ON TRACING THE MEMORY BEHAVIOR OF DALVIK APPLICATIONS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Ahmed Hussein

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

December 2013

Purdue University

West Lafayette, Indiana

Dedicated to my family.

ACKNOWLEDGMENTS

I thank Christopher Vick for his valuable guidance. He inspired me greatly to pursue the work captured in this thesis.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

ADB	Android Debug Bridge
SDK	Software Development Kit
CMS	Concurrent Mark-Sweep
DB	DragonBoard
DDMS	Dalvik Debug Monitor Server
GC	Garbage Collection
GCD	Garbage Collector Daemon
GUI	Graphical User Interface
ICS	Ice Cream Sandwich
JIT	Just-In-Time Compiler
JNI	Java Native Interface
MS	Mark-Sweep
NDK	Native Development Kit
VM	Virtual Machine

ABSTRACT

Hussein, Ahmed M. M.S., Purdue University, December 2013. On Tracing the Memory Behavior of Dalvik Applications. Major Professor: Antony L. Hosking.

The Dalvik virtual machine hosts all user applications for the Android platform. Written in the Java programming language, the performance of these applications is critical to the user experience of Android. Understanding the behavior of applications running on Dalvik is central to diagnosing application bottlenecks and also to improving the support provided by Dalvik to representative workloads. To date, no infrastructure for Dalvik has allowed understanding of application behavior in the context of the Dalvik implementation. This thesis develops and applies a memory profiling framework for measuring the platform-independent memory behavior of applications running on Dalvik. Validation of the resulting profiling framework is achieved by porting standard Java benchmarks with known memory behaviors so that they can execute on Dalvik with substantially similar profiles. The profiling framework thus allows evaluation of industry-standard Android benchmarks to be done with confidence.

1 INTRODUCTION

The mobile telecommunications industry has grown rapidly over the last three decades. Mobile applications are increasingly important as applications migrate from desktops to smartphones and other mobile devices. The web-site *comScore*™ released a study in May of 2012, showing that more mobile subscribers used mobile apps than browsed the web on their devices: 51.1% vs. 49.8%, respectively [4].

Mobile applications are available through application-distribution platforms, operated by the owner of the mobile operating system, such as *Apple App Store*,™ *Google Play*,™ *Windows Phone Store*™ and *BlackBerry App World*™.

In this thesis we examine the behavior of mobile applications (commonly referred to as *apps*) targeted to Google's Android platform [9]. While Android apps are Java programs, they represent an application space quite distinct from that of standard Java platforms for enterprise and server applications. Thus, it is likely that they will exhibit behaviors somewhat different from previous standard Java workloads. Having a profiling infrastructure that natively handles apps targeting Android allows exploration of that space, as well as comparison against other Java platforms.

1.1 Objective

Our objective is to analyze and profile the memory behavior of Android apps, and to relate that behavior to the underlying Dalvik virtual machine (Dalvik) on which they run. To do this we must understand not only the nature of the apps themselves, and their memory demands, but also the internal details of Dalvik's memory management and how it meets those demands. This thesis addresses both of these components for understanding memory behavior of Android apps.

Table 1.1: Language breakdown in Android

Language	Code lines	Comment lines	Comment ratio	Percentage
C	4,714,358	1,543,234	24.7%	35.2%
C++	2,262,766	801,948	26.2%	17.4%
Java	1,391,395	983,138	41.4%	13.1%
All	14,986,373	3,644,452	24.318%	100%

1.2 Contribution

Our contribution is instrumentation of the Dalvik memory manager to extract platform-independent (i.e., independent of Dalvik) metrics from execution of arbitrary Android apps, including both the industry-standard Android benchmark Quadrant [17], and a selection of standard Java benchmarks drawn from the SPECjvm98 and DaCapo suites [2, 5] that we have ported to Android [9]. The latter serve as reference points for validation of our profiling infrastructure, since their behaviors have previously been studied extensively [2, 6].

Intrinsic app memory behavior may influence power consumption, memory footprint, and responsiveness, but the ways in which Dalvik services that behavior may be as big an influence on those metrics. The tools that we develop in this thesis will allow app behavior to be correlated with the underlying implementation strategies used in Dalvik, to diagnose pathologies in those strategies, and to inform innovations in Dalvik's implementation that can better serve the needs of apps.

1.3 Android and Dalvik

Android is a free development platform based on Linux and open source [11]. There are 32 languages used in the Android source code repository. Table 1.1 shows the top languages used to write the Android platform [7].

Dalvik is designed and written by Dan Bornstein to be suitable for systems that are constrained in terms of memory and processor speed. Dalvik compiles Java application code into machine-independent instructions similar to Java bytecodes, which are then executed by Dalvik on the mobile device.

The main reason for a non-standard bytecode format is the fact that Dalvik uses a register-based architecture instead of a stack-based one. Dalvik uses an instruction set that can directly access local variables as if in registers, in order to reduce overhead of loading and storing values from the stack. The compiled Android code format is `dex` (Dalvik Executable) files which are in turn zipped into a single `apk` (Android Application Package) file on the device.

A Dalvik tool called *dx* converts Java class files into `dex` files. Multiple classes can be included in a single `dex` file. During the conversion process, all the constants shared between classes are included once for the sake of space optimizations. As of Android 2.2, Dalvik has a *just-in-time* (JIT) compiler. The latter can modify the executable code for further optimizations. Methods frequently executed may be inlined to reduce call overheads. Dalvik allows multiple VM instances to run at the same time and takes advantage of the underlying operating system (Linux) for security and process isolation.

Dalvik applications are usually developed in Java language using the Android Software Development Kit (SDK). The latter works on Windows, Linux, and Mac OS X, creating an app that can be deployed on any Android device. Native Development Kit (NDK) is another tool used to write applications or extensions in `C` or `C++`. NDK is used to write and deploy native libraries (as described in the following section).

1.3.1 Android Architecture

The Android platform is built as a stack of various layers running on top of each other such that the lower-level layers provide services to upper-level layers [3]. Figure 1.1 shows android layers.

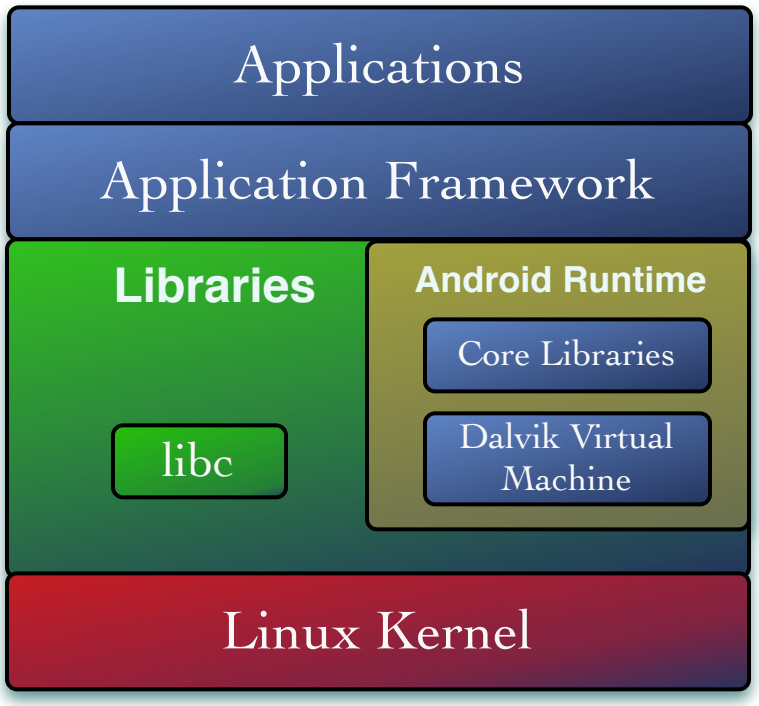

Figure 1.1.: Android architecture

Linux kernel: Android uses Linux for its operating system services such as networking, process management, and memory management. The Android kernel includes several additions and changes to the Linux kernel:

hardware support: the kernel has ARM architecture ports, and other necessary system code to support various Android devices;

shared memory: Android has a file-based shared memory system called *ashmem*, allowing processes that are not related by ancestry to share memory maps by name;

binder: a lightweight system for inter-process communication (IPC) and remote procedure call (RPC);

wakelocks: Android's default state is to sleep in order to reduce power consumption. Wakelocks prevent the device from going to sleep [8]; defining a wakelock of

type *IDLE* prevents the device from entering the low-power state, enabling it to be more responsive; defining a wakelock of type *SUSPEND* prevents the system from suspending; the currently defined wakelocks can be queried using the command `cat /proc/wakelocks`;

logger: a logging mechanism optimized for high-speed writes.

Native libraries: These are pre-installed by the phone vendor. The libraries represent the modules compiled down to native machine code to provide some of the common services needed by the apps. An example of these libraries are 2D/3D graphic libraries. The libraries are not standalone processes. Instead, they are called by higher-level programs.

Android runtime: The Android runtime includes Dalvik and the core Java libraries. The core Java libraries that come with Android are different from Java Mobile Edition libraries. We show the difference between the core libraries later in Section 4.2. Applications run as separate processes within the Dalvik kernel. Each application runs in its own instance of the Android run-time system, and the core of each instance is a Dalvik VM.

Application framework: The application framework provides services to apps. The most important parts of the framework are as follows:

activity manager: manages the app life cycle;

content provider: provides encapsulation of data that needs to be shared between applications;

notification manager: handles events such as arriving messages and alerts.

Application: The highest layer in the Android architecture is the application. An app represents a program with a distinct system identity that runs on the device and interacts with the user. Each app has a stack of components: Activities, Broadcast-Receivers, Services, and Content-Providers. Security features are provided through a *permis-*

sion mechanism that enforces restrictions on the specific operations that a particular process can perform [10].

1.4 Thesis Organization

We have only briefly described some of the main characteristics of Android and Dalvik. Further details appear in later chapters. In Chapter 2 we describe memory management in Dalvik. Chapter 3 defines the set of metrics we use to analyze the Dalvik memory behavior, followed by a description of our implementation methodology.

Chapter 4 lists the main characteristics of the apps and benchmarks used to validate the memory profiler. The description will be accompanied by details of the procedure we followed to port standard Java benchmarks to Android, along with some of the main differences between standard Java's core libraries and those of Android.

The specifications of the environment used to evaluate the profiler is described in Chapter 5, which also includes the results generated from our memory profiler for the Quadrant standard benchmark for Dalvik. Finally, we give conclusions and suggest directions for future work in Chapter 6.

2 DALVIK MEMORY MANAGEMENT

Android relies on *Linux-permissions* to achieve security; it runs each VM instance in a separate process. In order to reduce the overhead of this separation model, Dalvik shares common classes and objects across running VMs. Each application shares one master copy of all the read-only portions of the VM, using `copy-on-write`. As a result, Android can run more programs in a tightly-constrained memory environment. The first VM to start, called the *Zygote*, is the process responsible for loading the common objects into memory. In this chapter, we describe in detail the memory initialization steps and the garbage-collection implementation in Dalvik.

2.1 Garbage-Collection Implementation

Dalvik uses a *Concurrent Mark-and-Sweep* (CMS) algorithm [15] to collect dead objects. The following sections detail the internal data structures used to manage the live objects and the advantages and the short-comings of the current implementation.

2.1.1 Mark-and-Sweep

Mark-and-sweep (MS) was the first garbage collection algorithm to be developed that is able to reclaim cyclic data structures [15]. MS is categorized as a *tracing garbage collector* because it traces out the accessible objects in the program, called *reachable-objects* (whether in a direct or indirect way).

MS maintains a set of locations in heap that are not used by any objects. The allocation requests can be then satisfied by finding the appropriate block of free memory. When an allocation cannot be satisfied MS stops all threads in order to identify which objects are no longer reachable.

```
for each root variable r
  mark(r);
sweep();
```

Listing 2.1: Mark-Sweep algorithm

For the purpose of identifying reachable objects, *roots* are defined to be the set of local variables on the stack plus the static variables. The roots represent the starting points from which the algorithm traces the objects. All objects referenced by the roots are considered *directly* accessible by the program. On the other hand, an object is indirectly accessible if it is referenced by a field in some other (directly or indirectly) reachable object.

By definition, all reachable objects are alive. Otherwise, the object is considered garbage. MS consists of two phases as shown in Listing 2.1:

1. *mark*: finds and marks all accessible objects.

2. *sweep*: scans through the heap and reclaims unmarked objects.

In order to distinguish the live objects from garbage, the object status (mark/unmarked) is stored in some sort of data structure like tables. By default, all objects are unmarked when they are created. Thus, the marked field/entry is initially false.

An object `p` and all the object fields (indirectly accessible) can be marked by using the recursive mark method shown in Listing 2.2. During `sweep` shown in Listing 2.2, MS scans through all the objects in the heap in order to locate all the unmarked objects. The memory space filled by the unmarked objects is reclaimed during the sweep. Finally, it resets the mark field of every alive object to set it ready for the next cycle.

MS is guaranteed to terminate as the marked objects are not reinserted back into the unprocessed queues. `sweep` does not unmark an object until it is finished scanning.

MS has two major short-comings. Allocating an object of length `n` requires finding a contiguous set of memory locations to satisfy the request. The search is not guaranteed to succeed even if the free memory in the heap contains the required amount of free space due to the risk of fragmentation. The second short-coming known in MS is the suspension of

```
void mark(Object p)
    if (!p.marked)
        p.marked = true;
        for each Object q referenced by p
            mark(q);
void sweep()
    for each Object p in the heap
        if(p.marked)
            p.marked = false;
        else
            heap.release(p);
```

Listing 2.2: Mark and Sweep procedures

the running threads, known as *Stop-the-world*. Execution resumes only after the unmarked objects are reclaimed. Dalvik does not address the fragmentation short-coming. Instead, Dalvik allows running a *concurrent* collector to reduce pauses due to the suspension of the mutators.

2.1.2 Concurrent Mark-and-Sweep

CMS does not allow mutators to run until memory is exhausted [12]. As the mutators allocate, concurrent collectors can run concurrently in order to reduce the overhead of the pause time incurred from the suspension. The collector suspends the mutators at the beginning of the collection cycle to scan the roots. Having scanned the roots, the mutators can be resumed while the collector is running the mark phase. For the purpose of allowing mutators to run concurrently during the mark phase, a write barrier is used to keep track of the modified objects O. After the mark phase, the collector suspends the mutators one more time to visit the set of the modified objects O.

Finally, the collector resumes the mutators during the sweep phase. CMS acquires its main advantage from reducing the overall suspension time. In the context of smartphones apps, the pause time reduction leads to a faster response time to user interaction. Hence, CMS improves the user experience. Listing 2.3 shows the CMS implementation in Dalvik.

2.1.3 Memory Initialization

Constant values used by the VM are hardcoded in `system/build.prop`. By default, any constant defined in that file will override the default hardcoded values in the source code. An instant of Dalvik can run as a standalone process. In the latter case, the value can be passed as a command line argument.

The initial heap size is specified as `gDvm.heapStartingSize`. The maximum heap size is represented by `gDvm.heapMaximumSize`. `dvmHeapStartup` sets the

```
Object allocate(size)
  ref = new(size);
  add(ref, workList);  /* initializes the object to be marked */
  return ref;

void shade(ref)
  if (! isMarked(ref))
    setMark(ref);
    /* workList used to keep track of all the objects */
    add(ref, workList);

void scan(ref)
  for each field f in pointers(ref)
    if (*f != NULL)
      shade(*f);

void collect()
  suspendThreads();
  scanRoots();
  resumeThreads();
  mark();
  suspend_threads();
   /* revisit the references modified by the mutators */
  mark();
  resumeThreads();
  sweep();
```

Listing 2.3: Concurrent Mark and Sweep procedures in Dalvik

limit the heap is allowed to grow by setting the value of `gDvm.heapMaximumSize` to `gDvm.heapGrowthLimit`.

Dalvik has three main entities:

- `GCHeap`: holds all the tables used by the algorithm. Such tables are `card-table` and `weak-references`.

- `HeapSource`: holds an array of heaps. The active heap will be the first entry (zero index) while all the other heaps are kept in higher indexes. `HeapSource` also holds the `softLimit` used as a threshold against the current allocated bytes. Whenever the allocated bytes reaches the `softLimit` the allocation will fail to allow the heap growth. To indicate whether the heapSource instance was created within a zygote process, the field `sawZygote` is set to true. Finally, the field `targetUtilization` is used to keep the ideal utilization of the heap.

- `Heap`: holds a reference to `mspace` along with some information about the heap instance like the start address, size, allocated bytes, and the limit. `mspace` used in Dalvik is a *dlmalloc* implementation [13]. Memory in `dlmalloc` is allocated as *chunks* with `4-bytes` overhead for each chunk in order to maintain necessary metadata. Dalvik aligns the objects on `8-bytes` basis, which causes internal fragmentation when the actual size is not a multiple of eight. Unallocated chunks also store pointers to other free chunks in the usable space area.

Once the device starts, `dvmStartup` manages the initialization of Dalvik. The initialization will include GC, system threads, class loading and Java Native Interface (`JNI`) initializations. Class-loading and initial memory operations will be executed during the first instance of the first VM known as the *zygote*. The latter method calls `dvmGCStartup` which is responsible for doing all the memory manager initializations. It starts by allocating a contiguous region from Android shared memory. Once it succeeds in reserving the desired amount of memory, the allocated region will be passed to create `mspace` to manage the allocation/deallocation of objects through `dlmalloc`.

