Linear-Mark: Locality vs. Accuracy in Mark-Sweep Garbage Collection

Chiara Meiohas
chiara@cs.technion.ac.il
Technion
Israel

Stephen M. Blackburn
steveblackburn@google.com
Google
Australia

Erez Petrank
erez@cs.technion.ac.il
Technion
Israel

ABSTRACT

Tracing garbage collectors are widely deployed in modern programming languages. But tracing an arbitrary heap shape incurs poor locality and may hinder scalability. In this paper, we explore an avenue for mitigating these inefficiencies at the expense of conservative, less accurate identification of live objects. We do this by proposing and studying an alternative to the Mark-Sweep tracing algorithm, called *Linear-Mark*. It turns out that although Linear-Mark improves locality and scalability, the accuracy of Mark-Sweep outweighs the achieved enhancements. We present the Linear-Mark garbage-collecting algorithm and provide an evaluation that highlights the trade-offs between the Linear-Mark and the Mark-Sweep approaches. Our hope is that this research will inspire further algorithmic improvements, ultimately leading to better garbage collection algorithms.

CCS CONCEPTS

• **Software and its engineering** → *Runtime environments*; **Garbage collection**; • **Computer systems organization** → Parallel architectures.

KEYWORDS

automatic memory management, garbage collection, parallel garbage collection, Mark-Sweep garbage collector

1 INTRODUCTION

Managed programming languages have become highly popular and are widely used in various domains, from Big-Data platforms to mobile computing. They reduce the likelihood of memory-related bugs and improve the development process. Managed environments rely on a garbage collector that frees unneeded allocated memory.

A popular method for reclaiming unneeded memory is by use of tracing collectors. Mark-Sweep garbage collectors [26] reclaim unused memory by tracing and marking objects reachable from the program roots (the marking phase) and then reclaiming all unmarked objects (the sweep phase). Tracing collectors are employed in one way or another in most existing commercial environments [13, 16, 19, 25]. Most of the garbage collection time is spent in the marking phase, especially because it traverses the heap in a non-sequential order, which incurs higher rates of cache misses. In this paper, we study a variant of the Mark-Sweep algorithm, called *Linear-Mark*, which reduces the cache miss rate during the marking phase at the cost of relaxing marking accuracy in a conservative manner. Namely, we improve locality while ensuring that we never reclaim a reachable object, but we may fail to reclaim some of the unreachable objects.

We propose the Linear-Mark garbage collector, which replaces the graph-traversing marking phase of the Mark-Sweep algorithm with a linear-marking phase, that goes linearly over all objects in the heap. The sweep phase is left untouched, reclaiming all objects that are not marked in the marking phase (no matter whether Mark-Sweep or Linear-Mark was previously executed). The main idea of linear marking is to make a single sequential pass over the heap, in which it visits each allocated object in the the heap sequentially and marks their direct descendants. This marking is conservative; the marked objects are a super-set of the reachable objects. It will tend to mark some unreachable objects, simply because they are directly reachable from allocated objects, even though the allocated objects may themselves be unreachable. However, going sequentially over the heap and marking descendants on a separate small bitmap improves locality. In this paper we study this trade-off between locality and accuracy of the collection.

To understand the inaccuracy of the marking phase, we note that this variant of Mark-Sweep inherits properties of reference counting [12]. Reference counting is a reclamation method that maintains for each object a count of the pointers that reference the object. When a reference count of an object goes down to zero, the object is clearly not reachable and may be reclaimed. When objects are reclaimed with the reference counting algorithm, the reference counts of their descendants are decremented, and descendants may be reclaimed recursively. Our Linear-Mark identifies exactly the objects that have a zero reference count, with the notable difference that it does not use recursive reclamation of descendants of deleted objects because such a recursive deletion would foil the good locality of the Linear-Mark. Consequently, Linear-Mark is less precise and more conservative than reference counting.

Since the Linear-Mark does not reclaim all unreachable objects, the heap may be filled with unclaimed objects, similarly to cycles that may fill and exhaust the heap with a reference counting collector. Therefore, (similarly to reference counting) we occasionally invoke the standard marking procedure (from the Mark-Sweep algorithm) in order to accurately identify all reachable objects and adequately reclaim all other objects in order to reduce the heap size to exactly its live object set.

The scalability of garbage collectors is hard to obtain. While real-world architectures become more and more parallel with a growing number of cores, production garbage collectors are not able to scale to all available cores. Typically, beyond a predetermined core threshold, garbage collectors use fewer collector threads than the number of cores simply because they cannot efficiently use many cores. For example, in the Parallel Collector of HotSpot VM [27] and in G1 [3, 13], the default number of stop-the-world collector threads is 5/8 of logical processors when there are more than eight logical processors. On very large systems, the default

is 5/16 of logical processors. It is shown by Gidra et al. [18] that the OpenJDK 7 collector's performance degrades when using more than eight collector threads. Consequently, significant efforts have been dedicated to improve the scalability of garbage collectors on multicore platforms.

In light of the collection scalability difficulty, it is interesting to note that Linear-Mark is highly scalable, and the evaluation demonstrates a higher speedup for Linear-Mark compared to Mark-Sweep. For Mark-Sweep collectors, the scalability of the marking phase depends on the scalability of the heap graph's shape [2]. In contrast, the sweep phase is easily parallelized: the heap can be divided into regions, and different collector threads sweep different regions. Linear-Mark can be similarly parallelized: each collector thread linear-marks its assigned region. Linear-Mark's marking phase does not depend on the heap shape and is highly parallelizable.

To evaluate Linear-Mark, we compare Linear-Mark's performance to the original Mark-Sweep. And in order to understand the impacts of the various properties of the two collectors, we further measure various other properties on top of their performance. In particular, we evaluate their cache miss rate, their scalability, and handling of the different benchmarks. We compare the collectors using the DaCapo benchmark suite [6, 7] and a specialized benchmark that demonstrates the good benefits of Linear-Mark. Our general findings show that Mark-Sweep has shorter collection times than Linear-Mark on the DaCapo benchmarks. However, Linear-Mark achieves lower cache miss rates and higher parallelism. It turns out that the accuracy of the collection is very important. While the Linear-Mark collector does not outperform Mark-Sweep (except for specialized cases) we hope the ideas and analysis provided here may trigger subsequent research and may be useful to better understand the properties of tracing collectors and their value.

Organization. In Section 2 we describe the background for this work. In Section 3 we present the Linear-Mark algorithm, and discuss its advantages and disadvantages with respect to Mark-Sweep. In Section 4 we present a Synthetic micro-benchmark that exploits the benefits of the Linear-Mark collector. In Section 5 we discuss the implementation, and an evaluation is provided in Section 6. Additional related work is reviewed in Section 7, and we conclude in Section 8.

2 BACKGROUND

Garbage collectors identify and free memory that is no longer in use by the program. In this paper we study the Mark-Sweep garbage collector. We concentrate on stop-the-world parallel execution of the garbage collector. Namely, when memory is running out, the program is paused to run the collector to free memory. Multiple collector threads execute the collection in parallel, and when the collection completes, the program resumes.

2.1 Mark-Sweep

Mark-Sweep garbage collectors [26] trace all objects reachable from the program roots (similar to a graph traversal) and mark all *reachable* objects (also called *live* objects). Once finished tracing, the collector moves into the sweep phase and scans the heap to reclaim the space of all unmarked objects. These objects are unreachable from the roots (also called *dead* objects). The reclaimed space is then used for subsequent allocations.

