Improve Google Android User Experience with Regional Garbage Collection

Yunan He, Chen Yang, Xiao-Feng Li

China Runtime Technologies Lab, Intel Corporation
{yunan.he, chen.yang, xiao-feng.li}@Intel.com

Abstract. Google Android is a popular software stack for smart phone, where the user experience is critical to its success. The pause time of its garbage collection in DalvikVM should not be too long to stutter the game animation or webpage scrolling. Generational collection or concurrent collection can be the effective approaches to reducing GC pause time. As of version 2.2, Android implements a non-generational stop-the-world (STW) mark-sweep algorithm. In this paper we present an enhancement called Regional GC for Android that can effectively improve its user experience. During the system bootup period, Android preloads the common runtime classes and data structures in order to save the user applications' startup time. When Android launches a user application the first time, it starts a new process with a new DalvikVM instance to run the application code. Every application process has its separate managed heap; while the system preloaded data space is shared across all the application processes. The Regional GC we propose is similar to a generational GC but actually partitions the heap according to regions instead of generations. One region (called the class region) is for the preloaded data, and the other region (called the user region) is for runtime dynamic data. A major collection of regional GC is the same as DalvikVM's normal STW collection, while a minor collection only marks and sweeps the user region. In this way, the regional GC effectively improves Android in both application performance and user experience. In the evaluation of an Android workload suite (AWS), 2D graphic workload Album Slideshow is improved by 28%, and its average pause time is reduced by 73%. The average pause time reduction across all the AWS applications is 55%. The regional GC can be combined with a concurrent GC to further reduce the pause time. This paper also describes two alternative write barrier designs in the Regional GC. One uses page fault to catch the reference writes on the fly; the other one scans the page table entries to discover the dirty pages. We evaluate the two approaches with the AWS applications, and discuss their respective pros and cons.

Keywords: Small Device, Runtime System, Memory Management, Garbage Collection

1 Introduction

Recent advances in computing technology are enabling the widespread use of smarter mobile computing devices. Here smarter means more powerful software stack that supports a variety of applications. Garbage Collection (GC) is a key component of the systems to provide automatic memory management. GC helps to improve both the application developers’ productivity and the system’s security. However GC is not without drawbacks. In performance wise, automatic memory management could be less efficient than well-programmed explicit memory management. In user experience wise, the GC pause time cannot be too long to stutter the game animation or webpage scrolling. The user experience is more critical for smart devices since most of the applications are user-inactive.

Google Android is an excellent software stack for mobile computing system. Android user applications are programmed in Java language, built on top of a Java-based application framework and core libraries, deployed in bytecode form and running on a virtual machine called DalvikVM. Android memory management has a two-level design. Underlying is the dlmalloc module managing the process virtual memory. DalvikVM has a garbage collector built on the dlmalloc to manage the runtime objects life cycles. As of version 2.2, Android implements a non-generational stop-the-world (STW) mark-sweep GC. The occasional pauses of garbage collections sometimes are long enough to be noticeable to the users, which is not pleasant user experience.

There are usually two ways to reduce the collection pause time, i.e., generational GC or concurrent GC. Generational GC is designed based on the “generational hypothesis” that most objects die young. A generational GC partitions the heap into regions (generations) for different ages of objects. When a generation space becomes full, the objects referenced from older generations and root set are promoted to next older generation. In this way, only one generation space needs to be collected in one collection cycle, thus the collection time is reduced compared to the entire heap collection. A common generational GC design requires write barrier to catch the cross-generation references, and usually copies the surviving objects for promotion. Since Android GC is built on dlmalloc, which does not allow moving objects, it is hard to enhance it to be generational.

A concurrent GC usually has a dedicated thread running in parallel with the application execution. It tries not to pause the application when possible. Concurrent GC can achieve much shorter average pause time than STW GC, but it requires write barrier to catch the reference updates in order to maintain the correctness of the concurrent execution. Thus concurrent GC has lower overall performance than STW GC. A concurrent GC actually is a superset design of a STW GC. Under certain conditions it falls back to STW collection. For example, when the system available free memory is not enough to sustain the application’s execution before the concurrent GC finishes scanning the object graph, it has to suspend the application for GC to recycle the memory. The maximal pause time of a concurrent GC can be longer than its STW counterpart.

In this paper, we propose a regional GC for Android. The regional GC is similar to a generational GC but partitions the heap into regions according to the different regions' properties. It does not require copying objects from one region to another hence it is easy to implement in Android on top of dlmalloc. The regional GC can achieve the benefits of a generational GC in that it mostly collects only one region so the pause time is reduced largely. Compared to concurrent GC, the regional GC development is much simpler. More importantly the regional GC not only achieves shorter pause time, but also achieves higher performance than the original STW GC, which is impossible with a concurrent GC. The major contributions of this paper include:

- It discusses the design of Android GC, and then proposes a regional GC that exploits the Android heap properties.
- The paper evaluates the regional GC with a set of Android Workload Suite (AWS) to understand its characteristics.
- The paper also describes and evaluates two alternative write barrier designs in the Regional GC. One is to use page protection at user level; the other is to scan the page table entries in the OS kernel.

