur if a lookup operation for an evicted but still referenced object fails and current strategies for preventing these evictions, such as inflating cache capacity and weak eviction, are problematic.

Research Question 1: Misses of in-use entries do occur in LRA caches with sufficient capacities to have high hit rates. This hypothesis is relevant because it indicates that both FinalCache and LRAwe benefit from usage information. This hypothesis is supported by the empirical study.

Observation 2: Unlike the capacity parameters for existing caches, FinalCache's capacity parameter corresponds to the amount of additional storage consumed by the cache. FinalCache was motivated by hypothesis that usage info and meaningful capacity parameters are useful.

Research Question 2a: Total memory usage will be similar between FinalCache and LRAwe because the largest contributor to csize are in-use entries when csize is small. This is important because otherwise FinalCache would not be efficiently using memory.

Research Question 2b: To achieve a particular hit rate, FinalCache will require a smaller csize than LRAwe. Unlike LRA and LRAwe, FinalCache's capacity metric only represents memory usage by the cache.

3.2 Simulation

As described in Chapter 2, FinalCache's eviction heuristic is informed by Finalize after an object has become unused. This section describes simulation results for FinalCache, LRA, LRAwe driven from trace data collected from the Firefox browser. Firefox is a widely used open source web browser with a feature-rich logging framework. Firefox's cache is aware of reachability

events. Logged operations include add, update, get, delete, as well as eviction and doom events. The next section includes details concerning log generation and descriptions of these events.

The experiment entails running simulations following the sequence of cache operations logged by Firefox for each type of cache (LRA, LRAwe, and FinalCache) using various fixed csizes (see Chapter 2). Csize can be configured either in terms of number of entries or number of bytes. Cache capacity was limited and measured in units of cache entries.

For LRA the simulation tabulated hits, misses, in-use evictions, and in-use misses. In-use misses were not recorded for LRAwe and FinalCache because they cannot occur (see Chapter 1), Max entries refers to the combination of used and unused entries which differs between LRAwe and FinalCache.

3.3 LOG GENERATION

This section describes how Firefox's diskcache trace data was generated including which events were logged, how Firefox was configured, and what activity was recorded from each interactive session. Firefox was configured to collect the events summarized in. Table 3.1. The only FinalCache event without a direct corresponding Firefox event is Finalize. FinalCache's Finalize event was simulated using Successful Eviction and Doom events.

Table 3.1 List of logged Firefox DiskCache events

Event Type	Description
Addition	Insertion of a new cache entry. May result in the eviction of existing entry or entries.
Deletion	Complete removal of a cache entry.
Access	Updates the eviction heuristic
Successful Eviction	Capacity limit exceeded requiring removing at least on cache entry determined by its eviction heuristic. Firefox will not evict an entry currently in-use. This is one of two event types used by the simulation to determine usage.

Doom	An unsuccessfully evicted entry has become unused. The entry is removed from Firefox and is the second of two event types used by the simulation to determine usage.

Accurate simulation of LRAwe and FinalCache requires prompt reporting of usage events. Like LRAwe, Firefox's cache will not delete an unused entry. To ensure that Firefox's cache will promptly evict unused entries, it was configured with a small capacity. A capacity of zero would have been ideal because it would cause Firefox to log the removal of entries immediately after they became unused. Sadly, Firefox malfunctions when its cache's capacity is reduced to zero. A capacity of 16 KB is sufficient to prevent this malfunction. This is important because a large capacity will delay the eviction of unused entries so it was important to keep this parameter low for usage accuracy.

The median website transfer size per median requested resource for desktop browsers was approximately 27KB during March 2020 according to the HTTP Archive.[43] If an entry is unused at the time of eviction, Firefox will successfully evict it. However, if the entry was still in-use at the time of eviction, a separate event "Doom" will be recorded once the entry becomes unused. User activities in the session included: reading news articles, watching videos, and browsing comics. Firefox's cache was cleared before the start of each session and Firefox's smartsize feature (which dynamically adjusts cache capacity) was disabled.

3.4 LRA

As described in Section 3.3 in-use evictions and in-use misses can occur in access-based caches. In the Firefox trace, 6665 unique cache entries were looked-up 54703 times. The results in Table 3.2 below indicate that in-use evictions and in-use misses can indeed occur with access-based eviction and that the majority of evictions were of in-use entries.

Table 3.2 LRA Results

csize	Hits	Misses	Hit Rate	Evictions	In-use evictions	In-use misses
50	10252	44451	18.74%	9186	7341	4498
100	11081	43622	20.26%	8391	6587	3669
250	11678	43025	21.35%	7702	5969	3072
500	12416	42287	22.70%	7129	5436	2334
750	13669	41034	24.99%	6638	5005	1081
1000	14185	40518	25.93%	6216	4655	565
1250	14341	40363	26.22%	5921	4406	409
1500	14491	40212	26.49%	5645	4192	259
2000	14582	40121	26.66%	4973	3633	168
3000	15028	39675	27.47%	3765	2696	0
4000	15066	39637	27.54%	2433	1957	0
5000	15169	39534	27.73%	1680	1284	0
6000	15181	39522	27.75%	673	534	0
7000	15183	39520	27.76%	0	0	0

Observe that this trace's maximum hit rate was quite poor at 27.76% or 15183 hits. Additionally, csizes larger than 3000, nearly half the universe of cache entries, yielded less than one percent total increase in hit rate. Csize 3000 is also the point at which there were no more in-use misses. This suggests that not evicting in-use entries may have a greater impact on hit rate than not evicting unused entries. Figure 3.1 below plots the number of in-use events for various csizes. Notice the risk of in-use misses is still present for csizes less than 7000 due to in-use evictions.

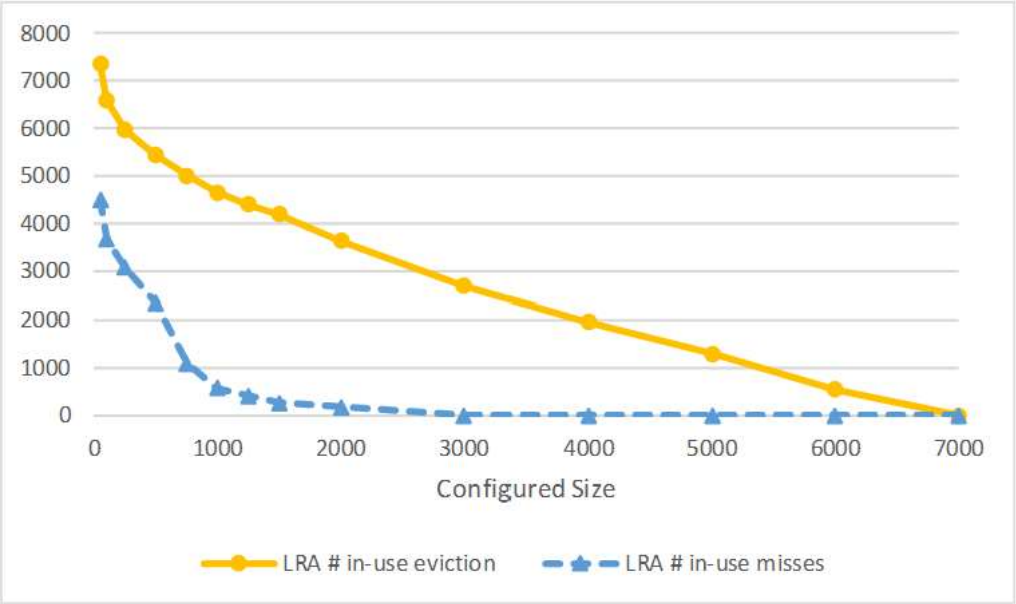

Figure 3.1 LRA in-use evictions and in-use misses

For accurate comparison with LRAwe and FinalCache, these experiments were performed with finer csize granularity between 6000 and 7000. LRA reached max hits at csize 6149 but continued to evict in-use entries for all csizes less than 6665, the total number of unique cached entries Recall that eviction is meant to reduce memory footprints and that cache capacity (csize) is an important tuning parameter. In this experiment, all csizes smaller than the universe of cacheable entries resulted in in-use evictions that do not reduce memory footprint. Further research is required to determine to the commonality this behavior.

3.5 Comparison of LRAwe and FinalCache

The previous section described the results of the simulation for LRA and showed that in-use misses and in-use evictions do occur. This section examines the results of the simulation using the same cache logs for both LRAwe and FinalCache. Recall that these two caches do not have in-use misses nor in-use evictions like in LRA. **Error! Reference source not found.** below tabulates

the number of hits and maximum total size for LRAwe and FinalCache. Four pairs of sizes resulting in similar hit rates are highlighted by color.

