Exploiting Memory Usage Patterns to Improve Garbage Collections in Java

Liangliang Tong
Department of Computer Science
The University of Hong Kong
lltong@cs.hku.hk

Francis C.M. Lau
Department of Computer Science
The University of Hong Kong
fcmlau@cs.hku.hk

Abstract

Copying-based garbage collectors are currently widely employed in JVM systems, as they provide not only cheap allocations but also fast collections. Comparing to their compacting-based counterparts, copying-based collectors trade space for time: they conservatively reserve half of the available heap for the purpose of copying live objects.[1] It is a common belief, however, that objects' survival rates are generally too low to make full use of the reserved memory. We find through experiments that the total live object sizes of Java programs are generally small and remain relatively stable over many collections, which provides a perfect opportunity for optimization. We analyze this phenomenon and propose a "skew-space"[2] collector that would reserve spaces of dynamically adjusted sizes coming from online predictions. The proposed collector has been realized using MMTk in the JikesRVM, and has shown promising improvements in the total execution time for the SPECjvm98 and DaCapo benchmarks.

Categories and Subject Descriptors D.3.4 [*Processors*]: Memory management (garbage collection)

General Terms Management, Algorithms, Measurement

Keywords Skew Space, Mark Compact, Semi Space

1. Introduction

Automatic dynamic memory management, also referred to as garbage collection (GC), was devised by John McCarthy while designing Lisp in the late 1950s [1]. It is commonly regarded as one of the biggest contributions in connection with the Lisp language [2]. GC has since been adopted in the design of many object-oriented languages such as Java.

There are in general two different approaches to constructing garbage collectors: reference counting and tracing [3]. Tracing-based collectors are usually more preferred because, unlike reference counting, they do not need extra space to store count numbers and can handle cyclic objects. This paper is about tracing-based collectors that move objects when performing GC—that is, compacting and copying collectors. For compacting collectors, the heap is traced to set apart the live and dead objects, and then the entire heap is traversed several times in order to compact the live objects into the lower end of the heap [4][17]. A typical copying collector [16] divides the heap into two equal spaces: the working space and the free space. Objects are allocated into the working space which when full triggers a GC. When this happens, the working space is traced and live objects are copied to the free space. The free space will then become the new working space for the next round. As the spaces are absolutely clean after every collection, moving-based collectors have the following merits:

- Both compacting and copying collectors move all the live objects into a contiguous space, which makes allocation fast and cheap: it amounts to simply advancing the bump pointer. Fragments are eliminated since all the free memory blocks are compacted during the process.
- Objects can be relocated to achieve placements that are cache-conscious.

The time to do a collection is an important performance measure. For copying-based collectors, the time depends on the live objects; whereas for compacting collectors, it depends on the entire heap. So copying-based collectors work much faster, and are thus widely employed in practice, particularly for handling the nursery space of generational garbage collectors [5][23]. However, the price to pay is dear as half of the memory space needs to be reserved for the following garbage collection all the time. Copying collectors are also called semispace collectors.

Copying collectors trade space for time. But whether there are always so many surviving objects to fill the reserved space is an issue. Obviously, at the time of GC, the amount of survivors must not be excessive in order that the collector can finish its work swiftly and there will be sufficient space left for the new objects. Otherwise, collection will be triggered too frequently, which is unacceptable. If indeed only a modestly substantial fraction of the objects in the working space would survive, then reserving half of the heap is too much and as a result, much memory resources are wasted. But can we reserve less than half of the space? This paper gives an affirmative answer which hinges on the assumption that the total size of live objects after each collection can be known. We describe a technique to predict such sizes, which is based on analyzing the pattern of the total size of live objects (or *live object size* in short) over time in typical programs.

We refer to the number of memory blocks needed to accommodate the surviving live objects as the memory requirement of a

[1] The group of *reachable* objects is the parent set of *live* objects. In a garbage collected language, reached objects are treated as live, and so in this paper we decide to use these two terms interchangeably.

[2] A skew-space collector reserves less than half of the heap space for copying, which is a key difference from semi-space collectors.

PPPJ '10 September 15–17, 2010, Vienna, Austria.
Copyright © 2010 ACM 978-1-4503-0269-2...$10.00

program. We conjecture that there is a maximum memory requirement for every program, and that the total size of live objects after a GC would remain similar to that after the previous collection. They should be true for most programs. Our conjecture is based on the following reasoning.

- If the memory requirement of a program keeps rising as if there is no limit, then this program will eventually exhaust the physical memory resource and generate a run-time exception. The maximum memory requirement should be considerably less than the reserved space in order that garbage collection will not be triggered too frequently.
- In many programs, for most of the time, the same code in a loop is repeatedly executed. This is a cyclic memory usage pattern.
- Many objects have a relatively short life time. If collections are not too frequent, allocations and de-allocations may just balance out space-wise.
- There are occasions when massive allocations happen. These objects however will likely be de-allocated soon, or otherwise memory resources will run out quickly.

A similar claim that live object sizes change little over time is vaguely given in [26] but it was neither expounded on nor supported by experiments. If ordinary programs indeed have the above-mentioned behaviors, we can then devise an algorithm to predict their memory usages and reserve a suitable amount of memory accordingly. In order to empirically test these conjectures, we carried out an experiment using all the applications in SPECjvm98 [30]. The results indicate that the memory requirement of most of the benchmarks is upper-bounded, and the total size of live objects remains relatively stable over the series of collections. These findings motivated us to deviate from the conventional half-sizing approach. We propose a dynamic free space reserving algorithm whereby the size of the reserved space is set according to the total size of the previous survivors. Because the survivor rate (i.e., percentage of objects surviving) tends to be low typically, the reserved space can be much smaller than before, leaving more space for object allocations and reducing the number of GCs. When in some rare cases the survivors' size exceeds the size of reserved space, a compacting collector is triggered to collect the remaining objects. This fallback strategy is similar to the approach adopted by [7], except that we customize it to fit a non-generational collector here.

The contributions of our work are as follows.

- We analyzed the memory usages of many Java programs, which showed that the live object size tends to reach a maximum value after some time and remain stable after each garbage collection.
- We propose a "skew-space" garbage collection algorithm that utilizes the above phenomenon to reduce the reserved space of copying-based collectors, show its compatibility with other improvement techniques, and analyze its applicability to bounded-size generational collectors.
- We implemented our skew space collector in JikesRVM [11] and evaluated its pros and cons.

The remainder of this paper is organized as follows. Sec. 2 gives an analysis based on the SPECjvm98 benckmarks, which provides a strong support to our conjecture. We propose a prediction-based strategy in Sec. 3, and describe a possible implementation of the proposed strategy. Sec. 4 reports on the experiments we have carried out and their results followed by some discussion on applying our algorithm to generational collectors in Sec.5. Related works are discussed in Sec. 6. We summarize our contributions and discuss possible future work in Sec. 7.

