Trash in Cache: Detecting Eternally Silent Stores

Jonathan Shidal Zachary Gottlieb
Ron K. Cytron
Washington University in St. Louis
shidalj@wustl.edu zachgottlieb@gmail.com
cytron@wustl.edu

Krishna M. Kavi
University of North Texas
Krishna.Kavi@unt.edu

Abstract

The gap between processing and storage speeds remains a concern for computer system designers and application developers. This disparity can be bridged in part by eliminating unnecessary stores, thereby reducing the amount of traffic that flows from the processor and first-level caches to the slower components of the storage subsystem. Reducing the "write" traffic can improve program performance, save power, and increase the longevity of storage components that have limited write endurance. Techniques have been proposed and evaluated for identifying various classes of stores that can be silenced. A relatively unexplored class of such stores are those that would write data that is dirty, but dead. Such data appears as if it needs to be written back to memory from cache, yet it can be proven that the application can never subsequently access the data.

In this paper, we suggest identifying garbage (trash) in cache, so that the dirty bytes associated with the trash need not be written to memory. We propose and evaluate a simple technique based on reference counting that finds a subset of these "eternally silent" (dead) stores. When applied to popular benchmarks, our results show that a significant fraction of the writes to memory can be silenced based on the impossibility of an application subsequently accessing the data.

Categories and Subject Descriptors B.3.2 [*Design Styles*]: Cache memories; C.0 [*General*]: [Hardware/software interfaces]; D.3.4 [*Processors*]: Memory management; D.4.2 [*Storage Management*]: Garbage collection

Keywords Write back, reference count, cache

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

The memory wall (the gap between processing and storage speeds) remains a concern to computer designers and application developers. Moreover, many groups are now researching non-volatile memories such as flash and phase change memories that can provide more scalable storage as opposed to DRAM, but which also have limited write endurance. Both of these problems can be alleviated if techniques can be found that decrease the amount of data that is written from the processor and first-level caches to other caches and main memories.

Several approaches for *silencing* stores are summarized in Section 2. All such techniques attempt to *squash* (eliminate) write operations from a running program, with the goal of saving time, power, and wear. In this paper, we present and evaluate a new technique for discovering data that is dirty, but dead, in a cache. Because the data is dirty, eviction would cause such data to be written from the cache, perhaps to an intervening cache, perhaps to main memory. Where our analysis is successful, such data is proven *dead*, in the sense that the running application cannot possibly access the data at any time in the future. Such data need not be written from cache, and we follow the work of others in saying that those writes have been *silenced*.

The most closely related work to ours uses explicit deallocation instructions (in languages such as C++) to clear the dirty bits of any deallocated storage still contained in cache [4]. Our technique is concerned with languages (such as Java) in which dead storage is automatically collected. The challenge here is to detect dirty but dead storage (i.e., trash) in cache, prior to the data's eviction and without any explicit advice from the running application that the data is dead.

This kind of information could be harvested from an actual garbage-collection cycle, except that the execution of such a cycle would surely cause most, if not all, data in cache to be evicted before it could be proven dead. We therefore turn to less invasive techniques, even though they are conservative in determining dead storage. In this paper we apply a specialized form of *reference counting* to detect dead data, which is implemented solely in the cache itself, requiring no extra storage or activity outside of the cache. We have prototyped our approach and conducted experiments whose results show that a significant fraction of the writes issued from DaCapo benchmarks can be found dead prior to eviction.

Our paper is organized as follows. Section 2 presents some background and related work. Section 3 details our approach. Section 4 describes our experiments and their results. Section 5 presents some conclusions and future work based on our results.

2. Background and Related Work

As originally defined, a *silent store* instruction writes values that exactly match the values already stored at the specified memory address [10]. One study [10] showed that 20% to 68% of the store instructions issued by some common benchmarks were silent. Efficient techniques for identifying this class of silent stores and *squashing* them have been considered. One study [9] reports the squashing of between 31% and 50% of a program's silent store instructions, implying that only 6% to 34% of the benchmarks' *total* store instructions are squashed. Those results imply there is room to find and eliminate more silent stores, and our paper presents one such effort.

