



Much Ado about Offset Vectors

Hayley Patton

Australian National University
Canberra, Australia
hayley.patton@anu.edu.au

Stephen M. Blackburn

Google
Sydney, Australia
Australian National University
Canberra, Australia
steveblackburn@google.com

Abstract

Offset-Vector Compaction (OVC) algorithms, such as the Compressor, are among the most widely deployed garbage collectors today, yet they have remained largely unexplored by the literature since the earliest algorithms were published two decades ago. Although an implementation in the Android Runtime is publicly available and a number of other variations on the algorithm have been developed or proposed, these have not been analyzed or evaluated with recent benchmarks or compared with modern server collectors. This gap in the literature may curtail innovation and exploration of new ideas while reinforcing unchallenged conventional wisdom.

This paper makes six major contributions. We: i) introduce a taxonomy of *offset-vector compaction* (OVC) algorithms; ii) review the history of OVC algorithms, including the Compressor, discovering designs that were previously unreported; iii) implement a family of offset-vector compactors in OpenJDK; iv) implement and evaluate for the first time the One Pass (OP) Compactor; v) introduce a branch-free algorithm for computing forwarding information; and vi) conduct the first evaluation of offset-vector compactors on OpenJDK with modern workloads.

We find that some of the variations we explore offer improvements over the original Compressor algorithm. We also find that, interestingly, the optimization proposed by the one pass (OP) Compactor does not improve over the Compressor. Our implementations are open source, allowing others to build on them. We hope that our analysis and evaluation of this important family of garbage collectors will enable future innovation and improved designs.

CCS Concepts: • Software and its engineering → Garbage collection.

Keywords: Garbage Collection, Parallel Garbage Collection, Compaction, Parallel Compaction



This work is licensed under a Creative Commons Attribution 4.0 International License.

ISMM '26, Boulder, CO, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2720-7/2026/06

<https://doi.org/10.1145/3814942.3816135>

ACM Reference Format:

Hayley Patton and Stephen M. Blackburn. 2026. Much Ado about Offset Vectors. In *Proceedings of the 2026 ACM SIGPLAN International Symposium on Memory Management (ISMM '26)*, June 16, 2026, Boulder, CO, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3814942.3816135>

1 Introduction

Twenty years ago, [Kermany and Petrank](#) published the Compressor garbage collection algorithm [19]. Through a number of interesting optimizations, including an offset table [1] and virtual memory primitives, the Compressor substantially improved over classic approaches to in-place compacting garbage collection [1, 3, 19, 27]. This development was important because although in-place compaction is attractive due to its space efficiency, traditional implementations are computationally expensive due to the number of passes they make over the heap at each compaction. Today, the Compressor is one of the most widely deployed garbage collectors, integrated into the Android Runtime (ART) and running on over a billion phones [13].

Despite holding such an important place in the garbage collection literature, the Compressor has not been extensively studied. Likewise, little is known about the existence of a number of established variations of the Compressor, or their performance. In particular, the algorithm has not been evaluated with modern server collectors, workloads, or hardware. Although an interesting refinement of the Compressor was published in 2024, it was published as an intellectual abstract, so did not include an implementation or evaluation [9]. Thus, one of the most important garbage collectors today is not well understood within the research community.

This paper starts with a survey of directly related work, and in doing so, identifies the family of Offset-Vector Compaction (OVC) algorithms, detailing for the first time the relationship between the original Compressor algorithm and contemporary implementations that derive from it, as well as discovering an unpublished implementation of offset-vector compaction that predates the original algorithm. We don't implement concurrent compaction, but nonetheless our implementation allows us to conduct an in-depth study of the algorithm and its variations with modern workloads and contemporary hardware. We conduct a detailed study of the performance characteristics of key variations on the algorithm. We provide the first implementation and evaluation

of the recently described One Pass (OP) algorithm [9]. We introduce a new branch-free algorithm for computing forwarding information and a new approach to parallelizing compaction, and evaluate both.

Our evaluation confirms that offset-vector compaction is competitive with other canonical algorithms. We find that the Android implementation’s slightly different use of the mark bit vector confers a modest performance advantage over the mark bit vector used in the original Compressor [13]. We find that a single-threaded implementation of the OP Compactor performs about the same as the original algorithm; but the multi-threaded version of the OP Compactor performs slower than a baseline offset-vector compactor, due to the synchronization added by the OP Compactor. We show that, in practice, the original algorithm spends little of its time in the area optimized by the OP Compactor, which helps explain this slightly surprising result. We find that the forwarding algorithm in the Android implementation and our branch-free algorithm are both substantially faster than the forwarding algorithm of the original Compressor. However, the benefit is mostly in speeding up the computation of forwarding addresses, due to the relative unimportance of computing the offset table in the overall performance profile of offset-vector compaction. Our new approach to parallelizing offset-vector compaction is effective, but because it compacts regions rather than the whole heap, it is inherently slightly less space efficient than the approach used in the Compressor. These findings highlight the value of having an open and performant implementation of the classic algorithm available to the research community.

Our hope is that this paper will fill an important gap in the memory management literature by describing and taxonomizing previously unreported foundational work, cataloguing and evaluating optimizations for the first time, and introducing new optimizations for OVC collection. This paper, and the performant open source implementation supplementing the paper, will expand the options available to language implementors and memory management researchers alike.

2 Background and Related Work

In-place compactors constitute an important class of garbage collection algorithms. They can fully defragment the heap while avoiding the need for the large copy reserve required by semi-space [8, 10]. However, in-place compaction is computationally expensive, which has led to numerous improvements over the earliest algorithms—the subject of this paper. The distinguishing features of the implementations and algorithms discussed in this section and our implementations in § 3 are summarized in Table 1.

2.1 Mark-Compact Algorithms

The Lisp-2 collector [27] is the original mark-compact algorithm. Like each of the compactors we discuss, it begins by

using a separate trace of the heap to mark all live objects. It then performs compaction in three passes over the heap:

1. **Compute Forwarding Addresses:** The collector assigns a *forwarding address* to each live object, which is where the object will be moved to in a later pass. The forwarding address must somehow be stored with the object, typically in the object header or in another word preceding the object.
2. **Forward References:** The collector updates every reference in every object to point to the referent’s destination address computed in the first pass.
3. **Relocate Objects:** The collector relocates every object to its destination.

The drawbacks of the Lisp-2 algorithm are that compaction takes a total of three passes over the heap, it requires storing a forwarding address for each object¹, and that the first and third compaction passes are difficult to parallelize. As the first pass calculates forwarding addresses, it assigns increasing addresses to each object, inducing a dependency between the operations performed on successive objects. The relocation pass must move a lower object before it may move a higher object which will overwrite its place. The algorithm may still be parallelized, by partitioning the heap between threads [11] or by careful synchronization between threads [21].

2.2 The IBM Mark-Compact

In the early 2000’s, IBM’s production JVM implemented a mark-compact algorithm [1] that improves over the Lisp-2 collector in two ways: i) it avoids storing forwarding pointers, and ii) it performs only two compaction passes. We refer to this algorithm as IBM MC.

