CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Region-specific cache structures for Android mobile platforms

Master's thesis in Embedded Electronics System Design

MAZDAK SANATI

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

Region-specific cache structures for Android mobile platforms

MAZDAK SANATI

Department of Computer Science and Engineering
Chalmers University of Technology
and
University of Gothenburg
Gothenburg, Sweden 2017

Region-specific cache structures for Android mobile platforms
MAZDAK SANATI

Supervisor: Sally A. McKee, Department of Computer Science and Engineering
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden 2017

To my parents, my wife and my son

Region-specific cache structures for Android mobile platforms

MAZDAK SANATI
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The popularity of Android operating system in mobile platforms, as well as constant development in Android systems as modern mini computers, increase the complexity and the size of the virtual memory. Therefore, the need for studying and categorizing Android memory system becomes essential for design and improvement of the memory organization. In this work, we purpose two modern cache organizations, namely, ***region cache*** and ***drowsy cache*** to enhance the Android memory performance. These cache designs are based on multiple simulations on gem5 while running Agave Android benchmark suite. We profile the Android virtual memory to examine which memory regions are accessed the most, and which cache blocks are reused during the run time for different applications. The simulations suggest that many memory regions' hit-rate can be improved by the proposed cache designs in this work.

Acknowledgements

I express my deepest gratitude to the following people:

My supervisor, **Sally A. McKee**, for your mentorship, valuable discussions and your endless support. You are awesome!

My examiner, **Per Larsson-Edefors**, for your patience and your support.

My friend and mentor, **Sakib Sistek**, for encouraging me to continue my studies.

Lars Svensson, for your guidance during my masters.

Zachary Yannyes, for helping me with understanding gem5 and scripting.

Martin Brown, for guiding me through the gem5 architecture.

Jessica Hovey, for proof-reading this work.

My friends **Mark, Katka, Enver, Kaisa, Amir, Nina, Robert, Elsa, Jessica, Daniel, Sophia, Farhang, Marjan, Sam, Pegah, Hoda, and Tobias**, you are epic!

My parents, **Kiandokht & Ali**, for believing in me and your support of a lifetime.

My parents in-law, **Homa & Jafar**, for you love and encouragement.

The little **HamHam**, you are the energy in my life.

My love, my best friend and my light in the darkness, **Mehrnaz**. Without you all of this was not achievable. *Love you.*

Contents

List of Figures

Abbreviations

Acronym	What (it) Stands For
API	**A**pplication **P**rogram **I**nterface
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
IO	**I**nput **O**utput
L1D	**L**evel **O**ne **D**ata
L1I	**L**evel **O**ne **I**nstruction
OS	**O**perating **S**ystem
RAM	**R**andom **A**ccess **M**emory
SRAM	**S**tatic **R**andom **A**ccess **M**emory
LRU	**L**east **R**ecently **U**sed
FIFO	**F**irst **I**n **F**irst **O**ut
IPC	**I**nstructions **P**er **C**ycle
ISA	**I**nstruction **S**et **A**rchitecture

Chapter 1

Introduction

A 2015 GlobalWebIndex survey [2] showed that 80% of internet users own a smart phone and almost half own a tablet. The majority of these almost two trillion mobile devices run on the Android Operating System. The Android execution environment has a unique and complex virtual memory layout. Our recent ISPASS 2016 submission (with M. Brown, Z. Yannes, M. Lustig, A. Sidelnikov, S. McKee, G. Tyson, S. Reinhardt) shows that even the simplest Android applications use dozens of code and data regions that are accessed by many threads. Improving both performance and power consumption requires understanding of how these regions are managed and used.

The Android memory layout differs from the usual Linux layout in that it allocates many smaller virtual memory regions within the traditional heap segment. For instance, native libraries, Java API code, compiled bytecode, and memory used by the graphics card are each allocated and managed as separate memory regions [1]. This allows runtime to tailor memory management according to how the regions are accessed. This, in turn, suggests that the memory hierarchy could also be tailored for specific usage patterns.

Android applications generate of many threads that potentially access many virtual memory regions simultaneously, and the resulting access streams are likely to cause conflict in cache. On one hand, the multiple working sets are unlikely to fit in cache. On the other hand, the interleaving of multiple access streams will obscure or remove spatial locality of reference, which challenges the effectiveness

of traditional, unified cache hierarchies.

In this thesis, we will identify common usage patterns in how different virtual memory regions are accessed by the many application and runtime threads. Based on our findings, we will propose cache structures designed to support the different access behaviors. We will then analyse the performance, power, and area impact of incorporating such structures in an Android platform.

