Using Prefetching to Improve Reference-Counting Garbage Collectors [⋆]

Harel Paz[⋆⋆1] and Erez Petrank[⋆⋆⋆2]

[1] IBM Haifa Research Laboratory, Mount Carmel, Haifa 31905, ISRAEL.
[2] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA.

Abstract. Reference counting is a classical garbage collection method. Recently, a series of papers have extended the basic method to drastically reduce its notorious overhead and extend the basic method to run concurrently and efficiently on a modern computing platform. In this paper we investigate the use of prefetching to further improve the efficiency of the reference-counting collector.
We propose potential prefetching opportunities for the advanced reference-counting collector and report an implementation of a collector that employs such prefetching. The proposed prefetch instructions were inserted into the Jikes reference-counting collector obtaining an average reduction of 8.7% of the memory management overheads.

1 Introduction

The performance gap between memory latency and processors' speed is increasing, causing memory accesses to become a performance bottleneck. Cache hierarchies are used to reduce this gap, but caches are of limited size and usually cannot hold the application's entire working set. Thus, cache misses typically form a performance bottleneck. Data prefetching is a technique for reducing or hiding the memory stalls caused by cache misses. Prefetching data to the cache before being accessed by the program hides the latency of loads that miss the cache, and improves the overall program execution time (as prefetch is a non-blocking memory operation). On the negative side, prefetching increases memory traffic, cache pollution, and the number of executed instructions. In addition, to achieve performance improvement, prefetch scheduling should be done with care. Data prefetched too early may be evicted from the cache before used, while a late prefetch will not mask the system latency.

Compiler-inserted data prefetching have been proposed for predictable access patterns such as accesses to arrays and certain pointer applications (e.g., [1–4]). Standard platforms usually automatically prefetch data from the memory whose access can be easily predicted.

Boehm [5] was the first to study the use of prefetching for garbage collection. Prefetching turned out to be very effective for a mark-sweep collector, especially since

[⋆] This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 845/06).
[⋆⋆] Email: paz@il.ibm.com. Work done while the author was at the Computer Science Dept. at the Technion.
[⋆⋆⋆] Email: erez@cs.technion.ac.il. On sabbatical leave from the Computer Science Department, Technion, Haifa 32000, Israel.

the collector accesses each object once in an order that may be predicted. Subsequent work [4, 6, 7] showed that prefetching was able to further reduce the cost of a tracing garbage collection. But all of this work has concentrated on tracing collectors and the potential of prefetching for reference-counting collectors remained open.

In this work we study the use of prefetching for reference counting. Reference counting [8] differs from tracing collectors in the sense that its cost is directly related to the execution of the program, rather than being proportional to the amount of live space. Also, it has better cache behavior, since the objects that are touched during the collection are allocated thereafter and used by the program. Traditionally, it was believed that reference counting had a high overhead and that it required an atomic update for any pointer modification, making it unsuitable for modern parallel platforms. However, it was lately shown [9, 10] that the overhead can be dramatically decreased and that the atomicity requirement can be eliminated. Thus, reference counting became a viable option again for modern computing. Subsequent papers [11–14] have further shown that techniques developed for tracing collectors, can be modified and extended to work with reference-counting collectors as well. This line of work is crucial to making reference counting compete with the efficiency of the thoroughly studied tracing collectors. This paper reports an additional such study, the use of prefetching to improve reference counting efficiency, and show that data accessed by a reference-counting collector can be partially predicted. Thus, prefetching can be used to reduce the cache misses' overhead incurred by a reference-counting collector and improve the collector efficiency.

The representative generic reference-counting algorithm that we use to develop the prefetching techniques in this work uses deferred reference counting [15] and employs the update coalescing write-barrier proposed in [9, 10]. The State-of-the-art efficient reference-counting collectors employ these mechanisms to obtain their efficiency.

We consider three main parts of the memory manager: (1) the reference-count increments stage, (2) the reference-count decrements and object deletion stage, and (3) objects' allocation. In accordance with these stages, we identify five major opportunities where data accesses can be predicted in advance, and prefetch instructions may be inserted to improve performance. We study these opportunities and measure the improved performance of each stage, and of the overall garbage collection execution. We do not study prefetch opportunities with a cycle collection algorithm. A cycle collector is based on some sort of tracing, and hence Boehm's work [5] already handles it.

Implementation and measurements. We have implemented the proposed prefetching insertions with the reference-counting collector supplied with the Jikes RVM [16]. We used the SPECjbb2000 benchmark, the SPECjvm98 benchmarks suite [17] and the DaCapo benchmarks suite [18]. We first measure the general behavior of these benchmarks and show that most objects are accessed multiple times by the reference-counting collector. This means that the original collector (without the prefetching) encounters a lot of cache hits, perhaps reducing the potential impact of prefetching on the execution. Moreover, repetitive accesses tend to be close in time. Nevertheless, the implemented prefetch instructions reduce garbage collection overhead by as much as 14.9% and on average by 8.7% when measured across all benchmarks.

Organization. We review the reference-counting collector in Section 2. The prefetch insertion opportunities of the reference-counting collector are presented in Section 3.