The collector uses three data structures: a mark bitmap, an alloc bitmap, and an offset vector. The mark bitmap is initialized during a mark of the heap performed before compaction. The alloc bitmap is used to indicate the destination of each compacted object.² IBM MC divides the heap into 2^N -sized and aligned *blocks* (they use 256 byte blocks). Blocks do not impose any restrictions on how objects are positioned in the heap; objects may span multiple blocks, and objects are not necessarily aligned with regards to blocks. The offset vector records the relocated address of the first live object in each block.

Unlike Lisp-2 and the Compressor-like algorithms, IBM MC performs relocation first, moving all objects. As it relocates, it updates the alloc bitmap to indicate the location of each relocated object, and updates the offset vector to indicate the address of the first live object in each block. It then performs a fixup pass which uses the alloc and mark

¹We may compress the forwarding information stored in each object [4], but we still must store some information per object.

²In the IBM JVM the alloc bitmap was also used to disambiguate ambiguous pointers, since the JVM did not have precise roots.

Table 1. The compaction algorithms and implementations discussed in this paper. Each of these algorithms start with a mark table produced by a separate heap traversal, which may be concurrent or stop-the-world.

Name	Work	Compaction Passes (heap + tables)	Branch Free Forwarding?	Mark Bits	Concurrent Compaction?	Parallel?
Lisp-2	Stygar [27] (1967)	3 + 0	N/A	N/A	No	No
CCL	Closure Common Lisp [22] (2001)	1 + 1	Yes	All	No	No
IBM MC	Abuaiadh et al. [1] (2004)	2 + 0	?	First	No	Inter-region
Sun PCC	Sun JDK 5.0u6 [24] (2005)	1 + 1	No	First and last	No	Inter-region
Compressor	Kerny and Petrank [19] (2006)	1 + 1	No	First and last	SIGSEGV userfaultfd	Memory remapping
ART CMC	Android Runtime [13] (2022)	1 + 1	Yes	All		No
One Pass	Cory and Petrank [9] (2024)	1 + 0	No	First and last	No	Various
OVC		1 + 1	No	First and last	No	Intra-region
OVC-clmul		1 + 1	Yes	First and last	No	Intra-region
OVC-ccl		1 + 1	Yes	All	No	Intra-region
SOP	Our work	1 + 0	No	First and last	No	No
SOP-clmul		1 + 0	Yes	First and last	No	No
OP		1 + 0	No	First and last	No	Intra-region
OP-clmul		1 + 0	Yes	First and last	No	Intra-region

bitmaps and the offset vector to update all references. The fixup phase redirects a pointer by first calculating the block number of the stale referent, a simple bit shift operation. It then looks up the offset vector to establish the address of the first relocated object within that block. Then it uses the mark bitmap to establish the ordinal number of the referent object within the block. With this, the alloc bitmap, and the address of the first relocated object, it can calculate the address of the relocated referent.

Abuaiadh et al. parallelise IBM MC by partitioning the heap into *areas*, and having each thread claim areas to relocate objects from and to. They allow objects to be relocated *between* areas, in particular into areas that have already had their contents compacted, in order to reduce the fragmentation that may arise from partitioning the heap.

This mechanism of computing a base address per block and then computing a small offset based on side bitmaps, rather than storing a forwarding address per object, is crucial to what we call the *offset-vector compaction* family of mark-compact algorithms. The offset-vector compaction algorithms allow for passes of the Lisp-2 collector to be fused and parallelized, as the metadata used to compute the forwarding address of an object exists independently of the object, and the metadata is not invalidated when an object is relocated. All of the collectors in Table 1 except for Lisp-2 follow this offset-vector compaction paradigm, and all of those collectors use fewer passes than the Lisp-2 algorithm.

2.3 Sun’s Parallel Compacting Collector

In December 2005, Sun released J2SE 5.0 Update 6, which included the *Parallel Compacting Collector* [24]. The collector is briefly described as mark-summary-compact, with the summarization phase implementing offset-vector compaction,

captured in this brief description: “*The summary phase calculates and stores the new location of the first byte of live data for each compacted region*”. In particular, the function `PSParallelCompact::summarize_blocks` in the first commit to OpenJDK 6 [25] computes a forwarding address for every 128-word *block*. The function `block_calc_new_pointer` then computes the live data between an object and the start of its block using the mark bitmap, and adds that offset to the address for that block to compute a forwarding address for that object. Sun PCC is parallelised in a similar manner to IBM MC, assigning *chunks* for threads to relocate objects out of and into. OpenJDK continued to implement this algorithm as the mature collector for the widely-used ParallelGC until JDK 23, where it was replaced with a Lisp-2-based algorithm [29], citing less memory usage as object headers can already hold a forwarding address in OpenJDK. Kenne also notes that the Lisp-2 algorithm allows objects to grow for *compact* address-based hashing, where the collector grows objects to store their hash codes when moving the objects [17]. offset-vector compaction algorithms cannot easily support resizing objects, as such algorithms use the sizes of objects before collection to determine forwarding addresses.

2.4 The Compressor

The Compressor [19] is a concurrent and parallel mark-compact garbage collector, initially implemented in Jikes RVM [2]. Although similar in spirit, it differs from IBM MC in significant respects: i) it does not require an alloc bitmap; ii) its first phase computes the offset vector without accessing the heap; iii) its second phase fuses Lisp-2’s forwarding and relocation passes; iv) it parallelizes the relocation forwarding and relocation pass, and v) it performs relocation concurrently with the mutator.

The Compressor uses an offset vector, like IBM MC. However, rather than establishing the offset vector during a pass over the heap, the Compressor calculates it from a pass over the mark bitmap. For this to work, the mark trace which precedes compaction must set *two bits* for each live object rather than the conventional singular mark bit. This means that the mark bitmap must resolve two bits per minimum object size — twice the resolution of a conventional mark bitmap, requiring a bitmap twice the size. The two bits correspond to the first and last words of the marked object. This allows the Compressor to calculate the number of live bytes and thus cumulatively compute relocation addresses for all marked objects in a single pass over the mark bitmap.

Unlike IBM MC, the Compressor uses the offset vector to store the address of the *first relocated byte* within each block (rather than the *first moved object*). This allows the Compressor to calculate the forwarding address of an object as the sum of the address stored in the offset vector and number of live bytes between the start of the block and the start of the object.

2.4.1 Parallel Relocation. Traditional single-pass, single-thread compaction naturally maintains an invariant that each relocated object will only be moved backward, to a *vacant* location (a location from which all prior occupants have already been removed). However, in a multi-threaded context, the destination of a relocation may be a memory region that is being processed by another thread, and may not yet be vacant.

The Compressor solves this problem elegantly using virtual memory remapping. It introduces two virtual spaces: *from-virtual-space* and *to-virtual-space*. Instead of calculating relocation addresses that overwrite the original heap (from-virtual-space), the Compressor targets a fresh virtual memory range (to-virtual-space). Mapping and unmapping happens as follows. At the start of relocation, to-virtual-space is not yet mapped. As each object relocation occurs, if the destination page in to-virtual-space is not mapped, a new page is mapped. Once all objects on the from-virtual-space page have been relocated, it is unmapped. This way, the number of physical pages in use at any time is at most T higher than the number of pages required by the single-threaded implementation, where T is the number of parallel relocation threads.