Table 3.3 Results of LRAwe and FinalCache

	LRAwe				FinalCache			
csize	Hits	H/C	Max Size	Unused	Hits	H/C	Max Size	Unused
0	14750	N/A	4988	0	14750	N/A	4988	0
50	14831	1.62	4992	4	14878	2.56	5038	50
100	14842	0.92	4995	7	14923	1.73	5088	100
250	14860	0.44	5022	34	15118	1.47	5238	250
500	14922	0.34	5034	46	15130	0.76	5488	500
750	14952	0.27	5085	97	15182	0.58	5738	750
1000	15087	0.34	5143	155	15182	0.43	5988	1000
1250	15122	0.30	5186	198	15183	0.35	6238	1250
1500	15122	0.25	5244	256	15183	0.29	6488	1500
2000	15130	0.19	5341	353	15183	0.22	6665	1677
3000	15182	0.14	5601	613	15183	0.14	6665	1677
4000	15182	0.11	5886	898	15183	0.11	6665	1677
5000	15183	0.09	6270	1282	15183	0.09	6665	1677
6000	15183	0.07	6527	1539	15183	0.07	6665	1677
7000	15183	0.06	6665	1677	15183	0.06	6665	1677

Throughout this section the notation of <value> : <csize> will be used to denote either a caching algorithm or a metric at a given csize. For example, LRA:0 refers to the configuration of

LRA with csize zero. Additionally, Hits:500 denotes the value of hits at csize 500 for any caching algorithm unless otherwise specified. If csize is left as a variable, such as in Hits:csize, then this denotes the hits for any particular csize and is only used here as part of an equation. For each algorithm, the rightmost "unused" columns indicate the number of unreferenced entries retained by the cache when it reached its maximum size ($MaxSize{:}\ csize - MaxSize{:}\ 0$). We also define a utility metric "hits per csize" (H/C) that represents how well the addition of csize entries above zero increased the number of hits ($\frac{Hits{:}csize - Hits{:}0}{csize}$). H/C is the relative benefit of increasing for LRAwe and FinalCache (H/C is not a valid metric for LRA as it has no $hits_0$). For example, an H/C of 1 would indicate that on average each csize entry was accessed once. We expect this number to trend towards zero as csize increases because csize will eventually grow faster than hits.

LRAwe:0 & FinalCache:0

For these two data points, both caches have the same max size. This is evidence that LRAwe and FinalCache are working as intended. LRAwe will weakly evict all entries as they added, maintaining references to entries that are still in-use by the client program. FinalCache will evict all entries as they become unused, meaning FinalCache will only reference in-use entries. As such, LRAwe and FinalCache have identical contents at every point during the simulation, resulting in both having the same number of hits and maximum size. Csize of zero also serves as a baseline comparison against other csizes.

LRAwe:500 & FinalCache:100

FinalCache needed a much smaller csize to achieve the same hit rate as LRAwe. FinalCache's H/C value of 1.73 and LRAwe's of 0.344 means that each csize entry of FinalCache was worth five times as much as the equivalent LRAwe csize cache entry.

LRAwe:3000 & FinalCache:750

FinalCache's utility has, compared to Gray, dropped to four times as high as LRAwe. It is also interesting to note that the number of unused entries in LRAwe is now much closer to that of FinalCache than it was in proportion to Gray. Both caches are near max hits though LRAwe required nearly a csize of nearly half the universe of entries.

LRAwe:5000 & FinalCache:1250

Both caches now have the maximum number of possible hits though much that was described about Blue holds true for Orange. One interesting data point is that maximum size and unused entries in LRAwe actually surpassed those of FinalCache.

The data suggests that FinalCache will have the same or better hit rate than LRAwe given the same csize. It is interesting that, at least in this session, FinalCache needed one quarter the csize of LRAwe to achieve the same hit rate. Additional testing would be required to determine if this is a general trend and is discussed in greater detail at the end of this chapter.

Figure 3.2 below considers LRAwe and FinalCache hits as a parametric function of maximum observed size. It illustrates that LRAwe and FinalCache have a similar hit rate when considered as a function of maximum size, though FinalCache's maximum size tends to be slightly greater than LRAwe's. This result suggests that FinalCache has a similar hit rate as LRAwe when they have a similar, maximum size.

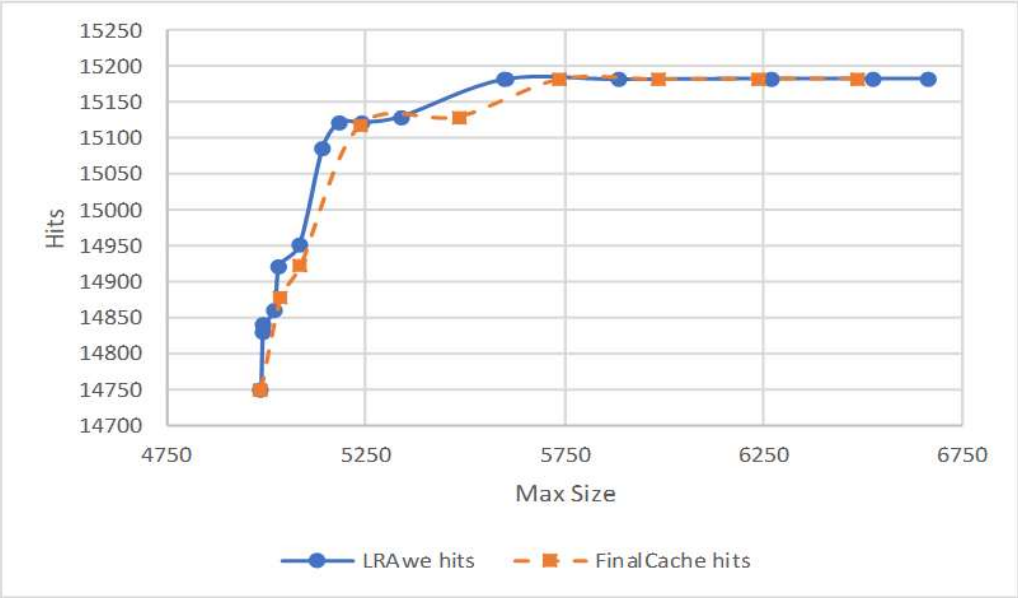

Figure 3.2 LRAwe and FinalCache hits as a function of maximum size

3.6 In-depth Discussion on FinalCache

The results from the previous two sections show that FinalCache generally had the largest maximum size. However, this data point is misleading because it does not identify when FinalCache reached this size during the course of the simulation. This section examines in-depth how FinalCache's size changes over time and why.

While the growth of the simulation could very well be caused by a memory leak, intuitively both LRAwe and FinalCache will accumulate very long-lived entries over the course of the run because some websites are ubiquitous. A cursory examination of the set of longer-lasting entries indicates that they were dominated by entries corresponding to URIs at Google, Facebook, Yahoo, and MSN. These websites are ubiquitous on the internet and is it is not surprising they remained in the cache.

Figure 3.3 below shows the cache size for FinalCache:0, LRA:0, FinalCache:750, LRAwe:3000, and LRA:3000 in intervals of fifty accesses. The horizontal axis indicates the

number of total cache accesses (as a proxy for time). FinalCache:0, which is equal to LRAwe:0, is plotted because it shows the number of in-use entries. As described in Table 3.3 above, FinalCache:750 and LRAwe:3000 have similar numbers of hits but different csizes. Of course, no discussion of LRAwe:3000 is complete without including LRA:3000 because they have the same csize.

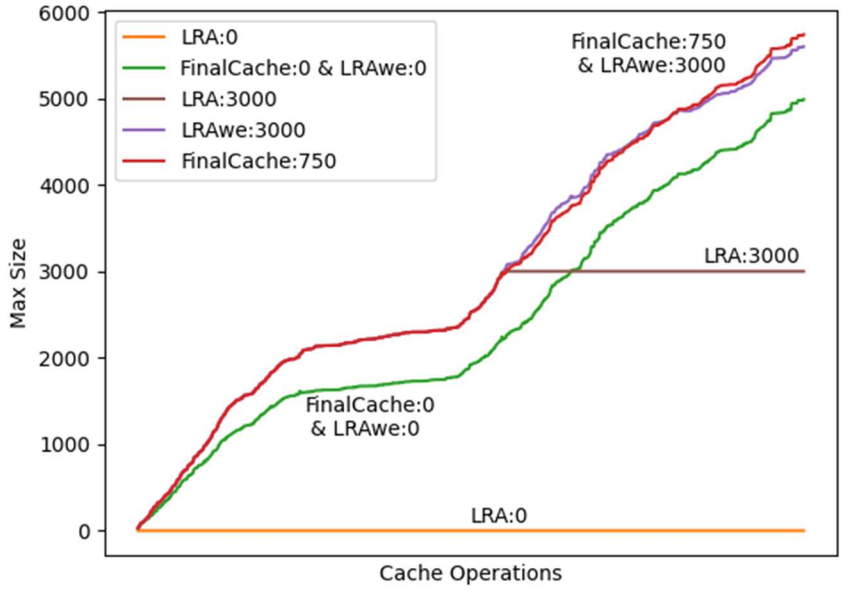

Figure 3.3 Maximum size over time. Each point is the maximum number of entries during and interval of fifty cache operations.

Maximum size increased over the course of the simulation (excluding LRA:0). This would be consistent with the presence of many long-lived entries. Analysis presented below also supports this assumption. LRA:0 evicts all entries as they are added and thus remains at maximum size zero for the duration of the experiment. After a warm up period where the caches accumulate csize cache entries, the plots begin to diverge.

FinalCache:0 and LRAwe:0 grew to approximately 5000 (4988). FinalCache:750, as expected (see Section 2.2), finished exactly 750 entries larger than FinalCache:0. LRA:3000 diverges from LRAwe:3000 and FinalCache:750 at a maximum size of 3000. This occurs because FinalCache and LRAwe adjust maximum size based on usage while LRA is fixed. While LRA:3000 may appear to be the smallest cache configuration (excluding LRA:0), it actually uses as much memory as LRAwe:3000 (and potentially more due to object duplication).

