PAPER

# A Phase-Adaptive Garbage Collector Using Dynamic Heap Partitioning and Opportunistic Collection

**Yangwoo ROH**[†a)], ***Student Member***, **Jaesub KIM**[†], ***and*** **Kyu Ho PARK**[†], ***Nonmembers***

**SUMMARY** Applications usually have their own phases in heap memory usage. The traditional garbage collector fails to match various application phases because the same heuristic on the object behavior is used throughout the entire execution. This paper introduces a phase-adaptive garbage collector which reorganizes the heap layout and adjusts the invocation time of the garbage collection according to the phases. The proposed collector identifies phases by detecting the application methods strongly related to the phase boundaries. The experimental results show that the proposed phase-adaptive collector successfully recognizes application phases and improves the garbage collection time by as much as 41%.

***key words:*** *application phase, escape pattern, garbage collection, heap organization, opportunistic collection*

## 1. Introduction

Garbage collection (GC) is an automatic memory management technique that is widely used in managed execution environments such as Java. GC relieves developers from error-prone explicit memory reclamation but application performance is hurt by the additional GC work. To reduce GC overhead, the collector should accurately predict the lifetime of objects and reclaim dead objects as early as possible. Otherwise, either the collector spends GC time in reclaiming live objects or the heap space is wasted due to dead objects.

However, several studies [1]–[5] show that accurately predicting object lifetimes is difficult to achieve. The prediction mechanisms in those works either fail to attain performance improvement or are not applicable to various applications. This is because objects usually have a wide range of lifetimes [1], [4] and even a single allocation site can allocate objects of significantly varying lifetimes as the application executes [2].

Therefore, most collectors rely on a simple heuristic to manage all objects, regardless of the object type, the allocation site or the time at which an object is allocated. The generational garbage collector (GenGC) [21] exploits the simple heuristic known as the *weak generational hypothesis*. This holds that the recently allocated young objects tend to die young. When an application runs out of space, therefore, GenGC preferentially collects the *nursery* (the region dedicated to young objects) rather than the entire heap. Surviving objects are considered long-lived and are promoted to the older region to avoid further processing.

Manuscript received January 15, 2009.
Manuscript revised May 5, 2009.

[†]The authors are with the Department of EECS, KAIST, Daejeon, Republic of Korea.
a) E-mail: ywroh@core.kaist.ac.kr

DOI: 10.1587 transinf.E92.D.2053

Although GenGC works well for many applications, it suffers from two shortcomings. First, the objects allocated since the nursery almost filled have little time to die [13]. Thus, GenGC promotes the short-lived objects which will die soon to the older region. Second, GenGC misses the best time for GC yielding high GC efficiency. Old objects sometimes die *en masse* [1]. However, GenGC rarely collects the older region and dead objects in that region are not collected in a timely manner. This results in poor heap space utilization.

These shortcomings indicate that it is important to consider application behaviors in an effort to improve GC performance. The live size variation of the SPECjvm98 benchmarks [26] is given in Fig. 1. We define a phase as a period of the similar variation pattern of the live size. A phase can be envisioned as resembling a *ramp* (rapidly growing), a *plateau* (roughly flat) or a *cliff* (greatly decreasing). During the ramp phase, the allocated objects are unlikely to die. At the cliff phase, many old objects die *en masse*. Hence, if the phases are identified, the collector can improve GC performance 1) by avoiding GC during the ramp phase and 2) by reclaiming old objects after the cliff phase.

Many researchers have strived to predict object lifetimes or application behaviors [4]–[20]. However, most works fail to reclaim objects in a timely manner because they invoke GCs when the space fills. Although some works find the good GC timing, they require an additional algorithm to predict where to find garbage.

If the collector knows the time when many objects die

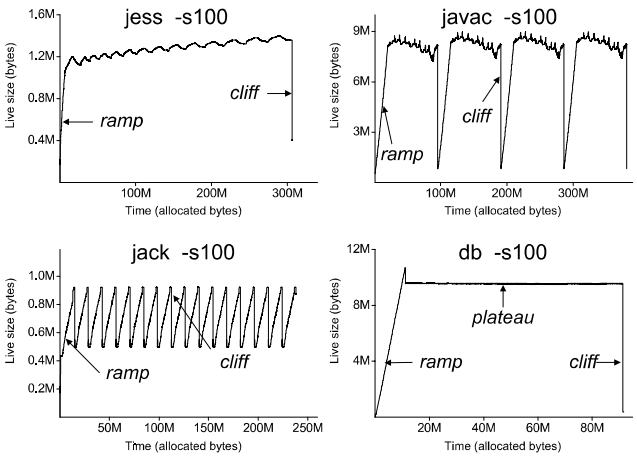



**Fig. 1** The live size variation of the benchmarks in SPECjvm98.

*en masse* and the region on which most dead objects reside, that region can be efficiently collected. To this end, we propose a phase-adaptive garbage collector, in which the heap is partitioned by application phases and the related objects are dynamically aggregated in the same partition using the connectivity information. Then, each partition is collected using either the *in-place* GC or the *phase-boundary* GC according to the associated phase. Especially, we attempt to reduce the overhead of GC during the ramp phase and the cliff phase. Phases are dynamically identified by monitoring application methods representing the phase boundaries. The proposed collector was implemented in the SableVM [25] and evaluated using SPECjvm98 [26] and Java Olden Suite [27]. Compared with the traditional generational copying collector, the proposed collector improves GC time by as much as 41%.

The remainder of this paper is organized as follows: Section 2 presents several related works. Section 3 explains the design of the proposed idea. The experimental results will be presented in Sect. 4. Section 5 discusses on the extension of our work. Finally, the conclusion will be given in Sect. 6.

## 2. Related Works

The *pretenuring* approaches [4]–[7] predict object lifetimes to directly allocate objects expected to be long-lived into the older generation. They can avoid the copying of long-lived objects. However, they still fail to reclaim dead objects in the older region in a timely manner, whereas we efficiently collect those objects by identifying the cliff phase.

The compile-time memory management approaches depend on static analyses to predict object lifetimes. Escape analysis [8], [9] finds the objects not escaping their creation stack frames. Those objects are allocated on the stack rather than the heap and automatically reclaimed at the end of methods. However, the detected objects tend to be short-lived. The GC for the older generation is unlikely to benefit from this approach. Liveness analysis [10]–[12] finds the code points at which an object is last used and reclaims it by inserting *free* statements at compile-time. Although this approach well reclaims individual dead object, it is challenging to reclaim a group of objects dying *en masse*. We efficiently reclaim those objects using the phase-boundary GC. Moreover, most compile-time approaches ignore the important features of Java such as dynamic class loading and reflection.

Some studies predict object lifetimes using the connectivity information. Hayes [15] uses the *key object opportunism* that a group of objects die *en masse* when a key object of that group dies. This is similar to ours in that it segregates objects and finds the GC timing (similar to the cliff phase) by the connectivity. However, it is not easy to find a key object and its deathtime. We exploit application phases to partition the heap and thus, the best GC timing for each region is easily determined using the phase. We also improve GC efficiency during the ramp phase. *Connectivity-Based GC* (CBGC) [16] uses the connectivity (gathered by pointer analysis) to statically partition the heap and to determine which region to collect. Like our collector, it focuses GC effort on the regions yielding the good GC efficiency. However, CBGC still invokes GCs when the heap space fills and fail to collect garbage in a timely manner. On the contrary, we associate application phases to each heap region. The number of the regions, the region to be collect and the GC timing are adaptively determined according to the phase.