The computing offset vector pass of the Compressor may also be parallelized by means of a parallel prefix sum algorithm, but Kermany and Petrank did not do so in their implementation, noting that it took only 5–7% and 11–13% of collection time on the SPECjvm98 and SPECjbb2000 benchmarks respectively (§5.2 of [19]).

2.4.2 Concurrent Offset Calculation and Relocation. The Compressor performs both offset calculation and relocation concurrently with the application. Concurrent offset

calculation is straightforward, although care must be taken to correctly account for and update newly allocated objects.

To implement concurrent relocation, the Compressor builds on the virtual memory technique it uses to implement parallel relocation. Like Appel and Li [3], Kermany and Petrank hold an invariant that the application can only access virtual-to-space pages, and use memory protection to ensure that the application does not prematurely access virtual-to-space pages. When the application follows a reference to a protected virtual-to-space page, the signal handler relocates objects from the corresponding virtual-from-space pages until the virtual-to-space page has been filled and all references it contains are forwarded. At that point the signal handler unprotects the page and the application continues.

2.4.3 Resolving Mark Bit Ambiguity. Because the mark bitmap uses one bit to denote the first word of an object and one to denote the last, without context it is ambiguous as to whether a given bit represents the start or the end of an object. This detail is not discussed in Kermany and Petrank’s paper [19]. We solve this by adding a boolean to each offset vector entry indicating whether the block starts within a live object; we store the boolean as the least significant bit of each value in the offset vector.

2.5 Clozure Common Lisp

Clozure Common Lisp implements a collector which we call CCL, that is very similar to the Compressor, but which as far as we are aware has never been published. During the development of this work, we investigated and established that that collector implementation dates at least to an open source implementation from 2001 [22], and there are references to its existence dating to 1996³. As far as we are aware there is no record of the CCL collector in the literature. The Compressor was developed independently. As far as we know, the link between these two collectors has not previously been made. The CCL collector is a generational, single-threaded, and fully stop-the-world offset-vector compaction algorithm. Like the Compressor, CCL is implemented in two passes, fusing relocation and reference forwarding. CCL also computes forwarding metadata from an offset vector and a mark bitmap, however CCL marks every live word of an object rather than the first and last words. The result is that the forwarding address of an object is computed by adding the *population count* of marked bits between the object and the start of the block containing the object to the address in the offset vector.

³In particular, the start of the file `lisp-kernel/gc.c` of [22] contains a change-log with references to functions in the current garbage collector from 1996.

2.6 The Android Runtime Mark Compact

The widely-deployed Android Runtime (ART) implements a collector we call ART CMC which is based on the Compressor with three notable differences [13]. First, like CCL, the marker sets a bit for every live word rather than one for the first and one for the last live words of each object. Second, unlike CCL, the marker also counts the live words in each block, only requiring the offset vector phase to compute an *prefix sum* of the live words in order to compute the offset vector. Third, it uses the more modern `userfaultfd` rather than `SIGSEGV` to implement page protection [13].

Prior to ART CMC, ART had an implementation of a Lisp-2-style collector [27]⁴. This implementation put a forwarding pointer in the mark word of each object.

2.7 The One Pass (OP) Compactor

Cory and Petrank describe the One Pass (OP) Compactor [9], which merges the two passes of the Compressor, interleaving computing the offset vector, moving objects, and updating references into a single pass. One of the contributions of this work is the first implementation of the OP Compactor.

The difficulty of reducing compaction to a single pass lies in forwarding objects in the same pass as computing offsets. When the referent to be forwarded points *backwards* to an object in a block that has already been processed, the offset vector will already be up to date and the relocation address can be straightforwardly computed. However, if the referent to be forwarded points *forwards* to a block whose offset has not yet been computed, then the target address is unknown, so the object can't be forwarded. The OP Compactor solves this by attaching a list of locations of unforwarded pointers to each such referent object. When the referent object is eventually relocated, the list of unforwarded pointers can at last be forwarded to point to the relocated object. To implement this efficiently, the collector uses *pointer threading* [16] with a per-object list they call the *threading list*.⁵ The collector traverses and fixes fields in the threading list of the object when the pass eventually visits the object.

Parallelizing the OP Compactor involves two main issues: multiple threads will not visit blocks exactly in address order; and there is a race condition between threads trying to append to a threading list of an object, while another thread fixes the fields on the list and moves the object. The first issue can be addressed by explicitly recording and tracking the status of each block, instead of assuming the status of each block by address order. Cory and Petrank address the second issue with two locking schemes. First, they use the status of each block to track the number of threads trying to

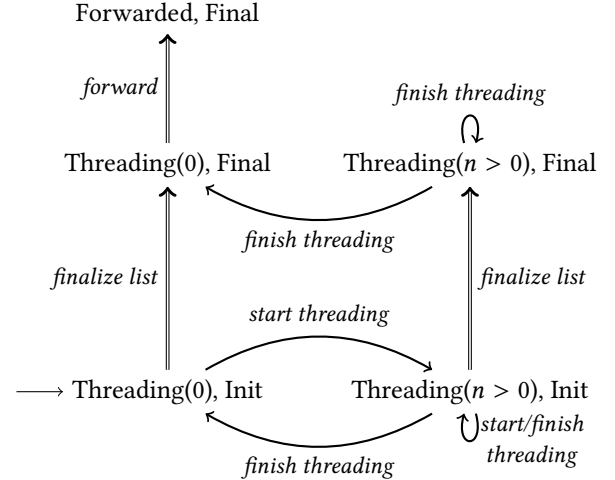


Figure 1. The states and transitions of blocks and status bits in the OP Compactor. Transitions made when moving blocks have double-stroke lines, and transitions made when threading objects have thick lines.

thread onto the block and disallow any objects to be moved on a block while any threads are trying to thread onto that block. Second, a thread which wishes to forward all slots on the threading list of an object must first tag the threading list with a *final* state, and any other threads which attempt to thread fields onto a threading list will observe that state and instead forward the field using the forwarding metadata which is computed before the final state is set. Figure 1 illustrates the states of blocks and status bits, and their allowed transitions.

3 Implementation

To explore the tradeoffs made by the various offset-vector compaction (OVC) designs (Table 1), we implement OVC in MMTk. Our focus is on offset-vector compaction, so our implementation does not implement concurrent marking [1] or concurrent compaction [19].

We implement three variations of *two-pass* offset-vector compaction, each with a different approach to live data calculation: i) using the Compressor's approach (OVC, [19]), ii) using carryless multiply (OVC-clmul, § 3.4), and iii) using CCL's approach (OVC-ccl, [22]). We also implement four variations of *one-pass* offset-vector compaction, a cross-product covering serial (S) and parallel implementations with and without carryless multiply (-clmul). We name the four collectors OP, OP-clmul, SOP and SOP-clmul.