Algorithm 1 Linear-Mark Algorithm

```
1:  procedure MarkRoots( )
2:      for objects in Roots do
3:          if !IsMarked(object) then
4:              setMarked(object)
5:
6:  procedure LinearMark(start, end)
7:      ref = start
8:      while ref < end do
9:          if isAllocated(ref) then:
10:             for fld in Pointers(ref) do
11:                 child = *fld
12:                 if !IsMarked(child) then
13:                     setMarked(child)
14:         ref = getNextWord(ref)
15:
16: procedure Sweep(start, end)
17:     scan = start
18:     while scan < end do:
19:         if !IsMarked(scan) then
20:             free(scan)
21:         else
22:             unsetMarked(scan)
23:         scan = getNextWord(scan)
24:
25: procedure LinearMarkCollection( )
26:     MarkRoots( )
27:     LinearMark(heap.start, heap.end)
28:     Sweep(heap.start, heap.end)
```

2.1.1 Marking Locality. Locality of execution in the marking phase is typically poor as tracing traverses objects in arbitrary memory order, depending on the shape of the heap objects graph. Garner et al.[17] showed that the principal bottleneck of the marking phase is poor locality. Boehm [9] and Garner et al. [17] proposed to use prefetching, Cohen and Petrank [11] presented a data structure-aware garbage collector, which improves the locality of the trace when tracing objects that belong to data structures.

2.1.2 Marking Parallelism. The scalability of the marking phase was studied in the literature [2, 30], showing that parallelism of the trace may be limited, depending on the shape of the live-objects graph in the heap. As the marking phase performs a graph traversal over the live objects in the heap, the workload of the collector threads may be unbalanced. For example, if the live-objects graph is a single linked-list, then only one collector thread can mark the live objects while other collector threads remain idle. This happens because one single collector thread discovers one descendant object at a time and does not have any additional work to share with other concurrent collector threads. In general, a deep and narrow live-objects graph may cause the marking phase to be not scalable, and the collector threads workload to be unbalanced. It was shown

in [2] that some of the Dacapo benchmarks [6, 7] have live-objects graphs with low scalability.

2.1.3 Sweeping. Unlike the marking phase, the sweeping phase has good locality and good scalability, which often makes the time spent on sweep a small part of the garbage collection process. The memory accesses when sweeping tend to be more sequential, which achieves higher locality. Due to the ease of splitting sweep work, sweep can be easily executed eagerly (at one shot at the end of the marking phase) or lazily (incrementally, when allocation requires). An eager sweeping can be easily parallelized by letting different collector threads sweep different address ranges of the heap. The areas are independent of one another, and high scalability is, therefore, easy to achieve. Lazy sweeping [21] can be easily organized by letting each mutator sweep an unswept area when space is needed. With lazy sweeping, the sweep time is distributed among the mutators and across time rather than done in a batch right after marking. Lazy sweep may incur better locality as an area is swept just before parts of it are allocated and used.

In this work, we refer mainly to the marking phase, which we will replace with a linear mark. The sweeping phase may be done eagerly or lazily. The important takeaway is how simply eager-sweeping can be parallelized, as this is what motivated Linear-Mark and the high scalability it achieved.

2.2 Reference Counting

Reference counting (RC) [12] collectors do exactly as the name suggests: they count references to objects. Naive RC stores the number of references to each object and collects an object if its count falls to zero. When an object is reclaimed, the reference counts of its children are decreased, which may cause additional object reclamations and recursive reference count updates. A write barrier is used to always store the correct reference count: when a pointer change is detected the previously referenced object's count is decreased, and the newly referenced object's count is increased.

There are two main drawbacks to Naive RC: it does not collect cycles data structures, and it has high mutator overhead. We will not discuss the latter limitation, but many optimizations were achieved to solve this issue [15, 22–24]. The reference count of objects in a cyclic structure is at least one; therefore, RC alone cannot reclaim such structures. A simple solution is to run an occasional tracing collector [14] to collect the floating garbage.

3 LINEAR MARKING

3.1 Linear-Mark algorithm

The Mark-Sweep algorithm consists of two phases: in the marking phase all objects transitively reachable from the roots are marked, and in the sweep phase all unmarked objects are reclaimed. The Linear-Mark algorithm modifies the marking phase. Instead of traversing the live objects in a DFS-style traversal, which is not cache-friendly and not always scalable, Linear-Mark simply marks all objects that are *directly* referenced by allocated objects, or from the roots. This can be done by marking the roots, and then going over the heap in address order and marking the children of each allocated object. The sweep phase remains the same as in mark-sweep.

Note the difference between Mark-Sweep and Linear-Mark: Mark-Sweep marks only live objects in the heap according to the traversal order. Linear-Mark marks all reachable objects, but it tends to mark additional objects. In particular, Linear-Mark will mark unreachable allocated objects that are referenced by other unreachable allocated objects. While this would prevent us from collecting some unreachable objects, it will never cause the reclamation of a live object. Thus, the correctness of the execution is still guaranteed, but the accuracy (and therefore efficiency) of the collection decreases. We discuss this inaccuracy below.

There are two advantages to linear marking that may justify the conservative (reduced) accuracy of the marking phase. First, the marking is sequential. It goes over the heap in address order and therefore incurs better locality. While making a heap pass sequentially, the marking phase also accesses two bitmaps: the alloc bitmap, which identifies the location of each known object in the heap, and the mark bitmap is used to mark live objects. Both tables hold one bit per heap word, and hence the size of each table is 1/64 the size of the heap. Linear-Mark traverses both the heap and the alloc bitmap in a sequential manner which improves locality significantly. Linear-Mark accesses the mark bitmap in an arbitrary order, depending on the descendants of each encountered object. But since the bitmap is small, this does not have a strong detrimental effect on the locality of the entire linear marking phase. The evaluation shows that the cache miss rate during the traversal is reduced on all cache levels (L1, L2, and L3).

The second advantage of linear marking is that it is easily parallelizable, in contrast to a regular (DFS-style) marking that may be impossible to scale, depending on the shape of the live-objects graph in the heap. Live-object graphs that are long and narrow in shape may result in poor scalability [2, 30]. In contrast, linear marking is easy to parallelize: different collector threads can simply scan separate regions in the heap. The heap can be divided into regions, and each of these smaller regions can be traversed independently of the other regions, except for an easy synchronization of writing to the mark bitmap. Thus, scalability of Linear-Mark is better than a regular marking traversal.

Algorithm 1 shows the pseudo-code of a Linear-Mark collector. Figure 1 briefly illustrates how the linear-marking phase works. Note the difference between Mark-Sweep and Linear-Mark: Mark-Sweep accesses only live objects in the heap according to the traversal order. Linear-Mark accesses all allocated objects in the heap, but sequentially.

3.2 Correctness

Linear-Mark never collects a live object. An object O is live by definition if transitively reachable by a program root. Therefore O is reachable by a path that starts with a program root and consists of allocated objects. As Linear-Mark marks all objects referenced by allocated objects, it will mark all objects in this path, including O.