1.1 Purpose and Goal

Cache hit rates can often be improved by increasing cache size, associativity, or block size. Increasing these parameters is not guaranteed to deliver better performance. Cost, complexity, access behavior, and access time constraints often limit the values these parameters can take [3]. Furthermore, larger structures dissipate more static power, and minimizing power is as important as delivering high performance for the mobile embedded systems we target.

Improving performance and/or reducing power dissipation may thus require employing more innovative designs. For instance, skewed associative [4], hash-rehash [3], and column-associative [5] caches all improve cache hit rate without changing the cache structure. Instead they use different mapping functions to mimic associativity while maintaining the simpler structure of a direct-mapped cache. Regardless of the structure of the main cache, adding a small, fully associative buffer to hold recently evicted blocks (on the assumption that data removed by conflicts are likely to be accessed again soon) can avoid ping pong effects caused by multiple hot data mapping to the same cache location [6]. Such victim cache is implemented as a separate structure, it adds no complexity to the main cache. Lee and Tyson [7] carry this separation a step further, demonstrating that employing separate region caches for stack, global, and heap data can significantly reduce power dissipation with little or no impact on performance. Other researchers show that adding tunable drowsy cache management [8] or decay cache [9] can further reduce power dissipation without hurting performance.

We will explore this rich cache design space to identify combinations of structures that better support the Android execution environment. To do so, we will first

characterize how different virtual memory regions are used by both the application and the Android runtime layers. In particular, we will track locality of reference (e.g., stack distance for temporal reuse), access granularities, sizes of memory footprints, numbers of reads and writes, and numbers of simultaneously accessing threads. Based on our analysis, we will then propose cache systems (different organizations in isolation or combination) to better support the common access behaviors among heavily used regions. We will use the gem5 simulator [10] to evaluate system performance. However, the size of these regions and whether if we can benefit from these designs in an actual system was not studied during this project due to the large scope of such work.

1.2 Challenges

The cross product of potential cache organizations represents a large design space. Evaluating every point in this space is infeasible, and thus we need a means of navigating the space efficiently. We can identify promising candidate structures via fast, trace-driven cache simulation before performing detailed modelling in a full-system simulator.

We will identify organizations that meet different design criteria – e.g., maintaining a given power budget, fitting within a specified area footprint, or meeting specific performance goals. Our detailed models should also let us identify common interaction patterns among the different cache structures such that we can derive some broad guidelines for achieving specific goals.

1.3 Limitations

The final product of this project is in the form of simulation data and simulation models, therefore we will not deliver any actual hardware products.

Chapter 2

Background

In this chapter we will first explain some of the related work in Android benchmarking, cache optimization methods, and special cache structures. These works helped us understand and define our limitations and set goals for our project. Later in section 2.2 we will give some theoretical background on memory hierarchies and cache design.

2.1 Related Work

This project relies significantly on utilizing Android benchmarks as a method of understanding Android memory organisation. Therefore, it is important to look for the best benchmarking tools that would provide us with the information needed in the process of studying Android mobile platforms' architecture.

2.1.1 Android Benchmark Suites

There is rich prior work in benchmarking and workload characterization (entire books have been written on the subject). We restrict our discussion to other Android benchmark suites and workload characterization studies. We broadly categorize benchmark suites into four groups: General purpose computer architecture benchmarks, plug-and-play "black box" Android applications, microbenchmarks/minibenchmarks (mini-programs with code that is representative of real applications but lacks full functionality), and Android benchmarks [1]. In this section we

will discuss some of the most popular benchmark suites from the general purpose benchmark suites (commercial) , down to the Android benchmarks and highlight their differences from Agave benchmark suite.

General purpose computer architecture benchmarks in the first category help architects measure performance of the processor, memory, and compiler on different systems. However because of Android applications' reliance on shared libraries and OS services, these benchmarks have compatibility issues with mobile devices and therefore lack the ability to fully characterize the performance of mobile platforms. General purpose computer architecture benchmarks include codes modeling parts of the functionality of real applications. For instance, MiBench [11] is a free, open source suite intended to represent the kinds of applications that run on embedded systems. The suite contains 35 applications that span six categories: automotive and industrial control, networking, security, consumer devices, office automation, and telecommunications. MiBench has been used most for computer architecture design space exploration. The applications are intended to be commercially representative. they carry out some of the functionalities found in real embedded systems in programs that are still small enough to run in (slow) architectural simulators.