3.1 Parallel Relocation

In its most stripped down implementation, our OVC uses a single thread to relocate the entire contiguous OVC heap, a straightforward application of the offset-vector compaction algorithm. We now describe our approach to parallelizing

⁴https://cs.android.com/android/_/android/platform/art/+/-/f8e5d8ce6cdd9048dc8e67f7343d3842189f7601

⁵Threading algorithms assume that we can reuse a word from the header for the threading list, using the data in the header to mark the end of the list; and they assume that we can arrange for header data and the addresses of fields to be distinguishable, such as by *tagging* either differently.

relocation. As discussed in § 2.4.1, if collector threads were to naïvely race to relocate individual objects, the simple invariant that each object will always be moved to a vacant location breaks down. One thread may attempt to relocate an object to a location that in the serial implementation would have been vacated, but in the parallel setting might still contain an object that has not yet been moved.

The Compressor addresses the challenge of parallel relocation by using two virtual semi-spaces, implemented via virtual memory primitives, which also serve to support the Compressor’s approach to concurrent compaction (§ 2.4.1).

We take a different approach to parallelizing relocation, partitioning the heap into *regions* of 256 kB, moving objects only within their containing regions (unlike IBM MC and Sun PCC).⁶ Each region is compacted by a single collector thread. Collector threads race to relocate regions until all regions have been relocated.

3.2 Handling External References

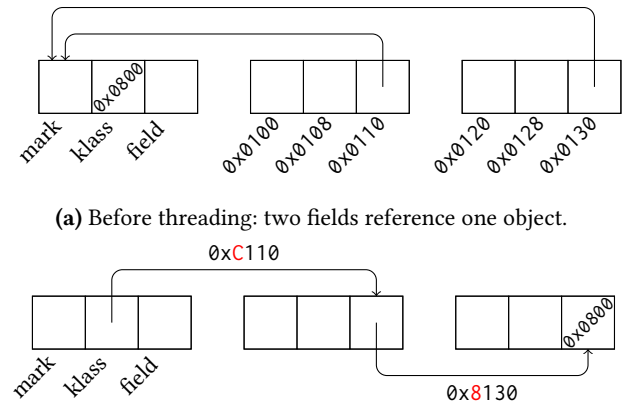
When objects are relocated, external references into the OVC heap must be updated (in addition to the references within the OVC heap). One source of such references is the root set. Another is other regions of the program heap that are managed by other policies, such as a large object space. We follow the approach of Ben-Yitzhak et al. [5], building a remembered set during tracing that records the address of all heap pointers which point into the compacted space. After computing the offset vector, we process the remembered set, updating each pointer to the relocated address.

3.3 Pointer Threading

Recall that while the OP Compactor can forward backward-pointing references as it goes (since the referent has been relocated), it must defer forwarding of forward-pointing references (since the referent is yet to be relocated) [9]. This is handled by associating a list of addresses of to-be-relocated pointers with each object. When the object is eventually relocated, each of the pointers in its list can then be updated to point to the object's new address. While in principle the per-object lists could be managed with some ancillary data structure such as a hash table of lists, the OP Compactor implements the per-object lists very space-efficiently, using pointer threading [16].

Pointer threading attaches a list of pointer locations to an object by overwriting a field of the object (typically in the header) to point to the location of the first pointer in the list. The contents of the first pointer of the list are overwritten to point to the second pointer in the list, etc. The list is terminated by the original value that was overwritten by the very first reference in the list. This is depicted in [Figure 2](#).

⁶We selected the region size to be the largest region size with good load balancing, attempting to minimize the fragmentation which would be induced by having small regions, while attaining the ample parallelism offered by having many regions.



(b) After threading: the class word of the object is the start of a linked list through the two fields, with tags in the most-significant bits (in red) denoting that the word is a pointer to a slot and that the list has been finalized.

Figure 2. The representation of a threading list in our binding for OpenJDK.

Pointer threading relies on a guarantee that the terminal value is distinguishable from the other elements in the list. This requires that both the overwritten value and the threaded pointers have at least one bit of state that can be systematically stolen. In the case where they are all word-aligned pointers, the two low order bits offer this, so for example, the overwritten value could be stored with the low order bit set to one, differentiating it from the threaded pointers, which have their low order bit set to zero. This is complicated in modern VMs which implement compressed pointers that typically use all 32 available bits.

In fact the OP Compactor requires *two* free bits in order to implement *parallel* threading. This is needed to manage the race that can arise between one thread consuming the list and another thread attempting to extend the list. One way to gain the needed two bits might be to restrict the compressed pointer address space to free up bits. Another is to disable compressed pointers altogether, which is what we have done for our OP Compactor implementations. We can then use the object header's klass word (a 64-bit pointer) as the head of the list of threaded pointers.

We keep the klass word intact at all times, noting that the most significant bit will be zero. We tag the addresses of slots with a one in the most significant bit, and we tag finalized threading lists with a one in the second-most significant bit, resulting in a representation as depicted in [Figure 2](#).

We added several methods to the interface for virtual machine bindings to MMTk, which atomically push slots to the threading list of an object, finalize the state of a threading list, non-atomically iterate the slots on the threading list of an object, and reset the threading list of an object, as in [Figure 3](#).

```

trait ObjectModel<VM: VMBinding> {
  // Producing the threading list
  fn push_threading_list(
    object: ObjectReference,
    slot: VM::VMSlot);
  // Consuming the threading list
  fn finalise_threading_list(object: ObjectReference);
  fn walk_threading_list(
    object: ObjectReference,
    f: &mut impl FnMut(VM::VMSlot));
  fn reset_threading_list(object: ObjectReference);
}

```

Figure 3. The necessary additions to the MMTk VM interface for implementing threading compaction.

The interface abstracts over how the threading list is maintained, and our implementation of our tagging scheme on klass words is contained entirely within the binding library for OpenJDK. This interface allows our implementation of the OP Compactor to interoperate with other runtimes which represent the threading list differently, due to the runtime using different object representations.

3.4 Carryless Multiply-Based Offset Calculation

First, we recap how existing OVC designs compute the offset vector, which is subsequently used to compute forwarding addresses. Naïvely, they could do so by using a simple mark bitmap, interrogating each marked object, establishing its type, finding its size, and then adding this to a cumulative sum that denotes the total live bytes parsed so far. In practice OVC designs compute the live bytes directly from the mark bitmap, avoiding costly interrogation of the marked objects.

During the marking phase, the Compressor marks both the first and last words of each live object. By marking both ends of the object, the Compressor is able to calculate the live bytes directly from the mark bitmap. To do so it must keep track of whether a bit denotes the first or last word of an object (§ 2.4.3). This algorithm performs little work on sparse heaps when using a *count trailing zeroes* operation to skip over cleared bits in the bitmap, but the algorithm must perform many iterations of the loop per word when the area being traversed is densely marked.

In contrast to the Compressor, the Android Runtime and Clozure Common Lisp set a mark bit for *every* word in each live object. On the one hand, this adds work to the process of marking an object ([15, p. 43]), but on the other hand, it allows the amount of live data to be trivially computed by performing a population count over each portion of the mark bitmap corresponding to a block of the heap. This is inherently *bit-parallel* and does not require any unpredictable branching based on the contents of the mark bitmap.

