Index-Compact Garbage Collection

Liangliang Tong and Francis C.M. Lau

Department of Computer Science,
The University of Hong Kong,
Pokfulam Road, Hong Kong
{lltong,fcmlau}@cs.hku.hk

Abstract. Automatic garbage collection is currently adopted by many object-oriented programming systems. Among the many variants, a mark-compact garbage collector offers high space efficiency and cheap object allocation, but suffers from poor virtual memory interactions. It needs to linearly scan through the entire available heap, triggering many page faults which may lead to excessively long collection time. We propose building an object reference index while tracing the heap, which in the following stages can be used to directly locate the live objects. As the dead objects are not touched, the collection time becomes dependent only on the size of the live data set. We have implemented a prototype in Jikes RVM, which shows promising results with the SPECjvm98 benchmarks.

Key words: Index, Virtual Memory, Compacting Garbage Collection

1 Introduction

In order to avoid the errors of manual memory management, the idea of a garbage collector to automatically reclaim dead objects was introduced [1]. But to precisely determine which objects will no longer be used by the program is undecidable. A somewhat conservative approach was therefore adopted which identified reachable objects by tracing the heap from the program roots [1], and many improvements followed, including some that took advantage of the presence or work around the limits of virtual memory.

Theoretically there is an unlimited address space in a virtual memory system. However, as the working set of a program increases its span in the virtual space, live objects mingle with dead objects and pages gradually become sparsely occupied (by live objects). Ultimately something must be done, otherwise many of the pages will be pushed to secondary storage which leads to frequent swaps [19]. Traditionally a free-list is used to mitigate the problem, unfortunately it would create memory fragments. So garbage collectors that move live objects together in space were devised and became popular.

There are two major kinds of moving garbage collectors: semi-space (also known as copying collector) and mark-compact. The former [14] is faster, but it needs to reserve half of available space for copying live objects. The latter [2] does not need to reserve any space, but takes much more time to do a collection. There are two reasons for the longer collection time:

- Compaction needs multiple passes over the objects, while copying takes only one pass.
- Some phases[1] of compaction will walk the entire available heap, including garbage objects, but semi-space collectors only need to touch[2] the live objects and hence their collection work is proportional only to the amount of live data [7].

Much research has been conducted to reduce the number of passes required by compaction, such as [3], but to the best of our knowledge nearly no attention has been paid to the second issue. Regarding this issue, we note that, as indicated in [17], unreachable objects tend to cluster together. In our experiments, the size of some clusters even exceeded that of a page. In the presence of virtual memory, such pages with only garbage are never or rarely visited and therefore should be evicted out of the main storage. To touch them will trigger many page faults hence prolong the operation time.

In this paper we propose an improvement to compacting collectors, called an index-compact garbage collector. It builds an address (index) table during marking. This index table contains all the references to the live objects in the available heap in address order. After all the live objects have been visited, this index is sorted by the values of the references to make it address ordered. In the following phases of garbage collection, the index is used to efficiently locate live objects for pointer adjustment and object compaction. Because the index is sorted, the corresponding movements of objects will not cause them to overlap and data will not be lost. During these phases, the garbage objects are never touched, which substantially reduces the working set of the garbage collector. We have implemented a prototype based on this idea on JikesRVM [12] and the experiment clearly showed that the collection time depends only on the size of live objects for the benchmarks tested.

The improvement does not come without a cost—we need at least extra space for the index and extra time to sort it. In the following, we expound on the overhead incurred by our algorithm and suggest several possible methods to mitigate its side effects. Considering that almost all the enterprise garbage collected systems are generational[9], we also give a separate discussion on how to build generations using our algorithm. Compared with copying, compaction saves resources but requires multiple phases to complete its work, so it will be interesting if we can somehow combine the two to achieve a balance between space and time.

Our contributions can be summarized as follows.

- We put forward the case that reducing page faults should be one of the main tasks of garbage collectors in a virtual memory based system.
- We propose the index-compact garbage collector which can avoid touching the garbage while compacting the working heap. The result is reduced page swaps, and the collection time can be made proportional only to the amount

[1] If a process needs to visit the heap from the start again, we call that a phase.
[2] An object is touched if any bytes in this object is visited.

of live objects. This mechanism can be even more effective if the collector is generational because of the higher infant mortality of young spaces.
– We have implemented a preliminary version of our collector in JikesRVM. The experiment behaved as expected and showed a collection time that is correlated with the size of live objects.
– We also suggest several techniques, such as cluster indexing and page remapping, that can further extend the proposed idea and improve the performance of the proposed collector. A fine-grain blending of copying and compacting collectors is discussed, which can achieve a balance between time and space costs.

The remainder of this paper is organized as follows. Sec. 2 provides a comparison between copying and compacting garbage collectors and gives the motivation for constructing an index for compacting collectors. Sec. 3 presents the basic design and implementation of our collector. Sec. 4 describes the experimental environment and reports the experimental results. Sec. 5 gives a discussion of the overheads and extensions of our algorithm. Related works are overviewed in Sec. 6. We summarize our contributions and point out possible future work in Sec. 7.