SPEC™ CPU 2006 benchmark provides performance measurements to compare compute-intensive workloads on different computer systems. The SPEC CPU™ 2006 benchmark is SPEC's CPU-intensive benchmark suite, with focus on the system's processor, memory subsystem and compiler. This benchmark suite includes the SPECint® 2006 benchmark (containing 12 different benchmark tests) and the SPEC® 2006 benchmark (containing 19 different benchmark tests). The SPEC CPU™ 2006 benchmark has several different ways to measure computer performance. One way is a single task computation; where speed measurement is performed. Another way to measure computer performance in SPEC CPU™ 2006 is through a capacity or rate measurement which is time constrained measurement for the maximum amount of tasks, this is called a throughput. [12]

Princeton Application Repository for Shared-Memory Computers (PARSEC) is a

benchmark suite composed of multithreaded programs for Chip-Multiprocessors (CMPs). PARSEC's workload includes system applications which mimic large-scale multithreaded commercial programs, mining and synthesis (intel RMS) applications, and applications in recognition. The PARSEC workload consists of 9 applications and 3 kernels all developed in Princeton University, Stanford University and intel (Microprocessor Technology Labs). Bienia et al. [13] show that PARSEC covers a wide spectrum of working sets, locality, data sharing, synchronization and off-chip traffic. PARSEC provides useful data on cache organization, it focuses on the desktop and server application.

Black box benchmarks in the second category are intended to help consumers evaluate and compare platforms, and they often compile and publish the results on the Internet. Products like Quadrant [14] and AnTuTu [15] provide information about screen, video, memory performance and battery lifetime. Others, like CPU Benchmark [16] and GFXBench [17], give more detailed information about specific components, such as CPU or GPU performance. These tools do not expose their benchmarking methodologies or give the user detailed control over how the benchmarks are run, and thus they cannot be used for the kinds of hardware/software explorations we seek to enable. We therefore focus on open-source programs that afford more flexibility in how they can be configured and in what information can be tracked.

Microbenchmarks in the third category are useful for studying specific aspects of system performance. For instance, Androbench [18] is a storage benchmarking tool with five microbenchmarks — two applications to measure sequential read and write performance, two applications to measure random I/O performance, and an SQLite benchmark for assessing the performance of database inserts, updates, and deletes. Users can set parameters like target partitions, file sizes, and number of transactions. Kim et al. [18] use AndroBench to demonstrate that the I/O performance of Android devices vary with file systems and storage mechanisms. While

useful, AndroBench cannot represent the complex storage patterns of actual applications running on the Android software stack. A more thorough evaluation of the I/O performance requires that operations be executed with variable inputs and time constraints. In the same spirit, Lin et al. [19] construct a suite of 12 small methods representing programming constructs like nested loops and recursive calls. Since the majority of Android applications manipulate multimedia, Lee et al. [20] construct AM-Bench, a benchmark suite, covering the essential multimedia activities of play-back, compression, computation, and rendering. The target applications include the native camera and gallery applications, plus a popular e-reader, barcode scanner, and 2D and 3D versions of a game development framework. From these, the authors extract Java methods implementing the applications' main activities to create a suite of 20 benchmarks. The benchmarks are open source, but running those extracted from the native applications requires replacing the full applications with the benchmarks, which, in turn, requires root access.

Benchmark suites described in this category have all proved to help the community evaluate the performance of different hardware. However none of them actually exercises Androids' real application and therefore can not give accurate information to help develop better hardware for mobile platforms.

Android benchmarks in the fourth category include BBench [21], the set of smartphone applications from Sunwoo et al. [22], MobileBench [23], and Moby [24]. Gutierrez et al. [21] port an Android web browser to the gem5 [10] full-system simulator to create the open-source BBench benchmark, which renders a set of websites. Sunwoo et al. [22] build on BBench, adding five more applications from traditional benchmarks, gaming applications, and productivity applications (which we heretofore refer to as the ARM® suite).

Pandiyan et al. [23] collect a set of four representative smart phone applications, including general-purpose interactive web browsing, education-oriented web browsing, photo browsing, and video playback. The result, MobileBench, is suitable for studies on real smartphone platforms as well as on full-system simulators.

Huang et al. [24] select 10 popular Android applications — including a web

browser, media players, social networking, and a game — to create the Moby suite. While source code is available for some of the applications, others like BaiduMap and Adobe reader are proprietary. Other than BBench, the ARM® suite uses applications for which users must pay (e.g., EEMBC [25]'s AndEBench) or that are only available in binary form.[1] MobileBench is freely distributed as binaries that (should) run on real platforms as well as simulators/emulators[2], and the authors provide ready-to-run gem5 images.

Gutierrez et. al [21] compare microarchitectural performance characteristics of BBench with SPEC™ CPU 2006 workloads [26], finding that BBench exhibits poorer instruction cache, instruction TLB, and branch predictor performances. Their gem5 BBench port facilitates exploring detailed behavior of an important class of Android applications.