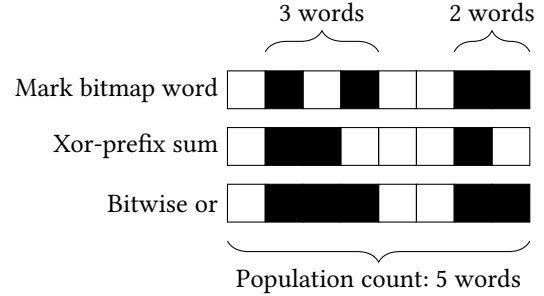


Figure 4. Computing the live data using carryless multiply. The least-significant bits are on the left.

We introduce a bit-parallel and branch-free algorithm for computing the offset vector and for computing forwarding addresses, which only requires the first and last words of live objects to be marked, identical to the original Compressor. The algorithm uses the *carryless multiply* instruction found in many contemporary processors⁷. The algorithm is based on the algorithm to detect quoted substrings in the *simdjson* JSON parser by Langdale and Lemire [20]. We may define a (normal) multiplication of a and b as summing a multiplied by each bit of b , i.e. $\sum_i ab_i \ll i$; the carryless multiplication computes an analogous bitwise-xor “sum” of partial results $\bigoplus_i ab_i \ll i$. Langdale and Lemire observe that carryless multiplication can be used to implement a *prefix sum*⁸ of the bitwise-xor operation across the bits of a word by fixing b to have all bits set, reducing the operation to $\bigoplus_i a \ll i$. (The i -th bit of the result of the prefix sum is $a_i \oplus a_{i-1} \oplus \dots \oplus a_0$.) The prefix sum of a word of the mark bitmap has almost all bits corresponding to live words being set⁹, with the exception that the last bit of each live object will be cleared. We can then force the last bit to be set by a bitwise-or of the result with the original word, and we can then compute the total live words as the *population count* of the result of the bitwise-or. An example of this algorithm is depicted in Figure 4. This algorithm performs a constant amount of work per word, regardless of the density of the heap, and the two operations involved are fast and constant-time: the carryless multiply and population count instructions have 4 cycle and 1 cycle latency respectively on Zen 4 [12], for example.

The carryless multiply and population count instructions are widely supported by current x86-64 hardware. However, these instructions are not always implemented in mobile processors, particularly in low-end systems-on-chips. Our

⁷This instruction is `pclmulq` on x86-64, `pmull` on AArch64 and `clmul` in the RISC-V Zbc extension.

⁸We can implement a prefix sum by other means, such as by a Hillis-Steele scan [14, 28]. But we found such algorithms are slower than using a carryless multiply, as Langdale and Lemire found in *simdjson*.

⁹...when the first bit set in a word is the start of an object. Else all the bits of the prefix sum must be inverted if the first bit set is the end of an object, which is implemented in both *simdjson* and OVC-`clmul` by *carrying* from the most significant bit of the prefix sum of the last word of the mark bitmap.

implementation performs feature detection at runtime, defaulting to using the carryless-multiply algorithm, and falling back to the original Compressor forwarding algorithm if the processor does not support both instructions. The collector generates *work packets* [31] which are specialised for the forwarding algorithm, avoiding any dispatch in the hot loops of the collector, whilst also keeping most of the implementation generic across forwarding algorithms.

4 Methodology

We evaluate all collectors on systems with a AMD Ryzen 9 7950X processor (with the Zen 4 microarchitecture and 64 MB of L3 cache), 64 GB of 4800 MHz DDR5 memory, running Ubuntu 22.04 on a solid-state disk. We use the jdk-21.0.9+3-mmTk fork of OpenJDK 21, and make our implementations publicly available [26].

We use all 22 benchmarks in the DaCapo Chopin benchmark suite [6]. We present benchmark execution times with the geometric mean of 10 iterations, performing each iteration after 4 warmup iterations. We limit each benchmark to a timeout of 180 seconds.

Compressed Pointers. By default, we leave OpenJDK’s compressed pointers enabled. However, because our OP Compactor implementation does not support compressed pointers (§ 3.3), we disable compressed pointers for all collectors in experiments that evaluate the OP Compactor.

Reference Processing. MMTk’s OpenJDK 21 binding is new and does not yet fully support reference processing, so we disable reference processing for all collectors in all evaluations.

OpenJDK Minimum Heaps. We use heap sizes relative to the minimum heap sizes of G1, when evaluating the offset-vector compaction collectors against other MMTk collectors in § 5.1. We found that 5 of the 22 DaCapo benchmarks (avrora, eclipse, h2o, sunflow and zxing) suffer from unexplained memory overheads. The problem is specific to MMTk’s new OpenJDK 21 binding, affecting all MMTk collectors, but not the native OpenJDK collectors. It may be related to a known problem with weak root processing in the OpenJDK 21 binding [30]. Because this problem is orthogonal to our evaluation of OVC and distorts any comparisons to the space efficiency of OpenJDK 21’s native collectors, we exclude these five benchmarks in § 5.1, but retain them in all other comparisons. We also found that Immix requires a larger minimum heap (3.0× that of G1) than Semi-space (which requires 2.4× the minimum heap of G1) when running h2, whereas Immix is more space efficient than Semi-space on all other benchmarks. We thus also exclude h2 in § 5.1.

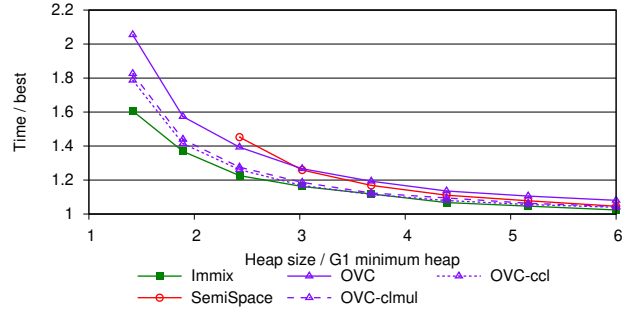


Figure 5. Overall performance of two-pass offset-vector compactors versus Semi-space and Immix. The graph shows the geometric mean of the normalized total time across the 16 DaCapo benchmarks we evaluate, as a function of heap size. The graph depicts the classic time-space tradeoff, with GC time dominating in smaller heap sizes.

5 Results

5.1 Overall Performance of OVC

Figure 5 depicts the overall performance of OVC, OVC-ccl, and OVC-clmul with respect to two other full-heap MMTk collectors; Semi-space [8, 10] and Immix [7], showing total execution time as a function of heap size.

Overall, OVC performs similarly to Semi-space (albeit being able to run in smaller heaps than Semi-space), and both OVC-clmul and OVC-ccl usually perform between Semi-space and Immix. Separating GC and mutator time (not shown), we see that mutator times are more or less constant and that Immix pays roughly a 4-5% overhead compared to the OVC collectors and Semi-space, which all have near-identical mutator performance. The mutator penalty incurred by Immix is slightly higher at smaller heaps, which is consistent with its allocator spending more time looking for holes in a densely fragmented heap.