Results and related work are discussed in Sections 4 and 5. We conclude in Section 6. Due to lack of space, implementation details and more results are provided in [19].

2 The reference-counting collector

We start by reviewing the reference-counting collector and presenting a pseudo code that will be used to explain the prefetch instructions insertions later (in Section 3). Basically, a reference-counting collector maintains a reference-count field for each object signifying the number of pointers that reference the object. A naive reference-counting system updates the reference counts during each pointer update via a *write barrier*. When a pointer is modified from pointing to O_1 into pointing to O_2, the write barrier decrements the count of O_1 and increments the count of O_2. When the counter of an object is decremented to zero, it is reclaimed. At that time, the counts of its children are decremented as well, possibly causing more reclamations recursively.

The reference-counting collector we refer to includes two major improvements, which substantially reduce the computational overhead required to adjust reference counters. First, it employs the deferred reference-counting method of Deutsch and Bobrow [15], which tracks only stores into the heap (ignoring local references stores). The second technique employed is the Levanoni-Petrank update-coalescing write barrier [9, 10], which records information on modified objects and uses it to update the reference counts during garbage collection. Consider a pointer slot that, between two garbage collections, is assigned the values $o_0, o_1, o_2, \ldots, o_n$. Instead of executing 2n reference-count updates for these assignments: RC(o_0)$--$, RC(o_1)++, RC(o_1)$--$, RC(o_2)++, $\ldots$, RC(o_n)++, only the two required updates are executed: RC(o_0)$--$ and RC(o_n)++.

To implement update coalescing, the collector we investigate employs two buffers: *ModBuffer* and *DecBuffer*. *ModBuffer* contains the addresses of the objects which were created or modified since the previous collection. Reading these addresses during the garbage collection gives us the o_n values, whose RC values should be incremented. The *DecBuffer* contains the addresses of the o_0 objects recorded before the objects in the *ModBuffer* were first modified after the previous collection. The reference counts of these objects should be decremented. Note that in the *DecBuffer* we have objects that were referenced in the previous collection by the objects that are in the *ModBuffer*.

To simplify this work, the collector we have used works in a stop-the-world manner. An involved mechanism is developed in the original paper [9, 10] to support collector concurrency and application parallelism.

2.1 Pseudo code

Next, we present a general pseudo code of a reference-counting collector which employs the coalescing write barrier. The pseudo code assumes the existence of two buffers, *ModBuffer* and *DecBuffer* as explained above.

Mutator cooperation. The mutators need to execute garbage-collection related code on two occasions: when updating an object and when allocating a new object. This is accomplished by the Update (Figure 1) Procedure and the New (Figure 2) Procedure. Procedure Update (Figure 1) describes the write barrier which is activated at each (heap's) pointer assignment[3]. During the first modification of an object after a collection, the write barrier records the modified object in *ModBuffer* and it sets its dirty bit. Next, the modified object's pointers are recorded in the *DecBuffer*. After the logging has occurred, the actual pointer modification happens. Procedure New (Figure 2) is used when allocating an object. Upon the creation of an object, its address is logged onto *ModBuffer*, and the *dirty* bit of the new object is set. There is no need to record its children slot values as they are all null at creation time.

Phases of the collection. The collector's algorithm runs in phases as follows.

- Mark roots: the objects directly reachable from the program roots are marked.
- Process ModBuffer: the collector clears the dirty marks of the objects logged in *ModBuffer*, while incrementing the reference counts of their current descendants.
- Reclaim garbage: the collector decrements the reference counts of the objects logged in *DecBuffer* while (recursively) reclaiming objects which have a zero reference count and which are not referenced by the system roots.
- Prepare next collection: un-marks the objects referenced from the program roots and prepares the buffers for the next collection.

Collector code. The reference-counting collector's code for a collection cycle is presented in Procedure Collection-Cycle (Figure 3). Procedure Process-ModBuffer (Figure 4) handles the objects logged in *ModBuffer*. These are the objects that were modified or created since the previous collection cycle. This procedure first clears the dirty bit of an object logged in *ModBuffer*, and then increments the reference count of the objects it references. Procedure Process-DecBuffer-and-Release (Figure 5) decrements the reference counts of objects logged in *DecBuffer*, and performs the recursive deletion if necessary. Procedure Prepare-Next-Collection (Figure 6) cleans the *Roots*, *ModBuffer* and *DecBuffer* buffers.

2.2 Allocation using segregated free lists

A garbage collector is accompanied by a memory allocator that serves the application's allocations requests and the collector's reclamations requests. We have built our implementation on the Jikes RVM, which uses the standard segregated free lists allocator [20–22] with the reference-counting collector. Since a couple of prefetch insertion opportunities are proposed for the allocator, we review this allocator below.

