Recycling Trash in Cache

Jonathan Shidal Ari J. Spilo
Paul T. Scheid Ron K. Cytron
Washington University in St. Louis
shidalj@gmail.com arispilo@wustl.edu
paultscheid@wustl.edu cytron@wustl.edu

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

Abstract

The disparity between processing and storage speeds can be bridged in part by reducing the traffic into and out of the slower memory components. Some recent studies reduce such traffic by determining dead data in cache, showing that a significant fraction of writes can be *squashed* before they make the trip toward slower memory. In this paper, we examine a technique for eliminating traffic in the other direction, specifically the traffic induced by dynamic storage allocation. We consider *recycling* dead storage in cache to satisfy a program's storage-allocation requests.

We first evaluate the potential for recycling under favorable circumstances, where the associated logic can run at full speed with no impact on the cache's normal behavior. We then consider a more practical implementation, in which the associated logic executes independently from the cache's critical path. Here, the cache's performance is unfettered by recycling, but the operations necessary to determine dead storage and recycle such storage execute as time is available. Finally, we present the design and analysis of a hardware implementation that scales well with cache size without sacrificing too much performance.

Categories and Subject Descriptors B.3.2 [*Memory structures*]: Design styles—Cache memories; C.0 [*Computer Systems Organization*]: General—Hardware/software interfaces; D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

Keywords cache, garbage collection, reference counting

ISMM'15, , June 13-14, 2015, Portland, OR, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

1. Introduction

The memory wall (the gap between processing and storage speeds) remains of concern to computer architects and application developers. The use of larger caches, particularly with newer sub-30nm technologies, will lead to higher energy consumption relative to energy consumed by the processing elements. Energy consumption also increases with higher traffic between main memory and caches. The inclusion of higher-density storage components for main memory (such as flash and phase-change devices [24]) 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). These technologies also increase the latency and energy consumption disparity between reads and writes, offering greater reward for techniques that avoid or suppress write operations to main memory. 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.

There is interest therefore in reducing the traffic between CPU and memory to save time, power, and wear. Several approaches for *silencing* stores from the CPU to memory have been proposed [18–20]. Those techniques generally try to show that a store is unnecessary—that it can be *squashed* before it makes the trip to main memory.

Recently, a technique [26] was introduced that identified a new class of squashable stores in garbage-collected languages. That technique discovered data in cache that was *dead*, in the sense that the application could never subsequently reference that data. Such data, even if marked dirty, need not be written on eviction. For languages with explicit deallocation, the death of data in cache is indicated directly by the application, and the stores associated with such data can also be squashed [8].

While finding and squashing dead stores in cache has benefits of its own, we investigate here the further use of such dead storage. We seek to *recycle* the dead storage to satisfy subsequent storage-allocation requests. The usefulness of storage mapped to cache is thus extended, as the storage

for one deceased object is bequeathed to a newly instantiated object. While we investigate this idea for the more difficult case of garbage-collected languages, its application for languages with explicit deallocation is straightforward.

Our paper is organized as follows. Section 2 presents some background and related work. Section 3 presents our main idea—recycling dead storage in cache—and evaluates it under optimal circumstances. A more realistic and practical treatment is presented in Section 4. Section 5 establishes metrics for storage requests and profiles the sizes of blocks available for recycling using our technique. Those results motivated our investigation of an extended form of our approach, which allows larger storage blocks to satisfy requests for smaller storage. The timing of the logic needed to implement our technique is presented in Section 6. The impact of our technique on a program's cache hit rates is presented in Section 7.

2. Background and Related Work

Our work builds on techniques for *squashing* (also called *silencing*) stores. A squashed store is a write from CPU or (L1) cache toward main memory, perhaps to to an intervening (L2 or beyond) cache, that can be aborted because it is provably unnecessary. The source of such stores could be an actual store instruction, but it could also be a write-back from cache caused by eviction. The seminal work in this area squashed stores because the cached value was already present in memory [18, 19], or because the cached value was temporally consistent with what could have been stored in memory [14, 20].

More recently, research has identified storage in cache that might disagree with values in main memory, but whose stores are squashable because the cached values can never again be referenced by the program. We are especially interested in this class of squashed stores. For languages with explicit deallocation, the `free` or `delete` instructions can be forwarded to the cache so that the associated cached locations can be marked non-dirty [8, 16, 21]. Subsequent eviction would then refrain from writing those values to main memory. For languages featuring garbage collection, an analogous approach [25] takes advantage of generational garbage collection. Following a nursery collection cycle, the entire nursery is known to be dead and *cache scrubbing instructions* are issued to the cache. The cache can then squash writes of dead data back to main memory. When the size of the nursery is similar to that of the last level cache (8M), this approach squashed about 60% of writes to main memory.