2 Comparison and Motivation

Semi-space collectors reserve half of the available heap and copy every reachable object to that space. Because the reserved space contains no object at the beginning of collection, there is no need to consider whether different objects may overlap or not. The active object tree is traced on the fly and every reached object is copied to the reserved half heap. After the collection, the live objects align in the new space by breadth-first order regardless of the addresses they are originally stored at. The situation of mark-compact collectors is different: live objects must be compacted in address order, or different objects may be moved to the same place and data will be damaged. We illustrate this situation in Fig. 1.

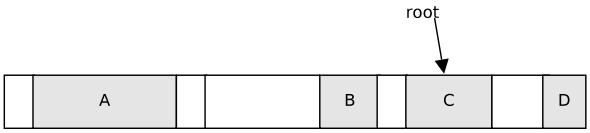


Fig. 1: The Traced Heap

In this figure, garbage objects are colored white, and live objects grey. Assume at the moment the root points at object C which is now marked and needs to be relocated. If this is a copying collector, this object will be immediately copied to the new space, and its header will store a forward pointer so that the following

pointers to this object can be updated. But in a compacting garbage collector, no extra space is reserved, and so this object must be moved to the start of the heap. If we do so, however, C will land right on the live object A, damaging its content. Therefore, a compacting garbage collector must first linearly scan through the whole available heap for live objects and mark them. Then beginning from the start of the heap, the collector walks through the objects (including the dead ones), and when encountering a live one, say A, it relocates the object to the start of the heap; and similarly for the following marked objects, which are, B, C and D.

Touching garbage objects can be detrimental, since they mainly reside on secondary storage, and this might trigger a page fault. It also unnecessarily enlarges the program's working set, pollutes the cache memory with the garbage, and leads to mass misses as a result. In view of this undesirable situation, we need a mechanism to keep track of live objects in address order after tracing the entire heap. In this paper we propose such a mechanism which employs an index table to store every live object reference.

3 Index-Compact Garbage Collector

3.1 Design

Traditionally a compacting garbage collector reclaims memory in four phases:

1. Compute the root set of the running program and push them into a FIFO queue. To start the tracing, pop an object reference out of the queue and completely scan it for pointers. The objects referred to by any pointers are marked and pushed into the queue. This operation continues until the queue becomes empty, at which time all the reachable objects have been marked as alive.
2. Scan linearly through the available heap where objects are allocated and calculate the forward addresses for the marked objects by adding up their sizes to the heap's start address.
3. Trace the active program tree again and update the pointers to the forward addresses.
4. Walk sequentially through the heap and move the marked objects to their forward addresses.

It can be seen that at least phases 2 and 4 need to touch (specifically to check the mark bit of) the garbage objects because there is no auxiliary information on how to locate just the live objects. If we can create and maintain a global data structure to store this information, we can skip over the garbage objects completely.

Fig. 2 shows an address index table where each entry points at the start address of an active object. In phases 2, 3 and 4, this index can be used to directly locate the live objects. With this index in place, a compacting garbage collector works as follows:

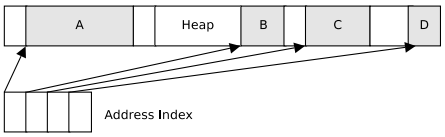


Fig. 2: The Index-Compact Garbage Collector

1. Compute the root set, and push all the object references into the *index* and iteratively trace them. Note that this time the object references are not popped out of the index. After completion, the index is sorted by the reference address values of the items.
2. Calculate the forward addresses using this index. Touching of garbage is therefore avoided.
3. Update the pointers of objects referred to by items in the index to their forward addresses.
4. Pop every item in the index and move the object pointed by it to the forward address.

The above descriptions shows that the index stores only the references to the reached objects, thus the garbage will never be touched. Consequently the number of page faults will be reduced. In this paper, we present a simple algorithm for our idea for the sake of understanding, and leave any enhancements which we will discuss in the following paragraphs to future implementations.

3.2 Implementation

Where to store the index is an issue. Since the index stores all the pointers to live objects, so it must be efficient. We cannot use the Java classes to implement a linked list for this purpose, because that will bring in extra object headers. In Java [30] this overhead comprises two words, which is too costly and will triple the overall size of the index.

We notice that every compacting garbage collector has some auxiliary data structure, such as the trace queue, which must be stored somewhere in the heap. The size of these data is largely unpredictable, and thus in real-life platforms the address space allocated for them is extremely large in order to cope with any unexpected cases. Because they are meta data there is no header to consume extra spaces. We therefore store the index in such an area.

Since this area has other usages with different data intersecting with each other, some data structure must be put in place to differentiate them. In this paper, we partition this area into 4KB blocks (whose size is identical to the page adopted by most current computer systems) and store two pointers (*next* and *pre*) at the end of each block allocated for the index. Inside each block the object references are stored in array style. Once a block is exhausted, we allocate a new block, and set up the *next* and *pre* pointers of the two blocks. This is depicted in Fig. 3. So for every block only two pointers are maintained, corresponding to a space overhead of less than 0.1%.