Sunwoo et al. [22] combine the SimPoint [27, 28] partial simulation methodology, Principal Component Analysis (PCA), and Fractional Factorial experimental design to greatly reduce gem5 simulation time for processor design space exploration and workload characterization. They developed six benchmarks that include traditional benchmarks, gaming applications, and productivity applications (including BBench). They confirm the findings of Gutierrez et al., that Android applications exhibit very different instruction-related characteristics from SPEC ™. To further facilitate simulation-based design space exploration, they design a noninvasive user-interface automation tool. Their SimPoint workloads have just a 2.5% average CPI error compared to full workloads.

Pandiyan et al. [23] use gem5 to explore the energy and performance of their MobileBench benchmark suite. They conduct a memory system study showing that using larger TLBs improves IPC and L2 utilization and that using a stride prefetcher reduces L2 miss rates.

Huang et al. [24] use Moby to analyse both micro architecture-independent and micro architecture-dependent Android application characteristics. For instance, their locality analysis finds that a highly associative instruction cache (64 or more ways) can manage over 80% of the instructions, and a four-way associative data

[1] www.eembc.org

[2] The links to the downloads were broken when we tried them recently.

cache can reuse nearly 70% of the lines due to high temporal locality. They find that their applications have working sets from 47-114 MB and they spawn 5-38 processes that access 16-31 libraries, but they nonetheless restrict their study to analysing the main benchmark application process' instruction references to the various Android virtual memory code regions.

Android has a very complex virtual memory organisation in contrast to C programs running on Linux. This complex execution environment helps computer architects to better support the execution characteristics, structures and resources for Android software stack. To help the community to benefit from these opportunities we developed Agave, an open-source benchmark suite designed to expose the complex interactions between components of the Android software stack [1].

2.1.2 Cache Studies

In this section we will briefly describe some of the work done by cache architects. First we will explain an older work by A.J. Smith describing the cache memory design that helped us understand the factors that can benefit the cache memories, later in section 2.1.2.2 we will give the reader information about two specific cache designs,namely region and drowsy cache that introduce new horizon to the cache architecture.

2.1.2.1 Cache Optimization

2.1.2.2 Special Cache Structures

There are many new designs for specialized cache organisations and many of these designs - based on their purpose and environmental set up - have proven to be successful in lowering the miss rate and power dissipation. Here we will briefly describe some of these designs chosen based on their relation to our project.

Region Cache: Although power consumption per instruction has been reduced by using methods such as improvements in instruction compression algorithms . [29] or compress instruction coding [30], data cache is still consuming a large portion of the power budget. Several methods have been introduced to reduce

the power consumption by partitioning the data cache into smaller components, unfortunately these components lower the performance by increasing the average latency and prolonging the execution time. [7]

Lee et al. [7] introduce a first level cache organization to exploit the memory references characteristics more efficiently. These semantically defined memory characteristics suggest cache partitions dedicated to three memory regions: stack, heap and global. The results show a 66% power consumption reduction on average while running MediaBench benchmark on a model that resembles Intel StrongARM SA-1101 with two small region-based data caches and a larger L1 data cache. The region-based L1 data cache reduces the power consumption without increasing the execution time by exploiting the spatial and temporal locality in stack and global data references. Lee et al. profile the heap memory reference during run-time as well as the stack function calls and global data references. They observed a total of 40% stack calls, 30% of heap and 30% of global references. To understand the locality of a cache line they calculate the number of cache hits for every single cache line before it is evicted. On average the stack cache lines have the highest lifespan followed by the global whereas heap cache lines have the shortest lifespan. They also ran experiments on the locality of each region by allocating a dedicated data cache to each region. The result shows that at a given cache size, stack has the best locality. In second place comes the global data followed by the heap. Based on this result Lee et al. suggest that an L1 D-cache structure with two small stack, global region cache and a larger heap cache allows level one data cache partitioning without having a noticeable effect on the access latency.

The region-based caching presented in Lee et al. is a horizontal partitioning method presented in the figure 2.1. It has two horizontally partitioned recombined caches (RC1 and RC2) along a larger regular cache. This design has some advantages: First, there are no memory conflicts between regions which eliminates the need for complex designs for each region (such as high associativity) which leads to faster cache access logic. Second, a smaller dedicated cache can store more data than a large combined cache. Finally, a smaller cache has less power dissipation if the performance is not changed.