2 Related Work

McCarthy invented the first tracing garbage collection technique: the mark-sweep algorithm in 1960 [1] and Marvin Minsky developed the first copying collector for Lisp 1.5 in 1963 [2]. Many tracing garbage collectors have been developed to manage the entire heap with mark-sweep or copying algorithm in a stop-the-world manner. GC spends considerable time in scanning the long-live objects again and again.

Generational GC [3] segregates objects by ages into two or more generations. Since "most objects die young" [3][4][5], it is cost-effective to collect the young generation more frequently than the old generation, hence to achieve higher throughout. Variations on generational collection include older-first collection [13] and the Beltway framework [14]. By collecting only a part of the heap, the pause time can be reduced as well. Another similar approach is Garbage-First GC [15]. Garbage-First GC partitions the heap into a set of equal-sized heap regions and remembered sets record pointers from all regions. So it allows an arbitrary set of heap regions to be chosen for collection. The regional GC in this paper is similar to the generational GC. But it's different from the approaches described above since it partitions the heap according to different region properties instead of object ages.

In order to further reduce the pause time of the STW collections, concurrent garbage collections have been developed [6][7][8][15]. Concurrent GC uses dedicated GC thread(s) to run concurrently with the mutator thread(s). Compared to a STW GC, since the total workload for a collection is the same and write barrier has runtime overhead, concurrent GC usually achieves shorter pause time but degrades the application overall performance.

There have been some efforts to share the data and/or code across different runtime instances. The Multi-tasking VM from Oracle has explored a few solutions [9]. They find the J2SE is not a good environment to justify the data sharing; instead, it is useful for J2ME environment, because the memory is scarce on small devices, and a large fraction of the footprint is taken by the runtime representation of classes. In these circumstances, a JVM that shares most of the runtime data across programs can be extremely valuable.

The write barrier used by many GC algorithms has been implemented in a number of ways, for example remembered sets [4], card marking [10][11], and etc. One variant of card marking write barrier uses the page protection provided by operating system [7].Meanwhile the GC community have investigated the possibility of using page dirty bits to replace the page fault handling for write barrier implementation [12]. The authors combine the remembered set and card marking schemes. Although a user-level page table dirty bit could be favorable, it has not been actually implemented. In our regional GC, the default write barrier is a page fault handling write barrier and we also implement a system call for Android that can effectively substitute the page fault handling write barrier.

3 An Overview of the Regional GC

The regional GC is designed specifically for the heap layout of DalvikVM in Google Android 2.2. In Android system, a demon process called Zygote is created during the system initialization. Zygote preloads system classes and some common data. A new application is started by Zygote forking a new process that runs the new application code in a new DalvikVM instance. The new process shares Zygote's space at the forking point. In this way, all the application processes in Android share a same space with Zygote in copy-on-write manner. This space holds the preloaded data and is seldom modified. We call this space the Class Region. After the new application is started, it then creates a new space for the application's private dynamic data. The application only allocates objects in this space and a collection is triggered when it is full. We call this space the User Region.

The default GC in DalvikVM is a mark-sweep collector. In marking phase, it scans the object graph from root set references to identify the live objects in both class and user regions. In sweeping phase, the collector sweeps only the user region. The class region and user region have following properties:

1. The class region is usually much larger than the user region for common Android applications. That means the marking time in the class region is much bigger than in the user region. We give more data in the evaluation section.
2. The class region is seldom written. The most data in this region are class objects. The class static data are stored outside of the region. The user region has all the runtime objects generated by the applications.
3. The class region is much more expensive to write than the user region. A first write to a page in this region triggers the copy-on-write handling in the OS kernel to allocate a new page for the writing process.

4. It is too expensive to sweep (i.e., write) the class region, so the class region is only scanned in marking phase for correct reachability analysis, but not swept in sweeping phase. The user region is both scanned and swept.

Our regional GC exploits the regional heap layout in Android. It has two kinds of collections as a generational GC: minor collection and major collection. The minor collection in the regional GC only collects (mark-and-sweep) the user region. The major collection behaves the same as the default Android GC.

Similar to a generational GC, write barrier is used to ensure the correctness. During the application execution, the write barrier tracks the cross-region references from other regions to the user region, and records them in a remembered set. When the heap is full, a collection is triggered. The remembered set is scanned together with the root set. The object graph traversal does not enter other regions except the user region.

The major collection is similar to the original mark-sweep GC. The remembered set is cleaned at the beginning of the major collection because it scans the all the regions from the root set. The only difference is that regional GC needs to remember all the cross-region references discovered in the marking phase. The remembered set is used for next minor collection together with those cross-region references caught by the write barrier.

