Spring 5-15-2016

Exploiting the Weak Generational Hypothesis for Write Reduction and Object Recycling

Jonathan Andrew Shidal
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Ron Cytron, Chair
Roger Chamberlain
Patrick Crowley
Viktor Gruev
Krishna Kavi

Exploiting the Weak Generational Hypothesis for Write Reduction and Object Recycling

by

Jonathan Andrew Shidal

A dissertation presented to the
Graduate School of Arts and Sciences
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

May 2016
St. Louis, Missouri

Contents

List of Tables

List of Figures

Acknowledgments

I would like to first thank my advisor, Ron Cytron, for continued guidance, encouragement, and friendship over the course of my time in graduate school. I feel fully prepared for the next steps in my career, in large part due to everything that I have been able to learn from Ron over the past few years.

I would like to thank my committee members: Roger Chamberlain, Patrick Crowley, Viktor Gruev, and Krishna Kavi. Also, I would like to thank Anne Bracy. The suggestions and advice they have given me throughout this process have been invaluable.

I cannot thank the office staff enough for all of their help, support, and for always keeping me organized: Jayme Moehle, Kelli Eckman, Myrna Harbison, Sharon Matlock, Lauren Huffman, Cheryl Sickinger, and Madeline Hawkins. They are real life savers.

Lastly, I would like to thank my friends and colleagues in the department for making my time in graduate school a thoroughly enjoyable experience, both in and out of the office.

Thank you NSF for supporting this work under grant 1237425.

Jonathan Andrew Shidal

Washington University in Saint Louis

May 2016

This dissertation is dedicated to my parents for always giving me love and support.

ABSTRACT OF THE DISSERTATION

Exploiting the Weak Generational Hypothesis for Write Reduction and Object Recycling

by

Jonathan Andrew Shidal

Doctor of Philosophy in Computer Science

Washington University in St. Louis, May 2016

Research Advisor: Professor Ron K. Cytron

Programming languages with automatic memory management are continuing to grow in popularity due to ease of programming. However, these languages tend to allocate objects excessively, leading to inefficient use of memory and large garbage collection and allocation overheads. The *weak generational hypothesis* notes that *objects tend to die young* in languages with automatic dynamic memory management. Much work has been done to optimize allocation and garbage collection algorithms based on this observation. Previous work has largely focused on developing efficient software algorithms for allocation and collection. However, much less work has studied architectural solutions. In this work, we propose and evaluate architectural support for assisting allocation and garbage collection.

We first study the effects of languages with automatic memory management on the memory system. As objects often die young, it is likely many objects die while in the processor's caches. Writes of dead data back to main memory are unnecessary, as the data will never be used again. To study this, we develop and present architecture support to identify dead objects while they remain resident in cache and eliminate any unnecessary writes. We show

that many writes out of the caches are unnecessary, and can be avoided using our hardware additions.

Next, we study the effects of using dead data in cache to assist with allocation and garbage collection. Logic is developed and presented to allow for reuse of cache space found dead to satisfy future allocation requests. We show that dead cache space can be recycled at a high rate, reducing pressure on the allocator and reducing cache miss rates. However, a full implementation of our initial approach is shown to be unscalable. We propose and study limitations to our approach, trading object coverage for scalability.

Third, we present a new approach for identifying objects that die young based on a limitation of our previous approach. We show this approach has much lower storage and logic requirements and is scalable, while only slightly decreasing overall object coverage.

Chapter 1

Introduction

The memory wall (the gap between processing and storage speeds) [70] remains of concern to computer architects and application developers. In terms of a storage hierarchy, the wall becomes less of a performance problem when computation can be shifted toward the local cache memories and away from main memory. However, two trends are driving main memory sizes and bandwidth to continue to increase. First, adding more processing cores on-chip continues to increase stress on main memory, as each core requires additional space and bandwidth. Second, the memory resources required by modern applications is increasing. For example, applications with automatic memory management often require a much larger memory footprint than the equivalent application with explicit memory management. Currently DRAM is used as main memory in most systems, however its scalability remains a question [1]. The inclusion of higher-density storage components for main memory (such as phase-change memory [56, 40, 71, 27]) increases the storage capabilities of a platform, but with an associated increase in access time and/or limitations on how often the storage can be written before wearing out (write endurance). Techniques to eliminate unnecessary writes can therefore help alleviate the high write costs of these storage devices as well as increase the lifetime of the devices.

The first goal of this work is to reduce write bandwidth to main memory, easing write bandwidth requirements of main memory as well as mitigating the high cost of writes in emerging memory technologies.

Automatic memory management has greatly reduced the burden of memory management to the programmer, allowing the programmer to focus full attention on design and other aspects of software engineering. Because of this, object-oriented languages with automatic memory management are continuing to grow in popularity. However, automatic memory management often uses memory inefficiently, requiring a larger memory footprint than an equivalent program executing with explicit memory management. In fact, to achieve similar performance, applications with automatic memory management require a $5X$ larger memory footprint than equivalent applications using explicit memory management. When given only a $2X$ larger memory footprint, automatic memory management degrades application performance by almost 70% [31]. Also, without tuning of the heap size and garbage collector, programs may suffer even larger allocation and garbage collection overheads [72, 38].

The second goal of this work is to reduce memory requirements for applications with automatic memory management without degrading performance.

Much research has been done on developing efficient object allocation and garbage collection algorithms [65, 47, 28, 2, 3, 12]. Most work in this area focuses on efficient software algorithms, however much less work considers hardware solutions [35, 55, 17, 51].

This thesis focuses largely on architectural additions specialized for object-oriented languages with automatic memory management. We aim to reduce the additional stress these languages place on the memory system by simultaneously reducing memory bandwidth as well as reducing the total amount of memory space used. An important observation has been

made about object-oriented programs. The *weak generational hypothesis* observes that most objects die young in these programs [65, 47]. We exploit this property throughout this work; in fact we rely on it to be successful. In the following sections, we give a brief introduction to each chapter and note the contributions each chapter makes towards our goals of reducing write bandwidth and reducing memory costs of applications with automatic memory management.

1.1 Contributions

1.1.1 Identifying Dead Data and Eliminating Writes

This Chapter presents work in our paper titled *"Trash in Cache"*, published in the *Proceedings of the 2014 workshop on Memory Systems Performance and Correctness* [62]. I was the first author and primary contributor for this work.

Prior work has shown that many writes out of the processor's caches are write-backs of dead data [14, 34, 59]. These writes are unnecessary, as dead data does not need to be maintained by the memory system. For example, [59] shows that up to 60% of the writes to main memory can be eliminated by simply allowing the garbage collector to communicate dead address ranges to hardware via a *clean* instruction. Prior work has waited for the explicit deallocation of memory space before notifying hardware of dead memory addresses (via an explicit deallocation by the programmer or via the garbage collector post collection cycle). Although successful at eliminating a large portion of writes to main memory, prior approaches suffer from *rot time* (the time between object death and the discovery of its

death). Rot times can be quite long, allowing many dead (but not known dead) cache lines to be evicted.

In this Chapter, we exploit the *weak generational hypothesis* to target young objects and proactively identify them as dead, allowing elimination of useless writes much earlier in the memory hierarchy. Hardware for *Cache Only Reference Counting* is presented. Reference counting [23], a technique for identifying dead objects without *rot time*, is implemented in hardware and limited to objects in the cache only. As reference counts are not maintained for objects post eviction from cache, our approach relies on objects dying young. As a preview, approximately 23% and 50% of writes out of the L1 and L2 caches respectively can be eliminated on average over several Java benchmarks.

1.1.2 Recycling Dead Data

This Chapter presents work in our paper titled *"Recycling Trash in Cache"*, published in the *Proceedings of the 2015 International Symposium on Memory Management* [63]. I was the first author and primary contributor for this work.

In this Chapter, we evaluate recycling of dead objects to satisfy future allocation requests. Additional hardware is presented to allow recycling of dead cache space identified by *Cache Only Reference Counting*. Recycling dead objects can facilitate reduced heap sizes or reduced garbage collection time, reducing the overall memory space needed by a program. Beyond reducing heap pressure, our approach to recycling also has positive effects on the memory subsystem. Recycling requests satisfied by our approach are known to be resident in cache, therefore no cache faults are caused by object initialization and overall cache miss rates are decreased. As a preview, an average of 27% and 45% of the total allocation requests from

a program can be satisfied by recycling dead cache space for L1 and L2 caches respectively. 12% of the total bytes requested are satisfied by recycling previously dead bytes.

In this Chapter, we also discuss and evaluate realistic, off the critical path, hardware implementations of *Cache Only Reference Counting*. We study how fast our reference counting logic must be relative to the cache in order to remain effective.

1.1.3 Designing Effective Hardware: Trading Coverage for Throughput

This Chapter presents ongoing work and work in our paper titled *Hardware-Based Reference Counting of the Youngest Generation*. This paper is currently under peer review. I was the first author and primary contributor for this work.

Cache Only Reference Counting proved difficult to implement with the required throughput to identify dead objects before they were evicted from cache. However, it showed many objects do die before eviction from cache and the benefits to the memory system can be significant.

In this Chapter, we present a novel hardware structure for identifying dead objects and recycling them. Our approach is largely based on *Cache Only Reference Counting*. However, we explore trading coverage of reference counting (less objects are reference counted at any given time) for reduced hardware costs. As a preview, our limited approach eliminates 85% of the writes that *Cache Only Reference Counting* eliminates from an L1 cache. However,

on chip storage cost is reduced from over 40 KB to under 1 KB. We show that our limited approach can be implemented to achieve throughput requirements necessary to remain effective when all proposed hardware is moved off of the processor's critical path.

Chapter 2

Identifying Dead Data and Eliminating Writes

2.1 Background

The memory wall (the gap between processing and storage speeds) [70] remains of concern to computer architects and application developers. In terms of a storage hierarchy, the wall becomes less of a performance problem when computation can be shifted toward the local cache memories and away from main memory. However, as more processing cores are being added on chip, main memory sizes and bandwidth continue to increase. The gaining popularity of object-oriented programming with automatic memory management only stresses the memory system further, as these programs generally require more memory space and incur more cache misses than an equivalent program run with explicit memory management [72, 38, 31]. Currently DRAM is used as main memory in most systems, although its scalability remains a question [1]. The inclusion of higher-density storage components for main memory (such as phase-change memory [56, 40, 71, 27]) increases the storage capabilities of a platform, but with an associated increase in access time and/or limitations on how often the storage

can be written before wearing out (write endurance). Techniques to eliminate unnecessary writes can therefore help alleviate the high write costs of these storage devices as well as increase the lifetime of the devices.

2.1.1 Useless Writes: Silent Stores

One class of unnecessary writes is *silent and temporally silent stores*. A silent store refers to a write that overwrites a previous value with the exact same value. [41] shows a significant amount of writes to main memory issued by some common benchmarks are silent (20% to 68%). One study [42] showed that efficient techniques can eliminate between 31% and 50% of these silent store instructions, implying a total decrease in writes of 6% to 34% of the benchmarks' *total* store instructions. The idea of silent stores has been generalized to *temporally silent stores* [43]. Such store instructions write a value to memory that changes its contents temporarily. A subsequent store instruction will revert the stored value to a previous value of interest, perhaps one that was once stored in memory or one that is available (but perhaps invalid) in another processor's cache. In a study of the latter case, over 40% of the communication misses due to supposedly invalid cache lines can be avoided if the stores that cause the invalidation are determined to be silent. This is particularly significant for multi-threaded applications running on multicore systems. A related effort [29] sought to decrease memory traffic that was introduced to accomplish reference counting. Reference counts often change briefly and then return to a previous value. This form of Lepak's temporally silent stores [43] takes advantage of knowing that the stores are due to (compiler-generated) reference counting. The experiments conducted using that idea squashed almost all of the increase in memory traffic that was attributable to reference counting. However, those studies were conducted without an accurate cache model.

2.1.2 Useless Writes: Dead Data

This chapter focuses on eliminating writes of dead data. Memory hierarchies in modern systems typically use write-back caches. In a write-back cache, on a store instruction, only the cache is modified (as opposed to a write through cache, where the value is written into the cache and into the next levels of memory). Each cache line maintains a dirty bit. If the cache line is modified the line becomes dirty and must therefore be written back to the next level of memory upon eviction from the cache. This is necessary to maintain correctness, as the modification to the line occurred only in the cache. Without being written back, important information could be lost, as the next level of memory has a stale value for the data stored in the evicted line. However, as dead data is no longer needed by the program, there is no need to maintain its correctness throughout the memory system. Therefore, any write-back of dead data is useless. If the address range can be discovered dead while in cache, special instructions much like PowerPC's *Cache Management* instructions [33] can be used to *invalidate* or *clean* (clear the dirty bit) cache lines that contain dead data. These lines will then be prevented from being written back to further levels of memory. Several previous works use similar instructions to reduce writes in this way; however, the approach presented in this chapter for identifying data that is dead is novel. [14, 34, 44] study languages with explicit memory allocation and deallocation, such as C++. In these works, an address range is known to be dead upon an explicit memory deallocation by the programmer, a *delete operation* in C++.

In applications with explicit memory management, data is often left to be deallocated at the end of a program even though it may not have been used for some time. The time between the actual death of data and the discovery of the data being dead has been termed *rot time* [16]. While data is rotting, it may be evicted from cache, leading to unnecessary writes

the previous work would not prevent. The focus of this chapter is to identify dead data as soon as possible (with zero *rot time*) in order to maximize the amount of write bandwidth that can be eliminated.

Where [14, 34, 44] focus on programming languages with explicit allocation and deallocation, we target languages that have automatic dynamic memory allocation such as Java. Languages with automatic dynamic memory management are becoming mainstream for programmers today. The burden of maintaining memory is moved away from the programmer, allowing the programmer to focus on algorithm design rather than memory allocation and deallocation. Programs using automatic memory management generally follow what is known as the *weak generational hypothesis*, allocating many short-lived objects (objects that die soon after allocation) [65, 47]. The excessive allocation of short-lived objects, or *bloat*, in these languages presents a significant opportunity to eliminate useless writes, as many objects die before eviction from cache. Researchers have tried to reduce the *bloat*, of programs by static and dynamic analyses [10, 9]. To date, the tools resulting from that research have not been made available.

In order to eliminate the useless writes caused by short-lived objects, we must first have a technique to identify objects as dead. Languages with automatic memory management consist of an allocator and a garbage collector. When the program requests storage for an object, the allocator is queried for a free chunk of heap space. The garbage collector, on the other hand, identifies dead objects and returns the dead heap space to the allocator to be reused. However, current garbage collection techniques are not ideal for eliminating useless writes of dead data. In the next section, current garbage collection techniques are discussed. We then propose a novel approach that is based on a well known garbage collection technique, with the purpose of identifying and eliminating useless writes of dead data in Section 2.2.

2.1.3 Garbage Collection Techniques

Garbage collectors use *reachability* to determine an object's liveness. The collector maintains a *root set* of structures available to the executing program that may contain references to objects allocated in the heap. The root set includes the execution stack, registers, and global variables. If an object is reachable from the *root set* then it may be accessed by the running program and is therefore live. However, if the object is unreachable, it can provably no longer be accessed by the running program and is dead. The heap space occupied by the dead object can then be returned to the allocator. There are two main classes of garbage collection algorithms, *reference counting* [23] and *tracing* [50]. We give a brief introduction to each type of algorithm here. A more thorough introduction to garbage collection techniques can be found in [66].

Tracing

The most prevalent garbage collection algorithms in modern systems are *tracing algorithms*. Tracing schemes work by periodically following pointers out from the *root set* and into the heap. Objects directly referenced by the *root set* are marked as live and are scanned for references to other objects. Those objects are then visited, and in this way references are recursively traced through the heap, marking all reachable objects as live. The unmarked areas of the heap are then known to be unreachable and therefore dead. Although incremental tracing algorithms have been developed, typically *stop-the-world* collectors are used. As the program may modify references, *stop-the-world* collectors prevent the live set of objects from changing during the collection cycle. [59] studies the effectiveness of communicating dead address ranges to hardware after a tracing collection cycle has run, as the dead address

ranges are known at this time. *Cache scrubbing* instructions are used to *clean* or *invalidate* dead cache lines and eliminate any unnecessary writes. The authors show that *cleaning* and marking the line as *least recently used* is most effective for eliminating memory traffic, leading us to use a similar instruction to eliminate writes. Although effective (eliminating 10 to 60% of the write traffic to main memory), this approach relies on the tracing collector to identify dead address ranges. A tracing collection cycle is typically triggered by the allocator when it no longer has space to satisfy the next allocation request. As collection cycles are only triggered periodically, an object may be allocated and die well before it is actually collected. This can lead to large rot times, during which dead objects can be evicted from cache. Tracing collectors are also very memory intensive, often touching large portions of the heap. Any dead data is likely to be evicted from lower levels of cache during the collection cycle, before information about their death can be passed to the cache. The goal of this work is identify dead objects as early as possible, preventing write-backs early in the memory hierarchy. Because of this, we choose to base our approach on *reference counting*.

Reference Counting

The second type of garbage collection algorithm, *reference counting*, maintains a count of pointers to each object in the heap. These pointers may be located in the *root set* or in objects in the heap, an object may even reference itself. When a new reference to an object is created, that object's reference count is incremented. Vice versa, when a reference to an object is overwritten or destroyed that object's reference count is decremented. Figure 2.1 shows 3 objects (A, B, and C) as well as their reference counts. The arrows represent a reference stored at the base of the arrow and referencing the object pointed to. When an object's reference count reaches 0 it is known to be *unreachable* by the program and it is

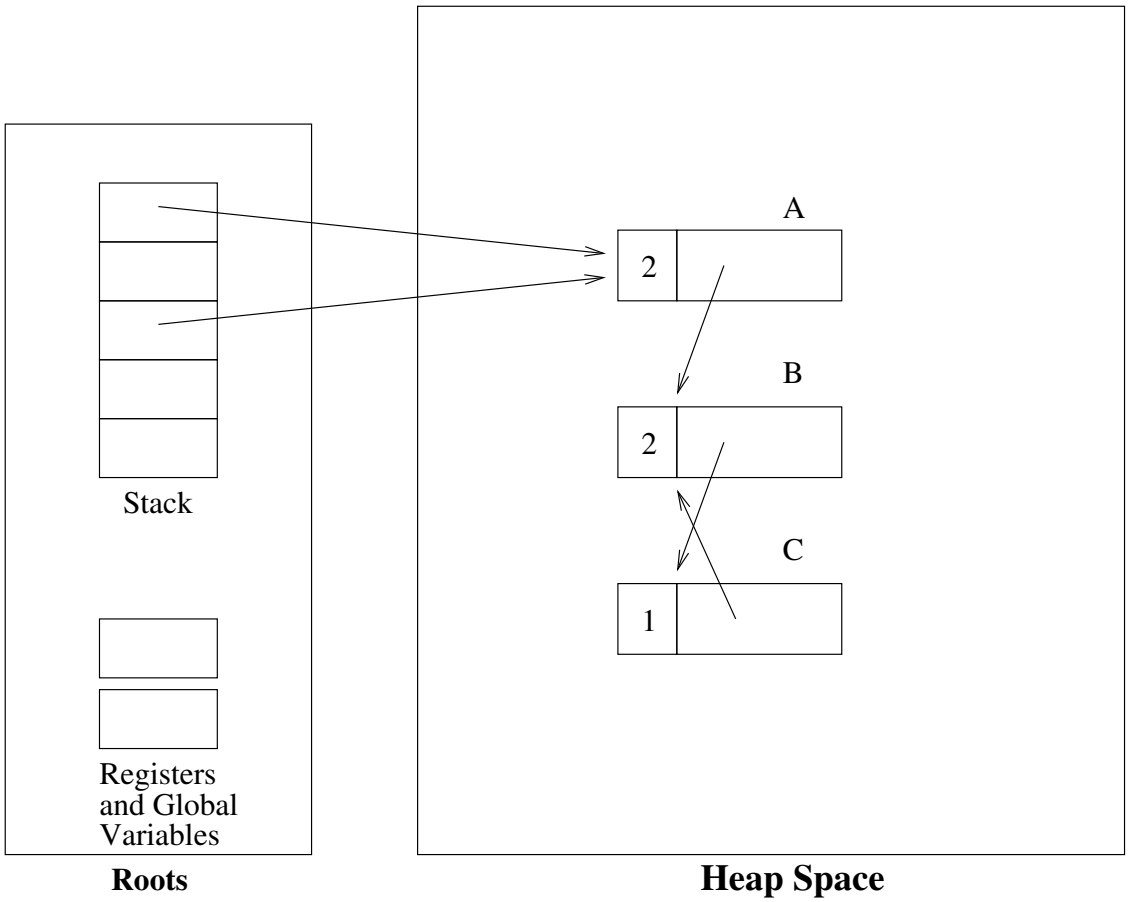

Figure 2.1: A simple reference counting example showing references from the root set into the heap as well as heap references from one object to another.

dead. For instance, if the references to object A stored on the stack were popped off of the stack, $A's$ reference count would be decremented twice (once for each popped reference) down to 0. The object, A in this case, can then be collected and returned to the allocator to be used to satisfy future allocations. Furthermore, any references contained by a dead object can no longer be reached through that object and become invalid. A reference becoming invalid is equivalent to it being destroyed, so the objects referenced must have their counts decremented as well. These decrements are known as *cascading decrements* because the death of one object can lead to decrements and possibly death of many other objects. Looking back

to the example, when A dies it is scanned for references before it is collected. Its reference to B is now invalid leading to a decrement of $B's$ reference count.

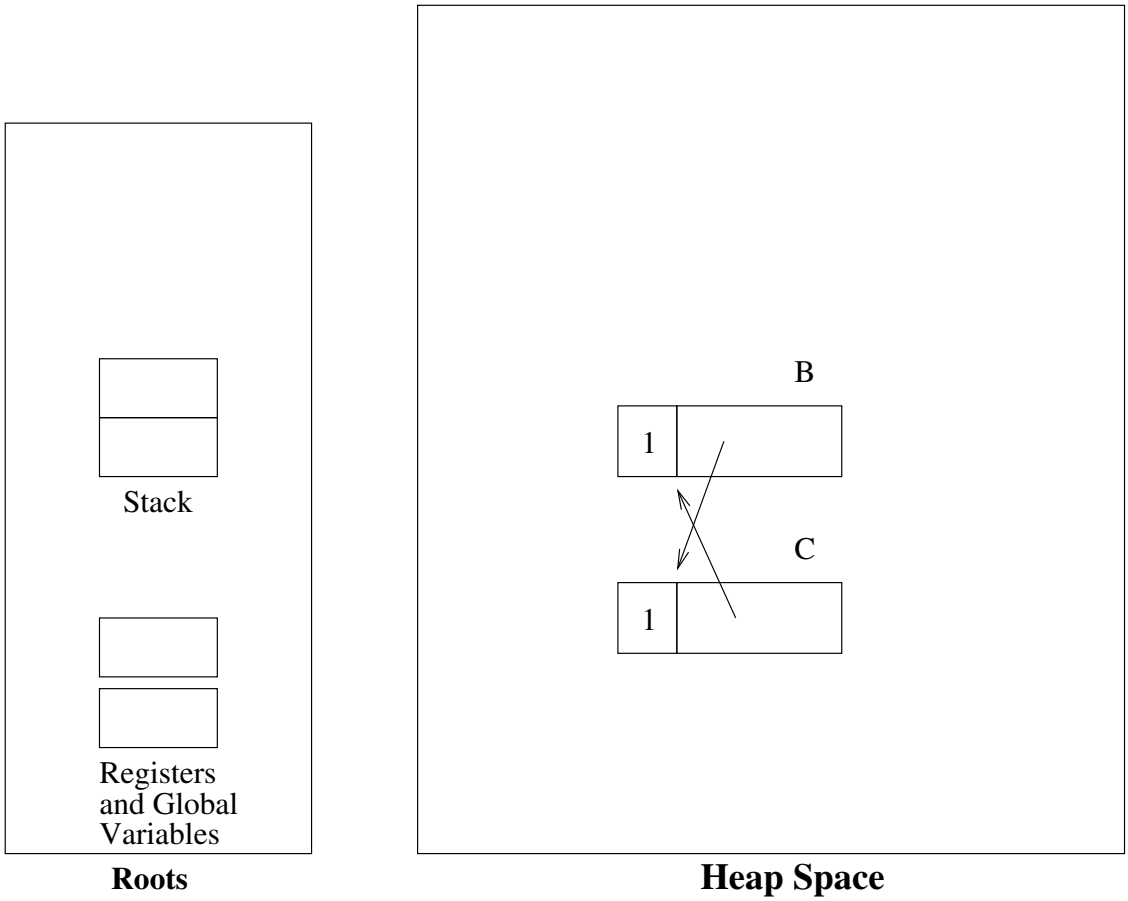

Figure 2.2: The result of Figure 2.1 after the stack references have been popped and A has been collected.

Figure 2.2 shows the result of this and also illustrates one issue that keeps reference counting from being widely used. Reference counting is unable to identify objects as dead that reference each other in a cycle [49]. As we can see, objects B and C are no longer reachable from the program's root set. However, because they reference each other, each of the object's reference count will remain at 1 and they cannot be collected. Because of this, reference counting must be paired with a technique to detect cycles and avoid memory leaks [22, 6].

The second issue with reference counting is memory inefficiency. Each time an object is allocated into the heap, extra space is also allocated in its header to maintain that object's reference count. This increases heap space usage. When a reference to an object is created or destroyed, the objects header must be updated to maintain the correct count. This is typically implemented via a write-barrier that is executed each time a reference is created, destroyed, or updated. A write-barrier for reference counting typically involves several instructions (load the current reference count, update it, and store it back). Therefore, each time a reference is updated, additional memory traffic is created. Although reference counting is not widely used in most systems, [4] implements a full reference counting implementation with cycle detection in Java. Other work has been done to make reference counting more efficient, as discussed below.

[25] notes that most reference count updates come from stack activity. *Deferred reference counting* is proposed, where only heap references contribute to an object's reference count. When an object's reference count reaches 0 it is not yet known to be dead because it may be referenced by the *root set*. Periodically, the *root set* is scanned and any object with a reference count of 0 that is not directly referenced by the *root set* is known to be dead. We note this introduces *rot time* to reference counting.

[58, 68, 64, 20] note that many objects are only referenced once, or are uniquely referenced. They use only 1 reference count bit for each object or pointer, specifying whether or not the reference is unique or not unique. An object that is not uniquely referenced cannot be collected by reference counting and must be collected by a backup garbage collector. However, this reduces the heap space and memory overhead of storing and updating reference counts.

[7] proposes static analysis to allow the compiler to eliminate reference count stores that are redundant or *temporally silent*. In [29], additional write traffic due to reference counting is addressed by avoiding write-backs of reference counts that are also *temporally silent* via *dusty caches*.

Because stack frames exhibit last-in, first-out behavior, the references from the stack can be optimized by collapsing all stack references into only the lowest reference on the stack as shown in [16]. Therefore, any reference count update due to stack activity that is not the bottom reference can be ignored.

Hardware accelerated reference counting as described in [35] creates dedicated hardware to accelerate software reference counting. Hardware structures are created to coalesce individual reference count updates into 1 update by keeping a running total of increments and decrements. The object's header, containing its reference count, is then only needed to be updated once for many actual updates. However, this approach introduces *rot time* into reference counting and maintains the need for additional heap space to store object's reference counts. Reference counting's ability to identify dead objects without *rot time* is needed to eliminate unnecessary writes of dead data as early as possible.

Despite reference counting's inefficiencies, an object is known dead with no *rot time*. This is important for eliminating unnecessary writes. Our approach, is therefore based on reference counting. We use a novel form of reference counting to identify dead data and eliminate unnecessary writes out of the cache. Our technique, *Cache Only Reference Counting* (CORC from here out), implements hardware reference counting limited to the cache only. Objects are reference counted from allocation until eviction from cache. This avoids the memory inefficiencies of traditional reference counting and maintains no *rot time* for the objects it can determine to be dead.

2.1.4 Other Related Work

Generational Garbage Collection

In a sense, CORC forms a subset of all heap objects similar to generational garbage collectors. [47] and [65] first proposed generational garbage collection as not all objects have similar lifetimes. Objects are divided into generations based on the objects' lifetime. Typically the heap is divided into a young generation and an old generation. Objects are allocated into the young generation. When the allocator runs out of space, the young generation is garbage collected. As most objects tend to die young [65], collection of the old generation is often not needed. Objects are promoted from the young generation to the old after surviving a set number of collection cycles. The old generation is only collected when sufficient space is not freed up by a collection of the young generation. Therefore, the young generation is collected much more often than the old generation and much work has gone into making young generation collection more efficient [28, 2].

Typically generational garbage collectors implement *tracing* collectors in both the young and old generations; however, work has been done to explore the use of hybrid collectors. So far, work in this area explores using a tracing collector in the young generation and reference counting in the old generation [3, 12]. This is successful because objects in the old generation are less likely to have their reference count updated frequently. The increased memory traffic due to reference count updates is therefore much less.

The subset created by CORC contains all objects that have not been evicted from cache since allocation. Therefore, most objects reference counted are very young. This is in contrast to the hybrid collectors described above. However, in this chapter we evaluate the effectiveness

of CORC at eliminating writes only. CORC's ability to recycle dead objects is studied in Chapter 3.

In Cache Reference Counting

In cache reference counting was first introduced in [55] for the Lisp programming language. All cells are the same size, and no implementation details were given. For this work, we investigate write reduction via in cache reference counting for Java, where objects may vary in size and alignment.

The most closely related work to CORC, [17] introduced a separate object cache and coprocessor. The coprocessor was evaluated in [18]. Objects are allocated into the object cache and reference counted while resident in the object cache. Objects are required to be aligned with the beginning of a cache line and only one object can be allocated per line, creating fragmentation in the heap. In contrast, CORC allows objects to be allocated as normal, into the standard memory hierarchy, with no alignment restrictions. Objects can be any size and are allowed to share cache lines, getting rid of any unnecessary fragmentation. [18] produced similar results when eliminating writes out of a level 2 cache as CORC, as shown in Section 2.3.

We now describe CORC in detail.

2.2 Cache Only Reference Counting

In many programming languages, including Java, when heap space is allocated for an object it is immediately *zeroed*. *Zeroing* writes $0's$ to the newly allocated space, bringing it into cache upon allocation if it was not already present in cache. Consider, object A is allocated and *zeroed*. CORC maintains $A's$ reference count while it remains in cache prior to its eviction. Outside of the cache itself, reference counts do not exist (for our purposes). This is done by augmenting each cache line with additional hardware to maintain meta data for the objects the line contains. For each object spanning a line, the line maintains:

- an Object ID (the object's virtual address)
- a Bit vector representing the bytes of the line occupied by the object
- a Frame ID associated with the lowest stack frame containing a reference to the object
- a Thread ID holding the thread the object is allocated by
- a Reference count for the object

Each line also maintains information about itself independent of the objects it contains. This includes:

- a bit vector representing which words of the line contain references
- a bit vector representing which words of the line are known to be dead

ISA modifications are required to pass reference counting instructions to the cache. The additional instructions needed to notify hardware of reference count updates are shown in Figure 2.3.

allocate(A,n)**:** a new object of n bytes is allocated at address A.

refstore(p,q)**:** a reference field at address p is set to the value (object address) q

refload(q)**:** a reference to the object at address q is loaded onto the runtime stack.

framepush: a new frame is pushed onto the runtime stack, in response to a method call.

framepop: the topmost frame on the runtime stack is popped, in response to a method's `return`.

returnref(q)**:** the currently executing method is terminating, returning a reference to (object address) q.

gcstart: a garbage collection cycle is starting

gcend: a garbage collection cycle is ending

Figure 2.3: Directives that interface between the running application and the cache.