A segregated free lists allocator holds, for each possible allocation size, a linked list of available memory. Upon an allocation request, a chunk is taken from the free list of the appropriate size. When a chunk is freed, it is returned to the appropriate free list. Jikes RVM implementation uses a *block-oriented* segregated free lists allocator [20] that works as follows. The heap is divided into blocks, partitioned to chunks of a single size. The free list of any given size consists of a chain of blocks. Each block has an

[3] When dealing with multithreading the write barrier must be modified by either using an atomic operation for the pointer assignment or the non-atomic extension proposed by Levanoni and Petrank [9, 10]. To simplify the discussion, we consider the simplest form of the write barrier. This treatment handles well atomic operations and does not change much if we adopt the more sophisticated non-atomic write-barrier.

```
Procedure Update(o: object, offset: int , new: object)
1.  if not o.dirty then      // OBJECT NOT DIRTY
2.      add o to ModBuffer
3.      o.dirty :=true     // SET DIRTY
4.      foreach pointer field ptr of o which is not NULL
5.          add ptr to DecBuffer
6.  write( o, offset ,new)
```

Fig. 1. Reference counting: Update Operation

```
Procedure New(size: Integer, obj: Object)
1.  Obtain an object obj of size size from the allocator.
2.  insert the address of obj into ModBuffer
3.  obj.dirty := true
4.  return obj
```

Fig. 2. Reference counting: Allocation Operation

```
Procedure Collection-Cycle
1.  accumulate all object directly reference by the program roots onto Roots
2.  Process-ModBuffer
3.  Process-DecBuffer-and-Release
4.  Prepare-Next-Collection
```

Fig. 3. Reference counting- Collection Cycle

```
Procedure Process-ModBuffer
1.  for each object obj whose address is in ModBuffer do
2.      obj.dirty := false
3.      // INCREMENT CURRENT REFERENT OF THE OBJECT obj
4.      for each pointer ptr of obj do
5.          increment rc of object referenced by ptr
```

Fig. 4. Reference counting- Process-ModBuffer

```
Procedure Process-DecBuffer-and-Release
1.  for each object obj whose address is in DecBuffer do
2.      obj.rc−−
3.      if $obj.rc = 0 \wedge obj \notin Roots$ then
4.          for each pointer ptr of obj do
5.              push ptr onto DecBuffer
6.          return obj to the general purpose allocator
```

Fig. 5. Reference counting- Process-DecBuffer-and-Release

```
Procedure Prepare-Next-Collection
1.  $Roots := \oslash$
2.  $ModBuffer := \oslash$
3.  $DecBuffer := \oslash$
```

Fig. 6. Reference counting- Prepare-Next-Collection

```
Procedure Build-Block-Free-List
1.  $markWordAddress$ := address of the first word in the block's bitmap
2.  $markWordEnd$ := address of the last word in the block's bitmap
3.  $previousFree$ := cursor address
4.  while $markWordAddress \leq markWordEnd$
5.      $markWord$ := word referenced by $markWordAddress$
6.      foreach $bit$ in $markWord$
7.          if $bit$ is not set then
8.              $objectRef$ := address of chunk relevant to $bit$
9.              write $objectRef$ into $previousFree$
10.             $previousFree$ := $objectRef$
11.     $markWordAddress$ += size of word
12. write null into $previousFree$
```

Fig. 7. Reference counting- Build-Block-Free-List

```
Procedure Process-ModBuffer
1.  prefetch the first object whose address is in $ModBuffer$
2.  $previous$ := $dummyObject$
3.  for each object $obj$ whose address is in $ModBuffer$ do
4.      prefetch the next object whose address is in $ModBuffer$
5.      $obj$.dirty := false
6.      // INCREMENT CURRENT REFERENT OF THE OBJECT $obj$
7.      for each pointer $ptr$ of $obj$ do
8.          prefetch the $rc$ field of the object referenced by $ptr$
9.          increment $rc$ of object referenced by $previous$
10.         $previous$ := $ptr$
11. increment $rc$ of object referenced by $previous$
```

Fig. 8. Reference counting- Process-ModBuffer with prefetch

```
Procedure Process-DecBuffer-and-Release
1.  prefetch the $rc$ field of the first object whose address is in $DecBuffer$
2.  for each object $obj$ whose address is in $DecBuffer$ do
3.      prefetch the $rc$ field of the next object whose address is in $DecBuffer$
4.      $obj$.rc$--$
5.      if $obj$.rc = 0 $\wedge$ $obj \notin Roots$ then
6.          prefetch the word containing the mark-bit relevant to $obj$
7.          for each pointer $ptr$ of $obj$ do
8.              push $ptr$ onto $DecBuffer$
9.          return $obj$ to the general purpose allocator
10.             //UNMARK THE MARK-BIT RELEVANT TO $obj$
```

Fig. 9. Reference counting- Process-DecBuffer-and-Release with prefetch

associated bit-per-chunk mark array (bitmap), which records the occupancy status of each chunk. When a chunk is allocated the relevant bit is marked. The bit is un-marked when the object held in this chuck is reclaimed. Unused blocks are kept in a block pool. If, upon an allocation request, the relevant free list is empty, a block is taken from the blocks pool, and the first chunk of this block is returned. An empty block (whose all chunks are free) is returned to the blocks pool, and may be used later with a different object size. The collector is responsible of returning empty blocks to the blocks pool.

