Science of Computer Programming 78 (2013) 445–457

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Skew-space garbage collection

Liangliang Tong *, Francis C.M. Lau

Department of Computer Science, The University of Hong Kong, Pokfulam Road, Hong Kong, China

ARTICLE INFO

Article history:
Available online 23 July 2011

Keywords:
Skew space
Mark compact
Semi space
Garbage collection
Space efficiency

ABSTRACT

Semispace garbage collectors relocate all the live objects in one step, which is simple and leads to good performance. Compared with mark-compact collectors, however, they need to reserve extra heap space for copying live objects. As much as half of the heap could be reserved as it is possible that all the allocated objects survive. In reality, however, most programs exhibit a high infant mortality, and therefore reserving half the heap is wasteful.

We have observed that the memory usage of many ordinary programs is relatively stable over the course of their execution. This provides an opportunity for online predictions to dynamically adjust and optimize the reserved space. Consequently, we propose a skew-space garbage collector that reserves space dynamically. This collector is implemented using the MMTk framework of the Jikes RVM and gives encouraging results against related garbage collection algorithms for the DaCapo and SPECjvm98 benchmarks.

1. Introduction

Automatic dynamic memory management, also referred to as garbage collection (GC), was devised by 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 to programming language design [2]. GC has since been adopted in the design of many object-oriented languages such as Java.

There are in general two approaches for constructing garbage collectors: reference counting and tracing [4]. Tracing-based collectors are usually preferred because, unlike reference counting, they do not need extra space to store count numbers and can trivially handle cyclic objects. This paper is about the kind of tracing-based collectors that would move objects around when performing GC, which includes compacting and copying collectors. With compacting collectors, the heap is traced to divide the live and the dead objects; the entire heap is then traversed several times so that the live objects are compacted into the lower end of the heap [5,19]. A typical copying collector [18] on the other hand would divide the heap into two equal spaces: the working space and the free space. The 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 which will become the new working space in the next round. The spaces are absolutely clean after every collection by these tracing-based collectors. These 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. The fragments are eliminated since all the free memory blocks are compacted during the process.
- The objects can be relocated to achieve placements that are cache-conscious.

* Corresponding author. Tel.: +852 28578272.
E-mail addresses: lltong@cs.hku.hk (L. Tong), fcmlau@cs.hku.hk (F.C.M. Lau).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.06.003

The time needed to perform a collection is an important performance measure. Copying collectors complete the collection in one step. Only live objects are touched during this step, and so the operation time depends only on the amount of live objects. A compact collector needs four steps to complete the job: tracing, calculating, updating and compacting. Unfortunately, both the calculating and the compacting steps need to walk over all the objects, and thus the collection time depends on the amount of all objects. Therefore, copying-based collectors are more widely employed in practice, particularly for handling the nursery space of generational garbage collectors [6,24]. However, there is a price for its efficiency: as much as half of the memory space needs to be reserved for the upcoming garbage collection at all times. Copying collectors are also called semispace collectors.

The semispace collector trades simplicity and efficiency for space. This is acceptable if indeed the reserved space is absolutely necessary. But will there always be so many surviving objects to fill the reserved space? Obviously, at the time of GC, the amount of survivors must not be excessive as the collector is expected to finish its work swiftly and there should be sufficient space left for the new objects. Otherwise, collection may end up being triggered too frequently, which is unacceptable. But indeed if only a reasonably small fraction of the objects in the working space would survive, then reserving half of the heap is too much. Such wastage of memory resources indirectly adds to the frequency of GC. But can we safely reserve less than half of the space? This paper gives an affirmative answer which rests 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 program, which is a variable over the course of execution. 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 the previous total size of live objects of the previous GC. We have observed that this is true for most programs. Our conjecture is built on the following reasons.

- If the memory requirement of a program keeps rising without a bound, then the program will eventually exhaust all physical memory resources and end with a runtime exception. The maximum memory requirement should be considerably less than the reserved space in order that the garbage collection will not be triggered too frequently.
- Many programs spend most of their time executing the same code in a loop repeatedly.
- Many objects have a relatively short life time. If collections are not too frequent, allocations and de-allocations may just balance out in terms of space.
- Occasionally, massive allocations may happen. The objects involved, however, will likely be de-allocated soon, or otherwise memory resources will run out quickly.

A similar claim that live object sizes change only modestly over time is vaguely given in [27] with no support via experimentation. If ordinary programs indeed have the above-mentioned behaviors, we can then devise an algorithm to predict their memory usages and reserve the appropriate amounts of memory accordingly. In order to empirically test these conjectures, we carried out an experiment using the applications in SPECjvm98 [31]. 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 encouraged us to deviate from the conventional half-sizing approach. In this paper, we propose a dynamic free space reservation algorithm whereby the size of the reserved space is set according to the total size of the previous survivors. As the survivor rate (i.e., percentage of objects surviving) tends to be low typically, the reserved space can be much smaller than that by the conventional approach, which leaves more space for object allocations and consequently reduces the number of GCs. In some rare cases where the survivors' size exceeds the size of the reserved space, a compacting collector is triggered to collect the remaining objects. This fallback strategy is similar to the one adopted in [9], except that we customize it to fit a non-generational collector.

The contributions of our work are as follows.