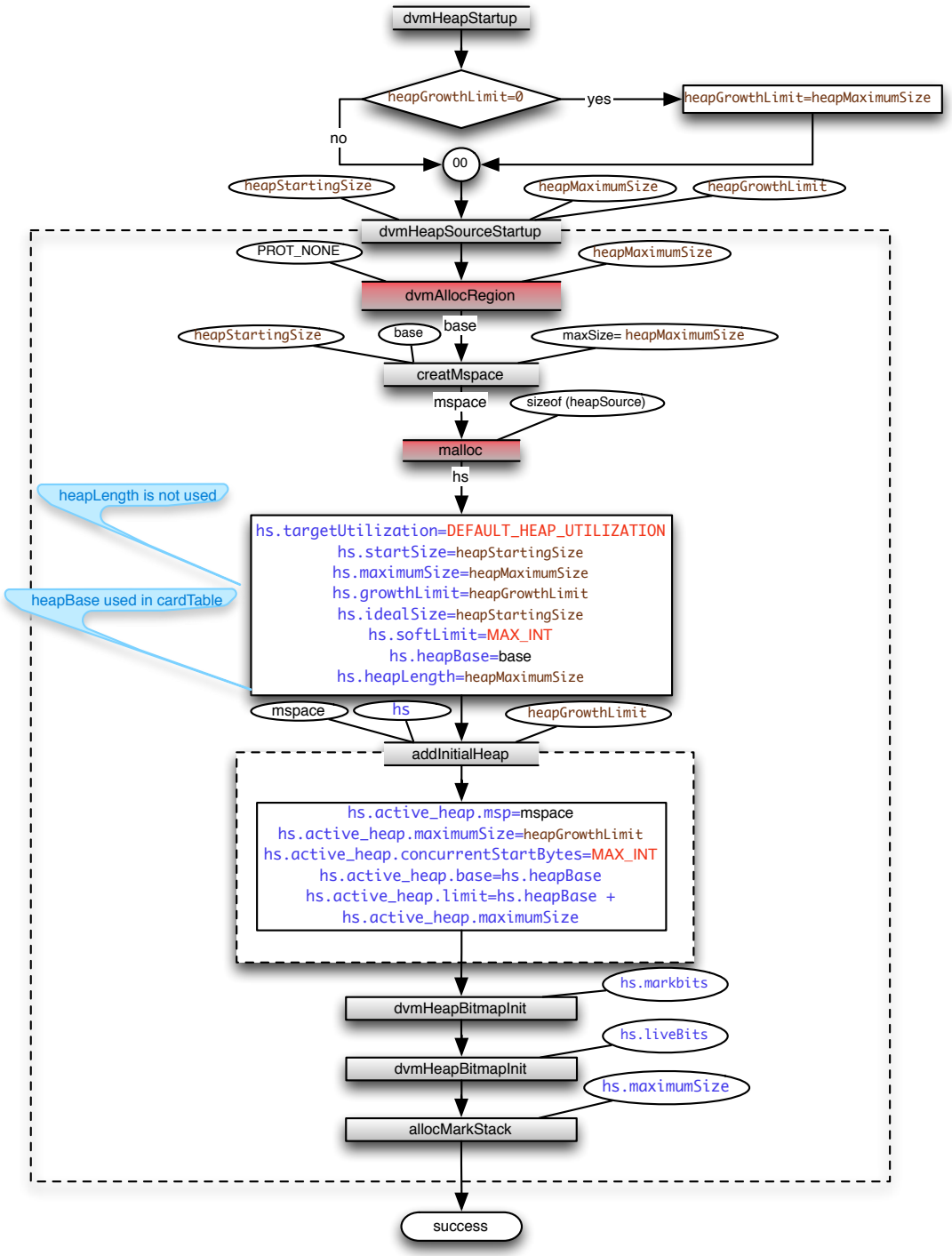

Figure 2.1.: Dalvik memory startup

During the initialization phase, the active heap is created in `addInitialHeap` which adds a new heap instance and copy it into the `heapSource-array`. Figure 2.1 shows the details of heap startup. Dalvik does not keep the GC-per-object metadata in the object header. Instead, it keeps tables to store all the marking-status, and the addresses of live data. There are three tables initialized from Android shared heap during the heap initialization:

1. `dalvik-bitmap1`: represents live-bits to identify the objects alive that should not be collected by a collector running concurrently with the mutator.

2. `dalvik-bitmap2`: represents the mark-bits to keep track of the marked objects during the mark-phase.

3. `dvmMarkStack`: used to keep track of the objects being marked while scanning their object fields.

4. `cardTable`: represents the data structure to keep track of the dirty objects.

Figure 2.2 shows the memory-layout after the heap's initialization is completed.

2.1.4 Memory-layout Overhead

Dalvik allocates objects with an `8-bytes` alignment. This is the minimum size allowed for an object. An empty object will have a reference to its class and another field (of four bytes) to hold the object's synchronization information. Both live-bits and mark-bits have one bit for every possible object in the heap. During the allocation process, `dvmHeapSourceAlloc` sets the live-bit of the object, while the mark-bit will be set during the marking phase.

The following equations calculate the memory overhead given an initial maximum size (`gDvm.heapMaximumSize`) of value `B` bytes. We can calculate the maximum number of objects allocated in the heap by assuming that all objects are of size `8-bytes` each. This leads to the worst case scenario shown in equation 2.1.

$$\text{objects} = B/8 \tag{2.1}$$

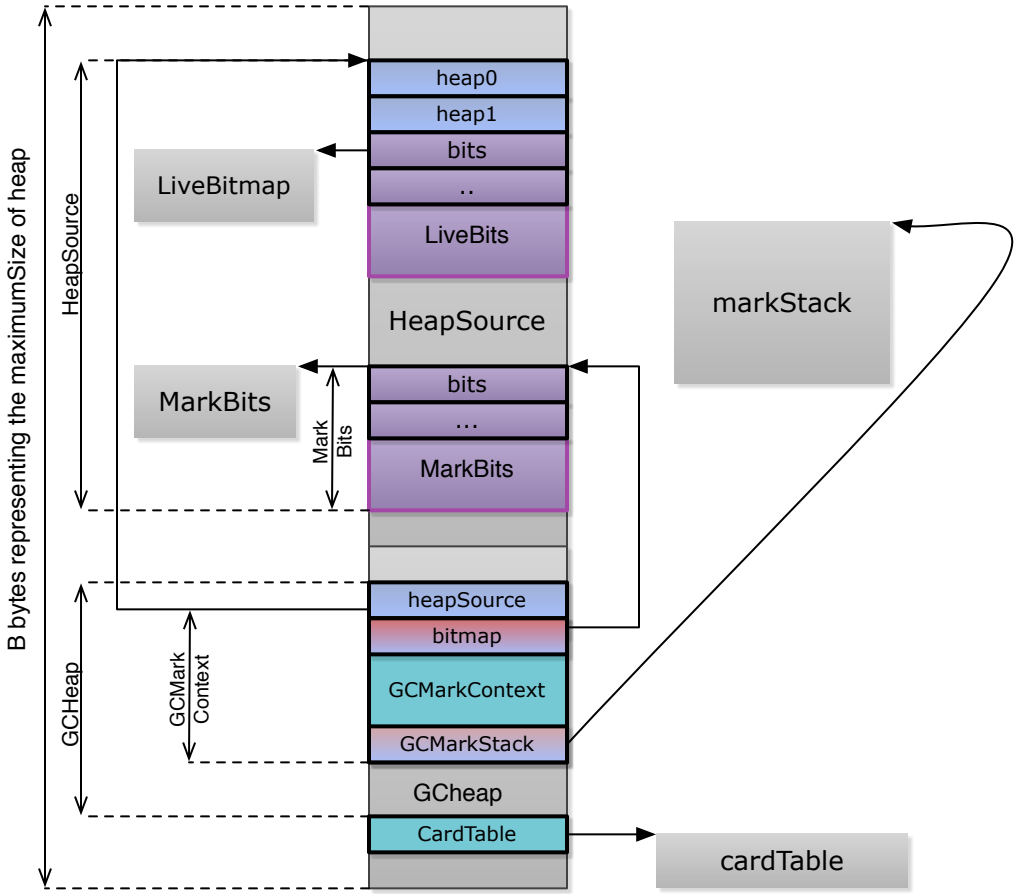

Figure 2.2.: Dalvik memory layout after initialization

The maximum number of objects allocated is used to derive the maximum size utilized by the bitmaps. Each `32`-bit word in a bitmap will cover `256` bytes of heap objects. This leads to accurate calculation of the total size required for each bitmap to cover the heap. Equation 2.2 shows this value with reference to the heap size `B`. Since Dalvik has two bitmaps (live/mark), the overhead will be twice the value calculated from equation 2.2.

$$\text{bitmaps} = B/64 \tag{2.2}$$

`markStack` is allocated from the shared memory and it should be capable of holding a number of references equivalent to the the maximum number of objects. As mentioned earlier, `dlmalloc` adds four bytes (`32-bit` architecture) for every allocated chunk (sin-

gle object in Dalvik's scope). The extra overhead is defined as a global variable called `HEAP_SOURCE_CHUNK_OVERHEAD`.

Dalvik takes into consideration the overhead introduced by `dlmalloc` when it initializes the `markStack`. As a result of the latter assumption, the minimum object size is `12` bits (without considering the 8-bytes alignment). Each entry in `marStack` holds a reference to an object in the heap which requires four bytes as we are limited to `32` bytes architecture for now. This leads to the value calculated by equation 2.3

$$\begin{aligned} \text{objects} &= B/(8+4) = B/12 \\ \text{markStack} &= (objects * 4)/12 \\ &= B/3 \end{aligned} \tag{2.3}$$

Finally, we need to measure the `cardTable` size. The initial calculated size takes into consideration the `8-bytes` alignment. Adding the alignment constraint to the minimum object size used in equation 2.3, we get minimum size of `16` bytes. Each object can be represented by a single bit in the `cardTable`. Hence, one bit is required to map two heap-bytes. Equation 2.4 calculates the value used to initialize the `cardTable`.

$$\text{cardTable} = B/128 \tag{2.4}$$

Putting it all together, the total overhead is the sum of all the heap components.

$$\text{cardTable} + \text{markStack} + \text{bitmaps} = 0.372 * B \tag{2.5}$$

In other words, a heap size of maximum size `B` bytes can request up to the value calculated in equation 2.6.

$$\text{heapSize} + \text{cardTable} + \text{markStack} + \text{bitmaps} = \lceil 1.372 * B \rceil \tag{2.6}$$

Assuming a heap of maximum size `64MB`, Table 2.1 shows the maximum size of each separate component in bytes. Since Dalvik is optimized for very strict memory constraints, a single VM does not map the pages unless they are used. This will guarantee that the memory pages allocated to each VM satisfies the physical needs and it will permit a larger number of VMs to be initialized concurrently. For example, the bitmaps are initialized to cover the allocated objects but not the entire heap.

Table 2.1: Memory overhead of Dalvik heap

Memory-Type	Size(Bytes)
Dalvik Heap	67,108,864
Card Table	524,288
Live Bits	1,048,576
Mark Bits	1,048,576
Mark Stack	22,369,622
Total	92,099,926

Multiple Heap Initializations

Before Dalvik calls `fork` for the first time, it finalizes the initialization steps by creating a new heap, then it sets the `zygote-heap` to be inactive. Since, all the allocation requests have to use the active heap (only a single heap is to be active at any point), the newly created heap will be the place from which all the objects are allocated. Dalvik copies zygote information from `heaps[0]` to the next entry `heaps[1]` to keep the active heap in entry zero at all times. Figure 2.3 shows the steps in details.

The advantage of using the `zygote` is to share the common objects/classes between VM instances. The forked VMs simply read the objects saved in `zygote-heap` without the necessary overhead required for initialization. The memory pages containing the shared objects are `copy-on-write` to guarantee that any one VM does not affect initialization of the other VMs.

Finally, Dalvik updates the `zygote` metadata by setting the maximum heap size to the current actual memory space used by the allocated objects. The remaining memory space (difference between initial maximum heap and the current heap size) will be used to initialize the newly allocated heap.

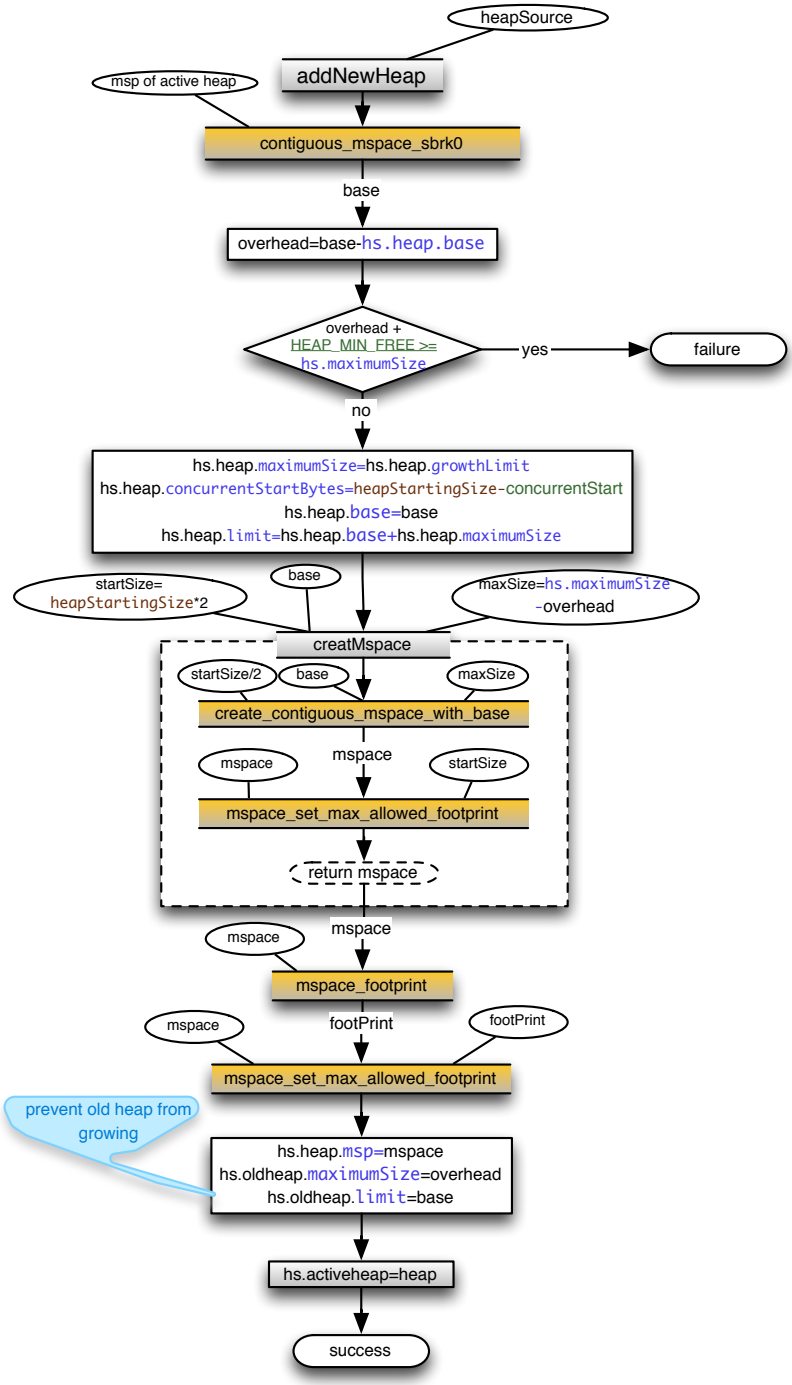

Figure 2.3.: Adding new heap

2.2 Garbage Collection Cycle

Dalvik defines different specifications for GC. Each specification is designed to fit an allocation scenario as detailed below. As we mentioned earlier, by default Dalvik starts with CMS. When CMS is disabled, Dalvik will run a stop-the-world MS collector. The GC configurations inside Dalvik are:

- `isPartial`: used to decide whether the collection cycle should consider all the heaps or not. When the value is set to true, the collector collects all the heaps except the one having limits less than the immune objects (objects created in `zygote` mode). When it is set to false, the GC will collect objects from the active heap.

- `isConcurrent`: defines whether the collector thread/mutator can run concurrently without stopping the world.

- `doPreserve`: preserves `softReferences`. The flag is set to false when Dalvik is doing an aggressive collection. Normal concurrent GC cycles will set this flag to true.

Figure 2.4 shows the object allocation procedure. The GC triggered in each case depends on the context of the execution in which the collector can act more or less aggressively. Collection modes are defined as:

- `GC_FOR_MALLOC`: does a partial collection, excluding the immune objects.[1] It is used by the mutator when the latter fails to allocate an object. The mutator puts the collector hat, checking whether a concurrent collection is already running and finally it suspends all the threads.

- `GC_CONCURRENT`: executed by GCD . During allocation, if the allocated bytes exceeds `concurrentStartBytes`, the mutator signals GCD to start a collection cycle. If the daemon times out, it trims pages from the heap and returns them back to the system. This collection is partial, concurrent and preserves soft-references.