In Jikes, each free list employs a cursor pointing to the next available chunk for allocation in the corresponding size. After allocating the chunk referenced by the cursor, the cursor is advanced to the next available chunk. To save scanning the bitmap during each allocation, a linked list, containing all the free chunks of a block, is created when the block is first employed after a collection. The Procedure Build-Block-Free-List which builds a block's free list is presented in Figure 7.

3 Prefetching for Reference Counting

We now proceed to describing the prefetch opportunities existing for a reference-counting collector (accompanied by a segregated free lists allocator), and the prefetch insertions that we have applied. We partition the discussion according to the collector phases.

3.1 Process-ModBuffer stage

Consider the pseudo code of the Process-ModBuffer Procedure presented in Figure 4. In this procedure, the collector clears the dirty marks of the objects logged in *Mod-Buffer*, while incrementing the reference count of their descendants. For this phase we propose two prefetch opportunities. The modified Process-ModBuffer Procedure, including the prefetch instructions, is presented in Figure 8. An explanation follows.

The first prefetch opportunity appears during the traversal of *ModBuffer*. The scan of each object referenced by *ModBuffer* imposes a potential cache miss. Since *ModBuffer* is traversed sequentially, this cache miss can be anticipated and avoided. A prefetching of the object that should be scanned in the next iteration is inserted just before scanning the current object. This follows a standard prefetch strategy for loops, placing prefetches for data accessed by the future loop iteration(s) (e.g., [3]). One can imagine prefetching further ahead, but we have obtained the most significant improvements by prefetching a single address ahead. Lines 1 and 4 in Figure 8 execute the proposed prefetch. We will later refer to this strategy as the *ModBuffer-traversal* strategy.

The second prefetch opportunity appears during the reference-count increments. Each increment accesses a reference-count field. To handle a potential cache miss, we slightly delay the increment of an object's reference count. When an increment to a count is required, the count of the object is prefetched; it is only incremented in the following loop iteration. The location of the count is stored in a temporary variable named *previous*. To avoid a special treatment to the first iteration and the implied 'if' statement, we use a dummy object whose reference count is incremented when the first count is prefetched. This delaying strategy is presented in lines 2 and 8-11 of Figure 8. We will later refer to this strategy as the *delay-increment* strategy.

3.2 Process-DecBuffer-and-Release stage

Figure 5 describes the Process-DecBuffer-and-Release Procedure, in which the collector decrements the reference counts of the objects logged in *DecBuffer*, and recursively reclaims the dead objects. This phase also offers two prefetch opportunities. The modified Process-DecBuffer-and-Release Procedure, which includes these prefetch modifications, is presented in Figure 9. A description follows.

Similarly to the Process-ModBuffer stage, the first prefetch opportunity for the Process-DecBuffer-and-Release stage occurs with the traversal of *DecBuffer*. The reference-count field is accessed for each decrement. This time, we do not typically touch the referent, unless we reclaim it. So only the reference-count field of the referent need be prefetched. Note that unlike before, the objects for which we modify the counts are directly referenced by the buffer (and not the children of the objects pointed from the buffer, as in the handling of the Process-ModBuffer Procedure). We exploit the loop prefetch strategy described in the previous stage and prefetch the reference-count field of the next object in the buffer before handling the current one. Lines 1 and 3 of Figure 9 present this prefetch strategy. We will later refer to this strategy as the *DecBuffer-traversal* strategy.

The second prefetch opportunity at this stage occurs during the reclamation of an object. Once an unreachable object is discovered (line 3 of Figure 5), the object is first scanned and all its descendants are recorded in the *DecBuffer*; only then the object is reclaimed. As described in Section 2.2, the reclamation of an object sums up to unmarking the mark-bit corresponding to this object. Accessing the mark-bit creates a potential miss and so we prefetch the relevant mark-bit word as soon as we realize that the object should be reclaimed. Namely, the prefetch is performed right after line 3 of Figure 5. This way, the miss penalty for unsetting the relevant bit later is reduced or even eliminated. Line 6 in Figure 9 presents this prefetch modification. We will later refer to this strategy as the *object-release* strategy.

3.3 Build-Block-Free-List stage

The fifth prefetch opportunity occurs with the segregated free lists allocator. While iterating over a block's bitmap words, we've inserted a prefetch to the next mark-bit word, before processing the current one. The Build-Block-Free-List Procedure was modified to exploit this loop prefetching strategy as presented in lines 6-7 of Figure 10.

4 Measurements

Platform and benchmarks. We have run our measurements on a dual Intel's Xeon 1.8GHz processors workstation. These processors have a 16KB sized L1 cache and a 512KB sized L2 cache. We have used the SPECjvm98 benchmark suite, the SPECjbb2000 benchmark[4] (both described in SPEC's web site [17]), and the DaCapo benchmarks [18][5].

[4] We have slightly modified SPECjbb2000, to run a fixed number of transactions instead of running during a fixed time period.