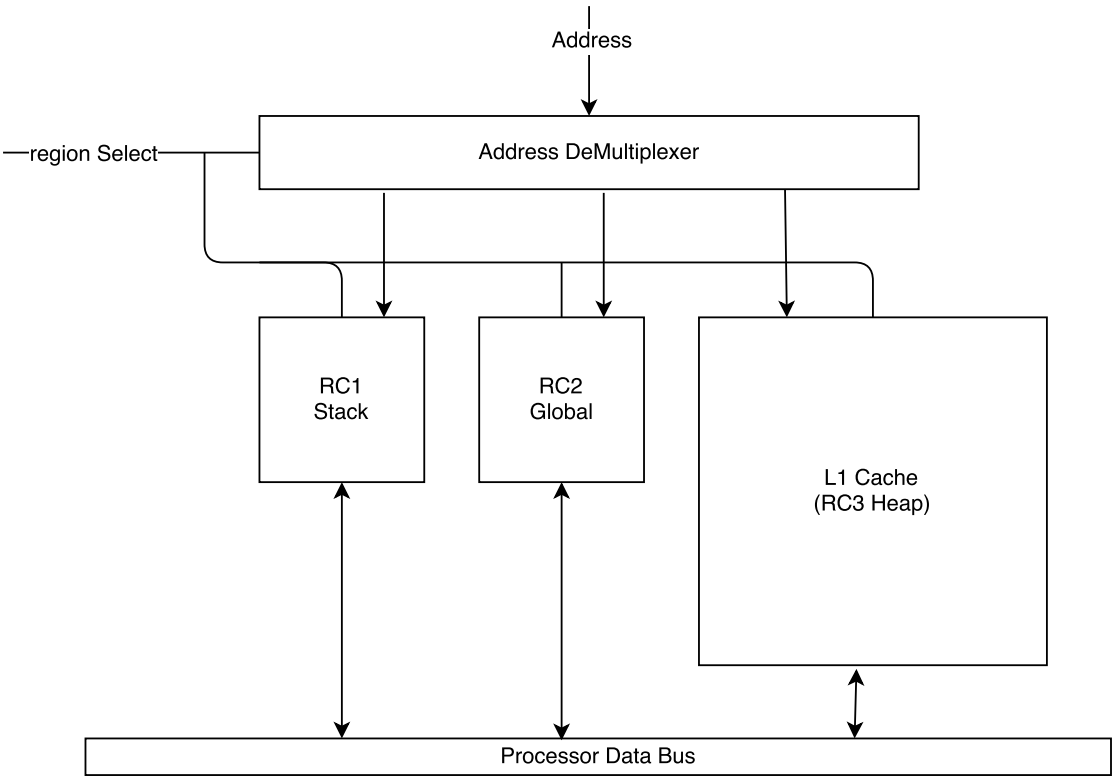

Figure 2.1: Region based level 1 data cache design with two small cache, namely stack and global, and a large heap cache

Drowsy Cache: Due to the ever-growing size of cache in embedded systems and the portion of the die they consume, it has become an important matter to consider the leakage current caused by the decreasing transistor voltage threshold. This leakage power can sometimes go up to 70% of total power in a 70nm processor. One method to decrease the leakage current is to reduce the voltage on the cache lines that remain idle most of the time. This technique is referred to as drowsy caching [8]. One of the most used policies is the *simple* policy where after several cycles all cache lines are put to sleep for a specific number of cycles. However, this policy is architecture specific.

Another drowsy policy is the *noaccess* policy which will turn off cache lines that are not accessed within a time window. Bhadauria et al. instead suggest another policy where they exploit the temporal locality and deliver a better energy saving system with almost no performance degradation in comparison to *simple* policy or no drowsy cache. Here we refer to this policy as *reuse distance* policy. The advantage of *reuse distance* policy is that it is not architecture-specific and therefore

allows prediction of leakage saving for individual benchmarks.

This drowsy cache model, only keeps several of the most recently used cache lines awake. Bhadauria et al. [8] show even better performance at simpler implementations while giving more control than *simple* policy with respect to enforcing a strict power budget. The *reuse distance* policy is shown to use less than 44% of the *simple* policy for L1 cache and 93% for L2 while maintaining 97.6% of the IPC of the same none-drowsy model. These results were achieved while running SPEC benchmark for a 32KB L1 D-cache with a 512KB L2 cache. The *reuse distance* policy supports performance better at larger cache sizes and higher clock rates while maintaining a strict power budget. The *reuse distance* is the technique we based our L2 cache design on. Furthermore we investigate the possibility of Android systems benefiting from this technique. However, due to the large scope of the implementation and its complexity we will did not implement the drowsy cache.

2.2 Cache Memories

The memory hierarchy is best explained by a pyramid chart where the CPU with relatively smaller size stands at the top of the pyramid (level 0) and the disk with the largest size lies at the bottom (Figure 2.2). Caches are memory buffers that dynamically load memory blocks accessed by the processor. Typically L1 and L2 are made on chip while L3 and above are usually off chip. L1 and L2 are usually built of SRAM cells and any cache in a lower level is usually built of DRAM cells. There is a proportional relation between the cache speed, hit rate, and cache size. Larger caches tend to store mode addresses and are more likely to contain the accessed address but they are slower because access times to memories are affected by wire delays. In addition, the larger memories require larger decoders and multiplexers which bring more complexity to the system.