By design, the Java Virtual Machine(JVM) has bytecodes that conveniently map to the instructions in Figure 2.3. For example, the *putfield* bytecode pops an address and some data off the top of the stack and stores the data to the address. If the data being stored is a reference, then this bytecode corresponds to a `refstore` instruction. Conveniently, the JVM is aware of the type of data being stored as Java is strongly typed. Each of the other reference counting instructions can be mapped to operations in the JVM in similar ways. As the JVM executes an operation that corresponds to a reference count instruction in Figure 2.3, the reference count instruction is passed to hardware as well. For languages not as strongly typed as Java, effort may need to be made by the programmer to identify when reference count instructions should be injected into the code.

As an overview of our approach, we begin by describing the most favorable scenario.

- An object T is allocated, and all of its bytes are contained in cache, perhaps spread across multiple cache lines as depicted in Figure 2.4. Throughout the rest of this

description, we assume (ideally) that none of the bytes of T suffers an eviction. The reference count held by the cache for T is initialized to 0 and is incremented to 1 as the reference to the newly allocated T is stored onto the runtime stack.

- Subsequent to T's allocation, the running program changes a field of object U, previously null, so that it references T by a `refstore` instruction. This action increments the reference count in the cache for T to 2.
- Subsequently, the field of U that references T may or may not be evicted from cache.
- In either case, a subsequent `refstore` to that field (say, to null) will cause the reference count in cache for T to drop to 1.
- Now the only reference to T is from the runtime stack. A subsequent pop of the stack frame referencing T will decrement the reference count for T to 0, which causes the cache to regard the bytes associated with T as dead.
- The dirty bits associated with T's bytes are cleared.
- Finally, the reference fields contained within T are visited, and the reference counts of any objects they reference are decremented.
- This in turn can trigger similar actions taken on other dead objects in cache. Although these decrements cannot be done concurrently, they can be performed off the critical path(in subsequent cycles) to avoid a significant delay in program execution.

As for T, the running program could not possibly access its bytes after it has been determined dead. When the bytes associated with T are evicted, they will not be written back from cache because their dirty bits have been cleared.

Crucial to the success of the above description is that T dwelled in cache long enough for it to be determined dead. For those lines of T that are evicted prior to determining T dead, our approach could not squash the associated dirty bytes' write backs from cache. Our initial optimism about the success of our approach stemmed from the widely accepted *weak generational hypothesis*, which has been verified for Java [36].

The experimental results we report in Section 2.3 confirm the viability of our approach. In the remainder of this section, we describe our implementation in greater detail and point out its inherent and addressable shortcomings.

Our experimental setup described in Section 2.3 includes a custom cache simulator, in which we implemented the cache protocol described here. Although the protocol is realized in software for this chapter, we developed it with a hardware realization in mind. The relevant details are described in this section.

The cache responds to the directives issued by the running application that are shown in Figure 2.3. For each such directive, we describe below the actions taken by the cache and how those actions can be realized as architectural support in hardware. We organize our discussion according to the heap and stack activity of a program running in a JVM. Although it may appear that registers have been ignored, the JVM implements registers as stack cells. Thus, our treatment of stack activity also covers register loads and stores in the JVM.

2.2.1 Heap Activity

The `allocate` instruction specifies that n bytes of storage have been allocated starting at address A. For Java, this directive is due to a `new`, `newarray`, or `clone` program

operation. Similar gestures in other languages are easily accommodated by our approach. For Java, all bytes of the specified storage are initialized to 0. This initialization is explicit in Java, and so the values must behave as if written to storage, though they may reside only in cache just after allocation. The cache responds to this directive as follows. Given the starting address A and extent of the allocation n, each line can determine which of its bytes, if any, are contained in this allocation. The line then records a mapping between the object (which can be represented by the starting address A) and that range of bytes. This action can be performed concurrently for each line in the cache.

Each cache line may host storage from different `allocate` instructions. For the purposes of our study, we placed no limit on the number of storage blocks a given cache line might host. However, no cache line can host more than $\lceil \frac{a}{b} \rceil + 1$ blocks, where a is the number of bytes in a cache line and b is the smallest block (least number of bytes) that can be allocated. For each cache line, our simulator maintains a list of mappings between objects and the range of bytes associated with the objects in that cache line. Hardware could place a limit on how many objects are recorded, and devote circuitry to recording the mapping of those objects. For those objects beyond the capacity of that hardware, their bytes would be written back from cache even if those objects are dead. In any case, the goal of this operation is to remember which portions of a cache line are associated with the blocks of allocated storage hosted by the cache line.

An illustration of a series of `allocate` instructions is shown in Figure 2.4. Suppose an `allocate` occurs for object `T` that occupies the portions of lines $0 - 2$ as shown. Subsequently an `allocate` occurs for `U` that occupies the portions of lines $2 - 4$ as shown. Mappings are established for the cache lines as follows:

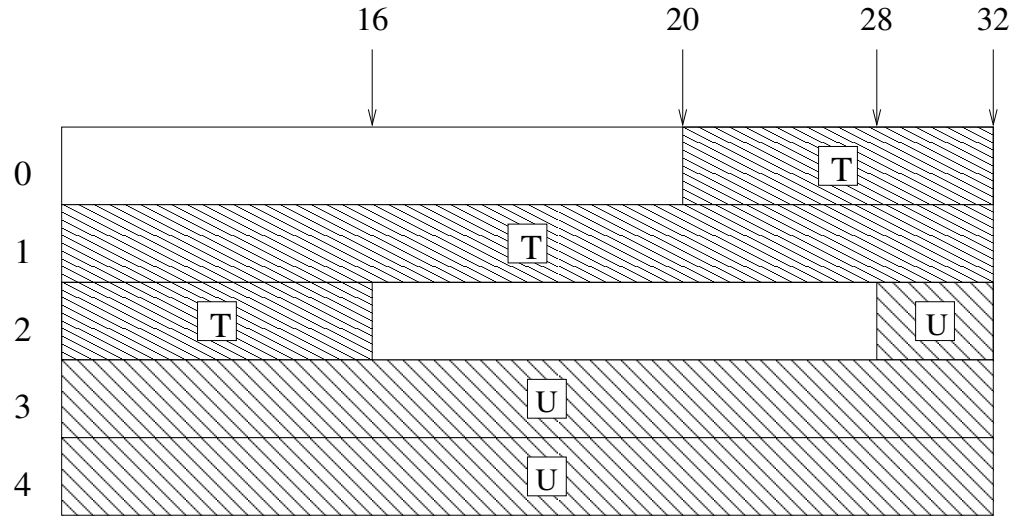

Figure 2.4: Illustration of `allocate` instructions. The cache shown here has 5 lines, each with 32 bytes.

Line	Hosted objects
0	T→{ 20 . . . 31 }
1	T→{ 0 . . . 31 }
2	T→{ 0 . . . 15 }, U→{ 28 . . . 31 }
3	U→{ 0 . . . 31 }
4	U→{ 0 . . . 31 }

After an allocation instruction, subsequent program activity may cause some (or perhaps all) of the lines associated with the allocation to be evicted from cache. For those evicted lines, our ability to determine dead but dirty bytes is lost. Such is the price paid for limiting the scope of reference counting to the cache itself. However, *any* cache lines that remain unevicted can still be tracked by our approach. In the example above, suppose that line 2 is evicted. The affected portions of `T` and `U` are no longer eligible for write back squashing by our approach. However, the other portions of `T` and `U` remain candidates for finding dead

but dirty storage. Thus it is possible that we squash some, but not all, of an object's dirty write backs from cache. We study the effectiveness of our approach in Section 2.3.

A `refstore` instruction specifies that a reference field located at address p is modified to point to address q. The cache must act at this point to account for the affected reference counts of storage blocks that are still contained in cache. This entails decrementing the reference count of the object that p referenced (say, r) prior to this `refcount` instruction, and incrementing the reference count of the object q that is referenced after the instruction. This is achieved as follows.

This `refstore` reflects a modification to storage, in particular to the field at address p. As such, that field must be in cache, which means that its value r before the instruction is in cache as well. The cache line that contains p can announce to all cache lines that the reference count of object r should be decremented, provided that r is not null. Each cache line can consult its mapping to determine if it holds any portion of object r, and, if so, can decrement the reference count for r. Our cache simulator works in that fashion. Alternatively, the cache can maintain a global (among all cache lines) reference count table, and decrement r's reference count in response to the aforementioned announcement.

The cache line that contains the field at address p will also see the newly stored value q, and a similar announcement can be made that the reference count(s) associated with q should be incremented, provided that q is not null.

The cache line containing address p also sets a bit corresponding to p in a bit vector, allowing *cascading references* to be handled quickly upon object death.

2.2.2 Stack Activity

Reference counts in cache must also account for references that are sourced from outside the heap. Because stack frames exhibit last-in, first-out behavior, the references from the stack can be optimized as shown in [16].

To track the stack activity, the `refload` instruction informs the cache that a reference has been loaded onto the stack, pointing to q. The cache must determine whether this is the first reference from the stack, and if so, remember the frame associated with the stack's references to q, summarized by the last-to-be-popped frame. In support of determining the proper frame, the cache is continually advised about stack activity via the `framepush` and `framepop` instructions. The stack-summarizing optimization used [16] works only for a single thread. Therefore, if a reference to an object is loaded via a `refload` instruction onto a thread's stack that the object was not allocated by, we cause the object's reference count to stick permanently at its highest value. Thus, we cannot currently squash the writes of objects referenced in this manner.

The `returnref` instruction notifies the cache that a reference is being returned to the previous stack frame. Suppose a reference to q is being returned, the cache lines containing q must decide if the stack frame q is being returned to is lower than the frame q is currently associated with. If so, $q's$ associated stack frame is decremented.

Finally, when a stack frame is popped the cache is notified via a `framepop` instruction. Each line determines if any objects it contains are associated with the frame being popped. If so, the object is no longer referenced by the stack and its reference count is decremented.

2.2.3 Object Death

When a cache line determines an object it contains has died, via a reference count decrement to 0, the line marks each word containing parts of the dead object as dead. If all bytes of the line are dead, the dirty bit is cleared and the line is set to be least recently used. This ensures the fully dead cache line will be the next evicted line in its set and that it will not be needlessly written back. This is shown to be more effective at eliminating writes than simply *cleaning* the cache line in [59].

In order to handle *cascading references* the line also consults the bit vector maintaining where references are stored in it (as set by the `refstore` instruction). If any reference location is inside the dead object, the cache reads the reference and announces the object to be decremented to all cache lines.

2.2.4 Shortcomings

In this section we summarize some limitations of our approach, as discussed above. We show in Section 2.3 that we nonetheless find significant savings of write backs from cache.

As described our technique has a few shortcomings.

- There is no way to track references outside of the cache.
- Objects in cycles remain unidentifiable as dead
- The technique cannot track references for objects shared by multiple threads or cores
- Additional hardware and ISA additions are needed

The first two shortcomings are acceptable because the goal of this technique is not garbage collection, therefore completeness is not a concern. Any objects that we do not identify as dead will eventually be identified as dead by the garbage collector. However, *Contaminated Garbage Collection* [16] is a conservative garbage collection algorithm that can collect these objects in cycles. We address the use of *contaminated garbage collection* along with reference counting as future work in Section 5.1. Any object that is shared across threads is marked as forever alive by our approach. However, preliminary data suggest the majority of short lived objects this approach finds dead actually die before they are shared so being conservative in our approach is reasonable. Hardware costs are evaluated further in Chapters 3 and 4.

2.3 Evaluation

2.3.1 Experimental Setup

There are two phases to the experiments we conducted:

1. For each benchmark we tested, we gathered a trace of data loads/stores into the heap as well as the cache directives described in Figure 2.3 issued by that benchmark.

2. We ran each trace through a cache simulator that includes the in-cache reference-counting technique.

To generate the traces, we instrumented the JVM (Java Virtual Machine) in `OpenJDK` (version 1.8.0). The JVM was instrumented to generate the instructions detailed in Figure 2.3, along with loads and stores of non-pointers and garbage collection cycles. Although the

above does include all activity generated *directly* by the application, it is important to note that some memory traffic is not included, namely the activity of the JVM itself. The JVM makes allocations outside of the garbage collected heap that are managed explicitly with (C++) new and delete operators, which we do not trace. Thus our results are accurate as if the Java code were compiled and its instructions were executed without interpretation. Moreover, the previous work we mention on eliminating dead writes in explicitly managed languages [14, 34] could address this traffic and could be combined with our work without interference. We further address this in Section 5.1. Our results would also hold for data caches that can be *split* according to application activity [52, 53].

We wrote a trace-based cache simulator to implement our approach. This simulator is highly componentized and greatly simplified our experimentation. The simulator processes the instructions shown in Figure 2.3 with the following exception. We assume that an application-level garbage-collection cycle will evict most, if not all, lines in a cache. As such, the collection cycle effectively flushes the cache. Moreover, the collection cycle could change the location of objects. For those reasons, it would be unfair for us to assume we could continue our approach through such a cycle. We therefore simulate a cache flush at the onset of JVM garbage collection and do not resume our approach until the cycle is complete. In similar fashion, context switches may be handled by invalidating all reference counting meta data stored in the cache lines and starting fresh. Any write backs that may occur during a garbage-collection cycle are not counted in our statistics. We believe the impact of these write backs would be minimal as less than 0.3% of lines in each trace file are created during garbage collection cycles.

Our tests were conducted using several of the `DaCapo-9.12-bach` benchmarks [13]. The benchmarks we used are summarized as follows, with more detail found at the website [13].

`avora`	simulates a number of programs run on a grid of AVR microcontrollers
`fop`	takes an XSL-FO file, parses it and formats it, generating a PDF file
`lusearch`	Uses lucene to do a search of keywords over a corpus of data comprising the words of Shakespeare and the King James Bible
`pmd`	analyzes a set of Java classes for a range of source code problems
`sunflow`	renders a set of images using ray tracing
`xalan`	transforms XML documents into HTML

For each benchmark, the first 50 million lines of tracing were captured. We observed that this prefix of a complete trace was sufficient to allow the JVM its initialization and to allow the benchmark to exhibit its standard (steady-state) execution behavior. The traces were created using an initial application heap size of 64MB.

While our results call for further experimentation on a wider variety of cache configurations, we limited our experiments for the purposes of this paper as follows.

2.3.2 Level 1 Cache

For our first experiments, we used the following configuration parameters:

- 2-way associativity, a single dirty bit per line
- 32 bytes in each cache line
- Object reference counts limited to two bits
- Write backs performed at the line level
- LRU replacement policy

With only two bits to represent a reference count, the maximum value of a reference count is 3, and at that point the reference count is *sticky* [5] and cannot be decremented. Experiments justifying this choice are left out for space considerations. We justify this choice in Section 2.3.4, where we study how the size of a reference count influences the effectiveness of squashing write backs.

We begin by limiting write backs to occur at the line level, which most closely resembles the granularity at which caches already manage write backs. This means that We track the death of data in a cache line at the level of the line's dirty bit, namely across the entire line. Thus, in this implementation, an entire line is either dead or not. As a result, the write backs from a line are either squashed across the entire line, or, if the line is dirty, the entire line is written back. We study this implementation because of its reduced cost for realization in hardware. We relax this limitation and study a more precise characterization in Section 2.3.3.

Although the above configuration details generally place us somewhat at a disadvantage, they seemed realistic in terms of minimal cost of architectural support for our approach.

For our experiments, the primary metric of interest is the fraction of squashed write backs. By this, we mean the fraction of writes that would have reached memory without our technique in place. In other words, if the program would normally have issued N writes, but with our technique in place, only K writes are issued, then we have squashed $\frac{N-K}{N}$ writes. In this way, the results we report are scaled in the interval $(0, 1)$, where a 1 would correspond to all writes being squashed.

We study the fraction of squashed write backs as a function of cache size. A larger cache usually results in fewer conflict misses and thus fewer evictions, allowing us more time to find objects' lines dead prior to their eviction. Figure 2.5 shows the fraction of squashed write backs found in caches ranging in size from 8KB to 128KB.

These results show that for a reasonably sized 32KB level 1 cache we can squash an average fraction of 0.304 writes from the level 1 cache to other levels of memory.

The start of each of our traces includes the JVM and Dacapo Benchmark Suite initialization. While our results for those portions of the traces are quite good, we also examined our approach as measured on the steady-state portion of the traces. Figure 2.6 shows the fraction of squashed write backs found in each benchmark's steady state of execution by skipping the first 10 million lines of each trace. At this point in the trace, each benchmark was beginning its standard execution behavior. Figure 2.6 uses the same configuration as the previous experiment. In the steady state, we continue to squash 22% of all write backs given a 32KB cache.

Figure 2.7 shows the data from both Figure 2.5 and Figure 2.6 for ease of comparison.

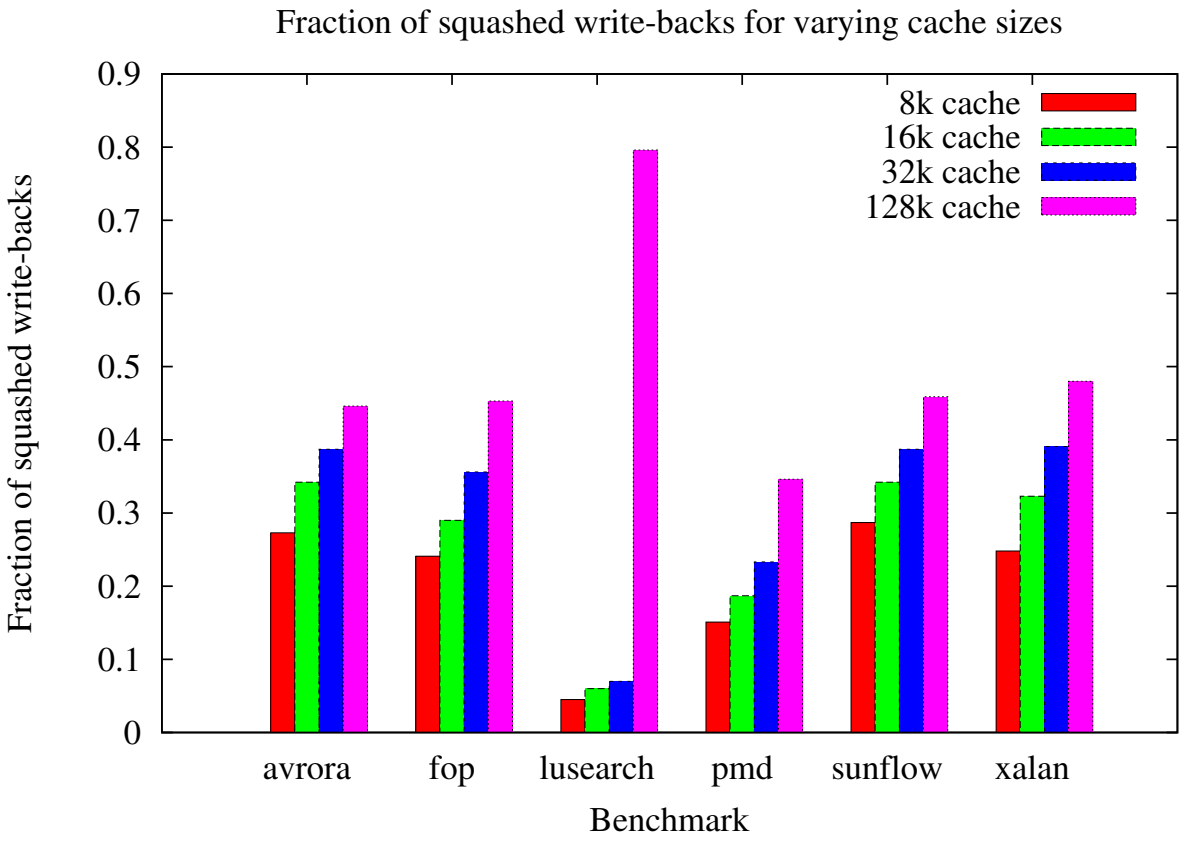

Figure 2.5: Fraction of squashed write backs found for varying cache sizes, each having 32-byte lines.

The experiments described below in Section 2.3.3 and Section 2.3.4 were conducted with the following configuration:

- 2-way associativity
- A single dirty bit per line
- 32 bytes in each cache line
- 32 KB total storage in cache

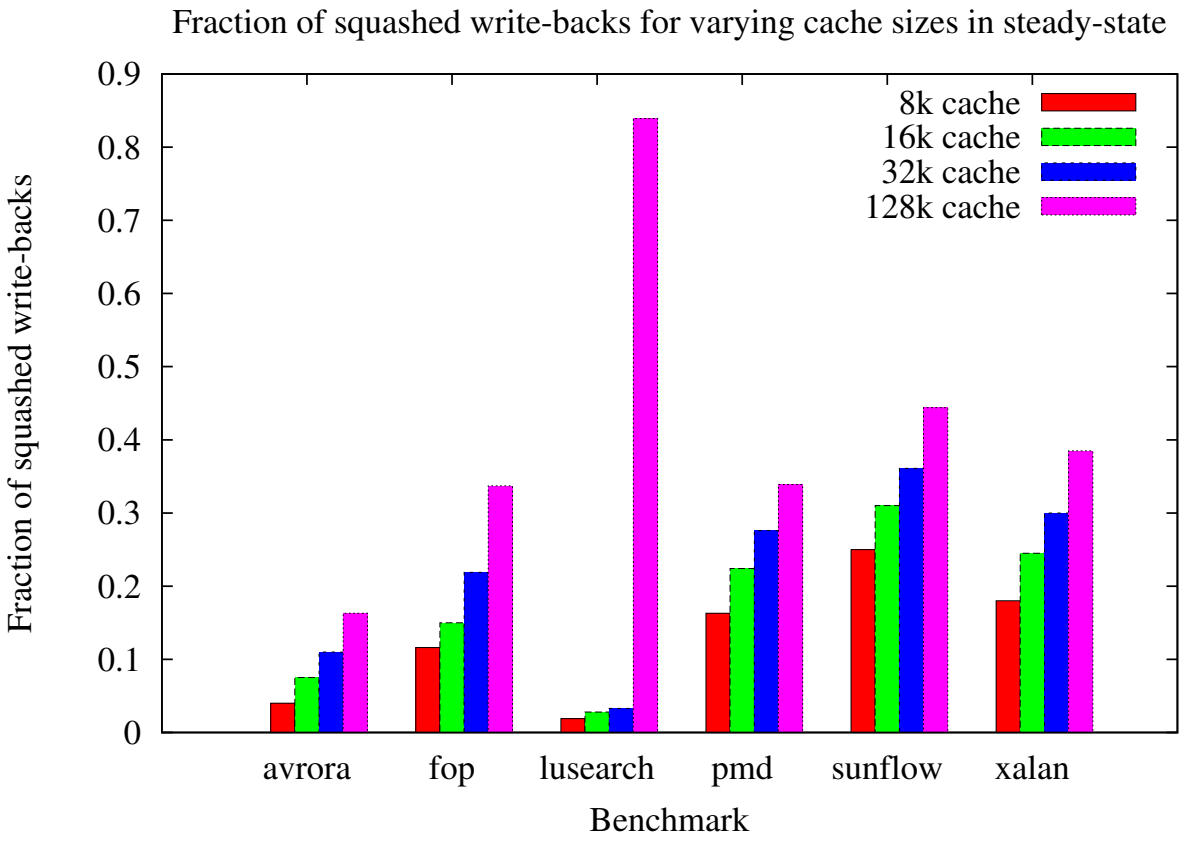

Figure 2.6: Fraction of squashed write backs found for varying cache sizes in the applications' steady state.

- LRU replacement policy

The unlisted parameters are the subject of experimentation below.

2.3.3 Granularity of Squashing

Above, we squashed the write backs at the line level, which leads to a relatively simpler hardware implementation. Here, we collected statistics as if write backs could be squashed

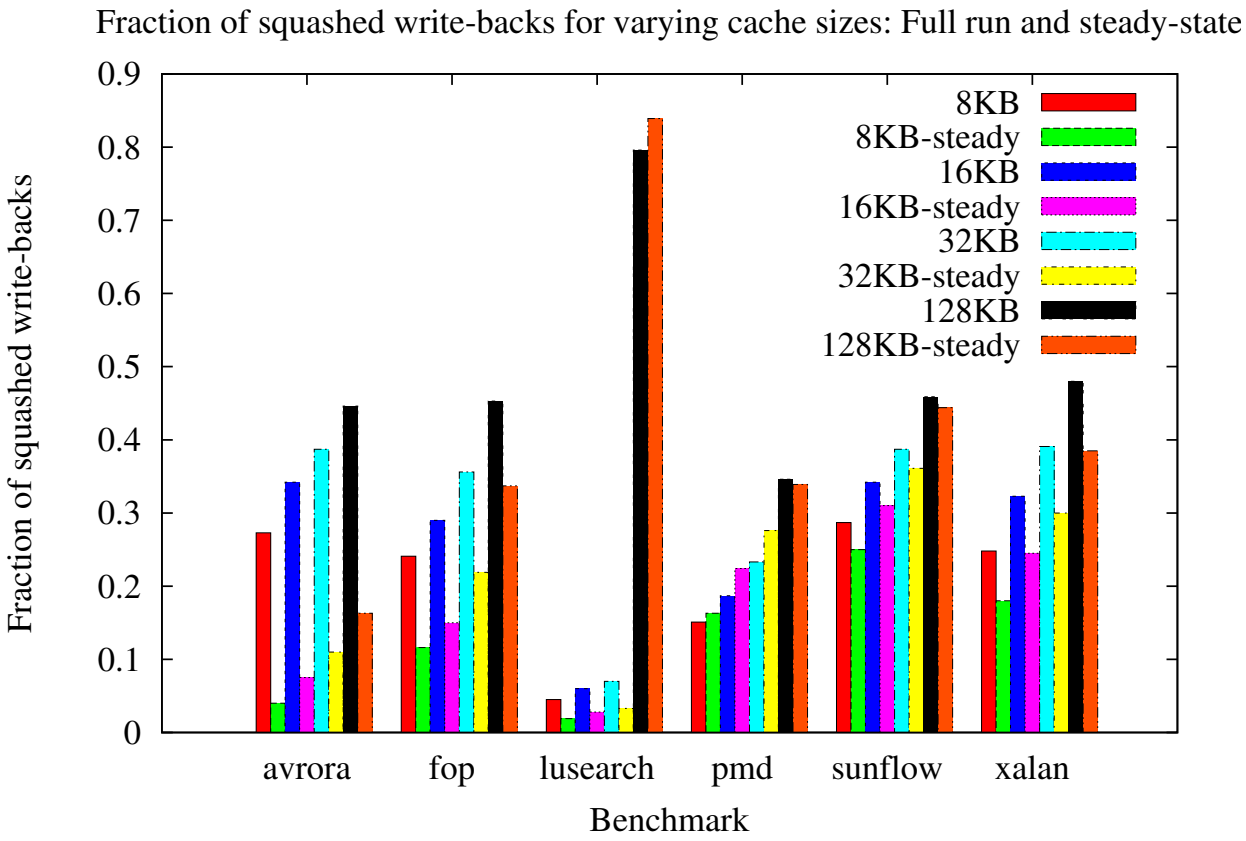

Figure 2.7: Total (steady-state + initialization) and steady-state fraction of squashed write backs for varying cache sizes.

at the byte-level instead of the full line-level. This would require support for detecting which bytes within a line need to be written back, and writing only those bytes from cache. Memory systems are not typically designed to accomplish this level of write back, but we were interested in how many more writes could be saved with such an implementation.

For example, suppose that 8 bytes of a 64-byte line are dead. At the byte-level, we can squash the 8 bytes and write back only the 56 live bytes. In the previous experiments the entire line would need to be written back.

Figure 2.8 shows the fraction of squashed write backs found, if counted at the byte level as compared with counting at the line level. Each reference count is still represented by only two bits in this experiment.

As expected, the fraction of squashed write backs is larger when write backs can be performed at the byte level, due to the fact we can now squash writes from a dead object that are contained in a cache line that is not completely dead. The relatively small increase in savings can be explained by prior observations that objects allocated close to each other often die together (because most objects die young). Therefore it is likely that objects sharing a cache line will die around the same time, leading to the entire line being dead.

2.3.4 Size of the Reference Count

Our experiments above used only two bits to accommodate an object's reference count in cache. Here we study whether two bits suffice to track reference counts accurately. However many bits are dedicated to a reference count, if an object reached its maximum reference count, then a *sticky count* was used. This means that the count cannot be changed in the future and the object can never be found dead.

Here we study a 2-way associative 32KB cache with a line size of 32 bytes, with write backs performed at the line level. Figure 2.9 shows the results of varying the number of reference-counting bits from 2 to 4. These results show that there is little value to using more than 2 bits, so we chose 2 bits for all other experiments.

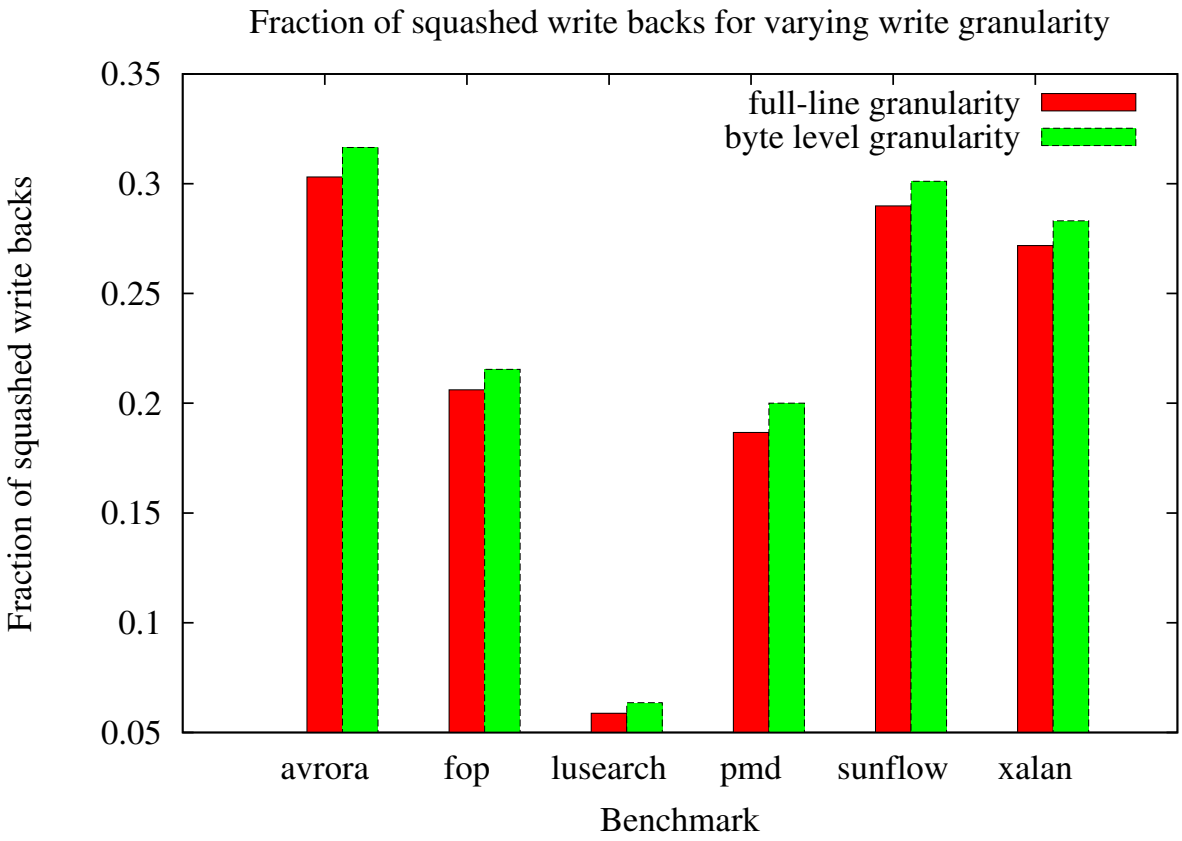

Figure 2.8: The fraction of squashed write backs as a function of granularity. Here, a 32KB cache with 32 byte lines was used.