*Beltway* [14] can flexibly layout the heap using the *increments* (small heap regions). However, its layout is statically determined, regardless of application phases, whereas PAGC repartitions the heap as the application executes. Beltway gives the young objects more time to die by collecting objects in decreasing age order but its GC is still triggered when the heap space fills. Similarly to CBGC, it fails to collect garbage in the older region in a timely manner.

Rather than object lifetimes, some studies predict the time yielding good GC efficiency. The *preventive memory management* [18] uses off-line profiling to detect the phase boundary. It invokes GCs only at the end of a phase while the heap freely grows in the middle of a phase. We rely on an online detection mechanism and prevent each heap region from unlimitedly growing by adaptively collecting it. Buytaert *et al.* [19] and Xian *et al.* [20] use *favorable collection points* identifed by off-line profiling and *allocation pauses* (the interval between two subsequent allocations) measured online, respectively. The GC timings detected by those schemes are similar to the cliff phase. However, none of both schemes gives the hints about where to find garbage. They require an additional algorithm to determine the region to be collected. On the contrary, we easily identify which region to collect at each phase because each phase is associated with the heap partitions.

## 3. Phase-Adaptive Garbage Collector

### 3.1 Overview

Our phase-adaptive garbage collector (hereafter abbreviated as PAGC) exploits application phases 1) to group objects according to the similarities of their lifetime [15], [16] and 2) to determine the best GC time for each group [18]–[20]. PAGC consists of the components listed below:

- *Phase identification*: PAGC identifies phases by finding application methods (termed *sink* methods) representing phase boundaries. (Sect. 3.4)
- *Heap partitioning*: The heap is dynamically partitioned at the phase boundaries. In addition, using the connectivity between objects and phases, the objects of similar lifetimes are aggregated in the same partition (Sect. 3.2).
- *Opportunistic GC*: Each partition is associated with a phase. Either the *in-place* GC or the *phase-boundary* GC is applied according to the phase behavior (Sect. 3.3).

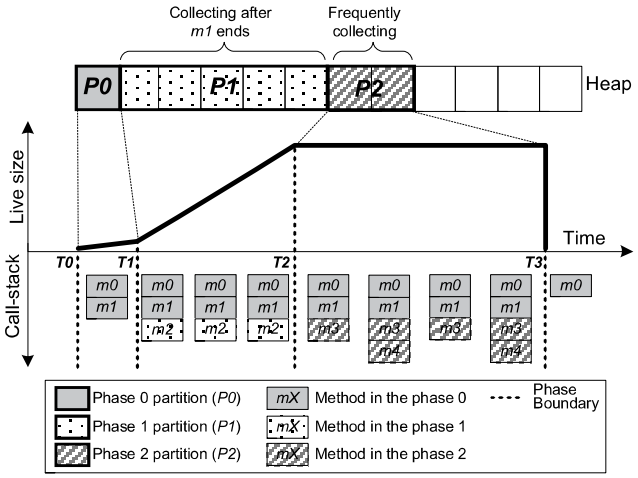



**Fig. 2** Overview of PAGC. The heap is divided at the phase boundaries, which are represented by the sink method.

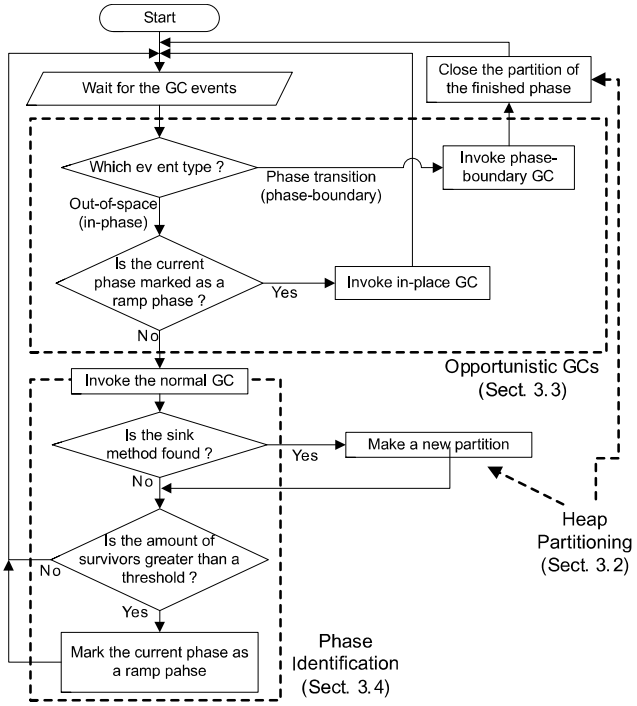



**Fig. 3** An overall execution flow of PAGC.

First, we introduce how the heap partitioning combined with the phase can improve the GC performance. The overall relationship between the phases, methods and heap partitions of PAGC is shown in Fig. 2. The application is initially assigned *P0* at startup. When the *m1* method calls the *m2* method at time *T1*, a ramp phase begins and *P1* is created. When the *m1* method calls the *m3* method at time *T2*, a plateau phase begins and *P2* is created. The plateau phase continues during the *m3* method and the *m4* method execution. After the *m1* method ends at time *T3*, a cliff phase occurs. Given this heap layout, PAGC can apply different GC techniques to each phase. During the ramp phase (*T1*→*T2*), the newly allocated objects are unlikely to die young. Thus, PAGC does not collect P1 but increases the size of *P1*. On the contrary, the objects allocated during the plateau phase (*T2*→*T3*) are likely to die young since the live size does not vary. PAGC frequently reclaims *P2*. Finally, at time *T3*, the collector forces GC for *P1* because the objects allocated during the previous ramp phase are likely to die. In this example, *the m1 method is a good sink method because it is involved in all phase transitions*.

However, PAGC relies on neither off-line profiling nor static analysis. Thus, the phase should be identified online, as well as the heap partitioning. Figure 3 depicts the overall flows of those processes of PAGC. Before any phase is identified, PAGC invokes GCs only when the heap space fills as the conventional collector. We define this type of GC as the normal GC. During every normal GC, PAGC tries to 1) recognize the sink methods and 2) determine whether or not the current phase is a ramp phase. Once the sink method is recognized, PAGC additionally monitors that method to identify the phase transition. If a phase transition occurs, PAGC invokes the phase-boundary GC for the associated partition. If the space is exhausted in the middle of a phase, PAGC determines whether or not to invoke the in-place GC according to the phase type.

To recognize the sink methods, PAGC monitors the object escaping pattern. However, unlike escape analysis [8], [9] finding objects whose lifetime is bound to the lifetime of its creation frame, *we find the stack frames to which the lifetimes of many long-lived objects are bound*. The detailed explanation is given in Sect. 3.4. To identify the ramp phase, PAGC measures the volume of objects survived the normal GC. This procedure is further described in Sect. 3.3.