[1] objects expected to stay alive in the VM

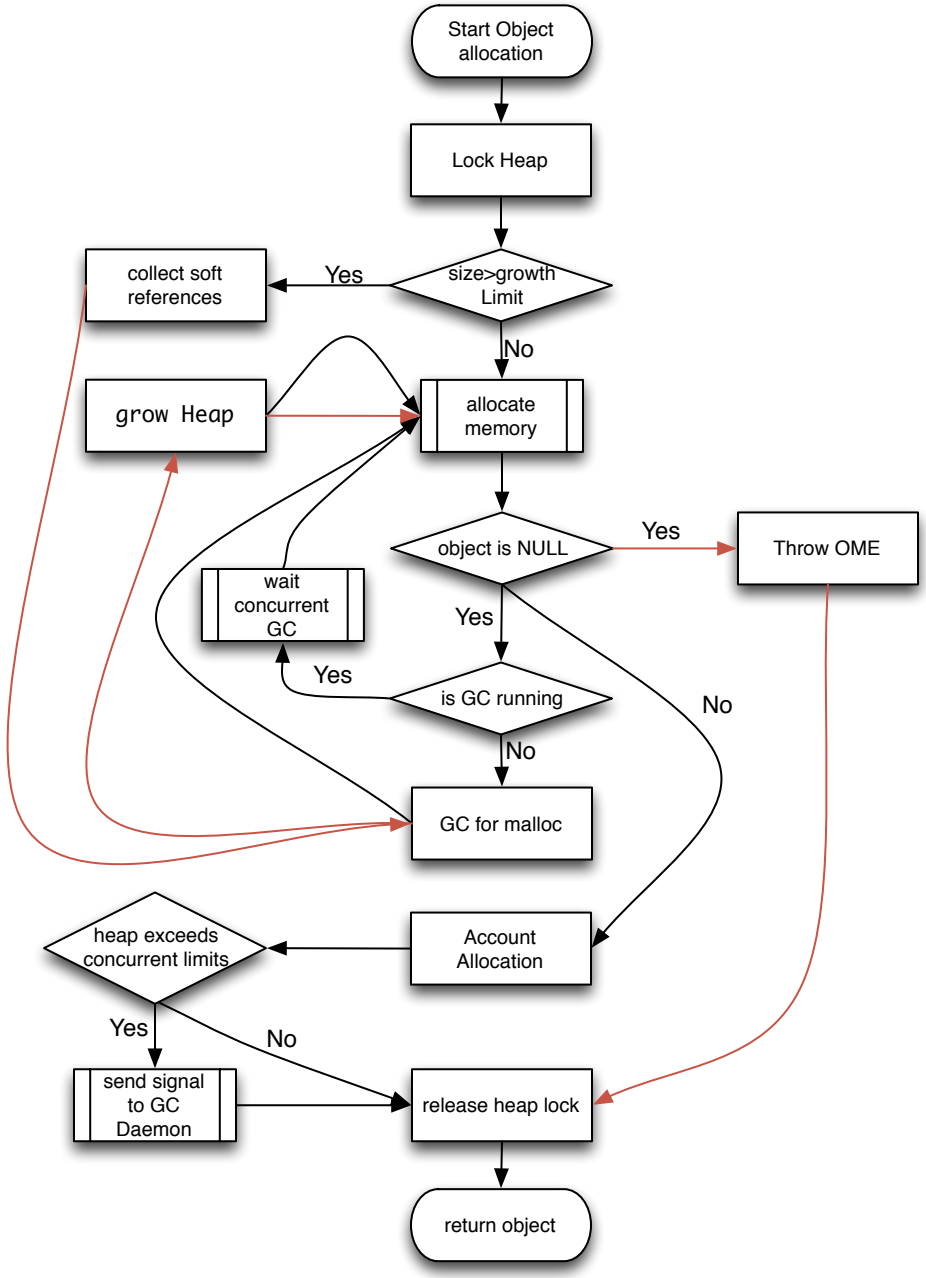

Figure 2.4.: Dalvik allocation flow

- `GC_EXPLICIT`: runs as a result of an explicit call from Java code. It is not partial, it is concurrent and it preserves soft-references.

- `GC_BEFORE_OOM`: is used before an out-of-memory exception. This scenario will be triggered when the memory is exhausted and all the collections fail to satisfy the size required.

2.2.1 Root Marking

The collection cycle starts by suspending all the threads to mark the roots. Roots are defined to be the set of all reachable objects from the global VM structure (`gDvm` structure). This holds the global data including:

- `loadedClasses`: hashed by class name, allocated in GC space.
- `primitiveClasses`: set of primitive types recognized by the system.
- `dbgRegistry`: registry of objects known to the debugger.
- `jniGlobalRefTable`: JNI global reference table.
- `literalStrings`: hash table of strings interned by the class loader.
- `jniPinRefTable`: JNI pinned object table (used for primitive arrays).
- `outOfMemoryObj`: preallocated throwable object for exception.
- `internalErrorObj`: preallocated throwable object for exception.
- `noClassDefFoundErrorObj`: preallocated throwable object for exception.

After visiting these tables, the marking thread visits all the thread stacks to identify the local variables and the objects directly accessible from each thread. Finally, The marker adds the internal tables (local monitor JNI threads) to the set.

2.2.2 Heap Synchronization

The heap is shared between all the threads in a single VM instance. This requires a mechanism between threads including mutators and collector threads. Each heap has a

lock to allow exclusive access by one single thread at a time. Any thread allocating or freeing must have exclusive access by acquiring the heap lock first.

As shown in Figure 2.4, each thread including GCD must acquire the `heapLock` before accessing the heap. In the case of the GCD , it needs to keep the lock while an atomic collection subtask is done.

A collection cycle starts from internal method `dvmCollectGarbageInternal` which assumed that the `heapLock` is already locked. The thread acting as the collector is of the following two types:

- mutator: when the allocation fails, the mutator acts as the collector; as a consequence, the the mutator will observe a longer allocation time. If some other thread is running concurrent GC, the mutator must wait until the collection cycle is completed.
- GCD : thread created for concurrent collection.

During an exclusive lock of the heap, threads cannot access the heap to allocate objects which affects the overall performance.

2.2.3 Controlling Garbage Collection and Heap Growth

The overhead introduced by frequent collection can be overwhelming. Mutators will be suspended frequently by the collector and they will remain suspended while their roots are scanned. The collection cycle will be completed with another round of suspension to finish marking the objects.

On the other hand, collecting less frequently leads to memory exhaustion. In this case, the amount of work done by the collector is larger and the mutators will freeze as long as the collector is still running (shifting to a stop-the-world collection). Keeping in mind the memory constraints the system was designed for, increasing the collection window will increase the heap size used by the VMs. This implies that the total pages requested from the operating system increases which may cause the system to crash in case the physical memory does not fit all these pages.

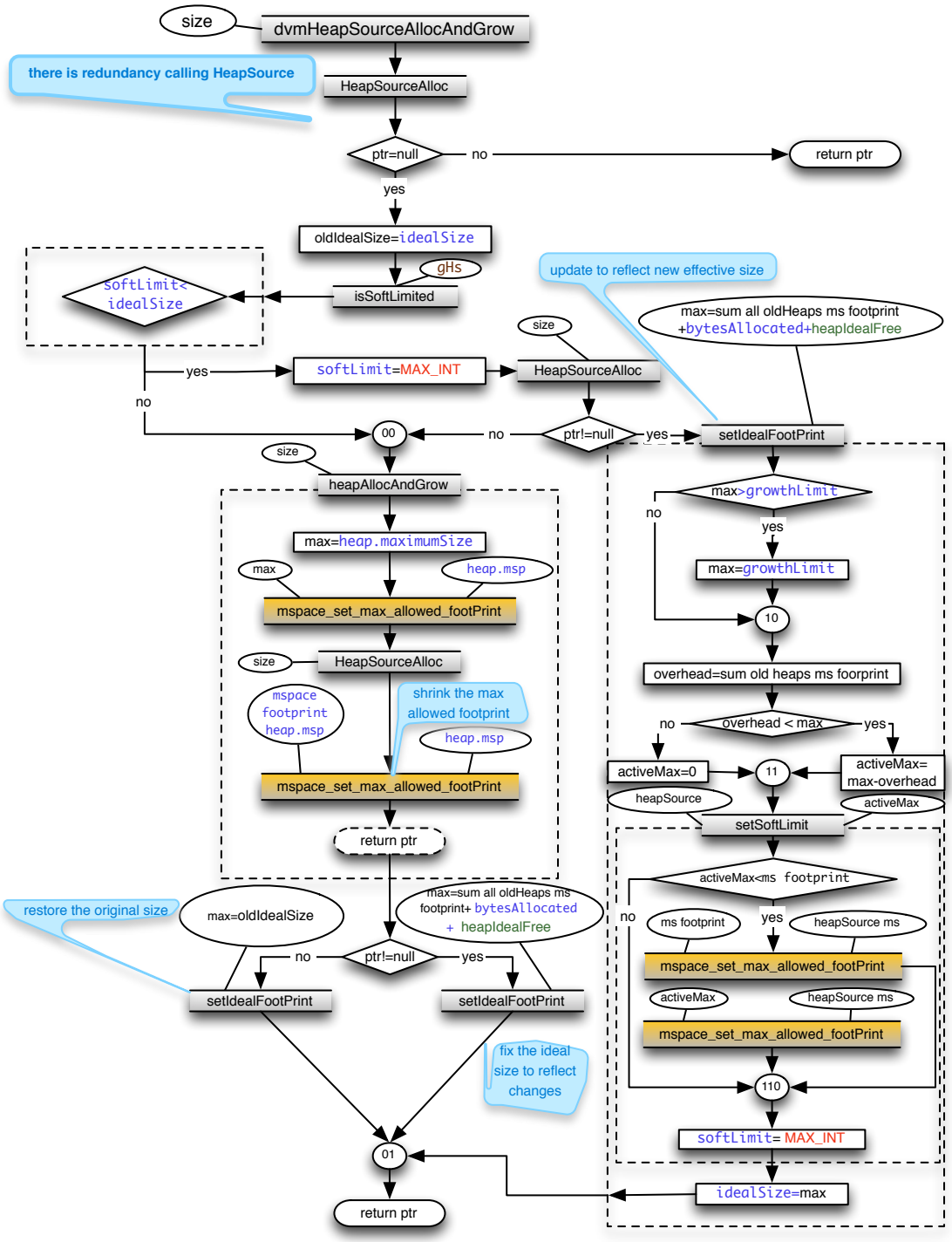

Figure 2.5.: Heap growth in Dalvik

CMS is used mainly to avoid such situations. In Dalvik running CMS, the mutator thread will check the available free bytes in a heap after every single collection. If the free bytes value falls below a threshold (defined to `50%` of heap-size), the mutator will signal GCD which in turn starts a collection cycle. In order to trigger a concurrent collection there is a set of hardcoded thresholds defined as:

- `HEAP_IDEAL_FREE`: represents the ideal number of free bytes in the heap. It is set to `2MB`.
- `HEAP_MIN_FREE`: used as a threshold for the current available free bytes. If the available bytes are below that threshold, the next collection will be stop-the-world. Otherwise, collection will remain concurrent. The initial value for that threshold is set to `HEAP_IDEAL_FREE/4`.
- `CONCURRENT_START`: used as a threshold to trigger a concurrent GC. Free memory below this value causes a GC signal. The default value is `128K`.
- `CONCURRENT_MIN_FREE`: used to decide whether the next collection should be concurrent or not. It is initialized to `(CONCURRENT_START + 128K)`.

Figure 2.5 shows the procedure to grow the heap size. Each heap structure holds a field called `concurrentStartByte`. In the case of `zygote`, this field will be initialized to `MAX_SIZE` (maximum value stored by four bytes). In newly allocated heaps, the same field is initialized to the difference between the minimum free bytes allowed and the concurrent start `(HEAP_MIN_FREE - CONCURRENT_START)`. In the post-allocation phase, the mutator will check whether the total number of allocated bytes reaches the value in `concurrentStartByte`. If true, the mutator will signal the GCD to start a new collection.

After a full MS, the collecting thread (mutator/ GCD) adjusts the heap size by calling `dvmHeapSourceGrowForUtilization`. This updates the foot-print to match the new target utilization ratio, then calculates the available free-bytes (`freeBytes`) in `mspace`, and finally updates `concurrentStartByte` as shown in Listing 2.4.

```
if (freeBytes < CONCURRENT_MIN_FREE) {
  /* Not enough free memory to allow a concurrent GC */
  heap->concurrentStartBytes = SIZE_MAX;
} else {
  heap->concurrentStartBytes = freeBytes - CONCURRENT_START;
}
```

Listing 2.4: Mark-Sweep algorithm

Setting a hardcoded threshold inside Dalvik regardless of the VM activity causes the GC to be triggered very frequently. We now report a common behavior on applications running on Android. Concurrent GC was triggered very frequently in many circumstances. When the VM compacted heap size is close to the threshold, any allocation request will cause the GCD to be signaled. In such a scenario, GCD is collecting a small percentage of the heap size (`1-2%`). Any subsequent allocation will trigger another cycle, and so on. This behavior will continue as long as the compacted heap is equal to the collection threshold. An ideal solution for such scenarios is to dynamically decide the threshold to trigger the GC.

GCD waits for a signal to start a CMS cycle. The amount of time it waits is hard-coded at five seconds, defined as `HEAP_TRIM_IDLE_TIME_MS (5*1000)`. When GCD times-out, it acquires the heap-lock and starts reclaiming free pages to return them to the system. Figure 2.6 shows the steps executed in a full MS cycle. The red arrows represents the control-flow executed by GCD . The collection is executed by calling `dvmCollectGarbageInternal` (defined by the dotted frame in the graph). The latter method takes the garbage specifications as arguments.

Trimming is hardcoded without any dynamic consideration of the application behavior. Pages can be trimmed from the heap despite the fact that shortly a mutator will request pages to allocate more objects. In that case, the work done by the trimmer thread is in vain and the collector thread will do more work growing the heap to fit the memory utilization.

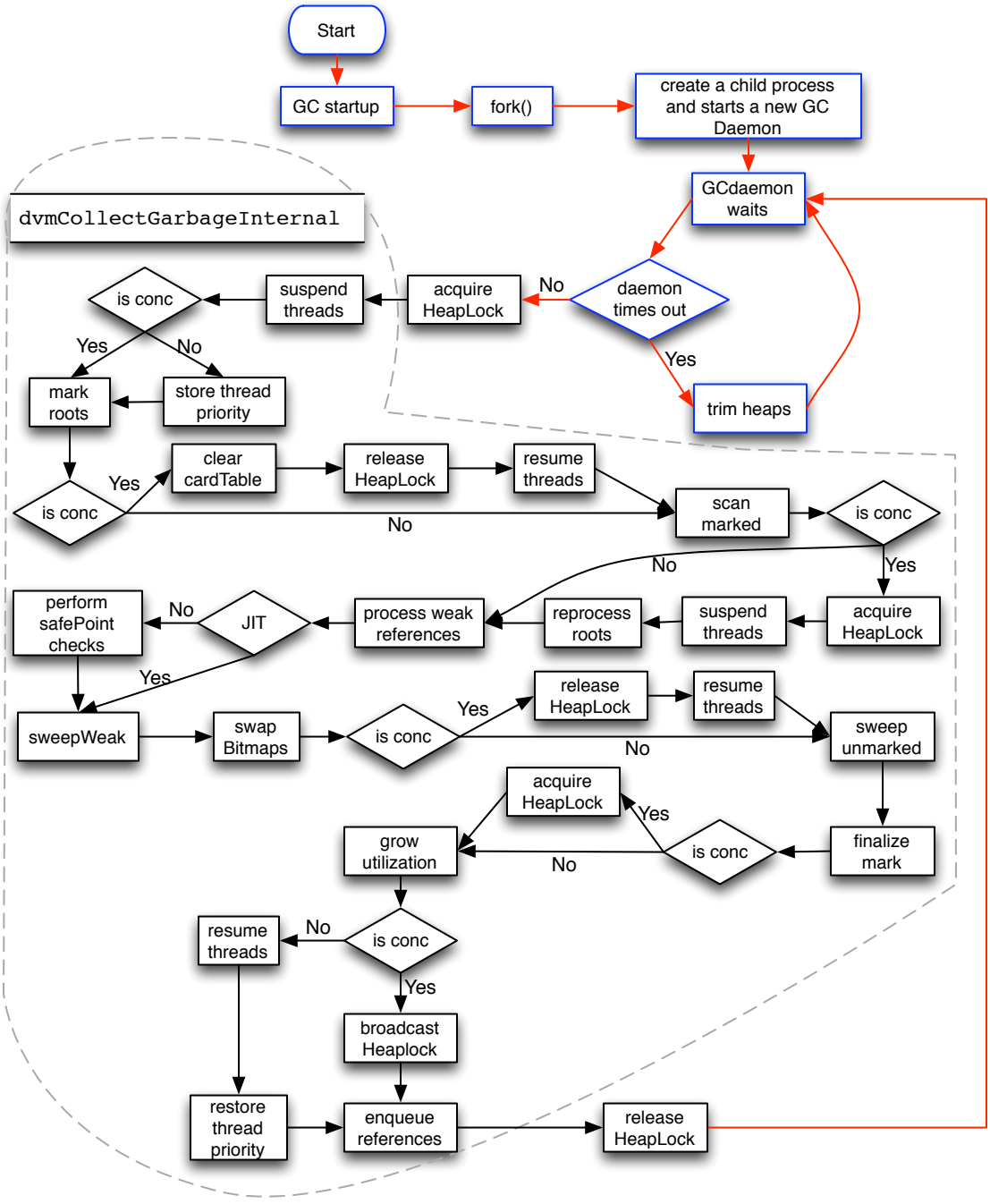

Figure 2.6.: Garbage collection steps

3 METRICS AND METHODOLOGY

In chapter 2, we explained the memory layer in Dalvik. In this chapter, we define a set of metrics to analyze application behavior while running on Dalvik. We emphasize the methodology and the implementation adapted to generate the measurements.

3.1 Profiling Scopes

Given the nature of multithreaded programs running on embedded systems nowadays, one must study the system from different perspectives. Providing information assuming a single threaded or a uniprocessor environment is insufficient for comprehensive system analysis. In order to satisfy this need, our system measures the same metric from different scopes: thread, object type, and object age.

In all the defined scopes, we maintain the illusion of a perfectly compacted heap. This makes the values generated from our system independent from the garbage collection specifications. In order to provide a generic analysis, the profiler ignores the overheads defined earlier in chapter 2. The profiler defines global counters for a single VM instance. The global counters are used to express the global status of the VM. Such status can be represented by the total allocated bytes and some other counters which will be explained in details in the following sections.

In order to provide a status for each monitored unit such thread or object type, we keep a set of global counters used a container for all the measurements generated for a given unit.

3.2 Metrics and Profiler Implementation

The profiling system has two main components:

- dalvik: is the implementation on top of Dalvik responsible of gathering the defined metrics and responsible of dumping log files to the local storage.

- script-generator: the part of the system responsible for pulling the log files from the device to generate reports and charts. The generator comprises a set of *monkeyrunner-scripts*.[1] These are used to automate the interaction with the device without user intervention.

