2013-01-01

Automatic Memory Management Policies For Low Power, Memory Limited, And Delay Intolerant Devices

Md Abu Jahid
University of Texas at El Paso, majahid@miners.utep.edu

AUTOMATIC MEMORY MANAGEMENT POLICIES FOR LOW POWER, MEMORY LIMITED, AND DELAY INTOLERANT DEVICES

MD ABU JAHID

Department of Computer Science

APPROVED:

Eric Freudenthal, Chair, Ph.D.

Luc Longpré, Ph.D.

Bryan Usevitch, Ph.D.

Salamah I. Salamah, Ph.D.

Benjamin C. Flores, Ph.D.
Dean of the Graduate School

to my

PARENTS

with love

AUTOMATIC MEMORY MANAGEMENT POLICIES FOR LOW POWER, MEMORY LIMITED, AND DELAY INTOLERANT DEVICES

by

MD ABU JAHID

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

August 2013

Acknowledgements

I would like to express my gratitude to my advisor, Dr. Eric Freudenthal, from the Computer Science Department at The University of Texas at El Paso, for his prudent advice, encouragement, patience and constant support. He always managed time for my thesis work even in his busiest day. I have learned so many things from him that will guide me in my career. I will always be grateful to him.

I also wish to thank the other members of my committee, Dr. Luc Longpré and Dr. Salamah I. Salamah from Computer Science Department and Dr. Bryan Usevitch from the Electrical and Computer Engineering Department, at The University of Texas at El Paso. Their suggestions, comments and additional guidance were valuable to the completion of this work.

I would like to thank Adrian, Gabe, Roger, David, Big Edward, and Edward from "Robust Autonomic Group" at UTEP for their contributions, suggestions, and encouragement.

NOTE: This thesis was submitted to my Supervising Committee on August, 2013.

Abstract

Mobile devices such as smartphones and tablets are energy and memory limited, and implement graphical user interfaces that are intolerant of computational delays. Mobile device platforms supporting apps implemented in languages that require automatic memory management, such as the Dalvik (Java) virtual machine within Google's Android, have become dominant. It is essential that automatic memory management avoid causing unacceptable interface delays while responsibly managing energy and memory resource usage.

Dalvik's automatic memory management policies for heap growth and garbage collection scheduling utilize heuristics tuned to minimize memory footprint. These policies result in only marginally acceptable response times and garbage collection significantly contributes to apps' CPU time and therefore energy consumption.

The primary contributions of this research include a characterization of Dalvik's "baseline" automatic memory management policy, the development of a new "adaptive" policy, and an investigation of the performance of this policy. The investigation indicates that this adaptive policy consumes less CPU time and improves interactive performance at the cost of increasing memory footprint size by an acceptable amount.

Table of Contents

Chapter 1

Introduction

This thesis describes a preliminary effort to examine the interaction of garbage collection scheduling and heap size on power constrained delay-intolerant systems. This sensitivity study on the interaction of heap growth upon energy consumption led to the development of alternate adaptive GC scheduling policy. Preliminary evaluation of this adaptive GC policy is highly encouraging.

1.1 Context

Languages that include automatic memory management are becoming increasingly prevalent in mobile systems. Early mobile offerings such as Apple's iOS only support languages with explicit memory management. In contrast, more recent entrants such as Google's Android and Windows Common Language Runtime (CLR) have attracted a large developer base accustomed to languages with automatic memory management and therefore require *garbage collection* (GC).

Computation consumes energy and GC is a computationally expensive operation [1]. Mobile devices are energy limited; batteries are expensive, heavy, and toxic. The rate of energy consumption is inversely related to time between recharge. This motivates a system design focus towards balancing energy consumption with responsiveness.

Our literature review indicates a dearth of prior research examining the impact of GC upon energy consumption of delay-intolerant interactive systems such as tablets and smart phones.

The focus of this research is to reduce the energy and computational resources required

for GC on mobile systems while neither causing significant degradation of performance nor significantly increasing utilization of another severely restricted resource.

1.2 Relevance Beyond Android And Dalvik

While this research focuses on Dalvik its conclusions may be relevant to a variety of resource- and response-time sensitive systems that implement automatic memory management. In addition to mobile platforms, front- and middle-tiers of online services are frequently implemented in Java or other languages with automatic memory management. Energy consumption in these facilities now dominates their operating costs [5].

While computing centers are not a principal focus, their power consumption is significant [2]. Since language runtimes requiring automatic memory management such as Java's J2EE are commonly used in those contexts, the approaches examined in this thesis report may have relevance in that context as well.

1.3 Research Contributions

This research was conducted in collaboration with the Robust Autonomic Systems group at The University of Texas at El Paso. The author's contributions to this research include

1. *Examination and analysis of the existing implementation of Dalvik to determine its GC scheduling strategy.*
2. *Implementation of a low-impact heap performance monitoring system for Dalvik.*
3. *Implementation and analysis of an initial sensitivity study of the interaction between heap size and energy consumption.*
4. *Collaboration on the design of an alternative adaptive GC scheduling strategy.*
5. *Implementation of this alternative strategy.*

6. *Initial evaluation of this alternative strategy.*

1.4 Android's Dalvik (J)VM

Dalvik is the virtual machine (VM) in Google's Android operating system that executes apps written in Android's variant of Java [10]. Dalvik is open source and was originally written by Dan Bornstein, who named it after the fishing village of Dalvk in Eyjafjrur, Iceland [3]. Dalvik is an integral part of the Android Operating System which is typically used on embedded devices such as mobile phones and tablets.

1.5 Why Android?

Android's Dalvik (J)VM has been selected as the target for this research due to the availability of source code and current dominance in the smartphone market. Google is activating 1.5 million android devices daily and their projection is a total of one billion devices by December, 2013 [4].

Android is used on a variety of power sensitive devices including mobile phones, tablets, and netbooks. Recently, Android is being used in embedded devices such as smart TVs and media streamers [3].

1.6 Dalvik GC Scheduling Policy

Dalvik implements both traditional *stop the world* (STW) and background (BG) GC. Background GC performs the computation in background using a separate low priority thread. Dalvik's baseline GC scheduling policy is substantially tuned to minimize heap memory footprint size while also providing acceptable interactive performance. Our investigations indicate that Dalvik can enter prolonged degenerate configurations where this strategy results in repeated low yield GC operations which consume a substantial fraction of CPU

energy. Android's Dalvik baseline GC policy is described in detail in Chapter 3.

1.7 Research Hypothesis

This research investigates a hypothesis that an automatic memory management policy for Dalvik can effectively manage both energy consumption and execution pause due to GC without inordinately increasing memory footprint. In order to evaluate the hypothesis, sensitivity studies using *exploratory* policies named MI2 and MI4 (described in Chapter 4) were implemented and evaluated.

1.8 Adaptive GC Scheduling Policy

The sensitivity study examines the interaction of heap size to overall CPU time spent in GC and the frequency of GC. Preliminary evaluation of the exploratory policies led to the development of alternate adaptive GC scheduling policy that completely avoids GC pauses for the applications we examined, consumed substantially less CPU time for GC, and caused only a moderate increase of heap size.

Chapter 2 summarizes relevant previous work in GC. The baseline policy is described in Chapter 3, and the exploratory and adaptive policies are described in Chapter 4.

Chapter 2

Background

2.1 Overview

This thesis examines policies related to automatic memory management, a periodic and computationally expensive operation [1]. This computation affects (principally) CPU energy consumption, and program responsiveness.

Dalvik's automatic memory management system implements a mark-and-sweep tracing collector with conservative heap growth policies.This thesis includes an investigation of the implications of these policies upon energy consumption, heap size, and responsiveness.

This chapter summarizes relevant aspects of energy consumption within mobile devices and contextualizes our investigation of automatic memory management within Android. This chapter concludes with a short discussion of self optimization that motivates strategies discussed in subsequent chapters of this thesis.

2.2 Characteristics Of Mobile Platforms

Mobile devices are energy constrained, memory-limited and delay intolerant. Reducing the operating time before requiring recharge is a significant design goal.

2.2.1 Power Consumption By Subsystems

Several subsystems including display illumination, radios, memory, and CPU all significantly contribute to the total power consumption. This thesis focuses on the power consumption by the CPU subsystem, which typically constitutes fifteen to twenty percent of

mobile devices' power budget [11].

The major subsystems(display, CPU, memory, radios) draw much less power when completely idle [19]. While power consumption rises with workload, this increase is not necessarily linear. For example, a GSM radio may be much more efficient transmitting a single large block of data than transmitting a lesser number of bytes in multiple smaller bursts [19].

A program execution speed generally increases sub-linearly with the CPU clock frequency due to limitations of the memory subsystem execution speed.

A mobile device's CPU can run at multiple clock frequencies. Android incorporates a governor that modulates CPU clock frequency. Its algorithm is configured to dynamically select the minimum CPU clock frequency permits the CPU to be idle at least 30% of the time. The rate of energy consumption over time generally increases quadratically with CPU frequency [12]. Therefore an increase in load that triggers an increase in CPU frequency can disproportionately increase executing programs' energy consumption.

2.3 Memory Management

Dynamic memory management enables programs to allocate variables whose lifetime and access patterns do not directly correspond to a program's control structure. This section describes the two families of dynamic memory management "explicit" and "automatic" memory management.

2.3.1 Explicit Memory Management

Explicit memory management relies on the programmer to indicate when to recycle individual objects. A familiar example of explicit memory management is the C language's *malloc* and *free* functions. The *malloc* function enables programs to allocate regions of memory and the *free* functions permits those regions to be explicitly freed when no longer needed.

Explicit memory management is a burden to the programmer and a significant source of programmer errors. These errors can be either due to premature recycling of objects that still need to be accessed or memory leaks due to inaccessible objects not being identified as recyclable [16] [15].

2.3.2 Automatic Memory Management

Explicit memory management benefits from the programmer's knowledge of program semantics for identifying recyclable objects. In contrast, automatic memory management systems perform additional computation to identify such objects [7].

Automatic memory management was first implemented by John McCarthy for the Lisp language during the late 1950's [9]. The key insight behind automatic memory management is that memory containing unreferenced objects can be recycled.