By default the regional GC always triggers the minor collection except two cases. One is when the remembered set has no free slot available; the other case is when a minor collection does not return enough free space.

The idea of regional GC is to reduce the marking time. It is orthogonal to the STW or concurrent collection, and can be used to reduce their marking time.

4 Regional GC Design Details

In this section we describe the details of our regional GC implementation in Android. We implement the regional GC in Android 2.2.

4.1 The heap layout

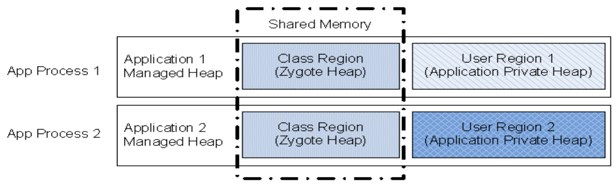
Shared Memory
App Process 1
Application 1 Managed Heap
Class Region (Zygote Heap)
User Region 1 (Application Private Heap)
App Process 2
Application 2 Managed Heap
Class Region (Zygote Heap)
User Region 2 (Application Private Heap)

Fig. 1. Regional heap layout designed in DalvikVM

Figure 1 illustrates the heap layout of DalvikVM, where two applications' process heap is shown. Each process heap has a class region and a user region. The class region is shared across the processes.

Class region is created and populated by the initializing Zygote process. The Zygote process is the parent process of all the application processes. All of them share the Zygote heap, i.e., the class region. The objects in the class region are mostly the class objects preloaded by the zygote process, and some other common system objects necessary for the initialization. An application never allocates objects in its class region.

The user region is created when an application is started and it is private to each application. An application allocates objects in the user region. The virtual memory management of the region is delegated to the underlying dlmalloc library.

A heap region is a contiguous space. Mark-bit tables are allocated outside the heap to indicate the objects status in the heap.

4.2 The minor collection

The minor collection is the default collection of the regional GC. It only scans and sweeps the objects in the user region. To ensure the collection correctness, all the references from outside of the user region are recorded in the remembered set by the write barrier during the application execution.

The Root Set And Remembered Set.

In the regional GC, a minor collection starts tracing from the root set and remembered set.

- Root set

As any other GC, the regional GC enumerates the root set from the runtime stack and global variables. Different from other GC, the regional GC does not enumerate the class static variables, because the regional GC wants to avoid the scanning of the class region. A class' static data is allocated with dlmalloc outside of the class region when the class object is loaded. There is no specific region for the class static data in Android, so the class static data can only be accessed via the class objects in the class region. Because the regional GC wants to avoid scanning the class region, it has to catch the references in the class static data with write barrier.

- Remembered set

The remembered set has the references to the user region from outside. It includes the references from the class region and from the class static data. Both of them are caught by the write barrier.

Since the remembered set is used by the minor collection, it must be prepared before a minor collection. There are following scenarios for the remembered set to record the references:

1. At the beginning of the application execution, the remembered set is empty. It starts to record the references to the user region from outside with the write barrier;
2. When the user region is full, and a minor collection is triggered, the remembered set is used together with the root set, but the content of the remembered set is kept

during the collection. After a minor collection, the remembered set continues to record the references;

3. When a major collection is triggered, the remembered set is cleared, and only the root set is used for tracing the live objects. During the collection, the references to the user region from the external are recorded in the remembered set. This is prepared for the next minor collection.

The Write Barriers

During the application execution, when the mutator modifies the object reference field (the field address is called the slot), write barrier is triggered and checks the positions of the source object and the target object. If source object is outside of the user region and the target object is in the user region, the reference slot is recorded in the remembered set. There are two kinds of write barriers in the regional GC. One is to track the static data updates, and the other is to track the class region update.

- Write barrier for the static fields

As we have explained above, in DalvikVM design, the static data of the classes are allocated outside the class region, and we have to use write barrier to catch the writes in reference slots. We instrument the VM execution engine (such as the interpreter and/or JIT compiler) with write barrier for static field operations, including opcode sput-object. The write barrier records the static reference slots in the field remembered set.

- Write barrier for the class region

For the class region, the regional GC uses page protection to catch the reference slots updates. The page protection write barrier is a variant of card marking. It depends on the underlying OS support to trap the writes to the protected virtual memory pages.

At the beginning of the application execution, a user-level signal handler for SIG_SEGV is registered. At a point before the first object allocated in the user region and also at a point after a major collection, the class region pages are protected to be read-only. Whenever there is a write into the class region, a page fault is triggered. The OS kernel then delivers a SIG_SEGV signal to the user application. When the execution exits from the page fault handling in the kernel, it enters the user registered signal handler. The signal handler changes the page protection mode to be read-write and records the virtual memory address of the page in the page remembered set.

Figure 2 illustrates how the page protection write barrier works.