3.3 Accuracy of the collection

The disadvantage of linear marking is its inaccuracy: it conservatively marks objects. To understand the inaccuracy in the identification of reachable objects, we look at the unreachable objects

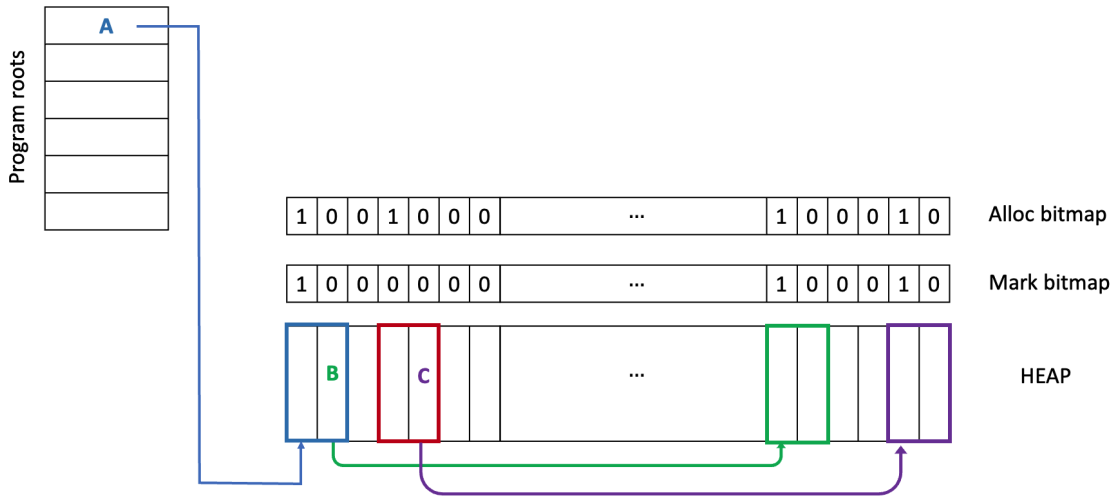


Figure 1: Linear marking phase. Linear-Mark uses a side alloc bitmap and a side mark bitmap. Objects pointed by the program's roots and children of allocated objects are marked.

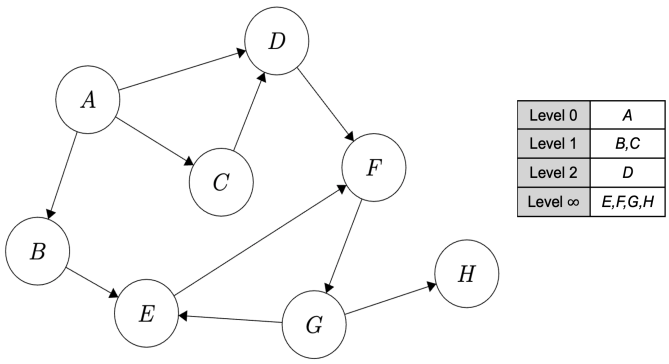


Figure 2: Example of the levels in a directed graph.

graph from a reference-counting point of view. Linear-Mark does not maintain reference counts, but just for this analysis, let us consider the reference counts as if we knew them for each object. It can be easily seen that linear marking marks exactly all objects with non-zero reference count. This is because objects with zero reference count do not have any objects referencing them in the heap and are thus not marked by linear mark, whereas objects with non-zero reference counts do have objects (or roots) referencing them, which will cause the Linear-Mark to mark them. This means that the Linear-Mark garbage collector (with a regular sweep) will reclaim exactly all objects that have zero reference count.

Like Reference-counting, Linear-Mark does not reclaim unreachable cycles (or strongly connected components). To address this limitation in practice, Linear-Mark will periodically invoke the regular Mark-Sweep garbage collector to target cycles and any floating garbage that Linear-Mark has not been able to reclaim.

To further explain the inaccuracy of the linear marking, let us examine the graph of unreachable objects at some specific collection. We define the level of an object O to be the length of the longest path from an object to O. Formally, denote by $O' \rightsquigarrow O$ a directed path (of pointers) from an object O' to an object O. And denote the length of this path by $\text{len}(O' \rightsquigarrow O)$. For any unreachable node O, we define $\text{level}(O) = \max\{\text{len}(O' \rightsquigarrow O) : O' \text{ is alloacted and unreachable}\}$. If the above set is empty, we say that $\text{level}(O) = 0$. Objects on Level 0 do not have any incoming edges and are not referenced by any other dead object. Objects that are part of strongly connected components (SCC), e.g., objects in a cycle, and objects that are reachable from an SCC are said to have level ∞. An example of a graph and the levels of its nodes is shown in Figure 2.

Using the levels terminology, Linear-Mark will collect only the objects on Level 0 in the unreachable objects graph. But note that in the next Linear-Mark collection, objects in level 1 in the previous collection can be reclaimed now, together with objects currently in level 0. Whenever we reclaim level 0, the objects of level 1 move to level 0. The objects on level ∞ will never be collected (ex. cycles will never be collected). When many objects are on levels higher than 0, the collection accuracy decreases, resulting in floating garbage. If most objects are on level 0, the collection accuracy is higher, and Linear-Mark collects most unreachable objects.

Evaluation with standard benchmarks shows that most objects are in level 0, but a non-negligible fraction of them are in higher levels. We occasionally run the Mark-Sweep garbage collector to collect cycles and floating garbage leftover from the previous Linear-Mark collections.

4 A SYNTHETIC MICRO-BENCHMARK

To demonstrate the full potential of linear marking we build a simple benchmark that highlights its advantages. The benchmark simply executes inserts and deletes to a linked list. Over time, the location of the nodes in the linked list becomes arbitrary, making a normal (DFS-style) marking that needs to traverse the linked list become cache-unfriendly. Also, scaling a regular mark is difficult,

because the tracing of a linked-list cannot be parallelized. Each scan of an object discovers a single descendant, that can then be scanned by a single thread.

We further nullify the next pointer of a deleted node, to make sure that it does not point to a node that may later become deleted. This assures the accuracy of a Linear-Mark is as high as possible because a dead object that is unlinked from the list is not referenced by any object. The resulting benchmark is very simple and is meant to manifest the advantages of Linear-Mark.

The synthetic benchmark starts by building a substantial-sized linked list, and it then simply repeatedly inserts a random key into the list and then removes a random key from the list. As shown in the evaluation, on this synthetic benchmark Linear-Mark outperforms a highly optimized tracing collector, and it also scales better.

5 METHODOLOGY

5.1 Hardware and Operating System

We use the Intel Xeon Gold 6338 processor, with a 2GHz clock and two sockets with 32 physical cores each (64 physical cores overall). Each core has a 32KB, 8-way, 64B line L1 instruction cache, 48KB, 12-way, 64B L1 data cache (overall 2MB L1i cache and 3MB L1d cache), and a 1.25MB, 20-way, 64B line L2 cache (overall 80MB L2 cache). Each socket has a shared 48 MB, 12-way, 64B line L3 cache (overall 96MB). The machine runs Ubuntu 20.04.5 and Linux 5.4.0-136-generic kernel.

5.2 OpenJDK and MMTK

Our evaluations on Java use Openjdk 11.0.17+8. We implement Linear-Mark in the Memory Management toolkit (MMTK) [4, 5], using the OpenJDK binding. We will make the code publicly available once anonymity is lifted. Our evaluations use Immix [8], a highly optimized Mark-Region collector with opportunistic evacuation. We do not describe the details of the Immix algorithm, but it is a tracing algorithm that sometimes evacuates objects from specific blocks in the heap (defragmentation). Because it is a tracing algorithm, we could implement Linear-Mark on top of it.