Java includes first class support for weak and soft *tentative* references intended to facilitate construction of object caches [8]. In order to satisfy a memory allocation request, Java's memory management subsystem may recycle memory containing objects only referenced by tentative references.

This additional computation in automatic memory management to identify the inaccessible objects can be divided in two basic phases:

1. *Garbage Identification*: Distinguishes accessible and inaccessible objects.

2. *Garbage Reclamation*: Reclaims memory allocated to inaccessible objects.

In automatic memory management (Garbage Collection) some properties must not be violated ("safety") and some are desirable. Those properties include

1. *Safety*: Objects that are accessible must not be reclaimed.

2. *Comprehensive*: All objects that can never be accessed again are recycled.

3. *Support tentative references*: Support for tentative references to objects that can be recycled when memory is limited. Java supports two classes reflecting two priorities soft and weak.

4. *Minimal interference*: Minimal interference between GC and mutator (program) activities.

There are two major approaches to garbage collection (GC): *reference counting* and *tracing* that represent different trade-offs among the desirable properties enumerated above.

2.4 Tracing Garbage Collection

Tracing is a comprehensive garbage collection technique to identify unreferenced memory regions. Dalvik's automatic memory management system implements a tracing algorithm that requires computation of transitive closure (TC) from the root set. Therefore tracing GC is not immediate, consumes significant amounts of energy, and can cause unacceptable pauses.

2.4.1 Mark-And-Sweep

Dalvik's automatic memory system implements a naive *mark-and-sweep* tracing algorithm without memory compaction. Since objects are never moved, the non-compacting collector may lead to a heavily fragmented heap. This is supported by our investigation described in Chapter 3.

The first *mark-and-sweep* algorithm was implemented by John McCarthy [9] in order to recycle allocated but unreachable memory regions. Like McCarthy's original algorithm, all garbage collectors "mark regions discovered in a search process in a program's current root set. The reachable objects are marked either by Boolean flags that are within a distinct "exogenous" structure or endogenously embedded within the objects themselves (see figure 2.1).

Once this marking process is complete, unmarked (and therefore unreachable) memory regions are identified as garbage and therefore available for subsequent (re)allocation (see figure 2.1).

The CPU computation time in mark-and-sweep garbage collector increases commensurately with the number of objects in the heap (therefore approximately with the heap size)[15][25].

The frequency and aggregate CPU time consumption of GCs are dependent upon heap size which expose several questions of how to reduce impact of GC computation.

1. Can GC trigger increase in CPU speed and therefore rate of energy consumption?
2. When to commence GC?
3. How to interleave GC computation with mutator execution?
4. Whether to perform GC over entire heap?
5. When to free tentatively referenced vars?
6. Smaller heap or less frequent GC?

2.5 Interleaving Of GC Computation And Mutator Execution

Interleaving of GC computation with the mutator execution exposes the need for coordination to ensure that mutators and GC threads concurrently accessing the same data structures are free from race conditions that can result in incorrect behavior. Incremental schemes can be single-threaded and thus can sidestep this coordination challenge by guaranteeing that critical GC and mutator operations are always executed to completion.

Techniques to permit GC to be performed concurrently or interleaved with the mutator (program) execution were discovered decades ago[6]. The naive *mark-and-sweep* algorithm

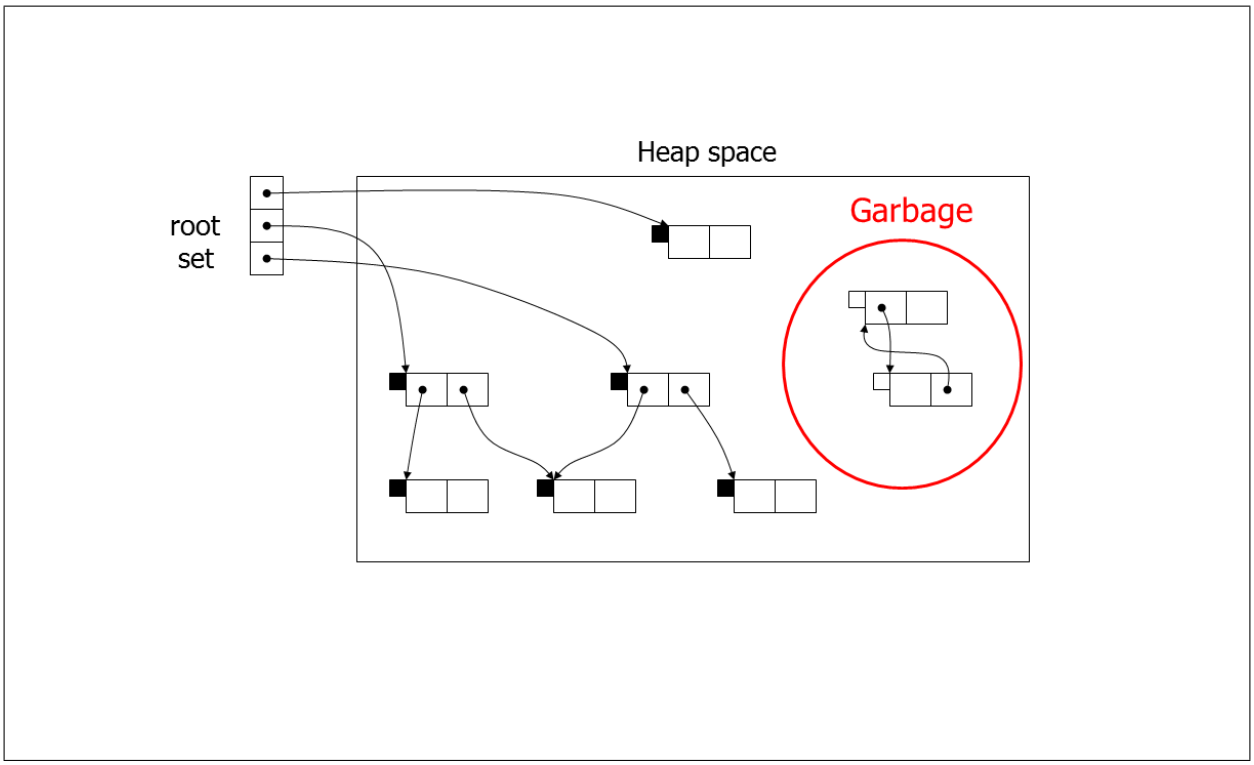

Figure 2.1: Tracing mark-and-sweep

avoids this coordination challenge by suspending the mutator threads during GC which is also known as *Stop the World* (STW) GC. Many automatic memory management systems including early Android version uses the STW GC. Since the mutator threads are dependent upon a (transiently) unsatisfied allocation, it is paused until a suitable memory region is identified - typically at the end of the GC computation. Therefore, the STW GC might cause unacceptable interface delay and may not be a good choice for delay intolerant mobile devices.

Concurrent garbage collectors conservatively identify unreferenced regions that were allocated prior to the commencement of the current GC and referred as BG GC. Incremental GC is explicitly scheduled by the language run-time system in a piecemeal fashion and is typically integrated into the allocation code. The primary difference between background and incremental approaches is in how the GC is scheduled. Background GC is implemented as a distinct thread that is scheduled by a conventional thread scheduler.

In order to determine the root set, BG GC initially scans all the active named variables. To avoid race conditions, mutator threads are suspended while this scan is performed. Recent Android releases implement BG GC. Our investigation indicates that this scan completes in ten to fifteen milliseconds, resulting in delays of short enough duration to be generally imperceptible by users.

Most of the modern mobile devices contain multi-core CPUs. If GC is implemented in a distinctly scheduled thread, a thread scheduler may be permitted to execute concurrently with mutator threads. However, this additional GC computation increases the work load and can trigger a governor to increase the CPU clock speed, increasing the energy required to execute both the mutator and the garbage collector.

2.6 When To Perform GC?

GC can be performed

1. *Reactively* when heap is exhausted and cannot satisfy an outstanding request. Requesting thread might pause until needed memory is discovered which can lead to unacceptable interface delay.

2. *Proactively* when triggered by some heuristic indicating a likely benefit from GC execution (e.g. The amount of free memory becomes less than a computed or preset threshold).

3. *Continuously* the garbage collector is repeatedly cycled whenever the CPU would otherwise be idle (i.e. all mutators are blocked). This policy aggressively reclaims memory containing unreachable objects at the cost of preventing the CPU from becoming idle.

2.7 Reducing GC Scope By Heap Partitioning

GC execution time (and power consumption) can be reduced if it is restricted to a portion of the heap. To achieve this, the memory management subsystem conservatively compiles lists of all objects within the heap partition(s) included in the restricted GC that may be referenced by objects within the partition(s) excluded from the restricted GC.

There are two main approaches to heap partitioning:

1. *Generational*: Exploits convenient property that many objects are short-lived. Newly created objects are allocated from a "non-tenured" partition. Long-lived objects are moved to a "tenured" partition. The scope of most GCs is restricted to the untenured partition.

2. *Regional*: Partitions are associated with different program activities. Memory is explicitly allocated from heaps associated with those activities, and garbage collection scope is frequently restricted to regions associated with recent program activities. Android exploits the partitioned heap properties to reduce apps start-up time by pre-loading common system objects into a copy-on-write partition shared among apps. Objects created after app start-up are allocated with a partition associated with the particular app [26].

2.8 Heap Headroom

The heap footprint is the total amount of memory (allocated and free bytes) being managed by the GC system. We describe the difference between footprint size and total amount of memory allocated to accessible objects as *heap headroom*.

Headroom can be computed at the completion of a GC. The maximum amount of memory that can be allocated before heap exhaustion is proportional to heap headroom.

This relationship can be exploited by heuristics within memory management subsystems to determine whether heap growth is needed to limit the frequency of GC. As discussed in

Chapter 3, the research described by this thesis examines problematic behavior caused by the heuristics related to headroom management in Dalvik and an alternative design with superior properties.

Heap size can dramatically impact the total execution time required for GC. Increasing a heap's headroom permits longer execution bursts between GC, thus potentially increasing the number of unreachable objects a heap contains that can be garbage collected. Therefore, greater headroom can increase the time required to perform GC.