- Step 1.The steps 1.a, 1.b and 1.c show the process of page protection. There are three objects A, B and C in three pages of the class region. D is allocated in the user region.
- Step 2.When the application executes C.m=D, the page fault is triggered.
- Step 3.The SIG_SEGV signal handler reads the fault address of the reference field C.m. And then changes Page 3 to be writable and remember it in the page remembered set.
- Step 4.When a minor collection happens; only the objects in Page 3 are scanned. Other pages of the class region are untouched in the collection.

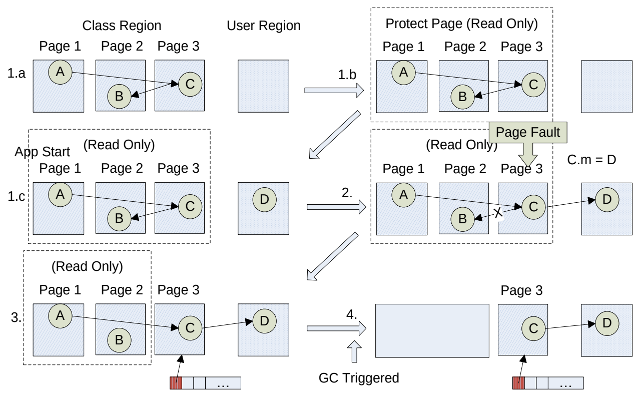

Fig. 2. Page-protection write barrier

In order to reduce the runtime overhead of page fault, we also implement an alternative write barrier for the class region. We call it PTE-scan write barrier.

- PTE-scan write barrier

The PTE-scan write barrier is different from the page protection write barrier; it does not depend on page fault signal handling. Instead, it implements a new system call in the Linux kernel to iterate the page table entries to track the dirty pages.

```
iterate_pte(pmd_t *pmd, unsigned long from, unsigned long to, func* operation)
{
    pte_t *pte = get_pte(pmd, from);
    for(addr=from; addr != end; addr += pagesize)
      *operation(pte ++, addr);
}

iterate_PMD(pud_t *pud, unsigned long from, unsigned long to, func* operation)
{
    //iteate PMD, it's similar to iterate_PGD
}
```

```
iterate_PUD(pgd_t *pgd, unsigned long from, unsigned long to, func* operation)
{
    //iterate PUD, it's similar to iterate_PGD
}

iterate_PGD(unsigned long from, unsigned long to, func* operation)
{
    pgd_t *pgd = get_pgd(from);
    for(addr = from; addr != end; pgd++, addr=next){
      next = start_address_in_next_pgd(addr, end);
       iterate_PUD(pgd, addr, next, operation);
    }
}
```

Fig. 3. Pseudo code of iterating memory area

In the regional GC implementation, the new system call "dirty_pages" scans all page table entries (PTE) belongs to the class region and checks the dirty bit in the PTE. If the dirty bit is set, the virtual memory address of the page is recorded. The system call records the dirty page addresses in an array and then copies the array back to the user space as the page remembered set.

The system call has two functionalities, one is to iterate the heap to record the dirty bit and the other one is to reset the dirty bit for the memory area. Both of them needs to recursively walk the page table for the memory area. It starts from PGD and scans the PUD, PMD and PTE in depth first order. Figure 3 is a pseudo code of walking the PTE.

During walking the memory area, the call back function is called for very page table entry. In clean dirty mode, if the pte dirty bit is set, then system erase the dirty bit. In get dirty mode, it record the address in array. Below is the pseudo code:

```
clear_dirty_callback (pte_t *pte, unsigned long addr){
  if(pte_dirty(*pte)) set_pte(pte, pte_mkclean(ptent));
}
dirty_page_callback(pte_t *pte, unsigned long addr){
  if(pte_dirty(*pte)) dirty_array[count++] = addr;
}
```

In the regional GC, the dirty_pages system call is invoked when a minor collection starts. Another system call “clear_dirty” is invoked at the beginning of the application exection and right after a major collection.

Marking Phase

When the user region is full, a minor collection is triggered and the application threads are paused. The first phase of the collection is the marking phase that identifies the reachable objects. The object graph tracing algorithm is a depth-first algorithm and similar to traditional mark-sweep GC, and reuses the Android GC infrastructure.

A mark-bit table is used to track the objects status. Since the objects in the heap are 8-bytes aligned, one bit in the mark-bit table maps to 8 bytes in managed heap. The marking phase firstly prepares the root set and remembered set to start from, and then traces the object graph to mark the live objects.

- Root set and remembered set preparation

The regional GC marks all the objects referenced by the root set in the mark-bit table. It then marks all the objects referenced by the field remembered set in the mark-bit table. Then the GC iterates the page remembered set. For each page in the set, the GC marks all the objects in the mark-bit table according to the allocation-bit table. (The allocation-bit table is similar to the mark-bit table. Whenever the mutator allocates an object, it marks a correspondent bit in the allocation-bit table.) This initial set of the marked objects are the root objects for object graph traversal.