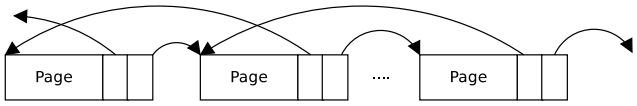


Fig. 3: The Structure of Overall Index

The index also eliminates the need of building a tracing queue and marking. When a garbage collection is triggered due to memory exhaustion, the root set is scanned and their object references are pushed into the index. We create an iterator to point to its first item. Then one by one, every item is checked for pointers. The reference of every object reached is added to the end of the index. After the object is entirely scanned, the iterator moves on to the next object and this process repeats until it meets the end of the index. We use an MSD radix exchange algorithm to sort this index, which is relatively quick and requires no extra space. This is also the reason why the index blocks are doubly linked, because this kind of algorithm needs to search from both top and bottom.

4 Methodology

Based on where to store the forward address, there are three types of compacting garbage collectors: Lisp-2 [6], break table [8], and threading [18]. Our algorithm can be applied to all of them, but we only select Lisp-2 to work on for illustration's sake. Similar improvements can be achieved for threading compactors by avoiding touching the garbage, and better optimizations are possible for table-based compaction as the break table can be completely removed.

4.1 Experimental Setup

The computer which runs our experiments has a 2 GHz Intel Core 2 Duo CPU and 2 GB main memory. Every core has an independent 8-way associative 32 KB L1 cache and shares a 4 MB L2 cache. We use Ubuntu 8.04 operating system [29] with kernel version 2.6.24-24.

Our collector is implemented on MMTk [11] of the Jikes RVM [12]. MMTk partitions the address heap of the RVM into the several spaces: metadata, immortal, large object and small object space. We modify the current mark-compact collector and create the index in the metadata space. The iterator is placed in the immortal space, since it will always be needed during the entire program execution. Literally the large object space stores objects that are larger than 32KB, and normal allocations and collections happen in the small object space.

All the applications in SPECjvm98 are tested in our experiments except *mpegaudio* which rarely allocates any new objects and triggers almost no collection. We calculate the average size consumed by the index at every collection and subtract it from the working heap space. In this way, the sizes of memory used by both the mark and the index compactor are approximately the same. In the

experiment we found that the index seldom exceeds 1 MB (See Fig. 1, and so the initial average size is set to this number.

The only assumption for our collector to work well is what makes a garbage collector run efficiently: the heap residency of an application, which is the ratio between the size of live objects and the heap size, must be low enough so that there is room for new allocations. We did not include other benchmarks, but it can be expected that if the heap residency is not too high then they will also present good performance. Average object size being too small may also affect the collector's efficiency, for it will result in an overly large index table. We figure that the minimum size of Java objects is 8 bytes (to store the header), and in fact, many previous experiments have suggested that the average size of Java object ranges from 20 bytes to 60 bytes, which will work fine for our algorithm.

4.2 Results

Table 1: Object Characteristics for SPECjvm98 Benchmarks

Benchmark	Average Object Size (bytes)	Average Cluster Number	Average Index Size (bytes)
compress	513	252	348720
db	26	1070	377265
jack	37	912	443048
javac	31	23263	1516896
jess	34	1204	503380
mtrt	24	780	973696

We firstly profiled MMTk to obtain the dynamic object characteristics of the SPECjvm98 benchmarks, which are summarized in Tab. 1. The table shows that the average object size is small, which is bad news for us because this means the number of objects would be large and correspondingly so would be the size of the index. For this particular situation, we offer several optimizations in Sec. 5.

Attention must be paid to the second column of the table, which represents a very common phenomenon of memory usage: objects are created en masse, and they also tend to die together. Although the third column suggests that the size of the index sometimes grows beyond 1 MB, the number of object clusters[3] remains moderate. This motivates us to propose in the discussion section the cluster-wise idea, as opposed to the simple address-wise way of building the index. Yet by employing our simple, address-wise indexing algorithm, the results are still encouraging.

[3] An object cluster is a continuous heap block with only live objects.