2. Live Objects and Their Sizes

In object-oriented programs, such as Java and C# programs, objects are constantly born and later become dead in the heap. Because of limited memory resources, garbage collectors automatically delete the dead objects in order to save space for the newly created ones. It is very rare that all objects would live forever and continue to occupy the address space at all times; such situations would easily exhaust the system's memory as more objects are being created.

Copying-based collectors divide the available heap into two parts: FromSpace and ToSpace. These two parts are made equal so that the survivors of one space can certainly be completely contained in the other space and the above highly unlikely situation can still be accommodated.

However, as suggested in [6] and [7], the amount of live objects to be copied can be significantly smaller than the reserved space. The live object size of a typical program would normally not exceed a certain maximum value and it will waste space if we reserve more space than this value. Although in [24] it is said that the live object ratio (i.e., live objects to all objects) is a variable over the course of program execution, our finding suggests that the live object size after every collection tends to remain relatively stable. By stable, we mean the difference in live object size between two successive collections is quite small. We refer to a particular period during which the live object size remains stable as a memory usage phase. After some time being in a certain memory usage phase, the program may move into another phase and stay there for a sufficiently long period of time.

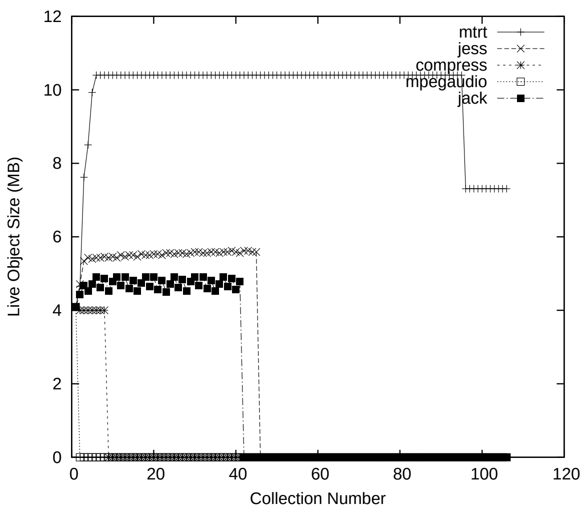
To demonstrate the above more concretely, we carried out an experiment to measure the live object size of the applications in the SPECjvm98 Benchmark using the semispace collector in MMTk [8]. The results are presented in Fig. 1[3].

It can be easily seen that in both sub-figures all the benchmarks reached their maximum memory requirements at some point, regardless of the heap size. Furthermore, except for *javac*, all the applications display a relative stable live object size over the series of collections and a number of phases. In the 25MB heap setting, for example, *mtrt* starts in a phase of around 4MB, which then increases sharply to 10.40MB, and then becomes stable for about 90 collections; afterwards, it drops down to 7.3MB and remains there until the end of the program. Overall there are three phases. Even more pronounced are the cases of *jess*, *compress* and *jack* whose live object sizes change very little in the entire execution; they linger around 5.5MB, 4.0MB and 4.6MB respectively. These applications practically have only one phase and so our prediction approach works best for them. The same phenomena happen also in the 50MB heap setting, although here the number of collections decreases dramatically for all the programs. Note that half of the heap would be 25MB, but none of the programs have produced a surviving object size larger than 25MB. In fact, a closer examination shows that *javac*'s live object size never exceeds 12MB. Compared with the reserved 25MB in the 50MB heap case, this represents a huge waste of precious memory resources.

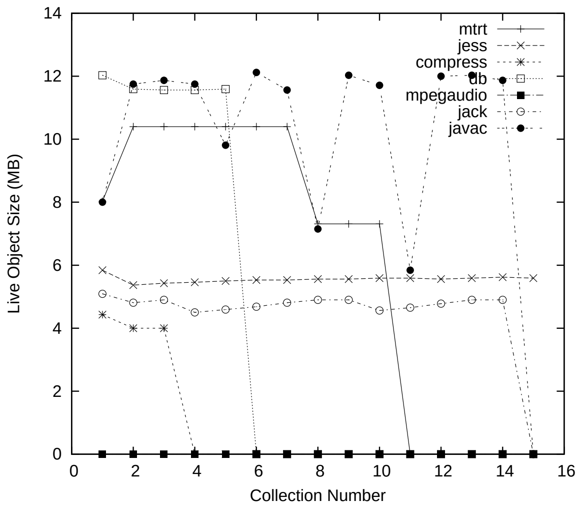
The above finding encourages us to try to predict the subsequent live object sizes, based on which we can adaptively adjust the size of the reserve space instead of always half of the heap.

The figure also reveals why the collection time of copying-based collectors decreases as the available heap space increases. Copying-based collectors take time proportional to the live objects, instead of the whole heap. The figure shows that, whatever the heap size may be, the live object size is more or less the same, and so is the time consumed by a single collection. Because of the larger heap in

[3] In the 25MB heap setting, *javac* and *db* are not runable due to insufficient memory. *mpegaudio* seldom allocates, so with a 25MB heap it only requests one collection, and with a 50MB heap the collector is not even triggered.

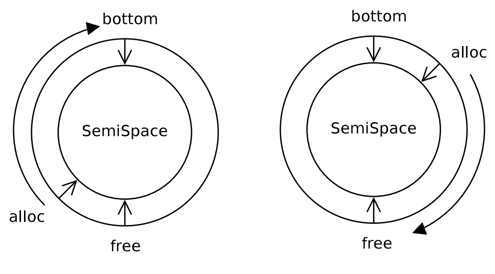


(a) 25MB heap

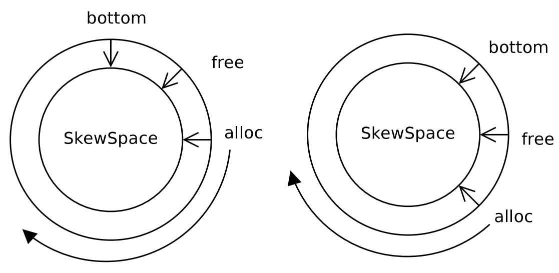


(b) 50MB heap

Figure 1: Size of Live Objects



(a) SemiSpace Collector



(b) SkewSpace Collector

Figure 2: Comparing Two Collectors

Fig. 1b, the number of collections is significantly smaller, leading to a much reduced overall collection time.

3. The Skew-Space Collector

The previous section has revealed the likely fact that the live object size can be significantly smaller than the space typically reserved in a semispace garbage collector, and this size tends to remain stable throughout the program's execution. In response to this, we propose a *skew-space garbage collector* whose mechanics is illustrated in Fig. 2.

The upper two sub-figure depicts the process of the traditional semispace collection. From pointer *bottom* to *free* is the reserved space, and from *free* to *bottom* the active space. Objects are allocated in the direction as indicated by the black solid pointer. When the *alloc* pointer coincides with the *bottom* pointer, a collection is triggered and survivors are scavenged into the space between pointer *bottom* and *free*. At the same time, the active space is set as the reserved space, and vice versa. Obviously, in order to prepare for the worst case where all the objects survive, semispace collectors must reserve half of the available heap. But unfortunately, most of the time, the reserved space is largely wasted, as survivor rate typically is far below 100%.