The idea of silent stores has been generalized to *temporally silent stores* [11]. 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 [7] 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 [11] 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.

The most closely related work to the approach we present here is a recent effort [4] that proposed squashing the stores of data that have been explicitly deallocated. The focus of that study was the energy saved by avoiding write backs of explicitly deallocated data. When an application written in a language such as C++ issues a `delete` operation, the dirty bits of the affected data are reset. If those bits are still clear on eviction, the data would not be written back from cache. However, a block of storage may be dead, in the sense that the application will not subsequently access its bytes, some time before that block is explicitly deallocated by the program. This period of time in which an object is *rotting* (dead but not yet buried) may be important, because the object's bytes may suffer eviction prior to the application issuing the explicit deallocation. Those bytes would be written needlessly from the cache.

Our focus in this paper is on languages such as Java that feature automatic storage management. The challenge here is to find dead data in cache sufficiently quickly and noninvasively so as to squash the write backs of such data from cache. Our technique does not collect such data in the sense of garbage collection. Instead, we use a noninvasive garbage-collection approach to determine that the data is dead and then reset the dirty bits of such data to avoid write backs.

Because the data we find using this approach is truly dead in the running program, the program cannot subsequently reference the data's addresses until an actual garbage collection cycle is performed and the associated storage became eligible for allocation again. It is thus provably safe for us to squash the writes of such data.

While our work concerns the dynamic discovery of data that need not be written to memory, complementary research efforts have tried to reduce the generation of that dead data in the first place. Generally, called *bloat reduction*, both static [2] and dynamic [6] techniques have been proposed. Combining bloat reduction with our technique is the subject of future work, once bloat-reduction program transformers are available.

3. Approach

The goal of our work is to determine data that is dead in cache prior to that data's eviction. Dead data cannot be referenced by an application, and such data would eventually be reclaimed via garbage collection.

An exact approach to this problem would involve running an accurate and complete garbage collection algorithm. For example, a mark-and-sweep algorithm [13] marks all of an application's live data. Any data left unmarked is dead, and the garbage collection algorithm could reclaim that data for subsequent allocation. At the same time, such data could also be marked nondirty if the data resides in cache. This idea suffers from the following difficulties:

- There is some time lapse between data becoming dead and the garbage collection algorithm's execution. While the dead data is rotting, it may be evicted from cache prior to the garbage collector's execution, and thereby be unnecessarily written from cache.
- Garbage collection is a memory-intensive activity, and as such it tends to make full use of the cache. Thus, it is

likely that the very execution of the garbage collector will evict most, if not all, data from cache before any of that data can be proven dead. Such data is also unnecessarily written from cache.

Because of these difficulties, we propose a limited application of *reference counting* to determine dead data in cache.

3.1 Reference Counting

Reference counting [13] associates a counter with each object in the runtime heap (the area from which objects are dynamically allocated). For each object T, its reference count accurately reflects the number of references to T. These references can come from any object (including T itself), from the stack, from registers, or from statically allocated structures. Maintenance of reference counts is typically accomplished via a *write barrier*: a segment of code that executes whenever an application executes an instruction that can affect reference counts. Such instructions include those that modify the contents of the runtime heap, change the values of registers, change the contents of the stack, or change the values of static variables.

The write barrier for reference counting is interested only in operations that affect references (pointers). When a reference r is modified, the write barrier considers the old value of r (r_{old}) and the value of r established by the instruction (r_{new}). If r_{old} is not null, then the reference count associated with the object referenced by r_{old} is decremented. Similarly, if r_{new} is not null, then the reference count at its referenced object is incremented. Finally, if a reference count becomes 0, no references exist to its associated object, and the storage associated with the object is then known to be dead.