[5] Measurements of _222_mpegaudio and _201_compress are not presented. _222_mpegaudio does not perform meaningful allocation activity. _201_compress main allocation activity con-

```
Procedure Build-Block-Free-List
1.  $markWordAddress$ := address of the first word in the block's bitmap
2.  $markWordEnd$ := address of the last word in the block's bitmap
3.  $previousFree$ := cursor address
4.  while $markWordAddress \leq markWordEnd$
5.      $markWord$ := word referenced by $markWordAddress$
6.      $markWordAddress$ += size of word
7.      prefetch $markWordAddress$
8.      foreach $bit$ in $markWord$
9.          if $bit$ is not set then
10.             $objectRef$ := address of chunk relevant to $bit$
11.             write $objectRef$ into $previousFree$
12.             $previousFree$ := $objectRef$
13. write null into $previousFree$
```

Fig. 10. Reference counting- Build-Block-Free-List with prefetch

The collector. We have inserted the proposed prefetch instructions into the reference-counting collector of Jikes [16]. Next, we have compared the original reference-counting collector of Jikes, against the collector modified to include these prefetch instructions. In both collectors, we have disabled the cycle collection algorithm. The cycle collector has a characteristic behavior that resembles tracing collectors and it may interfere with the comparison of the reference-counting collectors. For most applications this means a negligible increase in the heap size [13].
Testing procedure. Each benchmark was run ten times for both the original reference-counting collector and the modified reference-counting collector. We report the average of these runs. To guaranty a fair comparison of the garbage collection characteristics, we included only runs in which each benchmark performs the same amount of garbage collections on both collectors. The benchmarks' heap size, employed in our runs, doubles the minimum heap size required (by the reference-counting collector).

4.1 Prefetch improvements

Table 1 presents the improvements achieved by using prefetching. A negative percentage represents a performance improvement, while a positive percentage represents deterioration in performance. Columns 2-4 present the improvements achieved for each one of the reference-counting collector stages implemented by the Process-ModBuffer Procedure (presented in Figure 4), by the Process-DecBuffer-and-Release Procedure (presented in Figure 5), and by the Build-Block-Free-List Procedure (presented in Figure 7). These presented improvements are calculated relatively to the corresponding reference-count stages. Hence, for example, a -10.0% appearing on the second column indicates a 10.0% performance improvement of the **Process-ModBuffer** stage. To make the picture complete, the numbers in parenthesis (in columns 2-4) present the distribution of Jikes original reference-counting collector overhead within the three different

cerns cyclic structures, whose reclamation is not relevant in this work. The DaCapo benchmarks presented are the ones we were able to run with Jikes.

	overhead reduction				overall
Benchmarks	Process ModBuffer	Process DecBuffer and Release	Build Blocks	overall gc	benchmark improvement
jess	-0.1% (36.6%)	-0.9% (51.5%)	-11.9% (11.0%)	-1.8%	-0.9%
db	-11.2% (39.2%)	-6.7% (50.5%)	-8.0% (8.9%)	-8.5%	-0.9%
javac	-12.2% (31.6%)	-8.4% (38.6%)	-18.4% (28.4%)	-12.3%	-3.4%
mtrt	-12.3% (26.8%)	-3.3% (46.4%)	-12.3% (25.2%)	-8.0%	-1.5%
jack	-16.3% (31.3%)	-5.9% (54.5%)	-23.2% (11.6%)	-10.8%	-3.1%
jbb	-8.3% (24.9%)	-6.5% (34.0%)	-26.5% (40.1%)	-14.9%	-4.6%
fop	-12.5% (38.7%)	-8.1% (39.5%)	-11.3% (20.8%)	-10.5%	-2.1%
antlr	-16.1% (30.6%)	-8.4% (42.4%)	-24.3% (25.6%)	-14.6%	-1.3%
pmd	-8.7% (30.8%)	-8.4% (43.8%)	-12.3% (25.3%)	-9.6%	-3.3%
ps	3.0% (38.2%)	-3.7% (52.5%)	-13.0% (7.3%)	-1.7%	-0.6%
hsqldb	-18.8% (30.1%)	-11.0% (41.0%)	-17.6% (26.9%)	-14.9%	-4.6%
jython	-9.4% (32.4%)	-0.5% (50.8%)	-6.6% (15.7%)	-4.4%	-1.7%
xalan	2.4% (43.6%)	-1.2% (51.8%)	-24.4% (4.4%)	-0.6%	-0.6%
average	**-9.3%**	**-5.6%**	**-16.1%**	**-8.7%**	**-2.2%**

Table 1. Reduction in reference-counting overheads obtained by prefetching

stages. These do not add up to 100% as stages such as scanning threads' stack are not counted. The fifth column presents the overall reference-counting's performance improvement achieved. The sixth column introduces the benchmark's overall throughput improvement due to prefetching.

Normally, Jikes runs the Build-Block-Free-List Procedure lazily when a new block is selected for allocations. Therefore, while the Process-ModBuffer and the Process-DecBuffer-and-Release Procedures are activated once per a garbage collection cycle, the Build-Block-Free-List Procedure is activated numerous times during the benchmark run (i.e., between the collections). In order to accurately measure the time overhead of this procedure, we have slightly modified Jikes reference-counting collector (in both versions) to activate the Build-Block-Free-List Procedure continuously (non-lazily), for all non-empty blocks, once at the end of each collection.