The last noteworthy point of interest occurs around maximum size 5000, where LRAwe:3000 begins to use less memory than FinalCache:750. As mentioned in the previous section, this is also the moment at which LRA:3000 had no in-use evictions. It appears to be much more than a coincidence, but this experiment was not designed to determine if LRAwe will have the same hit rate as FinalCache once there are no longer in-use evictions. As such, further research is needed to determine if this is a general trend. Unfortunately, while Figure 3.3 indicates an accumulation of cache entries, it does not show their distribution as is depicted in Figure 3.4 below.

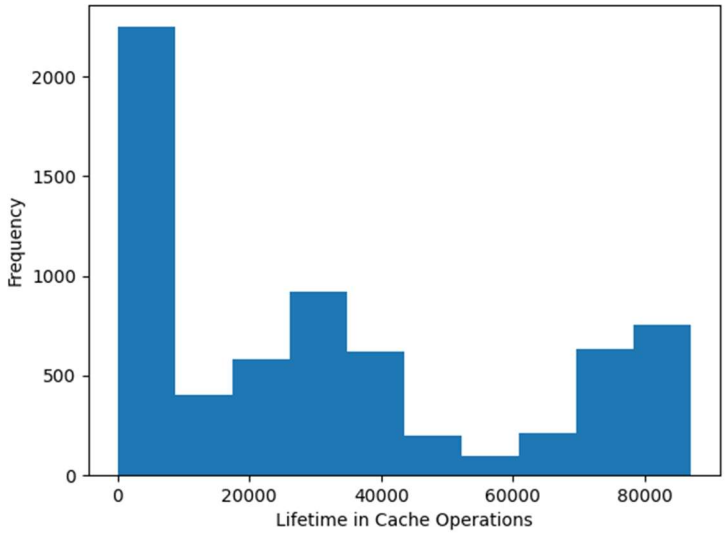

Figure 3.4 Entry lifetime for FinalCache:0 with buckets of 10000 cache operations

The largest bucket with nearly 2500 entries was for those entries that lived less than 10000 cache accesses by the end of the simulation. This is consistent with the common observation that most objects die young {cite}. There also are a significant number of long-lived entries (a third survived more than 50000 operations). These longer-lived entries are likely a source of in-use evictions.

The simulation results presented above were driven by traces that included exact usage information which enables exact LRU ordering. However, Tracing collectors create uncertainty with regards to when objects became unused. Frequent execution of tracing collection would provide more precise usage ordering but is very inefficient. As such systems with tracing collectors need to balance the anticipated yield of unused memory with the cost of running the garbage collector. Furthermore, runtime systems may indefinitely delay client notification by finalize or other interfaces. Figure 3.5 below shows the distributions of unused entries that would have been identified if tracing collection (delayed notification) were executed every 50, 250, and 500 cache operations.

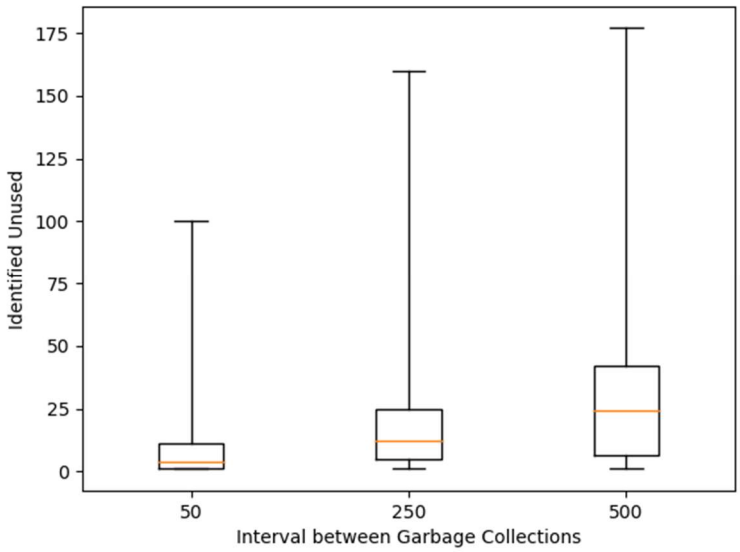

Figure 3.5 Distribution of number of entries identified as unused with various intervals between GC cycles

If garbage collection is executed frequently, then FinalCache's set of unused entries will closely approximate the exact usage information of reference counting. However, as the interval between collection cycles becomes larger, the number of identified unused entries grows as well. As such, FinalCache's eviction ordering becomes less accurate and more akin to NRU than LRU.

Ironically, when tracing garbage collection is run infrequently, FinalCache will have an inflated hit rate because entries will seldom be evicted. Unfortunately, should csize be smaller than the number of identified unused entries, then FinalCache's set of unused entries could be completely flushed, resulting in temporary loss of usage accuracy and possible poor hit rate. The extent that Tracing collectors affect FinalCache's behavior is a topic that warrants further investigation.

Chapter 4 Merit and Impact

FinalCache, as implemented, cannot be used as a drop-in replacement for existing software caches due to API differences necessitated by Finalization as described in Section 2.9. However, this prototype FinalCache can be integrated within C# programs in which a suitable Finalize can be added to cachable objects. Section 4.2 proposes a variant of Weak reference type called Liminal References based on the Liminal Model of FinalCache (see Section 2.5). Liminal references could decouple FinalCache from its entries by functioning like Weak references except in how their referents are cleared. Traditional Weak references are cleared after being identified as exclusively weakly reachable. In contrast, Liminal references would not be cleared until after all notifications have been removed from their respective Reference Queues (see Appendix C). Thus, Liminal references would allow a cached object to have multiple mappings in either one or multiple caches, in addition to allowing referent objects to be made reachable again after becoming exclusively liminally reachable.

FinalCache implicitly determines usage via a combination of Weak references and Finalize. The Weak reference allows the entry to become exclusively weakly reachable and Finalize temporarily resurrects the object, until the object is either requested or evicted. This technique, whereby a user program makes some memory management decisions, has applications beyond software caches. Sample applications are discussed in 4.3 Applications and Future Work including improved pre-mortem cleanup, object pools, user-level garbage collection, and custom reference types. The implicit tracking of object usage via the garbage collector creates some interesting opportunities for future research.

This work primarily focused on the viability of FinalCache and did not quantify how well it performs under various workloads. Nor did we investigate alternative eviction heuristics such as

frequency-based or greedy eviction in conjunction with reachability. Additionally, as shown in Section 3.6, it is important for FinalCache to identify unused objects soon after they become exclusively weakly reachable to provide accurate usage ordering. Traditionally, garbage collection has not been considered a high priority process. As such, a high priority FinalCache could find itself waiting on the low priority memory manager, a textbook example of priority inversion.

4.1 Merit and Impact

"Intellectual Merit encompasses the potential to advance knowledge and Broader Impact encompasses the potential to benefit society and contribute to the achievement of specific, desired societal outcomes." [29] FinalCache is not the first software cache to avoid the problem of in-use eviction, nor does it appear to substantially reduce total memory usage compared to weak eviction. However, FinalCache tracks cache entry usage to implement true least recently used eviction without the need for program-managed usage notification.

4.1.1 Intellectual Merit

The research described in this dissertation includes the following original observations and innovations.

Traditional LRU is actually LRA

From the perspective of the memory manager, an object is considered in-use if it is reachable from the root set. This view is unavailable to software caches unless they are explicitly informed of when an entry becomes unused. As such, extant caches approximate usage based on access history.

True LRU is possible with assistance from the Memory Manager

The memory manager identifies object usage as part of its efforts to make unused memory available to program. However, this information has traditionally not been exposed to program unless it was for the purposes of pre/post mortem cleanup. FinalCache is an example use-case where this information can drive program behavior.

Suitability of second chance strategy to resolve race condition

The interval between when an object has been identified as unreachable by the memory manager and when the program is informed of this event can results in a race condition. If an entry is accessed while Liminal, FinalCache could violate its design principles by evicting an in-use entry. This race condition can be avoided by utilizing second chance.

Languages have unsuitable APIs for implementing client-side memory management

The requirements necessary to implement FinalCache in a modern language require many features that are not often available in many languages. Even though C# included all the requisite features, the strong coupling between FinalCache and its entries to determine usage information leave much to be desired. A change to the semantics of Weak References and Reference Queues to ensure that referent objects would continue to exist until the notification is dealt with could decouple FinalCache from its entries. We call this variation of Weak References *Liminal* and they are discussed in Section 4.2.

4.1.2 Broader Impact

FinalCache's csize parameter directly adjusts FinalCache's impact on a program's memory footprint which we expect will simplify the tuning of cache size compared to LRA(we). FinalCache demonstrates the viability of client program memory management as well as the limitations of current interfaces between programming languages and their respective memory

manager. The benefits of program level memory management decisions have implications beyond caching. Section 4.3 describes examples in the domains of programming language design, memory management, operating systems, and software engineering.

4.2 LIMINAL NOTIFICATION

While FinalCache works as a proof of concept, its use of Finalize to determine entry usage impedes widespread deployment in a production environment. Each FinalCacheEntry must implement a Finalize method that informs FinalCache when the entry becomes Unused, creating high coupling between FinalCache and its entries. This also means that my prototype does not operate well with third-party objects not intended for use with FinalCache and therefore cannot be used as a drop-in replacement for existing caches (though there may exist use cases where this strong coupling is irrelevant). Ironically, FinalCache may require a name change when such an alternative exists as it would no longer be based on Finalize.