- Object graph traversal

In this step, the regional GC iterates the mark-bit table from the beginning to the end, tracing the live objects in depth-first order. The first step is processing root objects. For each marked object in the mark-bit table, the GC scans its reference fields to find out the referenced objects. If the referenced object is in the user region, the GC pushes the referenced objects in the mark stack. The next step is processing the mark stack. The GC gets a object from mark stack. If the object is in the class region, the GC does not proceed to scan it. If it's in user region, the GC put all the referenced objects in the mark stack. This process continues until the mark stack is empty. Figure 4 shows the execution flow of the process.

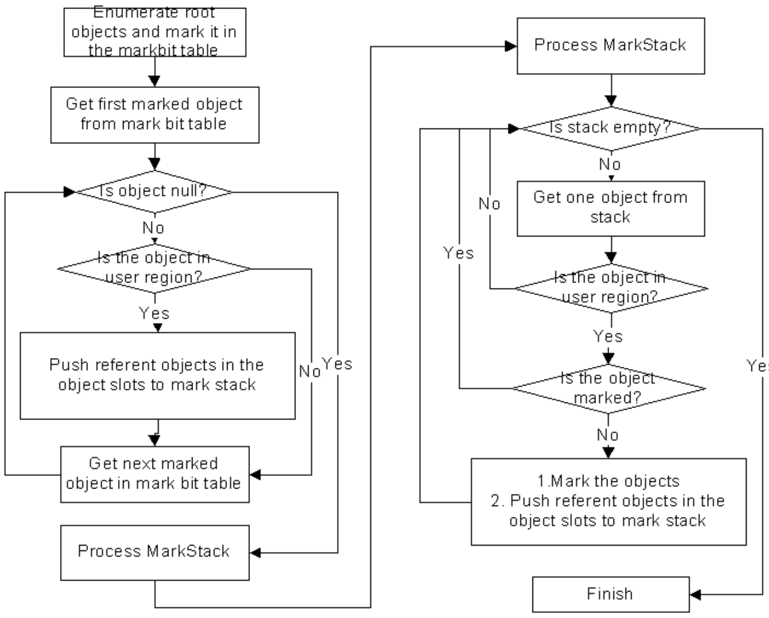
Enumerate root objects and mark it in the markbit table
Get first marked object from mark bit table
Is object null?
No
Is the object in user region?
Yes
Push referent objects in the object slots to mark stack
No
Yes
Get next marked object in mark bit table
Process MarkStack
Process MarkStack
Is stack empty?
No
Get one object from stack
No
Yes
Is the object in user region?
Yes
Is the object marked?
No
Yes
1.Mark the objects
2. Push referent objects in the object slots to mark stack
Finish

Fig. 4. Execution flow of the marking phase

The Sweeping Phase

The sweeping phase in the regional GC is similar to the traditional mark-sweep GC. The regional GC only recycles the free space in the user region. To find out all the dead objects in the user region, the GC performs "xor" operation on the bits of the mark-bit table and allocation-bit table. The results of the operation keep in the mark-bit table. All the dead objects are marked with 1. The GC iterates the mark-bit table in the area of the user region. All the dead objects are freed by passing them to dlmalloc free routine. The size of the free space is accumulated during the sweeping phase. If the free space is smaller than 20% of the total heap size, the next collection is scheduled to be a major collection.

4.3 The major collection

The regional GC needs the major collection to trace the entire heap from time to time to recycle the possible floating garbage retained by the cross-region references in the remembered set. The major collection in the regional GC has longer pause time than the minor collection since it traces both the class region and the user region. It also clears the remembered set before the collection, and rebuilds the set during the collection.

The major collection is expected to happen infrequently. It is triggered in two situations.

1. When the reserved space for the remembered set is full during the application execution, the next collection has to be a major collection to ensure the correctness;
2. When the size of the free space after a collection is smaller than 20% of the total heap size, the next collection is scheduled to be a major collection.

The Marking Phase

Similar to the marking phase of the minor collection, the application is paused during the major collection. The content of remembered set is discarded at the beginning of the major collection in order to trace both the class region and the user region. The live objects are identified by traversing from the root set references and marked in the mark-bit table.

During the marking process, all the references to the user region from external are recorded in the remembered set. The remembered set is prepared for the next collection, since a minor collection is the default to be scheduled for the next collection.

- Root set preparation

At the beginning of the major collection, the regional GC cleans the remembered set, including the field remembered set and the page remembered set. It then enumerates the entire root set references from the runtime stack and global variables. Those objects referenced in the root set are marked in the mark-bit table. The regional GC finishes this step with all the root objects marked in both the class region and the user region.

- Object graph traversal

Similar to the minor collection, the regional GC iterates the root objects in the mark-bit table from the beginning to the end, and traces the object graph in depth-first order. For each marked object in the mark-bit table, the GC scans its fields to find out the referenced objects and then scans those referenced objects recursively. If a live object in the class region has a field that contains a reference to the user region, the reference slot is recorded in the remembered set. The remembered set is prepared for the next collection.