The lower sub-figure describes two typical collections in our skew-space collectors. Between *bottom* and *free* is the survivors' space (that occupied by the live objects) in the last collection, between *free* and *alloc* is the reserved space for copying, and between *alloc* and *bottom* is the allocation space. We can see that this time the reserved space is not equal to the half of the available space, but the same as the survivors' size of the last collection. After every collection, the survivors are copied into reserved space, and then the following code is executed:

```
survivorSize = getCurrentSurvivors();
bottom = free;
free += survivorSize;
alloc = free + survivorSize;
endGC();
```

If in a very rare situation, the total size of survivors exceeds the reserved space, we trigger a compacting collection which requires no extra space for collecting the remaining objects. The root set of the compacting collection is the set of un-scanned objects in the preceding copying collection. As we do not need to do the compacting from the very beginning, but only need to finish a remaining task left behind by the copying collection, the compacting collection will consume less time than a normal compaction.

3.1 Implementation

We partition the whole available heap space into equal regions whose size is set to be eight OS pages.[4] Each region contains a header (see Fig. 3) which consists of information about the end of the actual data in this region and the location of the next region. The *next* header word of the last region points back to the first

[4] In moving collectors, large objects with size bigger than for example two OS pages are allocated in the large object space (LOS) to avoid the time of copying such a big object. Thus objects allocated in skew-space must be able to fit in a region.

region in this heap space. The reason for this is that in the skew-space collector, after every collection the *bottom*, *free* and *alloc* pointers must be moved. Once *alloc* reaches the end of the heap, there should be a way to tell the pointer where to allocate next. We achieve this by connecting the top of the heap to its bottom, so that *alloc* can freely move in this heap space. Furthermore, to support the compacting collector, we need to know how to linearly traverse the heap in allocation order; thus the information about where the next region resides is essential. The size of every region header is about eight bytes, so the space overhead is less than 0.2%.

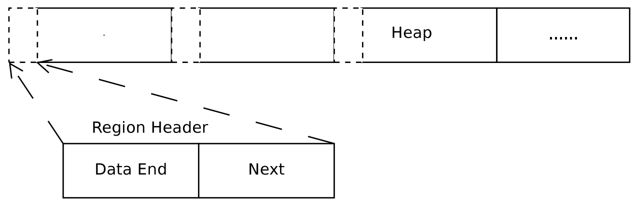


Figure 3: The Heap Structure

At the start of a program's execution, we reserve half of the heap space as there is no survivor information available. The *bottom* pointer points at the bottom of available heap, and *free* at the middle. Objects are allocated from the place designated by the *alloc* pointer, which is initially set to be the bottom of the heap. When the *alloc* pointer meets up with the *free* pointer, a garbage collection is triggered which moves the survivors to be after the *free* pointer. At the same time, the size of the survivors is calculated and recorded, based on which the three pointers are adjusted. Mis-prediction happens when the size of the survivors is too large, causing *free* to pass *alloc*. In this situation, the live object size curve shoots up as compared with the previous collection.

How to advance the *alloc* pointer is a tricky problem. Closer observations on Fig. 1 tell us that the live object size actually fluctuates, but not exactly stick around the stabilized line. If the *alloc* pointer is advanced by exactly the size of the survivors, mis-prediction will happen every time even when the live object size rises by just a little bit. To deal with this, we add two other parameters, *flucBytes* and *protectBytes*, and initialize the former to be 0.[5] Then upon every mis-prediction, the following code is executed:

```
flucBytes = PreSurvivors() - CurSurvivors();
flucBytes += protectBytes;
if(0 < flucBytes) survivorSize += flucBytes;
```

To explain the code, we generalize the variation of live object size as in Fig. 4. The fluctuation of the live object size may be upward and downward. So *flucBytes* should be added if we want to avoid false prediction on such tiny fluctuations. *protectBytes* is used to fight against the difference between the fluctuation wave tips, which is manually set to be 1% of the whole heap. In our experiment this percentage works well but it can also be set dynamically to cater to the characteristics of other programs. Note that if the survivor rate grows to be very high, augmented by *protectBytes*, the reserve ratio may exceed 100%. We added a test to check for this and stick to 100% if this should happen.

3.2 Compacting Collection

Although a false prediction is very rare, we still need a mechanism to tackle the situation. Another moving collector, mark-compact, can be utilized if *free* passes the *alloc* pointer. Our mechanism is similar to the method described in [7] for Appel-style generational collectors [25], except that the survivors are not copied into the mature generation but to the free space.

[5] In the first cycle of the collection, half of the available heap is reserved, and there is no need to reserve extra space.

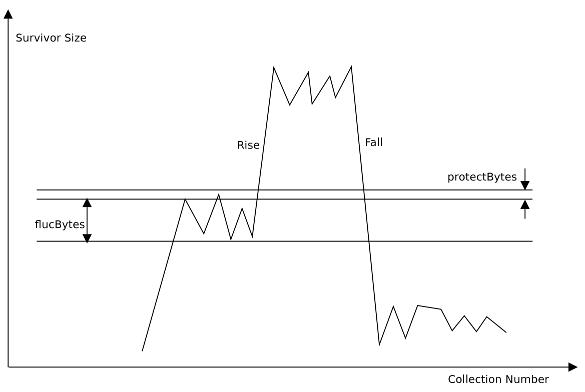


Figure 4: The Fluctuation Wave

A collection starts by computing the entire root set and pushing them into a trace queue. If an object is reached, it is first copied to the regions after the *free* pointer and also pushed into the queue for pointer scanning. An object will be popped out automatically after it is fully scanned. In this way, live objects are continuously pushed onto and popped out of the queue. The tracing is finished if the queue becomes empty.

For every object copied, its size is fetched and added to the *free* pointer. If the result is larger than the *alloc* pointer, while the tracing queue is not empty, we have a false prediction. At this juncture, the remaining objects in the allocation space cannot be copied into the free space; otherwise they may overwrite those live objects previously created in the allocation space. In this event, the remaining objects in the tracing queue can be used as the root set of tracing to prepare for a mark-compact collection. We start from these objects and mark those living objects in the allocation space. The remaining live objects are then compacted into the regions after the current *free* pointer.

It is worth pointing out that there is a difference between fall-back compacting and normal compacting. For our compacting collector, some of the objects in the operation area may already have been copied to the reserved space. When the process of marking comes across such an object, we simply update the referring pointers with the value stored in the forwarding word (it is installed by copying-based collectors). This object will not be marked, because it is already copied into the working space and fully scanned. The costs of compacting consist of that due to moving objects and that due to traversing the entire heap. Even if there is a false prediction, the difference between the pre-reserved space and the actual live object size is subtle. So we expect the time spent on the former to be not so much, as it is moving much fewer objects. The biggest problem is that, for a compacting collection, the entire heap will need to be traversed, including the garbage objects. Because of the LRU eviction mechanism used by virtual memory systems, we can expect that pages with a lot of garbage normally reside out of the main memory. The traversal of them, therefore, will result in a lot of page thrashing and extensively prolong the collection time.