Currently, the closest alternative available to Finalize are Reference Queues as defined in Java (RQ) used in conjunction with Weak (or Phantom) references. RQs notify a program that an object became exclusively weakly (or phantomly) reachable. Unfortunately, RQs cannot be used to resurrect unreachable objects, making them unsuitable for FinalCache (see Appendix C).

We propose a modification to Weak references and RQs called Liminal References. As described in Section 2.5, Liminal refers to the state between states. In this context, liminal means the interval between when the garbage collector identifies an object as unreachable and when the program is notified. Liminal references would behave almost identically to Weak references except for how the memory manager would clear their referents. Liminal referents would not be cleared until after the Liminal reference notification has been removed from its RQ, rather than on being identified as weakly reachable. Finalize in turn, will not be scheduled until after all notifications

to a liminal object have been cleared from their RQ; making them safer to use than C#'s unsafe Weak reference. This alternative interface would obviate FinalCache's need for the Finalize interface.

Another limitation of FinalCache is related to some assumptions made to simplify its design. FinalCache requires that a unique FinalCacheEntry can exist in one only FinalCache at a time, and a referent object is mapped only once as well (see Section 2.6). Liminal references should be able to address this problem.

Unfortunately, having an entry exist in multiple caches leads to another problem. Consider two FinalCaches using Liminal references configured with csizes zero and one. Regardless of the order with which liminal notification occurs, the smaller cache will evict all entries on liminal notification, where the larger cache will not. This creates the risk of an in-use miss resulting in entry duplication. To avoid this situation both caches would either need to be in communication with each other or connected to an underlying cache which is a topic that warrants additional research.

4.3 Applications and Future Work

FinalCache is one application of utilizing the memory manager to implicitly track object reachability. The major insight behind FinalCache is that not all unreachable memory needs to be immediately reclaimed by the automatic memory manager. Rather, there are situations where some decisions should be the responsibility of the client program. Liminal references, as described in 4.2 Fixing FinalCache with Liminal References, could be used to implement the following applications of client program memory management.

Pre-mortem cleanup

Finalize was intended to be the analog to the C++ destructor for languages with automatic memory management. Invoking the destructor before the object was recycled allows for important housekeeping tasks that can only be completed while the object is still reachable, such as closing open file descriptors. Unfortunately, its unreliability and variation among languages (and even between runtimes) motivated developers to utilize other strategies such as Phantom references and try-with-resource blocks. Unlike Finalize, Phantom references can only be used for post-mortem cleanup. Try-with-resource blocks can perform pre-mortem cleanup by automatically calling close, but only if the resource object only exists in the scope of its try-block and the object implements the autoclose contract.

Object Pools

A program might choose to limit the number of instances of a class as in the Object Pool design patterns. Object pools can be controlled by a counting semaphore so that when a thread needs one of its objects, the thread will block if no objects from the pool are available. Checking an object back-in releases the lock and reset the object's state for the next time it is checked-out.

Unfortunately, this explicit check-in of objects can lead to a memory leak. Should the program fail to return a checked-out object, the object pool will continue to consider the object as checked-out. A solution to this problem could be to use Phantom references to release the lock and create a new pooled object when the original became garbage collected. However, if the object pool wanted to only ever create an object once, this approach is not a valid solution. Alternatively, the object pool could use Weak references and resurrection via custom Finalize, much like FinalCache. However, this approach carries all the problems mentioned in pre-mortem cleanup.

User level garbage collection

Liminal notification could be used to implement user level memory management. This use case does add some overhead; however, it would have the advantages of not requiring explicit tracking nor unsafe memory references compared to traditional custom allocators. Because of liminal references low coupling, user level memory management could be useful as an assignment for students in system classes as well as a debugging tool.

Custom reference types

Java is the only language we found that includes multiple levels of non-strong reachability including Weak, Soft, and Phantom references. Liminal references could be used to create custom non-strong reference types without requiring a bespoke memory manager. For example, a reference type between Weak and Soft. If client programs had access to memory statistics, such as the amount of free space versus allocated, then Liminal references could be used to implement Soft or Priority references.

Unfortunately, Liminal references do not yet exist in the form described in 4.2 Fixing FinalCache with Liminal References. Further research is required to formally define, design, and implement this feature into a modern programming language like Java. Additionally, my preliminary results from Chapter 3 suggest more avenues of research listed below.

Csize and LRA

The results from Section 3.4 show that LRA risked in-use evictions for all csizes less than the total number of unique entries. However, there were no in-use misses for csizes above 3000, and LRAwe:0 reached a maximum size of 4988. Above what csize is an LRA cache likely to experience an in-use miss? Does ensuring that no in-use entry is ever evicted mean csize must equal the total number of unique entries?

A Priority Inversion

An object belonging to a very high priority thread needs to be identified as unused as soon as possible. A tracing collector may not run sufficiently often or with high enough priority to satisfy this constraint. Or the object in question became tenured and not likely to be identified as unreachable until after a complete garbage collection cycle in a generational collector. Essentially, a high priority thread is at the mercy of the memory manager. Formally expressing and solving this priority inversion is worth exploring.

Delayed Notification

Recall that FinalCache's eviction heuristic prioritizes retention of recently used objects. This prioritization will be incorrect if delayed notification obfuscates (or permutes) the sequence of unreachability reported to FinalCache. Even if no high priority threads ever needed an entry from FinalCache, delayed notification due to infrequent garbage collection or low priority Finalize, there still exists an impact on FinalCache itself. As described in Section 3.6, FinalCache still relies on accurate usage information to determine eviction ordering. Therefore, it is important to assess the possible impact on hit rate as FinalCache accumulates unused entries only to suddenly have its contents flushed.

Profiling FinalCache

This dissertation focused on exploring the design, implementation, and feasibility of FinalCache. How well exactly FinalCache performs with diverse types of workload in an actual runtime was not within the capabilities of my experiment. Additionally, this simulation only considered strict usage for eviction ordering. How would altering the eviction heuristic to be frequency-based or greedy-based affect FinalCache's performance?

4.4 SYNOPSIS

Software caches that utilize heuristics that do not include entry usage risk evicting in-use entries and entry duplication. Using weak eviction ensures that in-use entries will remain reachable by the cache but do not ensure entries will temporarily persist after becoming unused. FinalCache not only avoids the problem of in-use eviction, it also ensures that all entries persist until csize other entries become unused.

FinalCache also exposes the flaw in equating csize with memory footprint. Traditional LRA's csize includes both in-use and unused objects, but a cache's contribution to program memory footprint is in the unused objects it maps. When LRAwe's csize is greater than zero, LRAwe cannot determine which entries are currently unused and contributing to total memory footprint. However, FinalCache knows its impact on memory is csize because those entries are unused.

LRA's csize is also its maximum size but this too can be misleading as LRA risks entry duplication. LRAwe's maximum size is the sum of csize and all in-use weakly evicted entries. In contrast, FinalCache's maximum size is the sum of csize and all in-use entries. Table 4.1 below summaries csize and maximum size for LRA, LRAwe, and FinalCache.

Table 4.1 Summary of csize and maximum size

	LRA	LRAwe	FinalCache
csize	in-use and unused	in-use and unused	unused
maximum size	csize(officially)	csize + evicted in-use	csize + in-use

In a runtime with Synchronous Finalize, an FCE requires the three states Empty, Used, and Unused. However, Asynchronous Finalize splits the Finalize event into scheduleFinalize and executeFinalize. Accurately representing this split requires updating the model with new events

for Finalize and a new Liminal state. Unfortunately, client programs are unaware of when the event scheduleFinalize occurs and therefore cannot implement this model. Instead, FinalCache must use a Second Chance algorithm to ensure that a get event does not occur before Finalize could have been scheduled, avoiding a race condition.

There are also several language features necessary to implement FinalCache aside from Finalize and Weak References. These include unsafe Weak References, resurrection via Finalize, and the ability to enable/disable Finalize among others (see Section 2.9). As far as we are aware, C# is currently the only language suitable to build my prototype FinalCache.

Preliminary results show that in-use eviction and in-use misses are a problem for LRA, even with large values of csize. LRAwe and FinalCache follow a similar trajectory of hit rate as a function of maximum size, but FinalCache required a smaller csize than LRAwe to achieve the same hit rate. Though not a feature of the trace data, by using different intervals of accesses, we were able to extrapolate how FinalCache would behave if used with a Tracing collector. The more frequently garbage collection is run, the better the approximation of Reference Counting eviction ordering.

FinalCache exemplifies the case for some client-side memory management. Implicitly determining usage information can be done safely even with current program/memory manager interfaces, though Liminal References would help in this regard. Further research is required to profile FinalCache's performance with a larger data set and assess the impact of delayed notification. Additionally, work is needed to design and implement Liminal references to make FinalCache viable as a drop-in replacement for existing caches.