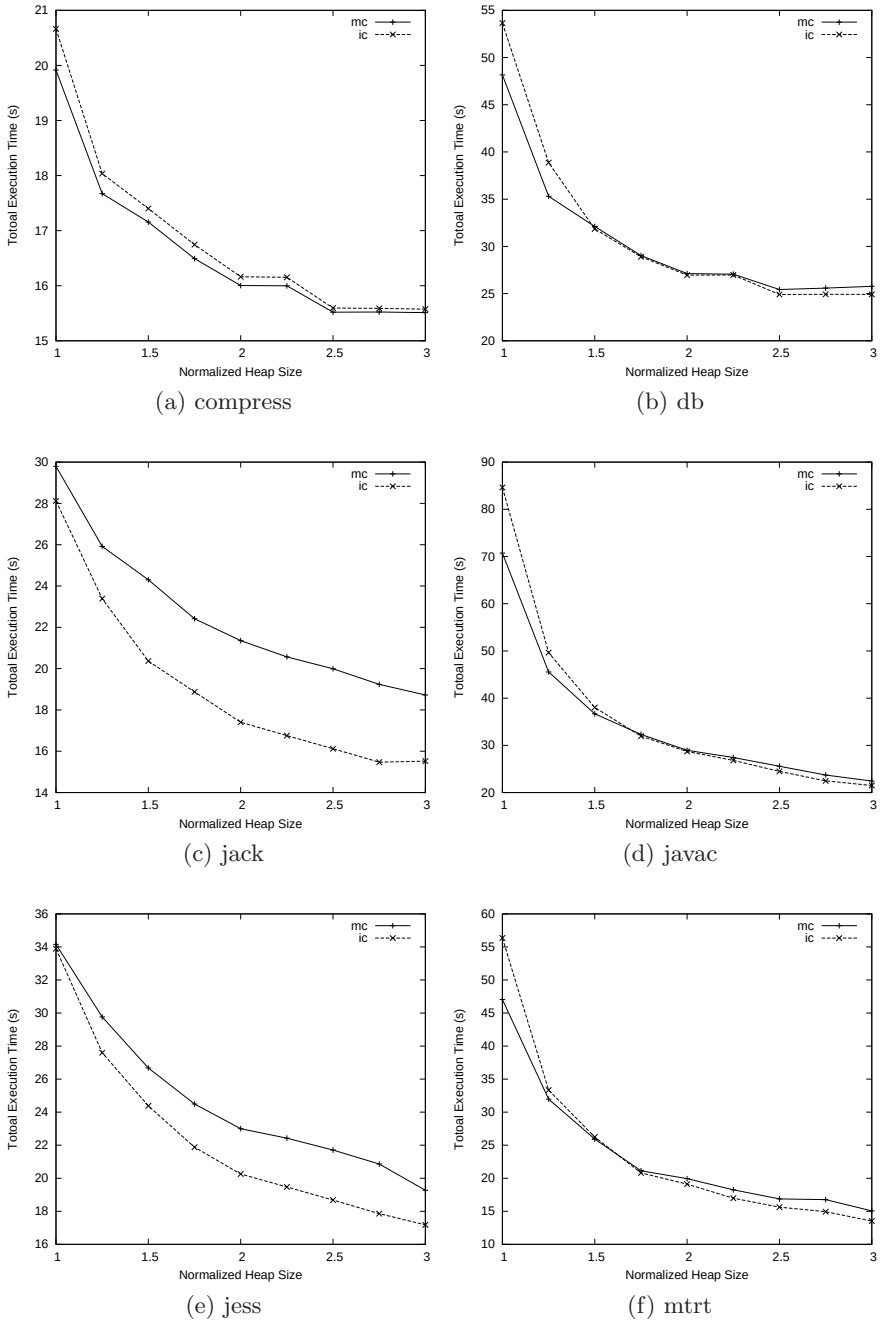
We implement the index compactor (*ic*) and compare its performance with that of a Lisp-2 mark compactor (*mc*), as reported in the following figures. Fig. 4[4] shows the overall benchmark execution times for both compactors. In the figure, two benchmarks, *jess* and *jack*, clearly demonstrate the superiority of *ic*, while for three other benchmarks, *db*, *javac* and *mtrt*, *ic* only wins after the heap grows beyond a certain size. This is reasonable, since the advantage of our compactor comes from not touching the garbage objects. The live object (survival) rate of the former two benchmarks is as low as 30% even for a $20MB$ heap; this rate would not come down for the latter three benchmarks until the heap grows to a certain extent. We tuned the heap size for these benchmarks, and found that the turning point is approximately at 40%. That is, for *ic* to outperform *mc* the heap residency should be less than this turning point rate. Furthermore, it is also the turning point where the execution time drops dramatically, since garbage collectors require enough space to work well.

This characteristic makes our algorithm perfect for applying to the young space of a generational collector, where this rate is well below 10%. Because of the same reason, the performance of the benchmark *compress* degrades with *ic*. After allocating about $4MB$ of normal objects in the small object space (where our algorithm is used), the program only creates large objects in the large object space. It can be seen from Tab. 1 that its average object size is very large as compared to that of the other benchmarks. To make the situation worse, the $4MB$ small objects are never disposed of until the end of the program execution, which pushes the live object rate to be close to 100% at every collection. In a nutshell, for *compress*, an index is redundant, because all the objects are alive. The degradation in performance as compared to *mc* is due to the extra computation time to build and sort the index.

Fig. 5 compares the collection times of the two compactors. It portrays a similar picture to Fig. 4, and shows a difference that increases monotonically as the heap grows. This is within our expectation that the performance of our algorithm improves as the heap residency reduces. Because of the similarity between *ic* and *mc*, there is virtually no difference in mutation time for both collectors. Combining the temporal performance of the above two figures, it can be perceived that the size of index table matters a lot. *javac* and *mtrt* generate the biggest indexes. As a result, the performance of *ic* will not outstrip *mc* until the heap approaches $40MB$. To reduce the size of index table, we propose several methods in the discussion section.

As we have stressed, *ic* does not touch the garbage objects at all, which contributes to the interesting characteristic of *ic* as presented in Fig. 6. In the figure, we can easily spot that the average collection time of *ic* is insensitive to the heap size, whereas this time increases as the heap grows for *mc*. It is worth pointing out that as the heap grows the number of collections decreases, and this is why the total collection number keeps falling while the average collection time remains roughly unchanged.

[4] The size of the heap is normalized by 20MB. That is, 1 denotes 20MB, 2 denotes 40MB, etc.