Yang et al. [22] observe that the virtual memory system bottlenecks (related to TLB entry or page replacement for large heaps due to GC operations) lacks physical locality. This phenomena is irrelevant for mobile platforms that do not implement virtual memory such as Android. They also observe that GC on pathologically small heaps whose size barely satisfies program memory requirements cause "excessive GC overhead." This excessive overhead is due both to the GC's low yield and frequent execution in reaction to heap exhaustion events. Our experiments on Dalvik described in Chapter 5 support this latter observation and a complementary conjecture that overall GC CPU time (and therefore energy consumption) is inversely related to heap size provided that the entire memory footprint can be contained within the system's translation cache.

2.9 GC Scheduling And Efficiency

We define GC efficiency as yield over CPU time consumed by GC (which approximately corresponds to energy consumption). Stop the world maximizes yield by delaying GC until heap memory is exhausted [17] and is the most energy efficient GC policy. STW GC is executed on heap exhaustion and program execution is paused until GC completion. This strategy can result in prolonged interface delays inappropriate for delay-intolerant devices like mobile phones and tablets.

As described above, proactive execution of GC can prevent these interface delays. Ideally, this proactive GC execution will complete shortly before heap exhaustion. The penalty

for premature GC is reduced yield and shorter intervals before the heap is again exhausted. Conversely, if GC commences too late, the heap can become exhausted prior to GC completion, which can delay an allocation request and potentially cause noticeable interface pauses colloquially referred to as *stutter* [32].

Persson and Cummins [17] observe that opportunistic computation of GC prior to heap exhaustion consumes CPU resources that may be needed by other processes or threads, and therefore can interfere with overall system progress. Independently scheduled GC threads from distinct services and apps further compounds this complexity by exposing the risk of priority inversion when the number of ready execution streams (including GC threads) exceeds available concurrency.

Various efforts have examined the prediction of heap exhaustion time and ideal time to begin GC such as Grose et al's patent on the algorithm to alert operators about imminent memory exhaustion [24] and Boehm's characterizations of the time required to execute GC [25]. IBM's WebSphere [17] and Oracle's Hotspot [18] contain GC schedulers that predict heap exhaustion but primarily target platforms with far less sensitivity to heap growth and power consumption than mobile phones

2.10 Limiting The Need For GC

The need for garbage collection can be reduced by (1) compiler optimizations that identify allocation requests that can be allocated and reclaimed programmatically [20], and (2) the integrating of reference counting with GC schemes to identify the common case of unreferenced acyclic data structures [21].

2.11 Need For Self Optimization

Mobile systems are frequently managed by users with limited technical understanding who install and execute apps with a wide variety of memory behaviors and performance con-

straints. Ideal performance from the resulting complex software ecosystem requires the determination of appropriate management strategies and parameter settings that may vary with the combination of apps executing at a particular time.

Multiple approaches to self optimization have been examined.

The Internet is highly dependent upon various adaptive protocols dependent only on locally available data that (generally) achieve stable behavior and high overall performance.

IBM's autonomic computing thrust [23] takes a complementary approach. They identify the fixed parameters within static configuration as a particularly troublesome source of system fragility. They advocate for a structured optimization approach where individual components of complex systems explicitly expose their requirements, configuration options, and control parameters in a fashion that facilitates off- and on-line analysis, configuration, and tuning.

Dalvik implements a single memory management strategy described in Chapter 3 that is driven by a few static parameters. The preliminary evaluation from our studies (described in Chapter 5) indicate that our adaptive approaches are effective.

Chapter 3

Android Dalvik's Garbage Collection Policy

3.1 Chapter Overview

This chapter describes Android Dalvik's baseline GC policy and its performance implications. This chapter begins with a description of Gingerbread's (Android version 2.3) baseline GC policy for scheduling garbage collection and heap growth management strategy. Our experiments (described in Chapter 5) indicate that this baseline policy results in the following undesirable behavior:

- Frequent low yield GCs.
- Significant fraction of CPU time spent on GC.

This undesirable behavior motivates the design of alternate policies described in Chapter 4.

3.2 History

The releases of Android prior to Gingerbread in 2010 employed only STW mark & sweep garbage collection which is triggered by memory allocation failures. Pauses caused by these foreground garbage collection cycles caused noticeable interface stalls that are commonly referred to as "stuttering" [32].

This undesirable behavior motivated Gingerbread's incorporation of BG GC. As is common for BG GC, Gingerbread's background GC is implemented by a (generally) low priority thread that is opportunistically scheduled during periods that the CPU (or a core) is available. When BG GC is complete, this thread is blocked until a triggering event (described below) is detected by a memory allocation operation.

3.3 Dalvik's Partitioned Heap

Dalvik's heap is partitioned into two regions in order to expedite application startup time and to promote memory sharing among apps. The progenitor of all Dalvik processes is named Zygote. Zygote loads all the common system packages into a heap named "Region 1" during system initialization. "Region 1" is shared with processes running Dalvik apps via copy-on-write. After startup, apps allocate new objects from a second heap named "Region 0."

3.4 Heap Footprint Size Management

Dalivk utilizes a modified version of Doug Lea's "dlmalloc" best-fit allocator [28]. When the current heap is unable to satisfy a request, the baseline policy will permit this allocator to extend the heap's footprint size up to a computed *soft* limit (softlimit).

The value of this softlimit is a critical component of the baseline policy and is adjusted both reactively when an allocation attempt fails and proactively to enable limited heap growth. Apps can indirectly influence the management of softlimit through a heap utilization factor (HUF) described in Subsection 3.4.2

3.4.1 Reactive Heap Growth

When heap exhaustion is detected by a failed allocation attempt, the soft limit is raised as needed in the states "3A" and "4A" of Figure 3.1.

This reactive limit adjustment is implemented by a process of temporarily escalating softlimit to a device-dependent absolute maximum footprint size (hardlimit, which is 32MB for the handsets we examined) followed by an allocation attempt. Afterwards, softlimit is reduced to the actual heap size after the allocation.

3.4.2 Proactive Heap Growth

Following each GC, the current ratio of heap free and allocated memory is checked and the heap soft limit is grown to maintain the predefined heap utilization factor (see states 2G and 3G in Figure 3.1). Apps can specify a heap utilization factors (HUF) which has a default value of 0.5 and is clamped within the range of 0.2-0.8 The HUF thus limits total heap footprint growth permitted without GC *beyond total allocated memory size* to 20-33% of the current allocation size (CurAlloc). The permitted growth beyond allocated size is also constrained to the range of 256kB and 2MB. Since these fractions and limits are relative to allocated (not footprint) size, this permitted growth amount may actually be zero due to external fragmentation.

The heap current soft limit and absolute limit, described in the previous Section 3.4, are defined as CurLimit and AbsLimit respectively. More formally, upon GC, the soft limit is set to

$$\max(CurLimit, \min(AbsLimit, CurLimit{+}2\text{MB}, \max(CurAlloc{*}\frac{1}{HUF}, CurLimit{+}256\text{KB})))$$

3.5 Baseline GC Scheduling Policy

Android versions Gingerbread (and later) employ both foreground and background GC:

- Foreground GC is triggered reactively upon heap exhaustion, and
- Background GC is scheduled proactively.

The baseline policy triggers foreground GC in the following conditions

- *Case 1: (Allocation Request)* Triggers STW GC when an allocation request fails.
- *Case 2: (Explicit Request)* Triggers STW GC when the apps explicitly invoke method *Runtime.GC()* or *System.GC()*) to perform GC.

Background GC is scheduled proactively when the amount of unallocated memory within the heap falls below 128kB [29]. This amount is only reduced by allocation, and it is checked only at those times [29].

3.6 Dalvik's Baseline GC Policy For Allocation

Dalvik's baseline policy will commence garbage collection as a result of a memory allocation request that either (1) reduces the amount of free memory below a fixed minimum free-space threshold (128kB), or (2) cannot be fulfilled without growing the heap beyond its current heap soft limit. The former case results in a background (BG) GC execution, and the latter results in STW GC.

GC is managed by a routine named *trymalloc* that attempts to allocate memory as requested and applies remediation on failure. The remediations sequence of action: (weaker to stronger)

1. Wait for BG GC completion (see state "1W" in Figure 3.1).
2. Perform GC clearing weak references (see state "2G" in Figure 3.1).
3. Grow heap (see states "3A" and "4A" in Figure 3.1).
4. Perform GC clearing both weak and soft references (see state "3G" in Figure 3.1).

From the policy flow chart (see Figure 3.1), the baseline policy effectively implements a strategy of frequent garbage collection that aggressively limits heap growth. This strategy may be appropriate when memory is severely limited such as was common in early Android handsets.

Our examination suggests that this policy is inappropriate for later devices with more generous memory sizes since a single heap's footprint is a small fraction of RAM. Footprint size for several apps including Gmail, a game called Modern War, Google Maps, and Camera are examined. They are all below 3% (see heap Footprint in Figure 5.3) of the 512MB of RAM installed even within the least expensive Android phone marketed by T-Mobile in June 2013. Furthermore, Dalvik imposes an absolute max heap size limit of 32-48MB [29], which is 6-9% of RAM size 512MB.

In the baseline policy, background garbage collection is scheduled when free memory falls below 128kB [29]. This constant low-memory threshold that triggers BG GC appears to be problematic. As indicated by rows labeled *Number of STW GC%* of Table 5.5, the majority of GC operations performed by Dalvik's baseline policy are STW GC. We observed repeated heap exhaustion events triggering STW GC while the heap had more than 128 kB of free space due to internal fragmentation.

As described in Section 3.4.2, proactive heap footprint growth is limited. As illustrated in Figure 5.3, external fragmentation of the heap is sufficient for *Gmail* and *Camera* apps to prohibit proactive growth (and therefore cause frequent low-yield GC operations) after 5 seconds of CPU time.