The Sweeping Phase

The sweeping phase is the same as in the minor collection. The regional GC only sweeps the dead objects in the user region. It does not sweep the class region, because to recycle one dead object may trigger a copy-on-write for one page. The benefit is negative.

When the sweeping phase is done, the regional GC should reset the dirty pages in the class region to prepare for the next minor collection. For the page-protection write barrier, the GC protects the virtual memory in the class region to be read-only. For the PTE-scan write barrier, the system call "clear_dirty" is invoked to clear all the dirty bits in the page table entries.

5 Experimental Results

We evaluate the regional GC with Android Workload Suite (AWS) 1.0. AWS 1.0 is a client side workload suite including 15 workloads with 23 scenarios for 2D/3D graphics, media, browser, productivity and VM engine. The AWS 1.0 is not designed for GC evaluation but a comprehensive workload suite for Android platform evaluation. We choose AWS 1.0 intentionally so that the data are representative for the real usages. We use the page-protection write barrier by default in the evaluation of the regional GC.

5.1 Overall performance impact

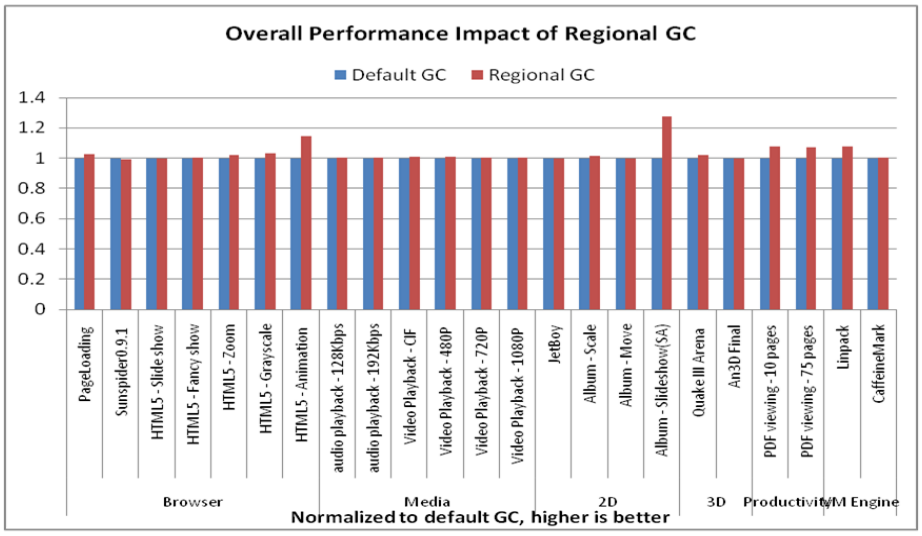

Fig. 5. Overall performance impact.

In figure 5, the overall performance impact of the regional GC is presented. Compared with the Froyo default GC, the regional GC has 3% performance gain in average across the 23 workloads/scenarios. As expected, performance of the object-intensive workloads has been improved significantly. For example, the graphic 2D workload Album Slideshow and the productivity workload PDF Viewing get 28% and 8% performance gains respectively. Meanwhile, the regional GC does not visibly impact the non-object-intensive workloads. For most of the browser and media workloads, the performance difference between the default GC and the regional GC is within ±1%.

5.2 Collection pause time reduction

For the smart phone consumers, the user experience is sometimes more important than the pure performance. The GC related user experience metrics include the average pause time of the collections, the maximal pause time and the number of pauses.

The average GC pause time is shown in Figure 6(A). The average GC pause time of the regional GC is reduced by 55% compared to that of the default GC. None of the workloads has bigger average pause time with the regional GC. This is reasonable since the regional GC omits the marking time for the class region in the minor collections. QuakeIII Arena does not have any collection happening.