Also for garbage collected languages, another approach using a form of reference-counting limited only to the cache has been proposed and evaluated [26], (hereafter, called CORC for cache-only reference counting). As a side effect of using reference counting, CORC identifies dead storage at the object level immediately upon object death. That work showed that an average of 30% of the bytes written back from L1 cache (32K) to main memory were squashed by finding dead but dirty cache lines. For an L2 cache (512K), 50% of its written bytes were squashed. While this approach finds squashable writes from the hardware side, cache-scrubbing [25] relies on a software garbage-collection cycle to discover dead storage. That cycle would surely evict all data in L1 and most data in L2 caches. On the other hand, CORC's hardware approach squashes writes in L1 and L2, squashing a similar amount of traffic (50% using only a 512K cache) as compared with cache scrubbing (60% using an 8M cache). These hardware and software approaches are complimentary, and future work could investigate their concurrent use.

Because we are interested in finding recyclable storage as early as L1 cache, we choose to build upon CORC, whose approach can be summarized as follows:

- The cache maintains a descriptor for each allocated object, which includes the object's size and reference count. These descriptors are created when an `allocate` instruction is sent to the cache, specifying the address and size of the allocated object.

  Storage allocated in this manner is obtained as usual from the runtime heap, but the cache is informed of the allocation by a special instruction so that it can track the liveness of the storage, as described below.
- The cache is equipped with logic to maintain reference counts for objects. The CORC studies showed a two-bit reference count sufficed for almost all objects whose death could be determined. With two bits for reference counts, the counts must *stick* [1] when they reach 3.
- The cache acts on instructions that affect reference counts, such as reference loads and stores, but caches ordinarily respond to those instructions anyway to provide or modify data. The main difference here is that the store of a reference (pointer) is distinguished from the store of data. A reference store that points away from p and toward q causes p's reference count to decrement and q's reference count to increment (assuming both are non-null).
- All reference-counting structures and activity are confined to the cache. The main memory is oblivious to all aspects of the CORC approach. Thus, the cache is able to track references to an object's lines only while they reside continuously in cache since their allocation.
- Success for CORC is a race between determining objects' death and write-backs (due to eviction or cache flush) of the objects' cache lines. If a line is evicted prior to determining its associated objects' death, then the write-backs must be realized. Alternatively, if all objects in a line are determined to be dead, then that line's write-back can be squashed.

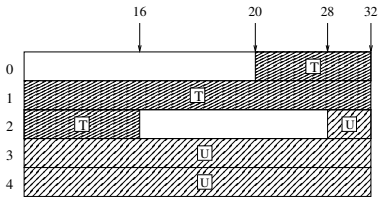

Figure 1. Cache lines holding objects T and U. These objects occupy 60 and 68 bytes, respectively

Figure 1 shows objects T and U, of sizes 60 and 68 bytes, respectively, allocated in 5 cache lines. Suppose the stack points to T and then some field in U points to T, causing its reference count to reach 2. If the stack drops its reference to T and the field in U points away from T, then its reference count drops to 0 and it is found dead by CORC. A subsequent cache flush would result in the following:

- Line 1's write-back would certainly be squashed, because the entire line has been determined to be dead prior to eviction.
- The squashing of the other two lines associated with T is less certain. If no other data has been allocated to line 0, then its writes can be squashed. 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 squashed, 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. Using CORC to identify dead space versus waiting for the garbage collector allows for heap space to be recycled between garbage collection cycles, effectively reducing the total heap space used for a given set of object allocations. This can lead to increased program execution time before a collection is needed.

In summary, as we describe in Section 3, we propose to *recycle* the storage from T to V, if a sufficient fraction of T's bytes are still resident when V is allocated.

Other related work Because CORC collects recently allocated (young) objects in cache, the technique is a form of generational collection [22], with the cache serving as an unusual form of nursery [2]. Because most collections would occur in the nursery, attention has been paid to making the collection of young objects more efficient [13]. Another aspect of the CORC approach is that it applies reference counting only to certain storage, namely objects that reside in cache. Selective application of reference counting has been considered with success [5].

A thorough study [6] of the costs of various styles of garbage collection has established the following:

- The cost of a write barrier is relatively low for most benchmarks. Such barriers are necessary for generational collectors, so most collectors impose a write barrier. The CORC approach uses this same write barrier to inform the cache of reference stores. It also requires an *allocate* barrier, but the number of allocations is expected and observed to be lower than the number of writes.
- Reference counting incurs substantial overhead when implemented throughout all memory. Neither our approach nor the CORC approach upon which we build requires *any* reference counting outside of the cache's logic, and we show in Sections 4 and 6 that the required logic executes sufficiently quickly to be effective.
- A hybrid generational reference-counting approach was shown to provide good performance [6], with reference counting applied only for certain mature objects. Our approach essentially conducts reference-counting in a form of a nursery (the cache) for free.