- We analyzed the memory usages of many Java programs; the results showed that the live object size tends to reach a maximum value after some time and remains stable after each garbage collection.
- In light of the above phenomenon, we propose a "skew-space" garbage collection algorithm to reduce the reserved space of copying-based collectors; we show its compatibility with other improvement techniques, and analyze its applicability to bounded-size generational collectors.
- We implemented our skew space collector in the Jikes RVM [13] and evaluated its pros and cons.

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

2. Live objects and their sizes

In object-oriented programs, such as Java and C# programs, many objects are created which later become unreachable garbage inside the heap. To conserve the limited memory resources, garbage collectors try to reclaim spaces occupied by

Table 1
Descriptions of the jvm98 and DaCapo benchmarks.

Benchmark	Description
compress	Modified Lempel–Ziv (LZW) method to compress data
db	Performs multiple database functions on memory resident database
jack	A Java parser generator that is based on PCCTS
javac	The commercial Java compiler from JDK 1.0.2
jess	A Java expert shell system based on NASA's CLIPS expert shell system
mtrt	A ray tracer that works on a scene depicting a dinosaur
antlr	Parses multiple grammar files, generates a parser and lexical analyzer
bloat	Performs a number of optimizations and analysis on Java bytecode files
fop	Takes an XSL-FO file, parses it and formats it, generating a PDF file
jython	Interprets the pybench Python benchmark
luindex	Uses lucene to index a set of documents
pmd	Analyzes a set of Java classes for a range of source code problems

these unreachable objects. It is very rare that all objects in a program would live forever and occupy the address space persistently; such a case would likely end with a memory exhaustion exception as new objects continue to be created.

Copying-based collectors divide the available heap into two parts: FromSpace and ToSpace. These two parts are purposely made equal in size so that the survivors of one space can be completely contained in the other space, and the rare "all objects surviving" situation can be accommodated.

However, as suggested in [7,9], the amount of live objects to be copied can turn out to 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 be wasteful if we reserve more space than this value. Although [25] stated 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 tends to remain relatively stable over the series of collections. Here "stable" means the difference in live object size between two successive collections is small. We refer to a certain period during which the live object size remains stable as a memory usage phase. After being in a certain memory usage phase for some time, the program may move to another phase and remain there for a period of time.

To demonstrate the above in a concrete way, we carried out an experiment to measure the live object sizes of the applications in the SPECjvm98 and the DaCapo benchmark suites using the semispace collector in the MMTk toolkit [10]. Short descriptions of these benchmark applications are given in Table 1. The results are presented in Fig. 1.[1]

It can be easily seen from both sub-figures that 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 relatively stable live object size over a series of collections and a number of phases. In the 25 MB heap setting, for example, *mtrt* starts in a phase of around 4 MB, which then increases sharply to 10.40 MB, and then is stable for about 90 collections; afterward, it drops down to 7.3 MB 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 at around 5.5 MB, 4.0 MB and 4.6 MB respectively. These applications practically have only one phase and so our prediction approach works best for them. The same phenomena happen also in the 50 MB heap setting, although here the number of collections decreases dramatically for all the programs. Note that half of the heap is 25 MB, but none of the programs have produced a surviving object size that is close to 25 MB, let alone exceeding it. In fact, a closer examination shows that *javac*'s live object size never exceeds 12 MB. So if indeed 25 MB are reserved as is done in the conventional approach, much precious memory resources would be wasted. Our observation also matches the generational hypothesis, which states that most objects die young.

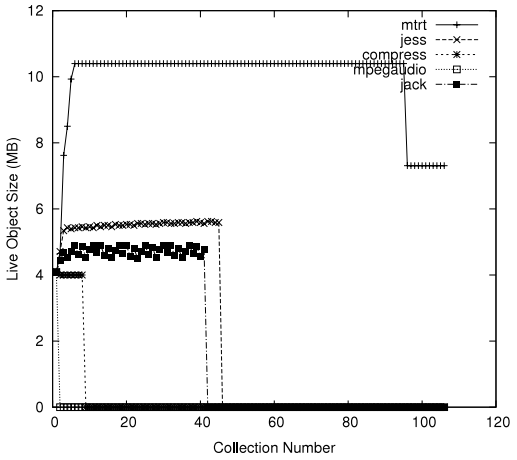
The above finding encourages us to try to predict the subsequent live object sizes, and based on which to adaptively adjust the size of the reserve space instead of always halving the heap.

The figure also reveals why the collection time of copying-based collectors decreases as the available heap space increases. Copying-based collectors take time that is 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. But because of the larger heap, as in Fig. 1(b), the number of collections is significantly reduced, leading to a much shortened overall collection time.

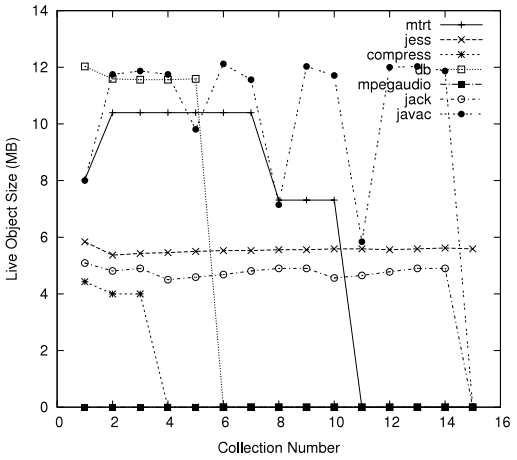
3. The skew-space collector

The previous section has revealed the fact that the live object size is likely to 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 that, we propose a *skew-space garbage collector* whose behavior is illustrated in Fig. 2.