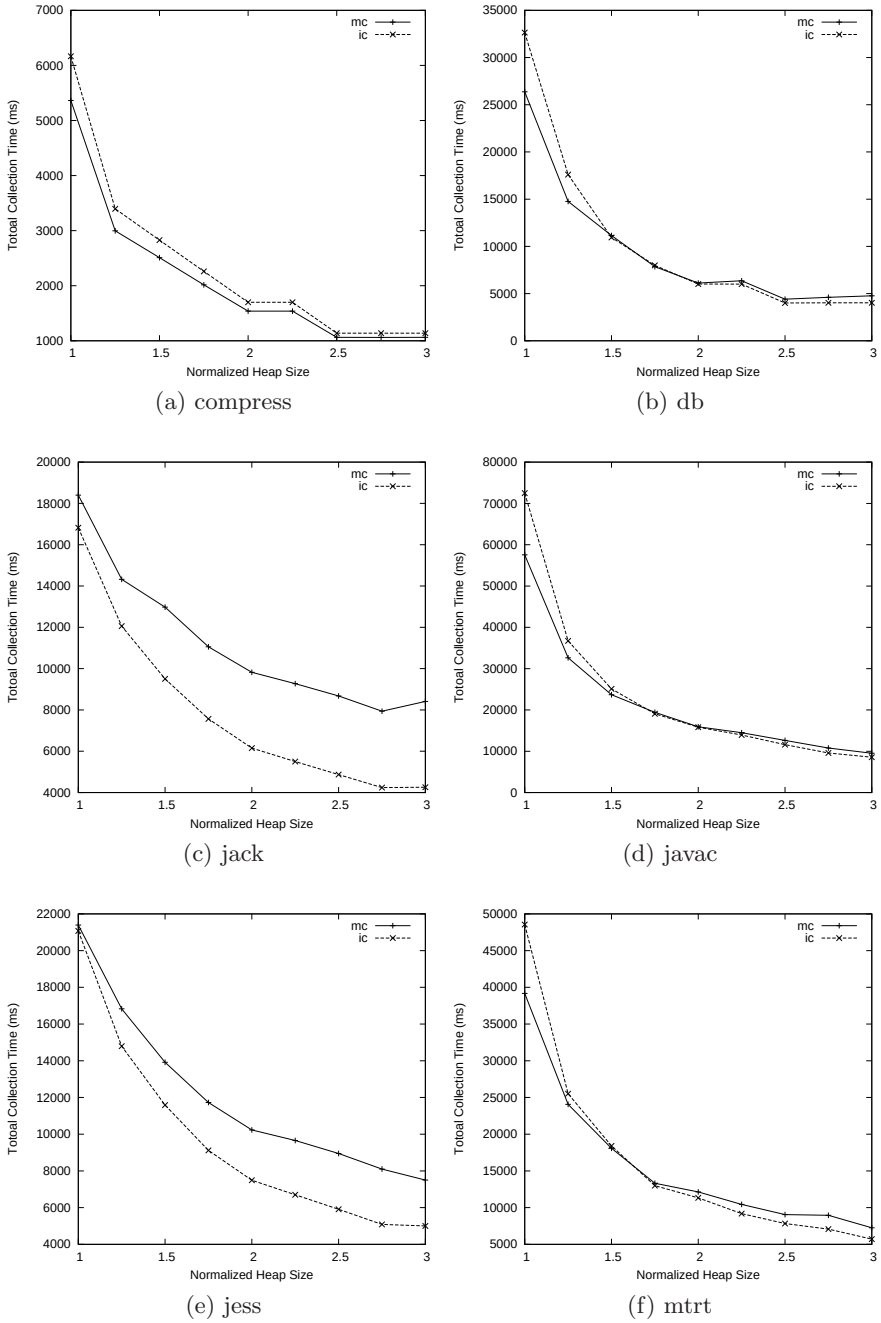


(a) compress (b) db

(c) jack (d) javac

(e) jess (f) mtrt

Fig. 4: Total Execution Time for Mark- and Index-Compact Collectors

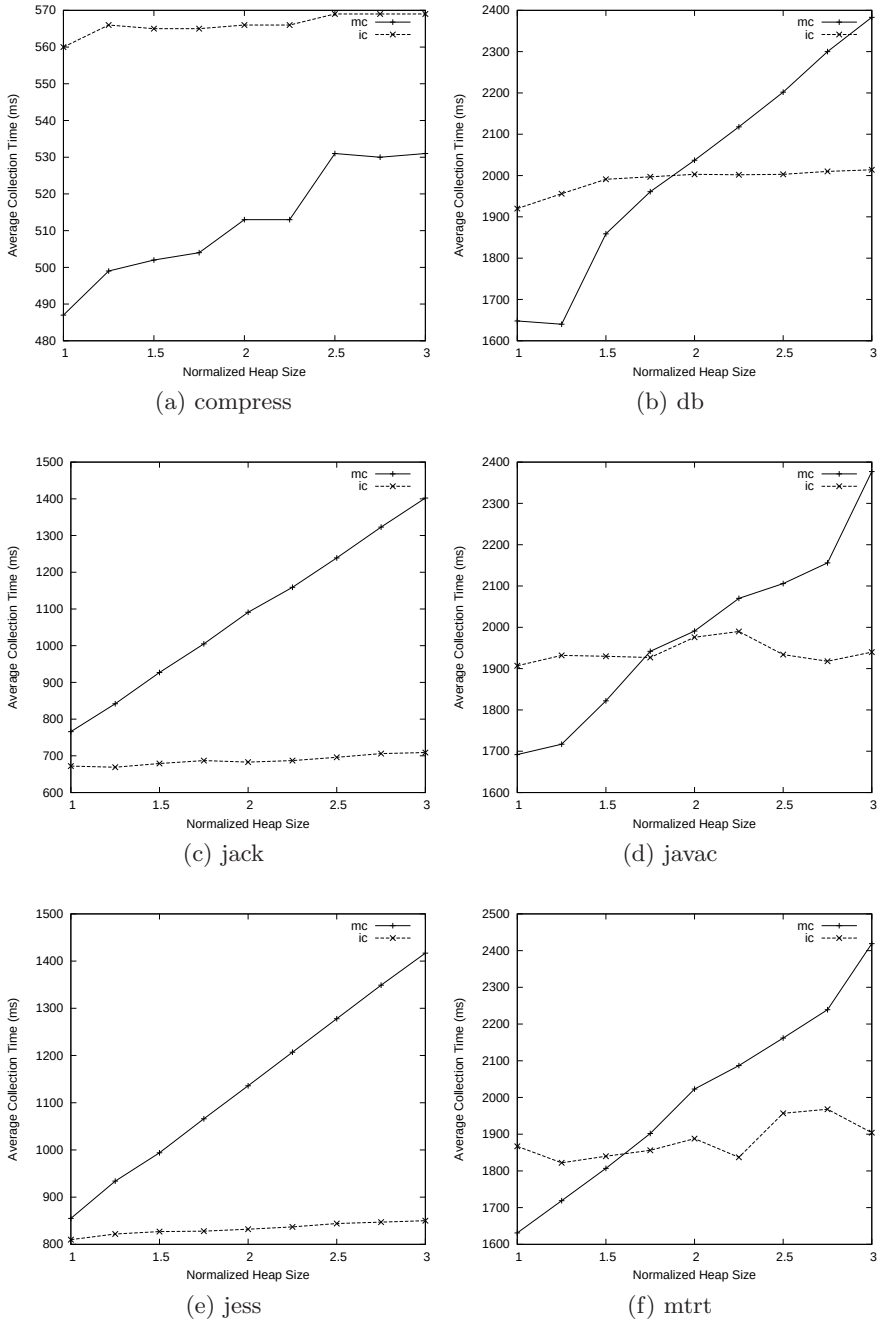


(a) compress (b) db

(c) jack (d) javac

(e) jess (f) mtrt

Fig. 5: Total Collection Time for Mark- and Index-Compact Collectors



(a) compress (b) db

(c) jack (d) javac

(e) jess (f) mtrt

Fig. 6: Average Collection Time for Mark- and Index-Compact Collectors

5 Discussions

5.1 Improvement Techniques

The above experiments show that after introducing an index table to guide the compactors, the collection time can be made dependent only on the size of live objects, instead of the size of heap which is the case for traditional compactors. For most of the time the two compactor versions operate in a similar fashion. Our algorithm requires some extra time to sort the index. Therefore, it must be due to avoiding touching garbage that the pages faults triggered by our collectors are reduce. As a result, the overall collection time is reduced.

Compared with traditional mark-compact, our compactor needs extra space to store the index and extra time to sort the index. In [28] it is revealed that most of the objects created in typical programs tend to be very small (See Tab. 1) and the number of objects tends to be large. For the SPECjvm98 benchmarks, the average object size ranges mainly from 24 to 37 bytes, whereas an index entry takes up four bytes (in order to represent an address in a 32-bit machine). It means that the size of the index can grow to be as large as one eighth of the total size of live objects, and occasionally it can be larger than $1MB$.