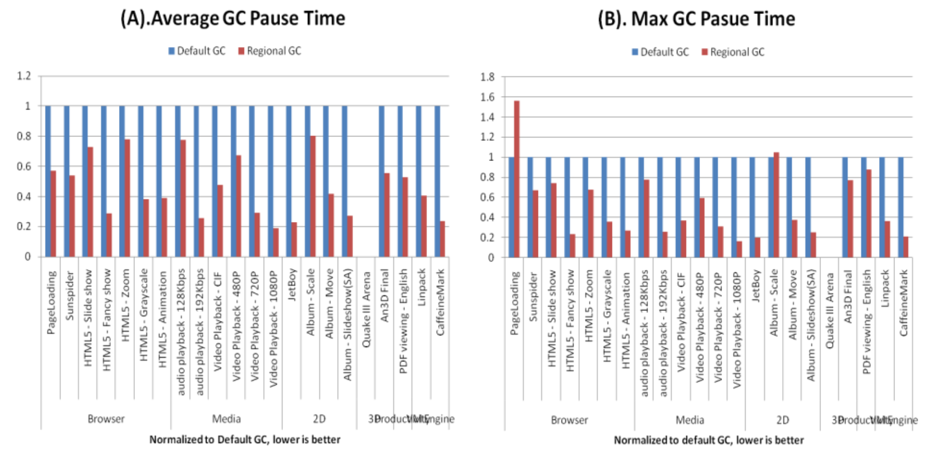

Fig. 6. GC pause time reduction

We show the maximal collection pause time for each applicaiton in Figure 6(B). The JetBoy and audio workloads has largely reduced maximal GC pause time significantly, because all the collections during these workloads execution are minor collection. The Pageloading and Album Scale workload has longer maximal GC pause time with the regional GC. We find that the longest pause time happens in major collections, and the marking time in them happens to be long due to application's execution dynamics.

5.3 Write barrier overhead

We implement the two alternative page write barriers in the regional GC: the PTE-scan write barrier and the page-protection write barrier. First, we measure the overhead of one page fault handling, including the signal processing, for the page-protection write barrier. We also measure the overhead of one round PTE scanning with different heap sizes. The data is given in Fig. 7(A).

The dash line is for the page fault processing overhead, and the dotted line is for the PTE scanning overhead, along with different heap sizes. From the chart, we can see that the overhead of one page fault handling is a constant with different heap sizes, while the overhead of the PTE scanning is proportional to the page table size, or the heap size. The two curves have an intersection point at heap size 3.5MB.

When a workload executes, the total PTE-scan write barrier overhead is roughly equal to the total collection counts multiplying the time of one round PTE scanning. The total overhead of the page-protection write barrier is roughly equal to the dirty page number multiplying the time of one page fault processing. We collect the dirty page number, collection count and average heap size in table 1.

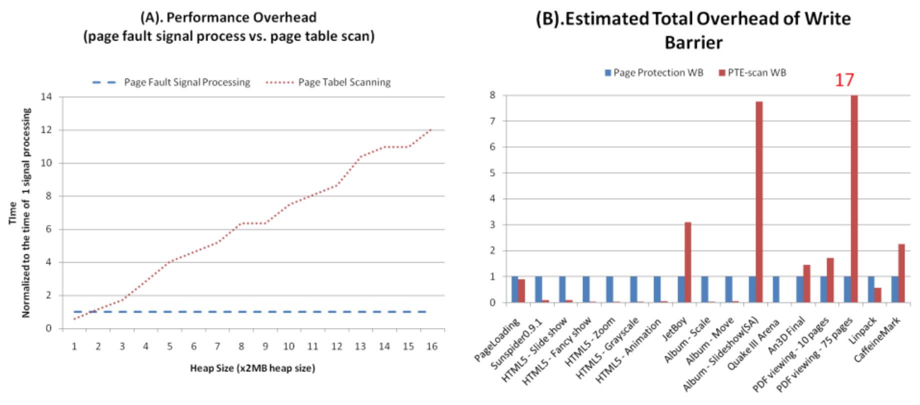

Fig. 7. Overhead of Write Barrier.

Workload	*Dirty Page Num*	*GC counts*	*Average Heap Size (MB)*	*Workload*	*Dirty Page Num*	*GC counts*	*Average Heap Size (MB)*
PageLoading	484	180	6.4	Album – Scale	66	2	3.2
Sunspider0.9.1	102	9	3	Album - Move	63	3	3
HTML5 - Slideshow	133	10	3	Album - Slideshow(SA)	65	500	2.7
HTML5 - Fancyshow	106	3	3	Quake III Arena	0	0	0
HTML5 - Zoom	85	2	3	An3D Final	83	45	7.2
HTML5 - Grayscale	109	2	3	PDF viewing - 10 pages	93	87	4.9
HTML5 - Animation	86	4	3	PDF viewing - 75 pages	93	724	5.8
JetBoy	63	2	2.7	Linpack	81	12	5.7
				CaffeineMark	44	89	3

Table 1. Workload statistics: dirty page number, collection counts and used heap size

Based on the data in Table 1, we compute the estimated write barrier overhead in Figure 7(B). The data are normalized to the page-protection write barrier overhead. We can see that the workloads have different preferences over the write barrier solution. The workloads such as the HTML5-related have very small PTE-scan write barrier overhead because they usually consume small heap size and seldom trigger the collections. For the workloads that have large heap size and frequent collections, the page-protection write barrier outperforms the PTE-scan write barrier.

We also measure the true runtime overhead of the write barriers. Figure 8(A) is the overall performance comparison between the two kinds of write barriers. We cannot see obvious difference between them. It is because the total runtime overhead of the write barrier is quite small with AWS 1.0, as shown in figure 8(B). Te page-protection write barrier overhead is within ±1% of the total execution time. Thus the performance impact of the different write barrier implementations is negligible.