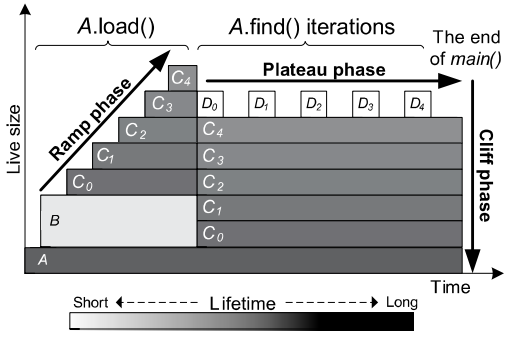
### 3.2 Heap Partitioning

We explain the heap partitioning of PAGC with an example pseudo code in Fig. 4 (a). In this example, it is clear that the phase transitions occur at the beginning of the *load* method and the *find* method. During the execution of each method, the application shows different phasic behaviors. While the *load* method executes, the live size rapidly increases due to the newly created objects, as denoted by $C_n$ (allocated by the code line 13). On the contrary, during the *find* method iterations, the live size looks roughly constant. This is because the objects $D_n$ (allocated by the code line 18) are short-lived and only one object among them is alive at any time. Therefore, to segregate objects by their lifetime similarity, PAGC divides the heap when the *load* method or the *find* method begins.

However, naïve heap partitioning may fail to aggregate objects of similar lifetimes. As shown in Fig. 4 (b), the color of objects $C_n$ is similar to the color of the object $A$. Namely, the lifetime of objects $C_n$ is similar to the lifetime of object $A$ rather than the object $B$. Unless we know the lifetime of an object ahead of its allocation, the objects of significantly different lifetimes may be collocated in a single partition. Then, it is difficult to determine when to collect and which

```
1: void main (...) {
2:     Database A = new Database();
3:     A.load();      // initialize database
4:     while (...)
5:         A.find();     // search data
6: }
7: class Database {
8:     Vector entries;
9:     void load() {
10:        DataStream B = new DataStream(...);
11:        B.load(...);      // load data
12:        while (...) {
13:            Entry C = new Entry(...);
14:            entries.addElement(C);    // add to database
15:        }
16:    }
17:    void find() {
18:        Enumeration D = entries.elements();
19:        ...; // access data via the object D
21: }
```

(a) An example pseudo code




(b) The live size variation of the above code. Each box and its color represent an object and its lifetime, respectively. The darker color means the longer lifetime.

**Fig. 4** A phased application example and its live size.

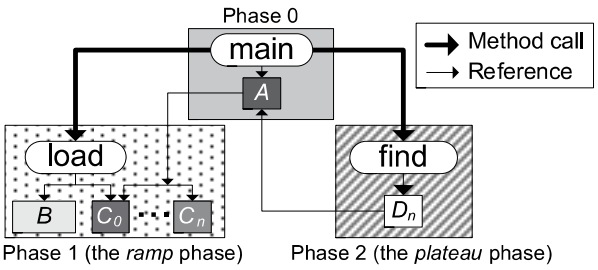



**Fig. 5** The object reference graph. The objects $C_n$ allocated in phase 1 is connected to the object $A$ in phase 0. Thus, they do not die when phase 1 ends because the object $A$ is still alive.

type of GC technique to use. To address this ambiguity, during the normal GC, PAGC relocates objects expected to live longer than the belonging phase. The destination of the relocation is determined by examining the connectivity between objects and phases, which is illustrated in Fig. 5. The long-lived objects $C_n$ become reachable from phase 0 via the reference field (the *entries* in the code line 8) of the object $A$. Due to that reference, the objects $C_n$ die only after the object $A$ dies. Consequently, the objects $C_n$ should be relocated to the partition to which the object $A$ belongs. Using the relocation, each phase contains objects which live no longer than the belonging phase.

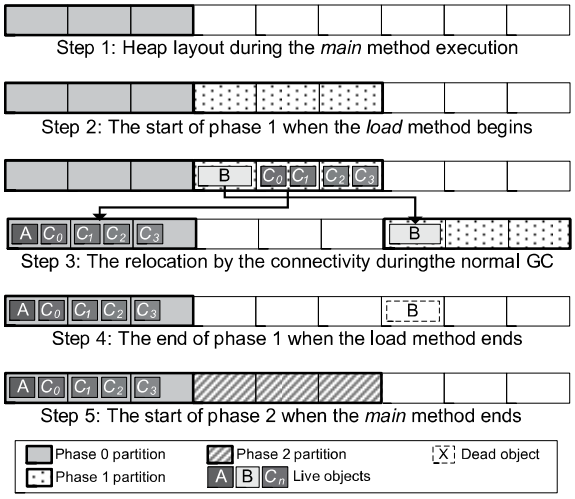



**Fig. 6** PAGC heap partitioning and object relocation.

Figure 6 illustrates these processes applied to the code in Fig. 4 (a). Each partition is initially assigned a small memory (three heap chunks in this example). At Step 1, the *main* method executes and the application is in phase 0. When the *load* method is called, a new partition is assigned to it (Step 2). As the *load* method proceeds, the space fills up and the normal GC is invoked (Step 3). During GC, the traditional generational collector promotes surviving objects, whereas PAGC relocates only the objects connected to another longer phase. Only the objects $C_n$ are copied to phase 0. As a result, the phase 1 partition contains the object $B$ whose lifetime is bound to that of phase 1. At Step 4, phase 1 ends and the object $B$ become garbage. This garbage is collected by the phase-boundary GC (Sect. 3.3). Finally, at Step 5, the phase 2 partition is created when the *find* method begins.

### 3.3 Opportunistic GC

PAGC suggests two types of opportunistic GCs: the *in-place* GC and the *phase-boundary* GC. The in-place GC attempts to reduce the overhead of GC invoked in the middle of a phase. First, we will explain why the in-place GC should be used despite the knowledge on the phases. By monitoring the sink method, PAGC identifies the time when the phase transition occurs but does not predict how many objects will be allocated. Moreover, the minimum size of a partition is significantly affected by the associated phase as well as the amount of objects. For example, during the plateau phase in Fig. 4 (b), regardless of the total volume of objects $D_n$, the space for only one object $D_n$ suffices because the object $D_n$ can be reused at each iteration. Therefore, it is hard to determine the size of each partition. Unless we allow a partition to unlimitedly grow during a phase, an application may run out of space. In PAGC, each partition is initially assigned a small space and the normal GC may be invoked in the middle of a phase. However, PAGC reduces GC overhead by