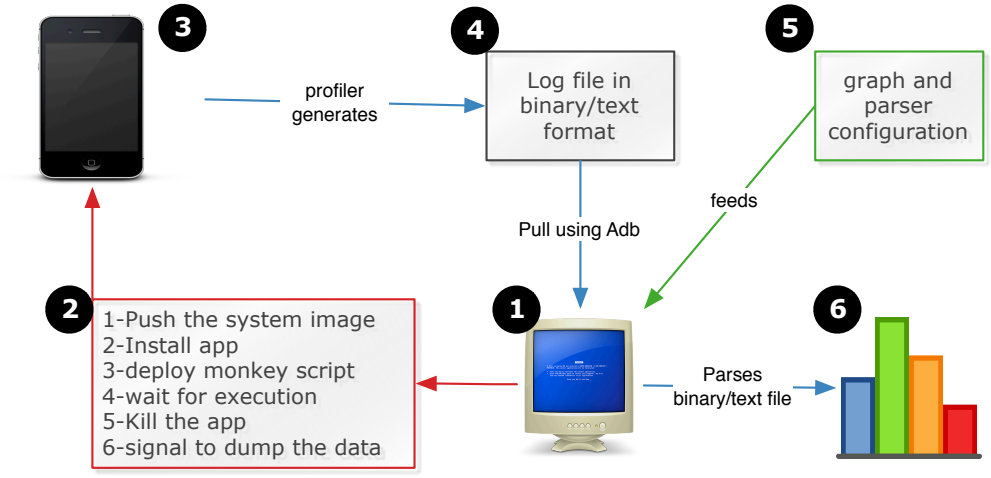

Figure 3.1.: Profiler diagram

Figure 3.1 shows the profiler including its two main components. There are six main steps to generate the information we are interested in:

1. build the image: we modify the system image and then we push the new build to the device.

2. script: a java program that will push the new image to the device following these steps:

   (a) push libdvm/system-image using adb

   (b) install the app to be profiled

[1] `http://developer.android.com/tools/help/monkeyrunner_concepts.html`

(c) reboot the device

(d) launch the app using monkeyscripts

(e) wait until the script terminates

(f) signal app to dump the log (SIGUSR2)

(g) signal the app to exit (SIGKILL)

3. in parallel with step2: The android device receives the commands through adb and the profiler will dump the information to local storage.

4. server archives the log file by pulling it from the Android device.

5. feed the script with an xml configuration to parse the file retrieved from the device. The scripts generate a set of graphs based on the specification read from the configuration file

3.2.1 Monitoring Applications Using Android Development Kit

Dalvik Debug Monitor Server (DDMS) displays thread and heap information on Android devices along with other information. DDMS provides a feature to track objects being allocated and to see which classes and threads are triggering these allocations. The reports generated by DDMS are very useful for a developer to evaluate his application. However, relying on DDMS would limit our system to the interface that DDMS offers.

DDMS communicates with the desktop machine through `logcat`. During our implementation, we found that dumping reports to logcat does not provide a reliable way to send all the information to the server. The reason is due to buffer overflow. Incoming messages after overflow will be dropped randomly.

`Android-API` provides a set of methods used to monitor some system information from the application level. In order to use these methods, it is necessary to change the source code of the applications. Changing source code is not feasible for the applications. Examples of the methods provided by the API include:

- `getAllocCount`: returns the objects allocated from the active heap.
- `resetAllocCount`: resets the objects counter to zero.
- `startAllocCounting`: enables the objects allocation profiling inside the VM.
- `stopAllocCounting`: disables the objects allocation profiling inside the VM.
- `startInstructionCounting`: enables the instruction counter inside Dalvik.
- `getInstructionCount`: returns the number of instruction being executed in Dalvik.

One of the advantages of building our system in Dalvik is to use system memory for storing system information which makes the profiling independent of the entities being profiled.

3.2.2 Metrics

We define a set of metrics to be gathered. While the metrics are common between scopes, the significance of each metric can be different for each one.

Live objects

Live-objects is defined as the number of accessible objects in the heap. Once a mutator acquires the lock to allocate a new object, the relevant counters will be incremented. After the object is initialized, the system will revisit in case some other counters need to be updated. The reason behind delaying the counters' updates is the fact that some information will be known only after an object is initialized, such as the object's class.

During memory collection, each collected object triggers a decrement operation to the counters. At the point of collection, the object metadata is still known to the system and tracking this information is still feasible. Live objects significantly reflects the allocation activity of a given entity (an entity can be a thread, class, app, etc.).

Life time

In the context of GCs, *time* at any point point is defined as the total memory allocated to that point in execution. Based on the previous definition, object life-time is defined as total memory allocated since the object was created until the object is freed. In order to accurately measure this value, we need to keep track of the time each object gets created. As described earlier, Dalvik does not store object headers. Instead, all the information is stored in global data structures (`bitmaps, card-tables`). We added four bytes per object to store the required information. The advantage of hiding such information from the GC data structure is to avoid any conflicts with the GC work.

When an object is allocated, the total allocated bytes *birth-time* is stored in the header. An object with birth-time equal to zero will be the oldest object in the system. During the collection, the collector thread will read the birth-time stored and use it to calculate the object life-time. Life time is an important metric to decide on the collection strategy frequently used to compare generational and non-generational GCs [14]. If the objects can be categorized as two different categories based on life-time, then we can create different heap regions. The area with smaller life-time will be scanned more frequently than an area with larger life-time. GC pause times should be reduced by scanning small heap regions in each cycle.

On the other hand, if objects tend to have the same life-time on average, then the generational-GC will add an unecessary overhead without gaining significant optimizations even with extra considerations such as variant nurseries [1]. We derive another metric known as *average-life-time* that aggregates the life-time of all the objects belonging to a certain entity. An entity with higher average has more long-lived objects in the system.

Cohort

Cohort is defined as a time window in the application run time. *Window* is predefined statically to divide the application run time into smaller units of time. A mutator allocating a new object will be filling the currently available cohort. The system keeps accumulating

the objects sizes being allocated to the cohort until the cohort is full. Once a cohort is exhausted, a new cohort will be used. Only one cohort is active at any given point of execution. Objects can span multiple cohorts as we leave no space gaps. We keep an array of preallocated cohorts from the system memory. Each cohort holds the metrics for entities allocated during the cohort active time such as *live-objects, live-bytes* and *average-life-time*. Cohorts are used to analyze application behavior with reference to time.

Objects size demographics

Dalvik allocates objects based on `8-bytes` alignment. We create a histogram for object-sizes by keeping a hash table allocated from the system-memory.

We avoided predefining any cut-offs in order to generate a histogram for all possible objects. Similar to the metrics described earlier, we can deduce some important information per object size. We used the histogram to study the effect of the non-moving property of the GC implementation. When a mutator fails to allocate an object, we count the number of times at which the object size is less than the total available memory in the heap. An object with size less than available memory would be allocated if the heap was compacted.

When the histogram is relatively balanced, a *segregated-free-list*[12] can be used to allocate the objects. When the histogram shows a dominant category, then creating a pool of items of that category can reduce allocation overhead and the same technique can reduce space gaps with a locality-tradeoff.

Pointer distances

This metric shows the relation between the objects allocated in the heap. Given a perfectly compacted heap, pointer distance is measured by the difference of date-of-birth in bytes between the target and source object. We keep a global count of the object references done (mutations). Then we force a full collection on the entire heap to keep the heap perfectly compacted. Oldest object has birth time of zero while the youngest object gets a birth time equal to the total allocations in the heap.

```
Object_pos(object) {
  //get the cohort
  cohort = cohorts[(object.BD/MAX_COHORT_SIZE)]
  //calculate the cohort position
  for(i < cohort.index) {
    cohort.pos += cohorts[i].size;
  }
  //calculate offset of object within the cohort
  object_offset =
    (object.BD % MAX_COHORT_SIZE) / (cohort.size/MAX_COHORT_SIZE);
  object_pos = cohort.pos + object_offset
}
```

Listing 3.1: Object position algorithm

When a reference field is updated, we measure the distance between the source and the target. If the source older than the sink, the distance between objects is considered to be in the positive direction. Otherwise, the distance is considered negative. The distance direction is significant to determine the allocation methodology across different applications.

The algorithm in Listing 3.1 shows the way we calculate the object position. This is the same techniques as used by Blackburn et al. [2]. The pointer distance can then be calculated as we show in Listing 3.2. We added the field offset to the pointer distance to get an accurate value. The offset field can be significant in large arrays or in large objects spanning multiple cohorts.

```
Pointer_distance(src, field, dot) {
  //get the position of both objects
  src_pos = Object_pos(src);
  dst_pos = Object_pos(dst);
  //consider the field offset
  distance = src_pos + offset(src, field) - dst;
  return log(distance);
}
```

Listing 3.2: Pointer distance algorithm

3.3 Scopes

3.3.1 Thread Scope

There is not enough information about how multithreading features are utilized on smartphone apps. Our profiler, through the thread analysis, shows the total number of threads, along with the live threads.

A high number of allocation-active threads implies that the application is utilizing the multicore features to some extent. The reports generated from our profilers include the system threads. It is feasible to exclude all system threads but we preferred to include them in the reports in order to build an intuition about the whole components.

Internal Threads

All the threads are native pthreads. Internal VM threads are in the "system" thread-group, while other threads are in "main" thread-group. The list of system threads includes:

- `Main`: the first thread created in Dalvik. Sometimes, it is called *UI-thread* (as we will describe in section 4.2.1).
- `GC`: daemon thread responsible for collecting garbage objects concurrently.
- `Compiler`: responsible for dealing with compilation tasks.
- `Signal-Catcher`: handles signals in Dalvik. It is used to dump thread stacks.
- `Stdio-Converter`: reads data from `stdout/stderr` and converts them to log messages.
- `JDWP`: Java Debugger Wire Protocol thread.

All internal threads are created by method `dvmCreateInternalThread` except `Main`. All threads are visible by default to the code running in the VM (the only exception is `JDWP`). Before starting a collection cycle, GCD sends a suspension request to all the

threads. A suspension request will be checked by each thread at certain points (called *safe-points*) in the main interpreter loop. As described in chapter 2, GCD only runs after all the threads respond to the suspension request.

Suspended threads wait on a conditional variable, and are signaled en masse. This raises a concern that objects allocated by a suspended thread are invisible elsewhere in the VM. Such objects will be collected by the garbage collector while they are still referenced by one of the suspended thread. To prevent these objects from being collected, each object is automatically added to a track-list and it will be removed from the list after it becomes visible to the root set.

Thread Profiling

Dalvik maintains a list of threads running in the VM called *thread-list*. Application threads are added to the same list. In order to accurately measure memory behaviors, we added four extra bytes in the object header to store the owner of each object (the thread allocating an the object). Each time an object is allocated, we determine the current thread, then we increment the counters of the thread accordingly. When a thread terminates, Dalvik removes it from the thread-list. After a thread gets detached, GC can collect the object-thread. To avoid the situation when we we have objects referencing a dead thread, we keep the thread-profiling records in a separate structure in the memory system to be independent of the actual life time of the thread. When a thread gets detached, we change the thread status to be `dead`.

Marking the object by its owner thread extended the information generated by the system to include live-objects, live-bytes, life-time, number of array objects, total allocated space per a single thread. Since the system keeps track of any thread added to the list at runtime, threads created during initialization phases (`zygote`) will be also listed in the profile system.

3.3.2 Class Scope

Class scope displays information by object types. Applications running on Android have different natures based on the following observations:

- applications used by user on daily basis rely heavily on graphical packages. Such applications are expected to allocate objects from Android packages.
- many VMs instances running on Android device are services initialized when the system boots. Services will use more of the libraries defined in the Android core system.

Grouping behavior by type can give pointers for potential optimizations. Optimizing a specific class allocation can lead to significant performance gains.

Class Profiling

In order to implement the profiling successfully, the injection is deferred until the object is initialized. A hash table is used to keep track of all the classes loaded in the VM. Once the object is initialized, the class referenced from the new object will be used to lookup the hash. If the class entry already exists, the counters will be updated accordingly. Otherwise, a new entry will be added to the hash(while marking the first time an instance was created). When the object is collected, the counters have to be updated by updating the relevant hash entry.

At any execution point, there will be a record for all the loaded classes. Some of these classes could be loaded before the application starts execution. This is due to the fact that the profiling system counts all the classes once the new heap is created (see chapter 2). The class record holds live-objects, current-bytes, total-bytes, total-instances and the average life time of class instances. We mark the time in which the first instance of that class was allocated. This can help in sorting the classes based on the time they were loaded in the application which by turn helps in excluding some events (i.e events triggered prior the application startup).

The class profiling can be extended further to aggregate metrics by class packages. The aggregation can show the metrics based on functionality such as graphical packages.

Extending Class-profiling

Blackburn et al. [2] built a benchmark "*DaCapo*" based on real Java applications. Built on top of real Java applications, DaCapo possesses object oriented characteristics relevant to real world systems. *Code complexity* and *Level of inheritance* are examples of the metrics used to evaluate the VM.

The same measurements can be generated from our system. It is also feasible to combine the profiling described in this section and section 3.3.1 to construct a relation between threads and the loaded classes during run time. Finally, mapping between threads and classes can identifies threads functionalities.

4 APPLICATIONS AND BENCHMARKS

In this chapter we show the set of the applications used to run the experiments. Our target is to run common Android applications available on Google Play™. In addition to applications available for android users, we ported a set of Java benchmark applications.

4.1 Android Benchmarks

There are many popular Android benchmarks available on Google Play store. The following list includes some of the most popular benchmarks excluding *Quadrant*[1] which will be described in details in this chapter:

- Sunspider tests the JavaScript language.
- Linpack tests how fast a computer solves a dense system of linear equations. The results of the score are displayed in millions of floating point operations per second (MFLOPS).
- Antutu Benchmark includes many tests like memory performance, CPU Integer Performance, CPU Floating Point Performance, 2D 3D Graphics Performance, SD card read-write speed, and Database IO performance testing. There is also "Antutu-3Drating-Benchmark" which focuses on 3D part of graphics processing unit (GPU) in Android devices.
- CF-Bench is a CPU and memory benchmark tool designed to handle multi-core devices. It tests both native as well as managed code performance.

[1] `http://www.aurorasoftworks.com/products/quadrant`

- Smartbench is a multi-core friendly benchmark application that measures the overall performance of the device. It is suitable for both productivity users and 3D gaming users.

As we have shown in the comparison between Jelly Bean and ICS, the software layer has a significant rule on the overall experience. The benchmarks provide a way to compare different devices. However, they do not provide a feedback on the software behavior regardless of the host. The user should be aware that the scores reported by the benchmarks are not sufficient to prove device superiority. All of the benchmarks described in the previous list are not available publicly. It is not feasible to validate the tests running on the benchmarks and to evaluate how relevant they are to the user applications. For example, a benchmark like Linpack[2] was mainly designed for performing numerical linear algebra on digital computers. Smartphone users may find that the significance of such benchmarks is not relevant to their daily use.

4.2 Porting Java Benchmarks

Since Dalvik is running applications originally written in `Java` language, we can run Java applications on Android to compare the behaviors with the same application running on Dalvik. This comparison has some limitations due to many reasons. Among these reasons is the huge difference in both scale and scope between the embedded devices and desktops.

4.2.1 Java Standard Libraries

Android supports many packages from Java `6.0` API.[3]. Some Java libraries are not fully supported on android and some others packages are completely left out. Android supports the higher level concurrent library `java.util.concurrent` which makes port-

[2] `http://www.netlib.org/linpack/`

[3] `http://docs.oracle.com/javase/6/docs/api/` To the current time, Java 7.0 features were not applicable on Android platform

ing multithreaded Java programs to Android platform a feasible process. Data structures defined in the same package are supported by definition. Android also supports some of the commonly used *Apache third-party*[4] libraries such as `apache http` which represents the core interface and classes of the `HTTP` components.[5] Android has its own system-calls to do the graphics. `java awt`, `javax swings`, `javax print` and `javax imageio` are not supported on android. Although, Android supports `javax.xml.parsers` package, it does not support the rest of the `javax.xml` package. There are many other libraries not supported on Android such as `javax.rmi` and `java.applet`.

4.2.2 Dependencies

In order to port Java programs, there are some few considerations that should be verified. Some of the steps can be automatically verified while some others need to be manually verified by the developer. Checking dependencies is the first step to port a Java program (excluding interface package like `swings` and `awt`).

If the java app has a set of dependencies (`jar` files), then all these jar files have to be successfully ported to Android. This step can be straightforward as long as all the libraries are supported as described in Section 4.2.1. Class file can be directly translated to Android executable file using `dx` tool which is available in android-SDK.

If any the libraries is not supported, then the code has to be modified to avoid importing unsupported libraries. In some circumstances, some libraries could be imported in the Java programs to provide a feature that does not fit with the embedded platform (i.e remote method interfaces). Removing the out-of-context classes can get rid of the unsupported dependencies.

[4] `http://apache.org/`

[5] `http://developer.android.com/reference/org/apache/http/package-summary.html`

4.2.3 User Interface

Java UI calls have to be replaced by Android APIs. Such step can be very tricky if the Java app is not built in a Model-View-Controller (MVC) separating the representation of information from the user's interaction. Among many issues that should be considered in this step, is the memory optimizations. GUI structure can affect the memory performance on Android device. The main reason causing memory leaks is the fact that views have a reference to the entire activity and therefore to anything the activity is referencing. Some practices are recommended to be used while redesigning the application to run on android in order to avoid memory leaks. Examples of good practices in Android GUI are:

- Recycling views instances
- Usage of weak references for static inner classes
- Avoiding long-lived references to activities

4.2.4 Threads

There are some considerations when dealing with threads especially the UI thread.

The first thread created in the VM is called "Main". The latter is the thread using which the application interacts with Android UI toolkit[6] and through which all drawing events are done. This qualifies the Main thread to be called *UI-thread*.

Android UI is not thread safe. In order to guarantee correct execution, it is necessary that any access to UI goes through the UI thread. Any other worker thread should only be used for doing computation and logic tasks.

4.2.5 File Manipulation

Android provides several options to store persistent data. Each application can have two different areas to store files:

[6] android.widget and android.view packages

- Internal: files stored in that area are private to the app and they are always available as long as the app is installed on the device.

- External: SD cards are used to store app files. The external storage may be removed at any time. Files stored on external storage can be shared between apps. Any app writing to external storage has to be configured to do so in the `manifest`[7] file.

When porting a Java program, the developer should decide whether the data is private or shared with other apps. Configurations and properties files (static files) are stored in `Android-Raw` folder. Accessing files from `Android-Raw` is done by android-API. Using Android-API to access files can be sufficient to make the application ready to run on Android device. Finally, an I/O wrapper can be used to replace `system.out` and `system.err` by redirecting log messages and exceptions to Android-log.