OVC-clmul and OVC-ccl significantly outperform OVC, with both collectors performing similar to Immix at larger heap sizes. All offset-vector compactors can run in smaller heaps than Semi-space and Immix.

Table 2 depicts the performance of the same collectors at 2.4× the minimum heap for G1, which is the minimum size we evaluated which allows Semi-space to run all benchmarks. The confidence intervals are mostly smaller than 2% and always smaller than 5%, except for spring on Immix having a 6% confidence interval and xalan having 18-24% confidence intervals. We thus exclude xalan from the statistical summary in Table 2. biojava performs exceptionally poorly on all offset-vector compactors, due to producing many references from large reference arrays in the large-object space to the compactor space. All our implemented offset-vector compaction collectors have to log such references in the remembered set whilst tracing, and then update

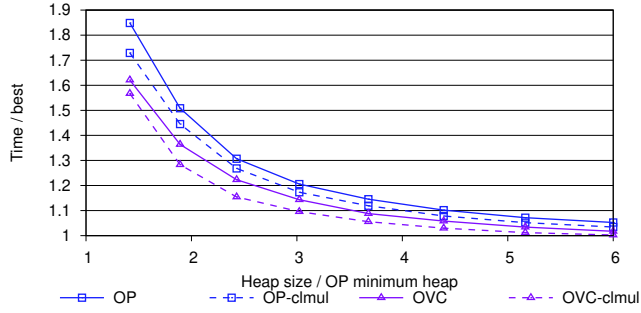


Figure 6. OP performance compared to OVC. OP systematically lags OVC performance. These evaluations are all conducted with compressed pointers disabled (§ 3.3).

the references whilst compacting. (The collector updates references in parallel, but there are still very many references to update.) Immix and Semi-space instead update such references while tracing, avoiding having to revisit any slots in the large-object space.

OVC-ccl almost always ties or slightly outperforms OVC-clmul. We had initially assumed that OVC-ccl marking all words of live objects (as opposed to OVC-clmul marking just the first and last words, § 2.5) would slow down tracing. However, setting all bits in a range is not necessarily a slow operation. Most objects are small, and so the relevant bits are likely to reside in the same bytes, allowing the generation of the mask and storing of the bits in constant time. All of the bits are also very likely to reside in the same cache line, so setting all bits should not induce more *cache ping-pong* than setting just the first and last bits. MMTk also allocates objects larger than 8 kB in a separate, non-moving *large object space*, which implies that OVC-ccl will mark at most 128 bytes of mark bitmap (which can span at most three 64 B cache lines) for one object. OVC-clmul must furthermore perform an additional carryless multiply per reference to forward than OVC-ccl, which is a small but noticeable slowdown; Fog [12] found that carryless multiplication has a four-cycle latency on the Zen 4 microarchitecture.

5.2 Evaluating OP Compaction

We now evaluate the performance of the One Pass algorithm. Recall that Cory and Petrank [9] proposed One Pass Compaction, but until now it has not been implemented or evaluated (§ 2.7).

Figure 6 shows the performance of OP and OVC implementations as a function of heap size, showing the original offset vector calculation used by the Compressor (§ 2.4, [19]) and the clmul variant of the algorithms we introduce (§ 3.4). (Although ccl (§ 2.5, [22]) performs slightly better than clmul in the OVC comparison (Figure 5), it is a more substantial departure from OP Compaction, so we have not implemented a ccl variant of OP.)

Table 2. Performance of two-pass offset-vector compactors versus Semi-space (SS) and Immix (IX) of each benchmark at a 2.4× heap. We present the absolute times for Immix in milliseconds and relative performance for other collectors. We exclude the results for xalan from the statistical summary due to having large confidence intervals (between 15 and 25%).

Benchmark	IX	SS	OVC	-clmul	-ccl
batik	1 095	0.99	0.97	0.97	0.97
biojava	6 339	1.08	1.86	1.63	1.64
cassandra	5 731	1.05	1.02	1.00	1.00
fop	374	1.02	1.02	1.00	1.00
graphchi	2 680	1.03	1.06	1.02	1.03
jme	6 817	1.00	1.00	1.00	1.00
kython	4 318	0.98	0.98	0.97	0.97
kafka	5 104	1.00	1.00	1.00	1.00
luindex	2 666	1.01	1.02	1.01	1.01
lusearch	3 476	2.09	1.73	1.46	1.39
pmd	1 356	1.45	1.17	1.00	0.99
spring	3 963	2.00	1.36	1.13	1.08
tomcat	3 829	1.11	1.13	1.08	1.07
tradebeans	8 937	0.59	0.54	0.49	0.49
tradesoap	3 665	1.13	0.91	0.84	0.82
xalan	1 076	1.97	1.78	1.38	1.35
min		0.59	0.54	0.49	0.49
max		2.09	1.86	1.63	1.64
mean		1.17	1.12	1.04	1.03
geomean		1.12	1.08	1.01	1.00

It is clear from Figure 6 that the single-pass OP algorithm systematically lags the two-pass OVC algorithm. At first, this result seems surprising, since OP was proposed as an optimization, fusing the two passes of the Compressor into one [9, 19]. However, the fused single pass of OP is intrinsically more complex than the separate passes of the Compressor. This is because the single pass must handle unresolved forward references (implemented with pointer threading § 2.7), which introduces data races that don't exist in the case where all references are resolved in the previous pass.

To understand this better, we implemented a strictly single-threaded version of OP with no support for collector parallelism (SOP). Figure 7 compares OP with our two-pass OVC implementation while constraining each of the garbage collectors to use only one thread. In each case we use the clmul variant, which has the best performance for each collector algorithm. The graph shows that removing support for parallelism almost completely removes the gap seen in Figure 6, largely explaining the surprising underperformance of OP.

Figure 8 shows the impact of clmul (§ 3.4) on the serial OP and OVC algorithms. It shows that, without the clmul

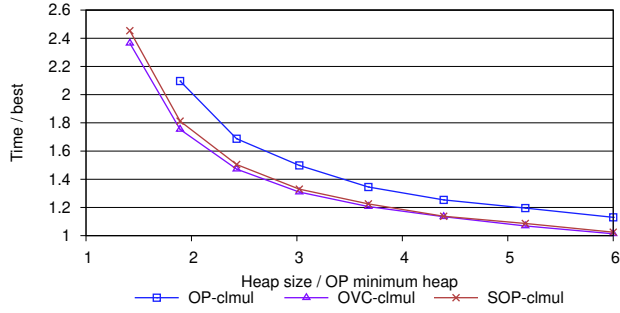


Figure 7. Single-threaded performance of OP compared to OVC. OP (OP-clmul) lags OVC (OVC-clmul) substantially. However, when we include in the comparison a version of OP with no support for parallelism (SOP-clmul), we see that OP pays a heavy price for parallelism, and that without this, OP is much closer to OVC in performance. These evaluations are all conducted with compressed pointers disabled (§ 3.3). OP-clmul times out on lusearch and sunflow at the 1.4× minimum heap when using a single collector thread, and thus requires a larger heap than in Figure 6.