One can see that the proposed prefetch strategies reduce the overall overhead of reference-counting for all benchmarks. This is emphasized by the last line of Table 1, which presents the average improvement of each column. For most benchmarks, prefetching is able to reduce the overhead imposed by each one of the three stages. Note, however, that the improvements are not steady among the different benchmarks and among the different stages. We study this issue in Section 4.2 below.

4.2 Reference-counting objects' access behavior

In order to understand the potential of prefetch instruction insertions, we investigate the memory access patterns of the reference-counting collector. Recall that tracing collectors traverse the application's live objects in an arbitrary order (depending on the object's graph). If a mark table is used by a tracing collector, each live object is likely

	window size				
Benchmarks	100	1000	10000	100000	1000000
jess	58.2%	64.1%	66.9%	68.0%	72.7%
db	15.1%	15.8%	16.8%	68.3%	80.2%
javac	31.3%	37.5%	40.5%	42.8%	50.9%
mtrt	13.6%	17.0%	18.5%	19.8%	22.1%
jack	25.9%	27.9%	29.0%	30.5%	36.3%
jbb	23.5%	30.5%	34.7%	38.2%	46.2%
fop	28.5%	33.4%	35.1%	37.0%	42.0%
antlr	23.2%	26.2%	28.1%	29.1%	33.3%
pmd	28.9%	32.0%	35.4%	39.9%	47.3%
ps	78.2%	79.5%	79.8%	80.2%	90.7%
hsqldb	26.0%	27.9%	29.5%	31.7%	40.0%
jython	54.5%	55.4%	56.1%	56.5%	57.1%
xalan	0.4%	0.6%	2.8%	99.0%	99.5%
average	**31.3%**	**34.4%**	**36.4%**	**49.3%**	**55.3%**

Table 2. Percentage of repeated object accesses (hit ratios) for the entire collection

to be read exactly once during a collection, since if it was already traversed, its corresponding mark bit in the mark table would indicate that it should not be touched (read) again. This one-touch behavior creates a high miss rate, highly suitable for prefetching.

The cache-miss behavior of reference counting is not that simple to describe. To analyze the way a reference-counting collector accesses objects, we ran the following profiling on memory accesses. We considered each scan of an object and each update of a reference-count as a single memory access[6]. Each such access may cause a cache miss. We recorded the address of each such access into gc-log files, one log file per collection. Next, each gc-log file was analyzed in the following manner. For a given window size *w*, we checked for each access, if the same address was accessed during the last *w* (distinct) accesses, creating a cache hit[7]. For each benchmark, we outputted the fraction of hits, i.e., the fraction of repeating accesses within the window size, as a function of the window size.

The results appear in Table 2. We ran the above measurements with five window sizes (addresses): 100, 1000, 10000, 100000, and 1000000. The smaller windows are more representative of L1 cache-miss behavior, whereas the larger window sizes represent behavior with L2 cache sizes.

The measurement should be read as follows. If a 40% percentage appears in the 100 column of a benchmark's line, it means that 40% of the accesses were to memory locations that have been previously accessed during the last 100 accessed (distinct) addresses. A higher percentage means high cache hit ratio and a low potential for effective prefetching. For a tracing collector, the corresponding measurement would produce an all zeros table (since an object is not traversed twice during a collection).

[6] In this liberal measurement, we counted an object's scan as a single access, although it may have involved multiple accesses, e.g., because a large object may contain several pointer slots.

[7] We always consider a first access of an object in a collection as a miss.

It turns out that unlike tracing collectors, the repeated access with reference counting is quite high and the repeated accesses have temporal proximity. Hence, many memory accesses hit the L1 cache, making the prefetch a burden, or hit the L2 cache, making the prefetch less effective. Nevertheless, as seen in Table 1, properly inserted prefetching instructions can improve performance substantially. A further exploration (phases profiling) of this memory accesses behavior is provided in [19] for lack of space.

Usually there is correspondence between benchmark improvement percentages (Table 1) and repeated object accesses behavior (Table 2). However, the correspondence is not perfect as repeated object accesses behavior is not the only parameter influencing prefetch improvements. For example, if many benchmark's objects do not contain pointers, then prefetching an object (in order to read its pointers) is a waste. Hence, even if the repeated object accesses fraction of this benchmark is low, prefetching could not improve performance much in this case.

4.3 Prefetch strategy profiling

Table 1 presented the prefetch improvements achieved for the different reference-counting stages. However, two different prefetch strategies were implemented in both the **Process-ModBuffer** stage and the **Process-DecBuffer-and-Release** stage. Table 3 breaks the overall improvement into the shares of each particular strategy.

Columns 2-4 of Table 3 present the **Process-ModBuffer** stage, displaying the effect of the strategies *ModBuffer-traversal* and *delay-increment* (presented in Section 3.1). As can be seen, the *ModBuffer-traversal* strategy is the major cause for the prefetch improvement of the **Process-ModBuffer** stage. An investigation into the reasons for these improvement differences is provided in [19], due to lack of space.