3.3 Parallelism

By its very nature, our skew-space collector can be implemented as a parallel procedure. Since the memory resources are acquired by regions, threads can be synchronized at this level. At first all the regions are in a global memory pool. Every thread start its execution by requesting a region from the pool, and once a region is successfully granted, allocation of objects can be happily performed there. If the current region is exhausted, this particular thread can again request another region or perform a local garbage collection. In this way, the coordination happens for regions as units, which are large enough to require only very little coordination effort.

The possibility of parallelism also adds to the reason why we partition the heap space into regions. In our skew space implementation, we resort to mark-and-compact garbage collection when the prediction fails. Due to the nature of the latter type of collectors, a linear scan through the heap is needed. But in a multi-threading environment, where the virtual address space is shared by all the threads, to define the address of the next object, an auxiliary data structure is used. Without the help of the region header, linear scanning through the memory resources consumed by a particular thread would not be possible.

4. Performance Evaluation

4.1 Experimental Setup

We use all the applications in SPECjvm98 except *mpegaudio*, because it seldom allocates and thus invokes extremely few collections. The benchmarks are run on a Dell desktop computer which has a 2 GHz Intel Core 2 Duo CPU and 2G main memory. Each core has an 8-way associative L1 data and code cache of 32KB, which is partitioned into lines of 64 bytes. There is only one unified 16-way associative L2 cache of size 4096KB, which consists of 64-byte lines.

Our skew-space collector is implemented using MMTk [8] in the JikesRVM [11]. JikesRVM is an open source virtual machine derived from IBM Jalapeno [12], with a design goal to balance between high performance and portability. It does not implement a bytecode interpreter, and adopts the Adaptive Optimization System [13] to support cutting-edge virtual machine technology and enable online feedback-directed optimizations. MMTk is the memory management toolkit for JikesRVM. Communication between them is conducted on the VM and exported interfaces. MMTk divides the available space into several areas, such as metadata, immortal, large object space, and other collector-specific spaces. The main class that implements a collector is called *plan*. In the basic *plan* class, the collector-independent spaces are created and the interface from JikesRVM is built. A *stopTheWorld* class extends *plan* by incorporating typical execution phases, which is furthermore extended by a specific collector, for example a copying collector, to create two semispaces, and to add other phases and make auxiliary calculations. From time to time, the *poll* method in *plan* is called. If either the heap or the space in use is full, a collection will be triggered.

We revise the semispace collector to implement our version of skew-space collections. The space reservation ratio is at first set to be 100%, because at that time no statistics are available for use in the prediction. After the program has started, the memory usage information is gathered and used to calculate the next ratio. In order to know whether our prediction is correct or not, we maintain a counter to keep track of the number of pages currently used by the mutators, and stop the copying if it exceeds the current reserved space. The objects lying between the *scan* and *free* pointers are then incorporated in the tracing root, from where the collector starts tracing the heap and marks touched objects to be compacted. Note that because of the specific design of MMTk we do not implement a cyclic heap (as described in Sec. 3), but check the number of pages instead to control how much virtual address space is to be used.

4.2 Results

We evaluate our skew-space collector (*sks*) against a semispace (*ss*) and a mark-compact collector (*mc*). *sks* achieves dynamically a balance between these two other collectors. *sks* reserves less than half of the heap space according to online predictions, so its performance is highly dependent on the accuracy of the dynamic predictions. If the predictions are sufficiently accurate, *sks* reserves less space than *ss*, and triggers fewer collections which ultimately leads to reduced execution time. In the event of a false (inaccurate) prediction, *sks* needs to call on *mc* to treat the remaining objects.

Compacting collectors (*mc* for instance) need to traverse the entire heap to compact live objects in the lower end of the heap. Since they must touch all the garbage, plenty of page thrashing can be expected, thus consuming much more time than copying collectors.

Fig. 5 shows the performance of the three collectors for the six benchmark applications and different heap sizes. In Fig. 5 we can see that most of the applications demonstrate a promising reduction in execution time. Tab. 1 shows the mis-prediction percentages for different applications upon various heap sizes. The majority of the false predictions are well below 15%, and some have 0%. Tab. 2 gives the "difference" in number of collections between *ss* and *sks*. It clearly shows that as the heap grows in size, this difference decreases accordingly, because the memory resource becomes not so tight. Note that "x times" in the first row of each table denotes the heap size divided by the minimum memory requirement.

In the figure, we can also see that the performance of *compress* is slightly degraded as compared to *ss*. We have analyzed the runtime characteristics of this application, and found that it seldom allocates new objects in the semispaces (most of its newly created objects are very large, therefore allocated in the LOS space). Apart from the first collection, the survivor rate in the semispace is always very close to 100%. This causes *sks* to reserve a space that is of the same size as that of *ss*, and reduces our implementation to be similar to the latter. Tab. 2 also helps to confirm our analysis, where there is no difference in number of collections for this benchmark. The slightly poorer performance of *sks* is due to the implementation of *sks* which introduces additional overheads.

We further tested our collector using a newer benchmark suite, DaCapo [9] (v. 2006-10-MR2). The results are presented in Fig. 6. We excluded *eclipse* and *chart* as they failed to work correctly for all the three collectors, due to *nullPointer* errors. Furthermore, among the other benchmarks, *hsqldb*, *lusearch* and *xalan* are multi-threaded applications, and so are also excluded from our evaluation. Note also that *mc* cannot run the *jython* benchmark under any heap size; therefore, in the corresponding diagram, only the performance of *ss* and *sks* are compared. From the figure, it is evident that *sks* outperforms *mc* and *ss* for almost all the benchmark applications.

4.3 Discussions

Copying and compacting garbage collectors represent two extremes: the former sacrifices all the space, while the latter all the time. There are some variants which take the middle way, for example [19]. It partitions the space into N multiple windows and reserves only one window for copying. If the survivor rate turns out to be less than $\frac{1}{N}$, then only one pass over the heap is needed. But if the rate grows to become larger, it displaces objects in different windows and triggers multiple passes on the heap. We have suggested possible performance improvements by conserving the fraction of the windows dynamically based on the occasion.