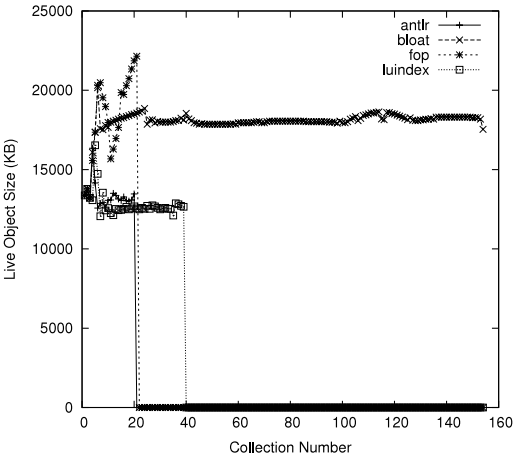
[1] In the 25 MB heap setting, *javac* and *db* were not runnable due to insufficient memory. *mpegaudio* seldom requested for an allocation; with a 25 MB heap there was only one collection, and with a 50 MB heap, GC was not even triggered.



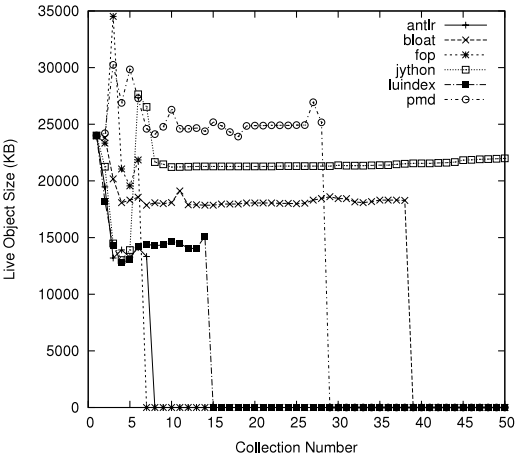
(a) 25 MB heap for jvm98 benchmarks.



(b) 50 MB heap for jvm98 benchmarks.

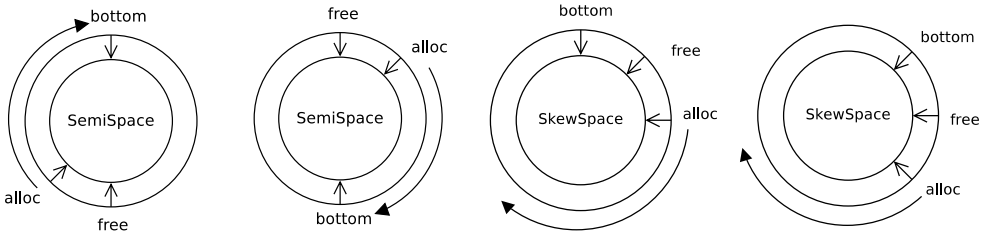


(c) 50 MB heap for DaCapo benchmarks.



(d) 100 MB heap for DaCapo benchmarks.

Fig. 1. Sizes of live objects.



(a) Semispace collector.

(b) Skewspace collector.

Fig. 2. Comparing two collectors.

In the two sub-figures depicting the process of the traditional semispace collection, from the pointer *bottom* to the pointer *free* is the reserved space, and from *free* to *bottom* is the active space. The 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 *bottom* and *free*. At the same time, the active space is set to be 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 most of the time such a reserved space is largely wasted, as the survivor rate typically is way below 100%.

The other two sub-figures describe two typical collections in our skew-space collector. Between *bottom* and *free* is the survivors' space (that occupied by the live objects) of the last collection, between *free* and *alloc* is the reserved space for

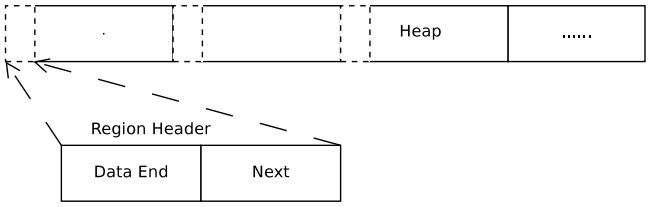


Fig. 3. The heap structure.

copying, and between *alloc* and *bottom* is the allocation space. We can see that this time the reserved space is not equal to half of the available space, but is the same as the survivors' size of the last collection. After every collection, the survivors are copied into the 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 the 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 to finish what is 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 each consisting of eight OS pages.[2] Each region has a header (see Fig. 3) containing 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 region in this heap space. The reason for this is that in the skew-space collector, every collection would cause the *bottom*, *free* and *alloc* pointers to move. When *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 move around freely in the 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 the region header is about eight bytes, so the space overhead is less than 0.2%.

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. The objects are allocated from the place designated by the *alloc* pointer, which is initially set to be the bottom of the heap. Later when the *alloc* pointer meets the *free* pointer, a garbage collection is triggered which moves the survivors to be beyond 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 relative to the previous collection.

How to advance the *alloc* pointer is a tricky problem. Close observations on Fig. 1 reveal that the live object size actually fluctuates, rather than always adhering to the "stable" line. If the *alloc* pointer is advanced by exactly the size of the survivors, mis-prediction will happen even when the live object size rises by just a little. To deal with this, we add two parameters, *flucBytes* and *protectBytes*, and initialize the former to be 0.[3] Then upon every mis-prediction, the following code is executed.