Although some applications of reference counting are somewhat common (for example, smart pointers in C++), the technique is not widely implemented to manage entire heaps for the following reasons:

- Extra storage is required throughout the heap to contain the objects' reference counts.
- Extra cost is incurred at each store due to the write barrier's activity.
- Maintenance of the reference counts increases the traffic between the CPU and memory.
- If an object T participates in a structure with cycles of references, then the reference count of T can never become 0. Correspondingly, any object referenced directly or indirectly from T will always have a positive reference count, and therefore be ineligible for collection by this technique. Common examples of such cycles include doubly-linked lists and trees whose nodes also reference their parent.

Except for the last issue, the form of reference counting we propose does not share the above disadvantages.

3.2 Cache-Only Reference Counting

We essentially propose architectural support for reference counting solely within the cache. Outside of the cache, reference counts do not exist (at least, for our purposes). We require no (software) write barrier, and we do not use reference counting to collect dead objects.[1] Instead, we use a reference count's transition to 0 to clear the dirty bits of the associated object. The reference counts themselves are not contained in memory. They are allocated instead in the cache, which manages their values as described below.

`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 runstack is popped, in response to a method's `return`.

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

Figure 1. Directives that interface between the running application and the cache.

Our approach requires the cache to detect the actions performed by the running application as described in Figure 1. Languages like Java have type systems that allow runtime knowledge of which stores are data and which stores are references. It is thus possible to interpret the above actions exactly. It is easiest to conceive of these actions as realized by instructions explicitly executed by the program. The Java Virtual Machine (JVM) conveniently has operations that map well to these actions. For example, the JVM's `putfield` and `putstatic` operations correspond to a `refstore` instruction when the affected storage is a reference. Other instances of those JVM instructions store data (of type `int`, `double`, etc.) instead of references. Where such instructions are not available, a cache can be similarly advised by cache directives in the spirit of the PowerPC's *data cache* instructions.

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. 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.

[1] However, in Section 5 we propose an approximation of this.

- 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 observation that objects tend to die young [1], which has been verified for Java [8]. In other words, it is likely that a newly allocated object will become dead (whether we can detect this or not) relatively soon after it is allocated. If objects die young, then there is a good chance that their bytes are still in cache at the time of their death.

The experimental results we report in Section 4 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.

3.3 Cache Implementation Details

Our experimental setup described in Section 4 includes a custom cache simulator, in which we implemented the cache protocol described here. Although the protocol is realized in software for this paper, 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 1. 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 the 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.

3.3.1 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 $\lceil \frac{a}{b} \rceil$ 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.

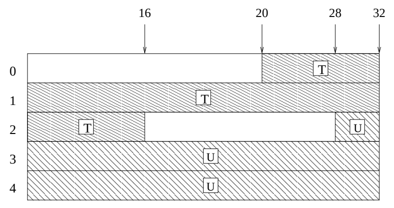

Figure 2. Illustration of `allocate` instructions. The cache shown here has 5 lines, each with 32 bytes.

An illustration of a series of `allocate` instructions is shown in Figure 2. 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:

Line	Hosted objects
0	`T`$\rightarrow\{\,20\ldots31\,\}$
1	`T`$\rightarrow\{\,0\ldots31\,\}$
2	`T`$\rightarrow\{\,0\ldots15\,\}$, `U`$\rightarrow\{\,28\ldots31\,\}$
3	`U`$\rightarrow\{\,0\ldots31\,\}$
4	`U`$\rightarrow\{\,0\ldots31\,\}$

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 4.

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. As described in Section 3.1, 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.

3.3.2 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 we have prevoiusly shown [5].

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.

This stack-summarizing optimization [5] works only for a single thread. We therefore detect if multiple threads concurrently have stack references to an object, and if this occurs, 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. We return to this issue in Section 4.2. Another limitation of our current approach is that object liveness cannot be tracked for portions of objects that suffer eviction prior to death.

4. Experimental Results

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 1 issued by that benchmark.
2. We ran each trace through a cache simulator that includes the in-cache reference-counting as described in Section 3.

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 1, 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 in Section 2 on eliminating dead writes in explicitly managed languages [4] could address this traffic and could be combined with our work without interference. Our results would also hold for data caches that can be *split* according to application activity [12].