A large-scale hardware-assisted garbage collector has been investigated [17], which deploys reference counts in hardware throughout all of memory. Objects that have been collected are made available for recycling (to the software allocator) 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 increased cache hit rates across all benchmarks.

An object-caching coprocessor has also been designed [9] and evaluated [10]. In that work, newly created objects reside in a special cache that is reference counted. A bit vector in hardware keeps track of available locations for recycling storage. The results reported in terms of squashed writes are similar to the CORC results for L2 caches [26]. By contrast, 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 CORC.

A reference-counting and recycling scheme was proposed for the Lisp language [23]. 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.

Confirming the benefits of reducing writes for wear-limited devices, one study shows the dramatic extent to which the life of phase-change memories (PCMs) can be extended by minimizing write-backs [12].

Finally, while all of the above work, including our own, benefits from short-lived objects, researchers have tried to reduce the *bloat* of programs by static and dynamic analyses [4]. To date, the tools resulting from that research have not been made available, so experimentation and comparison could be the subject of future work.

Contributions Based on the work described above and the results we present in the rest of this paper, our contributions can be summarized as follows:

- We implement storage recycling in cache for storage blocks of diverse size. We obtain very good results under ideal circumstances. In an L1 cache, approximately 28% of our benchmarks' storage allocation requests were satisfied by recycling storage on average. For an L2 cache, the rate rises to almost 50%.
- We evaluate the effectiveness of CORC and our storage recycling when the associated operations are moved away from a cache's critical path. Here, the relative speed of our logic compared with the cache's logic yields curves that show the point beyond which our approach is no longer effective. Based on timing obtained from a hardware implementation, we present results for an off-critical-path implementation under realistic conditions.
- Unlike a traditional nursery as used in [25], the storage we recycle is known to be present in cache. We present results showing the effects of storage recycling on allocation hit-rates and overall hit-rates.

3. Recycling Dead Storage in Cache

In this section, we describe our approach for *recycling* storage in cache that has been determined to be dead. 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 the CORC approach described in Section 1.
- 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, as evaluated in Section 7.

Conceptually, we deploy logic in the cache to keep track of available storage of various sizes [9, 23]. In the example of Figure 1, 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.

Deploying such logic directly in a cache could interfere with the cache's normal operations, which are intended to suffer no unnecessary delays. We consider moving this logic off the critical path in Section 4, but 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.

3.1 Approach

To facilitate comparison, we obtained traces for the 6 Da Capo [7] benchmarks studied in [26]. Those traces were generated by instrumenting the `OpenJDK` (version 1.8.0) JVM (Java Virtual Machine). The traces captured allocation requests, 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 [26]. These traces contain only those instructions necessary to implement the CORC approach: arithmetic instructions, register-only instructions, and jump instructions do not appear in the traces. We return to this point in Section 4.2.

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 the CORC technique to determine dead storage in cache, 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:

- CORC creates an object descriptor for each allocated object `T` that contains the virtual address and size of `T`.
- 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 1, 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 CORC does 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 CORC discovers the death of an object `T` 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 in Section 5 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 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 minimizes the number of garbage-collection cycles. On average, each trace had 2 such cycles.

3.2 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 2 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 line is squashed only if the entire line is known to be dead. Recalling Figure 1, 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 2. 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.

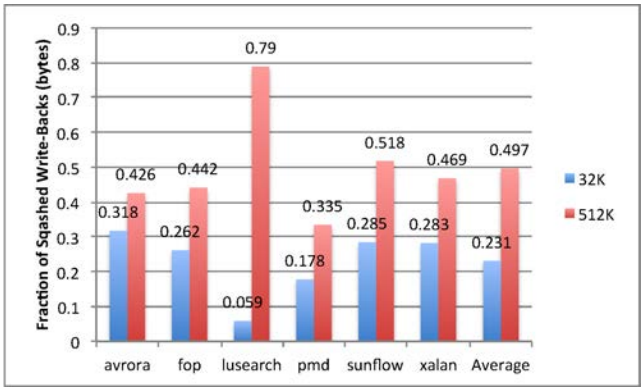

Figure 2. Our reproduction of the results from [26], but with a virtual memory interposed between the program and the storage subsystem. The minor differences between our results and [26] are attributable to the VM's assignment of physical addresses, and the associated changes to their mapping to cache sets.

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. We report on relaxing this constraint in Section 4.
- As illustrated in Figure 1, 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.

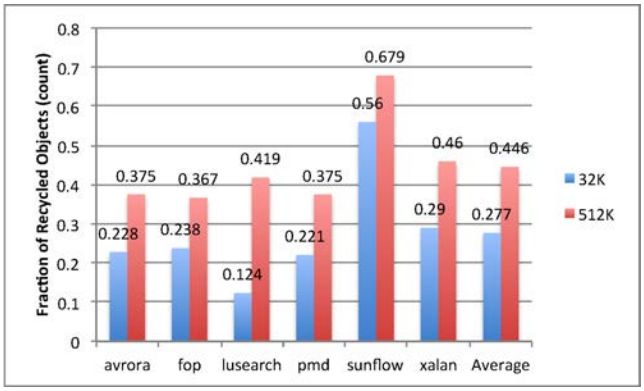

Figure 3. Fraction of allocation requests that were satisfied by the cache, using recycled storage.