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

To understand the code, we generalize the variation of live object size as shown 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 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, and augmented by *protectBytes*, the reserve ratio may exceed 100%. We added a test to check for this and keep to the 100% value if this should happen.

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

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

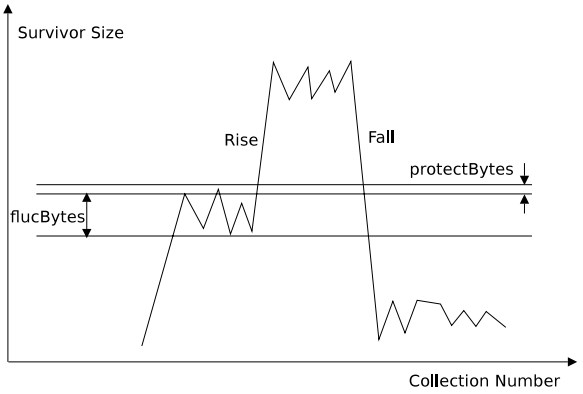


Fig. 4. The fluctuation wave.

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 strategy described in [9] for Appel-style generational collectors [26], except that the survivors are not copied into the mature generation but to the free space.

A collection starts by computing the entire root set and pushing the objects 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 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. We use the remaining objects in the tracing queue 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.

We should point out the difference between fall-back compacting and normal compacting. For our compacting collector, some of the objects in the operation area may already been copied to the reserved space. When the process of marking comes across such an object, it simply updates the referring pointers with the value stored in the forwarding word (which 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 include that due to moving objects and that due to traversing the entire heap. We expect the time spent on moving objects to be significant. 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 outside the main memory. The traversal of them, therefore, will result in a lot of page thrashing and significantly prolong the collection time.

4. Performance evaluation

4.1. Experimental setup

We employ all the applications in SPECjvm98 except *mpegaudio* which rarely requests an allocation 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.

Our skew-space collector is implemented using the MMTk toolkit [10] in the Jikes RVM [13]. The Jikes RVM is an open source virtual machine derived from IBM's Jalapeno [14]. It does not implement a bytecode interpreter, and adopts the Adaptive Optimization System [15] to perform online feedback-directed optimizations. MMTk is the memory management toolkit for the Jikes RVM. Communication between them is conducted on the VM and exported interfaces. MMTk divides the available space into several areas, including 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 the Jikes RVM is built. A *stopTheWorld* class extends *plan* by incorporating typical execution phases, which is further 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.

Table 2
The Percentages of False Predictions by *sks*.

Benchmark/Heap size	1	1.25	1.5	1.75	2	2.25	2.5	2.75	3
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
antlr	0	0	0	0	0	0	0	0	0
bloat	null	null	4%	0	0	0	0	0	0
fop	null	null	28%	13%	17%	0	20%	0	0
jython	null	null	3%	2%	3%	6%	3%	4%	4%
luindex	0	0	0	0	0	0	0	0	0
pmd	null	null	12%	14%	19%	20%	24%	21%	9%

These numbers are normalized by the size of minimum heap size.

Table 3
Number of Collections of *ss* minus that of *sks*.

Benchmark/Heap size	1	1.25	1.5	1.75	2	2.25	2.5	2.75	3
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
antlr	10	6	5	4	2	2	2	1	1
bloat	null	null	45	33	26	22	17	15	14
fop	null	null	5	4	3	2	1	1	1
jython	null	null	72	57	35	33	23	21	17
luindex	25	16	12	10	8	6	6	4	4
pmd	null	null	51	24	18	13	11	10	9

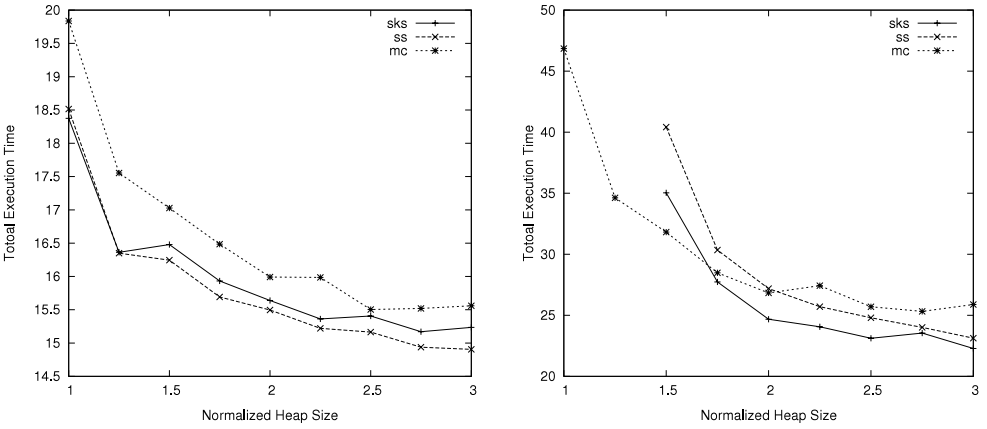
We revise the semispace collector to implement our version of skew-space collections. The space reservation ratio is at first set to be 100%, because at that time no statistics are available for the prediction. While the program is executing, memory usage information is gathered and used to compute the next ratio. In order to know whether our prediction is correct or not, we use a counter to keep track of the number of pages currently used by the mutators, and stop copying if the current reserved space is exceeded. 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 which will be compacted. Note that because of the particular design of MMTk we do not implement a cyclic heap, but instead we check the number of pages in order 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*). *ss* and *mc* are available in the MMTk environment, and the latter uses the LISP-2 algorithm to compact the heap. Essentially, *sks* tries to dynamically strike 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* would reserve less space than *ss*, and trigger fewer collections which ultimately would lead to reduced execution time. In the event of a false (inaccurate) prediction, *sks* needs to call on *mc* to treat the remaining objects.