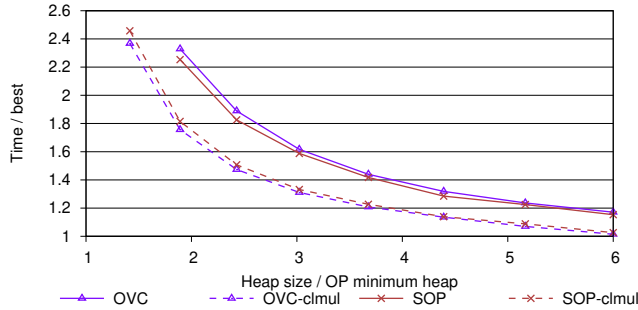


Figure 8. The impact of clmul on SOP and OVC, single-threaded. SOP outperforms OVC, but OVC-clmul outperforms SOP-clmul, showing that the clmul optimization benefits OVC more than OP.

optimization and without support for parallelism, OP (SOP) does indeed slightly outperform the two-pass OVC (OVC). However, when offset vector calculation and forwarding address computation are optimized with the clmul algorithm, the two pass algorithm (OVC-clmul) slightly outperforms OP (SOP-clmul). Introducing the clmul algorithm speeds up all of OVC, SOP, and OP (Figure 6), but OVC-clmul still outperforms OP-clmul in a parallel setting.

5.3 Discontiguity and Mutator Performance

As noted earlier (§ 5.1), the various OVC implementations show no significant difference in mutator performance. Recall that our parallelization strategy breaks the heap into regions, introducing coarse-grained discontiguity (§ 3.1). This could be a source of mutator overhead since allocating into a

contiguous heap causes objects to remain in allocation order, and allocating into regions only preserves allocation order within each region. Acquiring a new allocation buffer is also more difficult with a region-structured heap, as a mutator must find a region to allocate into in a discontiguous heap, instead of simply bumping a pointer for a contiguous heap. But acquiring a new allocation buffer is designed to be an infrequent operation by using reasonably large allocation buffers. Despite the theoretical drawbacks of using a discontiguous heap, we found no significant difference in mutator performance.

5.4 Threading Lists

Recall that, when forwarding a reference in a slot, the OP Compactor will use the offset vector to compute a forwarding address if the offset vector is ready, else the OP Compactor will thread the slot onto the threading list of the referent object (§ 3.3). A discontiguous heap, even when processed by a single thread, will likely affect the probability of whether a referent object has already been relocated when a slot is processed. Table 3 shows that the OP Compactor uses the offset vector most of the time on all benchmarks (i.e. the referent object has already been relocated). The contiguous OP Compactor only threads 9–43% (geomean 17%) of slots at 3.0× its minimum heap size, and the discontiguous OP Compactor threads 13–33% (geomean 20%) of slots at the same size. The discontiguous compactor almost always threads more slots, with the exception of sunflow threading 43% of slots in the contiguous heap and 29% of slots in the discontiguous heap.

The contiguous OP Compactor (and the Compressor) maintain an invariant that older objects will have lower addresses than newer addresses, whereas a discontiguous compactor only maintains this invariant inside each region. We can deduce that most references are from newer to older objects, which coincides with the faster and unsynchronized path of forwarding a slot in the OP Compactor. The discontiguous implementation of the OP Compactor cannot exercise this fast path as often, and so the discontiguity may degrade performance of the OP Compactor.

One could conceivably evaluate separately the overheads of the synchronization and discontiguity, by implementing and benchmarking either mechanism without the other. The synchronization overhead can be measured by adding synchronization to the serial OP Compactor, while retaining a contiguous heap. The overheads induced by the interaction between threading and discontiguity can be measured by adding regions to the serial OP Compactor, and ensuring that regions are compacted in address order. We leave this evaluation as future work.

5.5 Scalability

We evaluate the scalability of OVC and OP, with both the branchy and carryless multiply-based forwarding algorithms.

Table 3. The proportion of slots which are threaded in discontinuous and contiguous One Pass, at $3.0\times$ minimum heap size for contiguous One Pass, and the trends of proportions with regards to heap size. $\nearrow$ denotes an increase of threading with larger heap sizes, $\searrow$ denotes a decrease, $\rightarrow$ denotes little or no change, and $\rightsquigarrow$ denotes some change with no consistent trend with regards to heap size. All proportions of threading are stable for contiguous One Pass. (Benchmarks which do not collect at this heap size are excluded.)

Benchmark	Discontinuous	Contiguous
avroa	$\rightsquigarrow$ 0.30	0.20
biojava	$\nearrow$ 0.19	0.18
cassandra	$\rightsquigarrow$ 0.25	0.16
eclipse	$\rightsquigarrow$ 0.24	0.16
fop	$\searrow$ 0.19	0.18
graphchi	$\rightarrow$ 0.15	0.15
h2o	$\searrow$ 0.18	0.15
jython	$\rightsquigarrow$ 0.14	0.13
kafka	$\searrow$ 0.17	0.16
luindex	$\rightarrow$ 0.19	0.19
lusearch	$\searrow$ 0.24	0.20
pmd	$\searrow$ 0.20	0.15
spring	$\searrow$ 0.20	0.11
sunflow	$\rightarrow$ 0.29	0.43
tomcat	$\rightarrow$ 0.20	0.18
tradebeans	$\searrow$ 0.13	0.09
tradesoap	$\searrow$ 0.17	0.13
xalan	$\searrow$ 0.33	0.25
zxing	$\searrow$ 0.17	0.16
min	0.13	0.09
max	0.33	0.43
mean	0.21	0.18
geomean	0.20	0.17

Figure 9 shows that OVC and OP scale similarly regardless of forwarding algorithm, confirming the expectation of Cory and Petrank [9] that the parallel OP Compactor algorithm would not experience much contention. The branchy algorithm scales better than the carryless multiply-based algorithm, but the branchy algorithm is almost always drastically slower than the carryless multiply-based algorithm; the higher scalability is solely due to the branchy algorithm being slower with fewer threads [23]. We suspect that the carryless multiply-based algorithm is able to saturate memory bandwidth with fewer threads than the branchy algorithm.

6 Conclusion

Offset-Vector Compaction algorithms are among the most important garbage collection algorithms today and yet one of the most understudied. In this paper, we identify offset-vector

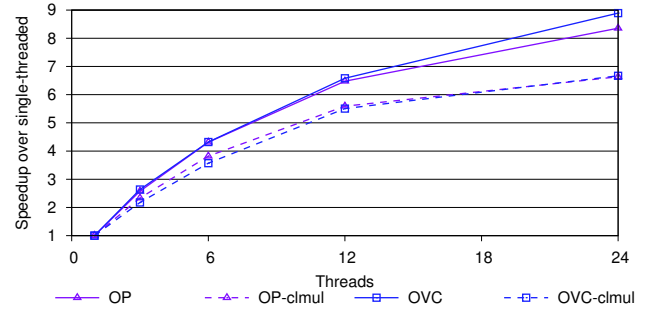


Figure 9. Scalability of the OP Compactor and OVC at $2.0\times$ minimum heap size for single-threaded OP. Both pairs of OP and OVC and OP-clmul and OVC-clmul scale similarly, and the branchy variants scale better than the clmul variants. We exclude batik and jme, which do not collect at this heap size, and h2 and h2o, which almost always time out at this heap size on a single thread.

compaction as the unifying concept that underpins a family of canonical collector designs that include the well-known Compressor [19], the less known, but widely-deployed Android implementation of the Compressor [13], and a number of virtually unknown implementations that emerged alongside the Compressor, mostly unpublished. We also provide the first implementation and evaluation of the newly developed One Pass Compactor [9]. The review, taxonomy, implementations and evaluations that we present here shed much-needed light on this important but poorly understood family of garbage collectors. We hope that by doing so, this work will spur more informed and more innovative collector designs.