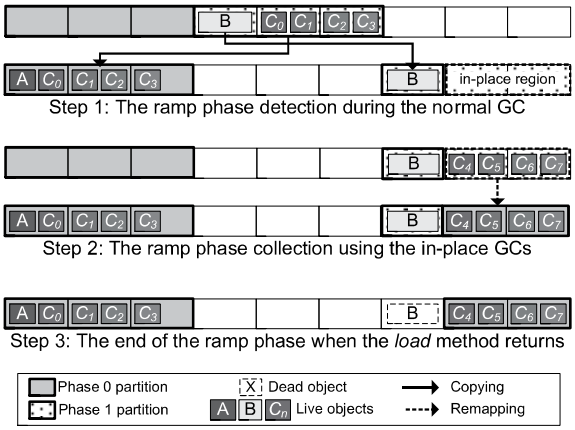



**Fig. 7** Opportunistic GC Type 1: the in-place GC.

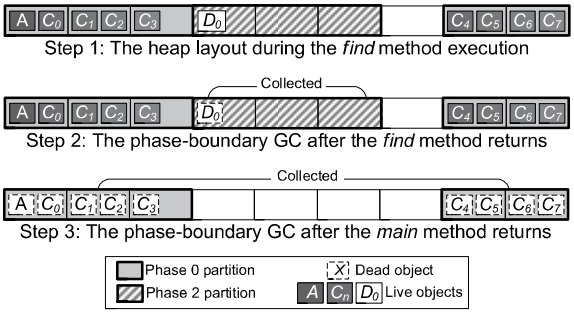



**Fig. 8** Opportunistic GC Type 2: the phase-boundary GC.

invoking different GC schemes in a phase adaptive manner. This is the reason why we identify the phase type as well as the phase boundary. If PAGC knows that the current phase is the ramp phase upon heap space exhaustion, the in-place GC is invoked. Otherwise, the normal GC is invoked. If most allocated objects are short-lived, the normal GC works well.

The in-place GC remaps the entire memory belonging to the ramp phase to a different partition with the expectation that *most objects allocated during the ramp phase will survive.* Neither the object traversal nor the object promotion is required. Thus, the GC overhead is almost removed. PAGC identifies a ramp phase by measuring the survival ratio of the previous normal GC. To reasonably satisfy the above expectation on the ramp phase, we regard a phase as a ramp phase only if the survival ratio is greater than 80% of the partition size. The in-place GC is disabled if any phase transition occurs (i.e., the end of the method created the current phase). Figure 7 shows the in-place GC applied to the code in Fig. 4 (a). At Step 1, phase 1 is regarded as a ramp phase because all objects of Phase 1 survived the normal GC. Then, PAGC marks a part of the phase 1 partition as the in-place region. If this partition again fills (Step 2), PAGC simply remaps the marked region of phase 1 to phase 0. Then, the additional two chunks are assigned to phase 1 to allow subsequent allocations. Finally, PAGC removes the phase 1 partition when the *load* method returns.

On the contrary, the phase-boundary GC is invoked at the end of a phase, which is identified by monitoring the sink methods. It collects the partition assigned to the finished phase. In addition, PAGC considers the cliff phase because a lot of long-lived objects are likely to die *en masse* in the cliff phase. In PAGC, the cliff phase coincides with the end of the sink method. This is because the lifetimes of many long-lived objects are bound to the lifetime of the stack frame of the sink method (Sect. 3.1 and 3.4). PAGC invokes the phase-boundary GC when the sink method returns. Because the phase to which the sink method belongs is also known, PAGC easily determines which partition to collect. An example of the phase-boundary GC applied to the code in Fig. 4 (a) is given in Fig. 8. At Step 1, the *find* method is running in phase 2. At Step 2, the *find* method returns and the *main* method resumes. Then, PAGC detects the end of phase 2 and invokes the phase-boundary GC for the phase 2 partition. At Step 3, PAGC detects the cliff phase because the *main* sink method returns. The long-lived objects *A* and *C* are successfully collected by the phase-boundary GC.

### 3.4 Phase Identification

The heap partitioning and the opportunistic GCs described so far rely on the phases represented by the sink methods. Now, we describe how to find the sink methods. From the example in Fig. 4, we observe three correlations between phases and long-lived objects: 1) the duration in which long-lived objects are dominantly allocated may be a ramp phase, 2) the duration in which few long-lived objects are allocated may be a plateau phase and 3) the duration in which long-lived objects die *en masse* may be a cliff phase. Hence, to find sink methods, it is important to find application methods affecting the behavior of long-lived objects.

For this purpose, we consider the object escape-pattern but exploit it differently than escape analysis [8], [9]. Basically, an object escapes its creation stack frame when it becomes reachable from other stack frame. The lifetime of an escaping object depends on the lifetime of the stack frame to which the object escapes. Combining this behavior with the above correlations, we derive two properties for the sink method. First, long-lived objects which die in the same cliff phase are highly likely to escape to the stack frame of the same method, which can be used for the sink method. Second, the volume of live objects reachable from that method is very large because a large volume of objects escaped to it die *en masse*. The *main* method in Fig. 4 (a) satisfies both properties. All long-lived objects $C_n$ die after the *main* method ends. The volume of live objects (the objects $C_n$ and $A$) of the *main* method is far greater than that of others.

We profiled benchmarks from SPECjvm98 [26] and Java Olden Suite [27] to gather the statistics on those properties. From the second column to the fifth column in Table 1, the volume and the average lifetime of each object category are presented. The lifetime of an object is measured in terms

**Table 1** Benchmark statistics.

| Benchmark | Short-lived objects | | Long-lived objects | | Detected Sinks | Long-lived objects escaping into each method | | The max live size |
|---|---|---|---|---|---|---|---|---|
| | Volume | Lifetime | Volume | Lifetime | | Volume (%) | Lifetime (%) | |
| javac | 311 MB | 39 K | 53 MB | 66 MB | *Javac.compile* | 44 MB (84%) | 56 MB (15.4%) | 7.2 MB |
| | | | | | *VM_Bottom* | 7 MB (13%) | 214 MB (58.8%) | 0.8 MB |
| raytrace | 154 MB | 2.8 KB | 4.2 MB | 149 MB | *Scene.RenderScene* | 3.7 MB (87%) | 150 MB (94.9%) | 3.7 MB |
| | | | | | *VM_Bottom* | 0.3 MB (8%) | 157 MB (99.4%) | 0.5 MB |
| jack | 218 MB | 4 KB | 8.8 MB | 14 MB | *VM_Bottom* | 7.7 MB (87%) | 14 MB (6.2%) | 0.9 MB |
| | | | | | *Main.RunBenchmark* | 1.0 MB (11%) | 3.2 MB (1.4%) | 0.1 MB |
| mpegaudio | 0.4 MB | 19 KB | 0.4 MB | 0.3 MB | *VM_Bottom* | 0.4 MB (97%) | 0.3 MB (37.5%) | 0.4 M |
| db | 76 MB | 3 KB | 10 MB | 79 MB | *Main.run* | 8.9 MB (85%) | 79 MB (90.8%) | 8.8 MB |
| | | | | | *Database.read_db* | 1.1 MB (10%) | 9 MB (10.3%) | 1.1 MB |
| compress | 15 MB | 24 KB | 90 MB | 101 MB | *Harness.inst_main* | 89 MB (99%) | 99 MB (94.3%) | 6 MB |
| jess | 289 MB | 3.5 KB | 2.7 MB | 124 MB | *Jess.run_jess* | 1.7 MB (63%) | 138 MB (47.3%) | 1 MB |
| | | | | | *Node2.CallNode* | 0.5 MB (20%) | 10 MB (3.4%) | 98 B |
| power | 24 MB | 2.1 KB | 0.9 MB | 23 MB | *Power.main* | 0.7 MB (79%) | 23 MB (95.8%) | 1.1 MB |
| | | | | | *VM_Bottom* | 0.2 MB (20%) | 24 MB (99.7%) | 0.2 MB |
| bh | 269 MB | 3.6 KB | 0.9 MB | 23 MB | *BH.main* | 2.5 MB (88%) | 26 MB (9.2%) | 1.1 MB |
| | | | | | *Tree.vp* | 1.0 MB (8%) | 1 MB (0.4%) | 476 B |
| health | 56 MB | 4.6 KB | 14 MB | 38 MB | *Health.main* | 13 MB (98%) | 38 MB (54.3%) | 11 MB |

of number of allocated bytes since the object allocation. In this paper, a long-lived object is defined as an object which lives longer than 20% of the maximum live size [4]. For example, the maximum live size of *javac* is 8.6 Mbytes (in Table 2) and the threshold is set to 1.7 Mbytes. Usually, object lifetimes are classified into *short*, *long*, or *immortal* [4]. However, the age-based classifications does not fit to PAGC because its heap partitioning relies on application phases. Thus, we simply use "long-lived objects" to refer to objects escaping to the sink method.