Figure 3 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 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.

3.3 L2 Cache

Figures 2 and 3 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 associativity. Because determining dead storage in cache is often a race between detection and eviction, we see (as did [26]) improved performance on the L2 cache. The `lusearch` benchmark is markedly improved in Figure 2, and the availability of the extra dead storage translates into ∼3x improvement for object recycling in Figure 3. On average, the L2 cache allowed nearly 45% of the application's storage requests to be satisfied directly by the cache.

3.4 Recycling: Objects or Bytes

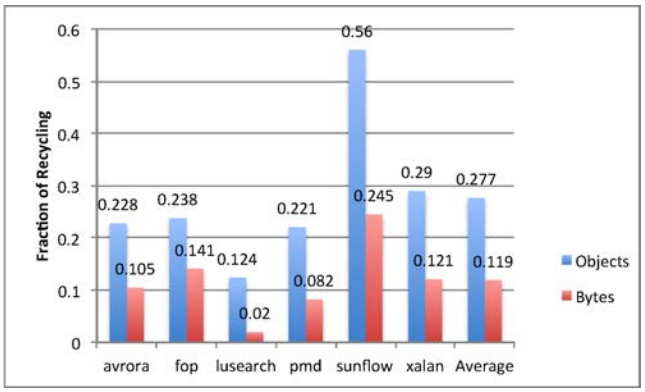

Figure 4. Results are shown for the 32K L1 cache. *Objects* refers to the fraction of object requests that were satisfied by the cache. *Bytes* refers to the fraction of all allocated bytes that were satisfied by the cache.

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 4 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 5, 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.

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`) 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.

4. 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 Section 3 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 our approach 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 Section 3?

In Section 6 we justify this section's results by presenting timing for a circuit to implement our approach.

4.1 Supporting Logic Off the Critical Path

Based on the above observations, we redesigned our approach so that the cache portion of the simulation handles only the traditional cache traffic. Moreover, we assume the cache is *non-blocking* [3] so that it operates at the best possible speed. All nonessential activity is relegated to a queue, which can then be serviced at various rates to allow for experimentation. The transactions moved to this queue include the following:

- all write-backs
- reference count increments and decrements: each increment or decrement occupies a unique queue slot
- morgue activity: when an object is found to be dead, the processing of its contained references is sent to the queue. The queue services one such reference per queue service cycle.

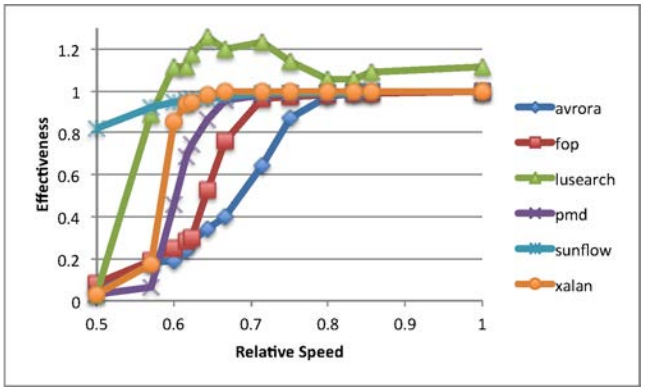

Figure 5. Degradation of our ability to find dead storage as the queue service rate slows relative to the cache rate.

Figure 5 shows the results of slowing the 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. At the left end of the graph, the queue service rate is half of the cache rate. 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 queue drops to 0.5 of the cache speed, our approach is completely ineffective. Figure 5 shows an anomaly for `lusearch`. Its original fraction of squashing writes for the L1 cache was only $\sim$6% (see Figure 2). When a line is found dead, it becomes invalid, and can be chosen by a cache set for subsequent allocation to any address that maps to that cache set.

For `lusearch`, introducing the queue delays this action, which has consequences for cache misses downstream. Because the `lusearch` squashing fraction was already so small, the data in Figure 5 is sensitive to the small rise in squashing with the queue in place for this one benchmark.

We see similar results in Figure 6 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.

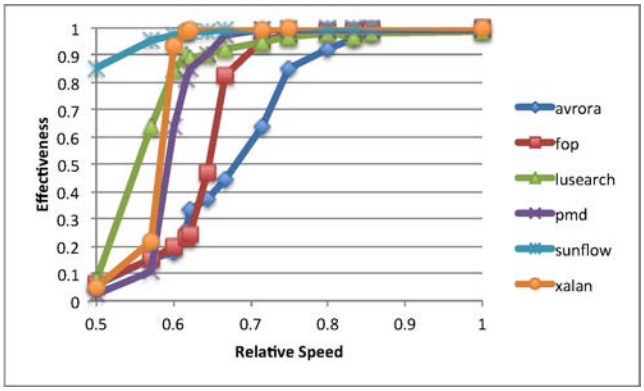

Figure 6. Degradation of our ability to recycle storage.