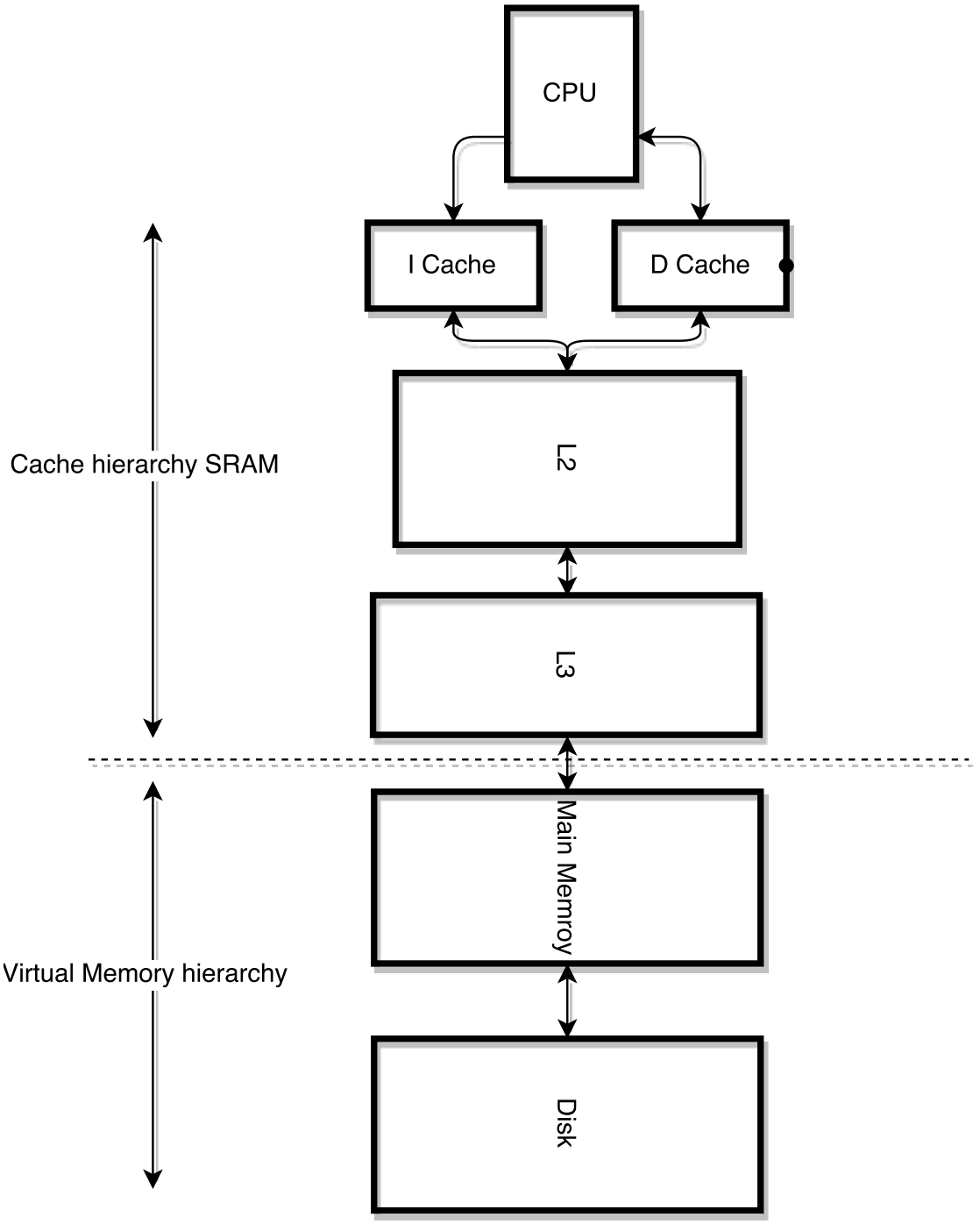

Figure 2.2: The memory hierarchy

2.2.1 Locality

When a process runs the main program it typically jumps quite often to many procedures and codes which each have their own data. therefore the *working set*[3] varies with time. When a process executes part of its code the miss rate in the

[3]Set of memory Addresses accessed by a process at any time

cache and memory are quite low; Furthermore, new area in the memory is accessed and the miss rate tends to rise when there is a change to another code module. Cache blocks and memory pages containing the previous working set must now be replaced by a new working set.