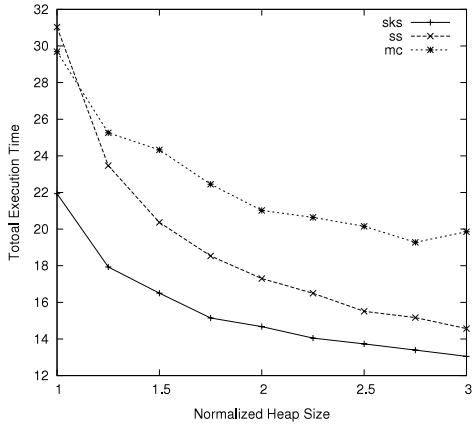
Fig. 5 shows the performance of the three collectors for the six benchmark applications over different heap sizes. In Fig. 5, we can see that most of the applications demonstrate a promising reduction in the execution time. Table 2 shows the mis-prediction percentages (the ratio between the number of collections with a fall-back compaction and the total number of collections) for different applications over the various heap sizes. "null" in the table means that for this heap size the collector is unable to run. The majority of the false predictions are well below 15%, and some have 0%. Table 3 gives the "difference" in number of collections between *ss* and *sks*. It clearly shows that as the heap grows in size, this difference decreases correspondingly, because the memory resource becomes not as tight. Note that the numbers in the first row of each table denote the heap sizes divided by the minimum memory requirement which for the jvm98 benchmarks is 25 MB.

In the figure, we can also see that the performance of *compress* is slightly inferior to that using *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, and are 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

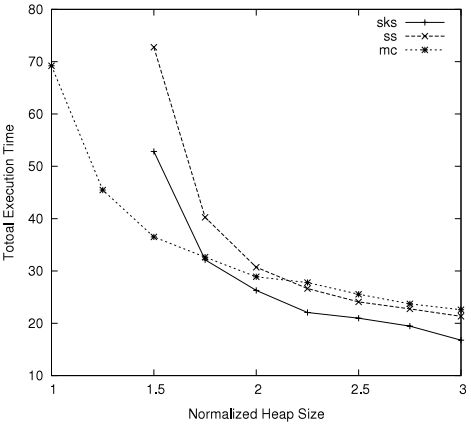


(a) compress.

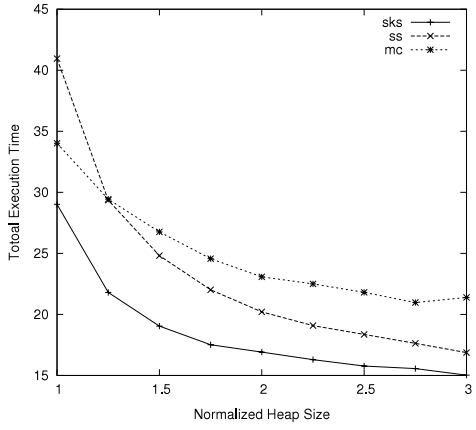
(b) db.



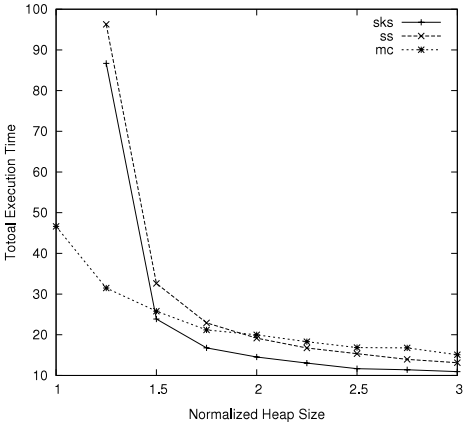
(c) jack.



(d) javac.



(e) jess.

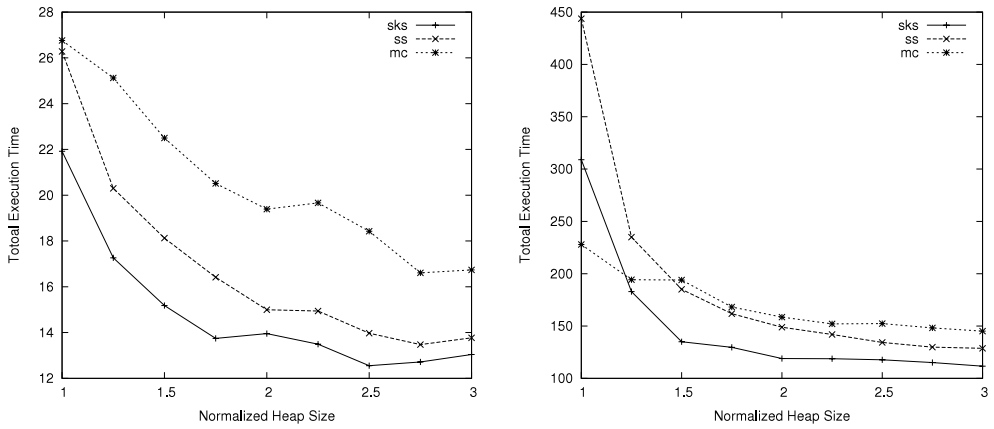


(f) mtrt.