As the size of the index grows, so does the space needed to store and the time spent to sort it. In Sec. 4, we mention that live objects are likely to cluster together. The number of clusters can be considerably smaller than that of live objects. Tab. 1 shows that this number falls well below 1000 for most of the benchmarks. This gives us a good opportunity to adopt another way of building the index. Fig. 7 shows a cluster-wise index, where every entry contains two pointers, pointing to the start and the end of a cluster respectively. Note that this time each index entry must be stored as a node of a linked list. We cannot construct an array for the index any more, because the tracing may not be address ordered. For instance, if the first and third cluster in Fig. 7 are traversed before the second, then an array structure is not adequate for handling this case.

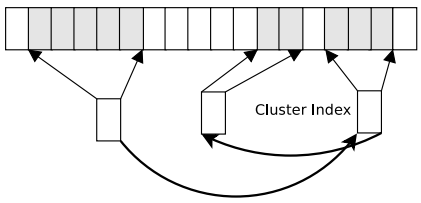


Fig. 7: The Cluster-wise Index Table

Having a cluster-wise index eases the pressure on space, but it may increase the computation time. In order to make sure that every first reference address of the block index is ordered, the index must be built as a linked list and every insertion requires a search for the desired insertion point. If the objects do not

cluster as much, such as the case of *javac*, this process will introduce considerable overhead. Furthermore, because the insertion happens for every live object, it will be better to build a hash for this index to accelerate searching. Because of space limitation and that the purpose of this paper is to introduce the basic indexing idea, we skip further details on and results for the cluster-wise variant.