This behavior of the memory-access explains the *locality property* of the program. There are two types of locality: *Temporal* and *Spatial*. Temporal locality explains the consecutive access to the same memory address, meaning if a memory address has just been accessed, there is a high chance that it will be accessed again very soon. This behavior is observed mostly in program loops where data and instructions are used several times. The spatial locality explains the memory addresses that are close to the requested address. This means that if an address is accessed the memory addresses close to its location have a high probability of being accessed in upcoming cycles. This behavior is common among related data items (variables and arrays) that are usually stored together. It is because of the locality property of the memory accesses that caches are successful. If process would have accessed memory randomly the miss rate in caches would have been unacceptable [31].

2.2.2 Cache Optimization Design Aspects

Considering a suitable cost, a cache design is supposed to accomplish four aspects [32]:

1. Hit rate: The probability of finding requested memory block in the cache.
2. Access time: The time to access the data that is in the cache.
3. Miss delay: The delay time in case of a cache miss.
4. Overhead: The overhead of updating main memory and maintaining multi-cache consistency.

In an ideal scenario we want the cache system to maximize the hit rate while minimizing other aspects mentioned above. However, there is a trade-off between Hit rate and access time, since if a cache hits then it will take some time to access the information inside the cache. In the remainder of this section we will discuss

some of the design parameters that can affect the aspects we mentioned here, however we will mainly focus on the miss/hit rate and the access time.

2.2.2.1 Cache Design

2.2.2.2 Data/Instruction Caches

The L1 cache is usually a split instruction/data cache to avoid the overlap of data accesses and instruction fetch. This has the advantage of increasing the cache bandwidth (read to write rate of data in the cache) and minimizing the access time. However, the hit rate may decrease and the two caches must be kept consistent. [32]

2.2.2.3 Cache Size

Although larger cache size results in increased hit rate, there are limits to the cache size. Cost is the first driving factor, physical size (fitting the cache on the chip), and access time are some other aspects that must be take into consideration.

2.2.2.4 Multilevel Cache

By increasing the cache size there will come a point where it is better to split the cache into two levels. The higher level is faster, smaller, and more expensive whereas the second level is larger. Through multi-level caching problems with the oversized cache can be eliminated.

2.2.2.5 Replacement Policies

Upon access to a memory block that does not reside in the cache, a cache miss is triggered and the *replacement policy* will select the *victim block*. This victim block must be in the cache line where the requested missing block is mapped. In a direct-mapped cache it is straightforward since the missing block is only mapped to a single cache line. However in a fully associative cache all cache blocks are candidate for the replacement and in a set-associative cache any block in the set is a candidate.

Random selection is the simplest replacement policy. However it does not help

the memory system to minimize the miss rate. Other replacement policies try to select the victim cache in such a way as to lower the miss rate. The ideal scenario would be if the replacement policy could predict the future and select the victim block that is accessed the farthest. This policy is called OPT (optimal). In OPT policy a block is kept only if there is a hope that it will be accessed next. In the OPT policy it is very probable that a block is replaced if it must stay longer to save the miss rate.

The LRU (least recently used)policy selects the victim blocks base on their residence time in the cache. Meaning the older block (without access) is more likely to be evicted. The LRU policy relies on the locality property of accesses to each block and checks all block for the last time they were accessed. The simplest way of tracking the access history is by assigning priority bits (also known as history bits) to each cache line. The higher priority lines are closer to eviction. The priority change is possible by incrementing the history bits by 1 modulo 4. This will make the history bit updating more complex and therefore the pseudo-LRU is used instead.

In pseudo-LRU the LRU block is not tracked accurately, and therefore history bit update is much simpler. One disadvantage of the LRU and pseudo-LRU policy is that the history bits are updated every time there is a cache access. The FIFO (first in first out) policy on the other hand will only update the history bit on a miss. The problem with the FIFO policy is that the blocks are evicted even if they are accessed often and therefore the miss rate might be higher than LRU. [31]

2.2.2.6 Cache Mapping

In a conventional processor the cache line and memory block are mapped based on the block address. The memory address is divided into two parts: the block address and the offset (Figure 2.3). Suppose we have a memory address of d-bits. Also let us assume that the main memory has addresses from 0 to $2^d - 1$. By dividing the memory into same size blocks (containing several addresses) we will have $\frac{2^d}{MemoryBlockSize}$ - 1 blocks. Now if we want to locate the memory block address requested by the processor, we have to divide the address by the block size. The

quotient is the block number. Keep in mind that this division is quite simple since block size is a power of 2. Let us assume the block size is 2^b bytes. This way, in a d-bit memory address the first b-bits are the offset and $d - b$ is the memory block address while the number of memory blocks is $2^{(d-b)} - 1$ (Figure 2.4). Next, let us assume that there is a cache memory in between the processor and the main memory with a total number of $2^{(c-b)} - 1$ lines where c is much smaller than d. Since the cache is much smaller than main memory, each cache line is going to have a subset of certain blocks from the main memory. A cache has two main parts: a directory memory and a data memory. The directory memory contains the tags (IDs) of the current memory block in a cache line plus the validation bit while the data memory contains a copy of the memory block.