2.3.5 Incremental Performance

Our results above provide summary statistics concerning how our approach works on average across a trace. We also studied how our approach worked in increments of 100,000 lines in each experiment. Each line corresponds to an instruction of the form shown in Figure 2.3. Our experiments in this section used a 2-way associative 32KB cache, with 32 byte lines, and write backs at the line level.

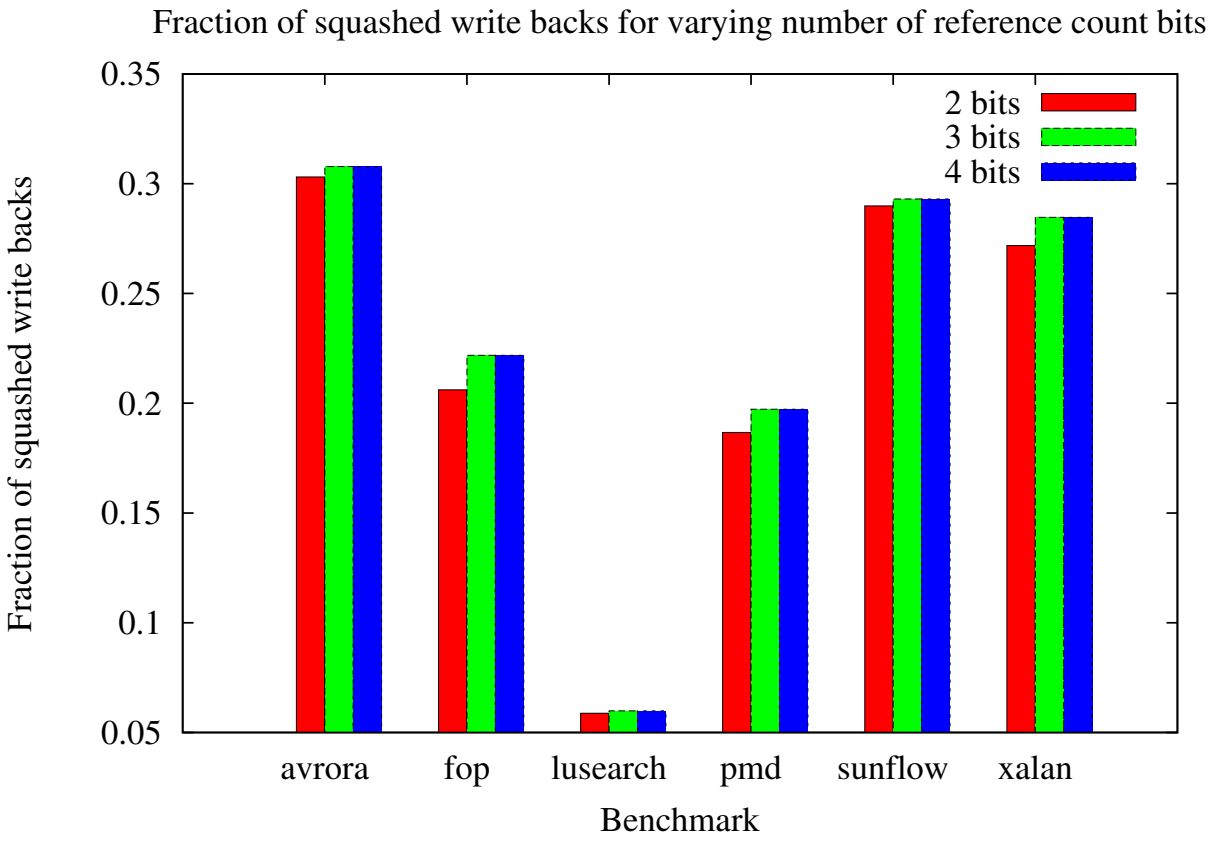

Figure 2.9: The fraction of write backs squashed as a function of the reference-count maximum value.

Figure 2.10 charts the fraction of squashed write backs versus the line number of the trace for the `sunflow` benchmark. Plotted are cumulative statistics as well as incremental statistics for the previous 100,000 lines.

All of our benchmarks exhibit similar behavior for the first ∼5 million lines. This is due to the initialization of the Dacapo Benchmark Suite before it begins running the application. After 5 million lines, we see periods of varying behavior as the benchmarks execute, some which have a high fraction of squashed write backs and others with a low fraction. Our experiments show increments sporting as high as 90% squashed write backs.

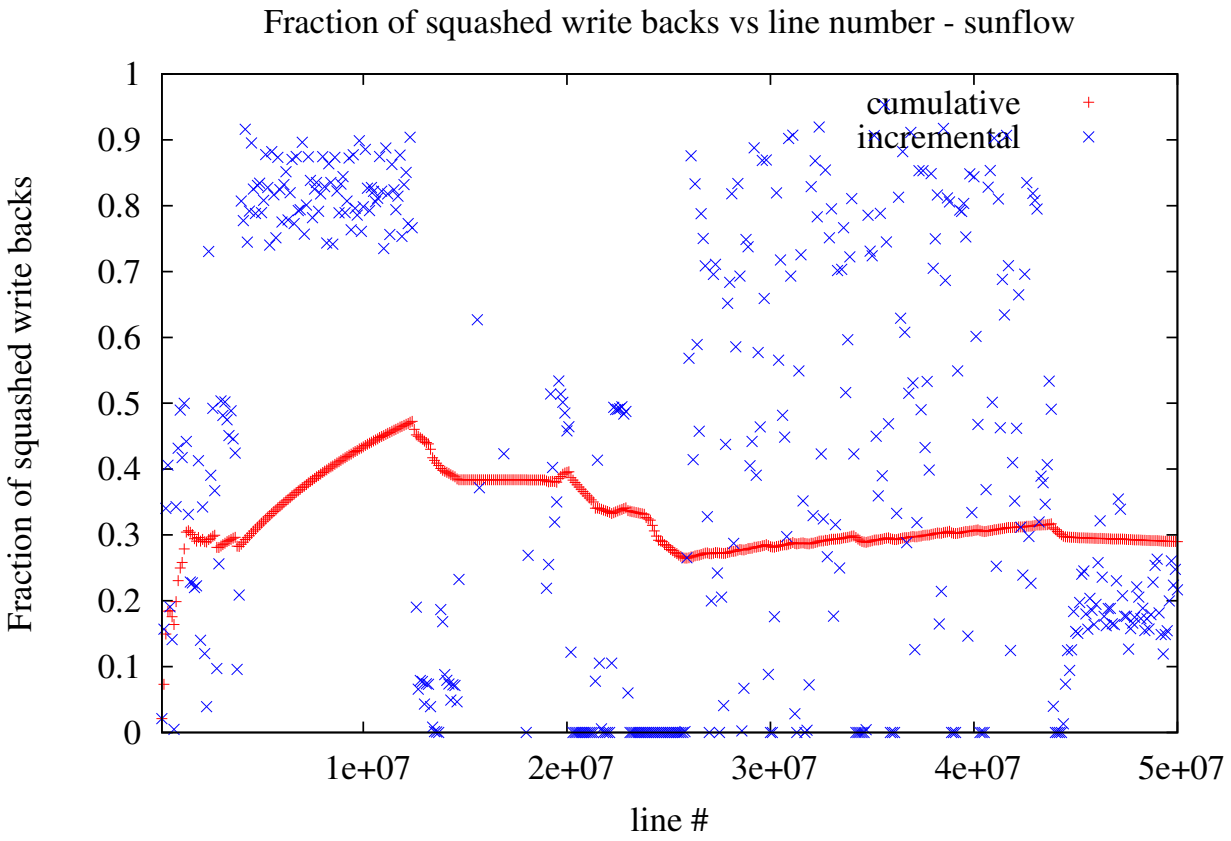

Figure 2.10: Fraction of squashed write backs vs the line number of the trace for the `sunflow` benchmark.

During periods where our approach is not as effective, it is possible the program is working on existing objects that are not currently dead, which leads to evictions of live, dirty objects. This behavior is demonstrated in the `lusearch` benchmark shown in Figures 2.11 and 2.12. These figures show the incremental and cumulative fraction of squashed write backs for a 32KB cache and a 128KB cache, respectively (all other cache parameters are as noted previously. For the larger cache, a much larger fraction of writes are squashed. Figure 2.12 shows the `lusearch` benchmark has approximately 70% of its write backs squashed by our approach. The results are much worse when the cache has only 32KB.

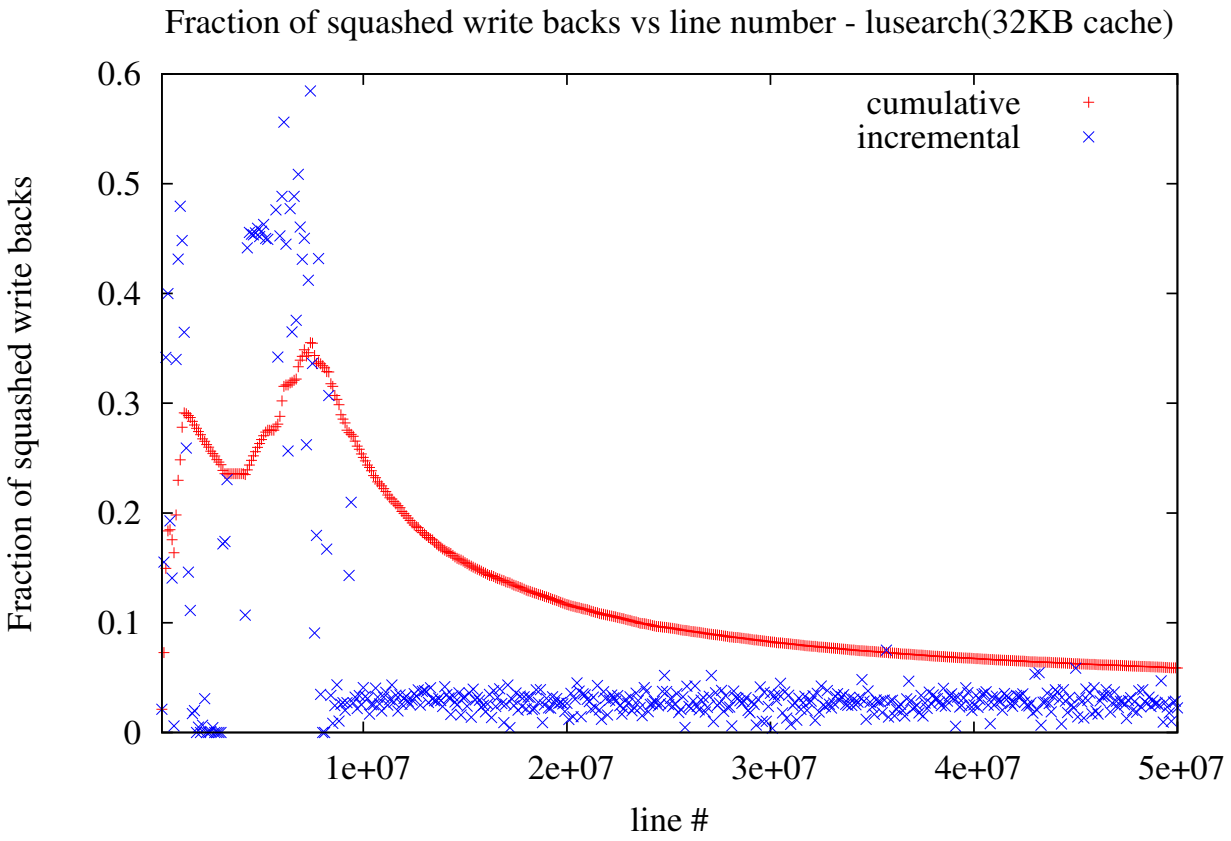

Figure 2.11: Incremental and cumulative fraction of squashed write backs for `lusearch` with 32KB cache.

At least for this one benchmark, we can tell that our approach suffers not because of inherent imprecision (such as objects in cycles), but instead we observe that lines are evicted before they are found to be dead.

2.3.6 Level-2 Cache Approximation

A 128KB or larger level-1 cache may be unrealistic, but such a cache might well be deployed as a level-2 cache. To get an idea of how well our technique may perform when expanded to

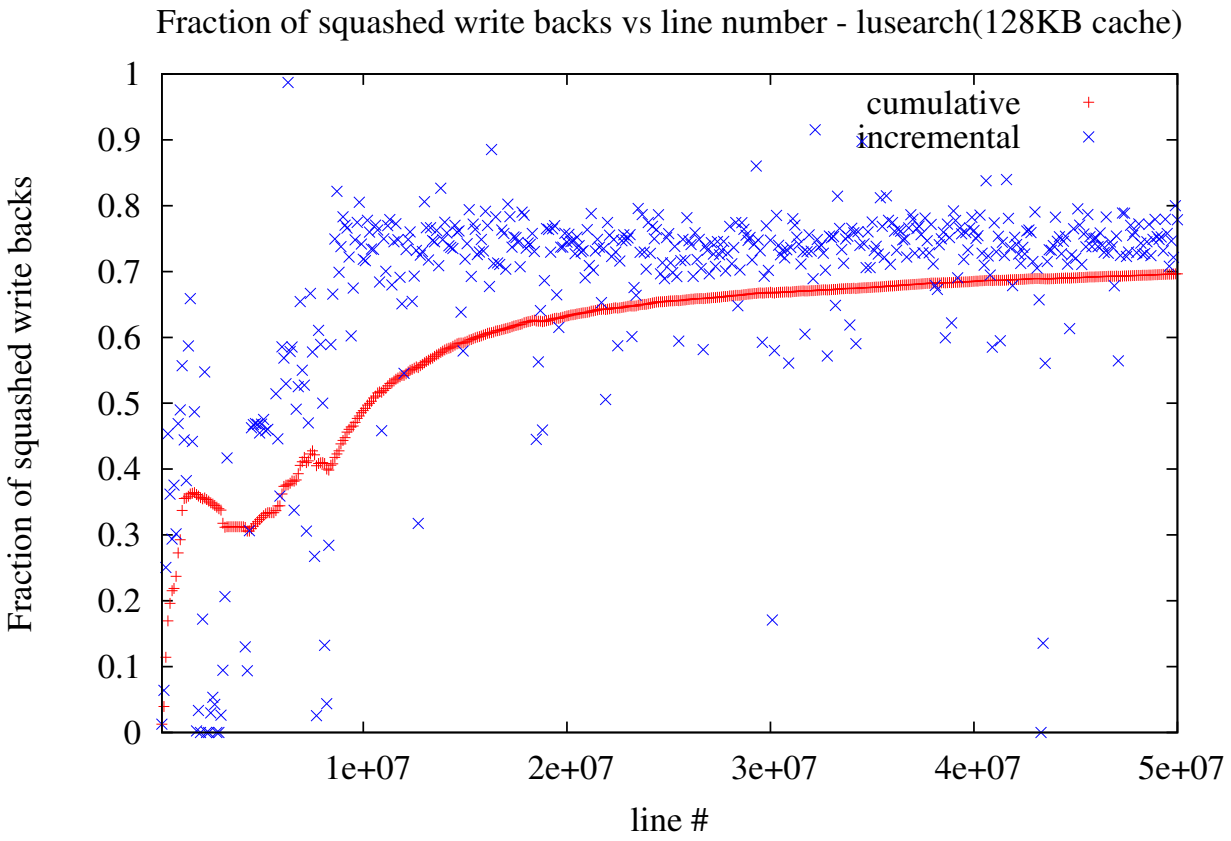

Figure 2.12: Incremental and cumulative fraction of squashed write backs for `lusearch` with 128KB cache.

level-2 cache we ran experiments with a large (512KB) level-1 cache. The other parameters of the experiment are as listed:

- 4-way associativity, a single dirty bit per line
- 64 bytes in each cache line
- Writes performed at line-level
- Object's reference counts limited to 3 bits

- LRU replacement policy

As we are interested only in seeing how many write backs can be squashed with larger caches; we do not set up a hierarchy of caches, but instead increase the size of the level-1 cache. The parameters chosen are reasonable for modern level-2 caches. Results are shown in Figure 2.13.

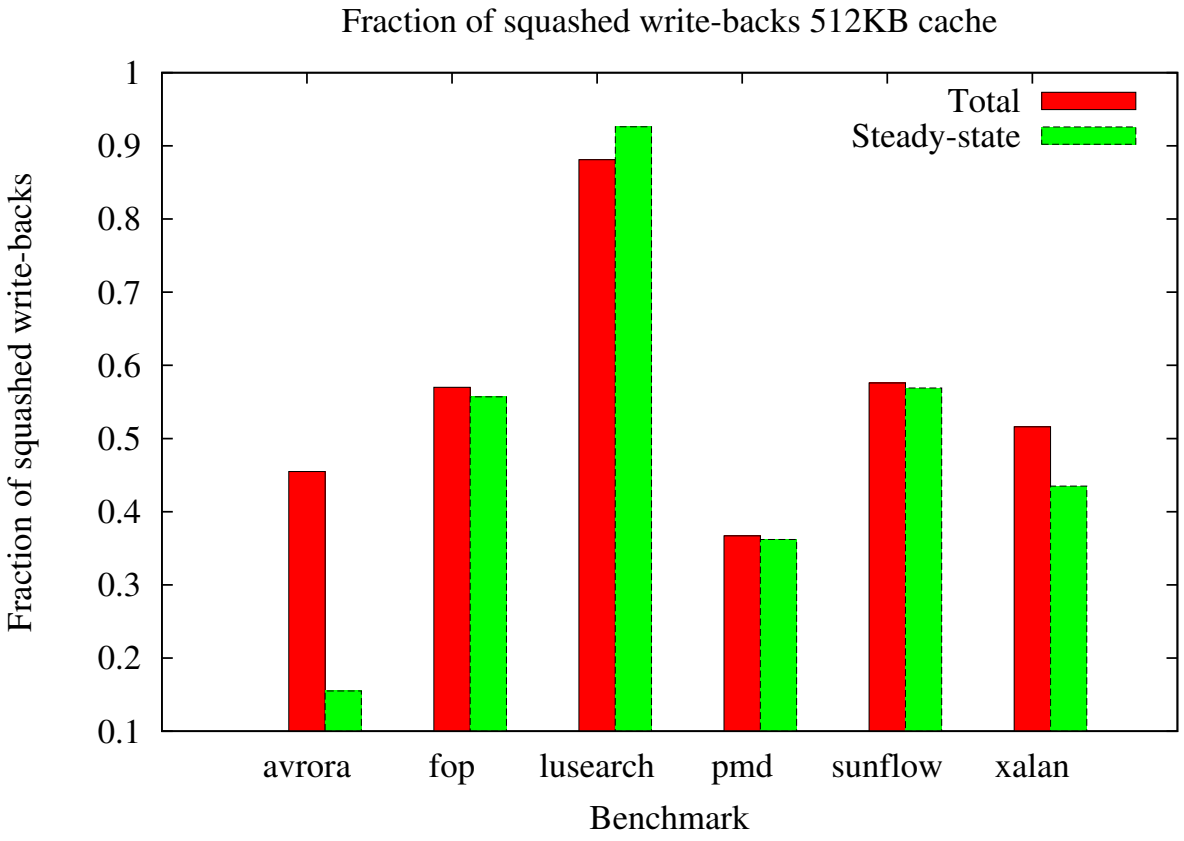

Figure 2.13: Total (steady-state + initialization) and steady-state fraction of squashed write backs with 512KB cache.

As shown in Figure 2.13, all benchmarks show a significant increase in the fraction of squashed write backs when given a larger cache. On average, 50% of all write backs are squashed in steady state execution. We note `avrora` generates little memory traffic, as its

working set fits almost entirely in a cache this large, so the fraction of total squashed write backs is dominated by the JVM and Dacapo startup. This increase in squashed write backs shows our technique is not limited by imprecision, but instead limited by lines evicted from cache before being discovered dead. Results show that expansion of our technique to level-2 caches has potential for significant write savings to higher levels of memory.

2.3.7 References from Multiple Threads' Stacks

We pin objects as live if ever multiple threads reference such objects from their stacks. Heap references from multiple threads pose no problem for our approach. To determine the effect of these gratuitously pinned objects, we ran the benchmarks in a mode where they use a single thread. In terms of the squashed write backs from cache, the results were only slightly worse when using multiple threads for these benchmarks. This could be attributed to good fortune: perhaps the threads did not often have stack references at the same time to the same object. In any case, a more complete treatment of this issue should be studied.

2.3.8 Performance Impact

Previous work that examined similar savings for programs with explicit deallocation [14] squashed write backs in L2 cache at the rate of almost 21%. We find on average 50% squashed write backs in our modeling of an L2 cache (of half the size of [14]). Lacking explicit deallocation instructions, we find the squashed writes through the architecture support presented in this paper. Obtaining strong performance for such garbage-collected programs can have a reasonable impact on energy savings and the longevity of devices with limited write endurance. Applying the analysis of [14], we find the lifetime "gain" for such devices

to be 2–doubling the useful life of such devices, as compared to their result of approximately 1.3.

Write backs are generally performed off the critical path of program execution as write ports are available on upper levels of memory. Because of this, we do not necessarily expect to see any direct performance benefit (i.e., cycles per instruction) from reducing the number of write backs. However, we would expect to see indirect benefits from reducing the total memory bandwidth between L1 and L2 caches.

Our approach requires extra storage and logic on chip. This added complexity will increase energy costs and could affect latency of some instructions. A hardware evaluation is needed to properly analyze the tradeoffs between reduced writes and this added logic. Hardware considerations are addressed in Chapter 3 and 4.

2.4 Summary

To conclude, in this chapter CORC was presented and studied. CORC consists of specialized hardware designed for programs using automatic memory management. As objects often die young in these languages, it is likely many objects die before they are evicted from cache. CORC targets these objects. The cache is augmented with storage and logic to maintain accurate reference counts for all objects allocated. However, once evicted from the cache, an object is no longer reference counted and cannot be determined dead until the garbage collector is run. When an entire cache line is discovered to be dead by CORC, the dirty bit for that line can be cleared. Therefore, it will not be written back upon eviction from the cache, reducing the total number of writes out of the cache. Through simulation, it was

shown that this approach can eliminate an average of 22% of the writes out of a small cache (32 KB simulating a L1 cache). When simulating a larger cache (512 KB simulating a small L2 cache), 50% of the writes out of the cache can be eliminated on average, increasing the lifetime of write limited emerging memory technologies by $2X$. In the next chapter we study recycling dead cache space identified by CORC to satisfy future allocation requests.

Chapter 3

Recycling Dead Data

3.1 Background and Related

Chapter 2 discussed how dead data in cache can be used to eliminate memory bandwidth by reducing writes from the processor caches to main memory. However, write elimination is not the only use for this dead data. In recent years, object-oriented languages with automatic memory management have become mainstream. This is great for software developers, as programming no longer requires explicit memory management by the programmer. However, object-oriented languages can be inefficient due to excessive memory allocation, often allocating many short lived objects [65]. When the allocator runs out of space to make new allocations, a garbage collection cycle may be triggered. The garbage collector itself is often memory intensive, likely touching large portions of the heap and essentially flushing any useful data from the processor cache. Garbage collectors typically pause useful execution of the application while running, leading to delays in execution. However, dead cache space (possibly identified by CORC) can be recycled between garbage collection cycles. Recycling of this dead space reduces heap pressure, allowing for smaller heap sizes or longer execution time before garbage collection is triggered. In this chapter, we study the recycling of dead

cache objects by modifying CORC for the benefit of decreasing allocation times as well as garbage collection overhead.

We start by discussing the main allocation schemes used by languages with automatic memory management.

3.1.1 Allocation Techniques

Contiguous Allocation

Object-oriented programs are notorious for allocating objects excessively. Due to this, contiguous allocation is often the allocator of choice as it is generally very fast. Contiguous allocation is achieved via a *bump pointer* scheme. The allocator is given a large chunk of unallocated heap space, as well as a pointer to the beginning (base pointer) and end of the space. When the allocator is asked for a chunk of memory of size n, it simply checks to ensure the $basepointer + n$ does not overrun the $endpointer$. If this is true, the allocator returns the current value of the $basepointer$ and increments the base pointer by n. If the condition is not true, a garbage collection cycle may be triggered.

Contiguous allocation creates spatial locality between objects allocated by the program, leading to very nice cache effects and improving performance of the program execution, particularly when contiguous allocation is used in the young generation of generational garbage collectors [11]. Because of this, many of the most popular collection algorithms have converged on generational collection with contiguous allocation in the young generation. We evaluate our approach to recycling dead data using *OpenJDK8's* serial garbage collector. It uses generational garbage collection with contiguous allocation in the young generation.

When considering contiguous allocation, space found dead between garbage collection cycles has an address lower than the current base pointer value used by the bump pointer allocator. Therefore, the allocator can reuse this dead space for a similarly sized allocation instead of bumping the pointer up and allocating new heap space. This decreases the amount of heap space used, allowing for smaller heaps or increasing program execution time before it must be paused for garbage collection. This is a benefit to proactively identifying dead data versus waiting for the garbage collector to identify data as dead as done in [59].

Free-list Allocation

The second allocation scheme maintains size segregated free-lists for different size classes. Each list contains chunks of contiguous heap space in the given size class. Upon allocation, the list corresponding to the smallest size class that the allocation fits into is consulted and returns a chunk of memory if available. Garbage collection may be triggered if no appropriately sized memory chunk is available.

Free-list allocation loses the benefits of spatial locality, however contiguous allocation is not an option when reference counting is used to collect objects as contiguous allocation requires a large contiguous chunk of memory to allocate into. CORC however, finds individual objects dead incrementally and these objects may not be contiguous in memory. For this reason, we use small free-lists in hardware to maintain information about dead objects identified by CORC. These free-lists can then be queried for a dead object to be recycled before consulting the software allocator. We note, losing spatial locality for allocations satisfied by our hardware is not an issue as all bytes of a dead object must be resident in cache for it to be valid for recycling in our approach. This is explained further in Section 3.1.2 and we discuss our approach in detail in Section 3.2.

Hardware and Hardware-Assisted Allocation

Although this work addresses young generation contiguous allocation, hardware allocators and hardware acceleration for common general purpose software allocators has been addressed. The most common of these general purpose allocators are Doug Lea's DLmalloc [39], used in LINUX, and Poul-Henning Kamp's Malloc(3) Revisted implementation [37], used in FreeBSD. For an overview of dynamic storage allocation techniques, consult [67].

However, even with good implementations of software general purpose allocators, many applications suffer large allocation overheads due to searching free-lists for appropriately sized free memory chunks. In [57], the authors evaluate the cache pollution caused by some common software allocators and show that an average of 26% of all cache misses can be eliminated by moving allocation into hardware. A second study, [46] shows that program execution can be decreased by up to 50% when memory management is moved into hardware. Therefore, work has been done to accelerate memory allocation by moving memory management into hardware [19, 15, 26]. However, hardware only allocators lose the flexibility of software allocation and do not scale well.

Therefore, [45] proposes a hybrid dynamic memory manager based on [37] and [19]. Software is responsible for creating page headers and finding a page with a free memory chunk available. Once a page is found, hardware is used to search the page bitmap in parallel to find an available free chunk. The hybrid memory allocator gives an average overall speedup in execution time of 12.7% over a software only allocator.

Contiguous allocation does not suffer overhead searching for free space; however, it does suffer a small overhead per allocation to increment and compare the bump pointer. Our work proposes a hybrid allocation scheme. We opportunistically satisfy an allocation request

in hardware via recycling. When no dead object is available, the allocation is handled by the software bump pointer scheme. A thorough study of the performance benefits to satisfying allocation requests through hardware recycling is left for future work, as discussed in Section 5.1.

3.1.2 Cache Effects

Recycling of dead objects in cache lends itself to another benefit. After an object is allocated it is often initialized (in Java, newly allocated heap space is immediately *zeroed*), meaning it is written immediately after allocation. Newly allocated heap space must often be fetched from main memory to perform this initialization if it has not already been prefetched in to the cache (a benefit of spatial locality). However, by reusing dead cache space to satisfy allocations these read misses can be avoided because the memory space being initialized is already present in cache as required by our approach. The effect of decreasing cache misses via satisfying allocations directly into dead cache space is evaluated in Section 3.3.2.

3.1.3 Other Related Work

[21] proposes using the copying phase of young generation garbage collection to relocate objects with high temporal affinity next to each other in memory, creating spatial locality and increasing cache performance. [32] addresses the additional stress placed on the virtual memory manager by garbage collection. Because garbage collection is memory intensive, many page faults are induced by the garbage collector. This work presents *bookmarking collection* to increase garbage collection throughput by eliminating most page faults caused

by garbage collection. These works could fit nicely alongside our work, further reducing the stress of automatic memory management on the memory system.

[51] and [60] propose garbage collection co-processors for real-time systems. The co-processor runs along side the CPU in order to bound pause times during garbage collection using *tracing* collection algorithms. These co-processors generally avoid the use of the cache and connect directly to the memory controller. The purpose of these works is to bound garbage collection pause times for real-time use, where as our work aims to increase execution time between garbage collection pauses for general purpose computing by recycling objects between garbage collection cycles. The techniques described in these works can be used alongside our hardware, speeding up collection cycles when they are needed.

[69] develops a full object-aware architecture to execute alongside the standard processor and traditional cache. The architecture is co-designed with a Java Virtual Machine (JVM) and has a separate cache, designed for objects, that is cooperatively managed by the JVM. An in-cache *tracing* garbage collector is designed, creating better memory efficiency for garbage collection.

As described in Section 2.1, [35] deploys reference counts in hardware throughout all of memory. Objects that have been collected are made available for recycling using a table maintained by hardware. Their approach is complete up to the point that reference counting can be successful, and their scheme integrates well with software-based collectors that can determine the death of objects involved in reference cycles (reference-counting cannot determine the death of such objects). While the cache performance of two of their benchmarks was improved, all of the other 8 benchmarks suffered more misses (up to 4%) in L1 cache. For L2, all benchmarks saw between 0.6% and 6.8% more misses. With our approach, we see decreased cache miss rates across all benchmarks. This is evaluated in Section 3.3.2.

The object-caching coprocessor described in Section 2.1 [17, 18] uses a bit vector in hardware to keep track of available locations for recycling storage into the object cache.

By contrast to the above approaches, our approach leverages a traditional cache: the lines associated with an object are evicted as usual based on program behavior, but we opportunistically recycle storage that is fully present in the cache, if it is found to be dead by our approach.

The reference counting scheme proposed for the Lisp language [55] also proposes recycling. All cells are the same size, and the cache must wait for the reference-counting operations to complete before any other service can be offered. We investigate recycling for programs written in Java, whose requests for storage can be diverse in size. We also examine the extent to which the operations related to recycling storage can be performed off of a cache's critical path while still offering reasonable levels of storage recycling.

3.2 Recycling Details

We now describe our approach to recycling dead objects. We use CORC as described in Chapter 2 to identify dead objects in cache. Consider again the example described in Section 2.2 based on Figure 2.4.