5.3 Linear-Mark configuration

Linear-Mark was implemented on top of the Immix [8] collector available in MMTK. Immix, like a Mark-Sweep collector, has a marking phase and a sweep phase. We implemented the Linear-Mark instead of Immix's trace, and the sweep phase remained untouched. In addition to the metadata Immix stores, Linear-Mark also uses an alloc bitmap, which identifies the location of each known object in the heap at word granularity. If the current collection is a defrag collection, we let Immix run instead of Linear-Mark.

In Linear-Mark, each worker thread takes a chunk of the heap and searches for allocated objects to mark the referents (children) as alive. The size of the chunk determines the unit of work processed by each thread, which affects load balancing, and may determine the overall performance of the linear marking phase of the collector. If we have a small heap with large chunk sizes, only a few worker threads will work during the linear marking phase. But if the chunks are too small, there may be higher contention as workers race to acquire chunks of work. We found that 32KB chunk size worked well for the workloads we evaluate. We used that size in the results we report here.

5.4 Benchmarks

We evaluate Linear-Mark and Immix using the DaCapo benchmark suite [6, 7, 20] and the synthetic micro-benchmark described in Section 4. We run each benchmark five times and average the results; this average is represented in the figures.

5.4.1 DaCapo benchmark suite. We used a snapshot of the Chopin release [20] of the DaCapo benchmark suite [6, 7]. We use 16 benchmarks of the DaCapo suite: *tradebeans* and *tradesoap* are omitted because they often fail on this version of OpenJDK. We also exclude *batik* as it does not produce any collections when using 4× the minimum heap of Immix.

For time-sensitive measurements, we perform three warmup iterations and collect information on the 4th iteration using the DaCapo harness. We averaged the results over five invocations. As discussed in the introduction, running Linear-Mark solo is not possible on many benchmarks because failing to reclaim a fraction of the objects does sooner or later exhaust the heap. For example, in *lusearch*, *tomcat*, *graphchi*, and *cassandra*, more than half of the dead objects are marked as live (as shown in Figure 7). While other benchmarks allow more object collection, we adopt a strategy that holds for all benchmarks and execute Linear-Mark and Mark-Sweep alternately. Mark-Sweep (of Immix) reclaims cycles and floating garbage left from the previous Linear-Mark collection.

5.4.2 The Synthetic benchmark. As detailed in Section 4, We implemented a synthetic benchmark that highlights the benefits of Linear-Mark. The benchmark uses four mutator threads that operate over a shared linked list, inserting and removing nodes in the list for randomly selected keys.

Linear-Mark does not require an occasional backup tracing collector to clear floating garbage in this micro-benchmark because it reclaims all unreachable objects. Recall Linear-Mark does not support defragmentation (opportunistic evacuation) collections; therefore, we disabled Immix defragmentation. This assures that all collections run the Linear-Mark algorithm when using the Linear-Mark collector. We conduct the evaluation using the following parameters: 4 mutator threads, a linked list with 2^{16} nodes, each mutator executed 2^{20} random insertions and deletions and the heap size was 52MB. We ran Immix and Linear-Mark with 1, 4, 8, 16, and 32 collector threads.

For each run, we perform three warmup iterations and collect the information on the 4th iteration. We then average the data over five invocations.

6 RESULTS

In this section, we provide an evaluation of the Linear-Mark collector. Its benefits are demonstrated on the synthetic micro-benchmark in Section 6.1. The evaluation on real benchmarks appears in Section 6.2. In each of these sections, we start by evaluating the locality behavior of the mark phase via the cache miss rates for both Immix and Linear-Mark. Next, we study the speedup of the two collectors over a growing number of collector threads, to evaluate the scalability of Linear-Mark against the scalability of Immix. Then,

we report the performance of the two collectors, and finally, we study the inaccuracy of Linear-Mark by checking the BFS-levels of the dead objects graph. The latter signifies how much time it will take to reclaim objects with Linear-Mark. Objects at level 0 are reclaimed immediately by Liner-Mark, whereas objects at level 1 are reclaimed at the subsequent collection cycle, after objects of level 0 are reclaimed. In general, the reclamation of objects at level i is delayed for i collection cycles. To understand the real behavior of the dead objects, we measure the levels under the exact identification of dead objects by the mark phase of Immix. Using Linear-Mark to investigate the dead objects graph would create artificial results due to the examination of floating garbage of levels higher than 0 from previous collections.

6.1 Synthetic Micro-Benchmark

When running Linear-Mark on this benchmark, no backup Immix collections are required, as the benchmark design assures Linear-Mark collects all unreachable objects.

Miss rates (%)	**Linear-Mark**	**Immix**
L1 miss rate	2.4	1.5
L2 miss rate	22.82	61.7
L3 miss rate	18.25	4.68

Speedup	**Linear-Mark**	**Immix**
4 threads	2.3	1.52
8 threads	3.45	1.65
16 threads	4.872	1.75
32 threads	7.07	1.46

Table 1: Evaluation of the synthetic micro-benchmark. The first table shows the L1, L2 and L3 cache miss rates of Linear-Mark and Immix. The second table shows the speedup of the collectors when using 4, 8, 16 and 32 collector threads.

6.1.1 Cache Misses. We examine the cache miss rates of Linear-Mark compared to those of Immix. We used performance counters to get L1, L2, and L3 cache misses and references. For L1 cache misses, we calculate only the L1 data load cache misses, as there isn't a perf counter for the L1 store misses on our machine. We analyze the miss rate (misses/references %) of the marking phase of Immix and the linear-marking phase of Linear-Mark.

The most relevant parameter here is the L2 miss rate, for which Linear-Mark has a dramatic decrease to about a third of the cache miss rate of Immix. This is where locality kicks in. The L1 cache is small, and therefore the marking of the descendant objects in an external mark bit-table (which incur many cache misses for such a small cache) makes the mark phase of Linear-Mark as prone to cache misses as any other collector. The linear traversal of the large heap is not effective in this case. Both collectors' L1 cache miss rate is very small, below 2.5%. As for L3, its size is 48MB on each socket, 96MB cache overall. The specific heap size of 52MB used here is favorable to Immix since it allows the entire live objects and external bit tables to fit into the cache. For Linear-Mark this specific size puts the heap plus additional external tables outside the L3 cache, which implies more cache misses. The same phenomenon is not typical to real-world benchmarks, as shown in Section 6.2.

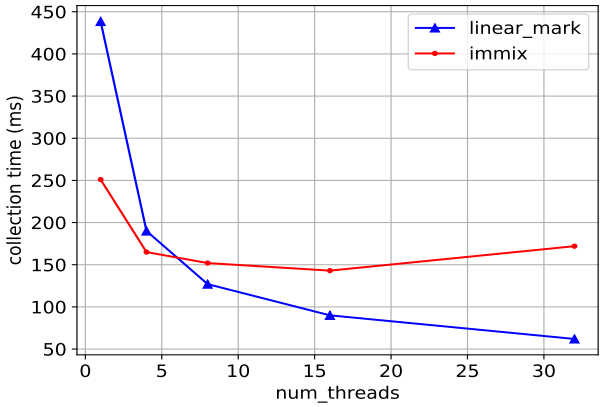


Figure 3: Collection time of Linear-Mark and Immix on the synthetic micro-benchmark.