The last 4 columns of Table 1 show the statistics on the sink methods of each benchmark. We measured the volume of long-lived objects escaping into the stack frame of each method. The top one or two methods ranked by their volumes are listed in column 6. The *VM_Bottom* method denotes the initial stack frame. Column 7 and column 8 show the volume of long-lived objects and their average lifetime, respectively. The values in parenthesis of column 7 and column 8 are the percentage in the total volume of all long-lived objects and the percentage in the total execution time, respectively. This result supports the first property described above. The majority of long-lived objects escape into one or two methods. For example, for *javac*, 84% of long-lived objects escape to the *Javac.compile* method. We can see that, for *raytrace*, *db* and *power*, the lifetimes of long-lived objects are close to the total execution time. This means that the ramp phase occurs early because most long-lived objects are allocated at the beginning of the execution. Finally, column 9 presents the maximum volume of live objects of each method. It supports the second property described above. As expected, most sinks can access live objects greater than hundreds of kilobytes, whereas the remaining non-sink methods (not shown due to space limitations) can access live objects less than tens of kilobytes.

We exploit the second property to detect the sink method at runtime. The first property is not suitable because the decision on the object longevity (short-lived or long-lived) can be made only after the maximum live size of an application and the object deathtime are known. During every normal GC, PAGC assesses the volume of live objects (termed the *max-escape-in*) for each method. If the max-escape-in of a method is greater than a given threshold, we simply regard the method as the sink method. In our evaluation, this threshold is set to 100 Kbytes.

Using this property, the collector can roughly predict the longevity of an object. If an object escapes into the sink, the object is considered long-lived. Then, it is possible to avoid the processing of the long-lived object by copying it out of the frequently collected region similarly to the pretenuring studies [5], [7].

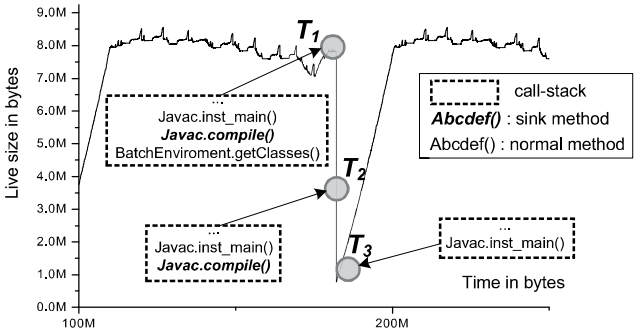
### 3.5 The Feasibility of the Sink Based Phase

In this section, we quantitatively show how well the sink reflects the phases of benchmarks. The sink of *javac* are depicted in Fig. 9 (a). *Javac* has the repetitive ramp, plateau and cliff phases. The *Javac.compile* method is identified as a sink. Each dashed box shows the application call-stack at the corresponding execution point. The end of the *Javac.compile* method at time *T3* coincides with the cliff phase. It also represents the phase boundary between the plateau phase (ending at time *T1*) and the ramp phase (beginning at time *T3*). We also present the execution result of *javac* in Fig. 9 (b). The phase-boundary GC occurs at the time marked with the circle. PAGC reclaims about 7.8 Mbytes of garbage with little copy overhead owing to the timely invoked phase-boundary GCs.

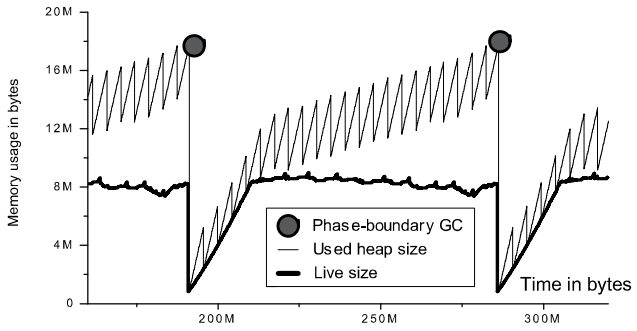
### 3.6 Online Sink Detection

We modified the object traversal order of the normal GC to measure the max-escape-in of each method. Basically, our traversal algorithm is based on the Cheney's copying algorithm [23]. In his algorithm, objects directly referred by

the root set (register, stack variables and global variables) are first copied. Then, the collector copies all objects that are transitively reachable (via the reference fields) from the objects already copied. Our algorithm differs from his algorithm in that the roots are chosen on a per stack frame basis. Namely, the traversed objects are grouped by the stack frame. As a result, we can easily measure the volume of live objects of each stack frame. Figure 10 illustrates our algorithm applied to the code in Fig. 4 (a). GC is invoked when the heap space fills during the *load* method execution. First, PAGC copies the object $A$ referred by the stack variable $V_A$ of the bottom method (i.e., the *main* method). Then, all objects $C_n$ referred by the object $A$ (via the reference $F_{entries}$) are copied. Those objects contribute to the max-escape-in of the *main* method. If no more reachable objects are left,




(a) The relationship between the sink and the phase.




(b) The heap usage of PAGC.

**Fig. 9** The phase adaptation result of *javac*.

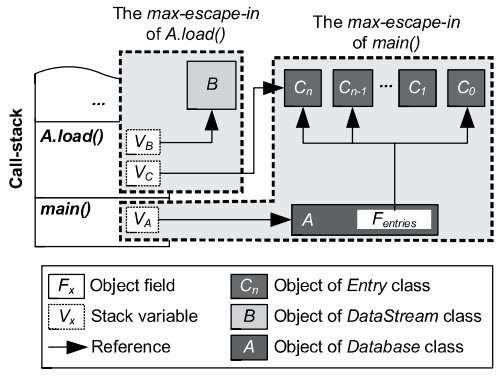



**Fig. 10** The sink detection mechanism based on the *max-escape-in* of each method.

the collector copies objects referred by the stack variables $V_B$ and $V_C$ of the *load* method. However, at this time, only the object $B$ is copied because all of objects $C_n$ have been already copied. The object $B$ contributes to the max-escape-in of the *load* method. In this example, the *main* method is identified as a sink method because its max-escape-in is very large. Then, a new partition is assigned for the subsequent object allocations of the *load* method. Hence, the objects in the partition to which the *main* method belongs are no longer exposed to GCs unless the phase transition occurs.