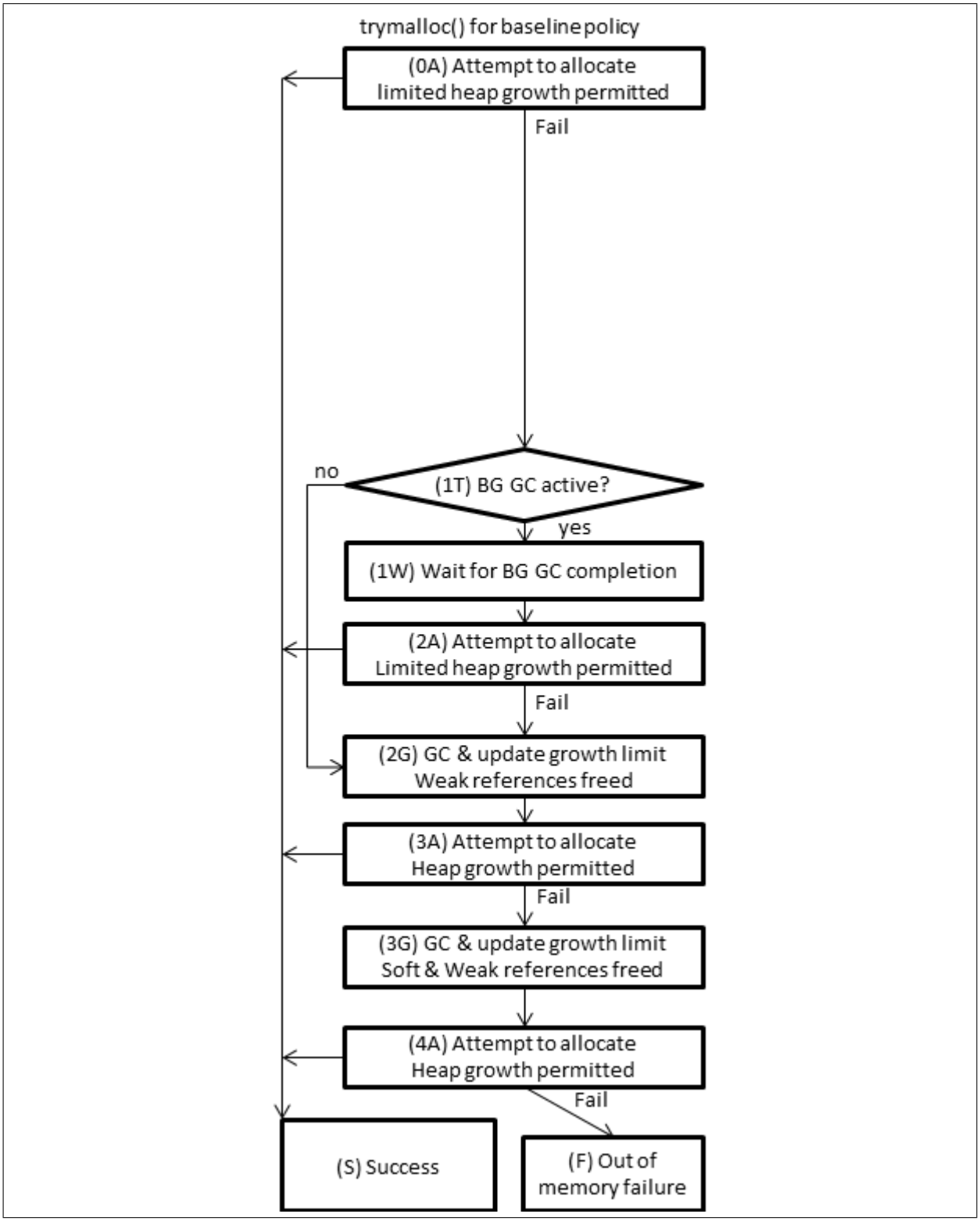

Figure 3.1: Baseline GC Policy

Chapter 4

Alternative Garbage Collection Policies

4.1 Chapter Overview

This chapter describes three alternative GC policies in the context of the baseline GC policy (described in Chapter 3) and their performance implications. The first two are called *exploratory* because they are used in initial sensitivity studies examining the impact of enforcing minimum GC interval upon heap growth.

Results from these studies motivate the design of a third *adaptive* GC policy with properties superior to both the baseline and exploratory policies. Our experiments (described in Chapter 5) indicate that the alternative adaptive GC policy reduces both the CPU time spent on GC and the user interface stalls with reasonable heap growth, compared to baseline policy.

4.2 Exploratory GC Policies

As described in Chapter 3, frequent low yield GCs in the baseline policy consume substantial amount of CPU time. These frequent GCs can also have negative impact on the performance of user interface. In order to reduce both the CPU time executing GC and user interface delay, the exploratory GC policies (MI2 and MI4) enforce a minimum inter-GC interval of at least two or four seconds respectively. While restricting the GC interval, the exploratory policies examine the sensitivity of heap size to GC frequency (and CPU

time/power consumption).

The results of this sensitivity study indicate that the exploratory GC policies reduce total GC CPU time with reasonable heap growth. (See exploratory GC policies (MI2 and MI4) in Figure 5.4 and Figure 5.3). The additional heap growth by the exploratory GC policies (1) decreases the total number of GC compared to the baseline policy, and (2) increases the execution time per GC. (CPU time for a single GC is roughly proportional to heap size [15].) Consequently, higher average STW GC pause time is observed in the exploratory GC policies (see Avg STW in Tables 5.1-5.4). Since the GC frequency is decreased with the increase of execution time per GC, the exploratory policy may cause longer (but less frequent) user interface stalls. This user interactive performance degradation motivates the development of an adaptive GC policy. The next section describes how the adaptive GC policy resolves this issue.

In order to examine the impact of enforcing minimum GC interval upon heap growth, two variants of exploratory policies are implemented and evaluated:

1. *MI2:* Enforces 2 seconds minimum interval between GCs.

2. *MI4:* Enforces 4 seconds minimum interval between GCs.

The difference between exploratory and baseline GC policy appears when allocation fails within predefined minimum interval time since the last GC (see state "0T" where T, T_{GC}, and MI are defined as the current time, last GC occurrence time, and minimum interval time respectively) in Figure 4.1). In this case (allocation failure), the exploratory GC policy grows the heap limit where the baseline policy performs STW GC.

4.3 Alternative Adaptive GC Policy

As described in the previous Section 4.2, the exploratory GC policy reduces total GC CPU time with the cost of heap growth and longer (less frequent) user interface stalls. Like exploratory GC policy (MI2), the adaptive GC policy (MI2A) also restricts total GC CPU

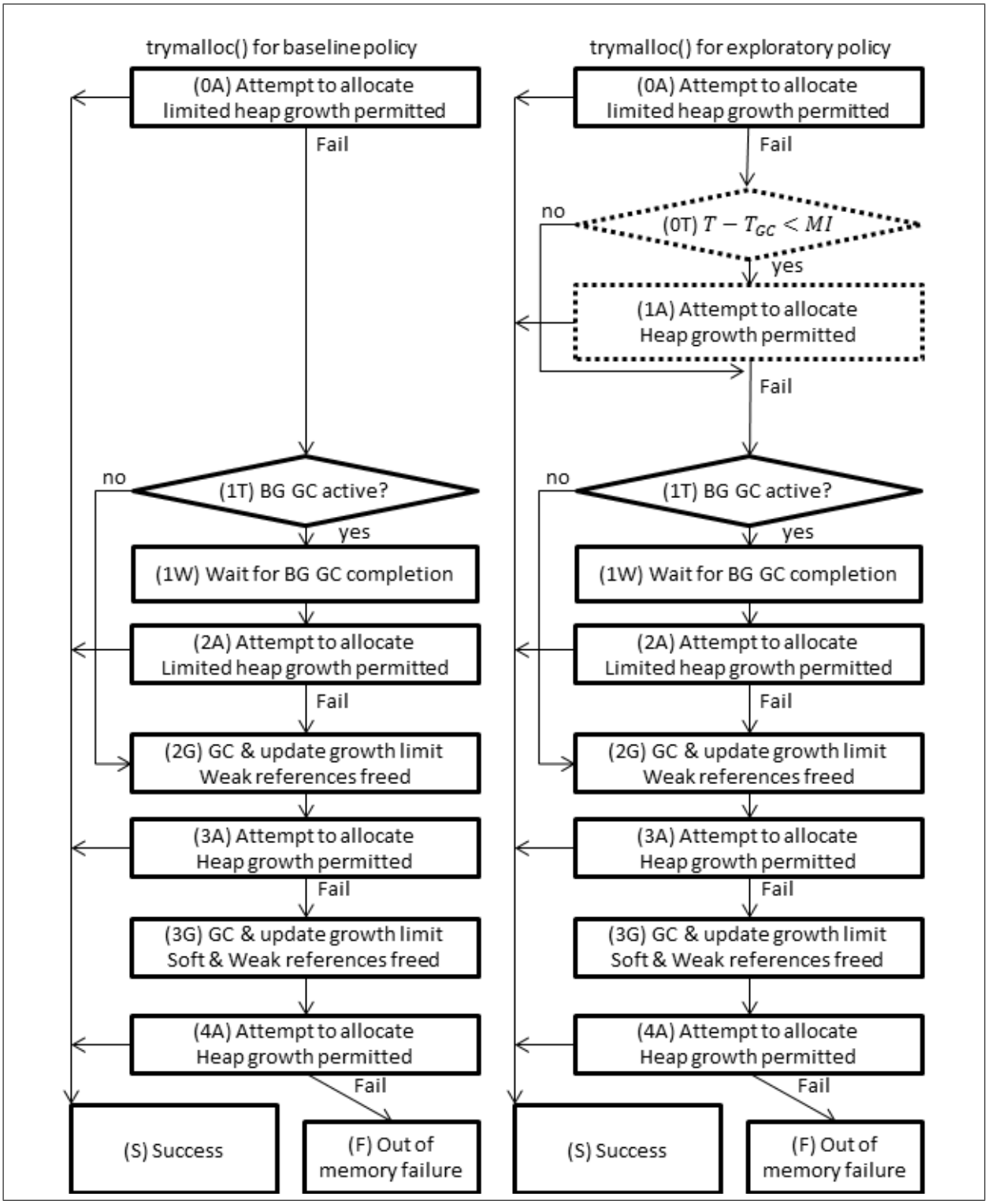

Figure 4.1: Baseline Vs. Exploratory GC Policy

time by applying minimum GC interval (2 seconds) and therefore results in larger heap size. In order to reduce the user interface delay, the adaptive GC policy never triggers STW GC unless the heap grows to the absolute max limit (32MB) (see state for MI2A "0T" and "1A" in Figure 4.2). As indicated by the number of STW GC% in the Table 5.6, the adaptive GC policy achieves its goal to eliminate STW GC since no apps' heap grows to the absolute max limit. The adaptive GC policy is intended to trigger only the BG GC in order to reduce the user interface delay. However, the background GC can pause the apps for 10-12ms during its root set scanning. This small STW event's duration is smaller than the screen refresh interval (17ms, measured using FPS Test apps). Therefore, BG GC may not cause noticeable interface stall.