Figure 2.4 shows objects `T` and `U`, of sizes 60 and 68 bytes, respectively, allocated in 5 cache lines. `T` is known to be dead by CORC. A subsequent cache flush would result in the following:

- Line 1's write-back would certainly be eliminated, because the entire line has been determined to be dead prior to eviction.

- Elimination of the writes of the other two lines associated with `T` is less certain. If no other data has been modified in line 0, then its writes can be eliminated. However, `U` partially occupies line 2. Unless `U` is found to be dead prior to line 2's eviction, its write-back cannot be squashed.

- When an application-level garbage collection cycle begins, CORC suspends its activities. At the end of the cycle, CORC restarts cold.

Our approach builds on the following observation: while `T`'s write-backs may not be completely eliminated, the storage associated with `T`, across all lines resident or evicted, is nonetheless known to be dead. If the application subsequently allocates some other 60-byte object `V`, then it could be given the storage that was once associated with `T`. This recycling of `T`'s storage will make the associated lines *live* again, but the writes of *all* of `T`'s data are effectively squashed when overwritten in cache by `V`.

Our goal here is to reuse such storage so that it can satisfy subsequent storage-allocation requests. For our purposes here, recycling can take place only when the following conditions are met:

- The recycled storage must have been determined to be dead. We employ our approach CORC.

- The cache lines associated with the recycled storage must be fully present. While relaxing this constraint would yield a larger fraction of recycled objects, we obtain good results with this constraint in effect and this has the best promise of improving a program's cache hit rate.

Conceptually, we deploy logic in the cache to keep track of available storage of various sizes [55, 17]. In the example of Figure 2.4, once the death of `T` has been observed, line 0 records a free slot of size 60 bytes, at the (virtual) address for `T`. The other lines associated with `T` need not record the available storage. An application essentially encounters a memory-allocation (malloc) barrier at a storage-allocation request. The barrier executes an instruction similar to the PowerPC's `dcbz` instruction, at which time the cache's available storage table is consulted. A suitable virtual address is provided and returned by the instruction if the requisite storage is available in cache, and the in-cache storage can be initialized to 0 as would be the case with `dcbz`. Otherwise, `null` can be returned and the run-time system can then turn to the storage heap to satisfy the request. This barrier could be imposed selectively: the correctness of the application does not depend on the barrier, as storage could always be provided directly by the run-time heap.

Deploying such logic directly in a cache could interfere with the cache's normal operations, which are intended to suffer no unnecessary delays. We first evaluate the efficacy of our approach on some Java benchmarks, assuming the logic associated with determining storage availability is executed in the cache, synchronously with all related cache behavior. We later evaluate our approach when moving the reference counting and recycling logic off of the cache's critical path.

3.3 Evaluation

3.3.1 Experimental Setup

To facilitate comparison, we obtained traces for the 6 DaCapo [13] benchmarks studied in Chapter 2. Those traces were generated by instrumenting the `OpenJDK` (version 1.8.0) JVM (Java Virtual Machine). The traces captured allocation requests (using contiguous allocation), storage-referencing requests for the heap and stack, method calls and returns, and onset and completion of the JVM's garbage-collection cycle. Each of these traces captured the first 50 million lines of JVM activity. Within each trace, the first 10 million lines are attributed to JVM startup and benchmark harness (Da Capo) initialization; the remaining 40 million lines are reflective of the rest of each benchmark's execution. These traces contain only those instructions necessary to implement the the limited reference counting approach: arithmetic instructions, register-only instructions, and jump instructions do not appear in the traces.

Some of our experiments were run on the full, 50 million line traces; others were run by sampling, skipping the first 10 million lines and applying simulation to the next 5 million lines. After 10 million lines, the benchmarks move beyond their start-up phase and exhibit their steady-state behavior. For a 32K-byte cache, the longer traces could take several days to complete simulation; the shorter ones completed in a few hours. The 512K-byte cache simulations run much longer, with some full-trace simulations taking up to a week to complete.

We implemented CORC, with one important change: we introduced a virtual memory (VM) system in the simulation for the following reasons:

- Real systems using our approach will most likely support virtual memory.
- Our cache must respond to a storage request of b bytes by furnishing the address of an available block. That address will subsequently be used by the application, so it must be valid in the application's address space, which is virtual.

For the results reported here, our VM consisted of 1024-byte pages. Virtual addresses were 64 bits, and physical addresses were 32 bits.

Our recycling approach works as follows. We first describe how information related to available object sizes is maintained:

- An object descriptor for each allocated object `T` that contains the virtual address and size of `T` is created on object allocation.
- The recycler is initialized with a value indicating the required fraction of an object that must be resident for the object's storage to be considered recyclable. For the results presented here, that fraction is 1.0.
- When any line is evicted, the objects contained in that line are updated to reflect the number of bytes still resident in cache for those objects. Recalling Figure 2.4, some lines may be fully occupied by `T`, but (at most 2) other lines may hold only the first or last portions of `T`.
- Any objects dropping below their required fraction that must be present become ineligible for recycling.

  It is possible that future program behavior could reload an object's lines, but we do not maintain information outside the cache. Thus, any subsequent stores of reloaded

lines cannot be squashed, and we currently do not consider any associated objects' storage for recycling.

- When the death of an object `T` is discovered in cache, the storage associated with `T` becomes available for recycling, if that storage is still eligible as described above.

  While the number of available block sizes could be large, we show that a small assortment of sizes could be maintained while still satisfying most requests.

- Post-mortem, any line evictions of `T`'s storage cause an update of the fraction of resident bytes, and if that drops below the required fraction that must be present, `T`'s storage becomes ineligible for recycling.

The set of available storage blocks is maintained using free-lists segregated by size. The first element of each size's list is referenced by a (hardware) table. The internal list's links for each element e are implemented using e's in-cache data, which is resident in cache but dead. The double links facilitate removing an element when its fraction of resident bytes drops below the required value. Otherwise, each list is manipulated only at its head: freshly dead objects of size b are inserted at the beginning of b's list, and requests for storage of size b are serviced at the list's head. Recycling then occurs as follows:

- When an object allocation request for b bytes is encountered, the software barrier issues a single instruction to poll the cache for a block of storage whose size could satisfy the request.

- If a block of size b is available, it is removed from its free list, initialized to 0, and returned to the application for use. Otherwise, `null` is returned.

- In a live implementation of our approach, the application would use the recycled block as if it had been initialized and returned by the heap allocator.

  However, our simulation traces were produced by a JVM without a recycler. Thus, while we may find a suitable block of b bytes at address p, the application's trace continues to reference storage by the address (say, q) returned by the heap allocator. We therefore take the following actions:

  - We simulate the action of our recycler in the trace by dynamically remapping all references in the interval $[p, p + b)$ to the interval $[q, q + b)$. Our simulator's VM component facilitates this mapping.
  - If an actual garbage-collection cycle is run during the trace, we abandon all information pertaining to recycling dead storage at the onset of the cycle. When the cycle completes, we restart the determination of dead storage and blocks available for recycling.

    The traces used for these experiments were generated on large heaps to minimize the number of garbage-collection cycles. On average, each trace had 2 such cycles.

3.3.2 Results

Experiments for an L1 Cache

Our experiments here concern a L1-type cache with the following characteristics: 32K bytes total, 32-byte lines, 2-way associativity, write-backs performed at the line level.

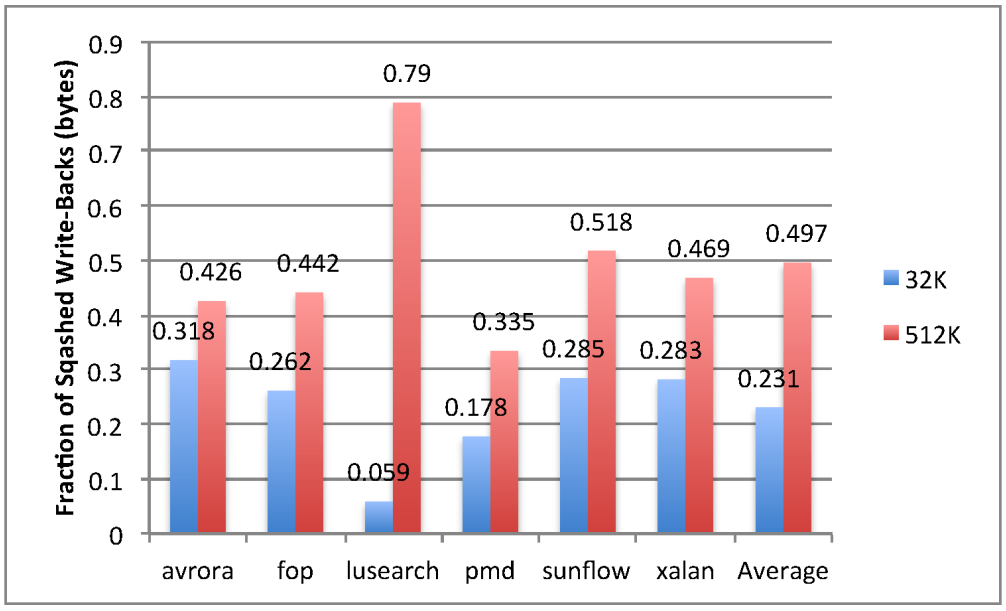

Figure 3.1: Our reproduction of the results from Section 2.3, but with a virtual memory interposed between the program and the storage subsystem. The minor differences between our results and those in Section 2.3 are attributable to the VM's assignment of physical addresses, and the associated changes to their mapping to cache sets.

Figure 3.1 shows the fraction of write-backs squashed for the L1 cache (and the L2 cache described below). On average, 23% of the bytes that would have been written to memory were squashed by determining dead storage in cache.

Note that the *squashed* bytes are those that would have otherwise been forced from the cache and written to memory. In CORC, a write-back of a line is squashed only if the entire line is known to be dead. Recalling Figure 2.4, the bytes in line 1 fall into this category. The other portions of `T`, while unsquashable, may nonetheless be available for allocation along with the squashable bytes of `T`. Thus, more storage may be available for recycling than those measured as squashed bytes, and the pool of bytes available for recycling may exceed those associated with Figure 3.1. Another point here is that a given sequence of dirty bytes can only be squashed once before being reallocated. On the other hand, that same

sequence of bytes could be recycled an arbitrary number of times if they become dead prior to a subsequent request.

We now turn to measuring the effectiveness of recycling dead storage. The experiments reported here were conducted as follows:

- All logic needed to find and recycle dead storage is performed synchronously with cache operations.
- As illustrated in Figure 2.4, an object can (and for 32-byte cache lines, typically will) span multiple cache lines. While write-backs can be squashed for any of those lines under the right conditions, we insist here that all of an object's bytes be fully present in cache to be eligible for recycling.

  This is managed by *shooting down* available-storage entries when any of their cache lines are evicted.

Figure 3.2 shows the fraction of allocation requests that were satisfied in the benchmarks' executions. For the L1 cache, an average of 27% of the allocation requests were satisfied by the cache itself. For most of the benchmarks, the reported aggregate recycling values in Figure 3.2 were observed throughout their execution. The `sunflow` benchmark was an exception: during a long period of its execution, *every* storage request was satisfied by recycling.

L2 Cache

Figures 3.1 and 3.2 also show results for a larger, 512K-byte cache, which could reasonably serve as an L2 cache. For our experiments, this cache had 64-byte lines, and 4-way

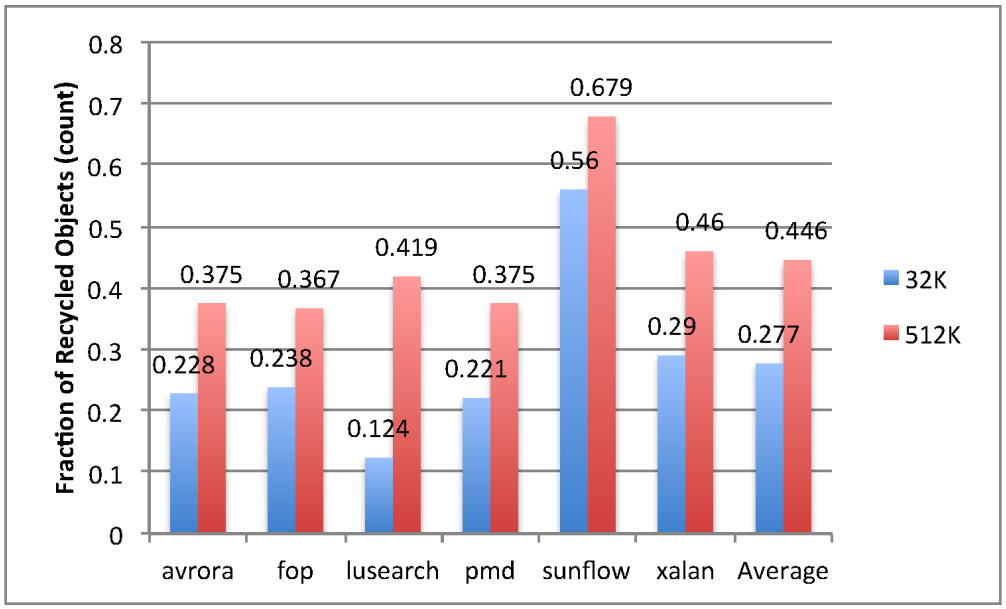

Figure 3.2: Fraction of allocation requests that were satisfied by the cache, using recycled storage.

associativity. Because determining dead storage in cache is often a race between detection and eviction, we see (as we did in Chapter 2) improved performance on the L2 cache. The `lusearch` benchmark is markedly improved in Figure 3.1, and the availability of the extra dead storage translates into ∼3x improvement for object recycling in Figure 3.2. On average, the L2 cache allowed nearly 45% of the application's storage requests to be satisfied directly by the cache.

Recycling: Objects or Bytes

The cost of satisfying a storage allocation request involves some relatively fixed cost, regardless of size, along with some cost related to the size of the request.

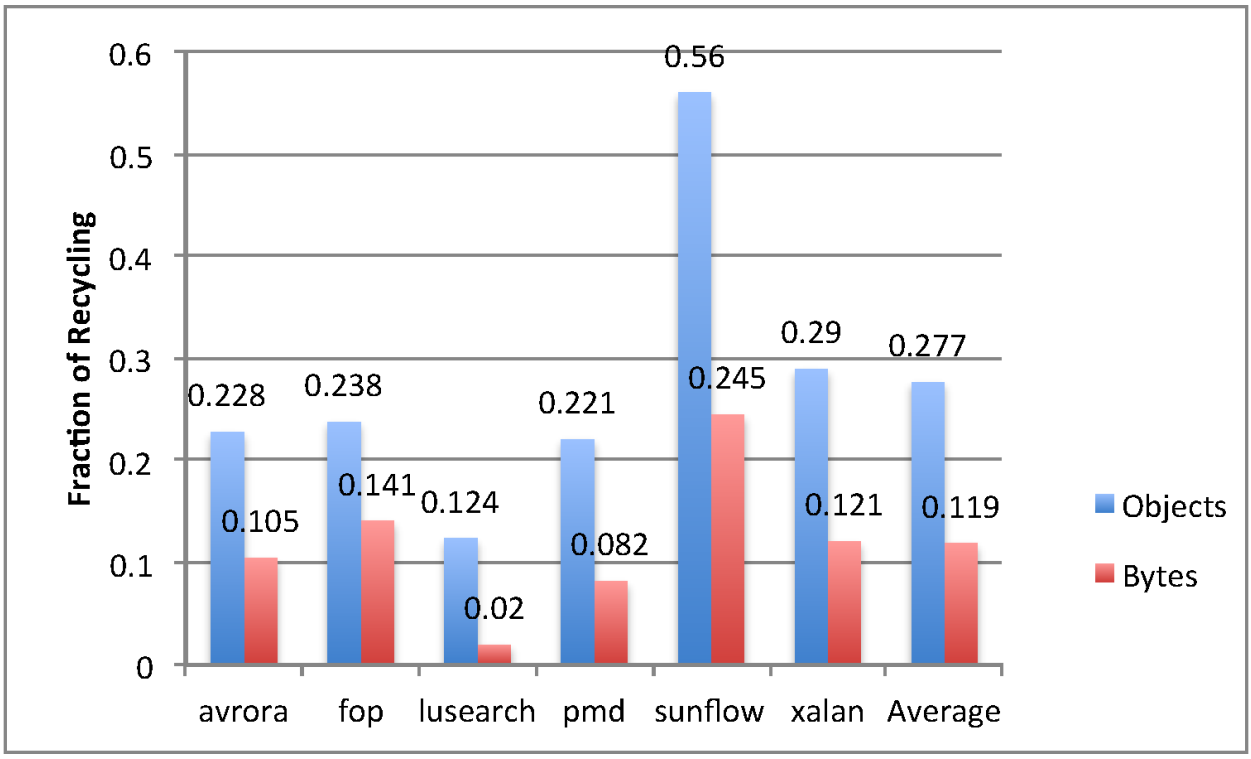

Figure 3.3: Results are shown for the 32K L1 cache. *Objects* refers to the fraction of object requests that were satisfied by the cache. *Bytes* refers to the fraction of all allocated bytes that were satisfied by the cache.

Figure 3.3 shows the fraction of recycling requests, by object count and by bytes allocated. The difference between the two can be explained by the skewed nature of allocation size requests. As we report in Section 3.3.2, most allocation requests are for small objects, such as 24, 32, or 40 bytes. Of course, the benchmarks do contain some requests for much larger storage, such as 32K bytes. Those requests are likely unsatisfiable in L1 cache, and they contribute to the overall storage allocation byte count.

However, we do see an average of 12% heap space reuse. This shows our approach can lead to reduced heap sizes or longer execution periods before garbage collection is triggered. A quantitative analysis of this is needed, but left for future work as described in Section 5.1.

Because most objects are small, and because each allocation request requires some constant overhead to find suitable storage, object-allocation-count may be a more important statistic. Moreover, for Java, the storage must be initialized to zero. While some systems offer cache instructions (e.g., PowerPC's `dcbz` [33] to facilitate this initialization, any storage not present in cache would incur a cache fault. With our approach, if storage is found in-cache, it is already present, so no faults are incurred. The instruction that requests the storage can also trigger the initialization, which can proceed concurrently in all cache lines associated with the storage.

Requested and Available Storage

Our in-cache storage allocator has thus far responded only by finding an exact fit for the requested storage size. While the results reported thus far are encouraging, it is possible that a request for b bytes could be satisfied by a recycled object whose size is greater than b. To investigate this idea, we next study the size of storage that is requested by our benchmarks and compare this with the list of recycled storage sizes that are available for allocation in cache.

We instrumented our simulator to accumulate the total number of storage-allocation requests by size, and the results are shown in Figure 3.4. To avoid clutter, we display results for requests that account for 75% of all storage-allocation requests. For any block larger than 96 bytes, its storage-allocation requests accounted for less than 1% of all allocations. As expected, most storage allocation requests are for relatively small blocks of storage. Each benchmark requested 32 bytes more often than any other size, and over all benchmarks, 32-byte blocks account for over 33% of all requests. Such blocks account for over 68% of

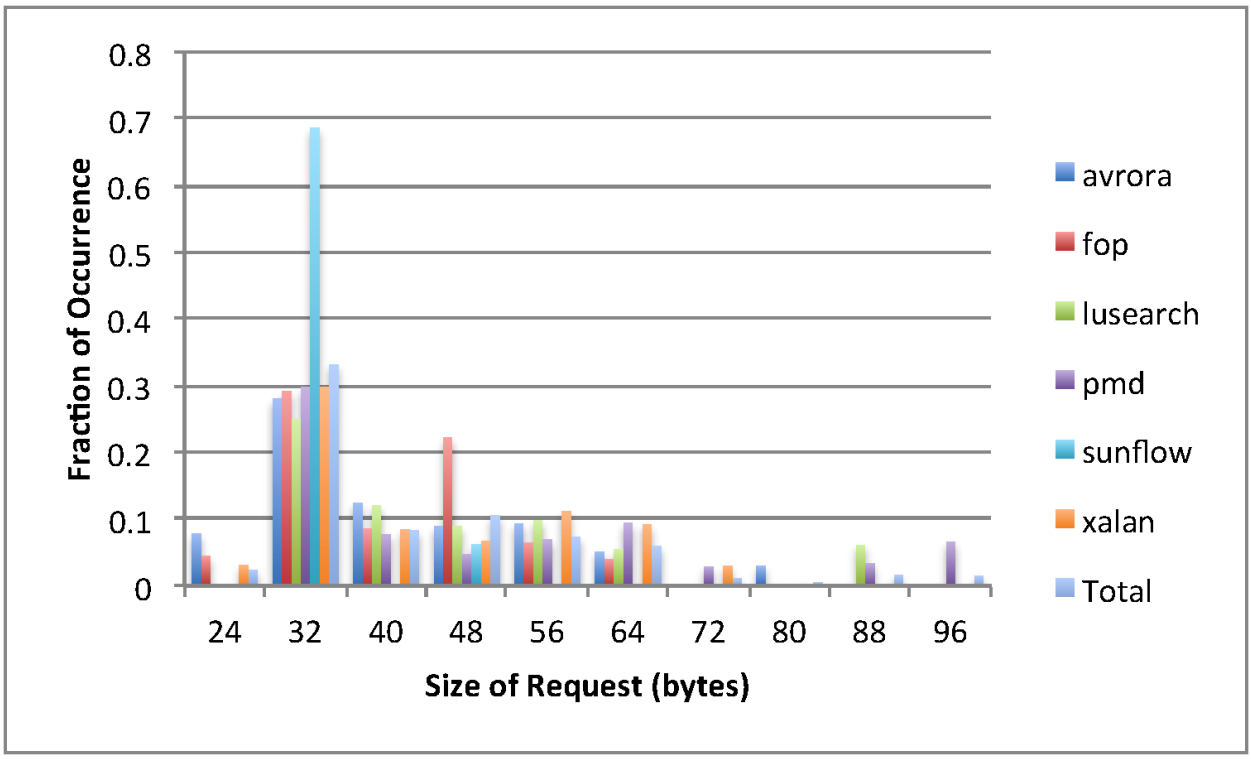

Figure 3.4: Size of allocation requests by benchmark, accounting for 75% of all requests.

`sunflow`'s allocation requests, which contributes to our success in recycling storage for that benchmark (Figure 3.2).

However, the benchmarks also make frequent use of slightly larger blocks. The `fop` benchmark uses 48-byte blocks 22% of the time, as compared with its 29% use of 32-byte blocks. This raises the question of how much our approach could be improved by considering the allocation of larger blocks to satisfy a storage-allocation request.

To study the availability of storage blocks, we sampled the cache's available sizes every 20,000 trace instructions, with the study conducted over the benchmarks' entire 50-million line traces. Figure 3.5 shows the results of the 750 samples. For each benchmark and size, the availability of the size among the 750 samples is shown. This experiment was conducted

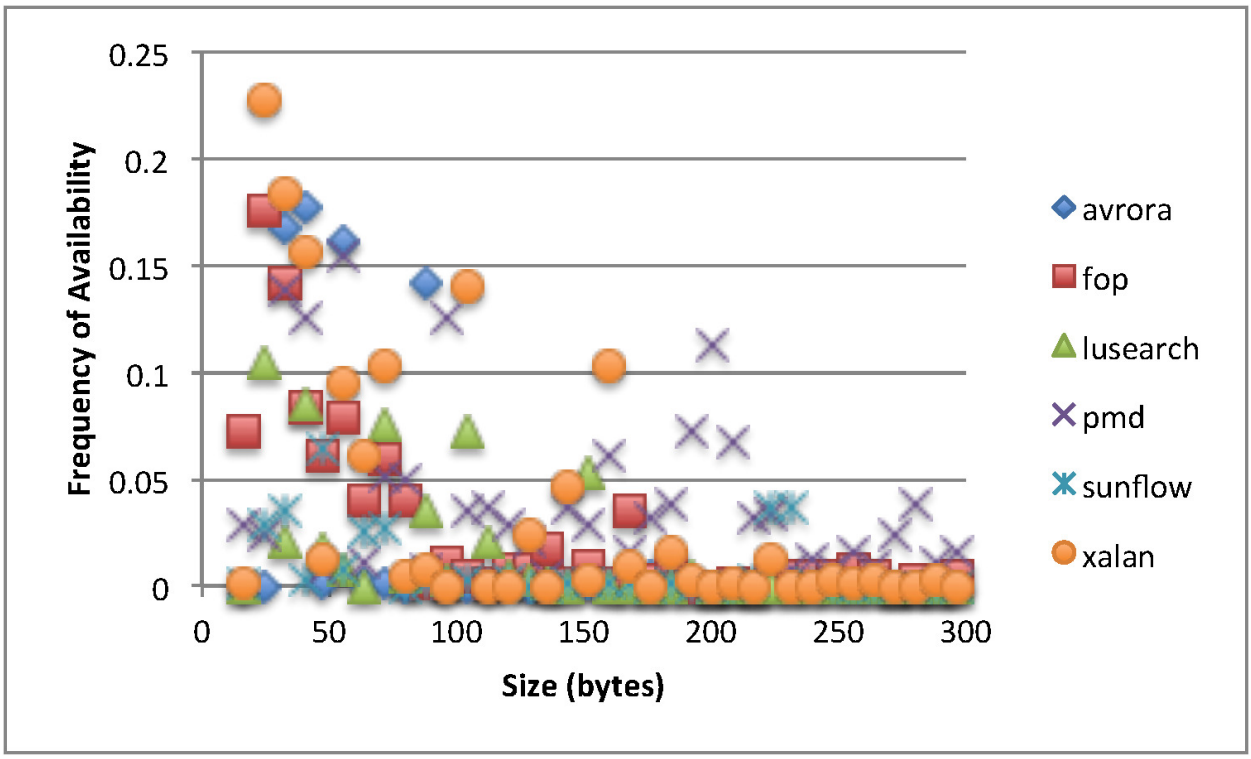

Figure 3.5: In-cache storage availability for a 32K cache.

while recycling was taking place. Thus, the available blocks shown in Figure 3.5 are present because they were not needed by exact-fit recycling. It is these blocks that could satisfy a request at or below that block's size.

To illustrate the potential for using larger blocks, Figure 3.6 combines data from Figures 3.4 and 3.5 for the `avrora` benchmark. For each size, its frequency of allocation and availability for a 32K and 512K cache are shown. Requests for 32-byte blocks occur 28% among all storage requests. However, a 32-byte block is only available in 17% of our samples. Looking at larger blocks, we find that 40- and 56-byte blocks are available in 18% and 17% of our samples, respectively.

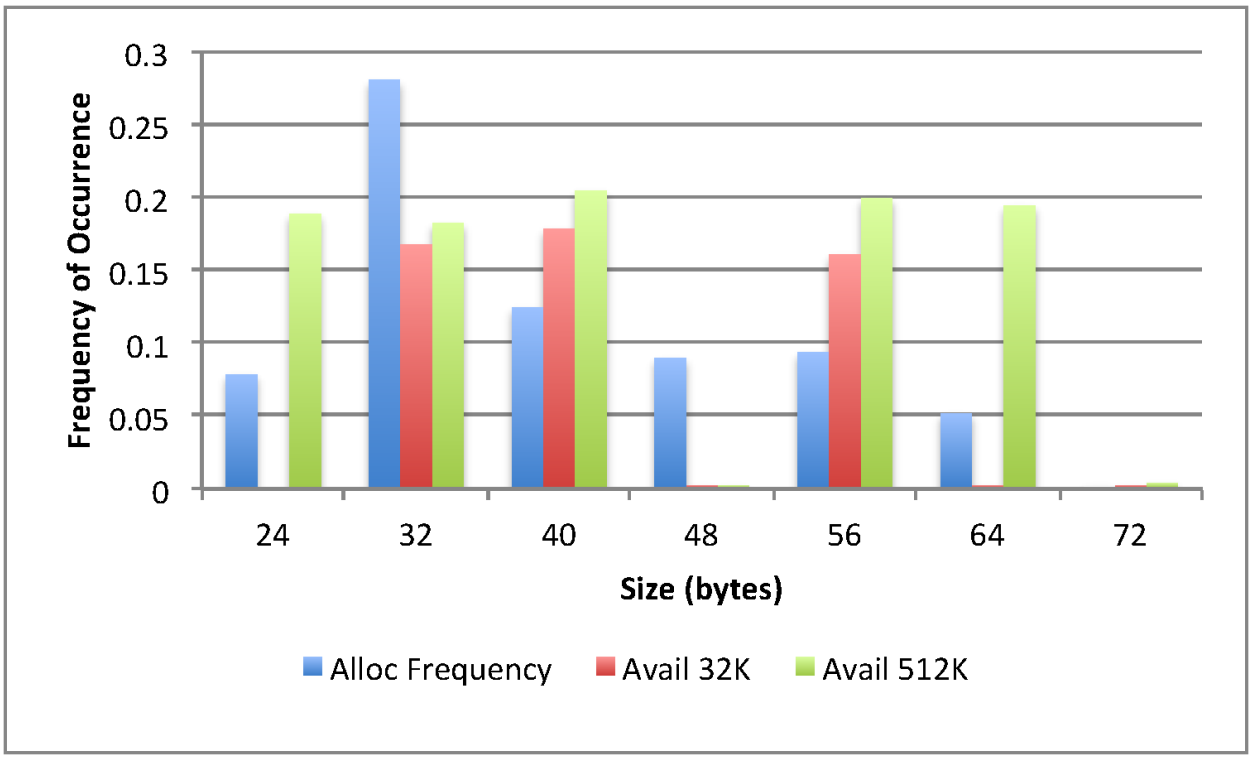

Figure 3.6: Size of allocation requests and recycled storage availability for 32K and 512K caches.

We modified our in-cache allocation strategy to look first for a block of the requested size, but if such a block is not available, then we attempt a first-fit-by-size (FFBS) to satisfy the request. For `avrora`, this will most likely result in a 40- or 56-byte block being allocated to a 32-byte request, if no 32-byte block is available. Suppose a 40-byte block is used whose virtual address is p. In terms of the effects of this allocation on the rest of the running application and the garbage collector, the size of the storage at p is still known to be 40 bytes by the runtime heap manager. While we found the storage to be dead in-cache, the collector must not have run yet, so recycling the storage at p makes the 40-byte block live. To the runtime heap manager, it is as if the 40 bytes of storage are still in use. Our in-cache allocator returns the 40-byte block, even though only 32 of those bytes will be used. Liveness is preserved

because all references to the 32-in-40 byte object are to p. For the purposes of reporting statistics about the fraction of bytes that are satisfied by in-cache allocation, we count the 32 bytes (not the 40) as being allocated by the cache.