## 4. Evaluation

The performance of PAGC is evaluated against the traditional generational copying collector (GenGC). GenGC has a nursery and an old generation. The nursery size is fixed to 25% of the heap size [14], which is set to be two times the maximum live size [14]. Both generations are collected using the copying scheme. We implemented PAGC and GenGC on the SableVM [25] and used 10 benchmarks from SPECjvm98 [26] and Java Olden Suite [27]. The basic characteristic of each benchmark is shown in Table 2.

### 4.1 Implementation Detail

The heap is separated into three distinct regions, each of which is depicted in Fig. 11. The *immortal* space is used to store static objects such as *Class* instances and never collected. The *large object space* (LOS) is used to store objects bigger than 8 Kbytes. This region is collected using the *mark-and-sweep* scheme [24]. Most objects allocated by *compress* are located in LOS. The remaining heap space is used for small objects. It consists of $2^n$-aligned fixed-size (64 Kbytes) chunks. GenGC statically assigns chunks to the nursery and the old generation. PAGC dynamically remaps

**Table 2** Benchmark characteristic.

| Benchmark | Configuration | Total allocation (MB) | The maximum live size (MB) |
|---|---|---|---|
| SpecJVM98 | | | |
| javac | -s100 -m1 -M1 | 364 | 8.6 |
| raytrace | -s100 -m1 -M1 | 158 | 4.2 |
| jack | -s100 -m1 -M1 | 227 | 0.9 |
| mpegaudio | -s100 -m1 -M1 | 0.8 | 0.5 |
| db | -s100 -m1 -M1 | 87 | 10.2 |
| compress | -s100 -m1 -M1 | 105 | 6.6 |
| jess | -s100 -m1 -M1 | 292 | 1.4 |
| Java Olden Suite | | | |
| power | -s100 -m1 -M1 | 24 | 0.9 |
| bh | -s100 -m1 -M1 | 155 | 0.5 |
| health | -s100 -m1 -M1 | 70 | 11 |

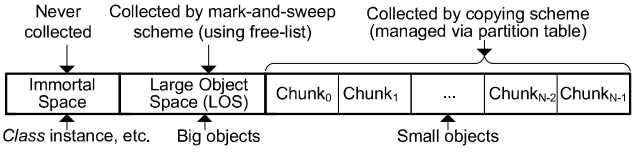



**Fig. 11** Overall heap organization.

chunks according to the phases. All user-initiated GCs (i.e., *System.gc*) are ignored. The partition management code is inserted before the object allocation. The write barrier for inter-partition references uses *remembered sets* [24].

### 4.2 GC Efficiency

First, the number of GCs of both collectors is given in Table 3. Although it is not relevant to the performance comparison because of a varying number of partitions of PAGC, it shows that PAGC tends to reduce the number of full GCs with more frequent partial GCs. For *javac* and *jack*, PAGC significantly reduces the number of full GCs. Second, we compare both collectors with respect to the *mark-cons-ratio* (MCR) in Fig. 12. It is defined as the total bytes copied during GCs divided by the total allocated bytes. PAGC outperforms GenGC by 19.9% on average (ranging from −3.3% to 49.7%). For *javac*, the phase-boundary GC plays a major role in the significant reduction of the MCR (an improvement of 39%). As shown in Fig. 9 (a), *javac* has very potential cliff phases in which the live size drops as much as 92% of the maximum live size. After each cliff phase, PAGC successfully invokes the phase-boundary GC and reclaims about 7.8 Mbytes of garbage with little cost. On the other hand, *raytrace*, *db* and *power* mainly benefit from the in-place GCs invoked during the ramp phase. *Raytrace* and *power* show the live size variation similar to that of *db* shown in Fig. 1. At the beginning of the benchmark execution, PAGC identifies the ramp phase of each benchmark and opportunistically invokes the in-place GCs during the ramp phase. For *jack*, the result is interesting. PAGC marginally improves the MCR despite the significant reduction of the number of full GCs. There are two reasons to consider. First, the live size difference of *jack* at the cliff phase (in Fig. 1) is smaller than that of *javac*. Thus, *jack* is far less improved than *javac* by the phase-boundary GC. Second, the cost of the frequent partial GCs offsets the benefit of the reduction in the number of full GCs. For *mpegaudio*, the number of total allocation bytes is just 0.8 Mbytes. Thus, the MCR is improved by 16.2% but the absolute improvement is negligible. For *compress*, both collectors show almost the same MCR. In particular, most objects of *compress* are allocated in LOS and separately managed. Thus, the MCR of *compress* is the lowest among all benchmarks although it allocates objects as much as 105 Mbytes.

**Table 3** Number of each collection.

| Benchmark | GenGC | | PAGC | |
|---|---|---|---|---|
| | Nursery | Full | Parital | Full |
| javac | 81 | 7 | 87 | 2 |
| raytrace | 81 | 0 | 83 | 0 |
| jack | 587 | 16 | 683 | 1 |
| mpegaudio | 4 | 0 | 3 | 0 |
| db | 19 | 0 | 23 | 0 |
| compress | 46 | 2 | 40 | 1 |
| jess | 465 | 2 | 466 | 2 |
| power | 76 | 0 | 77 | 0 |
| bh | 554 | 10 | 557 | 9 |
| health | 12 | 0 | 13 | 0 |

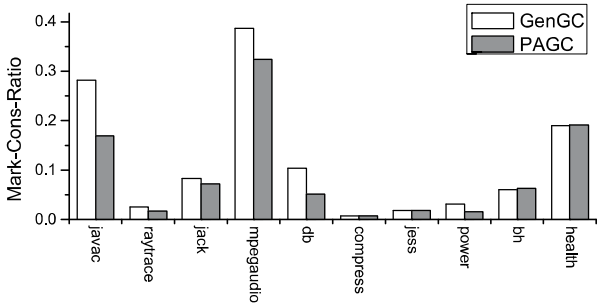



**Fig. 12** Mark-cons-ratio (total copied bytes total allocated bytes).

### 4.3 GC Pause

To show the responsiveness of PAGC, we present the average pause in Fig. 13. For *javac*, *raytrace*, *db* and *power*, PAGC greatly reduces the average pause as with the MCR. For *compress*, *jess*, and *health*, PAGC has no chance to invoke opportunistic GCs. Thus, it performs in a manner similar to GenGC. For *mpegaudio*, both of the number of GCs and the MCR are reduced, resulting in the longer average pause.