As described in Chapter 3, a constant free-space threshold (128Kb) is used to trigger BG GC by unblocking the BG thread in the baseline GC policy. This constant threshold turns out to be ineffective due to internal fragmentation (described in the Section 3.6). In order to avoid this problem, the adaptive GC policy uses adaptive free-space threshold which is calculated by utilizing recent history of the heap free memory. As indicated by the prediction success rate in Table 5.6, 83 to 94% of the total number of BG GC (scheduled by this adaptive free-space threshold) complete prior to the heap exhaustion. Approximately 6-17% of the total number of GC fail to complete before heap exhaustion and grow heap reactively. Upon heap exhaustion, the adaptive policy performs several operations:

1. Always performs reactive heap growth (as described in Chapter 3)
2. Triggers BG GC if the time since GC is at least 2 seconds.
3. Updates free-space threshold if it is first allocation failure since GC. (see states "0T" and "0U" for adaptive policy in Figure 4.2)

Heap internal fragmentation may cause allocation failure even though a large amount of memory is free in the heap. In the baseline GC policy, repeated heap exhaustion events are observed while the heap had more than the free-space threshold (128Kb) due to internal

fragmentation. This problem motivates the adaptive GC policy to apply periodic measurements of heap free space to conservatively predict the adaptive free-space threshold. In prior experiments, BG GC operations are observed to reliably complete in less than 375 ms(see the Max BG GC duration for the baseline GC policy in Table 5.4). The adaptive GC policy is intended to conservatively schedule a BG GC *500ms* prior to the (expected) heap exhaustion (thus leaving 125 ms of buffer time). To achieve this, heap free space is tracked at 100ms interval, and the amount of free space approximately 500ms prior to the first allocation failure since GC is used as a free-space threshold to trigger subsequent BG GC.

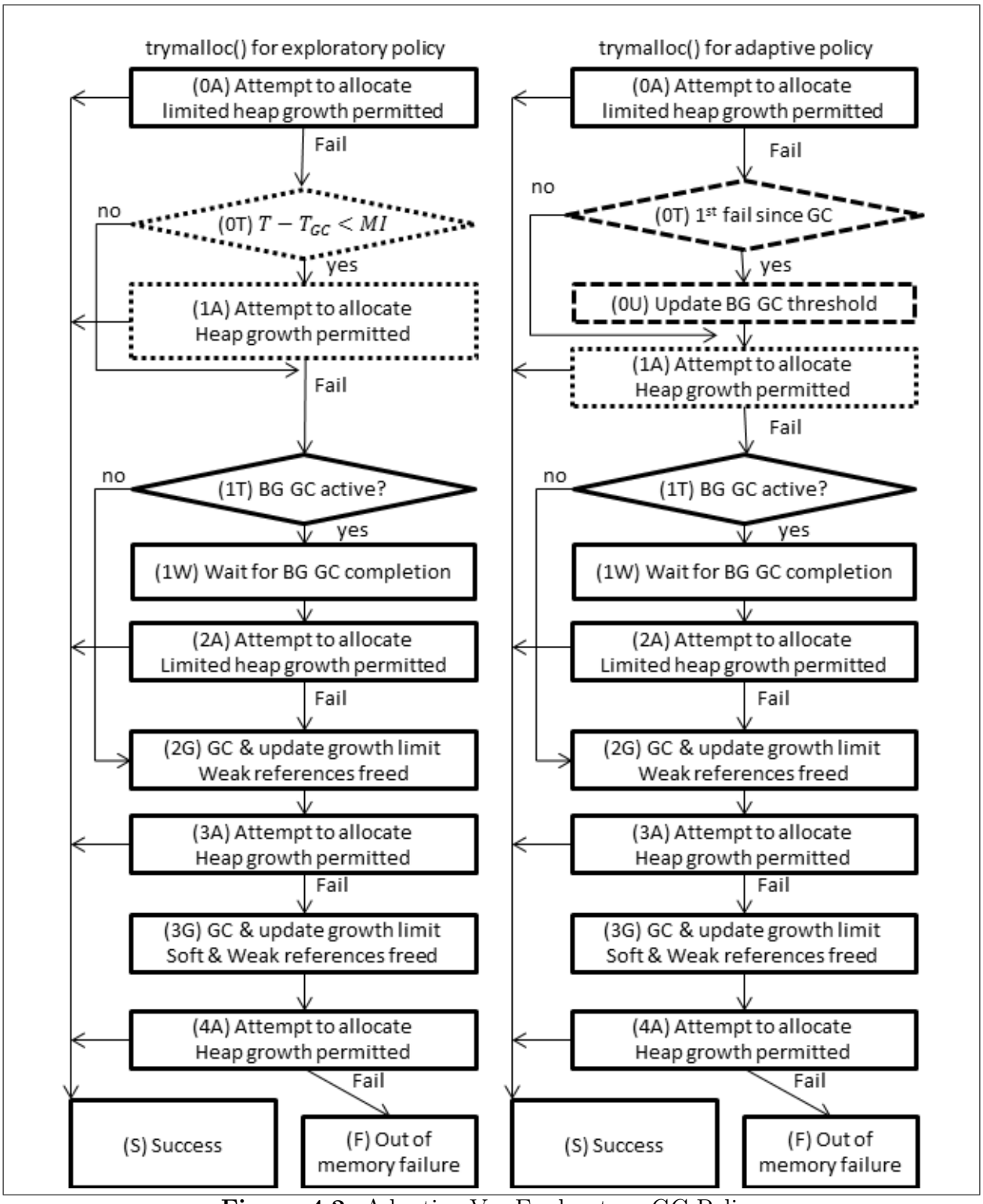

Figure 4.2: Adaptive Vs. Exploratory GC Policy

Chapter 5

Evaluation

Android's Dalvik (J)VM includes fine-grained event logging facilities primarily intended to aid apps developers in instrumenting, tuning, and debugging program behavior [30]. It causes sufficient overhead that its documentation recommends limiting its operation to short intervals. Since the focus of our research is resource utilization on a resource-constrained device, an alternate low-overhead logging mechanism that minimizes interference with normal system operation was implemented. To minimize system load, logged data is buffered and infrequently transferred to files stored within the on-board SSD persistent storage for offline analysis.

The experiments were conducted using a *Samsung Galaxy S Vibrant* running Cyanogenmod 7 (*Gingerbread*, v 2.3 Android). Among all the Android versions, the *Gingerbread* version is used in the highest number of devices. More than 1/3 of the total Android devices are using *Gingerbread* [27] and the GC policy of the current Android release (4.2, *Jellybean*) is nearly identical to Gingerbread's release. In order to provide an approximately consistent system configuration, the device was rebooted and left idle for one minute to initialize prior to each experiment. To determine sensitivity to user timing, the experiments were repeated at various rates of user input including deliberately slow interactions (e.g. taking of one photograph every 10 seconds). Instrumented behavior did not differ significantly between these runs. (see Figure 5.1 and Figure 5.2)