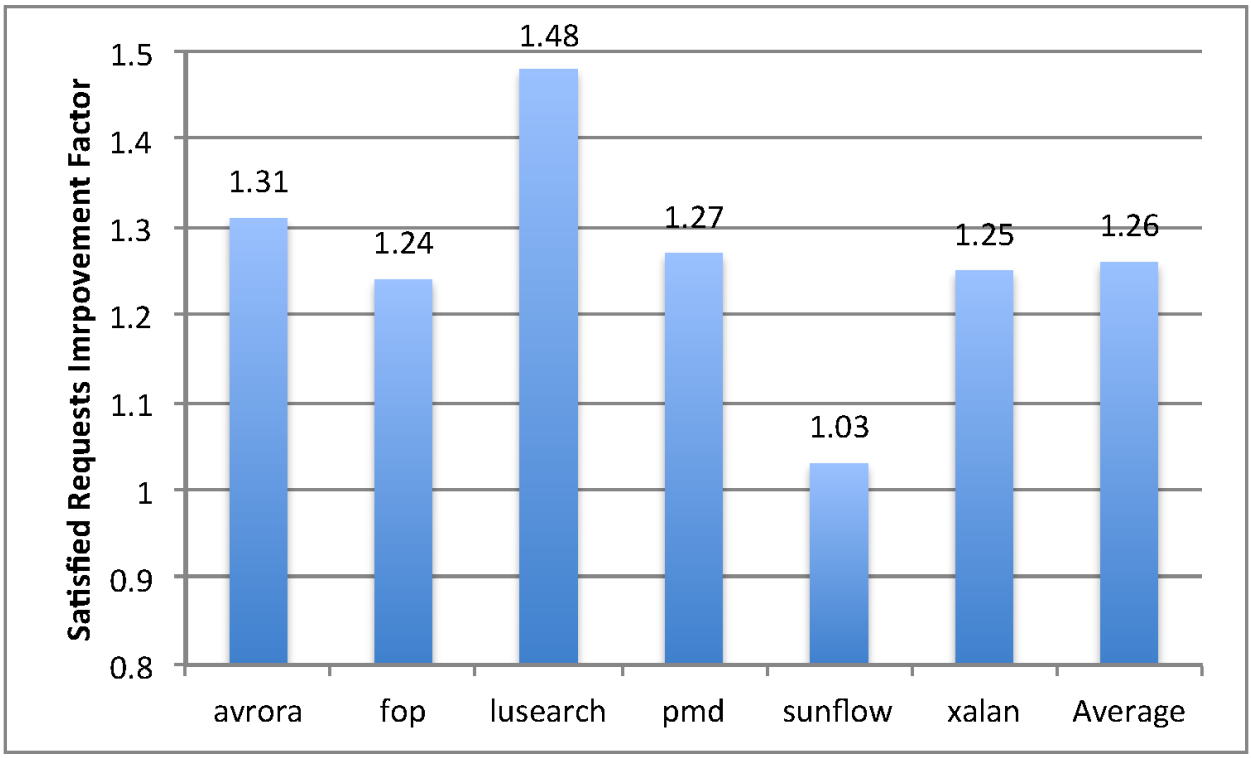

Figure 3.7: Improvement due to FFBS, displayed as the ratio of allocation requests satisfied by FFBS to the allocation requests satisfied by exact fit.

We ran our FFBS in-cache allocation policy on all benchmarks, and the results are shown in Figure 3.7. Overall, a 26% improvement is seen. The least affected benchmark was `sunflow`, which as stated previously already enjoyed strong recycling using just 32-byte blocks.

With FFBS in place, we again sampled the sizes of storage blocks available in cache, and the results are shown in Figure 3.8. Comparing Figures 3.5 and 3.8, we see that the larger block samples are mostly gone, having been used to satisfy the allocation of smaller blocks.

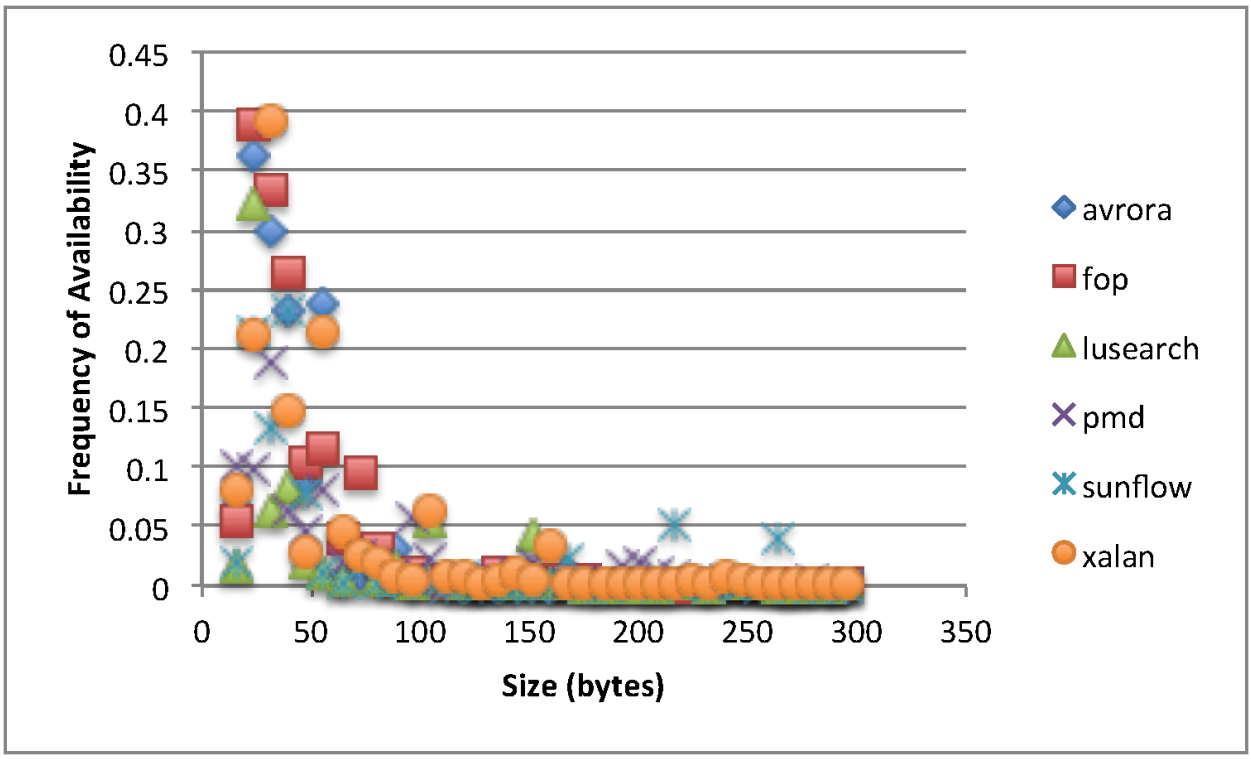

Figure 3.8: Availability of storage blocks in cache for FFBS.

Using a larger block may deprive a downstream allocation request of in-cache storage. However, Figure 3.9 shows that the fraction of allocated bytes satisfied in-cache actually rises slightly when larger storage blocks are used as necessary to satisfy an allocation. These results account for the requested storage size, not the size of the block actually used. Thus, if 32 bytes are requested and the request is satisfied by a 96-byte block, this counts only toward 32 bytes of storage satisfied in-cache.

Effects on Cache Misses

For allocation requests satisfied by the cache, all bytes of the allocated block must already be in cache. This should improve an application's hit rate for the following reasons:

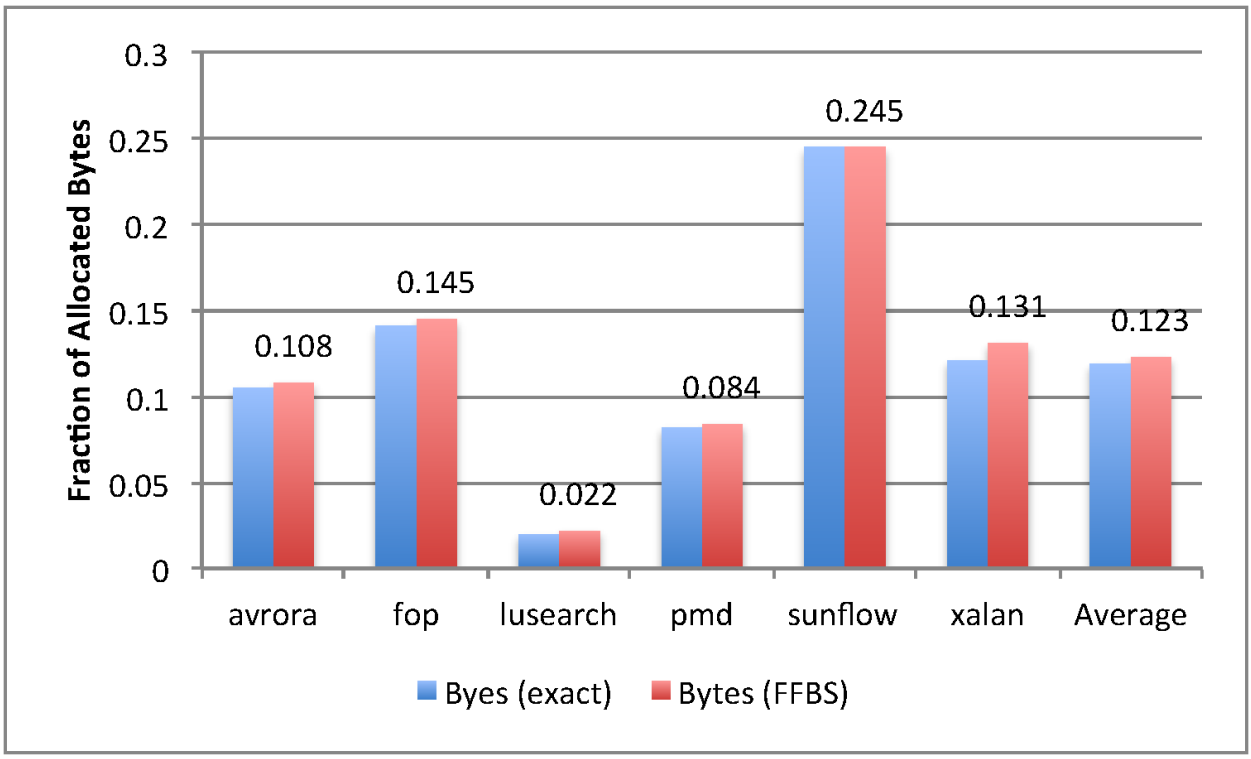

Figure 3.9: Comparison of recycled bytes for exact-fit and first-fit-by-size (FFBS). Data values for the exact-fit bars can be found in Figure 3.3.

- No cache faults would be experienced for (Java-mandated) initialization of the storage. Some caches offer special zero-initialization instructions, but many do not. In either case our in-cache allocation will experience only cache hits.

- By using recycled storage instead of freshly allocated heap storage, the pressure on the cache is reduced.

Figure 3.10 shows the improved hit rates for in-cache allocation. While the hit rate for allocations improves nicely, allocations account only for a small fraction of the accesses to an object's storage. Figure 3.11 shows the overall improvement, which ranges from 0.002 to 0.007, with an average improvement of 0.005. On average, the hit rate rose from 0.923

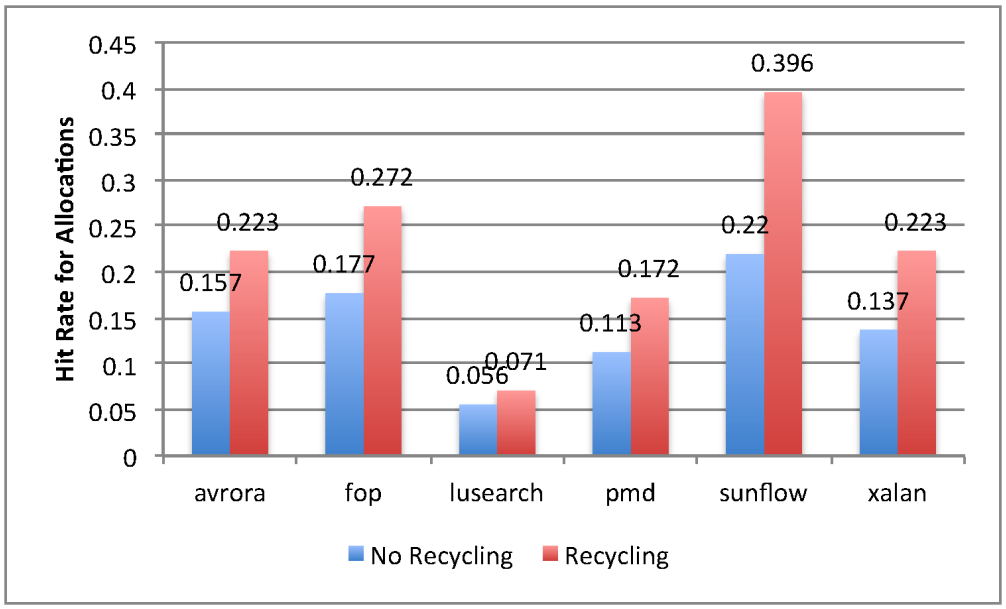

Figure 3.10: Hit rate for storage allocations and initialization.

to 0.928. If off-cache accesses are 10 or 100 times slower than L1 cache, this translates into a speedup of 1.03 or 1.06, respectively. Other speedups may be seen because of reduced heap pressure leading to longer execution time between garbage collection cycles and reduced CPU time handling allocations from the software heap; a quantitative study of this is the subject of future work, as described in Section 5.1.

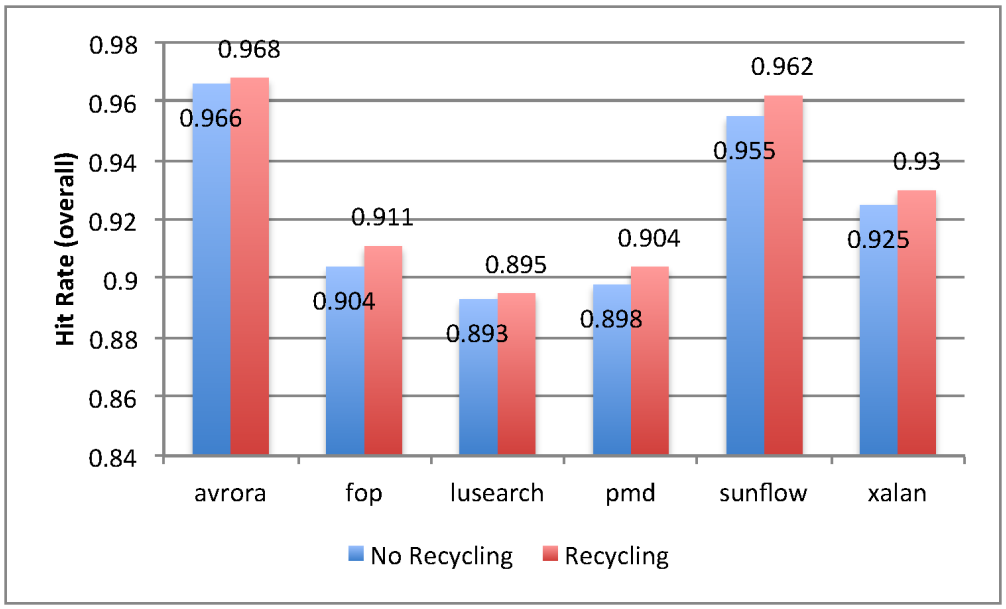

Figure 3.11: Overall hit rate comparison.

3.4 Limitations for Reference Counting

3.4.1 Caches and Critical Paths

Cache systems are designed for low latency, but the results presented thus far assume that all computations to determine dead and recyclable storage are carried out in the cache. Moreover, actions taken on behalf of a single cache instruction are thus far presumed to execute before the next cache instruction. As an extreme example, consider a reference p within an object that is be set to `null` by a JVM `putfield` instruction. If p was not previously `null`, and pointed to some object q, then q's reference count must be decremented. If that count reaches 0, then any objects referenced within q must have their reference counts decremented, which could cause further object deaths. In the worst case, this cascade of

object deaths could happen in every cache line. During this activity, the cache would be unavailable to handle loads and stores issued by the application.

While the results reported in Chapter 2 and so far in this chapter are encouraging, it is unreasonable to insist that all of the associated operations should be performed synchronously by a cache. In this section, we present the following:

- If the logic to support CORC is computed off the cache's critical path, how much slower can that logic run while still providing good results?
- Based on the profile of loads and stores relative to non-storage operations, how well can off-the-critical-path logic perform when compared to the results from Chapter 2 and previously in this chapter?
- Can hardware be designed to meet timing requirements for our approach to remain effective? What trade-offs may be made to maintain effectiveness?

3.4.2 Supporting Logic Off the Critical Path

Based on the above observations, we redesigned CORC so that the cache portion of the simulation handles only the traditional cache traffic. Figure 3.12 shows the new data path for instructions. All nonessential activity is relegated to one of two queues. The *Reference Count Queue* as shown in Figure 3.12 handles all reference count instructions, which execute off of the cache's critical path. However, the cache must be modified a bit to assist the reference counting structures. These cache modifications are on the critical path, however they do not affect cache latency much as they piggy-back on standard cache instructions. The modifications are as below:

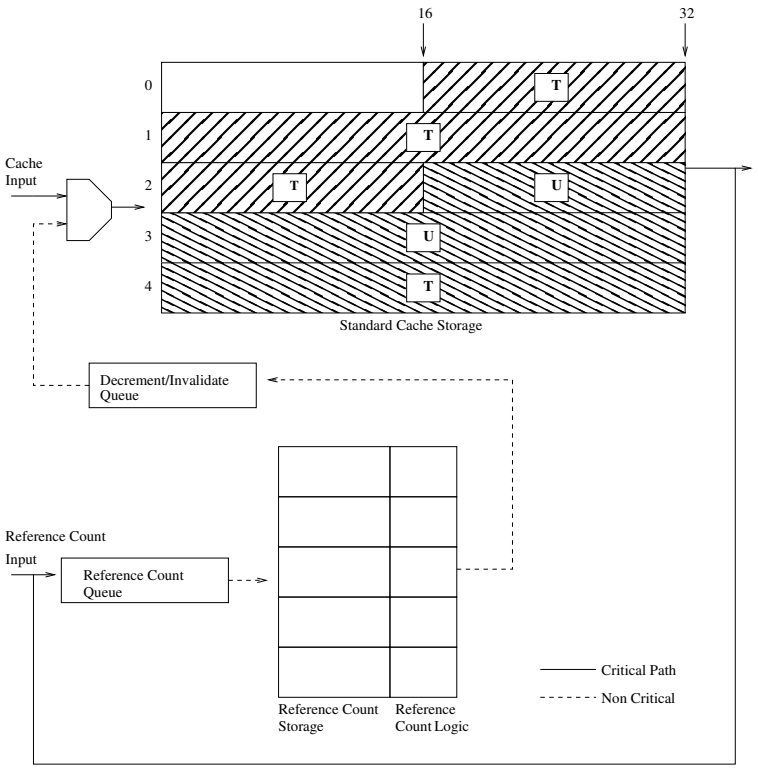

Figure 3.12: Figure showing the reference count storage and logic moved off of the cache's critical path.

- The *reference store* instruction not only increments the reference count of the object being stored, but also decrements the count of the object that is having it's reference overwritten. This overwritten value must be read from the cache. However, the cache is already performing a store operation, so we modify the cache's standard store to read the old value before overwriting it. The cache then passes the old value to the reference count queue as a decrement.

- In order to maintain consistency between the data currently in the cache and the data in the reference counting structures, the cache is modified to output a tag update instruction to the *Reference Count Queue* when a new address is loaded into cache.

The cache must also assist in reference counting in other ways that could delay standard cache operations. In order to avoid this delay a second queue is created for these operations and they are executed as the cache is free. The *Decrement/Clean Queue* includes:

- *Clean* operations - when a cache line is discovered dead, a *clean* is pushed into the queue from the reference counting logic. When the cache is not busy, it will unset the dirty bit for the cache line if it remains in cache. The line is also set to be the least recently used line. The line will then be the next to be evicted, potentially saving useful data in cache, and the write-back will be avoided as the line is clean.

- *Cascading decrements* - when an object is discovered dead, the reference counting logic knows where references are contained in that object. The address of a contained reference is pushed onto the *Decrement/Clean Queue*. These locations must then be read by the cache (a standard read instruction) in order to get the object ID to be decremented, which is then pushed onto the *Reference Count Queue*.

The *Decrement/Clean Queue* containing operations executed by the cache is emptied as the cache is not busy with standard loads and stores. The *Reference Count Queue* is serviced off the cache's critical path as the reference counting structure is available.

As cache's are designed to be as fast as possible, particularly the level 1 cache, and our added logic is more complicated than standard cache logic, we experiment to see how fast our reference counting structure must be in order to remain effective. Moreover, we assume the cache is *non-blocking* [8] so that it operates at the best possible speed. We vary the latency of the reference counting hardware versus that of the cache.

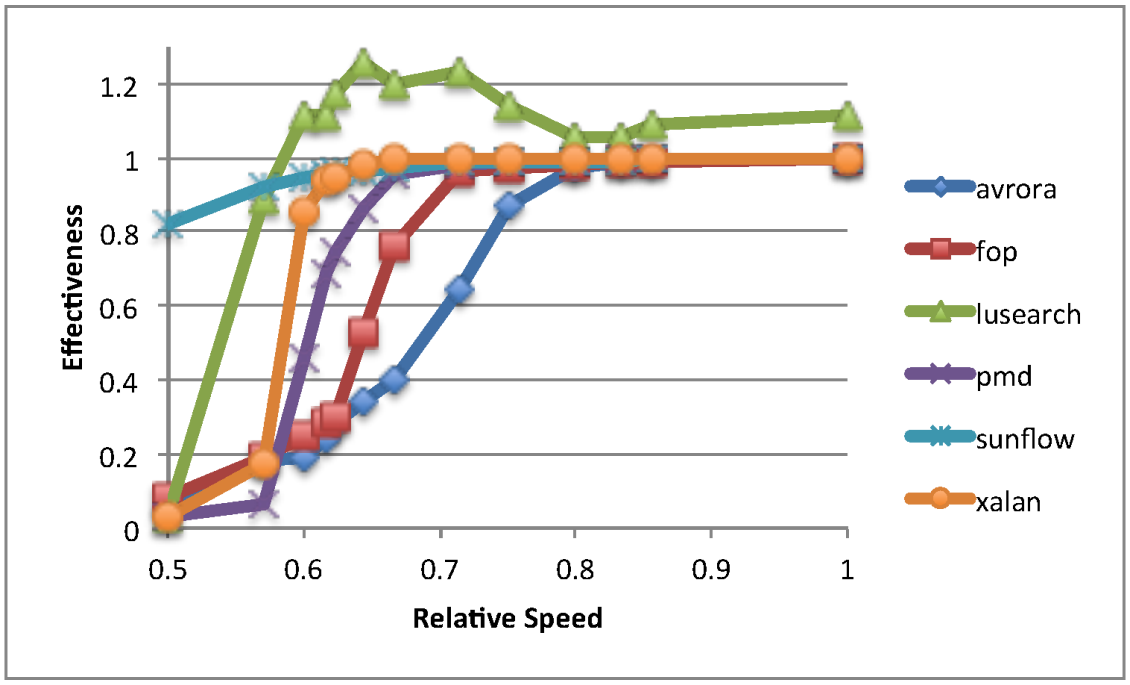

Figure 3.13: Degradation of our ability to find dead storage as the *Reference Count Queue* service rate slows relative to the cache rate.

Figure 3.13 shows the results of slowing the *Reference Count Queue's* service rate relative to the cache speed. At the right end of the graph, the queue is serviced at the rate of one action per cache transaction. Thus, a normal read or write would allow the queue to execute one action off the critical path (the cache and reference counting hardware have the same latency). At the left end of the graph, the queue service rate is half of the cache rate (reference counting hardware runs at 2X the cache latency). Here, a queue item is serviced only after two traditional cache transactions have been completed.

Although we ran these experiments across a wider range of relative speeds, the performance of our approach drops precipitously once the service rate drops below 0.6 of the cache speed. The one exception is the `sunflow` benchmark, which has a long period in which the same object sizes (32 bytes) dies and becomes recycled immediately. For the other benchmarks,

once the relative speed of our reference counting hardware drops to 0.5 of the cache speed, our approach is completely ineffective. Figure 3.13 shows an anomaly for `lusearch`. Its original fraction of squashing writes for the L1 cache was only ∼6% (see Figure 3.1). When a line is found dead, it becomes clean, and can be chosen by a cache set for subsequent allocation to any address that maps to that cache set without causing a write-back. For `lusearch`, introducing the queues delays this action, which has consequences for cache misses downstream. Because the `lusearch` squashing fraction was already so small, the data in Figure 3.13 is sensitive to the small rise in squashing with the queues in place for this one benchmark.

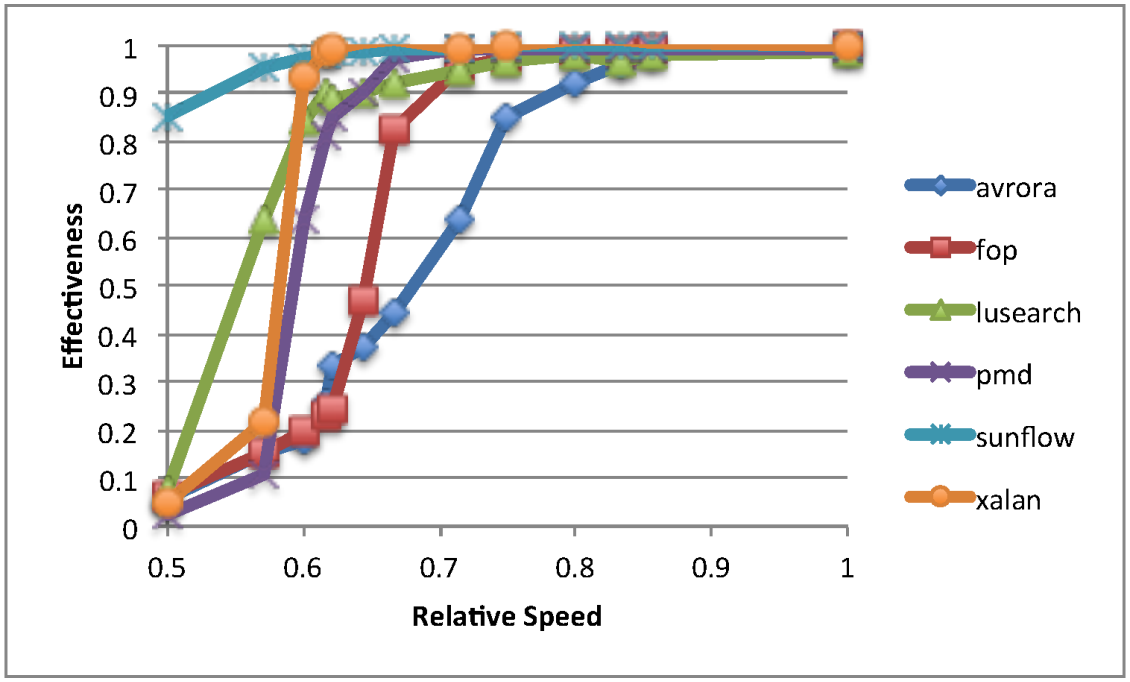

Figure 3.14: Degradation of our ability to recycle storage.

We see similar results in Figure 3.14 in terms of how object recycling is affected by slowing down the off-critical-path activity. Based on these results, it appears that for the 32K L1 cache, both storage recycling and the concomitant detection of dead storage fall of sharply

at some point. Across all of our benchmarks, the degradation begins at the relative speed of 0.8.

Realistic Pacing with Standard Cache Traffic

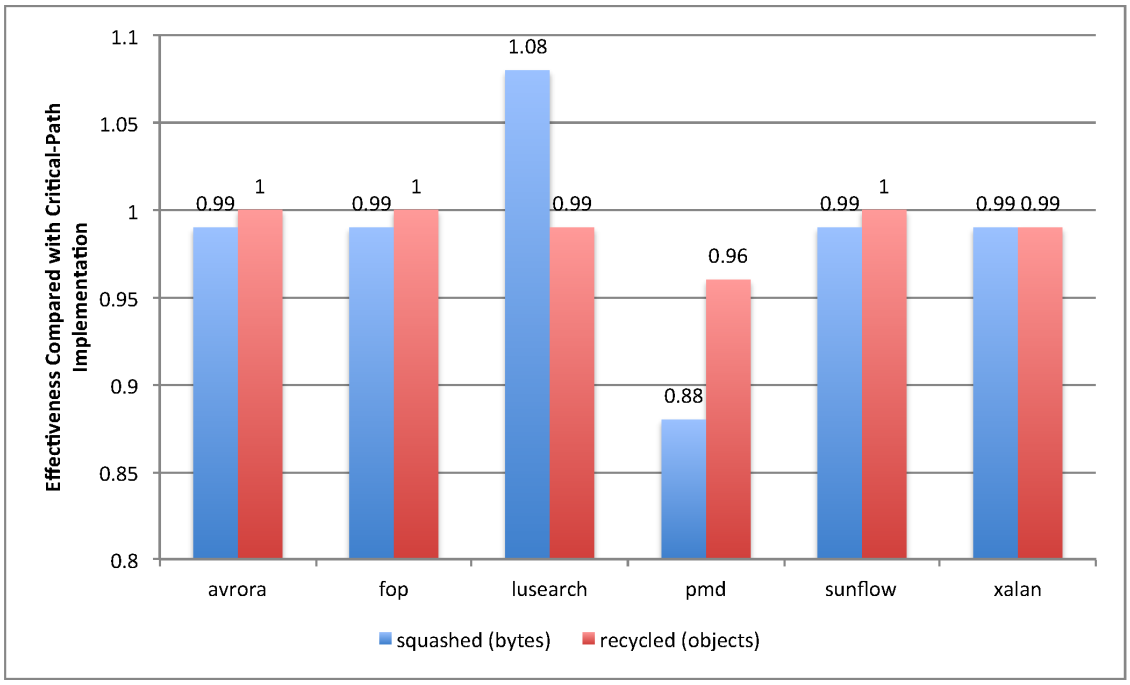

Figure 3.15: Results obtained with realistic work loads.

The analysis above presumes that all instructions involve the cache, but studies have shown that the ratio of loads and stores to other instructions is somewhere between 20% (for RISC architectures) and 50% (for non-RISC architectures) [30]. For Pentium architectures, one study showed loads and stores comprise 0.58% of all instructions with a standard deviation of 0.05 [24]. We modeled an instruction stream using those statistics, and results obtained as compared with our results from Section 3.3.2 are shown in Figure 3.15. The experiments presented here assume the following:

- The off-critical-path circuit runs at half the speed of the cache. Another view is that the off-critical-path circuit takes two cycles per *Reference Count Queue* slot where the cache would take one slot to respond to a hit (read or write).

- For every instruction that affects the traditional cache, we assume an average of 2.5 instructions that do not involve the cache. The mean of our normal distribution is set at 2.5, truncated with a min of 0 and a max of 5.

Most benchmarks operate near the best-case results presented in Section 3.3.2. While `pmd` suffered the most in terms of finding dead storage, its recycling rate is still robust at 96% of our best results.

3.5 Hardware Implementations

In order to evaluate timing requirements of our hardware, we implemented CORC in *VHDL* and targeted synthesis for the *Virtex-7 (XC7VX485T-2FFG1761C) FPGA*. The *Vivado Design Suite* default settings were used for synthesis. A simple cache model was also implemented in *VHDL* was targeted towards the same *FPGA* for comparison. We make the following realistic assumptions to simplify the hardware design:

- Allocations are aligned with word boundaries.

- Fields that hold a reference are aligned with word boundaries.

3.5.1 Full Implementation

We first implemented CORC in full as described in Chapter 2 with added storage and logic to maintain reference counts for each line in the cache.

Storage Requirements

To evaluate storage costs of CORC we consider the L1 cache used for experimentation in Section 3.3.2 with 64 bit virtual addresses and 32 bit physical addresses. We consider a cache word size of 8 bytes. Based on Java, the minimum allocation size of an object is 16 bytes. From this we see each cache line may contain bytes from 3 unique objects. Storage for 3 objects is therefore required per cache line.

Based on CORC details in Section 2.2, storage for each object consists of 102 bits. An additional 8 bits of storage is needed to maintain information about the line itself. Therefore, each cache line requires an additional 314 bits, or approximately 40 bytes of storage. The L1 cache evaluated has 1024 cache lines, leading to an additional 40 KB of storage to maintain reference count meta data. This essentially doubles the chip space required by the cache. As well as storage per cache line, logic is needed per cache line to maintain accurate reference counts as well.