Fig. 5. Comparisons using SPECjvm98 benchmarks.

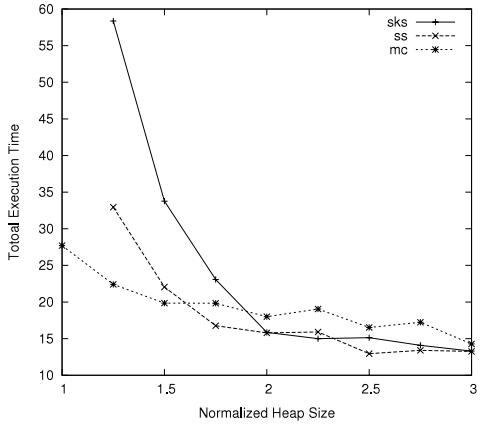
reserved by *ss*, and reduces our implementation to be similar to the latter. Table 3 also confirms our analysis, where there is no difference in the number of collections for this benchmark. The slightly poorer performance of *sks* is due to the relative complexity of our particular implementation of *sks*.

We further tested our collector using a newer benchmark suite—DaCapo [11] (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.

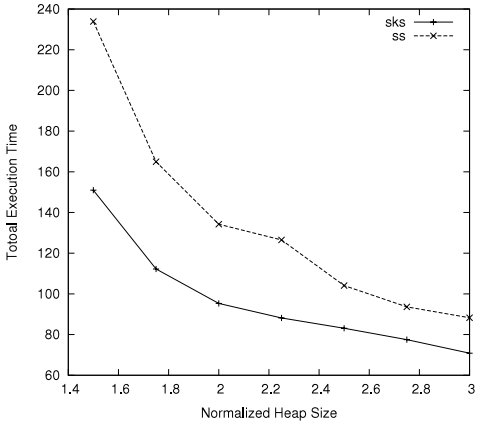


(a) antlr.

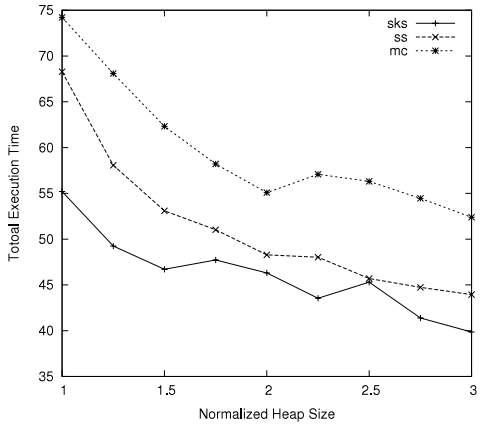
(b) bloat.



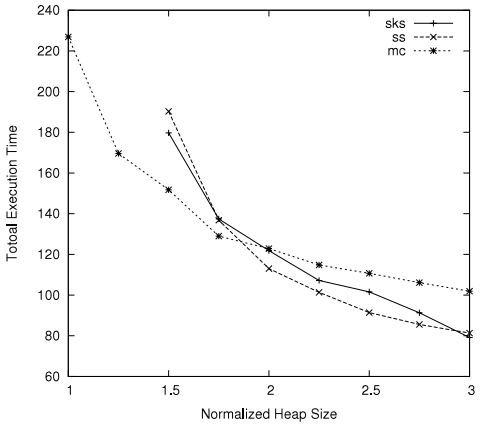
(c) fop.



(d) jython.



(e) luindex.



(f) pmd.

Fig. 6. Comparisons using DaCapo benchmarks.

Among the remaining benchmarks, *hsqldb*, *lusearch* and *xalan* are multi-threaded applications, and so are also excluded from our evaluation. Note also that *mc* could not run the *jython* benchmark under any heap size; therefore, in the corresponding diagram, only the performances of *ss* and *sks* are compared.

From the figure, it is evident that *sks* outperforms *mc* and *ss* for almost all the benchmark applications. Among the six applications, *antlr*, *bloat*, *jython*, *luindex* experience the largest improvements. In fact, the observations of Fig. 1 agree with

this finding. The total size of live objects seldom changes too significantly, although in the beginning all of the benchmarks go through a memory usage decline. A relatively sharp increase in memory consumption would lead to a false prediction. *fop* and *pmd* change more frequently in their live object size, so their performance experienced some degradation.

4.3. Comparison

Copying and compacting garbage collectors represent two extremes: the former sacrifices the space entirely, while the latter the time entirely. There are some variants which take the middle ground, for example [21], where they partition the space into N multiple windows and reserve only one window for copying. If the survivor rate turns out to be less than $\frac{1}{N}$, only one pass over the heap is needed. But if the rate grows larger, it displaces objects in different windows and triggers multiple passes on the heap. Our technique is similar to the windowing concept, but we adjust the fraction of the window to be conserved dynamically and flexibly according to the program's characteristics.