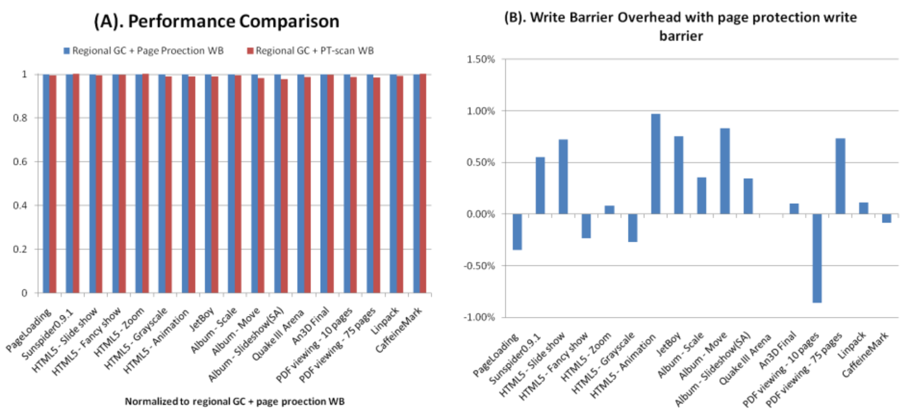

Fig. 8. Performance Comparison and Overhead

6 Conclusion

For smart phone software stack, the GC algorithm should consider both the pure performance and the pause time, in order to achieve best user experience. In this paper we present a Regional GC for Android that can effectively improve both application performance and user experience. The Regional GC we propose is similar to a generational GC but actually partitions the heap according to heap layout regions instead of the object ages.

A major collection of the regional GC is the same as DalvikVM's normal STW collection, while a minor collection only marks and sweeps one of the heap regions. In the evaluation with Android workload suite (AWS), the performance of 2D graphic workload Album Slideshow is improved by 28%, and its average pause time is reduced by 73%. The average pause time reduction across all the AWS applications is 55%. The regional GC can be combined with a concurrent GC to further reduce the pause time.

This paper also describes two alternative write barrier designs in the Regional GC. One is page-protection based, and the other is PTE-scan based. We evaluate the two approaches with the AWS applications, and find their overhead is not visibly impactful to the application performance.

As the next step, we are evaluating the applicability of the regional GC to a concurrent GC in Android, and further investigate the balance between the pure performance and user experience. The other work we are doing is to design an adaptive memory scheduling algorithm that can automatically balance the memory allocations among all the running applications.

References

1. John McCarthy. Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM. 3(4):184-195. April 1960
2. Marvin Minsky. A LISP garbage collector algorithm using serial secondary storage. A.I.Memo 58, MIT Project MAC, Cambridge, Massachusetts, 1963.
3. Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. Communications of the ACM, 26(6):419-429, June 1983.
4. David M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In ACM SIGSOFT/SIGPLAN Software Engineering Symposium on PSDE, pages 157-167. ACM Press, April 1984.
5. Robert A. Shaw. Empirical Analysis of a Lisp System. Ph.D thesis, Standford University, Palo Alto, California, Feb. 1988. Technical Report CSL-TR-88-351, Standford University Computer Systems Laboratory.
6. Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In Conference Record of the 20th ACM Symposium on Principles of Programming Languages, January 1993.
7. Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. SIGPLAN PLDI, 26(6):157-164, 1991.
8. Hezi Azatchi, Yossi Levanoni, Harel Paz and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In 18th Annual ACM SIGPLAN Conference on OPSLA’03, Aneheim, CA, 2003
9. Janice J. Heiss, The Multi-Tasking Virtual Machine: Building a Highly Scalable JVM, March 22, 2005. Available online at http://java.sun.com/developer/technicalArticles/Programming/mvm/.
10. Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In Proceedings of the Conference on OOPLSA, pages 23-35, New Orleans, Louisiana, Oct. 1989.
11. Patrick Sobalvarro. A Lifetime-based Garbage Collector for LISP system on General-Purpose Computers. Massachusetts Institute of Technology, Cambridge, MA, 1998.
12. Antony L. Hosking, Richard L. Hudson, “Remembered sets can also play cards”, In OOPSLA'93 Workshop on Garbage Collection and Memory Management, 1993.
13. Darko Stefanovic, Kathryn S. McKinley, J. Eliot B. Moss. Age-based garbage collection. In Proceedings of the Conference on OOPSLA, pages 370-381, N.Y. Nov. 1999.
14. Stephen M Blackburn, Richard Jones, Kathryn S. Mckinley, J Eliot B Moss. Beltway: Getting Around Garbage Collection Gridlock. In Proceedings of the ACM SIGPLAN 2002 Conference on PLDI, June 2002, Brelin, Germany.
15. David Detlefs, Christine Flood, Steve Heller, Tony Printezis. Garbage-First Garbage Collection. In Proceedings of the 4th ISMM, Oct. 2004. Vancouver, BC, Canada.