6.1.2 Parallelism. The scalability of Linear-Mark is evident in this benchmark; the collection time improves the more the collector threads (see Figure 3). We measure the speedup of the collectors on this benchmark: the speedup of a collector with x threads is the collection time on one thread divided by the collection time on x threads. Therefore the speedup value of any collection on one collector thread is always 1. We present the speedup of both collectors in Table 1. With 32 collector threads, Linear-Mark achieves a 7× speedup, whereas Immix achieves only a 1.46× speedup. Immix does not achieve more than a 1.75× speedup on any of the collector threads, which supports the claim that linked lists are not scalable heap shapes [2].

6.1.3 Performance. Figure 3 illustrates the collection time of Immix and Linear-Mark when using 1, 4, 8, 16, and 32 collector threads. There are two notable observations: (i) when using eight or more collector threads, Linear-Mark performs better than Immix, (ii) Linear-Mark's collection time improves when using more threads, but Immix's collection time does not improve when using more than four collector threads.

Linear-Mark has slower collection times than Immix when using 1 and 4 collector threads, but on 8, 16, and 32 threads, it performs better than Immix. Linear-Mark linearly scans the heap during the collection, whereas Immix traces only the live objects; consequently, Linear-Mark is slower when using a few collector threads because each collector thread must scan larger chunks of the heap. Additionally, Table 1 shows Immix has better L1 and L3 cache miss rates. Therefore, Immix performs better than Linear-Mark for a few collector threads, but when many collector threads are available, Linear-Mark outperforms Immix.

6.1.4 Conclusion. Linear-Mark shows better performance than Immix on this benchmark when using eight threads or more, which confirms Linear-Mark's high parallelism. Although the results achieved for the DaCapo benchmarks are dissatisfying, the results in this section support the design inspiration.

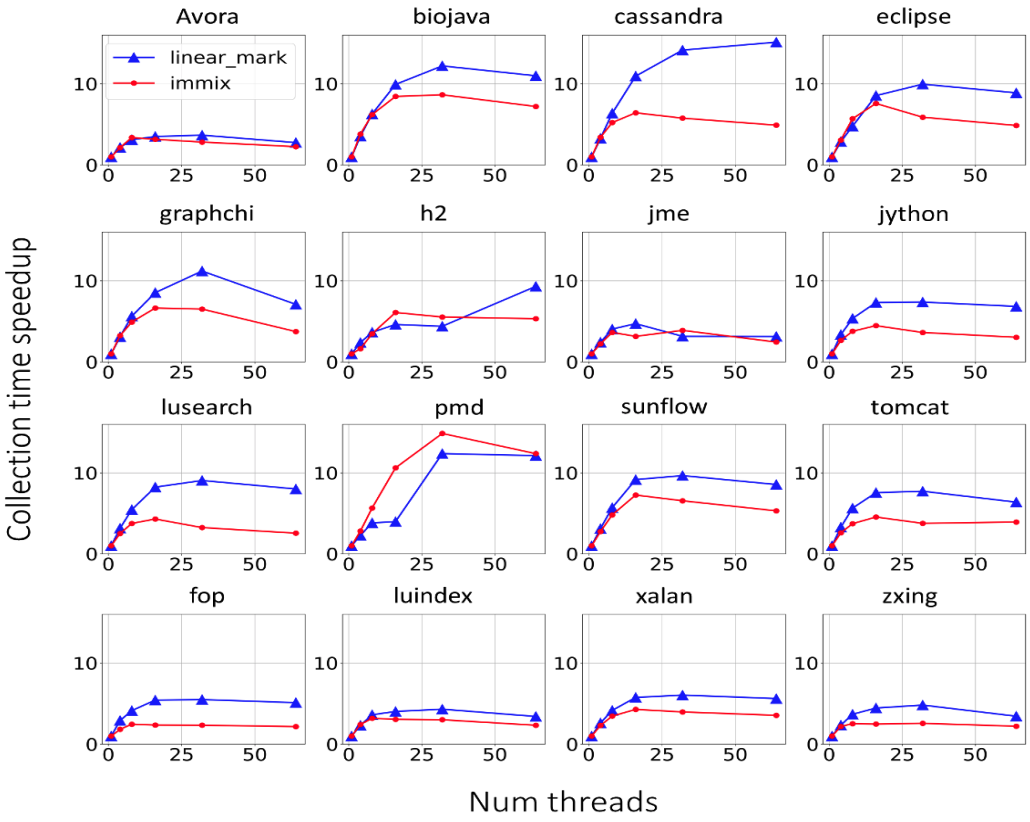


Figure 4: Collection time speedup on Linear-Mark and Immix on the DaCapo benchmarks.

6.2 DaCapo Benchmarks

In this section, we report the evaluation of Linear-Mark against Immix on the Dacapo benchmark suite.

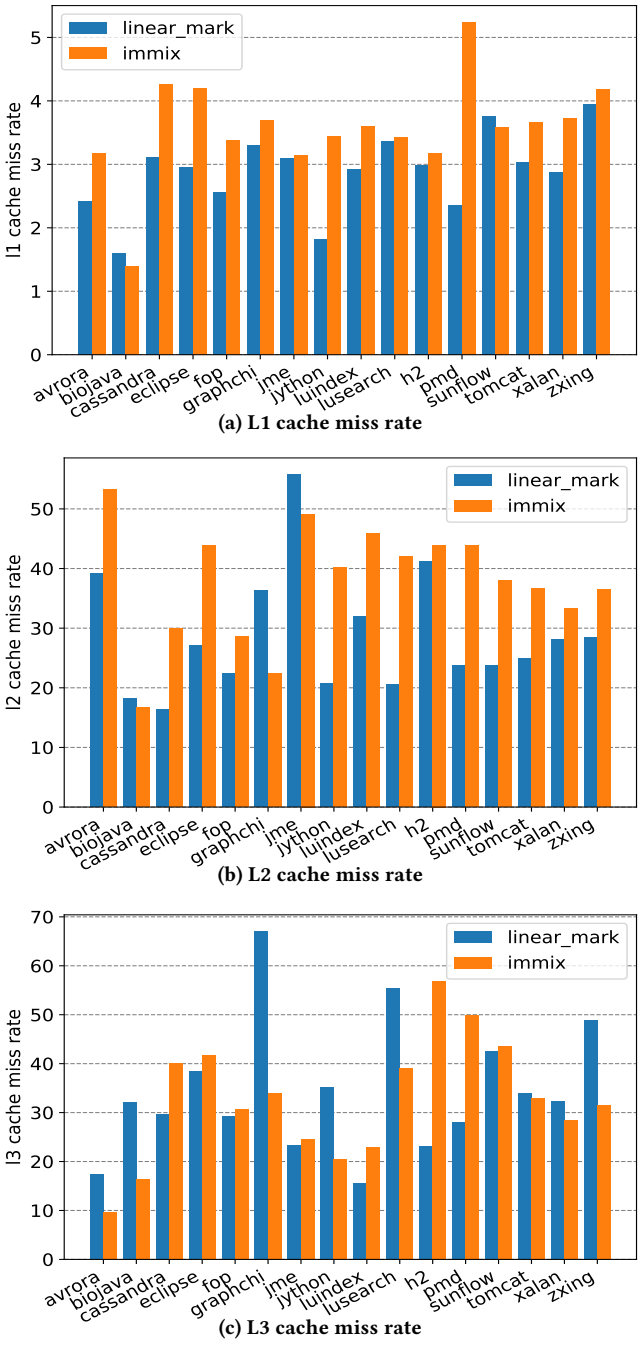
6.2.1 Cache Misses. We start with a comparison of the locality for the two collectors. When running Linear-Mark, we run the Linear-Mark collector and Immix collector alternately, in the hope of using Linear-Mark to improve locality and scalability (and therefore efficiency) and then using Mark-Sweep to prevent the growth of the space used by floating garbage. We run both collectors on 4x the minimum heap size for all benchmarks.