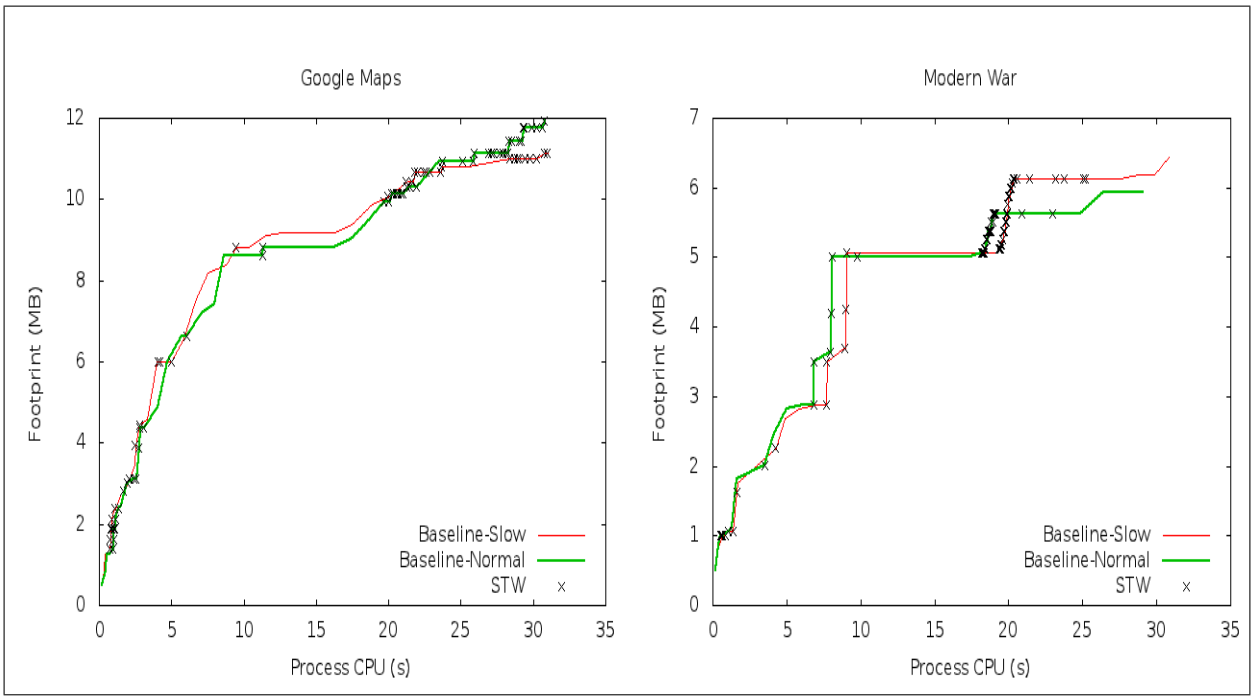

Figure 5.1: Heap Memory Footprint Slow Vs. Normal User

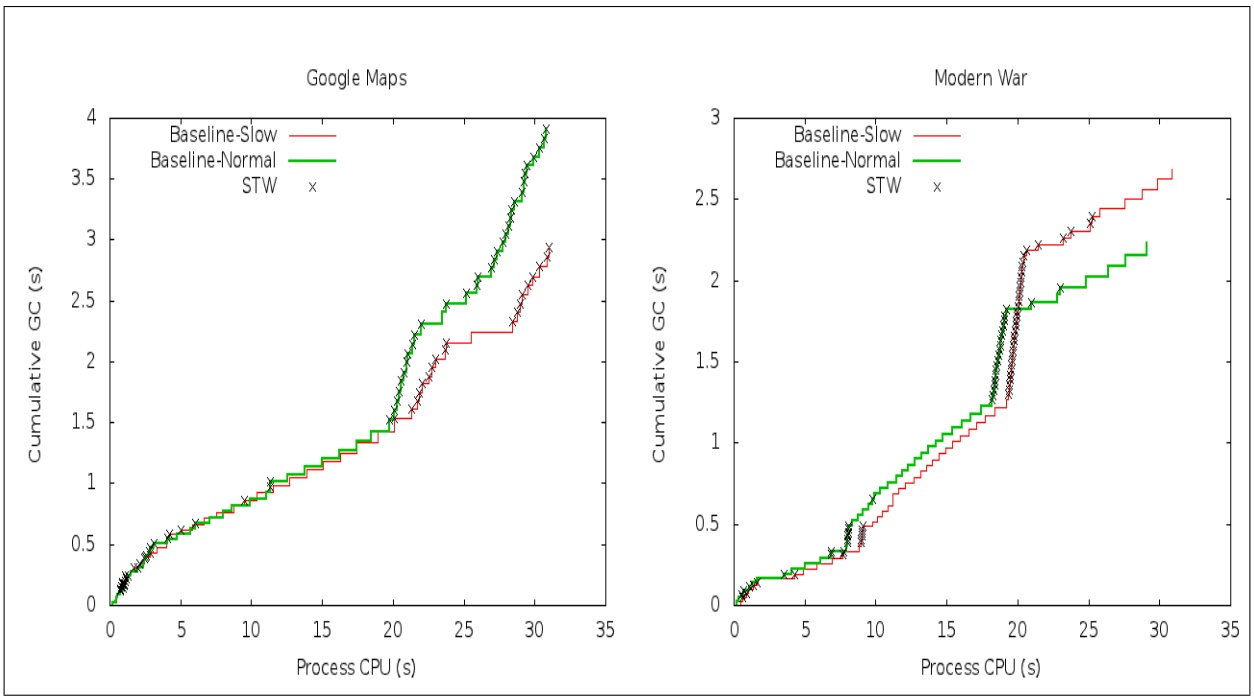

Figure 5.2: Total CPU time executing GC Slow Vs. Normal User

5.1 Descriptions Of Apps Used In Experiments

Our experiments examined memory behavior of the following collection of apps reflecting a variety of common classes:

- Google maps: This program requires frequent network communication and frequent network communication is a critical component when navigating to an uncached path of map. The same destination and point of origin were inputted, followed by the user traversing the path of the directions returned by the program.
- Camera: Memory and cpu usage patterns of this program are bursty. Twenty-five sequential photos were taken by the user, at the rate of approximately 2-3 seconds between photos.
- GMail: This program is generally blocked between user interactions. Fifteen e-mails were opened, scrolled through by the user, and closed, imitating the normal use of the Gmail.
- Modern War: This program (game) updates state frequently without input. This interactive game was played normally.

Data collected from those experiments are tabulated in Tables 5.1-5.4 . Figures 5.3, 5.4, and 5.5 illustrate heap footprint size, total CPU time spent in GC, and total application pause time due to GC as a function of app CPU time. The remainder of this section describes these experiments and results.

5.2 Results Analysis

The plots in Figure 5.3 indicate heap growth as a function of CPU time (for all apps from collection). The other plots in Figure 5.4 and Figure 5.5 indicate cumulative GC CPU time and cumulative STW pause time respectively as a function of CPU time. X-tic mark is used to indicate STW GC for all the plots.

The baseline GC policy generally favors STW GC over heap growth, and achieves consistently smaller heap size. As indicated by the Figure 5.3, the minimal heap size is achieved in the baseline policy. In the baseline policy, the maximum heap size is observed approximately 12MB for the Google Maps apps which is 2.3% of total memory (512MB) (see % of Total Memory for GMaps in Table 5.1-5.4). The baseline policy's overprotective approach (in regards to heap size) results in frequent low yield GCs. Normally low average GC interval (approximately 0.720 to 1 seconds) is observed in the baseline GC policy (indicated by the Table 5.1-5.4). Due to this low GC interval, around 4-20% of GC is observed to yield low (less than 1KB) (see number of GC yields in Table 5.5).

The frequent (and low yield) GCs consume of high fraction of apps CPU time in the baseline GC policy(as high as 26% for the Camera in Table 5.1-5.4) and introduce user interface delay. As indicated by the Table 5.5, 65-75% of GC is STW in the baseline policy.

As described in the Section 2.5, STW GC suspends all other threads during its operation. Therefore STW GC execution duration may contribute to the user interface delay. In the baseline GC policy average STW GC operation duration is 28-60ms except for the Camera (see Avg STW for baseline in Table 5.1-5.4). Though the Camera appears to have low (16 ms) average STW GC execution time, it enters prolonged degenerate configurations where the baseline GC policy results in repeated low yield GC that consume a substantial fraction of apps CPU time (26 %).

In order to reduce CPU time spent on GC, the exploratory GC policies enforce a minimum GC interval. As indicated by Figure 5.4, the exploratory policies lower overall GC CPU time by a factor of 3-4.

In the exploratory GC policies, heaps are 1.5-4 times larger than for baseline. The largest measured heap is 16MB, 3.1% of total physical memory 512MB (see % of Total Memory for Modern War in Table 5.1-5.4) and average STW GC duration increased by a factor of 1.5-2 (see Avg STW in Table 5.1-5.4). This increased duration of STW GC is observed and results in noticible interface pauses.

As described in Section 4.3, the adaptive GC policy favors heap growth over interface

delay. Since none of the heaps grow to the the 32MB hard limit on heap size, no STW GCs occurs.

Though this adaptive GC policy favors heap growth, the maximum heap size is observed approximately 12MB for GMaps (See MI2A in Figure 5.3) which is a small fraction (2.3 %) of a total RAM size of 512MB (see % of Total Memory for GMaps in Table 5.1-5.4). The adaptive GC policy reduces the GC CPU time by a factor of 2-4 compared to the baseline GC policy (see adaptive (MI2A) GC policy in Figure 5.4). Since the CPU consumes 15-20% of total Android phones power [11], the adaptive policy reduces approximately 4% of total power consumption for the CPU intensive apps like camera (see % GC CPU for Camera in Table 5.2).

As indicated by prediction success rate in Table 5.6, 83-94% of the total BG GC is completed without growing heap. The experiment's results indicate that adaptive policy achieves its goal to reduce both CPU time consumption and user interface delay with reasonable heap growth.