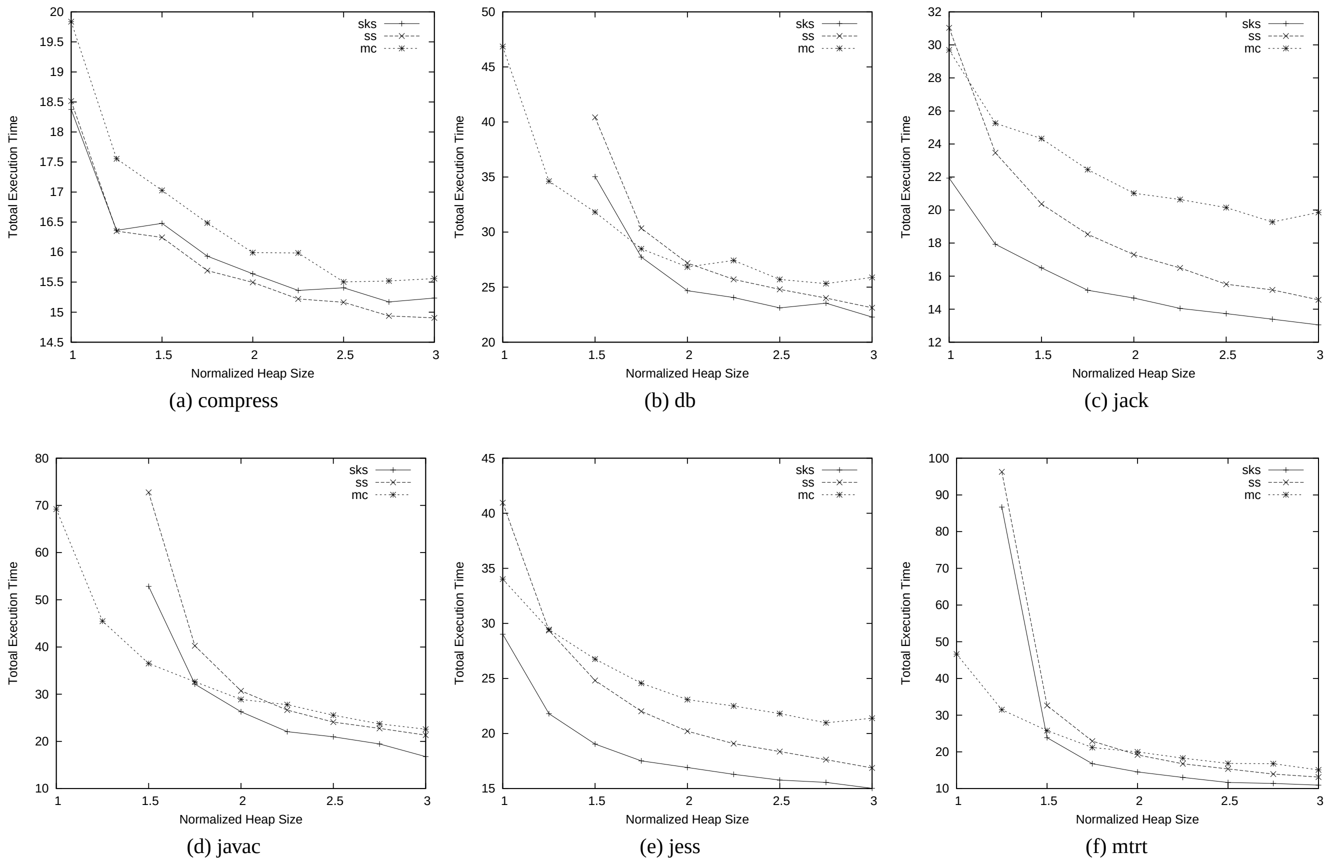
Our skew space garbage collector is not a new type of garbage collector, but a technique to combine the merits of both copying and compacting collectors. It is built on the observation that the size of live data objects will not vary too much at each collection. This phenomenon is entirely experimental and has only been tested on Java benchmarks, and so it may not be universally true for all possible programs. In that case, our collector can switch back to a normal semi-space collector, based on the false prediction rate.

Table 1: The Percentages of False Predictions by *sks*

benchmark	1 times	1.25 times	1.5 times	1.75 times	2 times	2.25 times	2.5 times	2.75 times	3 times
compress	0	0	0	0	0	0	0	0	0
db	null	null	10%	12%	0	0	0	0	0
jack	0	0	0	0	0	0	0	0	0
javac	null	null	25%	50%	42%	11%	25%	14%	0
jess	0	0	0	0	0	0	0	0	0
mtrt	null	5%	13%	9%	12%	0	0	0	0

Table 2: Number of Collections of *ss* minus that of *sks*

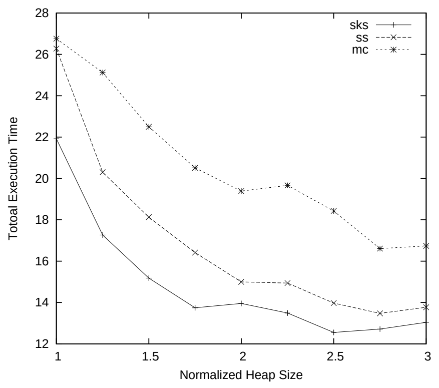
benchmark	1 times	1.25 times	1.5 times	1.75 times	2 times	2.25 times	2.5 times	2.75 times	3 times
compress	0	0	0	0	0	0	0	0	0
db	null	null	8	4	3	2	2	1	1
jack	29	19	13	11	9	8	6	6	5
javac	null	null	30	16	10	8	6	5	5
jess	32	20	15	12	9	8	7	6	5
mtrt	null	17	14	9	7	5	5	3	3



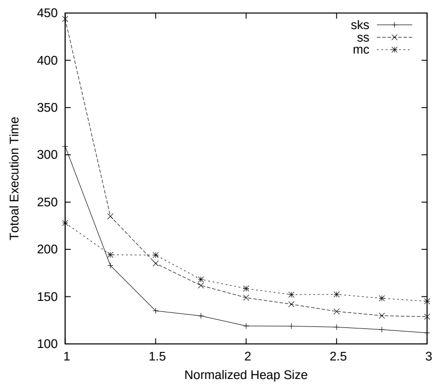
(a) compress (b) db (c) jack

(d) javac (e) jess (f) mtrt

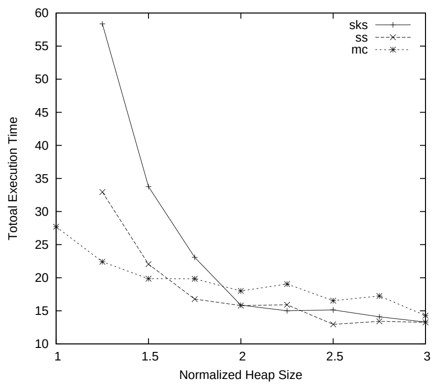
Figure 5: Comparisons using SPECjvm98 Benchmarks