4.2 Realistic Pacing with Standard Cache Traffic

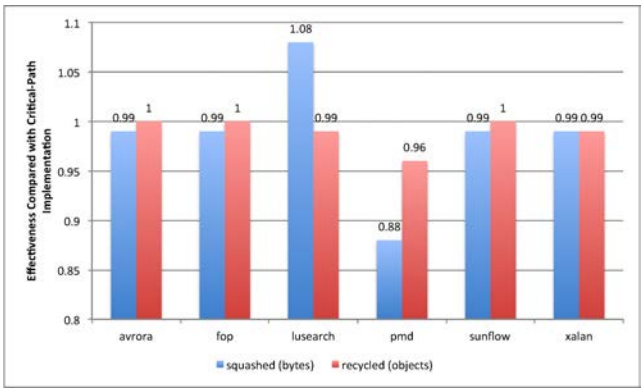

Figure 7. Results obtained with realistic work loads.

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) [15]. For Pentium architectures, one study showed loads and stores comprise 0.58% of all instructions with a standard deviation of 0.05 [11]. We modeled an instruction stream using those statistics, and results obtained as compared with our results from Section 3 are shown in Figure 7. Most benchmarks operate near the best-case results presented in Section 3. While `pmd` suffered the most in terms of finding dead storage, its recycling rate is still robust at 96% of our best results.

5. 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.

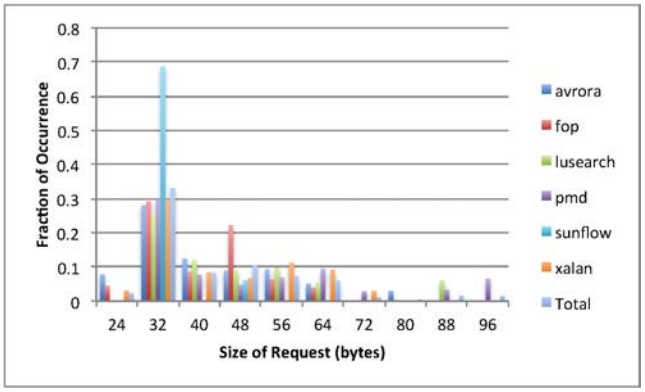

Figure 8. Size of allocation requests by benchmark, accounting for 75% of all requests.

We instrumented our simulator to accumulate the total number of storage-allocation requests by size, and the results are shown in Figure 8. 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 `sunflow`'s allocation requests, which contributes to our success in recycling storage for that benchmark (Figure 3).

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.

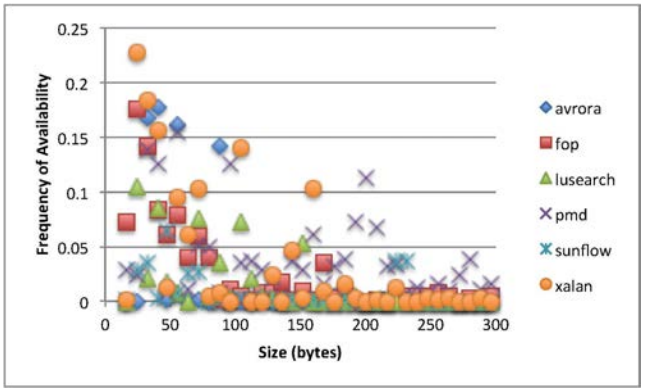

Figure 9. In-cache storage availability for a 32K cache.

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 9 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 while recycling was taking place. Thus, the available blocks shown in Figure 9 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.

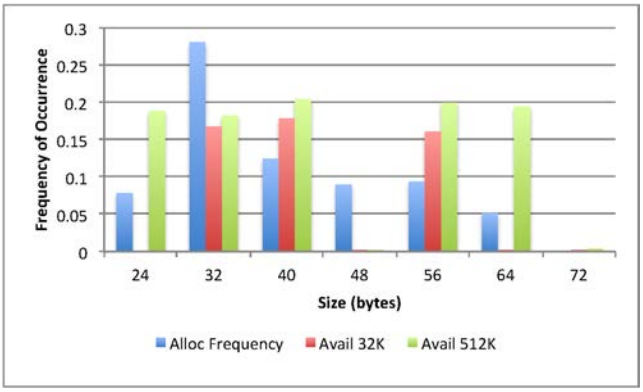

Figure 10. Size of allocation requests and recycled storage availability for 32K and 512K caches.

To illustrate the potential for using larger blocks, Figure 10 combines data from Figures 8 and 9 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.

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.