4.3 DaCapo

As mentioned earlier, Blackburn et al. [2] built a benchmark based on real-world `Java` applications.[8] Many DaCapo applications like fop, sunflow and batik have non-supported dependencies on Android platform. DaCapo defines a set of workloads to control the benchmarkworkload:

- large is defined for heavy loads

- default is used as lower workload

- small is used in some applications with heavy duties.

We ported two applications described in details in the following two sections.

[7] `http://developer.android.com/guide/topics/manifest/manifest-intro.html`

[8] 9.12-bach is the last DaCapo release published in 2009. There is an ongoing effort on mercurial repository that is not released yet.

4.3.1 Lusearch

Lusearch is a java application that uses `lucene`™[9] to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible. For Lusearch, the large workload runs `128` queries with thread limit equal to `128`. The default workload runs `64` queries with `64` thread limit. We excluded both workloads as we found that the number of threads and the queries is not relevant to Android apps.

Table 4.1: Lusearch characteristics on Dalvik

Characteristic	Value
Total-Allocation	135MB
Maximum-Live	800KB
Live Objects	8,000
Loaded Classes	551
Array Classes	68

`small` workload runs eight queries and limits the threads to eight. Table 4.1 shows Lusearch characteristics on Android with "small" configuration. The total allocation is `135MB` with a maximum of `800KB` during the run time. During the process of porting Lusearch, we substituted `lucene2.4` by `lucene3.6`. The `lucene2.4` byte code is relatively old to be translated by `dx` tool. In addition to `dx` incompatibility, `lucene2.4` had a class implementation `apache.lucene.search.RemoteSearchable` which depends on unsupported`Java-RMI` package. Substituting the `lucene` core library release used in DaCapo required some changes in the benchmark code since the API is different in both releases.

[9]Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.

4.3.2 Luindex

Similar to Lusearch, Luindex uses `lucene` to index a set of documents; the works of Shakespeare and the King James Bible. The work done to port this application is the same as described in Section 4.3.1. After a run is complete, DaCapo uses the size of the generated index files to validate the correctness.

4.4 SPECjvm98

SPECjvm98 is a benchmark suite that measures performance for `Java VM` (JVM) client platforms [5]. It contains eight different tests, five of which are real applications or are derived from real applications. Seven tests are used for computing performance metrics. One test validates some of the features of Java, such as testing for loop bounds. It is retired since the release of SPECjvm2008.

4.4.1 Compress

Compress is a simple Java compression utility. It is used for performance evaluation. In order to port Compress, it was important to remove all the user interface classes (`JFrames`) and to build a new interface for the Android device.

Table 4.2: SPECjvm98-Compress characteristics on Dalvik

Characteristic	Value
Total-Allocation	105MB
Maximum-Live	9MB
Live Objects	4,700
Loaded Classes	354
Array Classes	41

Table 4.2 shows the characteristics running on Dalvik with 105MB total allocation and 9MB maximum heap size. It was found that the majority of objects being allocated during the runtime are of type arrays. The object-oriented characteristics we measured on Compress matches the same results evaluated by Blackburn et al. [2].

4.5 Quadrant

Quadrant benchmark[10] tool is available on Google-Play. There are two versions that provide overall of twenty one tests covering the processor, memory, input, output, 2D graphics and 3D graphics performance. We used Quadrant Professional which include all the above tests.

Table 4.3: Quadrant characteristics on Dalvik

Characteristic	Value
Total-Allocation	22MB
Maximum-Live	6MB
Live Objects	225,000
Loaded Classes	1,401
Array Classes	87

Table 4.3 shows characteristics of running Quadrant Professional on Android. The runtime will trigger 22MB of total memory and maximum heap size of 6MB approximately.

[10]http://www.aurorasoftworks.com/

5 MEASUREMENTS

In this chapter, we show results from running our profiling system on a set of selected applications. The chapter starts by a detailed description of the environment used in the experiments.

5.1 Android Powered Device

We avoided using *android-emulator*[1] because of its functional limitations. For instance, running multithreaded applications on an emulator may not reflect the same behavior on powered devices.

Despite the fact that Android is available as an open-source project, changing the software layer of commercial smartphones is not feasible. The root branch allows a generic build which does not work on the device as the system image will be missing the necessary libraries to interface with the hardware components. Building Android from root repository is generic and the image built does not work on commercial smartphones.

In our experiments, we used DragonBoard™ Snapdragon™ S4 Plus APQ8060A Mobile Development Board (DB). DB has dual-core CPUs, Android 4.0 operating system, `WLAN/Bluetooth/FM` (WCN3660), GPS (WGR7640), 1GB SDRAM, 16GB eMMC, microSD, micro HDMI, mini-USB.

We pushed a modified image to DB using *Android Debug Bridge* (ADB)[2]. The step of running the modified system is considered a sanity check for the profiling system. A broken implementation will cause a failure during the system startup.

[1] available on android developer web site

[2] ADB is a versatile command line tool that allows communication with a connected Android-powered device

Operating System

The Operating system used in our experiments is Android 4.0, also known as *Ice Cream Sandwich (ICS)*. Google™ released *Jelly Bean*™ which was not supported by the time we develop our system.

Generally speaking, users report that Jelly Bean is faster on average in most of operations. This is due to some enhancements done in graphics layers. The graphic's tuning reflects faster execution time on Jelly Bean compared to ICS. Application running on Android rely heavily on graphical layer and small optimizations done in this layer will do a huge impact on the overall system. Even simple operation such as opening and closing apps on the device is affected by the graphics.

We compared the VM implementation in both release (Jelly-Bean and ICS) to verify whether the memory-management in Dalvik is the same or not. We concluded that there is no difference concerning the latter scope.

Beside graphics enhancements there are some other features added to Jelly Bean such as *Google Assistant* and voicemail enhancements which are not in the scope of this thesis.

5.2 Quadrant Results

In this section we show charts generated from our profiling system on Quadrant Professional. As we described in chapter 2, VM initialization is done by `zygote` which implies that some of the values are generated during the initialization phase.

Once Android system boots and `Zygote` finishes initialization, Quadrant will inherit some of the objects loaded by the parent process. In order to successfully monitor all the objects, the profiling has to start even before the app starts executing.

Any objects allocated or freed from the active heap (non-zygote heap) will be profiled by our instrumentation. We set the default maximum heap size to `64MB` and we force a perfectly compacted heap by collecting every `64KB` of allocation.

5.2.1 Heap Composition

In this section, we show the heap composition in Dalvik while running the full tests on Quadrant app. Each graph plots heap composition in lines of constant allocation as a function of time, measured in allocations. We set the cohort size to `256KB` (four times the size of GC window). The count of cohorts depends on the total allocations done by the app.

The top line corresponds to the youngest cohort representing the total volume of live objects in the heap. The bottom line represents the oldest cohort. Usually, the oldest cohort is filled during the creating of the new heap before the app starts.

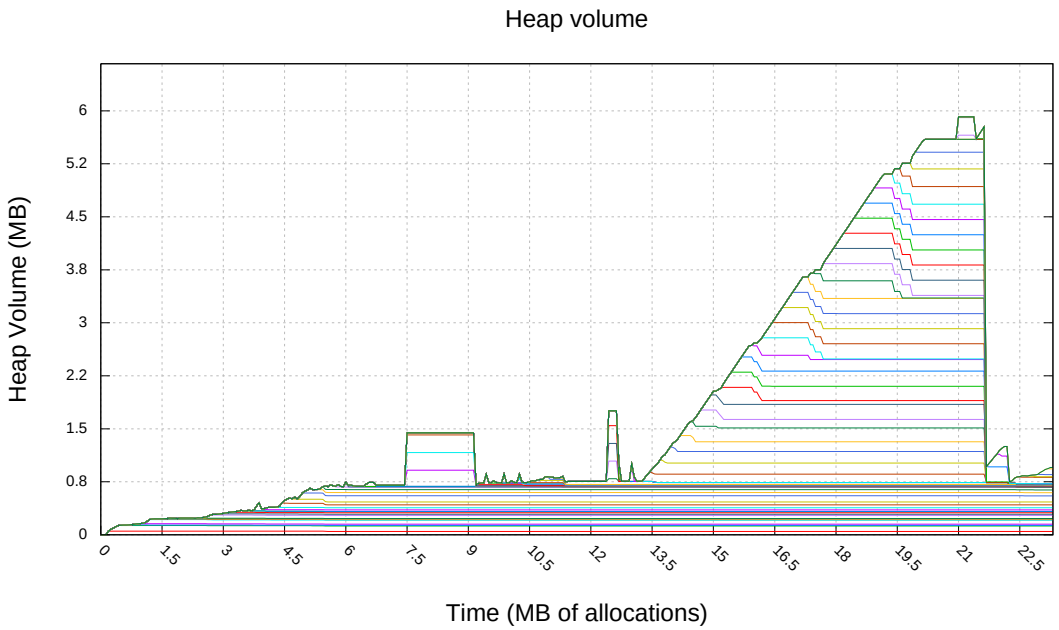

Figure 5.1.: Heap volume

Figure 5.1 shows the heap volume on y-axis, measured in bytes, as a function of time on x-axis, measured in total allocations. Y-axis indicates the maximum amount of memory allocated at any time, while the x-axis shows the total allocations. The gaps between each of the lines reflects the size in each cohort which implies that a gab should never exceed the maximum cohort size. When objects of a certain cohort are collected, adjacent lines move closer together till they merge when the cohort is fully collected.

It is noticeable that older cohorts tend to live longer than young cohorts. The older cohorts are also characterized by the constant size as they keep their size fixed during execution. This is due to the fact that cohorts allocated at the beginning of execution include classes, exception objects and objects used by Android activity once it starts. These objects tend to stay alive for the entire program execution.

On the other hand, young cohorts are filled by app objects. The cohorts endorsing these objects live as long as the objects are accessible. The latter cohorts shrink after each collection cycle which leads to the shape in Figure 5.1.

As we mentioned earlier, the cohorts can be used to analyze the app behavior over time. In the time frame `13.5-21.5MB`, Quadrant switches to a another test phase (2D and 3D graph) during which it allocates objects that are characterized to have bigger size and an average life-time equal to the test runtime. This leads to the "stair" shape as shown in the figure.

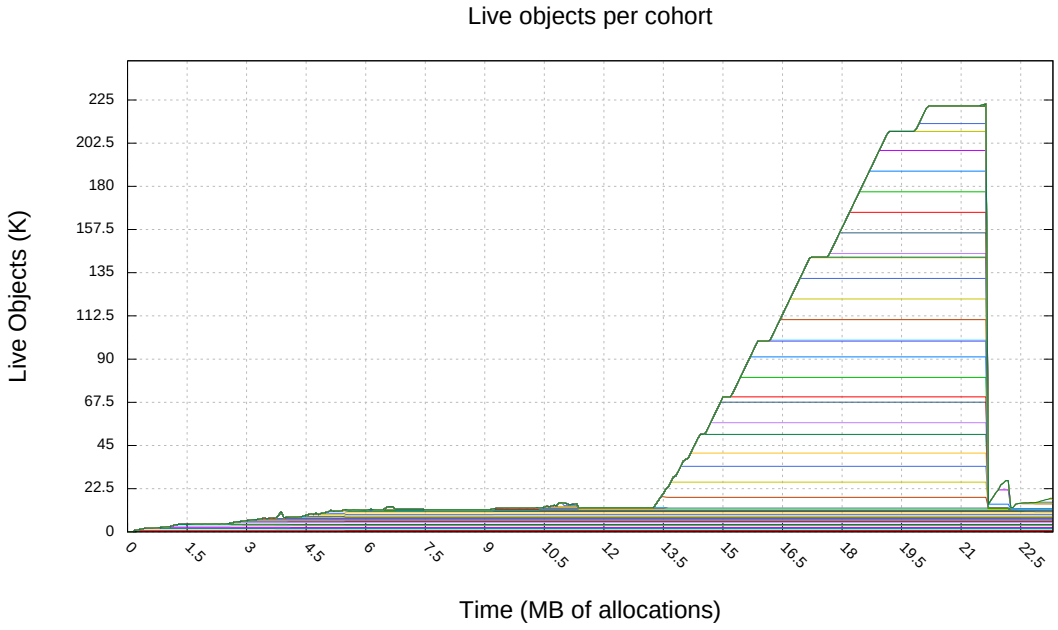

Figure 5.2.: Live objects per cohort

Figure 5.2 shows the number of live objects per cohort. The gab between each line is the number of live objects belonging to the upper line. Line will get closer to each other until they merge when all the cohort objects are collected.

The top line represents the total live objects in the heap at any point of execution. An object can span multiple cohorts but we count it only once, with the oldest cohort the object is spanning. This explains why some cohorts exist in heap volume Figure 5.1 while they do not in the live objects plot.

The maximum number of objects a cohort can have is calculated in equation 5.1:

$$\text{max\_objects\_per\_cohort} = \text{max\_cohort\_size}/8 \tag{5.1}$$

The minimum number of objects is zero since an object can actually span multiple cohorts. An example of such case can be highlighted by looking to both Figures 5.1 and 5.2 at execution times `7.9` and `12.5` on the x-axis.

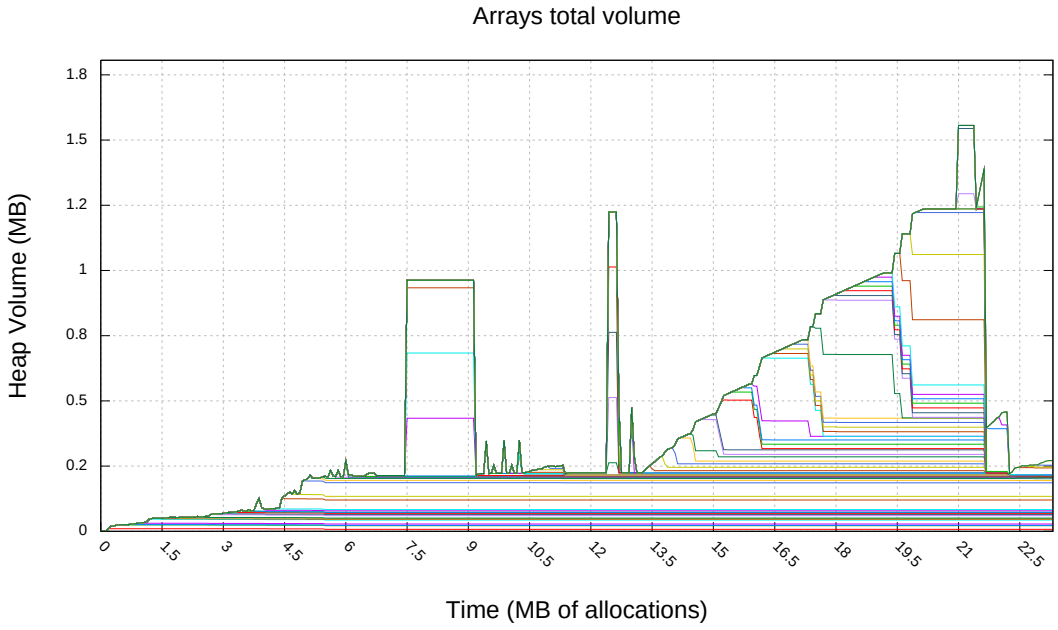

Figure 5.3.: Arrays total volume

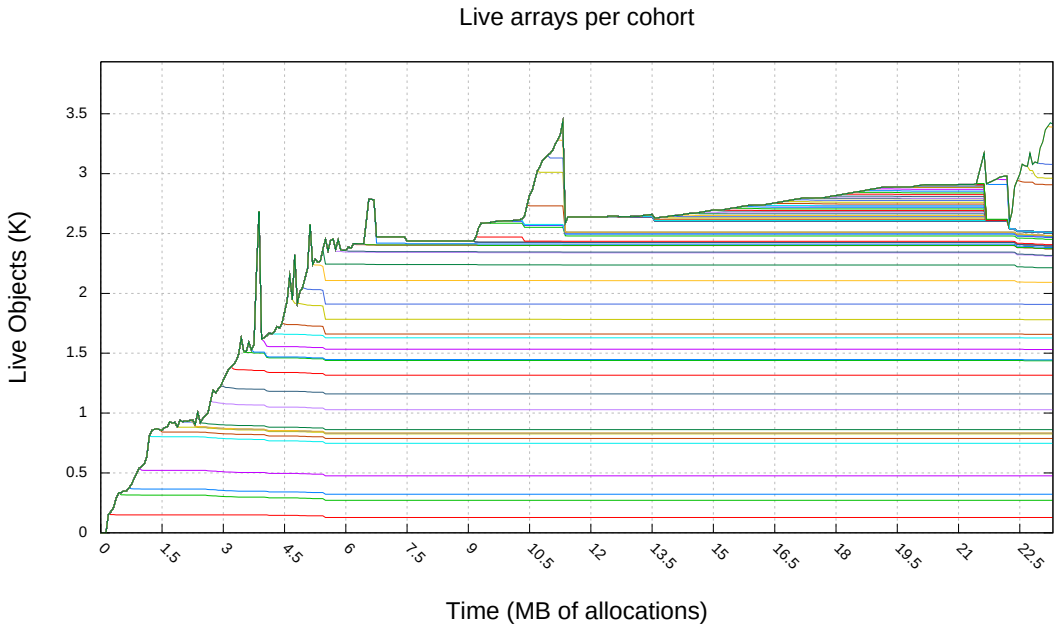

Figure 5.4.: Live arrays per cohort

Figures 5.3 and 5.4 show the live size and the live count including only array objects respectively. Similar to the live objects plots and the heap volume figures described earlier, we show the same metrics only when the object is an an instance of a class.

The data is displayed by cohort to reflect the array allocations behavior as a function of the execution time. A cohort that does not enclose arrays will not be represented in these plots. The gabs between lines represent the count or the size of the live arrays belonging to the cohort.

Figure 5.3 shows the total size allocated by arrays. By referring to the previous figures, we can see that execution time between `7.9-9` on the x-axis, the arrays use `60%` of total heap size and at point `21` arrays use `25%` of the total `6MB` heap size. Figure 5.4 shows the number of live array instances.