We wrote a trace-based cache simulator to implement our approach as described in Section 3. This simulator is highly componentized and greatly simplified our experimentation. The simulator processes the instructions shown in Figure 1 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. 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 [3]. 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. 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* and cannot be decremented. Experiments justifying this choice are left out for space considerations.

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.

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 3 shows the fraction of squashed write backs found in caches ranging in size from 8KB to 128KB.

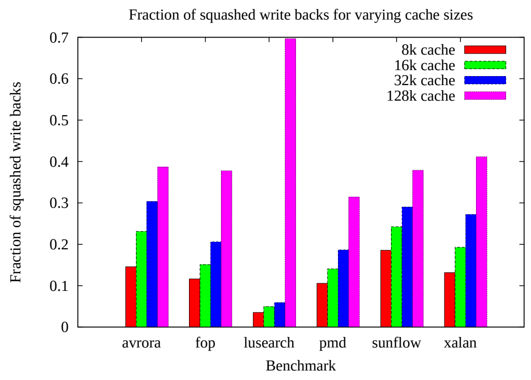

Figure 3. Fraction of squashed write backs found for varying cache sizes, each having 32-byte lines.

These results show that for a reasonably sized 32KB level 1 cache we can squash an average fraction of 0.219 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 4 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 4 uses the same configuration as the previous experiment. In the steady state, we continue to squash 13% of all write backs given a 32KB cache.

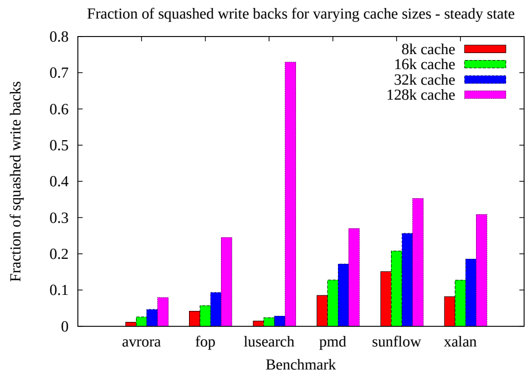

Figure 4. Fraction of squashed write backs found for varying cache sizes in the applications' steady state.

4.1 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 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 5.

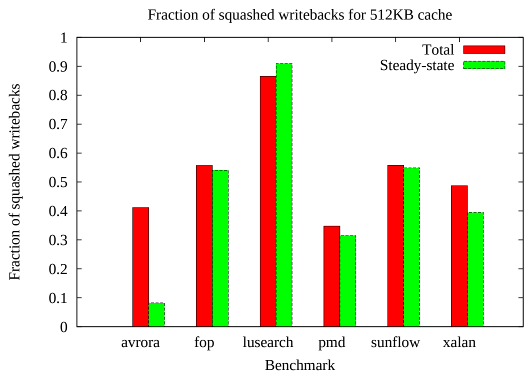

Figure 5. Total (steady-state + initialization) and steady-state fraction of squashed write backs with 512KB cache.

As shown in Figure 5 all benchmarks show a significant increase in the fraction of squashed write backs when given a larger cache. On average, 47% 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.

4.2 References from Multple Threads' Stacks

As we explained in Section 3.3.2, 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, as discussed in Section 5.

4.3 Performance impact

Previous work that examined similar savings for programs with explicit deallocation [4] squashed write backs in L2 cache at the rate of almost 21%. We find on average 47% squashed write backs in our modeling of an L2 cache (of half the size of [4]). 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 [4], we find the lifetime "gain" for such devices to be 1.87–nearly 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. We plan to address this directly in future work. However, we note that an implementation of the cache actions described in Figure 1 can balance cache latency with write squashing as follows:

- The reference actions in Figure 1 can be achieved via associative lookup of object identifiers, which are simply the addresses affected by the operations. Caches are already equipped with logic and associative structures to perform such lookups.
- All of the actions in Figure 1 could be performed off the critical path of L1 cache activity, so that the cache's reads and writes (hits) are executed at the cache's lowest possible latency.