We examine the cache miss rates of Linear-Mark compared to those of Immix. We used performance counters to get L1, L2, and L3 cache misses and references. For L1 cache misses, we calculate only the L1 data load cache miss rates. We analyze the miss rate (misses/references %) of the marking phase of the collectors as the sweep phase is identical. When evaluating the miss rates of the Linear-Mark collector, we average the miss rates of the Linear-Mark collections only (and not the Immix collections). When evaluating the miss rates of Immix, we average the miss rates of all the collections. The miss rates were essentially the same regardless of the number of threads. As a result, we present the cache miss rates when running both collectors with 32 collector threads.

Figure 5a illustrates the L1 cache miss rate of Linear-Mark and Immix. On most benchmarks, Linear-Mark has lower cache miss rates than Immix. It is worth noting that the L1 cache miss rate of about 3% for Linear-Mark means that L2 and L3 cache misses may only happen for the 3% of the memory accesses that were not served by L1. Of these memory accesses, on almost all benchmarks Linear-Mark has better (lower percentage of) cache misses on L2 and on L3. Linear-Mark has higher L1 miss rates in *biojava* and *sunflow*, higher L2 miss rates in *jme* and *biojava*, and higher L3 cache miss rates in *avrora*, *biojava*, *graphchi*, *lusearch* and *zxing*. The reason for the lower L3 cache miss rates of Immix on *lusearch* and *zxing* is that the average size of the live objects is significantly smaller than the L3 cache. However, the overall heap size is considerably bigger than the L3 cache, which causes more cache misses in Linear-Mark.

6.2.2 Parallelism. Next, we examine the scalability of Linear-Mark compared to Immix on the benchmarks. We compare the speedup of both collectors using $x \in \{1, 4, 8, 16, 32, 64\}$ threads. When running Linear-Mark, we run the Linear-Mark collector and Immix collector alternately. We ran the collectors on 4x the minimum heap size of Immix (see Table 2).

Figure 4 illustrates Immix and Linear-Mark's speedup on various threads on all benchmarks. On all benchmarks except *h2, jme* and *pmd* Linear-Mark clearly shows higher scalability. For *jme* the two collectors scale at about the same pace, for *h2* Linear-Mark only wins on 64 threads, and with *pmd* Immix scales better until 64 threads. For the other benchmarks, Linear-Mark clearly scales



(a) L1 cache miss rate

(b) L2 cache miss rate

(c) L3 cache miss rate

Figure 5: Cache miss rates during the marking phase of Immix and the linear-marking phase of Linear-Mark (both collectors used 32 collector threads).

better, where the advantage is especially noticeable when using a higher number of collector threads.

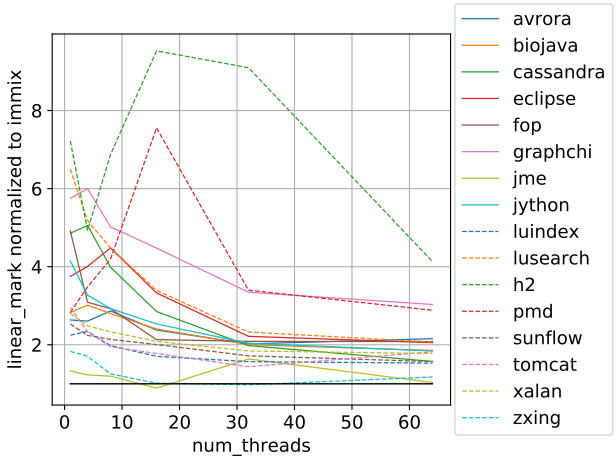


Figure 6: Linear-Mark collection time normalized to Immix collection time.

6.2.3 Performance. We evaluate Linear-Mark and Immix's collection time on the DaCapo benchmarks. The settings are the same as in Section 6.2.2.

The performance of Linear-Mark is lower than Immix. Figure 6 illustrates the Linear-Mark collector time normalized to Immix's collector time. The black line is 1; when the ratio is higher than 1, above the black line, Immix outperforms Linear-Sweep. These results show that Immix outperforms Linear Mark on almost all benchmarks.

Linear-Mark shows very long collection times compared to Immix on *h2* and *pmd*. These two benchmarks have the largest minimum heap: *h2* requires 2G min heap, and *pmd* requires almost 1G min heap; therefore, we run these benchmarks with an 8G heap and a 4G heap (see Table 2), respectively. Linear-Mark performs poorly on these benchmarks when using a few collector threads, but the gap reaches its minimum when using 64 collector threads. We can conclude that Linear-Mark should use the most collector threads available when the heap size is huge.

The gap between Linear-Mark and Immix is almost always (except for *pmd, h2* and *graphchi*) the highest when running a single collector thread. However, this gap decreases as more threads are utilized for the collection. This further confirms the results shown in Section 6.2.2: Linear-Mark is more scalable than Immix.

While Linear-Mark has better scalability and locality, the inaccuracy of identifying the unreachable objects is detrimental to the performance of Linear-Mark. It should be noted that Immix is a highly optimized collector, whereas Linear-Mark is an addition that could be further optimized. However, the difference in performance seems large enough to indicate that the implementation issue is not the main factor.

6.2.4 Inaccuracy of the Collection. Recall that we defined level(O) as the length of the longest path from any object to O. This section analyzes the levels of the graph spanned by the unreachable objects of a collection. Any unreachable object can only have paths leading to it from other unreachable objects. Thus, the level of an