We also measured the *minimum mutator utilization* [29], which means the minimum fraction of time that an application can run in a given window. For example, given a window of 10 ms, if an application runs at least 2 ms across all windows of the same size, the MMU is 0.2. Thus, x-intercept indicates the maximum GC pause. The MMUs of 4 benchmarks (showing the noticeable differences) are given in Fig. 14. For *javac*, despite the shorter average pause, PAGC shows little reduction in the maximum pause because it can not avoid full GCs. *Jess* and *bh* show similar behaviors. PAGC also shows worse MMUs when the window is smaller than $10^6$ $\mu$seconds. This is because the GC is invoked at short intervals due to the phase-boundary GCs. For *db*, PAGC can not reduce the maximum pause although it invokes no full GCs. This is because PAGC relies on the normal GC to identify a ramp phase, resulting in the maximum GC pause. *Raytrace* and *power* belong to this case. *Jack* shows a unique behavior. Its live size stays at the maximum shortly. The efficiency of full GC is sensitive to the

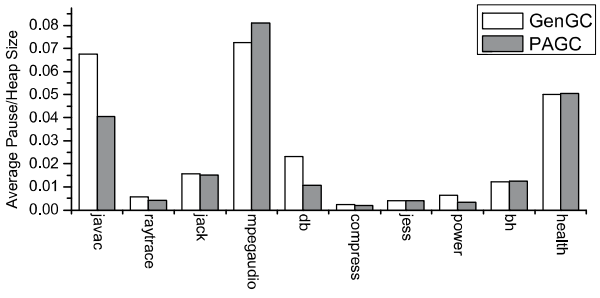



**Fig. 13** The average GC pause in terms of copied bytes. The y-axis is normalized to the heap size.

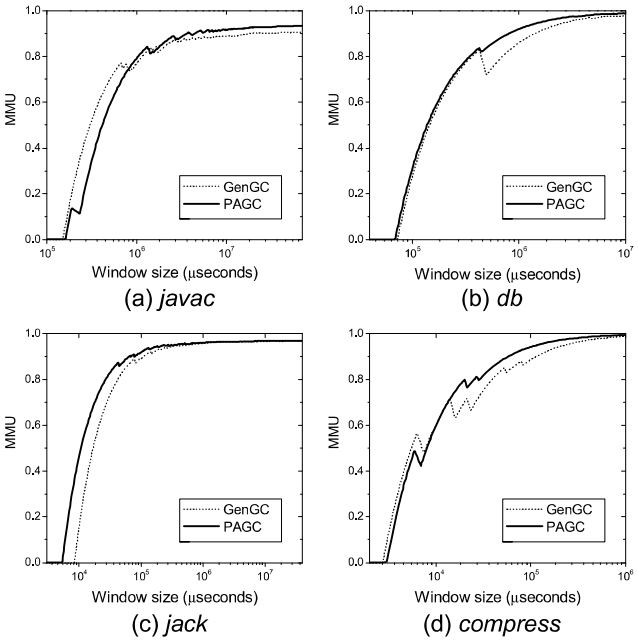



**Fig. 14** Minimum Mutator Utilizations (MMU).

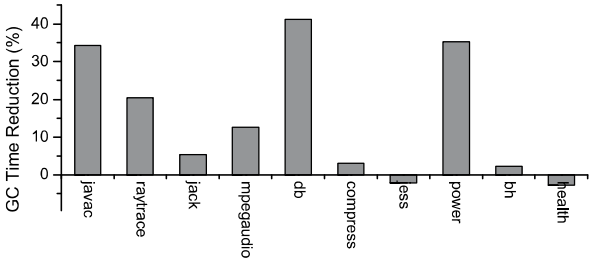



**Fig. 15** GC time reduction of PAGC relative to GenGC.

GC timing. This is the reason why PAGC still invokes full GCs but reduces the maximum pause. PAGC also shows better MMUs across most window sizes. This indicates that PAGC invokes short GCs at regular intervals instead of a few long GCs. For *compress*, the maximum pause and the GC overhead is very small because *compresss* dominantly allocates large objects separately managed in LOS.

### 4.4 GC Time and Overall Execution Time

This section shows the actual improvement of the GC time and the overall execution time. First, the reduction of the GC time of PAGC relative to GenGC is given in Fig. 15. The *javac*, *raytrace*, *db* and *power* are greatly improved. Either the phase-boundary GC or the in-place GC works well for these benchmarks. For *mpegaudio*, PAGC outperforms GenGC by 13% although PAGC find no chance for opportunistic GCs. Instead, the present connectivity based relocation of PAGC allows objects more time to die in the collected partition and reduces the total number of GCs.

Second, the improvement of the overall execution time is shown in Fig. 16. The graph shows reductions ranging

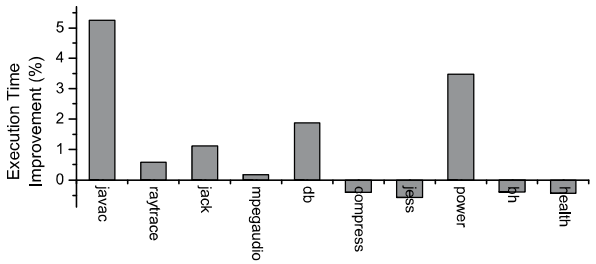



**Fig. 16** Execution time improvement of PAGC relative to GenGC.

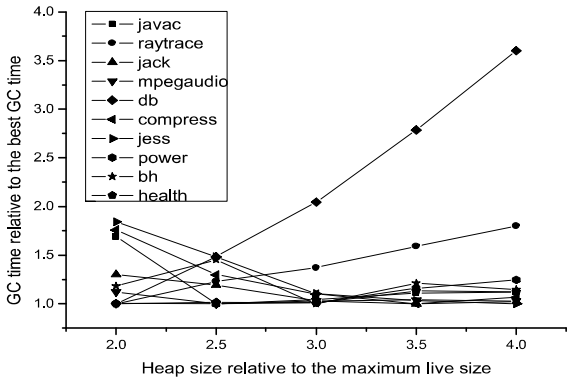



**Fig. 17** GC time with various heap sizes.

from −0.6% to 5.3%. Although the GC time reduction is significant (up to 41% for db), the total execution time is not greatly improved. It mainly results from the low overhead of GC on the present implementation. We used the SableVM interpreter (inlining enabled). The percentage of the GC overhead is at most 12% of the total execution time. However, on other fast JVMs, the GC overhead is usually larger than that of our JVM [20]. Therefore, we plan to evaluate PAGC on other high-performance JVM and expect that PAGC will achieve more improvement with respect to the total execution time. However, for *compress* and *bh*, PAGC marginally reduces the GC time but the total execution time is increased. There are two reasons for these cases: 1) although the partition management overhead is very low, it is not negligible for these benchmarks and 2) the write-barrier behavior changes. For *compress*, the write-barrier overhead increases by 8.0%. These overheads of PAGC outweigh the small improvement of the GC time.

### 4.5 Sensitivity to Heap Sizes

Finally, we show how PAGC is affected by the different heap sizes. Five heap size configurations of 2, 2.5, 3, 3.5 and 4 times the maximum live size are evaluated. These results are depicted in Fig. 17. Y-axis is the GC time relative to the best GC time of PAGC. As the heap size increases, the GC time of most benchmarks tends to be reduced but *db* and *raytrace* shows the increasing GC times. This is because the in-place GC is invoked for these benchmarks too late. The phase detection mechanism of PAGC is piggybacked onto the normal GC, which is increasingly deferred because the

invocation threshold of the normal GC is relative to the heap size. Consequently, the ramp phase of *raytrace* and *db* is identified too late, causing PAGC to miss the opportunities to invoke the in-place GC. Research on the optimal threshold for good GC performance and early phase detection is underway.