References

[1] Alexandra Barros and R Ierusalimschy. 2008. Eliminating Cycles in Weak Tables. *J. UCS* 14, 21 (2008), 3481–3497. Retrieved September 22, 2013 from http://www.jucs.org/jucs_14_21/eliminating_cycles_in_weak/jucs_14_21_3481_3497_barros.pdf

[2] Ludmila Cherkasova. 1998. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. Retrieved from http://www.hpl.hp.com/techreports/98/HPL-98-69R1.pdf?jumpid=reg_R1002_USEN

[3] SMS Daula, KES Murthy, and GA Khan. 2012. A Throughput Analysis on Page Replacement Algorithms in Cache Memory Management. *Int. J. Eng. Res. Appl.* 2, 2 (2012), 126–130. Retrieved April 23, 2014 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.417.1916&rep=rep1&type=pdf

[4] Brian Goetz. 2005. Java theory and practice: Plugging memory leaks with weak references. Retrieved September 26, 2013 from http://www.ibm.com/developerworks/java/library/j-jtp11225/

[5] Erik G. Hallnor and Steven K. Reinhardt. 2000. A fully associative software-managed cache design. *ACM SIGARCH Comput. Archit. News* 28, 2 (2000), 107–116. DOI:https://doi.org/10.1145/342001.339660

[6] John L Hennessy and David a Patterson. 2006. *Computer Architecture, Fourth Edition: A Quantitative Approach*. DOI:https://doi.org/10.1.1.115.1881

[7] Arun Iyengar. 1999. Design and performance of a general-purpose software cache. *Performance, Comput. Commun. ...* (1999). Retrieved from

http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=749456

[8] Brazil) Leal, Marcus Amorim (PUC-Rio and Brazil) Ierusalimschy, Roberto (PUC-Rio. 2005. A Formal Semantics for Finalizers. *J. Univers. Comput. Sci.* 11, 7 (2005), 1198–1214. Retrieved from http://jucs.org/jucs_11_7/a_formal_semantics_for/jucs_11_7_1198_1214_leal.pdf

[9] Donald Michie. 1968. Memo Functions and Machine Learning. *Nature* 218, April 6, 1968 (1968), 20–22. Retrieved August 2, 2020 from https://www.cs.utexas.edu/users/hunt/research/hash-cons/hash-cons-papers/michie-memo-nature-1968.pdf

[10] Druv Mohindra and Will Snavely. 2015. MET12-J . Do not use finalizers. Retrieved from https://www.securecoding.cert.org/confluence/display/java/MET12-J.+Do+not+use+finalizers

[11] Iliyan Nenov and Panayot Dobrikov. 2005. Memory sensitive caching in java. In *International Conference on Computer Systems and Technologies*, 1–6. Retrieved September 22, 2013 from http://ecet.ecs.ru.acad.bg/cst05/Docs/cp/SII/II.20.pdf

[12] Peter Norvig. *Technical Correspondence Techniques for Automatic Memoization with Applications to Context-Free Parsing*. Retrieved August 2, 2020 from https://www.aclweb.org/anthology/J91-1004.pdf

[13] Diogenes Nunez, Samuel Z Guyer, and Emery D Berger. 2016. Prioritized garbage collection: explicit GC support for software caches. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2016*, 695–710. DOI:https://doi.org/10.1145/2983990.2984028

[14] Anvita Saxena. A Study of Page Replacement Algorithms. *Int. J. Eng. Res. Gen. Sci.* 2, 4 . Retrieved July 29, 2020 from www.ijergs.org

[15] Michael A Steele, Sylvia L Halkin, Peter D Smallwood, Thomas J Mckenna, Katerina Mitsopoulos, and Matthew Beam. Cache protection strategies of a scatter-hoarding rodent: do tree squirrels engage in behavioural deception? DOI:https://doi.org/10.1016/j.anbehav.2007.07.026

[16] Adrian Veliz and Aleksandr Diamond. FinalCache GitHub. Retrieved March 8, 2020 from https://github.com/adrianveliz/FinalCache/tree/master

[17] PR Wilson. 1992. Uniprocessor garbage collection techniques. *Mem. Manag.* (1992). Retrieved September 22, 2013 from http://medcontent.metapress.com/index/A65RM03P4874243N.pdf

[18] Necko/Cache - MozillaWiki. Retrieved April 12, 2018 from https://wiki.mozilla.org/Necko/Cache

[19] new_lru. Retrieved from https://github.com/memcached/memcached/blob/master/doc/new_lru.txt

[20] LruCache | Android Developers. Retrieved from https://developer.android.com/reference/android/util/LruCache.html

[21] Weak References. Retrieved from https://msdn.microsoft.com/en-us/library/ms404247(v=vs.110).aspx

[22] 8.8. weakref — Weak references — Python 3.4.3 documentation. Retrieved June 24, 2015 from https://docs.python.org/3/library/weakref.html

[23] WeakReference (Java Platform SE 7). Retrieved March 14, 2018 from https://docs.oracle.com/javase/7/docs/api/java/lang/ref/WeakReference.html

[24] finalize() - NSObject | Apple Developer Documentation. Retrieved from https://developer.apple.com/reference/objectivec/nsobject/1418513-finalize

[25] Object.Finalize Method (System). Retrieved from https://msdn.microsoft.com/en-us/library/system.object.finalize(v=vs.110).aspx

[26] Object (Java Platform SE 7). Retrieved March 14, 2018 from https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#finalize()

[27] PEP 442 -- Safe object finalization | Python.org. Retrieved March 13, 2017 from https://www.python.org/dev/peps/pep-0442/

[28] GC Class (System) | Microsoft Docs. Retrieved July 29, 2020 from https://docs.microsoft.com/en-us/dotnet/api/system.gc?view=netcore-3.1

[29] GPG Chapter III. Retrieved August 5, 2020 from https://www.nsf.gov/pubs/policydocs/pappguide/nsf13001/gpg_3.jsp

[30] cache_1 noun - Definition, pictures, pronunciation and usage notes | Oxford Advanced Learner's Dictionary at OxfordLearnersDictionaries.com. Retrieved July 29, 2020 from https://www.oxfordlearnersdictionaries.com/us/definition/english/cache_1

[31] Kwang's Haskell Blog - Memoization in Haskell. Retrieved August 2, 2020 from https://kseo.github.io/posts/2017-01-14-memoization-in-hasekll.html

[32] CLiki: memoization. Retrieved August 2, 2020 from https://www.cliki.net/memoization

[33] F Sharp Programming/Caching - Wikibooks, open books for an open world. Retrieved August 2, 2020 from https://en.wikibooks.org/wiki/F_Sharp_Programming/Caching

[34] The try-with-resources Statement (The Java™ Tutorials > Essential Classes > Exceptions). Retrieved March 13, 2017 from https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

[35] Reference (Java Platform SE 7). Retrieved from https://docs.oracle.com/javase/7/docs/api/java/lang/ref/Reference.html

[36] Firefox. Retrieved from https://www.mozilla.org/en-US/firefox/

[37] Java SE Development Kit. Retrieved from https://www.oracle.com/java/technologies/javase-jdk14-downloads.html

[38] Download Python. Retrieved from https://www.python.org/downloads/

[39] NumPy. Retrieved from https://numpy.org/

[40] Matplotlib. Retrieved from https://matplotlib.org/

[41] Download .NET. Retrieved from https://dotnet.microsoft.com/download

[42] Git Download. Retrieved from https://git-scm.com/downloads

[43] 2020. HTTP Archive. Retrieved from https://httparchive.org/reports/page-weight?start=2020_03_01&end=2020_04_01&view=list

Appendix

These Appendices cover background information and related work relevant to FinalCache. FinalCache is not the first cache to utilize Weak References as part of its eviction heuristic and it is important to acknowledge the foundational work that has influenced its development. Appendix A discusses cache's in general, including the differences between hardware and software caches, as well as different eviction heuristics. Appendix B provides a high-level overview of Memory Management algorithms. Lastly, Appendix C defines reachability and describes the various mechanisms available to programmers by the Memory Manager to determine reachability.

APPENDIX A: CACHES

Caches are a collection of stored items that may be used at later date. The word comes from French *cacher* meaning "to hide".[30] A real world example of caches are the stockpiles of acorns made by squirrels as food for the winter.[15] A similar concept exists in the world of computing where the items are mappings of unique ids to values.

In hardware, a cache is dedicated fast memory in the CPU which stores a mapping of memory address to its copied value from RAM. Before a request is made to RAM, the CPU first checks if the requested item exists in cache. Successfully finding the entry in cache, called a hit, means the CPU does not need to stall waiting for the value to be copied from RAM. Otherwise, the value will be stored in the cache once it arrives from off-chip in the hope that future accesses will hit.[6]

Software caches behave in a similar manner as mentioned above, though their entries exist at higher level of abstraction. Examples include files from secondary storage or records from a

network database rather than simply memory addresses and values. While hardware caches have a physical memory limit inside the CPU, software caches can be configured to use as much RAM as required based on the needs of the application.

Another difference between hardware and software caches is how they evict entries. Since space is limited, caches must eventually evict entries, ideally those that will not be looked-up again after eviction. As such they tend to rely on the principle of temporal locality to evict the least recently accessed entry. While software caches can easily track exact entry usage, doing so in hardware is impractical requiring alternative eviction strategies.

Hardware

Computer programs live in secondary storage with their instructions and data which must be loaded into a small amount registers in the CPU. To accomplish this memory is structured in a hierarchy. Each level up the hierarchy contains a subset of data from the level below (excepting temporary values). Traveling up the hierarchy means faster, smaller, more volatile, and expensive memory. The hope is to speed up computation by having data that the CPU is likely to need as close to the CPU as possible. Figure A.1 below visualizes this hierarchy as a pyramid.