Timing Requirements

As shown in Figure 3.15, under realistic conditions our implementation needs to run at 0.5 relative speed to that of the standard cache to remain effective. Because each line executes

reference counting instructions in parallel, we believed the speed would be acceptable. However, the full implementation is unable to meet these timing requirements as it is scaled up to standard cache sizes. This is due to several reasons:

- Each cache line must contain logic to maintain the objects contained in it. When scaling up to level 1 cache sizes, such as the cache used for experimentation, there are 1024 copies of logic. Although the logic to maintain reference counts is simple, it becomes an issue when duplicated a large number of times.
- Reference counting instructions do not act on addresses, but object ID's or a frame as in the `framepop` instruction. Object IDs and object frames must therefore be searched in parallel which is expensive. As the number of cache lines increases this becomes an issue as fully-associative search is expensive in power and chip space.
- Object headers are stored per cache line, so the same object may have header information duplicated multiple times. This duplication of data requires additional chip space increasing signal delay.

These issues increase the amount of chip space needed to implement the full approach. Targeting an *FPGA* for synthesis, this leads to large wire delay as the data must travel longer distances as the design increases in size.

From these results we see our approach must be limited to remain scalable, as relative speed for reference counting instructions as caches increase to 32 KB in size (1024 lines) is well below the 0.5 necessary to remain effective. In the remaining sections, limitations are explored.

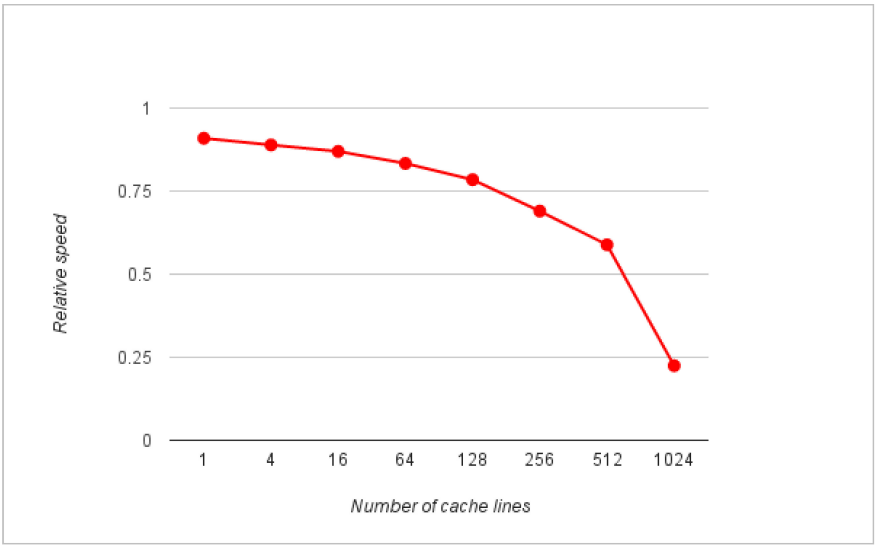

Figure 3.16: Comparison of the speed of our approach compared to the baseline cache speed as the number of cache lines increases. Here, all cache lines have 32 bytes.

3.5.2 Limited Object Size

In Figure 3.4 we saw that most object allocation requests are very small. In fact, more than 75% of all allocation requests are 96 bytes or less. Consider the full implementation with 1024 cache lines (32 bytes each) as described above. 75% of the objects span 4 lines or less. Now consider a reference count increment comes in for one of these small objects, a maximum of 4 lines will respond with work to do. This leaves 1020 copies of reference count logic unused. In this section we limit CORC to maintain reference counts for objects under a certain size threshold s. This gives us a major advantage:

- Most reference count instructions act on only 1 object. By limiting the object size, we also limit the number of cache lines an object can span. This means, for most instructions, a bounded number of lines need to perform updates per instruction. The number of copies of reference counting logic needed to handle most instructions can

then be limited as well. In the previous example, instead of being left underutilized, those 1020 copies of unused logic can be removed from the design.

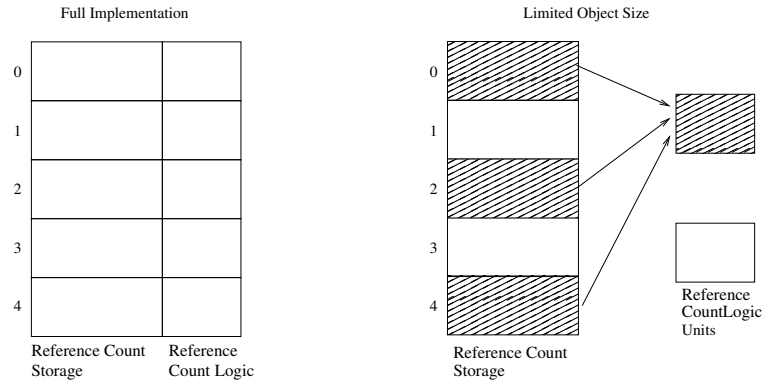

Figure 3.17: A diagram of the CORC when object sizes are limited versus that of the full CORC implementation where each cache line has its own logic.

Figure 3.17 illustrates how the logic units are mapped to each line of storage. Let l be the maximum number of lines an object can span after object sizes are limited to s bytes. Objects are allocated in a contiguous address range, so an object spanning l lines will span consecutive cache lines. Therefore, we can assign the l logic units in rotating order and guarantee that any reference count instruction acting on a single object can have at most 1 line respond for each logic unit. In Figure 3.17 we set $l = 2$. We see that any object can span at most 1 solid line and 1 patterned line, and is covered completely by only 2 logic units.

There are however some disadvantages to limiting the copies of logic. The `framepop` instruction for example may affect more than 1 object. In the full implementation, with logic per cache line, all responding cache lines could be handled in parallel. In this limited approach however, the `framepop` instruction must be handled incrementally. Also, storage for an object's meta data is still duplicated across all cache lines it spans, leading to large storage costs for this implementation as well as the full implementation.

The results from simulation are shown in Figure 3.18, using 96 bytes as the object size threshold and 4 copies of logic. We see on average 83% of our best recycling obtained on the cache's critical path (the 32K cache results shown in Figure 3.2).

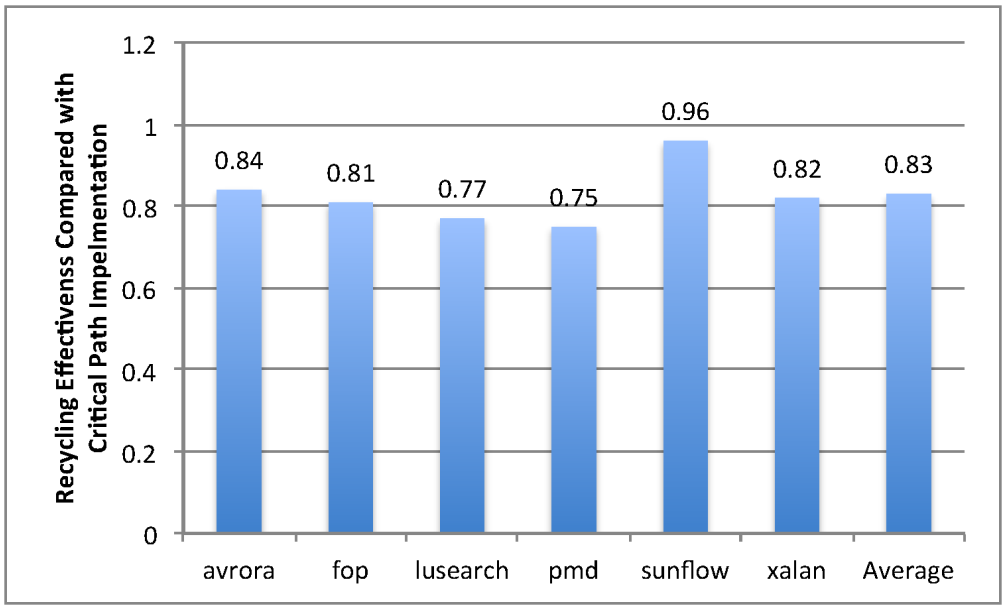

Figure 3.18: Results for CORC with limited object size. Four copies of our circuit are deployed to handle all cache lines of a 32 KB cache. Object size is limited to 96 bytes.

Although more scalable, this approach remains unable to meet speed requirements of 0.5 the standard cache's speed. We see that storage cost is the limiting issue for increased speed, as this limitation does not reduce the 40 KBs of extra storage required for object meta data.

3.5.3 Limited Number of Objects

Storage needs of CORC can be reduced by maintaining only 1 copy of each objects' meta data in a table, in contrast to 1 copy per cache line spanned as done previously. However, the logic required to maintain consistency between the object's meta data stored in the table and the

object's actual data stored in the cache becomes more difficult. Because of this, we propose creating a new generation of objects to reference count that is decoupled from the cache. The proposed subset of objects includes only the n *most recently allocated* (*MRA*) objects, where n is some constant. As the subset of objects reference counted by CORC includes all objects that remain in cache since allocation, it is likely that many of those objects would also be in the proposed subset of *MRA* objects. However, storage and logic to maintain the *MRA* objects' meta data is much smaller, as the meta data can simply be stored into a table working in first in/first out (*FIFO*) fashion with only 1 entry per object. Also, as most reference counting instructions act on only 1 object, only 1 copy of logic is needed to maintain accurate reference counts. As in the limited object size approach, `framepop` instructions may be handled incrementally. This approach can greatly reduce storage and logic costs.

However, decoupling reference counting from the cache has several issues:

- Objects are unbounded by size (and may contain many references to other objects), how do we limit the storage for maintaining references stored into the object to other objects?
- Information about the cache line, particularly which bytes in it are dead, is lost with this approach. Therefore, only lines that are spanned entirely by 1 dead object can be cleaned. This will likely reduce the amount of eliminated write bandwidth. Can we maintain cache line information somehow?
- As cache line information is lost, objects reference counted are not guaranteed to be in cache. How does this affect cache hit rates when recycling?

The proposed table of *MRA* objects is the focus of Chapter 4. Our approach is described in detail and the above issues and questions are addressed.

3.6 Summary

This chapter studies the details and benefits of recycling objects found dead via CORC to satisfy future allocation requests. It is shown that for a small 32 KB cache, an average of 27% of all allocation requests can be satisfied by recycling dead objects in cache. This accounts for 12% of the total bytes requested for allocation, leading to reduced heap pressure and potentially allowing the program to execute longer before garbage collection is triggered. When the cache size is increased to 512 KB, the number of allocation requests satisfied through recycling is increased to 45% on average. As caches are designed to be as fast as possible, we evaluate the success of CORC when moved off the critical execution path. We see that for realistic workloads, speed of the proposed reference counting instructions must remain at least $0.5X$ that off the cache. Several hardware implementations are considered. The full implementation of CORC is shown to be unscalable to L1 cache sizes, as chip space consumption is large creating long wire delays and high latency costs.

Tradeoffs are studied to reduce CORC's hardware costs, but also limiting CORC's coverage. Limiting the size of objects that are reference counted remains effective, as an average of 83% of allocation requests satisfied by CORC under ideal circumstances are also satisfied with this limitation. By limiting object size, the copies of logic needed to maintain reference counting meta data is greatly reduced and the design is more scalable than the full implementation. However, due to large requirements for storage (40 additional bytes per cache line for the

cache evaluated) and the required fully-associative search over each cache line, this approach is also unable to meet speed requirements to remain effective.

Limiting the total number of objects reference counted to only the most recent allocations is proposed. Storage for the *most recently allocated* objects' meta data is decoupled from the cache lines the object spans, allowing for only 1 copy of each object's meta data to be required for storage. This approach is described in detail and evaluated in the next chapter.

Chapter 4

Designing Effective Hardware: Trading Coverage for Throughput

4.1 Background and Related Work

Chapters 2 and 3 presented CORC to identify dead objects in cache and recycle those dead objects, however CORC proved difficult to implement in hardware with the timing requirements required to maintain effectiveness. We now present a much more scalable approach. In this chapter, reference count meta data is decoupled from the cache lines that objects span. We propose a hardware table containing the *most recently allocated* (*MRA* for the remainder of the thesis) objects, creating a new subset of objects to reference count. Only the *MRA* objects are reference counted, as opposed to all objects in the cache as in CORC. We call this new hardware table the *Infant Object Table* (or *IOT* from here out), as it contains only the youngest objects. The *IOT* allows a much simpler hardware design, as meta data for each object in the *MRA* subset is stored in only 1 *IOT* entry, except as described in Section 4.2.5. As with CORC, any cache lines known to be entirely dead can be *cleaned* preventing useless write-backs of dead data. The *IOT* also maintains information about the

liveness of an object, allowing for recycling without the use of free-lists. For an overview of related work, refer to the previous chapters in Sections 2.1 and 3.1.

Details of the *IOT* and the additional hardware structures required for executing off the critical path are described in the next section.

4.2 The Infant Object Table

In order to reference count the subset of *MRA* allocated objects, the *IOT* is proposed to maintain object meta data in *first in/first out* (*FIFO*) fashion. Upon allocation of a new object, the table entry associated with the least recently allocated object in the *IOT* is evicted to make room for the new entry. As in CORC, each table entry maintains meta data about its corresponding object. Although similar, there are several differences from the meta data maintained per cache line in CORC. This meta data is listed below. If no explanation is given, the meta data is the same as in CORC.

- An object ID (the object's virtual address)
- A valid bit - Set if the table entry is valid
- A dead bit - Set if the object is dead, enables recycling without free-lists as opposed to CORC
- A start address (the object's physical address) - Objects are no longer tied to cache lines as in CORC, so the full object interval is stored instead of only the interval covering a single cache line. The full object interval is maintained as a start address and length
- The length of the object - Not required by CORC

- The object's reference count
- A frame ID
- A thread ID
- Storage for any references stored into the object. This includes the reference being stored, the location of the reference, and a valid bit. This is opposed to a bit vector marking where in a cache line a reference is stored as in CORC

Each object's meta data is stored in 1 *IOT* entry, unless it spans multiple pages. This is described in section 4.2.5. Accurate reference counts are maintained for an object while its meta data remains in the *IOT*. After being replaced in the table, an object is no longer reference counted and is left to be collected by the software garbage collector.

4.2.1 Maintaining Reference Counts

The additional instructions needed to be added to the *ISA* for this approach are the same as those for CORC as listed in Figure 2.3. Unless specifically addressed below, instructions are handled in the same fashion as in CORC.

Allocation

The *IOT* works in similar fashion to a *FIFO* queue. It maintains a pointer to the least recently allocated (*lra*) entry in the table. Upon receiving an `allocate`, the *lra* entry is replaced by meta data for the newly allocated object. The pointer to the *lra* entry is then incremented. However, an `allocate` instruction may have been satisfied by recycling a dead

object already in the table. In this case, the table is searched for the recycled entry and it is replaced with the newly allocated object's meta data. The *lra* entry is not incremented in this case.

Stack Activity

As in CORC, the stack optimization used in [16] is also used to minimize reference count updates due to the stack. `Refreturn` and `refload` instructions are used to maintain the lowest stack frame associated with an object, as in CORC. However, we now describe an additional hardware structure required to allow for the use of this optimization. In order to maintain thread and stack frame information, a second hardware structure is introduced called the *Frame Table*. The *Frame Table* monitors thread creation and stores an entry for each active thread. Each thread entry has an associated stack frame counter that is incremented/decremented when a stack frame is pushed/popped via the `framepush` or `framepop` instructions. The *Frame Table* is queried to get the stack frame ID currently associated with a given thread, as required by the `returnref`, `refload`, `framepop`, and `allocate` instructions. Although not previously discussed, the *Frame Table* would also be necessary for CORC.

In contrast to the full implementation of CORC, `framepop` instructions must be handled incrementally by the *IOT*. Objects associated with the popped stack frame are handled one at a time rather than in parallel. This allows the *IOT* to have only one logic unit for the full table.

Reference Store

As in CORC, a `refstore` instruction consists of 3 microinstructions:

- The newly referenced object has its reference count incremented
- If a reference to an object was overwritten, that object must have its reference count decremented
- The location of the reference store must be marked, to facilitate *cascading decrements*

A major advantage of the *IOT* is seen here. Instead of storing only the location of a reference, the object ID of the referenced object at that location is also stored in the entry. Recall that CORC required the cache to be modified to read and announce the reference being overwritten before storing the new reference. This cache modification is no longer necessary, as the object ID to be decremented is stored directly by the *IOT*.

However, an *IOT* entry may represent a large object. That object may contain many references to other objects. Consider an entry storing meta data for an array of strings. Each location in the array contains a reference to a string object. Storage for references contained in an *IOT* must be bounded in some way. This is discussed and evaluated in Section 4.3.3.

Object Death

Upon object death, the *IOT* issues *clean* instructions to the cache for any lines spanned entirely by the dead object. For recycling, simply *cleaning* a cache line is preferred to *invalidating* the line as noted in [59] and described in Section 3.2. *Clean* instructions may

only be issued for lines spanned entirely by 1 object because the *IOT* does not maintain cache line information as in CORC. For a cache line partially spanned by a newly discovered dead object, it is unknown whether the remaining bytes in the line are dead. Preventing its write-back can cause important data to be lost. However, generational garbage collectors are typically paired with contiguous allocators. This creates similar locality between objects in memory and their *IOT* entries that can be exploited to further reduce writes. As consecutive object allocations span consecutive memory addresses, the *IOT* entries before and after the dead object's entry correspond to the objects located before and after it in memory. Therefore, if an object spans a partial line, the entries before and after it may be consulted to see if the remaining bytes in the line are dead. The locality of objects in the heap and in the *IOT* is evaluated in Section 4.3.4. To support execution off the critical path of the processor, *clean* instructions issued by the *IOT* are queued in the a new structure, the *Clean Queue*. As the cache is available, it can service *clean* instructions from the *Clean Queue*.

The *Clean Queue* is similar to the cache's queue in CORC, however the *Clean Queue* does not need to hold *cascading decrements*. As each *IOT* entry stores all object IDs referenced by the corresponding object, *cascading decrements* can simply be issued directly by the *IOT* upon object death. By contrast, CORC required a *read* of any contained reference locations to be queued. The cache could then read the object ID to be decremented and issue the decrement as it is available.

To summarize, by storing the object ID of a referenced object into the *IOT* rather than storing only the reference location (as in CORC) no additional modifications are needed to the standard cache beyond a *clean* instruction.

4.2.2 Off the Critical Path

The proposed reference counting instructions can be more complex than standard non-memory or memory instructions. We therefore avoid stalling the standard processor's execution by creating a *Reference Counting Queue* (*RC Queue* from here out). Reference counting instructions are then queued and executed as the *IOT* is ready. This allows reference counting instructions to execute slower, using more cycles, than non reference counting instructions without stalling the pipeline.

The reference counting instructions may be injected directly into the standard instruction stream. *Frame Table* lookup can be done in parallel with virtual memory translation if needed. Following translation and *Frame Table* lookup, the instruction can then be added to the *RC Queue*. However, adding reference counting instructions into the processor's pipeline increases the total instruction count for each program by an average of 21.7% using our methodology described in Section 3.4.2 for constructing a realistic workload. It is more acceptable to move reference counting off the critical path entirely. This requires a custom page table for the reference counting hardware, however this table can be very small. The objects represented by *IOT* entries are contiguous in virtual memory due to the contiguous allocation scheme. These objects are often small, so all objects in the *IOT* span only a few pages. For instance, we saw in Section 3.3.2 that over 75% of the total allocation requests in Java are for 96 bytes or less. Consider a 32 entry *IOT* with an average size of 96 bytes per object represented in the *IOT*. The objects represented in the table span a total of $3,072$ bytes. With 1 KB pages, translation for the entire *IOT* can be done with only a 4 entry page table. Any instructions that cause a miss in the *Reference Counting Page Table* (*RCPT*) can be dropped, as they will not match any object in the *IOT* either. In this configuration, we queue reference counting instructions immediately, postponing address translation and

Frame Table lookup. This allows the processor to filter reference counting instructions out of the instruction stream as quickly as possible. The configuration described is shown in Figure 4.1, note instructions are only processed through the grey arrow as the cache is free from standard memory instructions. In Section 4.3.6, we evaluate how much slower the *IOT* can empty the *RC Queue* relative to the throughput of the standard cache and maintain reasonable effectiveness.

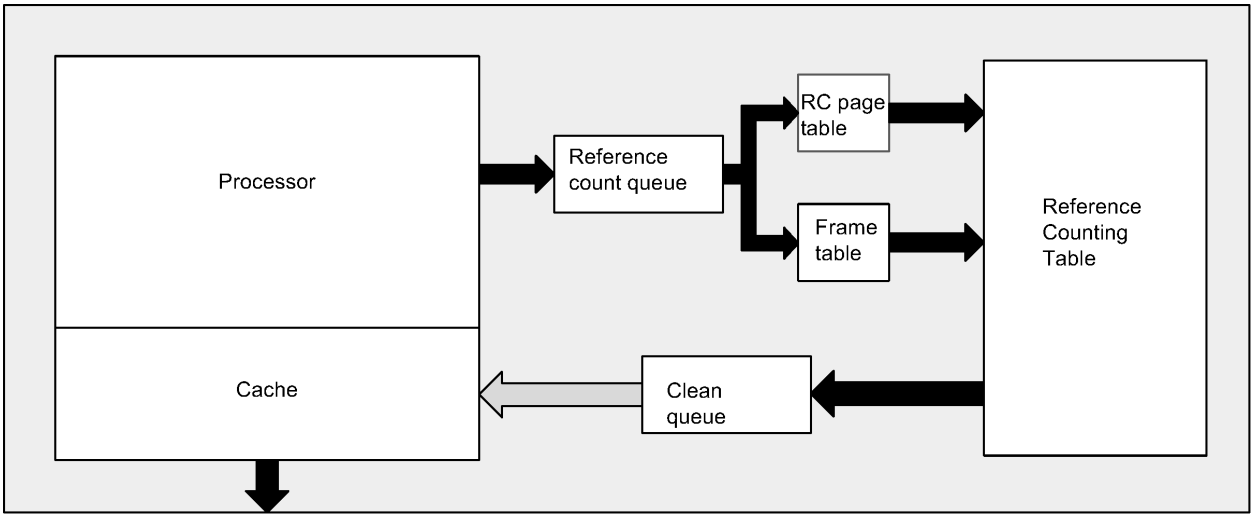

Figure 4.1: System configuration with the added reference counting hardware.

4.2.3 Reference Count Bits

We use reference counts that are *sticky* [5], meaning once an object's reference count reaches its max value it can no longer be decremented and the object cannot be identified as dead. This ensures a live object will not be mistaken as dead. Prior work shows that many objects are referenced very few times before death, in fact we saw CORC shows very little improvement on eliminating writes when using a reference count of larger than 2 bits. However, the reference count of an object is minimal in size when compared with the rest of the *IOT* entry.

We therefore use 3 bits to store reference counts, as CORC did see a slight improvement in object coverage when increasing to 3 reference count bits in Figure 2.9.

4.2.4 Recycling

Upon object death, the dead bit is set in the object's *IOT* entry. When an allocate is issued, an *allocation request* can be inserted via an allocate barrier. The *IOT* is then scanned for an appropriately sized object that is dead. If found, the object ID is returned to the allocator for reuse. If no appropriately sized dead object is available, *null* is returned and the allocator must allocate a new address in its usual manner. Because the *Clean Queue* is emptied as the cache has time, *clean* instructions targeting the newly reallocated address may remain in the queue. This can cause memory consistency to be broken if the newly allocated object is written and then subsequently cleaned. Therefore, upon reallocation of a dead object, any *cleans* remaining in the queue to that address space must be removed. The benefit of a *clean* instruction versus that of an *invalidate* instruction is seen when an address is recycled. Upon allocation, Java writes *zeroes* to the newly allocated object. If the line were simply invalidated, these writes would cause cache misses as the invalidated line must be fetched again from memory. However, by marking the cache line clean, it remains valid and these writes will result in cache hits. Therefore, allocation requests satisfied by recycling dead space in cache can essentially be allocated directly into the cache without needing to be fetched from memory. The effect of this on cache miss rates is evaluated in Section 4.3.5. In contrast to the free-lists required by CORC for recycling storage, the *IOT* requires only an additional dead bit for recycling.

4.2.5 Objects Spanning Multiple Pages

An object spanning multiple pages of memory may be reference counted as usual, however it will also require multiple *IOT* entries. The object will have an entry for each physical address interval it covers (it will have a separate interval for each page it spans). Therefore, if an object spans two pages it will require two *IOT* entries. During address translation, the *RCPT* separates any allocation spanning multiple pages into multiple `allocate` instructions to facilitate this. Reference counting instructions acting on objects spanning multiple *IOT* entries are handled incrementally. This allows us to eliminate write-backs of dead objects spanning multiple pages, as well as recycle those objects.

4.2.6 Handling Garbage Collection Cycles and Context Switches

During garbage collection cycles, objects and references may be moved around the heap. For simplicity and ease of implementation we choose not to track object movements. We also expect the cache to be emptied of application data during garbage collection.

Therefore, we simply pause execution of reference counting during garbage collection and start fresh when garbage collection is complete. As garbage collection is likely to cause the eviction of everything currently in cache, we flush the cache on the onset of a garbage collection cycle in our simulations. Write-backs issued during a collection cycle are not counted in our statistics. Due to the potential of objects and references being moved, we flush the *IOT* and both the *RC and Cache Queues* after garbage collection and start fresh.

Although not simulated in our results, context switches are handled in a similar fashion. The *IOT* and both queues are flushed upon a context switch, starting from scratch after the context switch is complete.

4.3 Evaluation

4.3.1 Experimental Setup

In order to evaluate our approach we use a combination of simulation and hardware timing validation (discussed in section 4.3.7). For simulation, we implemented all functionality of the proposed hardware structures into the trace-based cache simulator used for experimentation in Chapter 3. To facilitate comparison, traces for the six *DaCapo Benchmarks* [13] studied in previous chapters were also used. Because we do not trace any non-memory instructions that would be executed by the processor, we inject non-memory instructions into our instruction stream as in Section 3.4.2. This gives us a more realistic representation of the full workload of each benchmark. In all of these experiments, the full trace was run. We assume the standard cache executes as a non-blocking cache [8] (executes at its highest possible throughput).

We evaluate our approach on a small cache representing an L1 cache. The cache specifications are as in Table 4.1.

Size	32 KB
Type	Write-back
Associativity	2
Line size	32 bytes
Access hit time	2 cycles
Write-back granularity	Full line

Table 4.1: Cache specifications for experimentation and timing analysis

Evaluation for larger caches is left for future work, which is further discussed in Section 5.1.

We use 1024 byte pages for virtual memory, using 64 bit virtual addresses and 32 bit physical addresses.

We first vary different *IOT* parameters to determine suitable values for them. In these experiments we measure the fraction of eliminated write-backs. In other words, the fraction of writes that would have reached memory without the *IOT* issuing *clean* instructions.

4.3.2 Infant Object Table Size

We first look at how the number of entries affects the ability of our hardware to identify dead objects. The remaining *IOT* parameters are fixed with the following values:

- 1 reference stored per entry (chosen through experiments in section 4.3.3)
- 3 reference count bits per entry (chosen as described in section 4.2.3)
- Throughput of 2/3 that of the cache (chosen through experimentation and hardware timing validation in sections 4.3.7 and 4.3.6)

Figure 4.2 shows that as the table size increases, the fraction of eliminated writes increases as expected. However, as the table size increases above 32 entries, the added benefit diminishes. The 32 entry table eliminates 17.5% of the total writes out of the cache on average. That number only jumps 1.7% to 19.2% for a 128 entry table. This suggest a 32 entry table is suitable for hardware implementation, as the diminishing returns of larger tables will not make up for the added chip space used.

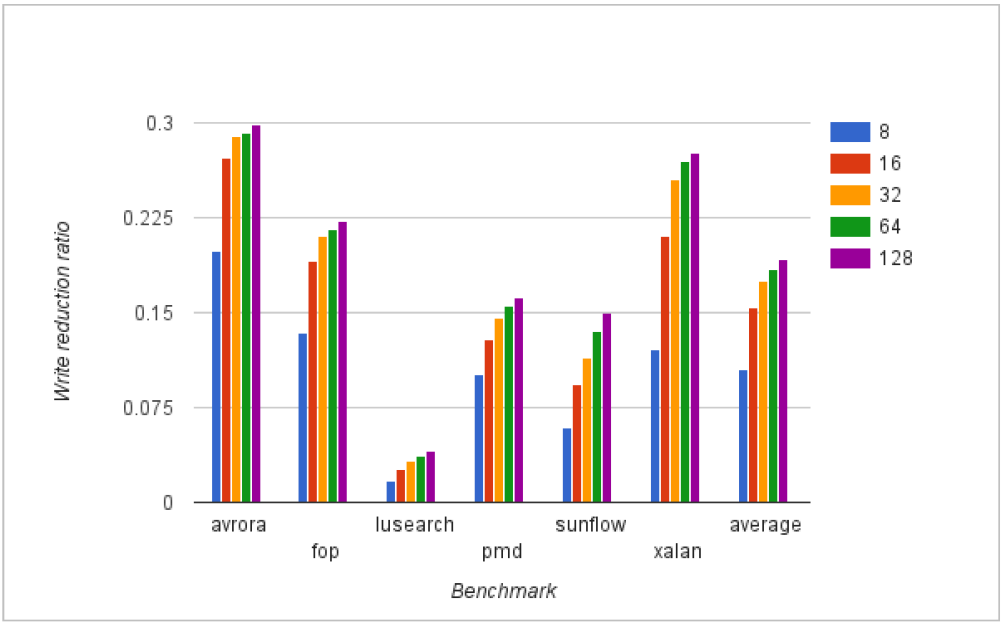

Figure 4.2: Fraction of writes eliminated for varying *IOT* sizes.

4.3.3 Number of References Stored per *IOT* Entry

As described in Section 4.2.1, each *IOT* entry contains storage to maintain the references stored into that object. We now examine how the number of references, r, stored per *IOT* entry affects the writes eliminated. This is implemented (in simulation and hardware) by storing only the first r unique reference locations in an object, any further reference locations are lost. We stick with the table parameters from the previous experiment, however we set the table size to 32 entries (via experimentation in section 4.3.2) and vary the number of references stored per entry.

It is clear from Figure 4.3 that there is little benefit to storing more than one reference per entry. In fact, many writes can be eliminated without even accounting for objects referencing other objects (14% on average). That number increases to 17.5% when each table entry is given storage for 1 contained reference, however there is only a 0.4% increase

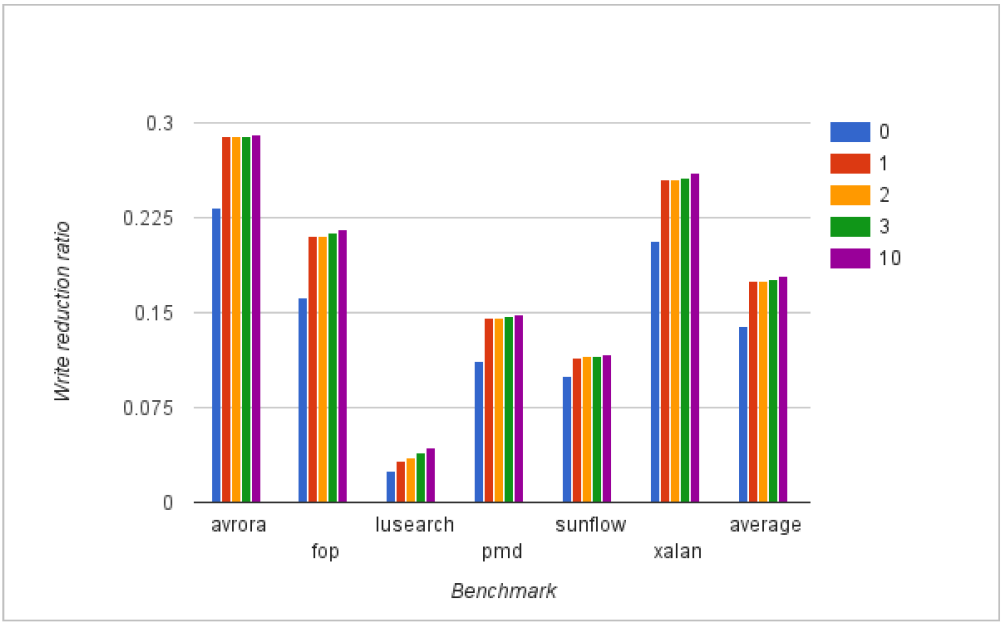

Figure 4.3: Fraction of writes eliminated for a 32 entry *IOT* with varying number of references per entry.

in write reduction when increasing the number of stored references per entry to 10. For all other experiments, and for hardware evaluation, we limit the references per table entry to 1. As each reference requires storage for the referenced object ID and the location of the reference, this leads to significant storage savings for the proposed hardware.