In the experiments we spotted that the residency of a considerable portion of the pages was nearly full. The extreme example is *compress* whose pages in the small object space are virtually all filled with live objects. For these pages, there is no need for compaction. Instead, we can remap the virtual addresses of these full pages so that they become continuous and update those pointers pointing to them. This would substantially decrease the size of the index and avoid the cost of moving full pages. For *compress*, in particular, compaction can be totally avoided in the small object space, which should help our collector to outstrip other normal compactors even when the heap becomes densely populated.

5.2 Generational Variant

Because most of the state-of-the-art collectors are generational, we suggest here how our algorithm can be applied to these collectors. As said before, the object mortality rate in the young space is much lower than that of the whole heap. It can be observed that most of the time, the survival rate is well below 10%. Since 40% is the observed turning point, it can be expected that our algorithm will perform excellently there. For older spaces, a free-list collector is probably enough, since it will not be touched as often. To fight against memory fragments, a compactor can be triggered from time to time to tidy up the room for these older spaces.

Copying-based collectors are most desirable for young space, because of their time efficiency. However, the low survival rate there makes it space inefficient, as it still needs to conserve half the available heap for copying live objects. Our collector can give a hand at this juncture to achieve a space-time balance for garbage collection, using an algorithm as described in [10]. This algorithm would manually set the portion of reserved space to be 30% of the working space, and fall back to a mark-compactor if this prediction turns out wrong during actual collection. This fallback compactor can now be replaced by our index-compact collector to avoid touching the garbage. Most of the objects in the to-be-compacted area would have already been copied to the reserved space, and what remains is a sparsely populated space for which our compactor will work better than an ordinary mark compactor. We will report our implementation of these ideas in a future paper.

6 Related Works

Since our algorithm is concerned with compaction and virtual memory, we briefly introduce several existing works that fall into these domains. For details on their implementation, please refer to [7].

6.1 Compact Collectors

Implementations of compacting garbage collectors can be classified into three classes: Lisp-2 [6], table-based [8] and threading [18][4]. In [20], a comparison between different compacting algorithms is given, and the authors argue that Lisp-2 is the most time efficient collector. Yet since all of these compactors need four phases and two heap passes to complete, none of them are good enough to be used widely in real-life systems.

Because compactors would move objects, they are frequently employed as an auxiliary method to curb memory fragmentation for non-moving collectors [22][21]. To take advantage of compactors' space efficiency over copying collectors, [16] designs a hot-swapping mechanism based on memory residency, and [10] resorts to compaction in case when its copying reserve prediction falls through. Note that in [10], the fallback compactor needs to touch the entire heap even after most of garbage have been collected by the preceding copying collector. Our algorithm should be a much better choice at this point than traditional compactors, as we have explained in the previous section. There are also research efforts on how to cut down on the phases required for compaction. For example, [3] combines marking and compaction into a two-step algorithm with one phase. But in any case, all of them need to touch the garbage objects.

6.2 VM-Aware Design

To design a VM-aware collector, some researchers have focused directly on reducing the overall consumed memory, for example, via object reuse [23]. Although there are works on how to reap the merits of virtual memory system [26][25][27], these proposed mechanisms are mostly ignored by current garbage collectors, as pointed out in [24] where the authors propose to build barriers between secondary storage and the main memory in order to avoid collection paging. We should also mention [5] which describes a concurrent, incremental and parallel algorithm designed for compactors. This collector uses two equal virtual address spaces to perform a copying-like compaction without touching the garbage. Their implementation is based on a markbit vector, whereas our algorithm uses an object reference index which is more portable and extensible to other usage scenarios, as has been explained in Sec. 5.

7 Conclusion

The slow collection time of compactors is a well known headache. They need to traverse the heap multiple times; while touching the garbage objects they trigger lots of page swaps. Many researchers have presented different techniques to reduce the number of phases in compaction, but little has been done on the problem of page swapping.

In this paper, we argue that virtual memory performance is one of the most important factors in the performance of garbage collection, and every garbage

collector should endeavor to minimize page faults. We then designed an indexing algorithm that can avoid touching the garbage objects for compactors. We have implemented a preliminary basic version of the algorithm and its collector in Jikes RVM. The results confirmed our point about page swapping during garbage collection, and showed improved overall performance over traditional mark compactors. To furthermore enhance our collector's performance, we have sketched out several related advanced methods, including an application of our algorithm in a generational collector.

Further work can be done to make this compactor more suitable for real use, such as to optimize the sorting algorithm, to make the compaction parallel, to reduce the phases by storing the relocation pointers in the index, etc. In real life, our algorithm may not be suitable for certain programs, for example Lisp programs whose objects are typically even smaller than those of Java. It will be an interesting exploration to see if we can dynamically decide whether to use an index or fall back to a traditional compactor.

Acknowledgement This work is supported in part by a Hong Kong RGC CERG grant (7141/06E). We are thankful to the anonymous reviewers and Prof. Ueda for their excellent comments and great help in the final stage of the writing.