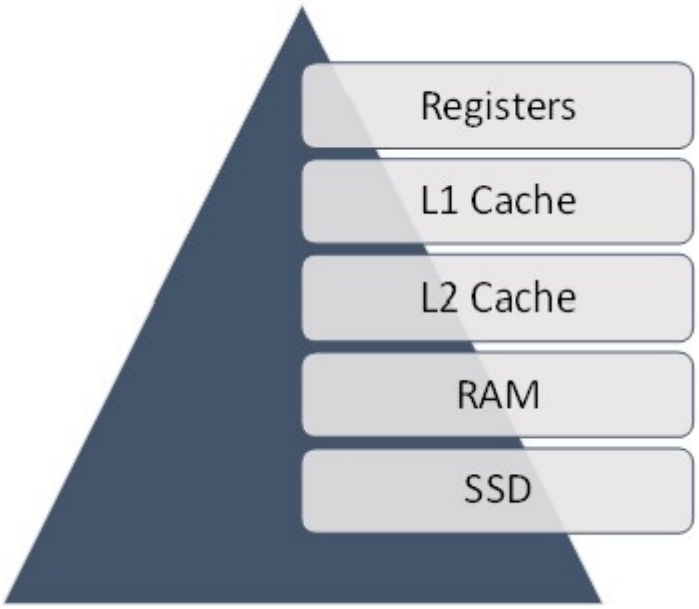

Figure A.1 The Memory Hierarchy

L1 cache is usually shared among two CPU cores and generally holds words of memory[6]. L2 cache is off-chip and shared amongst all CPUs and contains blocks of RAM*. Not having a value readily available in a register when the CPU needs it means recursively accessing the layer above until SSD. Each time the above layer does not have the value requested is called a miss and there are three types of misses.

Cold

Occurs the first time an entry is requested. This happens en masse when a program is initialized as no relevant values will be present in the cache. A common technique to avoid these misses is to pre-fetch all the values a program is likely to need before the program beings to run.

Capacity

The requested entry used to be in the cache but was evicted. This miss would not have occurred if the cache had a larger capacity. At some point however, increasing the size of the cache becomes cost prohibitive and some capacity misses are unavoidable.

[6] Oversimplified but associativity unnecessarily complicates this dialog and does not have an analog in FinalCache.

Conflict

This is where two or more different entry mappings must be stored in the same location. For example, if the cache hold sixteen entries and stores entries based on their address modulus sixteen, then addresses 0x10 and 0x20 will conflict and cannot simultaneously exist in the cache.

Software

Software caches share many properties with their hardware counterparts with some domain specific differences. These are the types of entries they hold, configurable capacity, response times, and eviction heuristics. The latter are discussed in their own dedicated section.

Types of Entry

Software cache entries can map IDs (also known as keys) to objects rather than addresses to values. Examples include database records, files, or larger inverted matrices. The value stored by the cache is likely a reference to object, rather than the original object itself.

Capacity

Hardware capacity is set during manufacturing. Conversely, software cache capacity can be dynamically adjusted during runtime to grow as large/small as needed. Capacity can be defined as the total number of entries or in terms of bytes. If the later, then to make room available when a new entry is added to cache may require evicting multiple entries. The former is much simpler as it would be one eviction per addition if the cache were full. More on eviction in the next section.

Response Time

When a request is made to the L1, the CPU must get a response on the order of clock cycles or stall. L2 is slower than L1 but still much faster than RAM. Software caches in contrast, exist in RAM and may return objects from unloaded pages, files from secondary storage, or data from over

the network. As such, a hit in a software cache can take thousands of clock cycles to return an entry.

Eviction Heuristics

All caches evict entries to limit the amount of memory they use. Which entry they evict though depends on the cache's goals, the information available to it, and program behavior. Many of these heuristics as based on the principle of temporal locality, which uses access history to predict future accesses. The following are common cache eviction heuristics.[3,14]

Least Recently Accessed (LRA)

A pure application of temporal locality. The evicted entry is the one which was accessed furthest in the past. This heuristic is commonly known as Least Recently Used but a name change was required to differentiate this evict heuristic from FinalCache's actual usage-based heuristic. Simple to implement in software but not likely to be found in hardware caches.

Least Recently Used (LRU)

Another pure application of temporal locality. The evicted entry is the one which was used furthest in the past and is FinalCache's eviction heuristic. Entries which are currently in-use are not included in the set of entries which are eligible for eviction. This is feasible in software but not possible in hardware caches.

Not Recently Accessed (NRA) / Second Chance

In some cases, simply knowing whether an entry has been accessed recently can be a suitable eviction heuristic, especially in hardware caches. This can be implemented by periodically clearing each entry’s "accessed" flag and then setting it on access. If the cache needs to evict an

entry it simply chooses one whose flag has not been set. The "Clock" algorithm for RAM page replacement is a well-known example of NRA combined with Second Chance.

Most Recently Accessed (MRA)

MRA is the exact opposite of LRA. This unusual heuristic may be useful in scenarios where there's loop iteration which might cause LRA to perform poorly. Once the loop terminates, the cache can switch to another heuristic like LRA.

Greedy

Some entries cost more to create than others or have higher priority. The pure greedy approach evicts the entry with the least priority. The drawback to pure greedy is that high priority entries may never get evicted. Greed Dual-Size combines Greedy and NRU by reducing the priority of cache entries overtime and increasing priority of individual entries when they are accessed. [2]

Frequency-Based

This heuristic segregates entries based on how often they are accessed, the principle being that entries which are accessed frequently are not ideal candidates for eviction. Memcached is frequency-based and starts entries in the cold set. Popular entries are transferred to the hot set. After which, they are transferred to the warm set on losing popularity. Entries are evicted using LRA from the cold and warm sets.[19]

Memoization and Caches

“The term memoization was coined by Donald Michie (1968) to refer to the process by which a function is made to automatically remember the results of previous computations.” Memoization and caches both trade space for potential future savings. Many programming languages include support for memoization including Haskell, Common-Lisp, and F#.[31–33].

Strategies for memorization include lookup tables, maps and caches. One limitation of memoization is that the memoized function should not have side-effects because these side-effects will not take effect if the memoized value is substituted for the function call. Additionally, memoization cannot be used if the function may return different values given the same input parameters such as random or downloading a dynamic website. [12][9]

APPENDIX B: MEMORY MANAGEMENT

Memory management is the process of reclaiming unused memory. This is important because unused memory can cause a program's total memory footprint to grow unnecessarily large and negatively impact system performance. Before this process was automated, developers needed to explicitly allocate and deallocate memory. Explicit management can result in difficult-to-identify bugs such as dangling pointers and memory leaks. Modern languages include some form of automatic memory management which relieves this responsibility from the programmer, improving reliability (need a better word). Automatic memory management, also known as garbage collection, comes in two major families: Reference Counting and Tracing. Reference counting system allocates extra memory per object to hold its reference count. This count is adjusted each time a reference is changed to/from an object. Once this count reaches zero the object can be safely recycled. Tracing collectors determine object liveness based on whether they are reachable from the set of named objects. Generally, which collector a runtime uses is irrelevant to the programmer unless they have a specific need to interface with it.[17]

Explicit

Explicit memory management requires programmers to allocate and deallocate memory. In the C programming language this is accomplished with calls to the functions malloc and free.

Explicit memory management is very efficient compared with Reference Counting or Tracing because it does not require the extra memory and updates associated with Reference Counting, nor does it need to search memory to identify liveness with Tracing. Applications which require this efficiency include embedded programs and operating systems (for now). However, explicit memory management has several drawbacks which automatic memory management avoids.

Mixing Algorithm Implementation and Memory Management

Does the algorithm for "Making a peanut butter and jelly sandwich" generally include a description of when to allocate or deallocate memory? Memory management is simply not a concern for many algorithms. As such, this logic must be included in any implementation with explicit memory management, leading to more complicate programs compared to those with automatic memory management. This both makes programs longer and more difficult to maintain especially for those new to the program or programming.

Failed Memory Allocations

In Java, new will always return a constructed object or throw an outOfMemoryException. Since this error is quite uncommon, developers do not include new in a try-catch block. Conversely, the function malloc can return a null pointer if the program runs out of heap space but doing so is not the same cataclysmic event it is in Java. Before throwing the exception, the Java Runtime will have performed a full garbage collection clearing weak/soft references, attempted to move the break, and/or compacted memory to reduce external fragmentation meaning there was truly no more memory available. If malloc fails, it is the programmer's responsibility to manage, and some strategies available to Java are not possible in C.

Dangling Pointers and Memory Leaks

A program with explicit memory management needs to free memory when it is no longer useful. The problem with doing so is determining when this occurs. Simply because one thread has finished using an object does not mean it is not still being used by another thread. Calling free prematurely could result in a dangling pointer. This is surprisingly similar to the problem of in-use eviction with LRA caches. Alternatively, if all threads using an object assume another thread will eventually free the shared object, it could result in no threads actually calling free. This situation is called a memory leak and is less likely to occur in languages with automatic memory management.

Reference Counting

Reference Counting memory management is a scheme which keeps track of the number of references that point to a given object. Additional memory is required per object to store this count. Each time a reference to an object is changed, either moved to or away from, the reference count for the referent is updated by the memory manager.