  Actions realized off the critical path may be incomplete at the time a cache line is evicted. If so, the writes of data from such lines may not be detected as squashable prior to eviction. While such writes may be unnecessary, no incorrectness follows from their issue.

Further investigation into these issues is clearly needed, but the results we have presented here justify such future efforts because our technique can eliminate a significant fraction of writes.

5. Conclusions and Future Work

In this paper we propose a limited form of reference counting in the cache only, with the intention of reducing memory traffic to slower levels of the memory hierarchy. We have shown that cache-only reference counting has the potential to reduce significantly the number of bytes that apparently must be written from the level-1 cache toward main memory. The logic required to implement our approach can be confined to the cache, and the information needed from the running program is readily available for languages of interest.

The limited application of reference counting in cache avoids the overheads normally associated with wholesale reference counting. The results we report in Section 4 are encouraging, and we hope they will inspire others as well as ourselves to further this work. We next describe some ideas and directions for further work.

- The approach described here is for a single cache and single core. Horizontally, this work should be expanded to consider multiple cores each with its own cache.
- Vertically this work should be expanded to consider applications of cache-only reference counting past the first-level cache. The results presented in Section 4.1 found that many more writes were squashed for most benchmarks in a 512KB cache than in a 32KB cache.
- Currently when an object dies in cache, its associated write backs are squashed, but the associated storage cannot be reused until an actual garbage-collection cycle finds it dead. The dead storage in cache could be *recycled* so that it can satisfy a subsequent allocation request. We are currently studying this idea, which could save traffic in both directions between cache and memory.
- Finally, there are other approximate techniques of finding dead objects that might be efficiently implemented in cache [5]. We plan to experiment with these techniques in the future.

References

[1] A. W. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.*, 19(2): 171–183, Feb. 1989. ISSN 0038-0644. . URL `http://dx.doi.org/10.1002/spe.4380190206`.

[2] S. Bhattacharya, K. Gopinath, and M. G. 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 &#38; Applications*, OOPSLA '13, pages 745–764, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. . URL `http://doi.acm.org/10.1145/2509136.2509522`.

[3] 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.

[4] S. Bock, B. Childers, R. Melhem, D. Mosse, and Y. 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, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-367-4. . URL `http://dx.doi.org/10.1109/ISPASS.2011.5762715`.

[5] D. J. Cannarozzi, M. P. Plezbert, and R. K. Cytron. Contaminated garbage collection. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 264–273, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. . URL `http://doi.acm.org/10.1145/349299.349334`.

[6] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, and J. Murphy. Patterns of memory inefficiency. In *ECOOP*, pages 383–407, 2011.

[7] S. Friedman, P. Krishnamurthy, R. Chamberlain, R. K. Cytron, and J. 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, Washington, DC, USA, 2005. IEEE Computer Society. . URL `http://dx.doi.org/10.1145/1147349.1147353`.

[8] R. E. Jones and C. 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. ISBN 978-1-60558-134-7. . URL `http://doi.acm.org/10.1145/1375634.1375652`.

[9] K. M. Lepak and M. 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. ISBN 1-58113-196-8. . URL `http://doi.acm.org/10.1145/360128.360133`.

[10] K. M. Lepak and M. 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. ISBN 1-58113-232-8. . URL `http://doi.acm.org/10.1145/339647.339678`.

[11] K. M. Lepak and M. H. Lipasti. Temporally silent stores. *SIGPLAN Not.*, 37:30–41, October 2002. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/605432.605401`.

[12] A. Naz, K. Kavi, W. Li, and P. Sweany. Tiny split data-caches make big performance impact for embedded applications. *J. Embedded Comput.*, 2(2): 207–219, Apr. 2006. ISSN 1740-4460. URL `http://dl.acm.org/citation.cfm?id=1370998.1371002`.

[13] P. R. Wilson. Uniprocessor garbage collection techniques (Long Version), 1994. URL `ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`. Unpublished, available at `ftp://ftp.cs.utexas.edu/pub/garbage/gcsu`