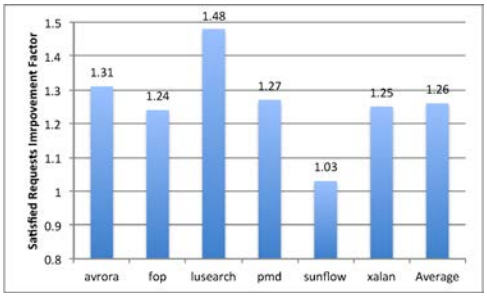

Figure 11. Improvement due to FFBS, displayed as the ratio of allocation requests satisfied by FFBS to the allocation requests satisfied by exact fit.

We ran our FFBS in-cache allocation policy on all benchmarks, and the results are shown in Figure 11. 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.

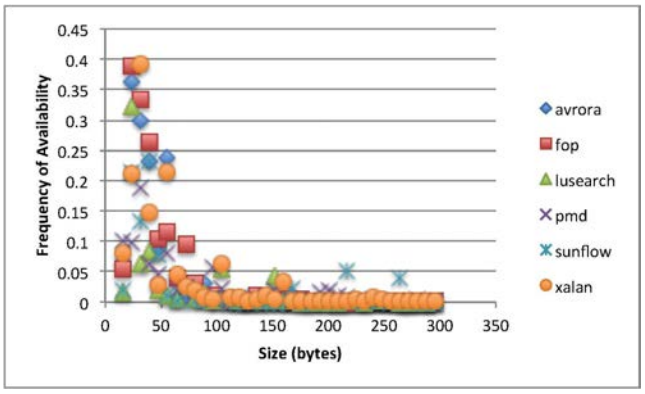

Figure 12. Availability of storage blocks in cache for FFBS.

With FFBS in place, we again sampled the sizes of storage blocks available in cache, and the results are shown in Figure 12. Comparing Figures 9 and 12, we see that the larger block samples are mostly gone, having been used to satisfy the allocation of smaller blocks.

Using a larger block may deprive a downstream allocation request of in-cache storage. However, Figure 13 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.

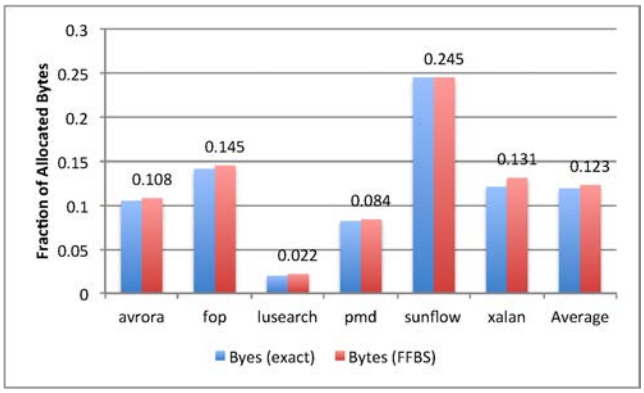

Figure 13. Comparison of recycled bytes for exact-fit and first-fit-by-size (FFBS). Data values for the exact-fit bars can be found in Figure 4.

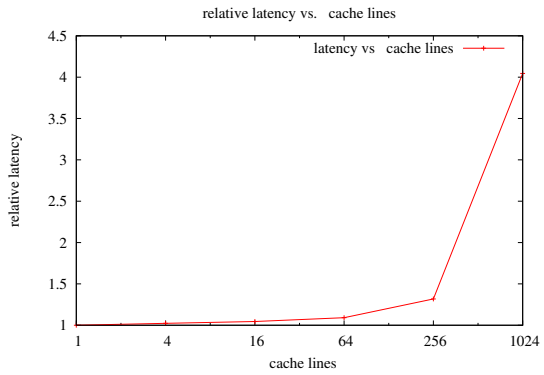

Figure 14. Comparison of the latency of our approach compared to the baseline cache latency as the number of cache lines increases. Here, all cache lines have 32 bytes.

6. Analysis of Hardware Implementations

We validate timing for our storage-recycling approach by implementing the needed logic in VHDL. We used the *Vivado Design Suite* to synthesize and implement the design targeting a Xilinx Virtex-7 VC707 board. The *Vivado Design Suite* was then used to estimate timing of the circuit. We compared our timing estimates with those of a configurable cache with no added logic. The baseline cache was also implemented in VHDL.

Section 6.2 describes a low-latency reference implementation, in which logic is duplicated for each cache line, allowing each line to perform necessary updates concurrently. We show that such a design is unscalable. Section 6.3 then sacrifices area for time, but shows that we can still obtain good results for our benchmarks.

6.1 Hardware Structures

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.

The chip space increases linearly with the number of cache lines, based on the hardware structures described below.