So long as this count is positive, the object is considered live and cannot be recycled. However, when the object's reference count reaches zero the object's memory can be freed; additionally, any references it may have had to other objects must be updated as well. For example, if the program clears the sole reference to the root of a tree, the entire tree will be freed.

Reference Counting avoids all the drawbacks associated with explicit memory management described earlier. Additionally, it is straightforward to implement. Lastly, Reference Counting is faster than Tracing garbage collection. Unfortunately, Reference Counting does have its own drawbacks/tradeoffs.

Slower than Explicit

Tracking each reference changes will take slightly longer than explicit memory management. This problem can be compounded with reference multiplicity. Consider a tree where n subtree roots reference the same set of n leaves. Freeing the leaves would require n*n/2 reference updates.

Increased Total Memory Footprint / Exceeding Capacity

Individually, one additional word of memory allocated to hold the reference count is basically negligible. However, this means an increased memory footprint overall. To save on memory this count could be stored in one byte rather than a word. Unfortunately, this places a limit on the maximum reference count at 256. There do exist strategies for dealing with this situation, but they are beyond the scope of this work.

Dealing with Cycles

Consider an object A which references an object B. Object B references object A. Neither object has a reference count of zero. In this scenario, both objects will remain in memory though they may be unreachable by any other object. One solution is to include a Tracing collector to find and free cycles when memory becomes limited. Another solution would be to convert B's reference from Strong to Weak which would break the cycle (more on this in Appendix C Non-Strong References).[1]

Tracing

Tracing collectors determine object liveness by identifying all the objects which are transitively reachable from the set of named objects otherwise known as the root set. Objects which are not identified in this search are therefore not reachable by the program and can be safely recycled. This is only possible because the memory manager has access to all of memory.

Tracing has several advantages compared with Reference Counting collectors. Firstly, Tracing collectors do not have to maintain reference updates*, though it does have to pay the costs of tracing, so this advantage is debatable. Secondly, it does not need to allocate additional memory for the reference count. And thirdly, Tracing collectors are invulnerable to cycles.

The costs of tracing vary because, unlike Reference Counting, there are several types of Tracing collector which have different design goals and are discussed in greater detail below.

Mark-and-Sweep

This algorithm has two phases. Marking is the process of visiting each object reachable from the root set and following their references, marking each object as it is visited. If the memory manager encounters a marked object, it does not follow it again. Once all objects have been marked, the collector sweeps through all of memory. Sweeping clears the marked flag and recycles those that were not marked.

The marked flag can be as small as one bit and included as part of the object's metadata and so does not require additional memory compared with reference counting. It is also fairly trivial to implement. One drawback of Mark-and-Sweep is it does take some time to complete because it must access most of the heap. Mark-and-Sweep also leads to external memory fragmentation as the sweeping process is likely to leave many small gaps in available memory. A compaction should be performed before throwing an OutOfMemoryException.

Copy-Compact

Compaction can actually be used as a garbage collection strategy. Copying-Compacting searches from the root set but it copies in-use objects to another section of memory. As objects are copied, their references are updated to point to the new location. Copied objects are "compacted" to create one long contiguous block of allocated memory. This eliminates any external

fragmentation but requires splitting memory to have the space available for copying. Copying can be faster than Mark-and-Sweep because it only needs to access the reachable objects rather than all heap objects, though it is slowed down due to the creation of deep copies.

Incremental

Mark-and-Sweep and Copy-Compact may noticeably impact program responsiveness. If the program is interactive, pausing the program for a garbage collection cycle may be undesirable. Incremental garbage collection allows the program to run during a garbage collection cycle. Both algorithms mentioned above can be run incrementally. This scheme uses either read or write barriers to determine if an object has been accessed during the cycle. These objects, and their descendants, are then considered live until the next cycle. This conservative approach will not reclaim all unused memory, but it may be worth it to not pause the program.

Generational

Generational, which can use either Mark-and-Sweep or Copy-Compact, is a concurrent algorithm based on the premise that new objects often die young. New objects are allocated in a section of the heap called the nursery. When there is memory pressure the garbage collector first searches the nursery for unreachable objects. If the program still requires additional memory the older generations may be collected as well. After surviving several nursery collections, the object can be transferred to an older generation. If a nursery object becomes referenced by an older generation, the garbage collector may not reclaim it because the older generations are not searched as often; though the nursery object may in-fact be unused, it will persist until its liveness can be verified. In that sense, Generational is conservative like Incremental.

APPENDIX C: REACHABILITY

One aspect that was omitted from the discussion of memory management is reachability. There are mechanisms which provide additional states between reachable and unreachable. These mechanisms allow the programmer more control over when and how objects should be recycled; the primary use of which is for object cleanup.

Finalize is a method run by the memory manager after an object has been identified as unreachable but before its memory has been reclaimed. This method is the analog of C++'s destructor. A major failing of Finalize is its implementation varies by programming language which has motivated developers to transition to alternative means of object cleanup. [10]

Non-strong references are a family of reference type which do not prevent their referents from being reclaimed by the garbage collector and need to be de-referenced before use. Referents which have already been recycled cause the non-strong reference to return null, rather than returning a new strong reference. These referents types can be used to break cycles, prevent memory leaks, and manage object cleanup.

Non-strong references can be used as part of software caches. Nenov and Dobrikov[11] created a cache which would evict entries to soft references and is the basis for what we call weak-evict caches. Their cache used soft references to allow the memory manager to maintain unused entries so long as there was an excess of memory available. Nunez, Guyer, and Berger developed a custom garbage collector and reference type specifically to be used by software caches called priority references which informs the garbage collector about the relative priority of cache entries.

Try-with-auto-close is another alternative to object cleanup. This is a contract an object makes by implementing Closeable. If this object is created in a try block, before the try-catch-finally block completes, the system will call the object's Closeable method. This is similar to

Finalize except the object is still reachable by the program when close is called. More detailed information is provided in Appendix C: Auto-close.[34]

Finalize

Finalize is a method associated with an object which has traditionally been used for pre-mortem cleanup, such as closing an object's sockets, before it is reclaimed by the memory manager. In many ways it is the automatic memory management equivalent to C++'s Destructor; however, there are some important distinctions namely:

- Programs cannot call Finalize, only the garbage collector can
- Scheduling of Finalize varies between languages and runtimes
- Objects may be resurrected, made reachable again by the root set, via Finalize

The usage of Finalize has become discouraged in favor of alternatives discussed later. This is because, as alluded to earlier, the implementations of Finalize are inconsistent and developers cannot depend on it. The following lists many of the reasons programmers choose to avoid Finalize.[10]

When to run Finalize?

Executing Finalize as soon as objects are identified as unused may unduly delay high priority threads from running. As such, the memory manager may queue Finalizers to run when the program is idle. Unfortunately, if the program never idles then no Finalize methods will ever be run, making Finalize unreliable.

There are mechanisms to explicitly inform the memory manager that Finalize should be run but these vary by language. C# has a method waitForFinalize which blocks the thread until all Finalizers have been run. Java had a feature called RunFinalizersOnExit but it has been disabled since Java 5.

Additionally, the order with which Finalizers are run is determined by the Garbage Collector. If one Finalize needs to be run before another it could create a race condition which would be easily avoided in C++.

To Resurrect or not to Resurrect?

Object resurrection support is language dependent. Objective-C expressly forbids object resurrection via Finalize. Java and Python allow resurrection but only once because Finalize is run at most once per object. C# can resurrect objects indefinitely. [24–27]

Non-Strong Reference Types

Non-Strong References are a family of Reference types which do not prevent the memory manager from recycling their referents. These References are sometimes called Weak References, but this is slightly inaccurate as Weak References are one form of Non-strong Reference. It would be more accurate to state that Non-Strong References are weaker than Strong. Traditional, or Strong, references keep their referents alive so long as the referrer is reachable.

Implementations of Non-Strong References vary by language, but all referents must first be de-referenced from the referrer object before they can be used. This de-referencing generates a new Strong Reference to the referent object. If the referent object has become unused, this de-referencing will return null. A simple analogy of this behavior is Schrodinger's cat. The programmer does not know if the referent object still exists (the cat is alive or dead) until they de-reference it (open the box).

In Reference Counting systems, a Strong Reference increases an objects reference count by one. Conversely, Weak References do not affect Reference Count. This behavior is useful when breaking cycles as depicted in Figure C.1 below.