4.3.4 Exploiting Contiguous Allocation

As mentioned in Section 4.2.1, consecutive allocations span contiguous memory, as contiguous allocation is generally used with generational garbage collection. Therefore, unless a page boundary is crossed, consecutive *IOT* entries represent consecutive objects in memory. We can exploit this locality between objects in memory and their *IOT* entries to further reduce writes. If an object dies, but does not span the entire cache line, the entries before

and after it can be checked to determine if the remaining bytes in the cache line are dead. If so, a *clean* instruction can be issued for the fully dead line. We now evaluate the benefits of exploiting the locality between objects in memory and their *IOT* entries. *IOT* size is fixed at 32 entries and 1 reference is stored per entry.

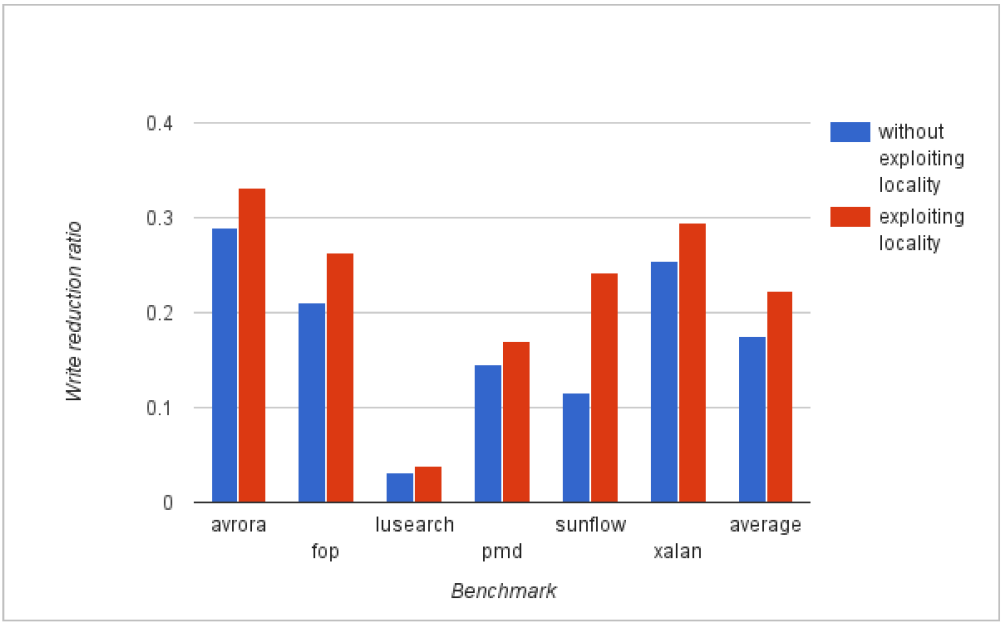

Figure 4.4: Fraction of writes eliminated when exploiting locality between object's *IOT* entries to identify fully dead lines spanned by more than 1 object.

Figure 4.4 shows the write reduction for each benchmark with and without exploiting contiguous allocation. The figure shows that the locality of *IOT* entries and the memory space they represent helps to further reduce writes out of the cache. Write reduction jumps to 22.4% from 17.5% when consulting entries before and after the dead object to *clean* as many cache lines as possible.

4.3.5 Recycling

Now that we have determined suitable IOT parameters, we study its effectiveness at recycling dead objects to satisfy allocation requests as described in section 3.2. These experiments are run using the table parameters decided through experimentation, as listed in Table 4.2.

Table size	32 entries
References per entry	1
Reference count bits	3
Relative throughput vs. the cache	2/3

Table 4.2: IOT parameters for experimentation

We look at two metrics when studying recycling. First, we measure the total number of allocation requests satisfiable by recycling as a fraction of all allocation requests made. Second, we measure the total number of bytes requested for allocation that can be satisfied by recycling as a fraction of the total number of bytes requested for allocation. This gives us an idea of how much heap pressure can be eliminated by recycling.

Although spatial locality can be exploited to further reduce writes, it will not benefit recycling. For simplicity, we do not merge smaller objects to satisfy an allocation request for a larger size, even if those objects are contiguous in memory. For these experiments, only an exact fit is considered for recycling. This means an allocation request for 48 bytes can only be satisfied by recycling a dead object of size exactly 48. Future work can address merging contiguous objects to satisfy a larger allocation request, as well as relaxing the exact fit constraint. This is discussed in Section 5.1.

Figure 4.5 shows an average of over 28% of allocation requests can be satisfied by recycling previously dead objects. On average, 12% of the total number of bytes requested for allocation are recycled from previously dead objects. This translates into a 12% reduction in heap

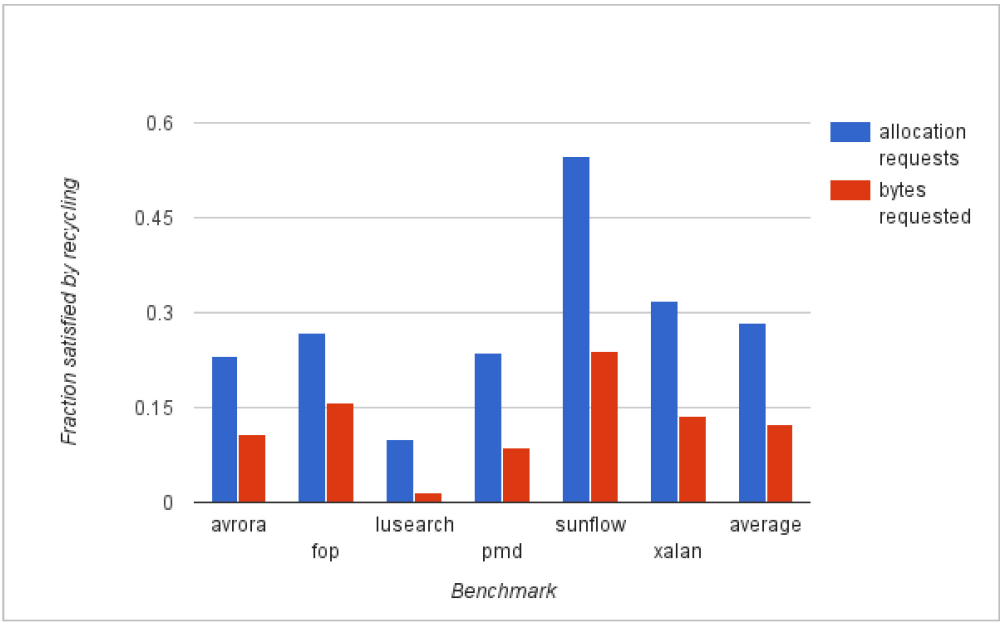

Figure 4.5: The fraction of allocation requests satisfiable by recycling dead objects. The figure shows the fraction of object requests satisfied as well as the fraction of bytes requested that are recycled.

pressure, allowing for smaller heaps for object-oriented applications or allowing the application to execute longer between garbage collection cycles. Most objects that die young tend to be small in size, so the allocation requests satisfiable by recycling are often small as well. This explains the difference in recycling percentage between requests and bytes requested, as large allocation requests are likely not to be satisfied via recycling dead space (accounting for only 1 unsatisfied request, but a large number of unsatisfied bytes).

Not all dead objects are recycled before eviction from the *IOT*, however the cache lines they occupied are still *cleaned* to reduce writes.

In Figure 4.6 we see the write reduction gains made by exploiting locality are almost completely nullified when recycling, leading us to avoid implementing this functionality in our hardware implementation. 20.3% of writes are eliminated without exploiting spatial locality

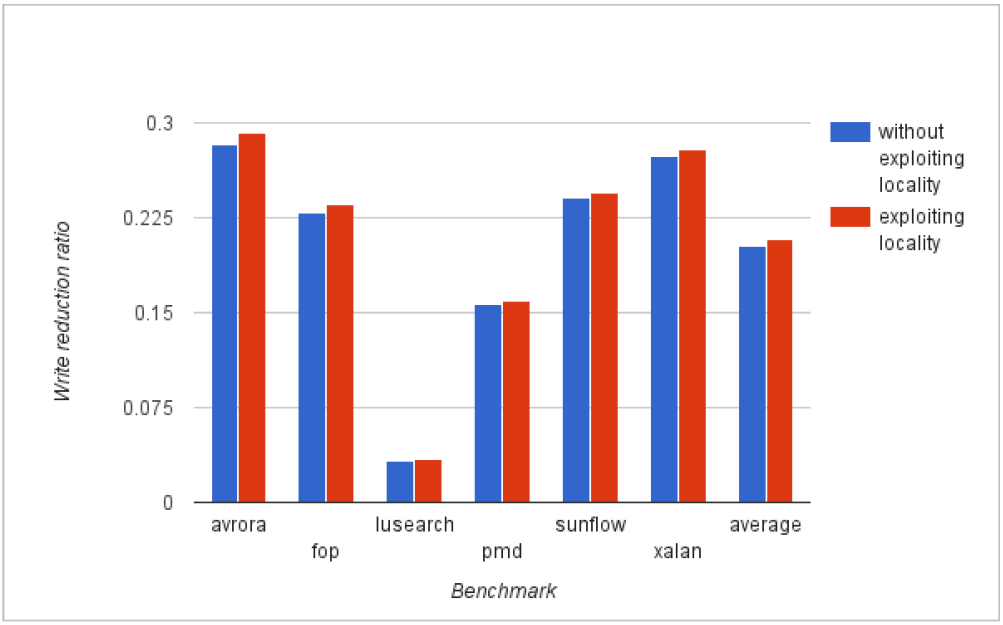

Figure 4.6: Fraction of writes eliminated while also recycling dead objects to satisfy allocation requests.

for write reduction. Exploiting locality leads to only a 0.4% increase in write reduction. This, along with the jump in write reduction from 17.5% to 20.3% when recycling is introduced (without exploiting locality in the table) can be explained by the following example. Consider a 32 byte object spanning 2 cache lines, 16 bytes in each, dies. Those cache lines cannot be *cleaned* because we cannot be sure they are entirely dead. However, the object can be recycled. Recycling the object prevents a new address range of 32 bytes from being allocated, and likely fetched into cache for initialization. This prevents any cache lines from being evicted from cache and potentially written back. The effect recycling has on cache misses due to allocating directly into the cache, avoiding the need to fetch newly allocated objects from memory, is now studied.

Effect on Cache Hit Rates

As write-backs are performed off the critical path, we expect no performance benefit from reducing writes out of the cache. However, satisfying an allocation request by recycling a dead object in cache can avoid cache misses as describe in Section 4.2.4. To analyze this, we study cache hit rates on allocation with and without recycling. We classify an allocation access as the cache access during the *zeroing* of an object after allocation.

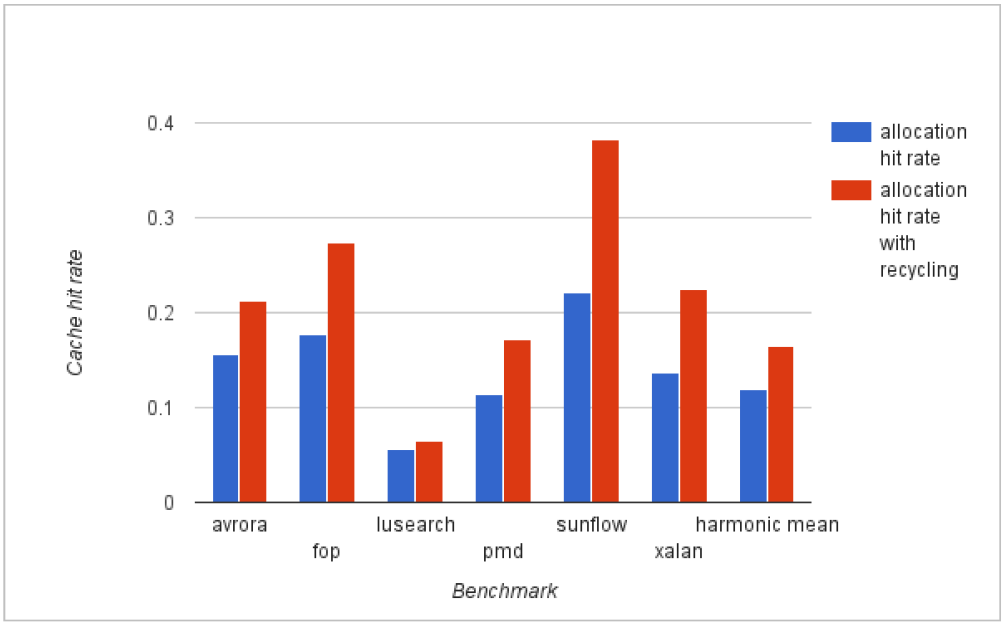

Figure 4.7: The cache hit rates on allocation accesses, with and without recycling of dead objects.

Figure 4.7 shows allocation hit rates for each benchmark as well as the harmonic mean over all benchmarks. We see similar improvement in hit rates when using the *IOT* as with CORC as shown in Section 3.3.2. This shows that although the *IOT* is decoupled from the cache, it is likely that the objects recycled by the *IOT* are resident in cache upon recycling. We see nice improvement in allocation hit rate over all benchmarks. However, allocation accesses account for only a small portion of all cache accesses.

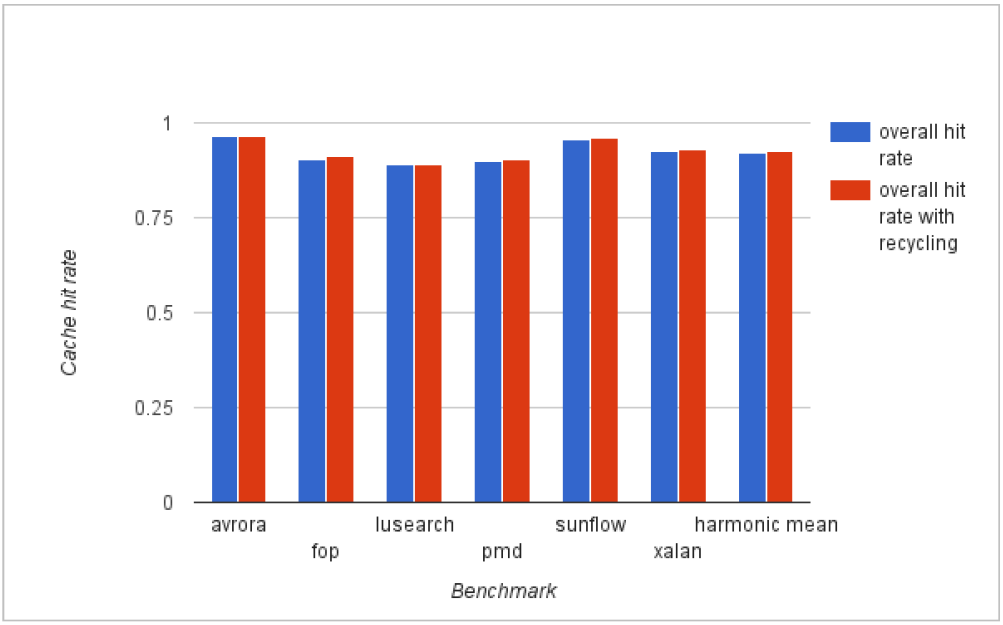

Figure 4.8: The overall cache hit rates with and without recycling.

Figure 4.8 shows cache hit rates for all cache accesses. As we see, cache hit rates improve much less overall, on average improving from 0.923 to 0.927.

4.3.6 Throughput Requirements

As described in section 4.2.2, the proposed reference counting instructions can be more complex than standard instructions. We queue reference count instructions in the *RC Queue* and allow them to execute as the *IOT* is ready. In these experiments we empty the *RC Queue* at varying rates relative to the throughput of the cache. As our approach depends on keeping up with the cache in order to find objects dead before eviction, executing too slow will cause our hardware to be ineffective. We assume the cache is non-blocking and therefore executes at its max possible throughput. We simulate this by assessing no penalty for cache misses in simulation, which gives the *IOT* no additional time to catch up on cache misses.

We study the effect of decreasing the throughput of the *IOT* relative to that of the cache by looking at the relative effectiveness at eliminating writes and recycling when compared against the *IOT's* write reduction and recycling ability in an ideal scenario. In the ideal scenario, reference counting instructions are executed without queuing and with no delay.

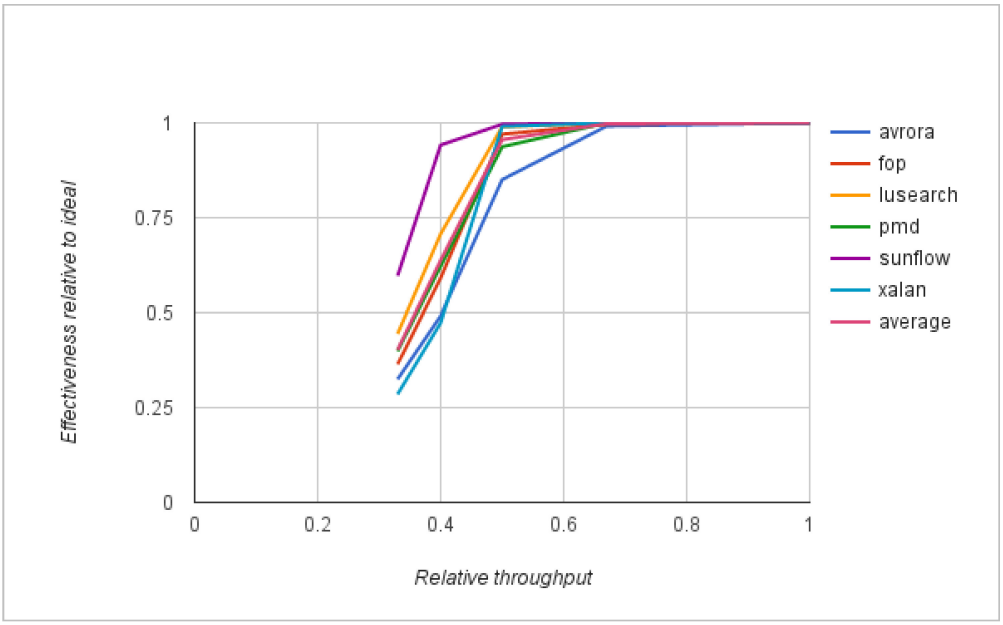

Figure 4.9: The relative effectiveness for recycling as the throughput of the *IOT* relative to that of the cache varies.

Figure 4.9 shows how the *IOT's* ability to recycle objects changes as its throughput varies. Figure 4.10 reports how write reduction is affected by varying throughputs. Both figures show the *IOT's* performance drops drastically as its throughput decreases below 1/2 that of the cache. Interestingly, when throughput decreases to 1/3, the number of writes out of the cache actually increases on average. This is due to the recycling of objects that were no longer in cache. When an object is newly allocated (not recycled), some bytes may share parts of a cache line with the previous allocation as they are contiguous in memory. Upon *zeroing*, the bytes in the shared cache line will result in cache hits. Now consider a dead

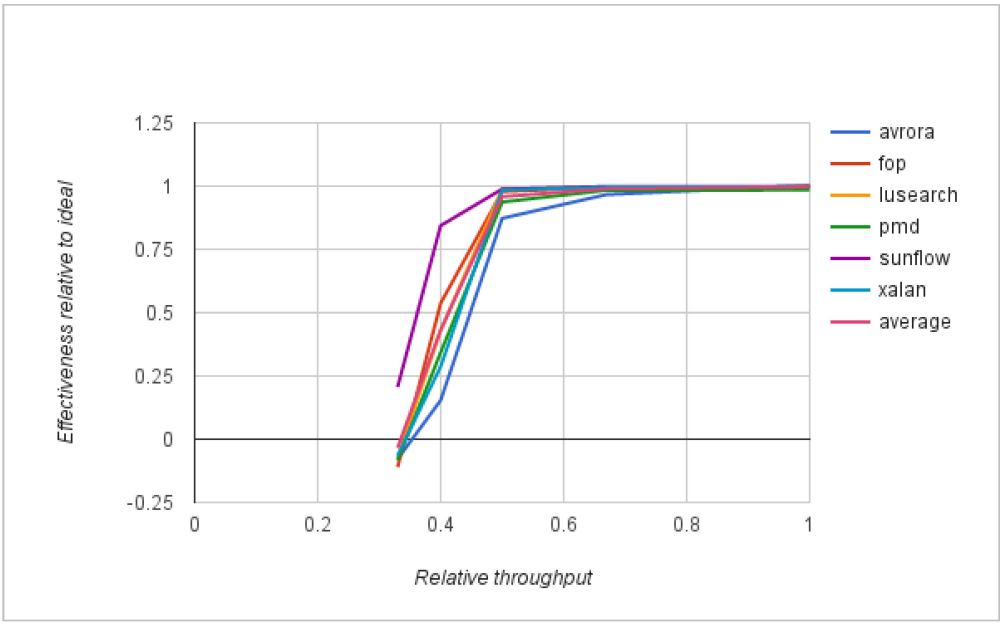

Figure 4.10: The relative effectiveness for write reduction as the throughput of the *IOT* relative to that of the cache varies.

object that is no longer in cache is recycled to satisfy the allocation request, the subsequent *zeroing* of the recycled space will result in all cache misses. This leads to more data being evicted from cache, resulting in a higher number of writes out of the cache.

Queue Sizes

Although the effectiveness of our approach only takes a slight hit as the throughput of the *IOT* drops to 1/2 that of the cache or below, lower throughput does increase the size of the queues required. The *RC Queue* is especially vulnerable to increases in size as it is emptied directly by the *IOT*. In order to study queue sizes required, we take a snapshot every $100,000$ trace instructions. In the snapshot, the maximum queue size for each queue

during that interval is reported. Figure 4.11 reports the mean maximum *RC Queue* size over all intervals.

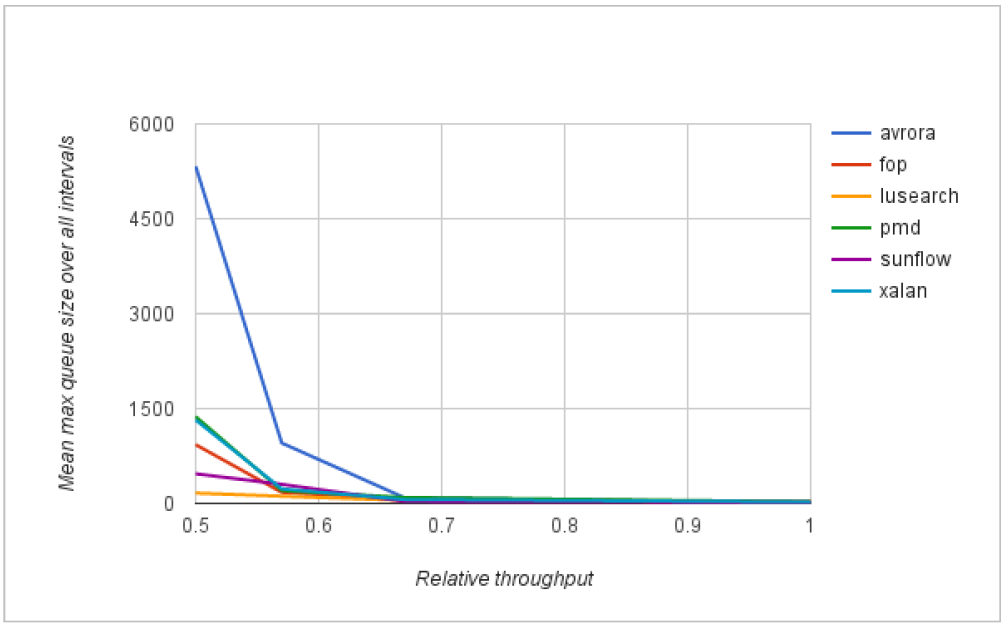

Figure 4.11: Mean of the maximum *RC Queue* size over all 100,000 instruction intervals.

As the relative throughput of the *IOT* decreases, *RC Queue* size increases as expected. Although effectiveness remains acceptable with relative throughput of 1/2, the *RC Queue* sizes begin to increase drastically (mean of over 5300 for avrora). However, at relative throughput of 2/3 all mean *RC Queue* sizes are under 100.

Mean queue sizes for the *Clean Queue* remain small (< 90 entries for all benchmarks and throughputs) as it relies on the cache being available and not the *IOT*.

There is likely not room on chip for large queues. It is also likely the queues will fill up at some point, as the maximum queue size seen over the entire benchmark can be much higher than the mean maximum queue size over each interval. *Clean* instructions can simply be dropped if the *Clean Queue* is full, as there are no harmful effects for not preventing

a write-back. However, dropping instructions from the *RC Queue* can break correctness. Consider for instance, an increment to an object's reference count is dropped due to a full queue. If space then opened up in the queue and a decrement to that same object was added and executed, the object may be identified as dead when it is not. To avoid this, when the *RC Queue* becomes full, we simply drop all reference count instructions until the queue is completely empty. We can then make all *IOT* entries invalid and start fresh, as if a context switch occurred as described in section 4.2.6.

4.3.7 Hardware Validation

In order to evaluate timing requirements of our hardware, we implemented the *IOT* in *VHDL* and targeted synthesis and implementation for the *Virtex-7 (XC7VX485T-2FFG1761C) FPGA*. The *Vivado Design Suite* default settings were used for synthesis and implementation. A simple cache model also implemented in *VHDL* was targeted towards the same *FPGA* for comparison.

Although our design is flexible, we evaluate timing using *IOT* parameters based on our simulated experiments. These parameters are listed in Table 4.3. General architecture specifications are shown in Table 4.4.

Table size	32 entries
References per entry	1
Reference count bits	3
Access hit time	2 cycles
Exploit locality for write reduction	No

Table 4.3: *IOT* parameters for timing analysis

Each *IOT* entry contains the fields listed in Section 4.2. As discussed in Section 4.2.5, an object spanning multiple pages has an entry in the table for each page it spans. Therefore,

Virtual address bits	64
Physical address bits	32
Page size (bytes)	1024

Table 4.4: Architecture specifications for timing analysis

the length of any entry is limited by the physical page size. For our experiments we used 1 KB pages, so for our calculations here we use 10 bits of storage for the length of an object. We use 16 bits for the Frame ID and Thread ID, relying on the *Frame Table* to translate from longer Thread IDs down to 16 bits. Using these numbers and the specifications shown in Tables 4.3 and 4.4, each IOT entry requires 240 bits of storage or 30 bytes. With 32 entries, the IOT requires a total of 960 bytes of storage. Each queue will require storage as well. We let each queue have storage for 128 instructions. *Clean* instructions consist of only a physical address and can be stored in 4 bytes, leading to 512 additional bytes of storage for the *Clean Queue*. The largest reference count instruction is a `refstore`. It requires 2 virtual addresses (the reference being stored and the location - pre address translation), accounting for a minimum of 16 bytes per instruction in the *RC Queue*. The *RC Queue* would then require a minimum of 2 KB of storage.

The cache specifications used for comparison are the same as those used in our simulated experiments and are listed in Table 4.1.

Post-synthesis, the IOT required $1.525x$ the latency of the cache. Post-implementation, the IOT required $1.769x$ the latency of the cache. These timing numbers give us relative throughput of approximately 2/3 that of the cache post-synthesis and 4/7 post-implementation. Although these numbers allow our approach to maintain good results with reasonable queue sizes, as seen in section 4.3.6, more timing analysis is needed as *FPGAs* are not always an accurate interpretation of what timing will be when targeting an *ASIC*. Targeting our design

to an *ASIC* is left to future work. Also, we acknowledge the current implementation of our hardware would require a slower clock on chip. We chose to implement our hardware in 2 cycles for ease of implementation and to easily compare the throughput of our design with that of the cache, as throughput was most important in our experiments in Section 4.3.6. Although left to future work, we believe the use of content addressable memory to store sections of the table entry requiring fully-associative search (object IDs, Frame IDs, and physical addresses) can achieve the required clock speed.

4.3.8 Comparison with CORC

We now discuss the tradeoff of decreasing reference counting coverage for increased throughput. Any object reference counted by the *IOT* would also be referenced counted by CORC, however the object may be evicted from either object subset at different times. It is likely, but not true in all cases, that the object would be evicted from the *IOT* before it is evicted from cache. As described in Section 4.2.2, a 32 entry *IOT* containing an average object size of 96 bytes spans only 3 KB of memory space. This is in contrast to the 32 KB's covered by the cache, although the cache may contain data that is not part of the subset of reference counted objects. Therefore, it is expected that the *IOT* has less coverage than CORC. However, our results show that the *IOT* is able to achieve recycling rates and write reduction rates only slightly less than that of CORC. In order to better understand the coverage of the *IOT* versus CORC, we run each approach with the 32 KB and 512 KB caches described in Section 2.3. Each trace is run to completion without recycling, using the cache simulator as in Chapter 2 (no virtual memory). Instead of a full implementation of the *IOT*, we simply limit CORC to reference count only the n *MRA* objects. This essentially simulates an n entry *IOT* that exploits spatial locality in the table to eliminate as many writes as possible.

All simulator variable are kept constant during all experiments. Figure 4.12 shows write reduction of the *IOT* with varying sizes relative to the write reduction of CORC for the 32 KB cache. We see that a 32 entry *IOT* identifies and eliminates approximately 85% of the writes that CORC eliminates. Although the *IOT* gives up some coverage, it requires much less storage space (960 bytes versus 40 KB, not including queues or the frame table necessary for both approaches) than CORC and is therefore a much more scalable solution.