The latter two figures show the significance of array optimizations on Dalvik's performance taking into consideration the time spent during arrays' collections and the impact of the pause times on the user experience. JIT optimizations can significantly speed up the execution time considering array operations. Optimizing the collection of large arrays can

lead to significant performance gains. Example of these techniques is to combine *arraylets* and *semi-space collection* to reduce the pause time introduced by the collector [16].

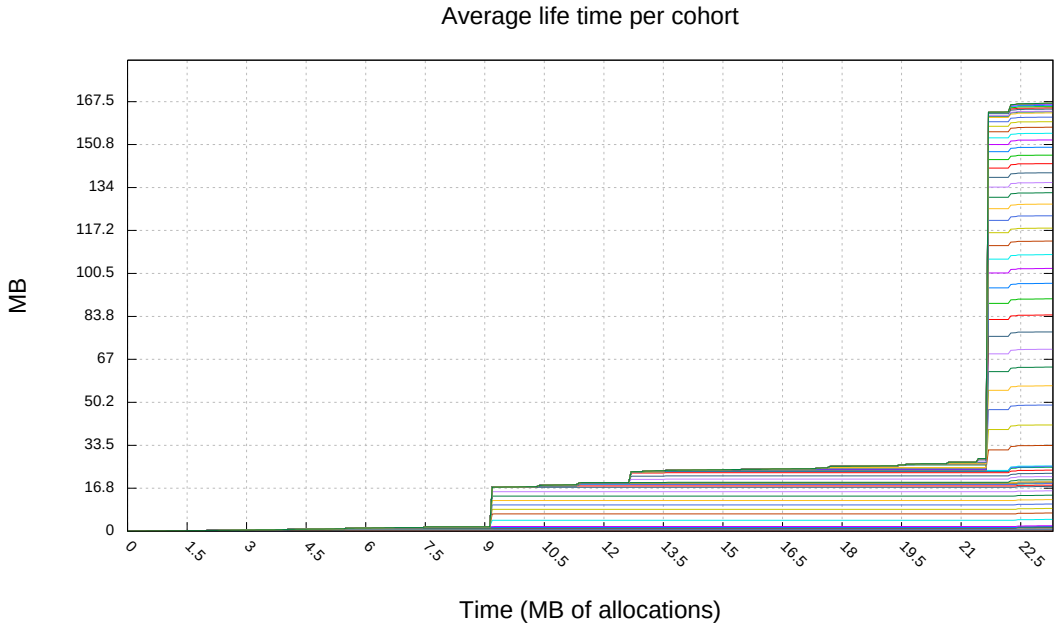

Figure 5.5.: Average life time per cohort

In Figures 5.5 and 5.6, we show the average life time of the instances for both objects and arrays respectively stacked by cohort.

The average life time can give insights on the collection rates since the average life time is updated only when objects get collected. Comparing both figures we can see that they have almost the same shape except the fact that Figure 5.4 is a subset of Figure 5.5.

5.2.2 Object Size Demographics

This section shows the object size demographics in Quadrant. As we mentioned earlier, we exclude the allocation metadata overhead. The results show that the demographics vary frequently with time.

Figures 5.7 and 5.8 show the object size demographics histograms. The histogram plots the percentage of objects on the y-axis versus the actual size on x-axis. The percentage is

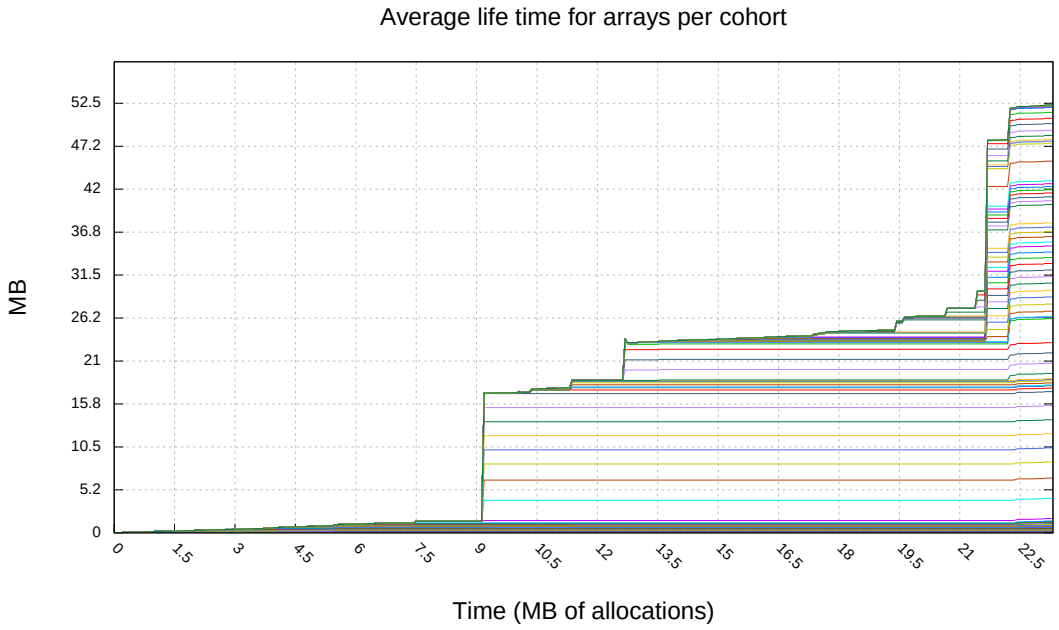

Figure 5.6.: Average life time for arrays per cohort

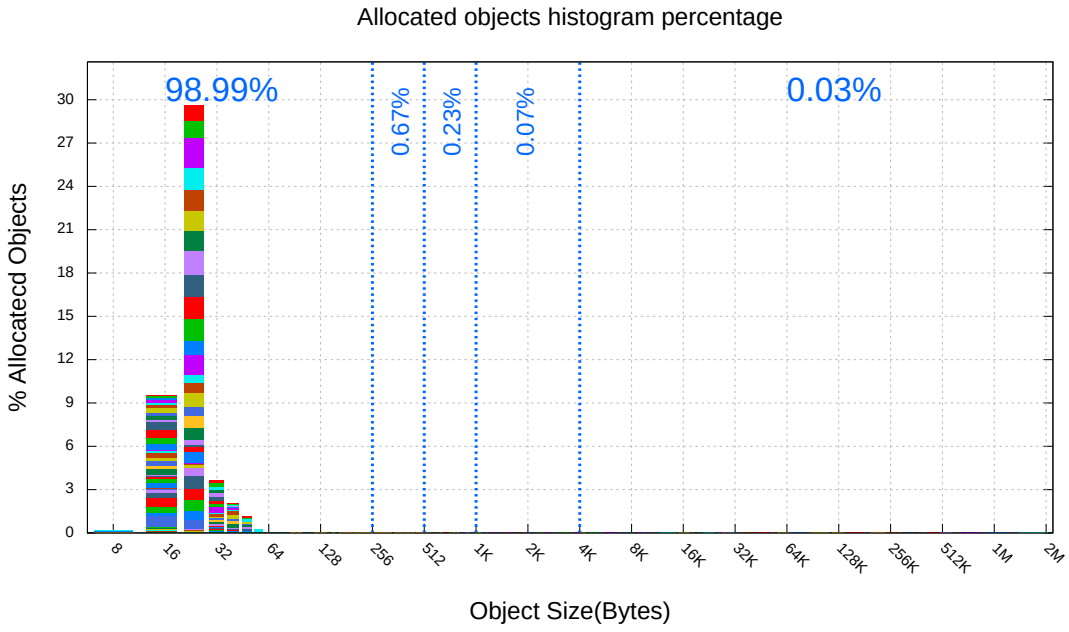

Figure 5.7.: Allocated objects histogram percentage

calculated based on the total of all object sizes. Each bar is stacked based on the count allocated by a single cohort. Displaying the stack by cohort shows the contribution of each cohort in allocating the object of the specified size (objects can belong to different classes) which Such information gives insight on the app behavior from the object size point of view.

Figure 5.7 shows the total allocated objects of each size over the entire runtime. The highest percentage of allocation is recorded by objects of size `24B`. By splitting the graph vertically, the graph shows that the highest allocation scores are recorded by objects of sizes less than `256B`.

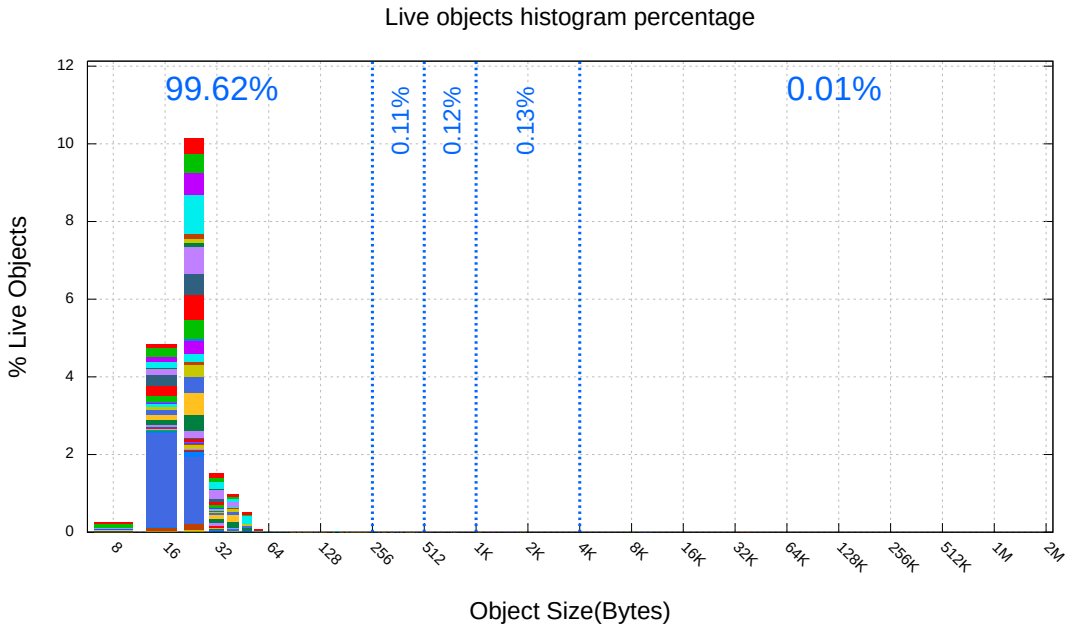

Figure 5.8.: Live objects histogram percentage

Similar to the total allocated objects, figure 5.8 shows only the live objects reported during the entire execution. The histogram shows that majority of small object sizes were allocated by the same cohort. This indicates that Quadrant has a phase in which it allocates many objects of sizes `16` and `24` respectively (including alignments).

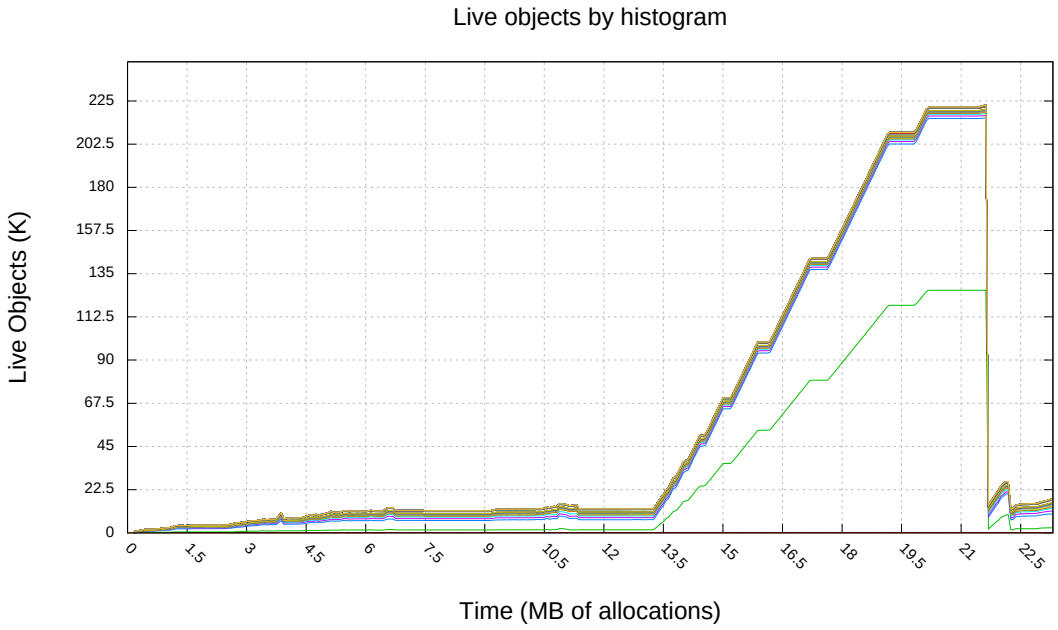

Figure 5.9.: Live objects by histogram

Figures 5.10 and 5.9 plot heap composition in lines of object sizes as a function of time, measured in allocations. These graphs show the heap composition based on object size demographics.

Live objects are grouped based on their size and each group is represented by a single line. The bottom line represents the objects of size `8B` while the top line represents the largest objects allocated in the heap. The gaps between each of the lines reflects the amount in each object size. When objects die and the number of live objects in the same object size is getting less, the lines get closer until they merge (when the live objects is zero).

Figure 5.9 shows the number of live objects during the runtime. The largest gaps in the plot shows that the majority of live objects are `16` and `24` bytes respectively as we pointed earlier from pervious graph. Since the x-axis shows the time, measured in allocations, we can see the change of Quadrant's behavior over time switching between test phases.

Finally, figure 5.10 shows the total memory occupied by a single group at any point of execution time. One single group can occupy space bigger than the sum of all the other

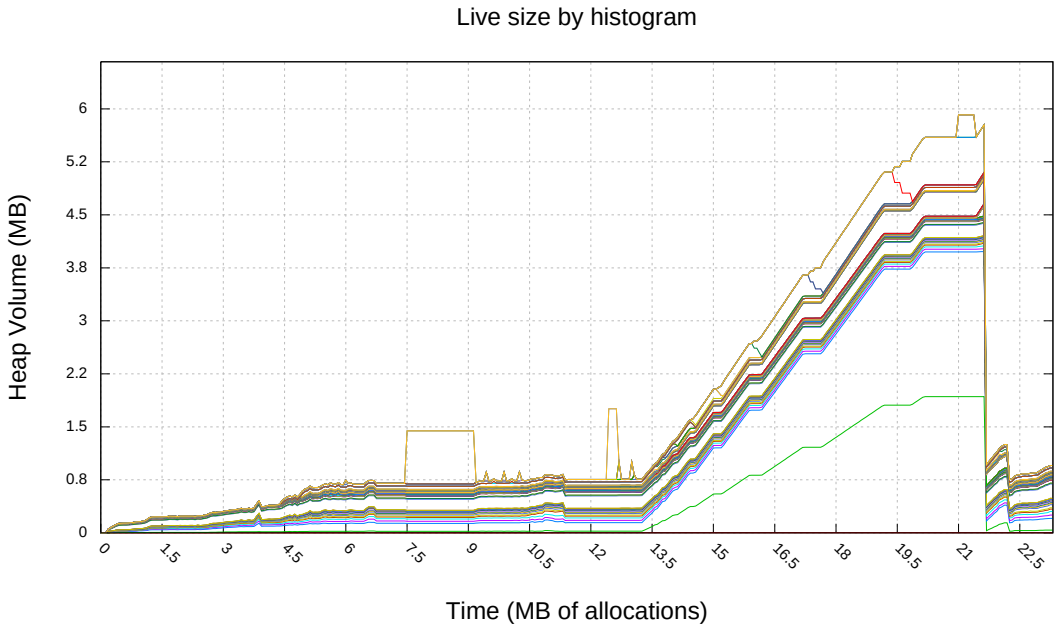

Figure 5.10.: Live size by histogram

objects. For example, on the x-axis, the period between `7.5-9MB` shows that the one single group occupies half of the heap size. The latter group has one single array.

5.2.3 Threads

The applications running on Dalvik are multithreaded. In this section, we show the metrics clustered by threads. The x-axis in the graphs represent time, measured in total allocations.

Figure 5.11 plots the contribution of each thread in the heap volume. Each line represents an active thread allocating from the VM heap. The gab between the lines represent the total size of heap occupied by that thread. The thread lines are sorted by the time, measured in total allocations, the threads are attached to the thread list inside Dalvik.

When a thread dies, the line representing will stay as long as the objects allocated are still alive. The lines merge only if all the objects allocated by a given thread.