Columns 5-7 of Table 3 present the **Process-DecBuffer-and-Release** stage, displaying the effect of the strategies *DecBuffer-traversal* and *object-release* (presented in Section 3.2). Here, the strategy responsible for most of the benefit is the *DecBuffer-traversal* strategy. This may be expected as the other prefetching strategy only applies to objects whose reference counts drops to zero. We have not further analyzed the objects' access behavior of the **Process-DecBuffer-and-Release** stage.

4.4 Hardware counters measurements

In order to better understand the effect of the inserted prefetch instructions, we have also measured several relevant hardware counters using PAPI (the Performance API [23]). These counters were measured during the garbage collection work of both versions of the reference-counting collector: with and without prefetching. Table 4 presents the difference of these counters between the versions for the entire garbage collection work. Columns 2-4 present the difference in the number of cycles stalled on any resource, the L2 load misses difference and the data TLB misses difference. Column 5 presents the overall garbage collection improvement (presented in Table 1).

It turns out that the strongest influence of the prefetching had been on the TLB misses rather than on the cache itself. The reason is probably that the prefetches were

Benchmarks	ModBuffer traversal improvement	delay increment improvement	Process ModBuffer improvement	DecBuffer traversal improvement	object release improvement	Process DecBuffer and Release improvement
jess	-5.5%	5.3%	-0.1%	-0.9%	-0.1%	-0.9%
db	-7.4%	-3.8%	-11.2%	-6.4%	-0.2%	-6.7%
javac	-9.2%	-3.0%	-12.2%	-6.8%	-1.6%	-8.4%
mtrt	-9.5%	-2.9%	-12.3%	-2.0%	-1.4%	-3.3%
jack	-13.8%	-2.5%	-16.3%	-4.3%	-1.7%	-5.9%
jbb	-7.9%	-0.3%	-8.2%	-5.6%	-0.9%	-6.5%
fop	-10.7%	-1.8%	-12.5%	-6.7%	-1.3%	-8.1%
antlr	-13.2%	-2.8%	-16.1%	-6.9%	-1.4%	-8.4%
pmd	-6.6%	-2.1%	-8.7%	-6.6%	-1.8%	-8.4%
ps	-4.0%	7.1%	3.0%	-3.4%	-0.1%	-3.7%
hsqldb	-12.5%	-6.3%	-18.8%	-10.0%	-1.0%	-11.0%
jython	-10.8%	1.4%	-9.4%	-0.1%	-0.4%	-0.5%
xalan	0.2%	2.1%	2.4%	-0.8%	-0.4%	-1.2%
average	**-8.5%**	**-0.7%**		**-4.7%**	**-0.9%**	

Table 3. A break of the prefetching improvement due to the two strategies involved in the Process-ModBuffer stage and the two strategies involved in the Process-DecBuffer-and-Release stage

issued too late and therefore only the TLB managed to gain some performance improvement. An attempt to issue the prefetch instructions earlier did not succeed due to the pay in temporary variables (and register pressure).

5 Related work

VanderWiel and Lilja [3] provide a detailed survey examining diverse prefetching strategies, such as hardware prefetching, array prefetching and other software prefetching.

Similarly to our approach, several previous studies proposed adding, by hand, prefetch instructions to specific locations in several garbage collectors. However, they all studied tracing collectors. Boehm [5] proposed prefetching objects that are pushed to the mark stack during the mark phase of a mark-sweep collector. This prefetch makes the first cache line of the object available later when popped from stack to be scanned. This prefetching strategy yields improvements in execution time, although suffering from prefetch timing problems: too early prefetches and too late prefetches. These timing problems were addresses by [6, 7]. Both suggested improved prefetching strategies to the mark phase by imposing some sort of FIFO processing over the mark stack, in order to control the time between the data prefetch and its actual access. In another related work, Cahoon [4] employs prefetching to improve the memory performance of a generational copying garbage collector.

Appel [24] emulates a write-allocate policy on a no-write-allocate machine by prefetching garbage before it is written (during its space allocation). Hence, the relevant cache line is allocated and the write (occurring during the object allocation) hits the cache.

Benchmarks	Cycles stalled	L2 cache misses	TLB misses	overall gc
jess	-8.8%	-1.1%	-17.3%	-1.8%
db	-4.3%	-0.7%	10.1%	-8.5%
javac	-17.4%	-1.8%	-6.6%	-12.3%
mtrt	-15.8%	-0.6%	-20.1%	-8.0%
jack	-20.0%	-3.2%	-21.1%	-10.8%
jbb	-14.8%	2.5%	-9.4%	-14.9%
fop	-6.6%	0.1%	-14.0%	-10.5%
antlr	-6.3%	0.1%	-15.1%	-14.6%
pmd	-9.6%	0.2%	-11.2%	-9.6%
ps	-2.0%	0%	-21.6%	-1.7%
hsqldb	-14.1%	-0.8%	-21.8%	-14.9%
jython	-4.8%	0.4%	-1.6%	-4.4%
xalan	-0.7%	0.2%	37.6%	-0.6%
average	**-9.6%**	**-0.4%**	**-8.6%**	

Table 4. Hardware counters measurements

6 Conclusions

We have studied prefetch opportunities for a modern reference-counting garbage collector. It turns out that several such opportunities typically exist and an implementation on the Jikes Research JVM demonstrates effectiveness in reducing stall times and improving garbage collection efficiency. In particular, the average garbage collection time was reduced by 8.7%.