References

- [1] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. 2004. An efficient parallel heap compaction algorithm. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 224–236. doi:10.1145/1028976.1028995
- [2] Bowen Alpern, C. Richard Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark F. Mergen, Janice C. Shepherd, and Stephen E. Smith. 1999. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1999, Denver, Colorado, USA, November 1-5, 1999*, Brent Hailpern, Linda M. Northrop, and A. Michael Berman (Eds.). ACM, 314–324. doi:10.1145/320384.320418
- [3] Andrew W. Appel and Kai Li. 1991. Virtual Memory Primitives for User Programs. In *ASPLOS-IV Proceedings - Forth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, USA, April 8-11, 1991*, David A. Patterson and Bob Rau (Eds.). ACM Press, 96–107. doi:10.1145/106972.106984
- [4] David F. Bacon, Perry Cheng, and David Grove. 2004. Garbage collection for embedded systems. In *Proceedings of the 4th ACM International Conference on Embedded Software (Pisa, Italy) (EMSOFT '04)*, Association for Computing Machinery, New York, NY, USA, 125–136.

- doi:10.1145/1017753.1017776
- [5] Ori Ben-Yitzhak, Irit Gofit, Elliot K. Kolodner, Kean Kuiper, and Victor Leikehman. 2002. An algorithm for parallel incremental compaction. In *Proceedings of the 3rd International Symposium on Memory Management* (Berlin, Germany) (ISMM '02). Association for Computing Machinery, New York, NY, USA, 100–105. doi:10.1145/512429.512442
 - [6] Stephen M Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. 2025. Rethinking Java Performance Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*. ACM. doi:10.1145/3669940.3707217
 - [7] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) (PLDI '08). Association for Computing Machinery, New York, NY, USA, 22–32. doi:10.1145/1375581.1375586
 - [8] Chris J. Cheney. 1970. A Nonrecursive List Compacting Algorithm. *Commun. ACM* 13, 11 (1970), 677–678. doi:10.1145/362790.362798
 - [9] Tomer Cory and Erez Petrank. 2024. The One Pass (OP) Compactor: An Intellectual Abstract. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management* (Copenhagen, Denmark) (ISMM 2024). Association for Computing Machinery, New York, NY, USA, 108–120. doi:10.1145/3652024.3665513
 - [10] Robert Fenichel and Jerome C. Yochelson. 1969. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM* 12, 11 (1969), 611–612. doi:10.1145/363269.363280
 - [11] Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. 2001. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001* (Monterey, California) (JVM'01). USENIX Association, USA, 21.
 - [12] Agner Fog. 2025. *Instruction tables*. Vol. 4. https://www.agner.org/optimize/instruction_tables.pdf
 - [13] Lokesh Gidra, Hans-J. Boehm, and Joel Fernandes. 2020. Utilizing the Linux Userfaultfd System Call in a Compaction Phase of a Garbage Collection Process. *Technical Disclosure Commons* (October 2020). https://www.tdcommons.org/dpubs_series/3671
 - [14] W. Daniel Hillis and Guy L. Steele. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (Dec. 1986), 1170–1183. doi:10.1145/7902.7903
 - [15] Richard Jones, Antony Hosking, and Eliot Moss. 2023. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (second ed.). CRC Press.
 - [16] H.B.M. Jonkers. 1979. A fast garbage compaction algorithm. *Inform. Process. Lett.* 9, 1 (1979), 26–30. doi:10.1016/0020-0190(79)90103-0
 - [17] Roman Kennke. 2024. Compact Identity Hashcode. <https://wiki.openjdk.org/spaces/lilliput/pages/124256303/Compact+Identity+HashCode>
 - [18] Roman Kennke. 2024. New Parallel Full GC. <https://wiki.openjdk.org/spaces/lilliput/pages/132022281/New+Parallel+Full+GC>
 - [19] Haim Kermany and Erez Petrank. 2006. The Compressor: concurrent, incremental, and parallel compaction. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) (PLDI '06). Association for Computing Machinery, New York, NY, USA, 354–363. doi:10.1145/1133981.1134023
 - [20] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *The VLDB Journal* 28, 6 (Oct. 2019), 941–960. doi:10.1007/s00778-019-00578-5
 - [21] Xiao-Feng Li, Ligang Wang, and Chen Yang. 2008. *A Fully Parallel LISP2 Compactor with Preservation of the Sliding Properties*. Springer-Verlag, Berlin, Heidelberg, 264–278. https://doi.org/10.1007/978-3-540-89740-8_18
 - [22] Clozure Common Lisp. 2001. OpenMCL 0.5.1 lisp-kernel/gc.c. <https://ccl.clozure.com/ftp/pub/ARCHIVE/openmcl-src-0.5.1.tar.gz>
 - [23] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST?. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Switzerland) (HOTOS'15). USENIX Association, USA, 14.
 - [24] Sun Microsystems. 2006. Memory Management in the Java HotSpot™ Virtual Machine. White Paper. <https://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf>
 - [25] OpenJDK developers. 2007. OpenJDK 6. https://github.com/openjdk/jdk6/blob/ece71901f6038f358f94359316c111b1b65a134a/hotspot/src/share/vm/gc_implementation/parallelScavenge/psParallelCompact.cpp
 - [26] Hayley Patton and Stephen M. Blackburn. 2026. Much Ado About Offset Vectors Artifact. <https://doi.org/10.5281/zenodo.20302085>
 - [27] P. Stygar. 1967. *LISP 2 Garbage Collector Specifications*. Technical Report TM-3417/500/00. System Development Corporation. https://softwarepreservation.computerhistory.org/LISP/lisp2/TM-3417_500_00_LISP2_GC_Spec.pdf
 - [28] Henry S. Warren, Jr. 2002. *Hacker's Delight*. Addison-Wesley Professional.
 - [29] Albert Yang. 2024. Parallel: Investigate Mark-Compact for Full GC to decrease memory usage. <https://bugs.openjdk.org/browse/JDK-8329203>
 - [30] Wenyu Zhao. 2026. JDK21 jme high memory overhead. <https://github.com/mmtk/mmtk-openjdk/issues/354>
 - [31] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2025. Work Packets: A New Abstraction for GC Software Engineering, Optimization, and Innovation. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 361 (Oct. 2025), 29 pages. doi:10.1145/3763139