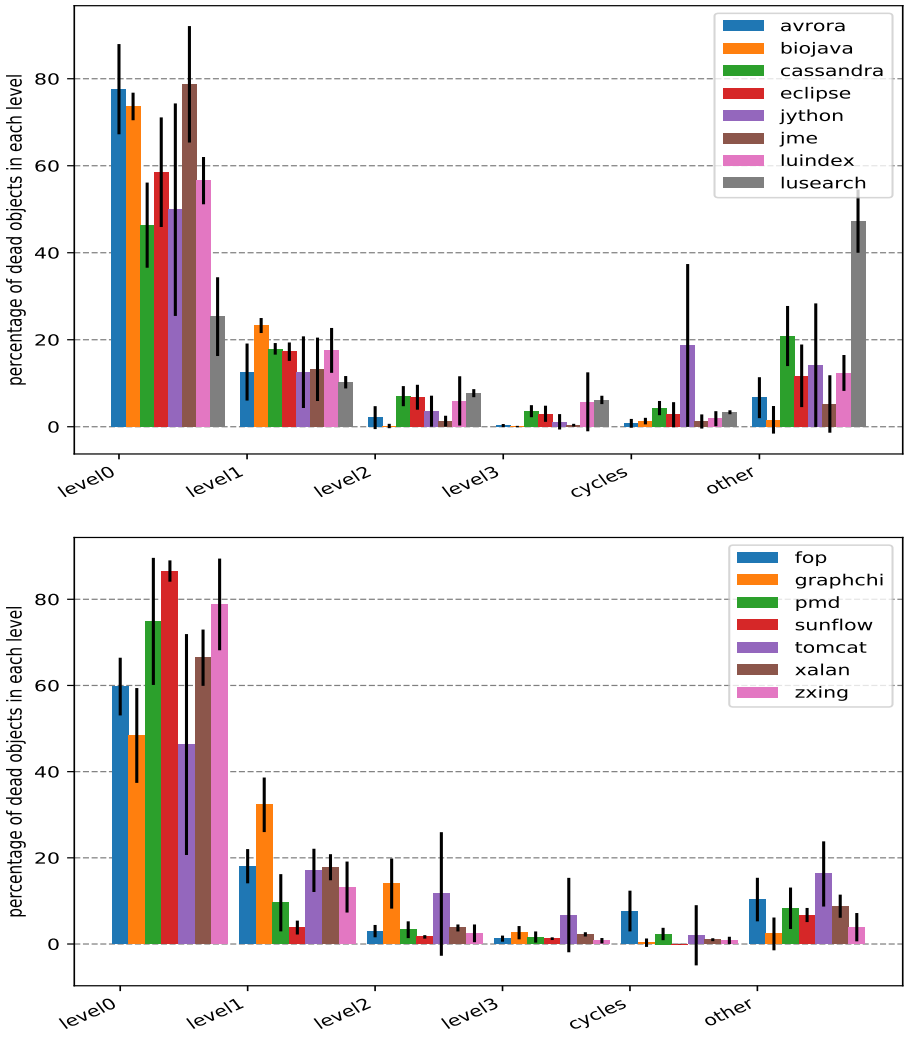


Figure 7: Percentage of the unreachable objects on each level for the DaCapo benchmarks lusearch, graphchi, pmd, jme, and zxing. The "other" level represents all unreachable objects on levels above 3, excluding cycles.

unreachable object O is the length of the longest path from any unreachable object to O. The levels are computed by running Immix, recording the unreachable objects at the end of a collection, and analyzing the graph spanned by these objects and the references between them. We ran the benchmarks on 4x the minimal heap of Immix and ran only one iteration, as these are not time-sensitive evaluations. We averaged the results over 5 invocations. As there are multiple collections in each execution, there may be different numbers of objects at Level i for each collection. Thus, we report the statistics on level percentages: mean and standard deviation. *H2* is not included in this evaluation, as it has a large heap, and we faced many difficulties retrieving the levels information.

Figure 7 shows the percentage of objects on each level: the bars in the figure represent the mean of the percentage of unreachable objects on this level, and the line shows the standard deviation. For example, *jme* and *zxing* both have, on average, 80% of all unreachable objects on Level 0. We display the first four levels (0,1,2,3) and then cycles (level=∞). The objects on levels higher than 3 and not contained in cycles are reported in "other".

For all benchmarks except *cassandra, lusearch, graphchi* and *tomcat*, most unreachable objects are on level 0. This means that Linear-Mark reclaims about 70% of the unreachable objects immediately. About 15% of the objects need to wait for the subsequent collection, and others wait more. Objects in a cycle, or reachable from a cycle, cannot be reclaimed by Linear-Mark. The benchmarks *cassandra, lusearch, graphchi* and *tomcat* suffer the most from the inaccuracy as fewer objects appear in Level 0. Therefore Linear-Mark marks most unreachable objects as live, resulting in its poor performance on this benchmark. Another benchmark on which Linear-Mark performs poorly is *pmd* (see Figure 6). Interestingly, the layout of the levels in *pmd* may seem acceptable: over 70% of unreachable objects

are on level 0, meaning Linear-Mark properly identifies over 70% of the unreachable objects. After analyzing *pmd*'s heap behavior, we noticed it has a low live objects count during collections, mainly because most objects in this benchmark have a low survival rate. This means that during the collections, Immix (and any tracing collector) needs to trace few live objects, which makes it very efficient. In contrast, Linear-Mark marks the reachable objects plus 30% of the unreachable objects, which in this case is a substantial overhead.

Linear-Mark performs the best on the benchmarks *jme* and *zxing* (Figure 6). On these benchmarks, the accuracy of Linear-Mark is high; 80% of unreachable objects are on level 0 (Figure 7).

We conclude that if many objects are on levels higher than zero, this harms Linear-Mark's performance, as it reduces its garbage identification accuracy and creates floating garbage.

Inaccuracy of traced objects. To understand why Linear-Mark fails to perform well, we further report in Table 2 how many objects are traced by each of the collectors for the Dacapo benchmark suite. Note the difference from Figure 7, where we check how many of the unreachable objects are reclaimed. Here, we check how many of the (reachable and unreachable) objects are traced. The first column specifies the heap size used for both Immix and Linear-Mark. The heap size was determined as 4× the minimum heap size required by Immix for the benchmarks. The following columns show the average number of traced objects and the average size of all the objects traced by Immix and Linear-Mark, where the average is taken over the multiple garbage collections in the execution. While we ran Linear-Mark and Immix (as a backup tracing collector) alternately, the statistics of Linear-Mark are taken only from the Linear-Mark collections. The last columns show the ratio of the size and number of the traced objects in Linear-Mark and Immix. For example, in Avrora, the space of the objects traced by Linear-Mark is 5× the space of the objects traced by Immix. All numbers represent the average on five invocations and the average over the multiple collections in each invocation.

7 RELATED WORK

7.1 Tracing Garbage Collectors

Most of the related work was covered throughout the paper, and we will now mention additional relevant studies.

McCarthy presented the first Mark-Sweep garbage collector [26] in 1960 for the LISP programming language. Mark-sweep has since been improved, and most modern collectors use tracing in one form or another [8, 13, 16, 19, 25]. As mentioned in Section 2, machines are getting more and more parallel, managed languages are increasingly used on large-scale multicore environments [10]. Therefore much effort has been put into optimizing scalability in garbage collectors. For example, load-balancing between collector threads is partially achieved using work-stealing [1, 22]. The Mark-Split collector [28] enhances the sweep phase making it proportional to the live data size.

7.2 Reference Counting

A different way of reclaiming space is reference counting [12]. A reference count is maintained for each object and objects whose reference count is decremented to zero can be reclaimed. When an object is reclaimed, the reference counts of all its descendants are decremented, possibly leading to further recursive reclamation. For many years reference counting has been considered inferior, because of its high overhead, and its inability to reclaim cycles (or any strongly connected components). Three dramatic reductions were obtained in deferred reference counting [15, 22], update coalescing [23, 24], and combining reference counting with copying [29]. Furthermore, parallelism with low overhead was enabled by [23, 24]. As a result, modern reference counting collectors deliver high performance, sometimes higher than tracing collectors [31]. Cycles are reclaimed by an accompanying concurrent tracing collector.