Investigating the memory access patterns of the reference-counting collector, we found out that, unlike tracing collectors, objects are accessed repeatedly, reducing the potential benefit due to prefetching. These measurements explain the effectiveness of the various strategies at the various stages. Repetitive accesses to objects increase the hit rate and reduce the efficacy of prefetch insertions.

References

1. Callahan, D., Kennedy, K., Porterfield, A.: Software prefetching. In: ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, New York, NY, USA, ACM Press (1991) 40–52
2. Luk, C.K., Mowry, T.C.: Compiler-based prefetching for recursive data structures. In: International Conference on Architectural Support for Programming Languages and Operating Systems. (1996) 222–233 SIGLAN Notices 31(9).
3. VanderWiel, S.P., Lilja, D.J.: Data prefetch mechanisms. ACM Computing Surveys **32**(2) (2000) 174–199
4. Cahoon, B.: Effective Compile-Time Analysis for Data Prefetching in Java. PhD thesis (2002)
5. Boehm, H.J.: Reducing garbage collector cache misses. In Hosking, T., ed.: ISMM 2000 Proceedings of the Second International Symposium on Memory Management. Volume 36(1) of ACM SIGPLAN Notices., Minneapolis, MN, ACM Press (2000)

6. Cher, C.Y., Hosking, A.L., Vijaykumar, T.: Software prefetching for mark-sweep garbage collection: Hardware analysis and software redesign. In: Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA (2004) 199–210
7. van Groningen, J.: Faster garbage collection using prefetching. In C.Grelck, Huch, F., eds.: Proceedings of Sixteenth International WOrkshop on Implementation and Application of Functional Languages (IFL'04), Lübeck, Germany (2004) 142–152
8. Collins, G.E.: A method for overlapping and erasure of lists. Communications of the ACM **3**(12) (1960) 655–657
9. Levanoni, Y., Petrank, E.: An on-the-fly reference counting garbage collector for Java. In: OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications. Volume 36(10) of ACM SIGPLAN Notices., Tampa, FL, ACM Press (2001)
10. Levanoni, Y., Petrank, E.: An on-the-fly reference-counting garbage collector for java. ACM Transactions on Programming Languages and Systems **28**(1) (2006)
11. Azatchi, H., Levanoni, Y., Paz, H., Petrank, E.: An on-the-fly mark and sweep garbage collector based on sliding view. [25]
12. Blackburn, S.M., McKinley, K.S.: Ulterior reference counting: Fast garbage collection without a long wait. [25]
13. Paz, H., Petrank, E., Bacon, D.F., Rajan, V., Kolodner, E.K.: An efficient on-the-fly cycle collection. [26]
14. Paz, H., Petrank, E., Blackburn, S.M.: Age-oriented garbage collection. [26]
15. Deutsch, L.P., Bobrow, D.G.: An efficient incremental automatic garbage collector. Communications of the ACM **19**(9) (1976) 522–526
16. Alpern, B., Attanasio, C.R., Cocchi, A., Lieber, D., Smith, S., Ngo, T., Barton, J.J., Hummel, S.F., Sheperd, J.C., Mergen, M.: Implementing Jalapeño in Java. In: OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications. Volume 34(10) of ACM SIGPLAN Notices., Denver, CO, ACM Press (1999) 314–324
17. SPEC Benchmarks: Standard Performance Evaluation Corporation. http://www.spec.org/ (1998,2000)
18. DaCapo benchmark suite: The dacapo benchmark suite - version beta051009. (http://www-ali.cs.umass.edu/DaCapo/)
19. Paz, H.: Efficient Memory Management for Servers. PhD dissertation, Technion, Israel Institute of Technology, Department of Computer Science (2006)
20. Boehm, H.J., Demers, A.J., Shenker, S.: Mostly parallel garbage collection. ACM SIGPLAN Notices **26**(6) (1991) 157–164
21. Wilson, P.R., Johnstone, M.S., Neely, M., Boles, D.: Dynamic storage allocation: A survey and critical review. In Baker, H., ed.: Proceedings of International Workshop on Memory Management. Volume 986 of Lecture Notes in Computer Science., Kinross, Scotland, Springer-Verlag (1995)
22. Jones, R.E.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, Chichester (1996) With a chapter on Distributed Garbage Collection by R. Lins.
23. PAPI: The Performance API. (http://icl.cs.utk.edu/papi/overview/)
24. Appel, A.W.: Emulating write-allocate on a no-write-allocate cache. Technical Report TR-459-94, Department of Computer Science, Princeton University (1994)
25. OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications. In: OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications. ACM SIGPLAN Notices, Anaheim, CA, ACM Press (2003)
26. Proceedings of the 14th International Conference on Compiler Construction. In: Proceedings of the 14th International Conference on Compiler Construction, Edinburgh, Springer-Verlag (2005)