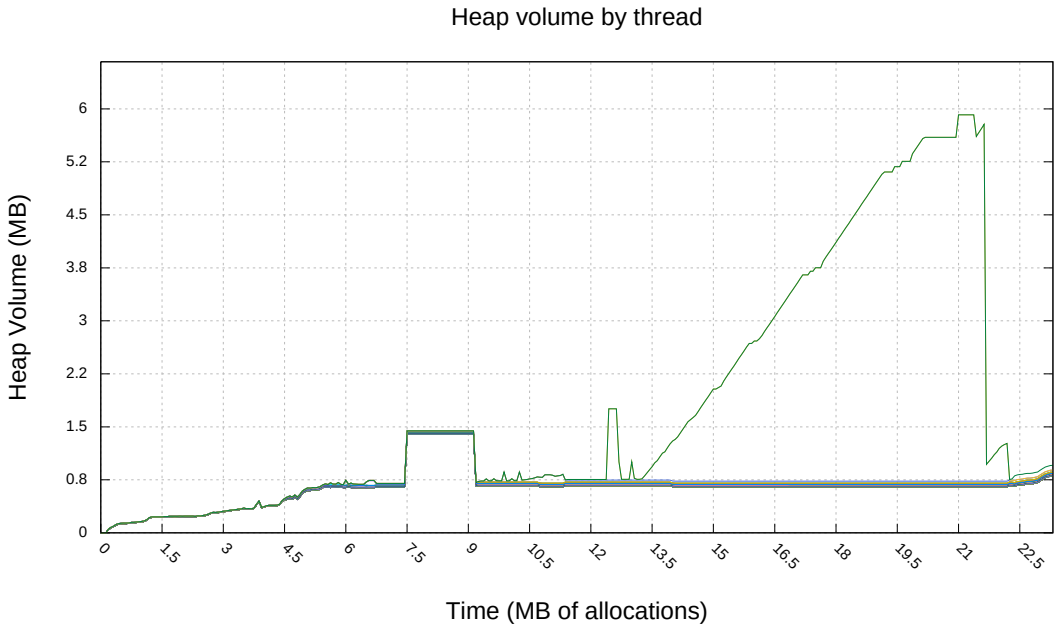

Figure 5.11.: Heap volume by thread

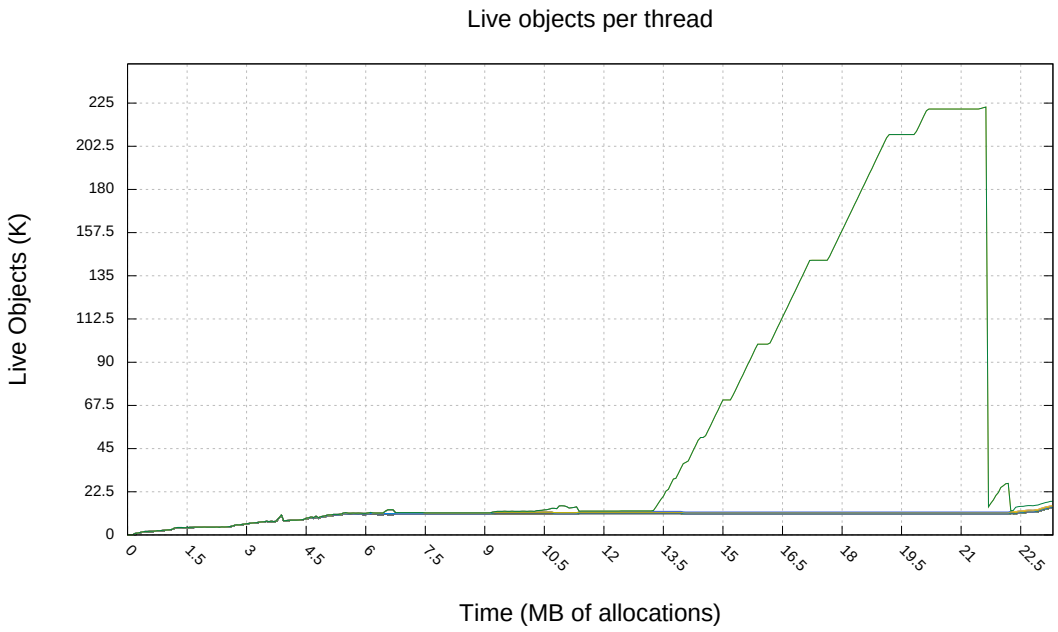

Figure 5.12.: Live objects per thread

We can see that there is one mutator doing aggressive allocation at a time. The other mutators tend to maintain their portion of the heap. The latter observation implies that Quadrant relies on one mutator We conclude that the graph tests done in Quadrant rely heavily on one single mutator doing all the memory allocations.

Figure 5.12 plots the number of live objects clustered by the owner thread. The figure reflects the frequency of the allocation rather than the heap volume used by the thread. The chart suggests that there is one dominant mutator with frequent allocation requests.

Referring to the synchronization mechanism used inside Dalvik (see Section 2.2.2), the exclusive access incurs unnecessary overhead on the app execution since the mutators allocate in a mutual exclusive fashion. The execution phase on x-axis framed `13.5-21.5MB` of allocations sets an example for such behavior. Each time, the mutator allocates an object, it has to lock the heap. The overhead of the heap could be reduced if the mutator has its own private heap.

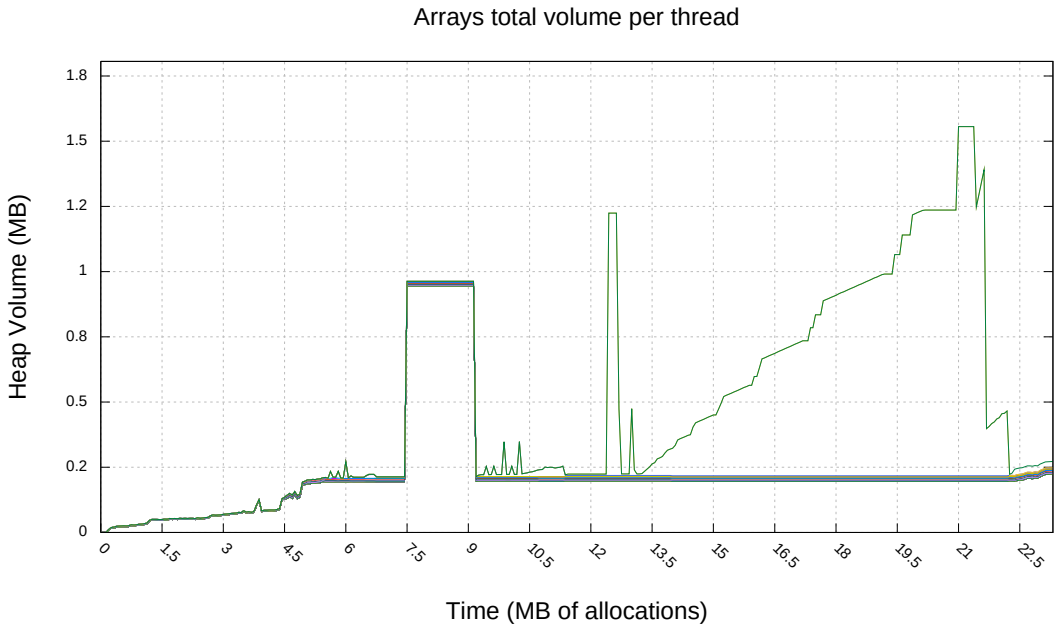

Figure 5.13.: Arrays total volume per thread

Figures 5.13 and 5.14 show the live size and the live count of arrays allocated by each thread.

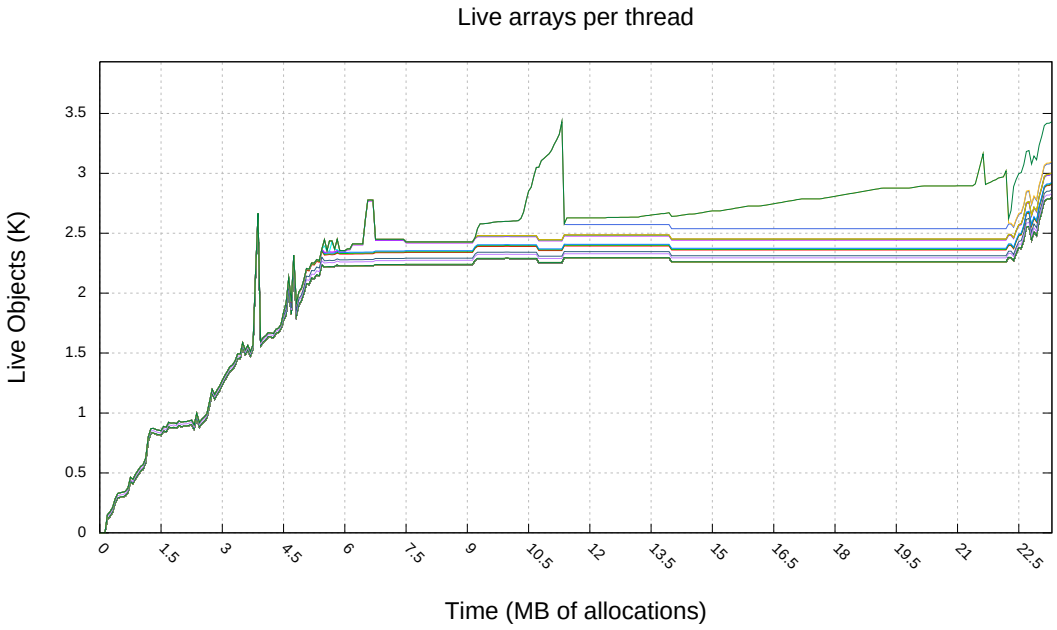

Figure 5.14.: Live arrays per thread

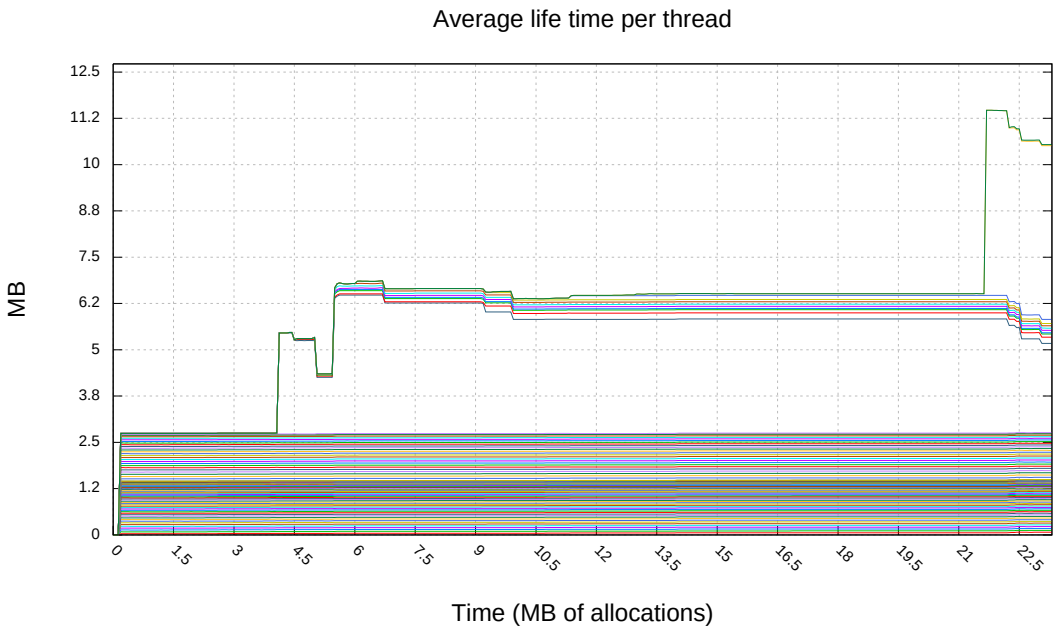

Figure 5.15.: Average life time per thread

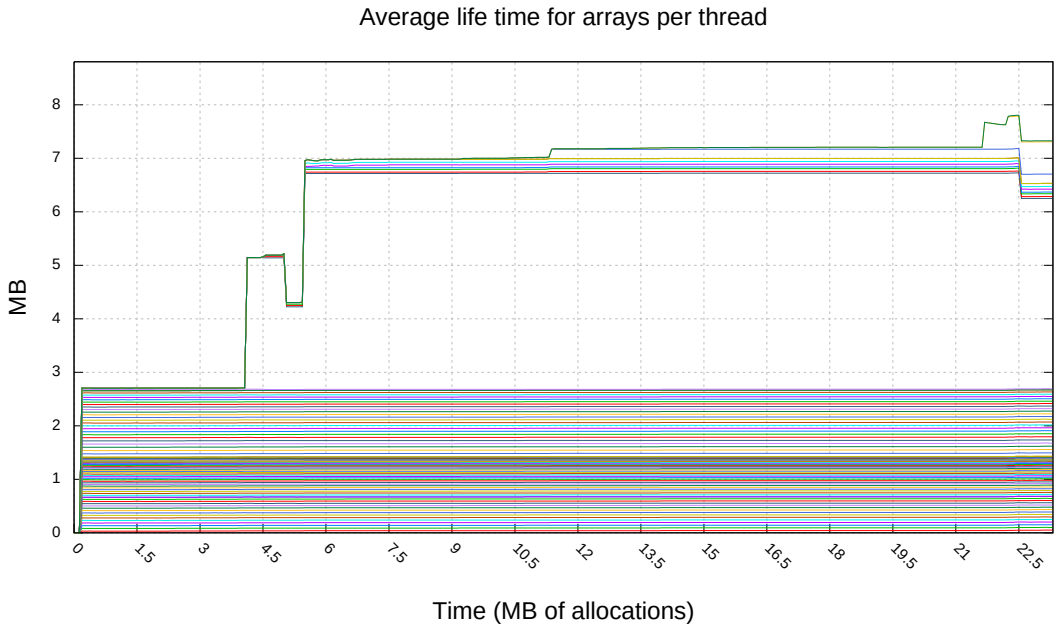

Figure 5.16.: Average life time for arrays per thread

Figures 5.15 and 5.16 show the average life time of objects and arrays per allocator thread. The number of lines displayed in these figures is different from the previous plots. This is due to the fact that the average life time metric takes into consideration all the objects being collected. Since, we start profiling once the new heap is created, many threads were attached to the thread list during the parent life time (the ancestor processes representing the path from zygote to the current app). All objects allocated during that path will influence the average life time.

5.2.4 Object-oriented Analysis

This section shows the metrics clustered by object types during Quadrant's life time. As soon as the new heap is created we start profiling the classes of the objects being allocated in the new heap.

As mentioned in chapter 3, the information is deferred until the object gets initialized. Before the initialization, the memory manager only knows the memory required, including the alignment, by the object.

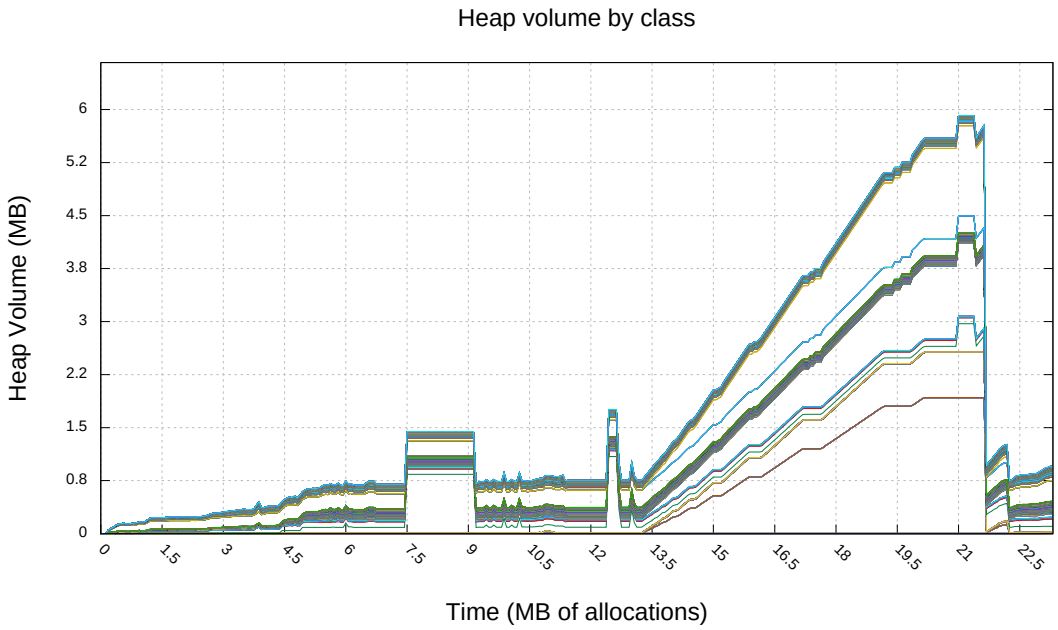

Figure 5.17.: Heap volume by class

Figure 5.17 shows the space occupied by each object class. The lines are sorted by class name. The lines merge when the objects are collected from the heap.

Figure 5.18 shows the number of live objects per class while Figure 5.19 shows the average life time of objects allocated for each class.

To summarize the information in previous plots, the tables from 5.1 to 5.3 show the total loaded classes and the classes of type arrays. The plots show the classes loaded since zygote created the app VM.

Tables 5.1 through 5.3 list the top records of class names sorted by live objects and show the percentage of total live objects. Each table shows the live, total objects and the total volume used by the given entry.