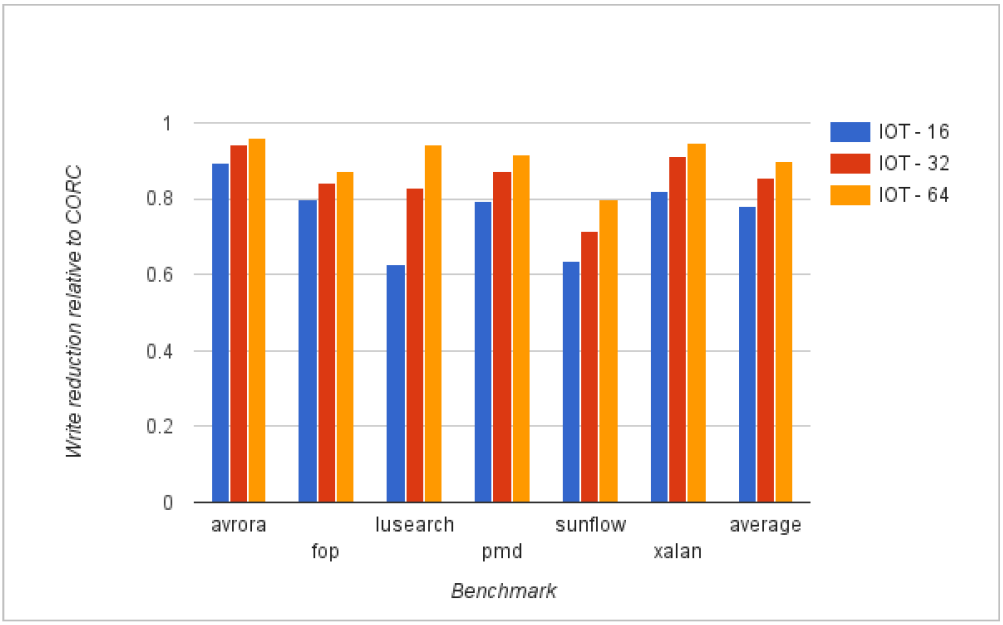

Figure 4.12: Write reduction of various sized *IOTs* relative to the write reduction of CORC for a 32 KB cache.

Figure 4.13 shows write reduction of the *IOT* with varying sizes relative to the write reduction of CORC for the 512 KB cache. To achieve similar write reduction coverage of 85% that of CORC, an *IOT* of only 128 entries is necessary. However, a thorough evaluation of the *IOT* with larger cache sizes is left for future work and is discussed in Section 5.1.

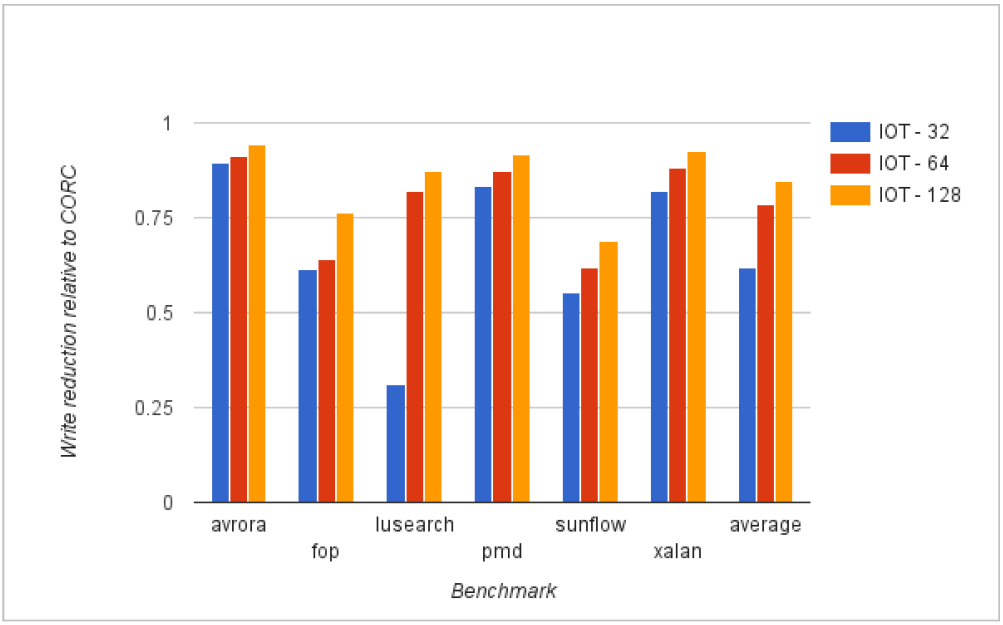

Figure 4.13: Write reduction of various sized *IOTs* relative to the write reduction of CORC for a 512 KB cache.

4.4 Software Implementation

Although the *IOT* is proposed here as a hardware implementation, we have shown that reference counting only a very limited number of the *MRA* objects can identify many objects as dead. Many of those objects are in cache upon death. Therefore, an efficient software implementation of the *IOT* paired with a *clean* instruction, can effectively eliminate many useless write-backs.

As discussed in Section 2.1.3, traditional software reference counting is particularly inefficient due to the added memory traffic of reference count updates. By maintaining reference counts for only the *MRA* objects, this memory traffic can be reduced. As contiguous allocation creates spatial locality between objects, all objects in the *IOT* are in a contiguous chunk of virtual memory. Therefore, each object in the table has a virtual address between the

least recently allocated object in the table and the most recently allocated object. The beginning and end of the address range represented in the *IOT* can easily be maintained in global variables (or dedicated registers). As part of the write-barrier required for software reference counting, a simple check can be made to ensure the object having its reference count updated resides in the address space covered by the *IOT* prior to performing the update. Any memory traffic for reference count updates of objects not in range is avoided. Thorough evaluation of this idea is left for future work and is discussed in Section 5.1.

4.5 Summary

To conclude, we have introduced and evaluated the effectiveness of novel hardware structures that can be placed alongside the standard processor to identify dead objects. Cache lines occupied by dead objects can then be *cleaned* to avoid unnecessary write-backs of dead data through the memory hierarchy. The memory space associated with previously dead objects is also reused to satisfy future allocation requests, decreasing heap pressure and potentially increasing effective execution time between garbage collection cycles. The proposed *IOT* performs reference counting on only the *MRA* objects, as many objects die young in programming languages with automatic memory management. Using realistic parameters and realizable throughput requirements, the *IOT* satisfies an average of 28% of the total number of allocation requests by recycling dead objects. On average, 12% of the total bytes requested for allocation can be satisfied by reusing previously dead bytes. An average of 20.3% of the writes out of the L1 cache to the next level of the memory hierarchy are eliminated the *IOT* as well. Using the *IOT* instead of CORC to identify dead objects produces a slight drop in reference count coverage; however, the *IOT* requires much less storage space on chip and is

capable of meeting throughput requirements necessary to successfully eliminate writes out of the cache and recycle dead objects.

Chapter 5

Conclusions

In this thesis, two hardware techniques (CORC and the *IOT*) were designed to allow hardware and software to work together for increased memory efficiency. The *weak generational hypothesis* observes that programs with automatic memory management tend to allocate many short-lived objects. Due to this, many objects die while remaining in cache. The proposed hardware techniques exploit the *weak generational hypothesis*, targeting only recently allocated objects that are likely to die. We identify dead objects and significantly reduce write bandwidth by *cleaning* cache lines known to be dead, eliminating the unnecessary write-backs of dead cache lines. Dead objects are also recycled, reducing the overall heap pressure of a program. This can allow smaller heap sizes to be used without losing performance, or can increase the effective execution time of the program before it must be paused for garbage collection.

Hardware implementations for each technique were evaluated. CORC proved to require latencies to high to remain effective when scaled up to standard cache sizes, leading to the design of the *IOT*. The *IOT* executes in largely the same manner as CORC, however it is limited to cover a smaller subset of objects than CORC. This limitation allows the *IOT* to be implemented with much lower storage requirements and greatly reduces the number of

entries that must be searched in fully associative fashion. Due to these benefits, the *IOT* is able to meet throughput requirements necessary to remain effective. We show that under realistic conditions, 12% of the total bytes allocated are satisfied by recycled memory space and 20.3% of all writes out of the L1 cache are eliminated.

5.1 Future Work

5.1.1 Targeting an ASIC and Evaluating Energy Cost/Savings

In this work we target synthesis and implementation of our hardware design to an *FPGA*. We target both the baseline cache and our structures towards the same hardware to facilitate throughput comparisons, however *FPGAs* have overheads associated with them that *ASICs* do not. Targeting our hardware designs towards an *ASIC* can better validate timing of our hardware.

Each write in any system has an energy cost associated with it; therefore, each write eliminated saves energy. One study, [61], shows eliminating all useless writes (including, but not limited to writes of dead data) can reduce power consumption due to the cache-memory subsystem by 15% over the evaluated benchmarks using explicit memory management. However, our proposed hardware requires additional logic on chip which consumes energy as well. A thorough evaluation is needed to compare the energy cost of running our hardware versus the energy saved by reducing memory bandwidth. After accurate power estimates are acquired through targeting our design for implementation on an *ASIC*, those numbers can be built into an energy aware simulator to study energy costs or savings of our proposed hardware structures.

5.1.2 Evaluating the Infant Object Table with Larger Caches

Increasing *IOT* Entries for Larger Caches

We have shown CORC to be unscalable for larger caches; however, the *IOT* has not been thoroughly evaluated on larger caches. In Figure 4.13, we saw that an *IOT* of 128 entries is large enough to cover the majority of objects found dead by CORC for a 512 KB cache. As the additional chip space required by the *IOT* and the number of entries involved in fully-associative search scales linearly with the number of entries in the *IOT*, we expect our hardware to run slower as the number of entries increases. However, we also expect objects to remain in the level 2 cache longer, allowing the *IOT* to run slower. A thorough evaluation of the *IOT* with increased entries and increased cache sizes is needed.

Multi-level Cache Hierarchies

As a secondary approach to fully evaluating the *IOT* on larger caches, thorough evaluation for multi-level cache hierarchies is needed. We saw that a significant number of writes can be reduced out of the level 1 cache and that level 1 cache misses can be reduced effectively with a 32 entry *IOT*. However, a quantitative approach of the effects of the *IOT* on the full memory hierarchy is needed. How does eliminating writes in the level 1 cache affect level 2 and level 3 cache accesses? How are main memory accesses affected? How does eliminating writes and recycling dead objects as early as possible compare to waiting and eliminating writes after garbage collection as in [59]? These are all examples of questions that should be answered, as most modern systems employ the use of multi-level cache hierarchies.

The Infant Object Table in Software

We showed that a small IOT is effective at identifying dead objects and discussed a software implementation in Section 4.4. Future work can implement the described write barrier and evaluate the overhead associated with reference counting the IOT in software, as well as evaluate the effects on the memory system and garbage collection overheads. As a software implementation is not bound by timing constraints, it can scale up to larger sizes easier than the hardware implementation. However, the tradeoff for larger IOT sizes would be increased memory traffic overhead. A thorough study of the IOT in software is needed to determine the cost of the additional write barrier and memory traffic overhead as the size of the IOT is varied.

5.1.3 Relaxing Exact Fit Recycling Requirements

As described in Section 4.3.5, we only allow allocation requests to be satisfied by a dead object of the exact size of the request. This constraint can be relaxed to allow an allocation request to be satisfied by any dead object of at least the requested size. However, this can create fragmentation in memory.

A similar, but perhaps more interesting question to study would be: Can we exploit contiguous allocation for recycling as well as write reduction? As objects in consecutive IOT entries are likely to span contiguous memory, we can combine the memory space represented by consecutive dead IOT entries to form larger chunks of contiguous dead space to satisfy larger allocation requests.

5.1.4 Contaminated Garbage Collection

Contaminated Garbage Collection [16] is a garbage collection technique that is able to collect objects in cyclical structures. It works by associating an object with the lowest possible stack frame the object may be directly or indirectly (through a heap reference stored into an object associated with this frame) referenced by. When a stack frame pops, the set of objects associated with that frame are known to be dead. However, similar to reference counting, it is limited by not being able to identify the death of some objects. The information needed by Contaminated Garbage Collection to maintain object liveness is similar to the instructions needed by reference counting. The *IOT* can implement Contaminated Garbage Collection alongside (or in place of) reference counting, to increase the coverage of objects that can be identified as dead.

5.1.5 Quantitative Analysis of Recycling on Garbage Collection and Allocation Overheads

We have shown that the *IOT* can satisfy 28% of all allocation requests and 12% of all bytes requested for allocation through recycling. A thorough evaluation of these results on allocation and garbage collection overheads is needed.

As an overview, each allocation request to the software allocator incurs a cost. In the case of contiguous allocation, this cost is typically a few instructions:

- load the current bump pointer value and the value of the end of the allocation space
- increment the bump pointer by the size of the allocation request

- check that the incremented bump pointer does not overrun the end of the allocation space
- store the incremented bump pointer value

However, if available, an allocation request can be satisfied by the *IOT* in hardware in only a single instruction. An evaluation of this on allocation overheads is suggested.

When considering garbage collection, recycling dead objects can increase effective program execution time between garbage collection cycles. However, as pointed out in [35], although time may be increased in between garbage collection cycles, the amount of work done in the garbage collection cycle may be increased. This is because the garbage collection algorithm used in the young generation requires work proportional to the number of live objects. Recycling can lead to higher survivor rates in the young generation, meaning more work per young generation collection cycle. An evaluation of this tradeoff, increased time between collection cycles (leading to fewer collection cycles over the full execution of the application) for increased time per collection cycle, is needed when using the *IOT*.

5.1.6 Memory Access Tracing at the Microarchitectural Level

As described in Section 2.3, our traces currently trace memory accesses at the bytecode level, which gives us an accurate representation of the memory traffic created by the application in the application's heap. However, we do not trace all memory traffic. Memory traffic created by the *JVM* executing the application is ignored, as well as any memory traffic generated by the operating system. In order to evaluate the effectiveness of the *IOT* when considering all memory traffic, we propose one of the following 2 methodologies for future work:

- We can accurately trace all memory traffic created by the system during execution of the benchmark. We can achieve this by running our modified *JVM* on top of a memory profiling tool such as Valgrind [54]. Proper interleaving of instructions from the tracing implemented in the *JVM* and from the memory tracing tool would be necessary, as well as proper matching of object IDs (virtual addresses) to the physical addresses seen by the memory profiler. These traces can then be run in our trace-based cache simulator.

- A full-system simulator, such as Simics [48], can be modified to implement the needed *ISA* modifications and functionality of the hardware additions. The modified *JVM* can then be run on top of the simulator, passing reference count instructions to the simulator as needed.

References

[1] Process integration, devices, and structures. *International Technology Roadmap for Semiconductors*, 2013.

[2] A. W. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.*, 19(2):171–183, February 1989.

[3] Hezi Azatchi and Erez Petrank. Integrating generations with advanced reference counting garbage collectors. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 185–199, Berlin, Heidelberg, 2003. Springer-Verlag.

[4] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 92–103, New York, NY, USA, 2001. ACM.

[5] David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 50–68, New York, NY, USA, 2004. ACM.

[6] David F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 207–235, London, UK, UK, 2001. Springer-Verlag.

[7] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Commun. ACM*, 20(7):513–518, July 1977.

[8] Samson Belayneh and David R. Kaeli. A discussion on non-blocking/lockup-free caches. *SIGARCH Comput. Archit. News*, 24(3):18–25, June 1996.

[9] Suparna Bhattacharya, Kanchi Gopinath, and Mangala Gowri Nanda. Combining concern input with program analysis for bloat detection. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 745–764, New York, NY, USA, 2013. ACM.

[10] Suparna Bhattacharya, Mangala Gowri Nanda, K. Gopinath, and Manish Gupta. Reuse, recycle to de-bloat software. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 408–432, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 25–36, New York, NY, USA, 2004. ACM.

[12] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 344–358, New York, NY, USA, 2003. ACM.

[13] Blackburn, S. M. *et al.* The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, 2006.

[14] Santiago Bock, Bruce Childers, Rami Melhem, Daniel Mosse, and Youtao Zhang. Analyzing the impact of useless write-backs on the endurance and energy consumption of pcm main memory. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 56–65, 2011.

[15] H. Cam, M. Abd-El-Barr, and S. M. Sait. A high-performance hardware-efficient memory allocation technique and design. In *Computer Design, 1999. (ICCD '99) International Conference on*, pages 274–276, 1999.

[16] Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 264–273, 2000.

[17] J. Morris Chang and Edward F. Gehringer. Object-caching for performance in object-oriented systems. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer &Amp; Processors*, ICCD '91, pages 379–385, Washington, DC, USA, 1991. IEEE Computer Society.

[18] J. Morris Chang and Edward F. Gehringer. Performance of object caching for object-oriented systems. In *Proceedings of the IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration*, VLSI '93, pages 83–91, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[19] J. Morris Chang and Edward F. Gehringer. A high-performance memory allocator for object-oriented systems. *IEEE Trans. Comput.*, 45(3):357–366, March 1996.

[20] Takashi Chikayama and Yasunori Kimura. Multiple reference management in flat ghc. In *Logic Programming, Proceedings of the Fourth International Conference, Melbourne, Victoria, Australia, May 25-29, 1987 (2 Volumes)*, pages 276–293, 1987.

[21] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the 1st international symposium on Memory management*, ISMM '98, pages 37–48, New York, NY, USA, 1998. ACM.

[22] Thomas W. Christopher. Reference count garbage collection. *Software: Practice and Experience*, 14(6):503–507, 1984.

[23] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, December 1960.

[24] Patrick Crowley and Jean-Loup Baer. On the use of trace sampling for architectural studies of desktop applications. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '99, pages 208–209, New York, NY, USA, 1999. ACM.

[25] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Commun. ACM*, 19(9):522–526, September 1976.

[26] Steven M. Donahue, Matthew P. Hampton, Ron K. Cytron, Mark Franklin, and Krishna Kavi. Hardware support for fast and bounded-time storage allocation (extended abstract). In *Second Annual Workshop on Memory Performance Issues (WMPI*, 2002.

[27] Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. Increasing pcm main memory lifetime. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 914–919, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[28] Daniel Frampton, David F. Bacon, Perry Cheng, and David Grove. Generational real-time garbage collection: A three-part invention for young objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, pages 101–125, Berlin, Heidelberg, 2007. Springer-Verlag.

[29] Scott Friedman, Praveen Krishnamurthy, Roger Chamberlain, Ron K. Cytron, and Jason E. Fritts. Dusty caches for reference counting garbage collection. In *Proceedings of the 2005 workshop on MEmory performance: DEaling with Applications , systems and architecture*, MEDEA '05, pages 3–10, 2005.

[30] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman Publishers, San Francisco, California, 1990.

[31] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 313–326, New York, NY, USA, 2005. ACM.

[32] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 143–153, New York, NY, USA, 2005. ACM.

[33] IBM. Power isa version 2.06 revision b. 2010.

[34] Ciji Isen and Lizy John. Eskimo: Energy savings using semantic knowledge of inconsequential memory occupancy for dram subsystem. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 337–346, 2009.

[35] José A. Joao, Onur Mutlu, and Yale N. Patt. Flexible reference-counting-based hardware acceleration for garbage collection. In *Proceedings of the 36th Annual International*

Symposium on Computer Architecture, ISCA '09, pages 418–428, New York, NY, USA, 2009. ACM.

[36] Richard E. Jones and Chris Ryder. A study of java object demographics. In *Proceedings of the 7th international symposium on Memory management*, ISMM '08, pages 121–130, New York, NY, USA, 2008. ACM.

[37] Poul-Henning Kamp. Malloc(3) revisted. http://phk.freebsd.dk/pubs/malloc.pdf.

[38] Jin-Soo Kim and Yarsun Hsu. Memory system behavior of java programs: Methodology and analysis. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '00, pages 264–274, New York, NY, USA, 2000. ACM.

[39] Doug Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html.

[40] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[41] Kevin M. Lepak and Mikko H. Lipasti. On the value locality of store instructions. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 182–191, New York, NY, USA, 2000. ACM.

[42] Kevin M. Lepak and Mikko H. Lipasti. Silent stores for free. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 22–31, New York, NY, USA, 2000. ACM.

[43] Kevin M. Lepak and Mikko H. Lipasti. Temporally silent stores. *SIGPLAN Not.*, 37:30–41, October 2002.

[44] Jarrod A. Lewis, Bryan Black, and Mikko H. Lipasti. Avoiding initialization misses to the heap. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 183–194, 2002.

[45] Wentong Li, Saraju Mohanty, and Krishna Kavi. A page-based hybrid (software-hardware) dynamic memory allocator. *IEEE Comput. Archit. Lett.*, 5(2):13–13, July 2006.

[46] Wentong Li, Mehran Rezaei, Krishna Kavi, Afrin Naz, and Philip Sweany. Feasibility of decoupling memory management from the execution pipeline. *J. Syst. Archit.*, 53(12):927–936, December 2007.

[47] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983.

[48] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.

[49] J. Harold McBeth. On the reference counter method. *Commun. ACM*, 6(9), September 1963.

[50] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[51] Matthias Meyer. An on-chip garbage collection coprocessor for embedded real-time systems. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 517–524. IEEE, 2005.

[52] Afrin Naz, Krishna Kavi, Wentong Li, and Philip Sweany. Tiny split data-caches make big performance impact for embedded applications. *J. Embedded Comput.*, 2(2):207–219, April 2006.

[53] Afrin Naz, Krishna Kavi, JungHwan Oh, and Pierfrancesco Foglia. Reconfigurable split data caches: a novel scheme for embedded systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 707–712, New York, NY, USA, 2007. ACM.

[54] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42:89–100, June 2007.

[55] Chih-Jui Peng and Gurindar S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical report, University of Wisconsin-Madison, 1989. Report 860, CS Department.

[56] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[57] Mehran Rezaei and Krishna M. Kavi. Intelligent memory manager: Reducing cache pollution due to memory management functions. *J. Syst. Archit.*, 52(1):41–55, January 2006.

[58] David J. Roth and David S. Wise. One-bit counts between unique and sticky. *SIGPLAN Not.*, 34(3):49–56, October 1998.

[59] Jennifer B. Sartor, Wim Heirman, Stephen M. Blackburn, Lieven Eeckhout, and Kathryn S. McKinley. Cooperative cache scrubbing. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 15–26, New York, NY, USA, 2014. ACM.

[60] William J. Schmidt and Kelvin D. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 76–85, New York, NY, USA, 1994. ACM.

[61] C. Shelor, J. Buchanan, and K. Kavi. Quantifying wasted write energy in the memory hierarchy. In *International Conference on Computers And Their Applications*, 2014.

[62] Jonathan Shidal, Zachary Gottlieb, Ron K. Cytron, and Krishna M. Kavi. Trash in cache: Detecting eternally silent stores. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 8:1–8:9, New York, NY, USA, 2014. ACM.

[63] Jonathan Shidal, Ari J. Spilo, Paul T. Scheid, Ron K. Cytron, and Krishna M. Kavi. Recycling trash in cache. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 118–130, New York, NY, USA, 2015. ACM.

[64] W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some practical methods for rapid combinator reduction. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 159–166, New York, NY, USA, 1984. ACM.

[65] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 157–167, New York, NY, USA, 1984. ACM.

[66] Paul R. Wilson. Uniprocessor garbage collection techniques (Long Version), 1994. Unpublished, available at `ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps`.

[67] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 1–116, London, UK, UK, 1995. Springer-Verlag.

[68] David S. Wise. Stop-and-copy and one-bit reference counting. *Inf. Process. Lett.*, 46(5):243–249, July 1993.

[69] Greg Wright, Matthew L. Seidl, and Mario Wolczko. An object-aware memory architecture. Technical report, Mountain View, CA, USA, 2005.

[70] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[71] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 14–23, New York, NY, USA, 2009. ACM.

[72] Benjamin Zorn. The measured cost of conservative garbage collection. *Software: Practice and Experience*, 23(7):733–756, 1993.

Appendix A

Data

Data for any Figures not found here can be found at: `https://github.com/shidalj/research.git`

cache size	8k	16k	32k	128k
avrora	0.273	0.342	0.387	0.446
fop	0.241	0.29	0.356	0.453
lusearch	0.045	0.06	0.07	0.796
pmd	0.151	0.187	0.233	0.346
sunflow	0.287	0.342	0.387	0.459
xalan	0.248	0.323	0.391	0.48

Table A.1: data for figure 2.5

cache size	8k	16k	32k	128k
avrora	0.04	0.075	0.11	0.163
fop	0.116	0.15	0.219	0.337
lusearch	0.019	0.028	0.033	0.839
pmd	0.163	0.224	0.276	0.339
sunflow	0.25	0.31	0.361	0.444
xalan	0.18	0.245	0.3	0.385

Table A.2: data for figure 2.6

cache size	8KB	8KB-steady	16KB	16KB-steady	32KB	32KB-steady	128KB	128KB-steady
avrora	0.273	0.04	0.342	0.075	0.387	0.11	0.446	0.163
fop	0.241	0.116	0.29	0.15	0.356	0.219	0.453	0.337
lusearch	0.045	0.019	0.06	0.028	0.07	0.033	0.796	0.839
pmd	0.151	0.163	0.187	0.224	0.233	0.276	0.346	0.339
sunflow	0.287	0.25	0.342	0.31	0.387	0.361	0.459	0.444
xalan	0.248	0.18	0.323	0.245	0.391	0.3	0.48	0.385

Table A.3: data for figure 2.7

granularity	full-line granularity	byte level granularity
avrora	0.3031	0.3165
fop	0.2061	0.2154
lusearch	0.0588	0.0636
pmd	0.1867	0.2
sunflow	0.2899	0.3011
xalan	0.2718	0.2831

Table A.4: data for figure 2.8

# bits	2 bits	3 bits	4 bits
avrora	0.3031	0.3078	0.3078
fop	0.2061	0.2218	0.2218
lusearch	0.0588	0.0599	0.0599
pmd	0.1867	0.1973	0.1973
sunflow	0.2899	0.2931	0.2931
xalan	0.2718	0.2847	0.2847

Table A.5: data for figure 2.9

trace length	Total	Steady-state
avrora	0.455	0.155
fop	0.57	0.557
lusearch	0.881	0.926
pmd	0.367	0.362
sunflow	0.576	0.569
xalan	0.516	0.435

Table A.6: data for figure 2.13

size	8	16	32	64	128
avrora	0.1981667437	0.2718715673	0.2891716383	0.292698027	0.2987532459
fop	0.1335851991	0.1909606744	0.2101530397	0.2160605246	0.222353969
lusearch	0.01718685369	0.02604190395	0.0323131975	0.03676485514	0.04131890676
pmd	0.1007736561	0.1288279136	0.1462680582	0.1557577767	0.1615182266
sunflow	0.05977738307	0.09331675548	0.1149413854	0.1361077561	0.1500828893
xalan	0.1211082095	0.2107104453	0.2550715302	0.2695480246	0.2765726971
average	0.1050996742	0.1536215433	0.1746531415	0.184489494	0.1917666558

Table A.7: data for figure 4.2

# of references	0	1	2	3	10
avrora	0.2326290907	0.2891716383	0.2894782808	0.2897635748	0.291411293
fop	0.1617468029	0.2101530397	0.2106926149	0.2131226238	0.2154191789
lusearch	0.02478566051	0.0323131975	0.03569039486	0.03891783536	0.04390221239
pmd	0.1119614616	0.1462680582	0.1464781804	0.1467508581	0.1489627554
sunflow	0.09939135583	0.1149413854	0.1157122558	0.116053286	0.116973357
xalan	0.2064022834	0.2550715302	0.2558217668	0.256172604	0.2602716511
average	0.1394861091	0.1746531415	0.1756455823	0.176796797	0.1794900746

Table A.8: data for figure 4.3

Benchmark	without exploiting locality	exploiting locality
avrora	0.2891716383	0.3323888227
fop	0.2101530397	0.2639435078
lusearch	0.0323131975	0.03772429133
pmd	0.1462680582	0.1701102473
sunflow	0.1149413854	0.2428277087
xalan	0.2550715302	0.2941740212
average	0.1746531415	0.2235280998

Table A.9: data for figure 4.4

Benchmark	allocation requests	bytes requested
avrora	0.232	0.107
fop	0.268	0.158
lusearch	0.099	0.015
pmd	0.238	0.088
sunflow	0.548	0.239
xalan	0.319	0.136
average	0.284	0.1238333333

Table A.10: data for figure 4.5

Benchmark	write reduction-std	write reduction-locality
avrora	0.2835006929	0.2914986279
fop	0.228623411	0.2349437382
lusearch	0.0327597616	0.03390955137
pmd	0.1560443556	0.1590999497
sunflow	0.2413700414	0.2449520426
xalan	0.273772937	0.2790632448
average	0.2026785332	0.2072445258

Table A.11: data for figure 4.6

Benchmark	allocation hit rate	allocation hit rate w/recycling
avrora	0.156844803	0.2127143593
fop	0.1775065156	0.27284479
lusearch	0.0563545612	0.06551685783
pmd	0.1131611385	0.1726441765
sunflow	0.220264327	0.3830671864
xalan	0.1374811826	0.2239569149
harmonic mean	0.1190362909	0.1643953277

Table A.12: data for figure 4.7

Benchmark	overall hit rate	overall hit rate w/recycling
avrora	0.965896339	0.966665077
fop	0.9040129011	0.9102913696
lusearch	0.8914958165	0.8923601685
pmd	0.8982007582	0.9035977896
sunflow	0.9568824913	0.9624051651
xalan	0.9260928426	0.931005234
harmonic mean	0.9228782151	0.9268533678

Table A.13: data for figure 4.8

throughput	ideal	1	0.67	0.5	0.4	0.33
avrora	1	1	0.9914529915	0.8504273504	0.4914529915	0.3247863248
fop	1	1	0.9962825279	0.970260223	0.5910780669	0.3643122677
lusearch	1	1	1	0.9898989899	0.7070707071	0.4444444444
pmd	1	1	1	0.9369747899	0.6218487395	0.3991596639
sunflow	1	1	1	0.996350365	0.9416058394	0.5967153285
xalan	1	1	1	0.9905956113	0.473354232	0.2852664577
average	1	1	0.9979559199	0.9557512216	0.6377350961	0.4024474145

Table A.14: data for figure 4.9

throughput	ideal	1	0.67	0.5	0.4	0.33
avrora	1	1.001674133	0.9666033628	0.8724482706	0.1538614241	-0.07861807931
fop	1	0.997833479	0.9959513662	0.9784937242	0.5365150799	-0.1105302121
lusearch	1	0.9977407164	0.9944407931	0.9885392705	0.4294008106	-0.07284478037
pmd	1	0.9845804362	0.9836704292	0.9369264007	0.3430052477	-0.08420935555
sunflow	1	0.9989517409	0.9984765954	0.9881997374	0.8434714031	0.2063062092
xalan	1	0.9978260007	0.9959332153	0.9848324795	0.2874077492	-0.06536418994
average	1	0.9964344177	0.9891792937	0.9582399805	0.4322769524	-0.03421006801

Table A.15: data for figure 4.10

throughput	1	0.67	0.57	0.5
avrora	12.12774451	88.39321357	956.0459082	5330.846307
fop	17.05389222	63.38522954	175.4690619	930.4031936
lusearch	33.46107784	48.0499002	113.506986	163.508982
pmd	26.436	93.244	204.262	1373.604
sunflow	18.19361277	30.2255489	302.2095808	469.1357285
xalan	22.85341365	61.63654618	230.1686747	1320.514056

Table A.16: data for figure 4.11

Benchmark	IOT - 16	IOT - 32	IOT - 64
avrora	0.8966408269	0.9431524548	0.9612403101
fop	0.797752809	0.8426966292	0.8707865169
lusearch	0.6285714286	0.8285714286	0.9428571429
pmd	0.7939914163	0.8712446352	0.9184549356
sunflow	0.6356589147	0.7157622739	0.7984496124
xalan	0.820971867	0.9130434783	0.9488491049
average	0.7823464912	0.8552631579	0.9002192982

Table A.17: data for figure 4.12

Benchmark	IOT - 32	IOT - 64	IOT - 128
avrora	0.8967032967	0.9142857143	0.9428571429
fop	0.6157894737	0.6421052632	0.7631578947
lusearch	0.3110102157	0.8206583428	0.8740068104
pmd	0.8337874659	0.8719346049	0.9155313351
sunflow	0.5520833333	0.6180555556	0.6875
xalan	0.8217054264	0.8798449612	0.9263565891
average	0.6184249629	0.7830609212	0.8451708767

Table A.18: data for figure 4.13