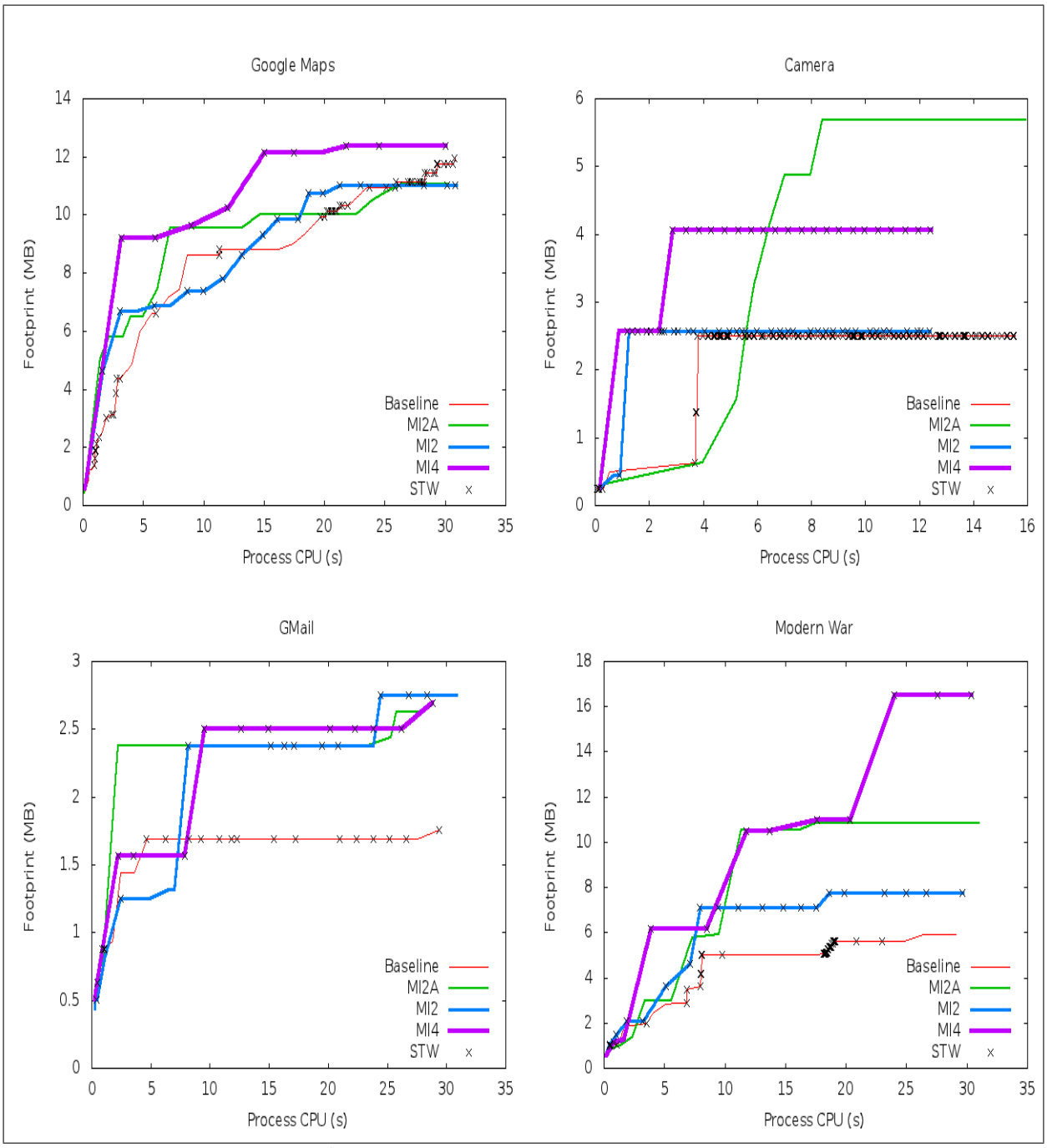

Figure 5.3: Heap Memory Footprint

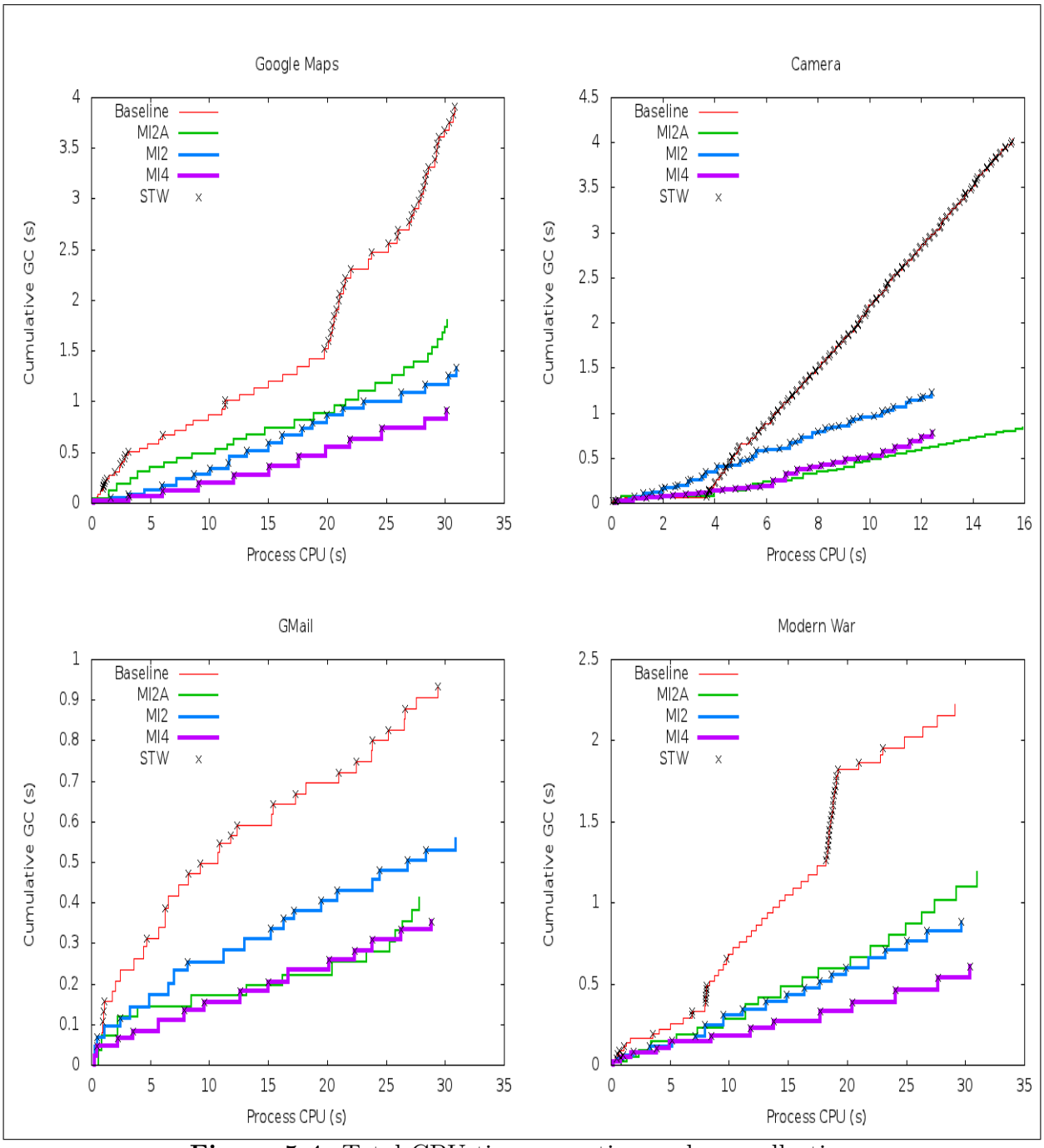

Figure 5.4: Total CPU time executing garbage collection

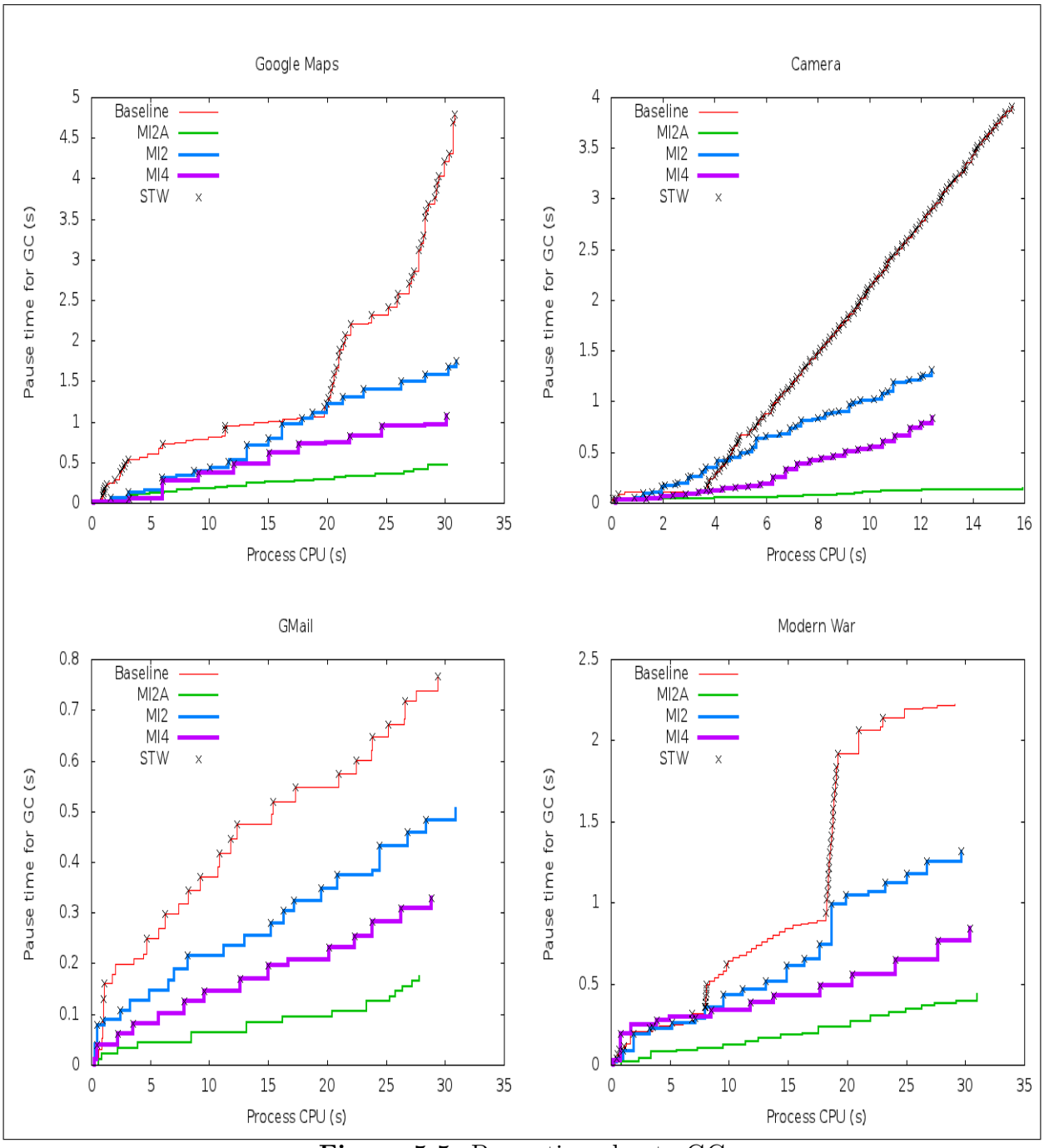

Figure 5.5: Pause time due to GC

Table 5.1: Memory Management Statistics For Google Maps

Application: Google maps navigation		Baseline	MI2	MI4	MI2A
Test Duration	App CPU Time (s)	30.3	30.9	30.1	30.2
	Experiment Duration (s)	74	52.8	51.9	90
GC	Avg GC interval (s)	1.1	2.4	4.33	3.11
	GC % CPU	12.4	4.3	3.04	5.94
STW	Min STW (ms)	18	27	39	0
	Max STW (ms)	95	91	110	0
	Avg STW (ms)	59	63	78	0
	Median STW (ms)	69	66	84	0
BG GC Duration	Avg BG GC (ms)	120	106	135	NA
	Max BG GC (ms)	289	124	181	NA
Heap Footprint	Max Heap Footprint (MB)	12	11.26	12.6	11.33
	% Of Heap Max Limit (32MB)	37.5	35	38	35
	% Of Total Memory (512MB)	2.3	2.1	2.4	2.1