8 CONCLUSION

Tracing garbage collectors are widely used. However, tracing collectors have poor locality and fundamental limits to their scalability. In this paper, we investigated a mitigation of these disadvantages by introducing Linear-Mark, a Mark-Sweep variant that trades collection accuracy in order to improve locality and scalability. The evaluation shows a nice reduction in cache miss rates and improved parallelism by up to 2.75x. But the reduction in accuracy makes this collector less performant in typical scenarios. We hope this study of alternatives will inspire further algorithmic improvements that may eventually lead to better garbage collector algorithms.

9 ACKNOWLEDGMENTS

We thank Kunshan Wang from the MMTK team for the insightful discussions. This work was supported by the Israel Science Foundation Grant No. 1102/21.

REFERENCES

[1] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Puerto Vallarta, Mexico) *(SPAA '98)*. Association for Computing Machinery, New York, NY, USA, 119–129. https://doi.org/10.1145/277651.277678

[2] Katherine Barabash and Erez Petrank. 2010. Tracing Garbage Collection on Highly Parallel Platforms. *SIGPLAN Not.* 45, 8 (jun 2010), 1–10. https://doi.org/10.1145/1837855.1806653

[3] Monica Beckwith. 2013. Garbage First Garbage Collector Tuning. https://www.oracle.com/technical-resources/articles/java/g1gc.html#:~:text=-XX%3AParallelGCThreads%3Dn [Accessed: June 2023].

[4] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and Realities: The Performance Impact of Garbage Collection. *SIGMETRICS Perform. Eval. Rev.* 32, 1 (jun 2004), 25–36. https://doi.org/10.1145/1012888.1005693

[5] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. 2004. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings. 26th International Conference on Software Engineering*. IEEE, 137–146.

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, 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. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, 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. 2006. *The DaCapo Benchmarks: Java Benchmarking Development and Analysis (Extended Version).* Technical Report TR-CS-06-01. ANU. http://www.dacapobench.org.

[8] Stephen M Blackburn and Kathryn S McKinley. 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. *ACM SIGPLAN Notices* 43, 6 (2008), 22–32.

[9] Hans-J Boehm. 2000. Reducing garbage collector cache misses. *ACM SIGPLAN Notices* 36, 1 (2000), 59–64.

	Heap Size (MB)	Immix Count	LM Count	Immix Size (MB)	LM Size (MB)	LM/Immix Size	LM/Immix Count
avrora	36	73255	561568	5	25	5	7.66
biojava	776	129278	14921309	161	519	3.22	115.4
cassandra	1014	904103	14765875	127	796	6.26	16.33
eclipse	1972	998969	15887950	228	1179	5.17	15.9
fop	144	151014	1706111	10	98	9.8	11.3
graphchi	1300	467286	6296656	155	957	6.17	13.47
h2	9472	19866405	123127858	953	8721	9.15	6.12
jme	196	65107	1042051	6	53	8.83	16
jython	312	509610	4144156	32	249	7.78	8.13
luindex	116	54448	525424	14	30	2.14	9.65
lusearch	484	208696	4476037	27	421	15.59	21.44
pmd	3700	6463828	43597272	429	2151	5.01	6.74
sunflow	528	710217	9825628	57	450	7.89	13.83
tomcat	440	525515	4398874	42	308	7.33	8.37
xalan	264	188192	1632704	48	125	2.6	8.67
zxing	880	87435	3185518	39	305	7.82	36.4

Table 2: DaCapo benchmark marking statistics for Immix and Linear-Mark. Including the heap size, the average number of traced objects and average overall size of traced objects in the executions of the two collectors. The last columns represent the ratio of the Linear-Mark count/size and the Immix count/size.

[10] Maria Carpen-Amarie, Patrick Marlier, Pascal Felber, and Gaël Thomas. 2015. A Performance Study of Java Garbage Collectors on Multicore Architectures. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores* (San Francisco, California) *(PMAM '15)*. Association for Computing Machinery, New York, NY, USA, 20–29. https://doi.org/10.1145/2712386.2712404

[11] Nachshon Cohen and Erez Petrank. 2015. Data structure aware garbage collector. In *Proceedings of the 2015 International Symposium on Memory Management*. 28–40.

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

[13] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-First Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1029873.1029879

[14] John DeTreville. 1990. *Experience with concurrent garbage collectors for Modula-2+*. Digital Equipment Corporation Systems Research Center.

[15] L. Peter Deutsch and Daniel G. Bobrow. 1976. An Efficient, Incremental, Automatic Garbage Collector. *Commun. ACM* 19, 9 (sep 1976), 522–526. https://doi.org/10.1145/360336.360345

[16] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An Open-Source Concurrent Compacting Garbage Collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Lugano, Switzerland) *(PPPJ '16)*. Association for Computing Machinery, New York, NY, USA, Article 13, 9 pages. https://doi.org/10.1145/2972206.2972210

[17] Robin Garner, Stephen M Blackburn, and Daniel Frampton. 2007. Effective prefetch for mark-sweep garbage collection. In *Proceedings of the 6th international symposium on Memory management*. 43–54.

[18] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 2012. Assessing the scalability of garbage collectors on many cores. *ACM SIGOPS Operating Systems Review* 45, 3 (2012), 15–19.

[19] Google. [n. d.]. Golang GC guide. https://tip.golang.org/doc/gc-guide June, 2023.

[20] DaCapo Group. 2021. *DaCapo Benchmarks Evaluation Snapshot 29a657f*. https://doi.org/10.5281/zenodo.6475255

[21] R. J. M. Hughes. 1982. A semi incremental garbage collection algorithm. *Software - Practice and Experience* (1982). https://doi.org/10.1002/spe.4380121108

[22] Richard Jones, Antony Hosking, and Eliot Moss. 2016. *The garbage collection handbook: the art of automatic memory management*. CRC Press.

[23] Yossi Levanoni and Erez Petrank. 2001. An On-the-Fly Reference Counting Garbage Collector for Java. *SIGPLAN Not.* 36, 11 (oct 2001), 367–380. https://doi.org/10.1145/504311.504309

[24] Yossi Levanoni and Erez Petrank. 2006. An On-the-Fly Reference-Counting Garbage Collector for Java. *ACM Trans. Program. Lang. Syst.* 28, 1 (jan 2006), 1–69. https://doi.org/10.1145/1111596.1111597

[25] Per Liden and Stefan Karlsson. [n. d.]. ZGC: A Scalable Low-Latency Garbage Collector. https://openjdk.org/jeps/333 June, 2023.

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

[27] Oracle. 2023. HotSpot VM, Parallel Collector GC tuning. https://docs.oracle.com/en/java/javase/17/gctuning/parallel-collector1.html#GUID-5A7866BE-59DF-44AD-B51A-274DE3F2BF59 [Accessed: June 2023].

[28] Konstantinos Sagonas and Jesper Wilhelmsson. 2006. Mark and split. In *Proceedings of the 5th international symposium on Memory management*. 29–39.

[29] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S McKinley. 2013. Taking off the gloves with reference counting Immix. *ACM SIGPLAN Notices* 48, 10 (2013), 93–110.

[30] Fridtjof Siebert. 2008. Limits of parallel marking garbage collection. In *Proceedings of the 7th international Symposium on Memory Management*. 21–29.

[31] Wenyu Zhao, Stephen M Blackburn, and Kathryn S McKinley. 2022. Low-latency, high-throughput garbage collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 76–91.