Table 5.3 only lists the classes of the objects being allocated after the app starts execution. In the latter table, the live objects and the total objects are in same ranges compared to

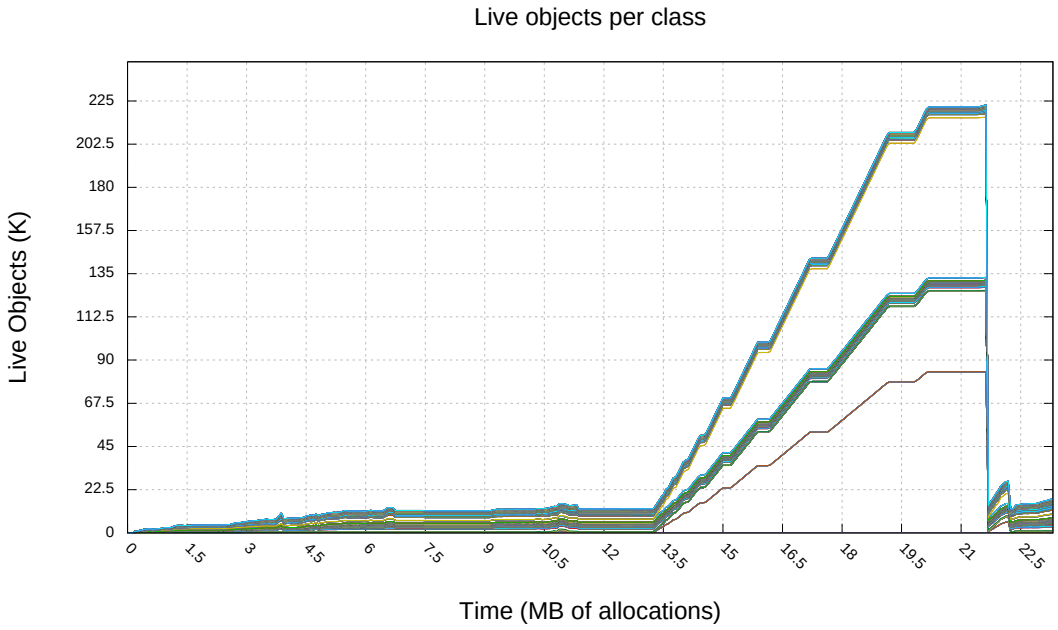

Figure 5.18.: Live objects per class

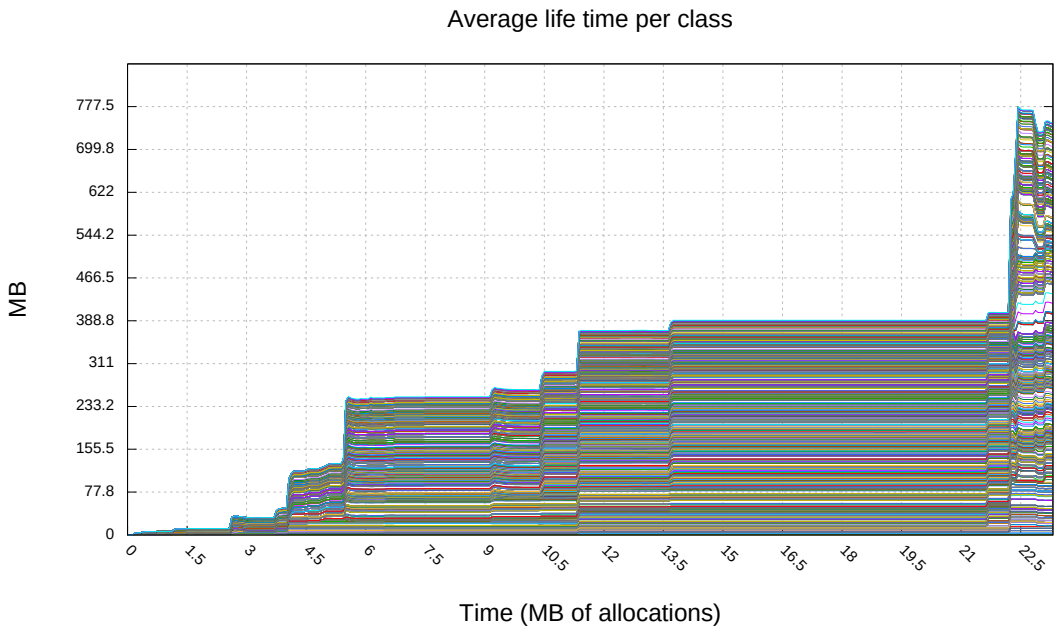

Figure 5.19.: Average life time per class

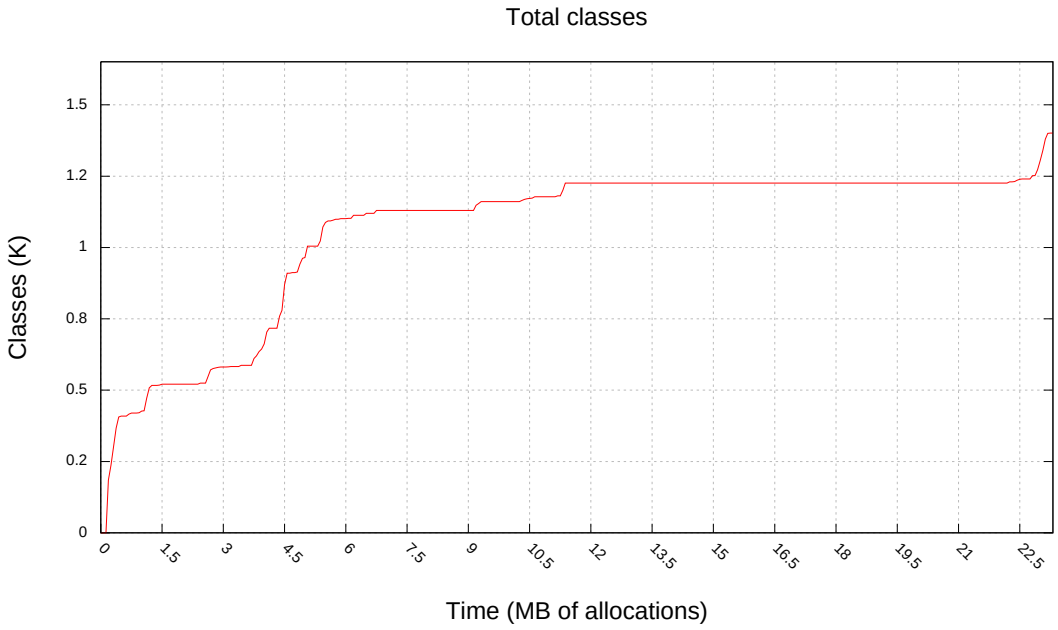

Figure 5.20.: Total classes

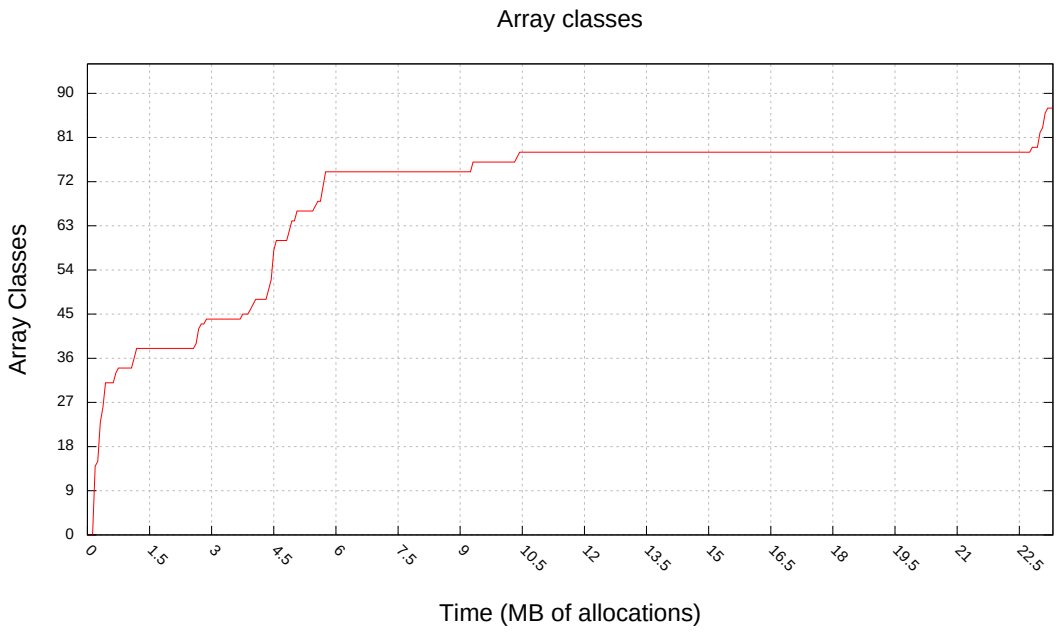

Figure 5.21.: Array classes

Table 5.1: Class statistics

Class Name	Live		Total	Size
	Val	%		
EM	4,532,891	31.83	8,518,737	103.9MB
java/lang/Short	4,498,389	31.58	7,029,736	68.8MB
Ff	2,266,325	15.91	4,359,615	34.6MB
java/lang/String	377,336	2.65	7,854,947	8.8MB
[C	354,643	2.49	9,039,962	19.4MB
java/lang/Class	352,896	2.48	353,082	57.9MB
java/util/HashMap$HashMapEntry	125,798	0.88	565,892	2.9MB
[Ljava/lang/Object;	118,449	0.83	2,174,394	55.2MB
java/util/ArrayList	92,651	0.65	1,791,794	2.1MB
java/lang/ref/FinalizerReference	66,468	0.47	381,121	2.6MB
nG	56,883	0.4	158,012	1.3MB
hI	48,611	0.34	62,646	1.1MB
java/lang/ref/WeakReference	47,861	0.34	110,936	1.1MB
[I	40,651	0.29	793,001	2.7MB
android/graphics/Rect	33,975	0.24	93,279	808.8KB
yk	32,626	0.23	35,781	775.5KB
rJ	32,042	0.22	33,139	758.1KB
org/apache/harmony/luni/lang/ reflect/ListOfTypes	30,945	0.22	688,077	493.4KB
java/util/LinkedHashMap$LinkedEntry	26,533	0.19	158,049	841.2KB
java/util/HashMap	25,689	0.18	258,355	1.2MB
[Ljava/lang/reflect/Type;	25,669	0.18	261,068	498.6KB
mT	23,775	0.17	23,775	375.1KB
java/lang/reflect/Field	23,646	0.17	921,851	929.1KB
xN	23,533	0.17	29,574	560.1KB
[Ljava/lang/Class;	20,308	0.14	2,310,210	395.3KB
iC	18,308	0.13	18,308	429.1KB

the first table. Table 5.3 finally lists the same information by aggregating the measurements based on the package to which the object class belongs to. The table shows the frequency of allocation libraries during the runtime. This represents useful information to see the heavy dependencies of an app running on Android.

5.2.5 Pointer Distances

In this section we show the pointer distance between the source and sink objects with reference to the ages of the objects. Relatively large object distances will influence MS performance. The trace algorithms incur an overhead as the locality and cache are invalidated by accessing objects from different memory pages.

Figure 5.22 shows the relative distances between the sources and targets of pointers mutations in the heap. As we described earlier, the positive distances represent the old to young object references, while negative values are pointers from young to old objects. We consider zero distances a negative references.

We group the distances by power of two buckets and the latter figure shows the percentage of mutations in a given bucket, normalized to the total number of mutations in the time period. The x-axis is time measured by total mutations.

The mutations are more common in the negative direction. An object is created first then it is used by some other younger objects. At some given point, we can see that the mutations in negative directions are `99%` of the total mutations. We found that the majority of positive mutations have value less than or equal to eight bytes.

Figure 5.23 shows the relative distances between the sources and targets of pointers mutations in the heap by taking a snapshot of the heap. The shape of the curve is different from the curve we showed earlier in figure 5.22 because the latter only shows the distances incurred by field mutations while the latter graph shows all the distances between objects in the heap including non-mutable references. The main characteristic of the snapshot the graph is that the positive direction is dominant.

Figure 5.22.: Pointer mutations distance

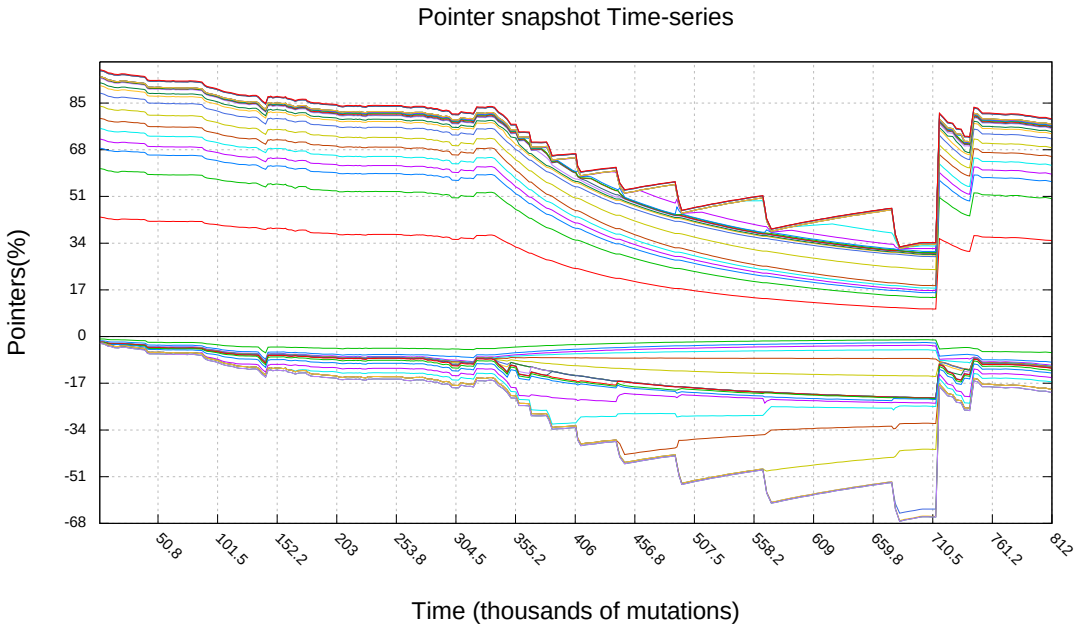

Figure 5.23.: Pointer distance snapshot

Table 5.2: Classes loaded from Quadrant app

Class Name	Live		Total	Size
	Val	%		
com/aurorasoftworks/quadrant/core/J	2,804	10.02	5,313	43.8KB
com/aurorasoftworks/quadrant/core/ay	2,402	8.58	2,402	76.4KB
com/aurorasoftworks/quadrant/api/ device/DefaultDeviceScore	2,096	7.49	2,096	81.9KB
[Lcom/aurorasoftworks/quadrant/core/ h;	1,087	3.88	1,681	35.8KB
com/aurorasoftworks/quadrant/core/as	1,076	3.85	2,041	16.8KB
com/aurorasoftworks/quadrant/core/g	1,076	3.85	1,076	50.4KB
com/aurorasoftworks/common/android/ ui/b	1,006	3.6	1,006	23.6KB
com/aurorasoftworks/quadrant/core/aA	730	2.61	743	11.4KB
org/codeaurora/Performance	447	1.6	738	7KB
com/aurorasoftworks/quadrant/core/p	343	1.23	343	5.4KB
com/aurorasoftworks/quadrant/ui/ professional/ QuadrantProfessionalApplication	277	0.99	277	19.5KB
com/aurorasoftworks/quadrant/core/aq	264	0.94	264	6.2KB
com/aurorasoftworks/quadrant/api/ xml/ConfiguredXStream	262	0.94	262	22.5KB
com/aurorasoftworks/quadrant/api/ xml/ConfiguredXStream$$anon$1	262	0.94	262	4.1KB
com/aurorasoftworks/quadrant/api/ xml/SimpleResultConverter	261	0.93	261	2KB
com/aurorasoftworks/quadrant/api/ xml/AndroidBenchmarkScoreRQConverter	260	0.93	260	2KB
com/aurorasoftworks/quadrant/api/ xml/ DefaultAndroidDeviceInfoConverter	260	0.93	260	2KB
com/aurorasoftworks/quadrant/core/Z	241	0.86	241	3.8KB
com/aurorasoftworks/quadrant/core/aa	241	0.86	241	3.8KB
com/aurorasoftworks/quadrant/core/at	241	0.86	241	3.8KB

Table 5.3: Packages allocated in Quadrant

Package Name	**Live**		**Total**	**Size**
	Val	%		
java/lang	5,589,671	86.52	27,460,461	198MB
java/util	399,084	6.18	4,256,702	12.3MB
android/	339,096	5.25	1,467,524	22.8MB
apache/	47,870	0.74	1,449,564	978.2KB
aurorasoftworks/	27,534	0.43	52,780	1.2MB
java/nio	24,629	0.38	547,882	916KB
thoughtworks/	13,775	0.21	56,292	249.5KB
java/io	8,422	0.13	98,551	277.8KB
java/text	5,361	0.08	11,793	95.6KB
java/math	2,760	0.04	2,760	75.5KB
dalvik/	1,385	0.02	1,939	32.5KB
codeaurora/	447	0.01	738	7KB
libcore/	413	0.01	112,195	16.2KB
javax/microedition/	208	0	949	212.9KB
jfree/	74	0	83	3.3KB
kxml2/	0	0	200	0

6 SUMMARY

Android became an important computing platform. Understanding the behavior of applications is very important to build a comprehensive analysis of the runtime behavior. Hence, identify the potential optimizations that can lead to effective system improvements. Analyzing such systems is indeed more complicated since it includes many factors such as user interaction, power dissipation, the software and the hardware components.

Deciding on what could significantly influence the system performance needs a set of benchmark tools that can be used to generate profiling information critical to the system usage. We implemented a profiling system on top of Dalvik to generate measurements for a set of metrics commonly used in literature in order to evaluate the memory behaviors of applications running on Android systems.

We used the profiler to generate measurements from a set of applications. The latter set includes two common Java benchmarks: DaCapo and SPECjvm98. Based on the profiles we generated, we found that Dalvik is tuned to use less space. The latter restriction leads to unnecessary overheads by frequently triggering a collection. Synchronization between VM threads introduces a significant overhead as we find that the there is one single mutator allocating from the heap. The atomic operations could be reduced by having a private allocator per thread.

The memory manager has an influence on the app response time. Long collection cycle causes the app to freeze which by turn negatively affects the user experience. Some UI events are more sensitive to pause times such as “scrolling”. The memory manager influences the power consumption on Android considering the number of times the collector in each app. Finally, Dalvik does not dynamically adapt to memory consumption and it tends to keep the same threshold no matter how the allocation behavior is.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, 1989. ISSN 1097-024X. doi: 10.1002/spe.4380190206.

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 169–190, October 2006. doi: 10.1145/1167473.1167488.

[3] Ed Burnette. *Hello, Android: Introducing Google's Mobile Development Platform*. Pragmatic Bookshelf, 2010. ISBN 978-1-934356-56-2.

[4] comScore. US mobile subscriber market share, May 2012. URL http://www.comscore.com/Insights/Press_Releases/2012/8/comScore_Reports_June_2012_U.S._Mobile_Subscriber_Market_Share.

[5] Standard Performance Evaluation Corporation. SPECjvm98: Documentation, release 1.03 edition, March 1999. URL http://www.spec.org/jvm98.

[6] Sylvia Dieckmann and Urs Hoelzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. Technical report, University of California at Santa Barbara, Santa Barbara, CA, USA, 1998.

[7] Black Duck. Android language summary. URL http://www.ohloh.net/p/android/analyses/latest/languages_summary.

[8] eLinux. Android power management. URL http://www.elinux.org/Android_Power_Management.

[9] Google. Android official site, . URL http://android.com.

[10] Google. Android developers, . URL http://developer.android.com.

[11] Google. Android Source, . URL http://source.android.com/.

[12] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 2011. ISBN 1420082795, 9781420082791.

[13] Doug Lea. A memory allocator. URL http://g.oswego.edu/dl/html/malloc.html.

[14] 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. ISSN 0001-0782. doi: 10.1145/358141.358147.

[15] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960. doi: 10.1145/367177.367199.

[16] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 146–159, Toronto, Ontario, Canada, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806615.

[17] Aurora Softworks. Quadrant. URL http://www.aurorasoftworks.com/products.