Our skew space garbage collector is not a new type of garbage collector, but the result of combining the merits of both copying and compacting collectors. It is built on the observation that the size of live objects will not vary too much from one collection to the next. This phenomenon is entirely experimental and has only been tested using Java benchmarks, and so it may not be universally true for all possible programs. For any program that behaves differently, our collector can switch back to a normal semispace collector, based on the false prediction rate. We argue, however, that owing to the statements proposed in Section 1, most program behaviors should obey this observed rule.

Although Table 2 suggests that most of the benchmark applications experience relatively low false prediction rates, several of them still show a high rate. This reflects a defect in our algorithm, which is that it cannot foresee those abrupt increases in memory requirements. Compilers may help tackle this problem by incorporating static analysis techniques such as those mentioned in [8] before the program is actually compiled.

5. Discussions

5.1. Compaction

In our experiments, although most of the time the predictions are correct, there are still several cases where the reserved space turns out to be not big enough for copying the live objects. For the remaining live objects after the skew-space copying, we resort to a mark compactor to relocate them as a whole to the end of the heap. This ensures the correctness of our algorithm, but the cost is high. Our experiments suggest that a compaction on average takes 0.4 s for jvm98 and 0.7 s for the DaCapo benchmarks. How to reduce this cost will be a meaningful pursuit.

Mark compactors linearly squeeze live objects into one end of the heap; in the process, they need to traverse the entire heap several times, touching both garbage and live objects. This affects the performance very much. In our collector, before a false prediction, most of the live objects must have already been copied into the reserved space, which makes the working space sparsely populated with live objects. Allocated objects are never visited once they become garbage or invalid. Due to the presence of virtual memory, these objects have likely been swapped out of the main memory. Touching them may result in many page faults and increase the collection time substantially.

As mentioned above, after a false prediction the working space is mainly populated with garbage and invalid objects. There must be many continuous free memory blocks. Our experiments using the benchmarks show that these blocks normally span over a couple of pages. If we build an index where every bit corresponds to a page in the working space and use this index to avoid touching these free pages, the touching of garbage objects can be considerably reduced. This is one possible way to deal with the problem.

5.2. Generational collectors

Our algorithm is very simple and therefore can be easily combined with other techniques (some are mentioned in Section 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.

According to the size of the nursery [21], 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 [24], only promote objects when they become 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 [26] triggers a collection when the heap is half full and therefore is more space efficient. However, the main problem with VG is that when the mature space grows to become too big, it squeezes the nursery into a small space and causes collection to take place too frequently. To tackle these problems, BG imposes a minimum size to which the nursery can be reduced. It starts a garbage collection immediately after the heap becomes half full or the minimum threshold is reached. We pick VG to discuss the application of our skew-space algorithm, but it can also be built on other collectors to dynamically conserve copying space.

For most of the VG types of garbage collectors there are two issues that need to be addressed. Take the Appel-style collector as an example.

- It always reserves half of the space so that the worst scenario where all the objects survive can be accommodated. But this scenario is very rare or indicates inefficiently assigned memory resources if it does happen many times.
- Although sounds simple, how to determine when the heap will become half occupied is a tricky problem. An Appel-style collector solves it through an indirect way. We suggest an alternative in the following.

It can be expected that, for generational collectors, the nursery survival rate is even lower than that of the whole heap. The experimental results show that normally this rate is well below 10% and sometimes drops very close to zero, which also explains why the static algorithm in [9] would work well. In view of a high infant mortality, traditional Appel-style collectors waste most of the reserved space. So clearly it makes sense to use space according to the memory usage pattern and reserve less than 50% of space for copying. The basic rule is the same as that applied to non-generational collectors: if the prediction fails, a compacting collector is called to serve both the minor and the major collections.

It is worth noting when to trigger a major collection. Objects are continuously allocated into the working space, which will trigger a minor collection when half of the working space is exhausted. As more and more live objects are promoted into the mature space after each collection, to determine at which collection the half heap will be consumed becomes a problem. The Appel-style collector will not call for a major collection until it finds that more than 50% of the heap is allocated. It then relocates part of the live objects and updates extra pointers to clear the space. Since we can predict the memory usage of these programs, a similar algorithm can be used to predict when this point will be reached and trigger a collector in advance.

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 [16] based on the argument that they are in fact duals of each other. [17] 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 around, which are directly 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 [18], and compacting collectors compact them [19]. 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 in moving collectors, the equivalent action is to advance the bump pointer by the requested size. But there are prices to pay. Semispace collectors need to reserve half of the available space for copying the survivors, while compacting collectors need to touch the heap several times. In [3], an index for compactors is proposed, which can be used to avoid touching the garbage objects at all. The performance of their collections increases as the heap residency drops, and so we can expect that it will work well as our fallback mechanism.

6.2. Space efficient collectors

As copying-based collectors need to reserve half of the available heap whereas compacting-based collectors do not, for any particular program there must be a point in the time-space tradeoff where the two kinds of collectors achieve the same performance. [23] 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, [20] 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 [21] 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 to one window which means that only one frame is reserved. [12] resorts to a 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 performance. The above two 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 [30] 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. [7,9] 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 advantages.

- We take advantage of the phenomenon we observed about live object size to dynamically set the reserved space, which is more accurate and precise. [30] does not reveal any implementation details, whereas [9] reserves space manually and therefore cannot adapt to the specific characteristics of different applications. In particular, [7] 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- and Appel-style generational collectors.
- We have discussed and devised different strategies to apply our algorithm to different generational collectors.