The number of objects each line can contain is bounded by $\lceil \frac{s}{m} + 1 \rceil$ where s is the size of the line in words and m is the minimum object size in words. As an example, a 64 byte cache line with 4 byte words and a minimum object size of 6 words can contain up to $\lceil \frac{16}{6} \rceil + 1 = 4$ objects. Each object contains storage for: an object id (virtual address, 64 bits), a reference count (2 bits), an interval describing the overlap with the cache line, represented as a bit mask with 1 bit per word in the cache line (16 bits using the example above), a thread ID (3 bits), a frame ID (8 bits), and a valid bit.

Each line contains a tag (20 bits) and 3 additional bit masks with 1 bit per word in the line. These masks keep track of the words in the line that were allocated into cache (and therefore valid to be CORC), dead, or that contain a reference to another object.

Based on the example cache above (32 KB, 32 bit physical addresses, 64 byte lines, 4 byte words, 2-way associative) we require 444 bits (approximately 56 bytes) of added storage as well as a copy of our circuit to maintain the stored reference count information per cache line.

6.2 Reference Implementation

Figure 14 compares the latency of our implementation with the baseline cache, where both have 32-byte lines. For cache sizes up to 16K (i.e., up to 512 lines), the relative latency remains under a factor of 3, which would provide the results presented in Section 4. However, the curve is rising sharply and at 32K caches, the latency of our reference count instructions is over 4 times that of standard cache instructions. The dramatic increase in latency can be attributed to the increased fanout of input signals, increased data transport time, and increased logic to aggregate and handle output from each line appropriately.

6.3 Reasonable Implementation

Based on the results presented in Section 5, we see that most object recycling takes place at or under 96 bytes. If we limit the size of the largest recyclable object, then the hardware for maintaining reference count information need not be replicated for each line to remain responsive. For a 32K cache with 32-byte lines, a 96-byte object can span at most 4 lines. We therefore designed a version of our approach with 4 copies of our circuit, with each copy responsible for a given subset of the cache's lines. Based on the virtual memory and associativity, we can ensure that the 3–4 lines that respond to transactions on 96-byte objects are spread among the 4 copies of our circuit.

The results from simulation are shown in Figure 15. We see on average 83% of our best recycling obtained on the cache's critical path (the 32K cache results shown in Figure 3).

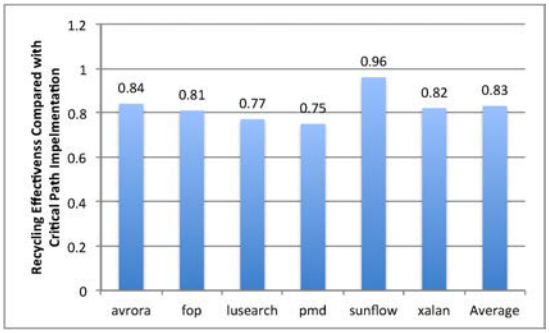

Figure 15. Results for reasonable hardware. Four copies of our circuit are deployed to handle all cache lines of a 32K cache. Object size is limited to 96 bytes.

7. 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:

- 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.

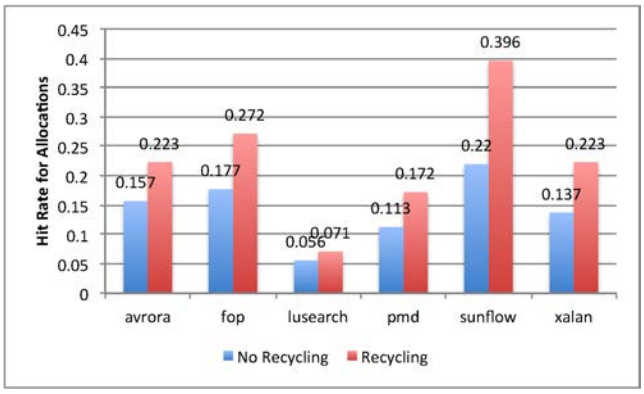

Figure 16. Hit rate for storage allocations and initialization.

Figure 16 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 17 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 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; this is the subject of future work.

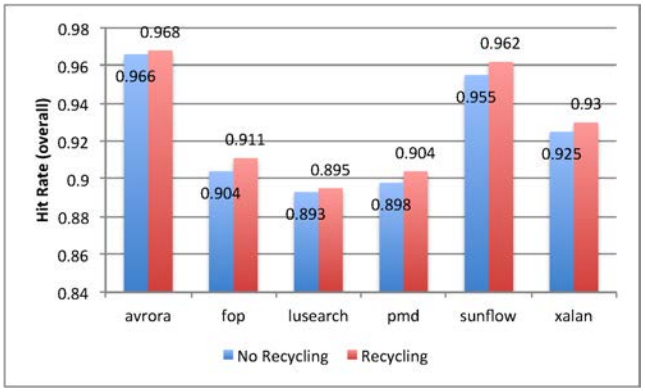

Figure 17. Overall hit rate comparison.