Table 5.2: Memory Management Statistics For Camera

Application: Camera		Baseline	MI2	MI4	MI2A
Test Duration	App CPU Time(s)	15.5	12.8	12.4	15.9
	Experiment Duration (s)	266	106	110	324
GC	Avg GC interval (s)	1.25	2.5	4.24	9.55
	GC % CPU	26	9.9	6.2	5.25
STW	Min STW (ms)	14	14	14	0
	Max STW (ms)	21	60	70	0
	Avg STW (ms)	15.5	29	30	0
	Median STW (ms)	15	21	23	0
BG GC Duration	Avg BG GC (ms)	49	110	66	NA
	Max BG GC (ms)	103	128	66	NA
Heap Footprint	Max Heap Footprint (MB)	2.56	2.62	4.16	5.8
	% Of Heap Max Limit (32MB)	8	8	13	18
	% Of Total Memory (512MB)	0.5	0.5	0.8	1.1

Table 5.3: Memory Management Statistics For Gmail

Application: Google Mail		Baseline	MI2	MI4	MI2A
Test Duration	App CPU Time (s)	31	30.8	31.7	27.8
	Experiment Duration (s)	316	118	77	86
GC	Avg GC interval (s)	8.5	5.6	4.83	5.4
	GC % CPU	3.15	1.8	1.1	1.48
STW	Min STW (ms)	18	19	19	0
	Max STW (ms)	49	26	26	0
	Avg STW (ms)	25	22	22.8	0
	Median STW (ms)	25	24	24	0
BG GC Duration	Avg BG GC (ms)	85	97	77	NA
	Max BG GC (ms)	275	150	104	NA
Heap Footprint	Max Heap Footprint (MB)	1.8	2.8	2.75	2.69
	% Of Heap Max Limit (32MB)	5.6	8.7	8.5	8.4
	% Of Total Memory (512MB)	0.3	0.5	0.5	0.5

Table 5.4: Memory Management Statistics For Modern War

Application: Modern War		Baeline	MI2	MI4	MI2A
Test Duration	App CPU Time (s)	31	33.7	30	30.9
	Experiment Duration (s)	48	139	48	52
GC	Avg GC interval (s)	0.72	6.6	3.75	2.38
	GC % CPU	7.26	2.77	2	3.8
STW	Min STW (ms)	19	17	27	0
	Max STW (ms)	46	68	77	0
	Avg STW (ms)	28	42	52	0
	Median STW (ms)	30	43	50	0
BG GC Duration	Avg BG GC (ms)	105	122	191	NA
	Max BG GC (ms)	375	151	409	NA
Heap Footprint	Max Heap Footprint (MB)	6	7.9	16	11
	% Of Heap Max Limit (32MB)	18	24	50	34
	% Of Total Memory (512MB)	1.1	1.5	3.1	2.1

Table 5.5: Memory Management Statistics For Baseline Policy

Baseline Policy	Camera	GMaps	GMail	ModernWar
% of GC that are STW	65	74	69	56
% of Low Yield GC (<1kB)	20	3.4	7.2	14

Table 5.6: Memory Management Statistics For Adaptive Policy

Adaptive Policy (MI2A)	Camera	GMaps	GMail	ModernWar
% of GC that are STW	0	0	0	0
% of Low Yield GC (<1kB)	0	0	0	0
Prediction Success Rate%	83	87	93.75	90.7

Chapter 6

Conclusion

Our analysis of Dalvik's baseline GC policy indicates that it effectively minimizes heap size at the expense of CPU time, and to some extent, interactive responsiveness. Mobile devices are memory-limited and delay intolerant. However, device cost, weight, environmental impact, and operating time before requiring recharge are also important consideration. Extended capacity batteries contribute to cost, weight, and toxic waste. Our experimental results indicate that the battery capacity required can be reduced through a better garbage collection policy that consumes a limited amount of additional memory already available on most currently produced mobile handsets, with the secondary benefit of improved interactive responsiveness.

References

[1] G. Chen and R. Shetty and M. Kandemir and N. Vijaykrishnan and M. J. Irwin and M. Wolczko, "Tuning Garbage Collection in an Embedded Java Environment," *In Proc. the 8th International Symposium on High-Performance Computer Architecture*, IEEE Computer Society Washington, DC, 2002, pp. 2–6.

[2] U. D. of Energy, "Quick start guide to increase data center energy efficiency Tech," US Department of Energy, Tech. Rep., 2010.

[3] Wikipedia, "Dalvik (Software)," https://en.wikipedia.org/wiki/Dalvik_(software)

[4] Google, "Android devices activation statistics," http://www.inquisitr.com/621597 /google-android-shipping-1-5-million-devices-each-day-1-billion-by-december/.

[5] O. Svanfeldt-Winter, S. Lafond, and J. Lilius, "Cost and energy reduction evaluation for arm based web servers," *Proceedings of the 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, IEEE Computer Society Washington, DC, USA, 2011, pp. 480–487.

[6] Hans-J. Boehm, Alan J. Demers, Scott Shenker, "Mostly parallel garbage collection," *Proceeding PLDI '91 Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, ACM New York, NY, USA, 1991, pp. 157–165.

[7] Matthew Hertz, Emery D Berger, "Automatic vs. explicit memory management: Settling the performance debate," *Technical Report CS TR-04-17, University of Massachussetts Department of Computer Science*, MA, USA, 2004.

[8] Kevin Donnelly, J. J. Hallett, Assaf Kfoury, "Formal semantics of weak references,"

ISMM '06 Proceedings of the 5th international symposium on Memory management, ACM New York, NY, USA, 2006, pp. 126–137.

[9] Wikipedia source, "Garbage Collection process in computer science," http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

[10] Wikipedia source, "Android OS," http://en.wikipedia.org/wiki /Android_operating_system

[11] Aaron Carroll, Gernot Heiser, "An analysis of power consumption in a smartphone," *Proceeding USENIXATC'10 Proceedings of the 2010 USENIX conference on USENIX annual technical conference* , USENIX Association Berkeley, CA, USA, 2010, pp. 21–21.

[12] Nikzad Babaii Rizvandi, Javid Taheri, Albert Y. Zomaya, Young Choon Lee, "Linear Combinations of DVFS-Enabled Processor Frequencies to Modify the Energy-Aware Scheduling Algorithms," *Proceeding CCGRID '10 Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, IEEE Computer Society Washington, DC, USA, 2010, pp. 388–397.

[13] Wikipedia source, "Android Version Hisotry," http://en.wikipedia.org/wiki /Android_version_history

[14] Wiki, "Common Language Runtime(CLR)," http://en.wikipedia.org/wiki/Common _Language_Runtime

[15] Paul R. Wilson, "Uniprocessor Garbage Collection Techniques," *in IWMM '92 Proceedings of the International Workshop on Memory Management*, Springer-Verlag London, UK, 1992, pp. 1–42

[16] P. Savola, "Lbnl traceroute heap corruption vulnerability," http://www.securityfocus.com/bid/1739

[17] Persson and Cummins, "Java technology," *IBM style: Garbage collection policies, IBM developerWorks (web)*, 2006.

[18] O. Inc, "Java SE 6 HotSpot[tm] virtual machine garbage collection tuning," http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html

[19] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system call tracing," *Proceedings of the sixth conference on Computer systems*, ACM New York, NY, USA, 2011, pp. 153–168.

[20] Jaies E. Hicks and James Hicks and James E. Hicks, "Compiler-Directed Storage Reclamation Using Object Lifetime Analysis," *Technical report*, 1992.

[21] Rifat Shahriyar, Stephen M. Blackburn, Daniel Frampton, "Down for the count? Getting reference counting back in the ring," *Proceedings of the 2012 international symposium on Memory Management*, New York, NY, USA, 2012, pp. 73–84.

[22] Ting Yang, Matthew Hertz, Emery D. Berger, Scott F. Kaplan, J. Eliot B. Moss, "Automatic heap sizing: taking real memory into account," *Proceedings of the 4th international symposium on Memory management*, ACM, New York, NY, USA, 2004, pp. 61–72.

[23] Kephart, Jeffrey O. and Chess, David M. , "The Vision of Autonomic Computing," *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, January 2003, pp. 41–50.

[24] Vanessa Grose, John Nistler, "Automatic prediction of future out of memory exceptions in a garbage collected virtual machine," Patent, 2005.

[25] Boehm, Hans, "Web notes on GC Complexity," http://www.hpl.hp.com/personal/Hans_Boehm/gc/complexity.html

[26] Yunan He, Chen Yang, Xiao-Feng Li, "Improve google android user experience with regional garbage collection," *NPC'11 Proceedings of the 8th IFIP international conference on Network and parallel computing*, Springer-Verlag Berlin, Heidelberg, 2011, pp. 350–365.

[27] Google Inc, "Platform versions," http://developer.android.com/about /dashboards/index.html

[28] Doug Lea, "Doug lea malloc," http://en.wikipedia.org/wiki /C˙dynamic˙memory˙allocation

[29] Android Open Source Project, "Heap.cpp," https://android.googlesource.com /platform/dalvik/+ /master/vm/alloc/Heap.cpp

[30] Google, "Using DDMS," http://developer.android.com/tools/debugging/ddms.html

[31] Android, https://kernel.googlesource.com/pub/scm/linux/kernel/git/mattst88/glint /+/57439f878afafefad8836ebf5c49da2a0a746105/drivers/cpufreq/cpufreq˙ondemand.c

[32] Web, "Suttering In Game," http://stackoverflow.com/questions/13300939/stutter-in-android-game-garbage-collection

Curriculum Vitae

Md Abu Jahid was born in Faridpur, Bangladesh on December 31, 1984. He graduated from Bangladesh University of Engineering and Technology (BUET) with a bachelors in Computer Science and Engineering degree in 2007. After completing his undergrad study, he joined one of the largest software development companies in Bangladesh, CodeCrafters Intl. He had been working there for four years as a software developer. In the fall of 2011, he entered the Graduate School of The University of Texas at El Paso. While pursuing a masters degree in Computer Science he worked as a Teaching Assistant for the courses Computer Architecture and Elementary Data Structure and Algorithm. He also worked as a Research Assistant in Robust Autonomic Group.

Permanent address: 716, West Yandell Dr, Apt 17
El Paso, Texas 79902