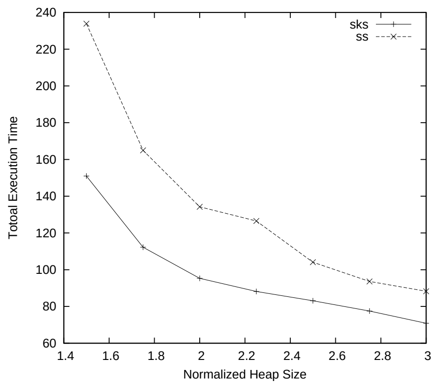
(a) antlr



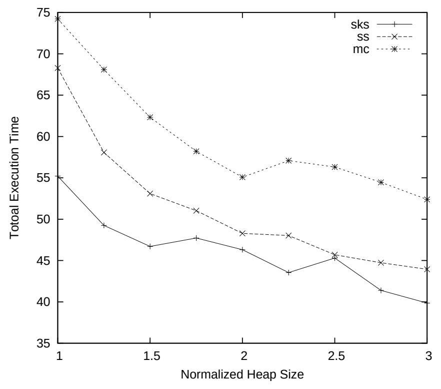
(b) bloat



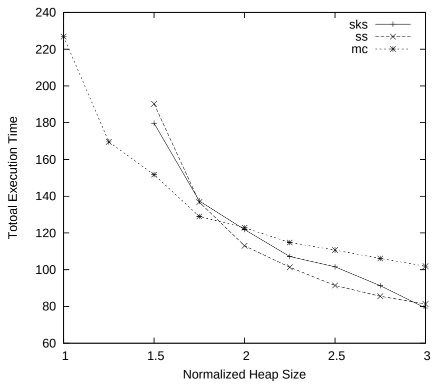
(c) fop



(d) jython



(e) luindex



(f) pmd

Figure 6: Comparisons using Dacapo Benchmarks

We argue, however, that owing to the statements proposed in Sec. 1 Program behavivors should obey this rule.

5. Generational Collectors

Our algorithm is very simple and therefore can be easily combined with other techniques (some are mentioned in Sec. 6) to achieve better performance. For generational collectors, the algorithm can be applied in two areas: to dynamically adjust the portion of reserved space and to determine when to trigger a full heap collection, which will be discussed in the following subsections.

5.1 Classifications and Comparisons

According to the size of the nursery, [19] generational collectors can be classified into three groups: Fixed Generation (FG, which maintains an unchanged size of the nursery), Variable Generation (VG, which allows the nursery to grow and shrink progressively) and Bounded Generation (BG, which sets a minimum size threshold).

FG collectors, such as [23], only promote objects when they are old enough; this requires additional age information inside the object headers. Some of the implementations remove the age information by partitioning the nursery into buckets, but this incurs extra copying costs. Furthermore, FG normally collects the heap when the memory usage exceeds $\frac{HeapSize}{2} - NurserySize$, whereas VG [25] triggers a collection when the heap is half full and therefore is more space efficient. The main problem with VG, however, is that when the mature space grows too big, it squeezes the nursery into a small space and causes collection to take place too often. To tackle these problems, BG imposes a minimum size to which the nursery can be decreased. It starts a garbage collection immediately after the heap becomes half full or the minimum threshold is reached. Because of its merits, we picked BG for applying our algorithm. Our algorithm can also be built on other collectors to dynamically conserve copying space.

5.2 Skew Space Generational Collector

As illustrated in Fig. 7a, a traditional Appel style generational collector starts by employing half of the available heap as its nursery, where newly created objects are continuously allocated. A garbage collection will be triggered if the nursery becomes full, and then the live objects are pushed to the very end of the reserved space which forms the mature space. This process repeats until the whole available heap is half occupied, at which time a full heap collection will be conducted to tidy up the spaces.

For the traditional Appel's collector, there are three issues that need to be addressed:

- It always reserves half of the space in case the worst scenario happens, that is, when all the objects survive. But this is very rare or indicates inefficiently assigned memory resources if it does happen too often.
- Although it seems simple, but how to determine when the heap will become half occupied remains to be a tricky problem.

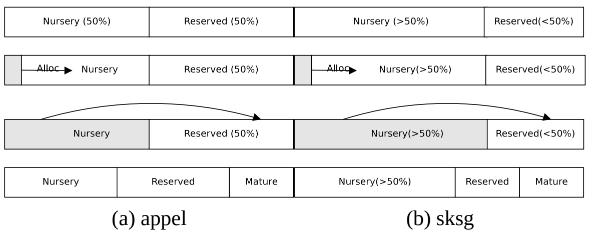


Figure 7: Appel's and Modified Generational Collectors

Appel's collector solves it through an indirect way. We are going to present an alternative.

- As the mature generation continues to grow, it will squeeze the nursery into a small space and result in frequent collections. Therefore, we need to put a limit on how small the nursery can be reduced to.

It can be expected that, for generational collectors, the nursery survival rate is even lower than that of the whole heap. Experimental results show that normally this rate is well below 10% and sometimes drops to very close to zero, which also explains why the static algorithm in [7] works well. In view of a high infant mortality, traditional Appel's collectors waste most of the reserved space. We therefore propose to reserve space according to the memory usage pattern (Fig. 7b) and reserve less than 50% of space for copying. The basic rule is the same as that applied to non-generational collector, and so if the prediction fails, a compacting collector is called upon for both minor and major collections.

We maintain two rates of survivals: $MinorRate$ for minor and $MajorRate$ for major collections. As usual, $MinorRate$ is the result of a division between nursery survivals and original nursery size upon a minor collection, while $MajorRate$ is a division upon a major collection. Once the nursery is exhausted, it will first multiply $MinorRate$ with the size of the nursery ($NurSize$) to produce the prospective size of minor survivors (denoted as $ProsMiSize$); it then calculates the predicted major survivors' size (denoted as $ProsMaSize$) in the same way. At last the following formula is evaluated:

$$ProsMiSize + ProsMaSize + MatureSpace \geq FullHeap$$

If it is satisfied a major collection must be triggered; otherwise only a minor collection. Compared to the method adopted by Appel-style generators to determine the time for a major collection, our algorithm does not re-locate part of the live objects and hence avoids the extra pointer-updating overhead.

In order to tackle the third issue, a target policy is needed. Here we set our aim at keeping a smooth collection pace, as collections being too frequent would leave little time for the objects to die. That drives us to adopt a bounded-size generational collector. For this collector, a special parameter, threshold nursery size, together with the previous strategy, would define when to trigger a major collection. The collector will degenerate to non-generational if the nursery size is set to be half of the heap space. The smaller this parameter becomes, the more collections will be triggered. We propose to use a feedback mechanism to dynamically adjust this parameter, in order to keep to the same number of allocations between any collections. We use the percentage of the entire available heap to control the threshold nursery size and manually set it to 5% at the very beginning. Then the program keeps running and triggers minor collections; meanwhile a count ($AllocMean$) is maintained to store the number of allocations which is averaged over the whole period. This process continues until it consumes more than $MajorRate$ amount of space or the nursery size drops to this threshold. At this time a major collection is triggered to reclaim any garbage. At the same time, we compare the allocation mean to that of the previous major collection. If it is larger then the threshold percentage, the latter is increased by 2.5%, and vice versa. But this percentage is not allowed to rise higher than 25%; otherwise the expanded nursery would be too large. With all this we can make sure that the size of the nursery will never become too small and the pace of collections will never be too fast.