## 5. Future Work

So far, we have described PAGC in the context of single-threading. Now, we discuss on the multi-threading issues. By separately managing objects of each thread, it is possible to apply PAGC to multi-threaded programs. However, the simple management on a per-thread basis does not fit to objects shared by multiple threads. All threads may not execute in the same phase and thus, the shared objects can not be reclaimed by the opportunistic GC invoked by a certain thread. One possible solution is to segregate objects into the shared heap and thread-local heaps [28]. PAGC is still applied to the thread-local heaps, whereas the shared heap is managed by the traditional space-based GCs.

It would be also interesting to predict the behavior of shared objects. We expect that the interaction between threads should be considered as well as the behavior of each thread. For example, parallel applications usually synchronize the execution results of each thread at some execution points. That synchronization interval can be considered as a single phase. Then, the objects is segregated by this phase and the GC can be opportunistically invoked at the phase boundary.

## 6. Conclusion

We introduce a phase-adaptive garbage collector using application phases to partition the heap and to opportunistically invoke GCs for each partition. The proposed idea is based on the observations that most long-lived objects escape into the stack frames of a few sink methods and that those sinks are strongly related to the boundaries of application phases. The prototype was implemented and the performance was evaluated using 10 benchmarks. The evaluation result indicates that the proposed idea can identify application phases well and improve the GC efficiency significantly.

### References

[1] S. Dieckmann and U. Hölzle, "A study of the allocation behavior of the SPECjvm98 Java benchmarks," Proc. 12th European Conf. Object Oriented Programming, pp.92–115, July 1999.
[2] R. Jones and C. Ryder, "A study of Java object demographics," Proc. Int'l. Symp. Memory Management, pp.121–130, 2008.
[3] D. Stefanović, K.S. McKinley, and J.E.B. Moss, "On models for object lifetime distributions," Proc. Int'l. Symp. Memory Management, 2000.
[4] S.M. Blackburn, S. Singhai, M. Hertz, K.S. McKinely, and J.E.B. Moss, "Pretenuring for Java," Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications, pp.342–352, Oct. 2001.
[5] M. Jump, S.M. Blackburn, and K.S. McKinley, "Dynamic object sampling for pretenuring," Proc. Int'l. Symp. Memory Management, 2004.
[6] P. Cheng, R. Harper, and P. Lee. "Generational stack collection and profile-driven pretenuring," Proc. Conf. Programming Language Design and Implementation, 1998.
[7] W. Huang, W. Srisa-an, and J.M. Chang. "Adaptive pretenuring for generational garbage collection," Proc. IEEE Int'l. Symp. on Performance Analysis of Systems and Software, pp.133–140, March 2004.
[8] D. Gay and B. Steensgaard, "Fast escape analysis and stack allocation for object-based programs," Proc. Int'l. Conf. Compiler Construction, vol.1781 of Lecture Notes in Computer Science, Springer-Verlag, 2000.
[9] J. Whaley and M. Rinard, "Compositional pointer and escape analysis for Java programs," Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications, pp.187–206, 1999.
[10] S.Z. Guyer, K. McKinley, and D. Frampton, "Free-Me: A static analysis for automatic individual object reclamation," Proc. Conf. Programming Language Design and Implementation, pp.365–375, 2006.
[11] S. Cherem and R. Rugina, "Compile-time deallocation of individual objects," Proc. Int'l. Symp. Memory Management, pp.138–149, 2006.
[12] S. Cherem and R. Rugina, "Uniqueness inferrence for compile-time object deallocation," Proc. Int'l. Symp. Memory Management, pp.117–128, 2007.
[13] D. Stefanović, K.S. McKinley, and J.E.B. Moss, "Age-based garbage collection," Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications, pp.370–381, Oct. 1999.
[14] S.M. Blackburn, R. Jones, K.S. McKinley, and J.E.B. Moss, "Beltway: Getting around garbage collection gridlock," Proc. Conf. Programming Language Design and Implementation, June 2002.
[15] B. Hayes, "Using key object opportunism to collect old objects," Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications, pp.33–40, Oct. 1991.
[16] M. Hirzel, A. Diwan, and M. Hertz, "Connectivity-based garbage collection," Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications, 2003.
[17] P.R. Wilson and T.G. Moher, "Design of the opportunistic garbage collector," ACM SIGPLAN Notices, vol.24, pp.23–35, 1989.
[18] C. Ding, C. Zhang, X. Shen, and M. Ogihara, "Gated memory control for memory monitoring, leak detection and garbage collection," Proc. Workshop on Memory System Performance, pp.62–67, 2005.
[19] D. Buytaert, K. Venstermans, L. Eeckhout, and K.D. Bosschere, "Garbage collection hints," Proc. 1st Int'l. Conf. HiPEAC, pp.233–248, Nov. 2005.
[20] F. Xian, W. Srisa-an, and H. Jiang, "MicroPhase: An approach to proactively invoking garbage collection for improved performance," Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications, 2007.
[21] L. Henry and C. Hewitt, "A real-time garbage collector based on the lifetime of objects," Commun. ACM, vol.26, no.6, pp.419–429, June 1988.
[22] Java HotSpot Garbage Collection, http: java.sun.com ja- vase technologies hostspot gc index.jsp
[23] C.J. Cheney, "A nonrecursive list compacting algorithm," Commun. ACM, vol.13, no.11, pp.677–768, 1970.
[24] R. Jones and R. Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons, 1996.
[25] E. Gagnon and L. Hendren, "SableVM: A research framework for the efficient execution of Java bytecode," Proc. USENIX Java Virtual Machine, April 2001.
[26] Standard Performance Evaluation Corporation. SPECjvm98 benchmarks, http: www.spec.org osg jvm98
[27] B. Cahoon, Java-Olden Benchmarks, http: www-ali.cs.umass.edu ˜cahoon olden
[28] T. Domani, G. Goldshtein, E.K. Kolodner, E. Lewis, E. Petrank,

and D. Sheinwald, "Thread-local heaps for Java," Proc. Int'l. Symp. Memory Management, pp.76–87, 2003.
[29] P. Cheng and G. Belloch, "A parallel, real-time garbage collector," Proc. Conf. Programming Language Design and Implementation, May 2000.

**Yangwoo Roh** received the BS and the MS degrees in Electrical Engineering from Korea Advanced Institute of Science and Technology (KAIST) in 1997, 1999 respectively. He is currently a PhD student at KAIST. His research interests include garbage collection, virtualization, and operating systems.

**Jaesub Kim** received the BS degree in Electrical Engineering from Kyungpook National University in 2000 and the MS degree in Electrical Engineering from Korea Advanced Institute of Science and Technology (KAIST) in 2002. He is currently a PhD student at KAIST. His research interests include sensor network, internet server systems, and operating systems.

**Kyu Ho Park** received the BS degree in Electronics Engineering from Seoul National University, Korea in 1973, the MS degree in Electrical Engineering from the Korea Advanced Institute of Science and Technology (KAIST) in 1975, and the PhD degree in Electrical Engineering from the University de Paris XI, France in 1983. He has been a Professor of the Division of EECS, KAIST since 1983. He was a president of Korea Institute of Next Generation Computing in 2005–2006. His research interests include computer architectures, file systems, storage systems, ubiquitous computing, and parallel processing. Dr. Park is a member of KISS, KITE, Korea Institute of Next Generation Computing, IEEE and ACM.