References

1. J. McCarthy: Recursive Functions Symbolic Expressions and Their Computation by Machine. In: Communication of the ACM, Volume 3, Number 4, 184-195 (1960)
2. R.A. Saunders: The LISP System for the Q-32 Computer. In: Berkeley and Bobrow, 220-231 (1964)
3. J.J. Martin: An efficient garbage compaction algorithm. In: Communications of the ACM, Volume 25, Number 8, 571-580 (1982)
4. F.L. Morris: A Time- and Space- Efficient Garbage Compaction Algorithm. In: Communications of the ACM, Volume 21, Issue 8, 662-665 (1978)
5. H. Kermany and E. Petrank: The Compressor: Concurrent, Incremental, and Parallel Compaction. In: ACM Conference on Programming Language Design and Implementation, 354-363 (2006)
6. R. Jones, R. Lins: Garbage Collection: Algorithm for Automatic Dynamic Memory Management. John Wiley&Sons (1997)
7. P.R. Wilson: Uniprocessor Garbage Collection Techniques. In: Proceedings of the International Workshop on Memory Management, 1-42 (1992)
8. B.K. Haddon, W.M. Waite: A Compaction Procedure for Variable Length Storage Element. In: The Computer Journal, Volume 10, Number 2, 162-165 (1967)
9. H. Lieberman, C. Hewitt: A Real-time Garbage Collection Based on the Lifetimes of Objects. In: Communication of the ACM, Volume 26, Number 6, 419-429 (1983)
10. P. MaGachey, A.L. Hosking: Reducing Generational Copy Reserve Overhead with Fallback Compaction. In: International Symposium on Memory Management, 17-28 (2006)
11. S.M. Blackburn, P. Cheng, K. S. McKinley: Oil and Water? High Performance Garbage Collection in Java with MMTk. In: International Conference on Software Engineering, 137-146 (2004)

12. B. Alpern, S. Augart, S.M. Blackburn: The Jikes Research Virtual Machine Project: Building an Open-source Research Community. In: IBM Systems Journal special issue on Open Source Software, Volume 44, Number 2, 399-417 (2005)
13. B. Alpern and C. R. Attanasio, J. J. Barton: The Jalapeno Virtual Machine. IBM Systems Journal, volume 39, number 1, 211-238 (2000)
14. C.J. Cheney: A Nonrecursive List Compacting Algorithm. In: Communication of the ACM, Volume 13, Number 11, 677-678 (1970)
15. H.B.M. Jonkers: A Fast Garbage Compaction Algorithm. In: Information Processing Letters, Volume 9, Number 9, 25-30 (1979)
16. P. M. Sansom: Combining Single-Space and Two-Space Compacting Garbage Collectors. In: Proceedings of the Glasgow Workshop on Functional Programming (1991)
17. M. Wegiel and C. Krintz: The mapping collector: virtual memory support for generational, parallel, and concurrent compaction. In: International Conference on Architectural Support for Programming Languages and Operating Systems, 91-102 (2008)
18. D.A. Fisher: Bounded Workspace Garbage Collection in an Address Order Preserving List Processing Environment. In: Information Processing Letters, Volume 3, Issue 1, 25-32 (1974)
19. H. D. Baecker: Garbage Collection for Virtual Memory Computer Systems. In: Communications of the ACM, Volume 15, Number 11, 981-986 (1972)
20. J. Cohen and A. Nicolau: Comparison of Compacting Algorithms for Garbage Collection. In: ACM Transactions on Programming Languages and Systems, Volume 5 , Number 4, 532-553 (1983)
21. Y. Ossia, O.B. Yitzhak and M. Segal: Mostly Concurrent Compaction for Mark-Sweep GC. In: International Symposium on Memory Management, 25-36 (2004)
22. T. Printezis: Hot-swapping between a mark&sweep and a mark&compact garbage collector in a generational environment. In: Symposium on JavaTM Virtual Machine Research and Technology Symposium, 20-32 (2001)
23. Z.C.H. Yu, F.C.M. Lau and C.-L. Wang: Exploiting Java Objects Behavior for Memory Management and Optimizations. In: Asian Symposium on Programming Language and Systems, 437-452 (2004)
24. M. Hertz, Y. Feng and E.D. Berger: Garbage collection without paging. In: ACM SIGPLAN conference on Programming language design and implementation, 143-153 (2005)
25. T. Yang, E.D. Berger and S.F. Kaplan: CRAMM: virtual memory support for garbage-collected applications. In: Symposium on Operating systems design and implementation, 103-116 (2006)
26. P.R. Wilson, M.S. Lam, T.G. Moher: Effective "Static-graph" Reorganization to Improve Locality in Garbage-Collected Systems. In: ACM SIGPLAN Notices, Volume 26, Issue 6, 177-191 (1991)
27. D. Spoonhower, G. Blelloch and R. Harper: Using Page Residency to Balance Tradeoffs in Tracing Garbage Collection. In: ACM/USENIX international conference on Virtual execution environments, 57-67 (2005)
28. Y. Shuf, M. Gupta, R. Bordawekar and J.R. Singh: Exploiting Prolific Types for Memory Management and Optimizations. In: ACM Symposium on Principles of Programming Languages, 295-306 (2002)
29. The Ubuntu Operating System. `http://www.ubuntu.com`
30. The Java Hotspot Virtual Machine, White Paper. `http://java.sun.com/products/hotspot/index.html`
31. The SPEC Java Virtual Machine Benchmarks. `http://spec.org/jvm98`