Again we implemented our collector with MMTk then compared it to a generational copying garbage collector. The result is available in Fig. 8. It is expected that the improvement is not as great as that of its full heap counterpact. This is because right now most of the time only the nursery is collected. Its size is small and it leaves little space for optimizations.

6. Related Work

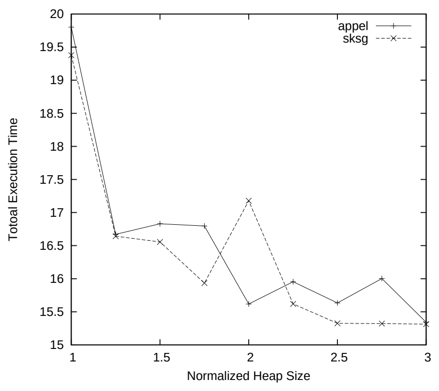
After McCarthy's proposal to use an automatic memory management strategy for Lisp, garbage collectors were widely investigated, with two main implementation approaches: reference counting and tracing. These two approaches are then unified in [14] based on the argument that they are in fact duals of each other. [15] compares the performance of two different tracing techniques: mark-and-sweep and stop-and-copy, and points out that while the former is slightly more expensive than the latter, it consumes significantly less memory. In the following we discuss some works on collectors that move objects, which are more related to our paper.

6.1 Moving Collectors

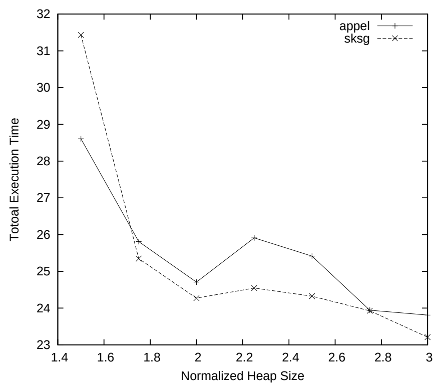
There are basically two different types of collectors that move live objects: semispace collectors copy their survivors [16], and compacting collectors compact them [17]. The advantage of moving collectors is that they provide very cheap allocation. In mark-sweep collectors, allocation requests are satisfied by searching free-lists, whereas for moving collectors, the equivalent is just to advance the bump pointer by the requested size. But there are prices to pay. Compacting collectors need to touch the whole space several times, and semi-space collectors need to reserved half of the available space for copying the survivors.

6.2 Space Efficient Collectors

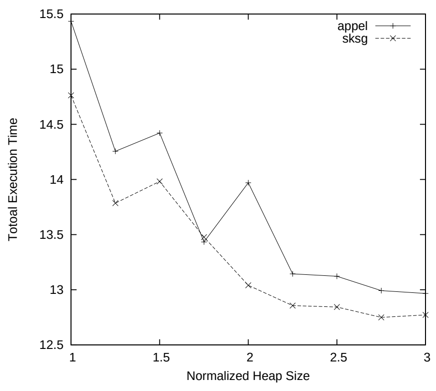
Because copying-based collectors need to reserve half of available heap whereas compacting-based versions do not, for a particular program there must be a point in the time-space tradeoff where two kinds of collectors achieve the same performance. [21] analyzed theoretically how memory residency (the ratio between live memory and the total heap) affects the performance of both collectors, and adopted a dual-mode collector which switches between copying and compacting according to the memory usage. In order to reduce the space reserved in a full-heap copying collector, [18] partitions the heap into a mark-sweep space and a sliding copying space. The copying space is relatively small and slides linearly over the entire heap in order to tidy up scattered objects in the mark-sweep space. Their algorithm works well and its copying space can make use of our algorithm to further reduce the reserved amount. The mark-copy collector proposed in [19] divides the mature space of a generational collector into multiple windows and maintains uni-directional remember sets from higher windows to lower ones. It defers full collections until the free space drops down to one window which means that only one frame is reserved. [10] resorts to region-based memory management strategy which mixes marking and copying in one pass to provide space efficiency and to achieve faster reclamation and better mutator's performance. The above two



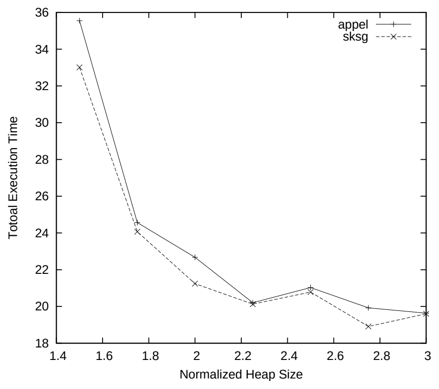
(a) compress



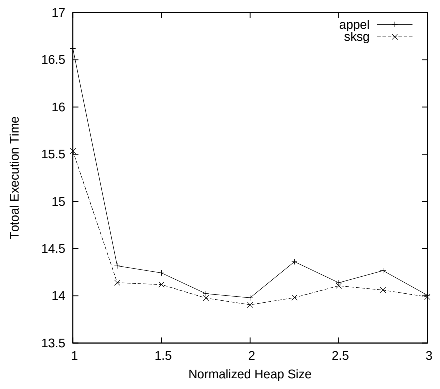
(b) db



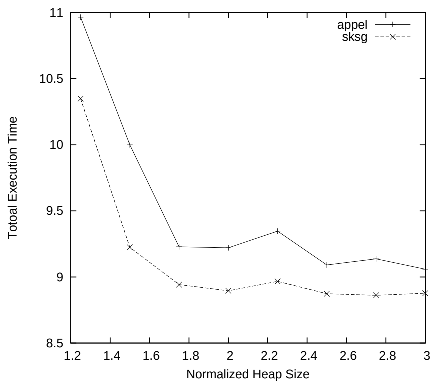
(c) jack



(d) javac



(e) jess



(f) mtrt

Figure 8: Comparisons between Appel's and Our Collectors

cutting-edge collectors target at the mature space while ours at the young space, and so combining either of them with our implementation will be an interesting investigation.

In recognition of the fact that the live object size must be smaller than the reserved space, Sun Microsystem's HotSpot virtual Machine [29] implements a parallel copying generational collector with a fixed nursery size, where survivors are always promoted to the mature generation after a collection. It would not always reserve enough space for copying all the live objects, and when there is not enough memory, it compacts the nursery. [6] and [7] implement similar strategies for Appel-style generational collectors. The three collectors above are very similar to our skew-space collector, but our collector has the following merits:

- We take advantage of the phenomenon we observed about live object size to dynamically set the reserved space, which is more accurate and precise. [29] does not unveil any implementation details, whereas [7] reserves space manually, and therefore cannot adapt to the specific characteristics of different applications. In particular, [6] reserves space by calculating the average size of live objects, which cannot be all that accurate because a single unusually high or low survivor rate would pollute all the predictions.
- Our algorithm is very general and can be extended to both Ungar's and Appel's style of generational collectors.
- We have discussed and can devise different strategies to apply our algorithm to different generational collectors.

There are also plenty of work to reduce the space consumed by non-copying garbage collectors, for example, [20], which introduces objects reuse in the design of garbage collection, and [27] which eliminates forwarding pointers and shortens object headers, etc.