8. Conclusion

We have shown the benefits of a cache recycling its dead storage to satisfy storage-allocation requests of a program. If implemented directly in cache, our studies show that ∼28% of a program's requests can be satisfied in cache for a 32K cache, with almost 50% satisfied in a 512K cache. We have studied the effects of moving our approach off of a cache's critical path, characterizing the decrease in ability to track and recycle dead storage. Based on program characteristics, we developed an implementation that scales well with cache size while providing on average 83% of the critical-path implementation. That implementation meets timing based on analysis of our circuit's model alongside a traditional cache model, and the cache executes without any adverse affects of our circuit. All of our benchmarks experienced improve hit rates by satisfying their allocation requests in cache to the extent possible.

Future work should include a study of the savings experienced in the runtime system by moving allocations to cache. Our studies insisted that all lines of a recycled storage block be present in cache to allow such a block to be recycled. Future work should examine the benefits of relaxing that constraint. The benefits of our approach are magnified in large caches, such as the L2-sized cache we studied. Data resides longer in L2 than L1 prior to eviction, and larger objects occuply relatively fewer lines. Future work should study a complete system (e.g., deployed on an FPGA embodying our circuit) to measure the benefits throughout the application's execution stack. While we save power by decreasing memory traffic, our circuit also consumes power. Future work should examine the overall power savings of our approach.

Acknowledgments

This work was funded by the National Science Fundation under grant 1237425. The authors thank the reviewers for their comments, and Mark Bober of Washington University's Engineering IT department for enabling our experiments to run on the department's cloud infrastructure.

References

[1] D. F. Bacon, P. 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. ISBN 1-58113-831-8. . URL http://doi.acm.org/10.1145/1028976.1028982.

[2] H. G. Baker. Infant mortality and generational garbage collection. *SIGPLAN Not.*, 28(4):55–57, Apr. 1993. ISSN 0362-1340. . URL http://doi.acm.org/10.1145/152739.152747.

[3] S. Belayneh and D. R. Kaeli. A discussion on non-blocking/lockup-free caches. *SIGARCH Comput. Archit. News*, 24(3):18–25, June 1996. ISSN 0163-5964. . URL http://doi.acm.org/10.1145/381718.381727.

[4] S. Bhattacharya, M. G. Nanda, K. Gopinath, and M. 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. ISBN 978-3-642-22654-0. URL http://dl.acm.org/citation.cfm?id=2032497.2032524.

[5] S. M. Blackburn and K. 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. ISBN 1-58113-712-5. . URL http://doi.acm.org/10.1145/949305.949336.

[6] S. M. Blackburn, P. Cheng, and K. 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. ISBN 1-58113-873-3. . URL http://doi.acm.org/10.1145/1005686.1005693.

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. . URL http://doi.acm.org/10.1145/1167473.1167488.

[8] 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, 2011. ISBN 978-1-61284-367-4.

[9] J. M. Chang and E. 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. ISBN 0-8186-2270-9. URL http://dl.acm.org/citation.cfm?id=645460.654558.

[10] J. M. Chang and E. 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. ISBN 0-444-89911-1. URL http://dl.acm.org/citation.cfm?id=645943.674513.

[11] P. Crowley and J.-L. 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. ISBN 1-58113-083-X. . URL http://doi.acm.org/10.1145/301453.301573.

[12] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. 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. ISBN 978-3-9810801-6-2. URL http://dl.acm.org/citation.cfm?id=1870926.1871147.

[13] D. Frampton, D. F. Bacon, P. Cheng, and D. 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. ISBN 3-540-73588-7, 978-3-540-73588-5. URL http://dl.acm.org/citation.cfm?id=2394758.2394767.

[14] 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, 2005.

[15] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman Publishers, San Francisco, California, 1990.

[16] C. Isen and L. 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.

[17] J. A. Joao, O. Mutlu, and Y. 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. ISBN 978-1-60558-526-0. . URL http://doi.acm.org/10.1145/1555754.1555806.

[18] 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.

[19] 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.

[20] 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.

[21] J. A. Lewis, B. Black, and M. 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.

[22] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983. ISSN 0001-0782. . URL http://doi.acm.org/10.1145/358141.358147.

[23] C.-J. Peng and G. S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical report, University of Wisconsin-Madison, 1989. Report 860, CS Department.

[24] M. K. Qureshi, V. Srinivasan, and J. 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. ISBN 978-1-60558-526-0. . URL http://doi.acm.org/10.1145/1555754.1555760.

[25] J. B. Sartor, W. Heirman, S. M. Blackburn, L. Eeckhout, and K. 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. ISBN 978-1-4503-2809-8. . URL http://doi.acm.org/10.1145/2628071.2628083.

[26] J. Shidal, Z. Gottlieb, R. K. Cytron, and K. 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. ISBN 978-1-4503-2917-0. . URL http://doi.acm.org/10.1145/2618128.2618133.