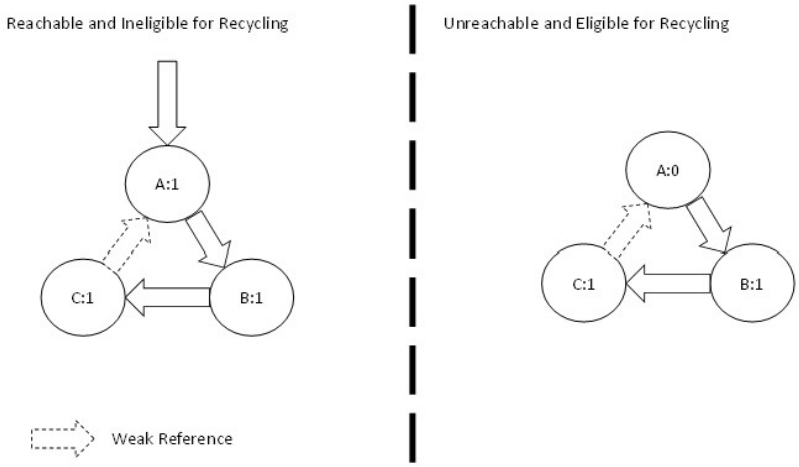

Figure C.1 Weak References break cycles

Tracing collectors determine reachability slightly differently. If a live object is reachable by at least one Strong Reference, the object is considered Strongly Reachable. Otherwise, its reachability is determined by the strength of the remaining Non-Strong References. Soft References are stronger than Weak but less so than Strong. Softly reachable objects persist longer than Weak so long as there is excess memory available. Phantom References are the weakest Non-Strong Reference type and they are unique in that they cannot be de-referenced. The figure below lists the reachability levels of objects in Java.[35]

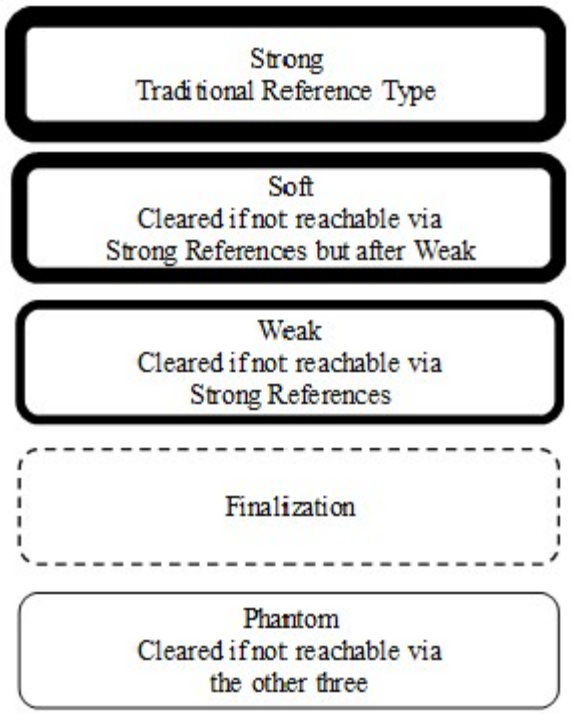

Figure C.2 Strength of various reference types in Java

Java also includes a mechanism called a ReferenceQueue where the memory manager will enqueue Reference objects when their referents become Exclusively Soft, Weak, or Phantom Reachable. This is useful for post-mortem cleanup for example, a programmer might use a Phantom Reference to be notified that a very large object is no longer resident in memory before beginning to load in another very large object. Non-Strong referents are generally cleared when they are no longer strongly reachable for safety. C# includes an unsafe Weak Reference object which is not cleared until after Finalize has been run and FinalCache could not have been built without it.

Caches and Non-Strong References

Non-Strong references are sometimes used as an ad-hoc cache, Soft References in particular. The Soft Reference acts as the (key, value) mapping with eviction determined by the

garbage collector. However, Non-Strong References can also be used in conjunction with proper software caches.

Weak Keys and Values

LRUCache provides two mechanism for utilizing Weak References with subtle but important distinctions between them. Weak Values are similar to the mapping described above except using Weak References instead of Soft. If the value becomes unused by the program before it is evicted, the mapping will be cleared. Alternatively, LRUCache can use Weak Keys instead which will do the same when the key becomes unreachable.[20]

Nenov and Dobrikov

The precursor to FinalCache, Nenov and Dobrikov created what we are calling a Weak-Evict cache. The goal of their cache was to take advantage of unused memory by maintaining Soft References to evicted entries. So long as the system had no pressure on memory, evicted objects could persist in the cache which increases the hit rate. Our simulation used Weak References instead, where entries would be cleared as soon as they became unused. It would have been nice to include Soft References in our testing but that would have introduced lots of variability/uncertainty into the mix. [11]

Nunez et. al.

Priority References build on the work done by Nenov and Dobrikov. The problem with Garbage Collector dependency for reclaiming Soft References is that the cache has no control over which entries are freed and when. Their solution was to build a custom Soft Reference type which specifies the relative and/or global importance of each cache entry. In so doing, the Garbage Collector could make informed decisions concerning which entries should be freed and when.[13]

Auto-close

AutoCloseable is a contract in which the Java runtime will, when the object goes out of scope, automatically call close() on the resource and is a form of pre-mortem clean-up. This differs from Finalize in that close() is not placed on a separate queue to be dealt with when the program is idle. Rather, the method is called immediately. Additionally, this feature cannot resurrect objects. AutoCloseable is best used with the try-with-resources block which was first introduced in Java 7 and updated in Java 9.[34]

Try-with-resources differs from the traditional try-catch-finally block in that it 1) will invoke close for all resources inside the block and 2) better manages thrown exceptions. Most of what would traditionally go in the finally block can be omitted in the try-with-resources block though the latter can still use finally if desired. In Java 7, all AutoCloseable resources need to have been declared inside the block. Java 9 included enhancements where resources declared outside the block would be automatically closed as well so long as the programmer included them inside the try block's parenthesis.

APPENDIX D: ARCHITECTURE AND IMPLEMENTATION

The source code implementing FinalCache and the simulators that generated results presented in this dissertation are published as [16] and are freely available on GitHub . This Appendix describes the organization, architecture, implementation, and build requirements of FinalCache and accompanying simulation at the time of publication. Aleksandr Diamond was the primary author of the simulation while an undergraduate student.

Below is a list of required software to re-create our results and/or demo FinalCache.

- Firefox (version 35 or later)[36]

- Java JDK (version 8 or later)[37]
- Python 2.x (you may need to migrate if using Python 3.x)[38]
- numpy[39]
- matplotlib[40]
- .NET Core[41]
- bash or similar (Windows now includes the ability to run bash via the wsl)
- git (optional)[42]

fire_logs

This directory contains the data used to generate the graphs from Chapter 3. Due to the individual file size limit from GitHub, this ~500MB file required partitioning into ~50MB chunks. Before running the simulation to verify the results from Chapter 3, this original file must be reconstructed. All other files in the directory must also be deleted otherwise the simulation will include them as well (a planned but not implemented feature was the ability to run more than one simulation at once).

Prototype

The file Program.cs has three classes: Program, MyCache, and MyCacheableObject. Program contains the main method which creates an instance of MyCache and several MyCacheableObjects. The program inserts these objects into the cache and then processes to lose the references to all but one of them. This remaining object is gotten between GC and finalize requiring a second finalize to become unused. MyCache and MyCacheableObject largely behave as described in Chapter 2 except this implementation uses a sentinel boolean Gotten variable internal to the object rather than a Gotten state. This variable is set on access, reset on Finalize, and an entry can only become unused when this flag is not set. Functionally, this scheme is

equivalent to the Gotten model from Section 2.7. Note that this implementation is not optimized for performance nor is thread safe as these features were superfluous to showing that FinalCache was possible to implement in existing runtimes.

Simulation

The parents for the majority of these classes are LRUCache and TestUtils. LRUCache includes the basic functionality of a cache that is extended for the three cache types and TestUtils contains the features necessary to interpret the data for each cache. Each cache type includes a Test class that executes the simulation with various csizes. FiguresTest provides an overview of the types of cache operations included in the raw data. HistogramTest, IntervalFinalCacheTest, and SizeTest generate output files with specific information for graphing a histogram of cache entry lifetimes, a box and whisker plot of the number of identified unused entries per tracing collection/delayed notification, and a line graph of several cache configurations overtime.

Tools

These scripts are used to both generate the raw data for simulation as well as filter the raw data for debugging. fire_log.sh configures Firefox with a limited cache size and disables smartsize to prevent Firefox from adjusting the initial configuration. It also clears the existing cache to ensure each run is independent. Be sure to save the raw data from a prior run before executing this script.

Vita

Adrian Edward Veliz received his Bachelor of Science in Computer Science from The University of Texas at El Paso (UTEP) with a Minor in Mathematics in 2011. He also received his Master's in Computer Science from UTEP in 2019 and will receive his Doctor of Philosophy from UTEP in 2020. Adrian was also awarded an ASPIRE Fellowship in 2019 and the Google-CAHSI Dissertation and Faculty Start-Up Award in 2020.

Adrian worked as a Software Developer for several companies beginning as an undergraduate student. He maintained COBOL programs on a mainframe that processed coupons for Currey Adkins. He also developed a Section 508 compatible website to teach reading comprehension strategies for Instructional Support Services at UTEP. Adrian twice interned at State Farm Insurance where he maintained PL/1 and Java programs. He worked at ExxonMobil where he created known vulnerability-free templates for deployment of small projects.

Before entering Graduate School, Adrian worked as a Peer Leader on a research project aimed at introducing high school students to Computer Science topics via their graphing calculators in Algebra class. Adrian continued with the project for two years during graduate school before investigating Android's memory management system for possible improvements to user responsiveness and battery life. During the course of this research, Adrian found his current research topic of improving program interaction with the automatic memory manager.

Adrian has been a Teaching Assistant for several systems classes including Computer Architecture, Theory of Operating Systems, and Computer Networks and Protocols. He also teaches high school students introductory Computer Science topics as part of Microsoft Philanthropies TEALS program. Adrian will soon teach Computer Science at Olympic College in Bremerton Washington.