7. Conclusion

In this paper, we conjecture a memory usage phenomenon of typical applications, which is that live objects' total size would reach a maximum value and tends to remain relatively stable over many collections. We have carried out experiments using the benchmarks in SPECjvm98 to show that it is indeed true. In response, we design and implement a skew-space garbage collector that can be used in both embedded and general machines. Our skew-space collector improves copying-based collection by dynamically adjusting the reserved space to achieve space and consequently time efficiency. It capitalizes on the above phenomenon to predict how much space should be reserved, and resorts to a compacting collector in the rare event when the prediction turns out to be false. Compared with traditional semi-space collectors, our skew-space collector improves the total execution time of the tested benchmarks. Because of the simplicity of the design, our algorithm can also be applied to the young space of a generational collector, or work in a concurrent fashion to perform real-time and space efficient collections. It will be particularly interesting if both existing static approaches [28] and our dynamic memory usage prediction algorithm can be combined to achieve potentially greater improvements.

Although in our experiments most of the Java benchmarks exhibit the above phenomenon, the live object size of some real-life programs may vary and deviate from this phenomenon considerably. In the future, we plan to devise a strategy to record and analyze the variations, and then based on the false prediction rate to determine if *protectBytes* should be adjusted to reserve more space, or, in the worst case, to fall back on a semi space collector. Furthermore, the criterion for when to trigger compacting collection can be refined—for example, to compact only when the *alloc* pointer runs into a live object. Although not tested, we believe that other programs, for example C# and Smalltalk programs, may also bear the same characteristics so that our algorithm can be applied to them. Lastly, our implementation of the skew-space collector will always resort to a compacting collector to recycle the remaining live objects. Compacting is very expensive and how to reduce its cost will be a meaningful pursuit.

Acknowledgments

This work is supported in part by a Hong Kong RGC GRF grant (7141/06E).

References

[1] J. McCarthy: Recursive Functions Symboloc Expressions and Their Computation by Machine. In: Communication of the ACM, Volume 3, Number 4, 184-195, 1960.

[2] R. Jones and R. Lins: Garbage Collection: Algorithm for Automatic Dynamic Memory Management. John Wiley&Sons, 1997.

[3] P. R. Wilson: Uniprocessor Garbage Collection Techniques. In: Proceedings of the International Workshop on Memory Management, 1-42, 1992.

[4] B. K. Haddon and W. M. Waite: A Compaction Procedure for Variable Length Storage Element. In: The Computer Journal, Volume 10, Number 2, 162-165, 1967.

[5] H. Lieberman and C. Hweitt: A Real-time Garbage Collection Based on the Lifetimes of Objects. In: Communication of the ACM, Volume 26, Number 6, 419-429, 1983.

[6] J. M. Velasco, K.Olcoz and F. Tirado: Adaptive Tuning of Reserved Space in an Appel Collector. In: European Conference on Objected-oriented Programming, 542-558, 2004.

[7] P. MaGachey and A. L. Hosking: Reducing Generational Copy Reserve Overhead with Fallback Compaction. In: International Symposium on Memory Management, 17-28, 2006.

[8] S. M. Blackburn, P. Cheng and K. S. McKinley: Oil and Water? High Performance Garbage Collection in Java with MMTk. In: International Conference on Software Engineering, 137-146, 2004.

[9] S. M. Blackburn, R. Garner and C. Hoffman: The Dacapo benchmarks: Java benchmarking development and analysis. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 169-190, 2006.

[10] S. M. Blackburn and K. S. McKinley: Immix: a Mark-Region Garbage Collector With Space Efficiency, Fast Collection, and Mutator Performance. In: ACM Conference on Programming Language Design and Implementation, 22-32, 2008.

[11] B. Alpern, S. Augart and 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.

[12] B. Alpern, C. R. Attanasio and J. J. Barton: The Jalapeno Virtual Machine. IBM Systems Journal, volume 39, number 1, 211-238, 2000.

[13] M. Arnold, S. J. Fink and D. Grove: Adaptive Optimization in the Jalapeno JVM. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 47-65, 2000.

[14] D. F. Bacon, P. Cheng and V. T. Rajan: A Unified Theory of Garbage Collection. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 50-68, 2004.

[15] B. G. Zorn: Comparing Mark-and-Sweep and Stop-and-Copy Garbage Collection. In: ACM Conference on LISP and Functional Programming, 87-98, 1990.

[16] C. J. Cheney: A Nonrecursive List Compacting Algorithm. In: Communication of the ACM, Volume 13, Number 11, 677-678, 1970.

[17] H. B. M. Jonkers: A Fast Garbage Compaction Algorithm. In: Information Processing Letters, Volume 9, Number 9, 25-30, 1979.

[18] B. Lang and F. Dupont: Incremental Incrementally Compacting Garbage Collection. In: ACM SIGPLAN Notices, Volume 22, Issue 7, 253-263, 1987.

[19] N. Sachindran and J. E. B. Moss: Mark-copy: fast copying GC with less space overhead. In: ACM SIGPLAN Notices, Volume 38 , Issue 11, 326-343 , 2003.

[20] Z. C. H. Yu, F. C. M. Lau and C. L. Wang: Object Co-location and Memory Reuse for Java Programs. In: ACM Transactions on Architecture and Code Optimization, Volume 4, Issue 4, 232-268, 2008.

[21] P. M. Sansom: Combining Single-Space and Two-Space Compacting Garbage Collectors. In: Proceedings of the Glasgow Workshop on Functional Programming, 1991.

[22] E. W. M. Dijkstra, L. Lamport and A. J. Martin: On-the-fly Garbage Collection: An Exercise in Cooperation. In: Communication of the ACM, Volume 21, Number 11, 611-612, 1978.

[23] D. Ungar: Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. In: ACM SIGSOFT Software Engineering Notes, Volume 9, Issue 3, 157-167, 1984.

[24] D. Ungar and F. Jackson: Tenuring policies for generation-based storage reclamation. In: ACM SIGPLAN Notices, Volume 23, Issue 11, 1-17, 1988.

[25] A. W. Appel: Simple generational garbage collection and fast allocation. In: Software Practice & Experience, Volume 19, Issue 2, 171-183, 1989.

[26] D. A. Barrett and B. G. Zorn: Garbage collection using a dynamic threatening boundary. In: ACM SIGPLAN Notices, Volume 30, Issue 6, 301-314, 1995.

[27] D. F. Bacon, P. Cheng and D. Grove: Garbage collection for embedded systems. In: International Conference On Embedded Software, 125-136, 2004.

[28] E. Albert, S. Genaim, M. G. Zamalloa: Live heap space analysis for languages with garbage collection. In: International symposium on Memory management, 129-138, 2009.

[29] The Java Hotspot Virtual Machine, White Paper http://java.sun.com/products/hotspot/index.html.

[30] The SPEC Java Virtual Machine Benchmarks http://spec.org/jvm98.