There is also a body of work on reducing the space consumed by non-copying garbage collectors; for example, [22], which introduces objects reuse in the design of garbage collection, and [28] which eliminates forwarding pointers and shortens object headers.

7. Conclusion

In this paper, we conjecture a memory usage phenomenon in typical object-oriented 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 in support of this conjecture. We then designed and implemented 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 semispace 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 interesting to see whether or not our dynamic memory usage prediction algorithm can be combined with existing static approaches [29] to achieve potentially greater improvements.

Although in our experiments most of the Java benchmarks display 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 exhibit similar characteristics so that our algorithm can be applied. Finally, 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.

Acknowledgement

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, Communication of the ACM 3 (4) (1960) 184–195.
[2] R. Jones, R. Lins, Garbage Collection: Algorithm for Automatic Dynamic Memory Management, John Wiley & Sons, 1997.
[3] L. Tong, F.C.M. Lau, Index-compact garbage collection, in: Asian Symposium on Programming Languages and Systems, 2010, pp. 271–286.
[4] P.R. Wilson, Uniprocessor garbage collection techniques, in: Proceedings of the International Workshop on Memory Management, 1992, pp. 1–42.
[5] B.K. Haddon, W.M. Waite, A compaction procedure for variable length storage element, The Computer Journal 10 (2) (1967) 162–165.
[6] H. Lieberman, C. Hweitt, A real-time garbage collection based on the lifetimes of objects, Communication of the ACM 26 (6) (1983) 419–429.
[7] J.M. Velasco, K. Olcoz, F. Tirado, Adaptive tuning of reserved space in an appel collector, in: European Conference on Objected-oriented Programming, 2004, pp. 542–558.
[8] V. Braberman, F. Fernandez, D. Garbervetsky, et al. Parametric prediction of heap memory requirements, in: Proceedings of the 7th International Symposium on Memory Management, 2008, pp. 141–150.
[9] P. MaGachey, A.L. Hosking, Reducing generational copy reserve overhead with fallback compaction, in: International Symposium on Memory Management, 2006, pp. 17–28.
[10] 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, 2004, pp. 137–146.
[11] S.M. Blackburn, R. Garner, C. Hoffman, The DaCapo benchmarks: Java benchmarking development and analysis, in: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2006, pp. 169–190.
[12] S.M. Blackburn, 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, 2008, pp. 22–32.
[13] B. Alpern, S. Augart, S.M. Blackburn, The jikes research virtual machine project: building an open-source research community, in: Open Source Software, IBM Systems Journal 44 (2) (2005) 399–417 (special issue).
[14] B. Alpern, C.R. Attanasio, J.J. Barton, The Jalapeno Virtual Machine, IBM Systems Journal 39 (1) (2000) 211–238.

[15] M. Arnold, S.J. Fink, D. Grove, Adaptive optimization in the Jalapeno JVM, in: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2000, pp. 47–65.
[16] D.F. Bacon, P. Cheng, V.T. Rajan, A unified theory of garbage collection, in: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2004, pp. 50–68.
[17] B.G. Zorn, Comparing mark-and-sweep and stop-and-copy garbage collection, in: ACM Conference on LISP and Functional Programming, 1990, pp. 87–98.
[18] C.J. Cheney, A nonrecursive list compacting algorithm, Communication of the ACM 13 (11) (1970) 677–678.
[19] H.B.M. Jonkers, A fast garbage compaction algorithm, Information Processing Letters 9 (9) (1979) 25–30.
[20] B. Lang, F. Dupont, Incremental incrementally compacting garbage collection, ACM SIGPLAN Notices 22 (7) (1987) 253–263.
[21] N. Sachindran, J.E.B. Moss, Mark-copy: fast copying GC with less space overhead, ACM SIGPLAN Notices 38 (11) (2003) 326–343.
[22] Z.C.H. Yu, F.C.M. Lau, C.L. Wang, Object co-location and memory reuse for Java programs, ACM Transactions on Architecture and Code Optimization 4 (4) (2008) 232–268.
[23] P.M. Sansom, Combining single-space and two-space compacting garbage collectors, in: Proceedings of the Glasgow Workshop on Functional Programming, 1991.
[24] D. Ungar, Generation scavenging: a non-disruptive high performance storage reclamation algorithm, ACM SIGSOFT Software Engineering Notes 9 (3) (1984) 157–167.
[25] D. Ungar, F. Jackson, Tenuring policies for generation-based storage reclamation, ACM SIGPLAN Notices 23 (11) (1988) 1–17.
[26] A.W. Appel, Simple generational garbage collection and fast allocation, Software Practice & Experience 19 (2) (1989) 171–183.
[27] D.A. Barrett, B.G. Zorn, Garbage collection using a dynamic threatening boundary, ACM SIGPLAN Notices 30 (6) (1995) 301–314.
[28] D.F. Bacon, P. Cheng, D. Grove, Garbage collection for embedded systems, in: International Conference On Embedded Software, 2004, pp. 125–136.
[29] E. Albert, S. Genaim, M.G. Zamalloa, Live heap space analysis for languages with garbage collection, in: International Symposium on Memory Management, 2009, pp. 129-138.
[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.