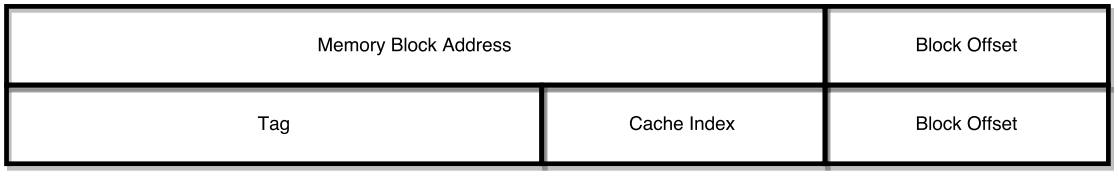

Figure 2.3: Memory address and memory blocks

Direct-mapped Caches: In a Direct-mapped cache, a given memory block and cache line are always mapped together. The location of the cache line is achieved by hashing the memory block. The simplest hashing method is bit-hashing where a field of the block address (usually the least significant bits in memory block address) selects the cache line (Figure 2.4). The rest of the bits (most significant bits of the block address) from the ID go to the cache's directory memory. Data memory has a width of 1 word and the cache line size can be anything from two words and above. The number of words per cache line defines the cache block size. Let us assume a cache line size of two words for the data memory. This way the height of the data memory is twice the directory memory (in a narrow cache)[4]. The ratio between the height of data and the directory is $W = 2^w$, where W is the number of words per cache line. With a block size of $B = 2^b$, S being the number of lines ($S = 2^s$) and a memory address size of $D = 2^d$, the number of tag bits that define the block in cache line is d - s - b.

[4]In a wide cache the data and directory memory have the same height

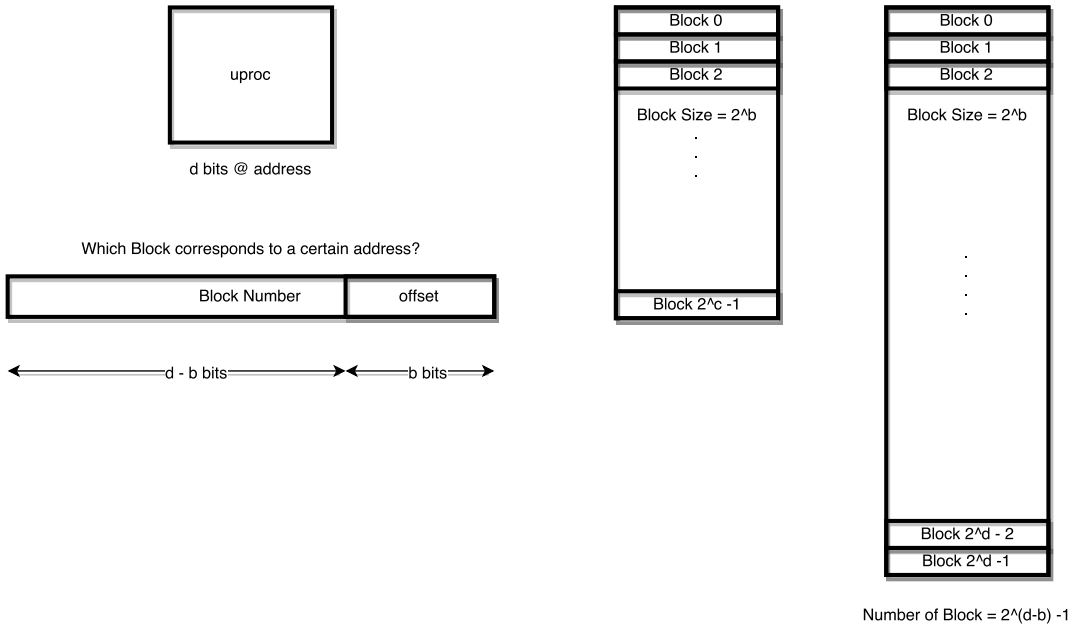

Figure 2.4: Memory address fields for direct-mapped access

When there is a read access, it proceeds in two steps: First the Cache indexing, where the directory memory of the cache is fetched with the s least significant bits of the block address and data memory is fetched with the s least significant bits of block address plus w most significant bits of the block offset. Second is the tag checking, the tag in the directory map is checked and compared with the most significant bits of the block address. Furthermore, the valid bit is checked. If both tag and state (valid) bit check, the cache hits; otherwise a cache miss is triggered. Figure 2.5 shows the structure of a wide cache. In the event of a cache miss, the system can significantly benefit from a wide cache since the time it takes to reload a block is one cycle of a data memory whereas in a narrow cache it takes W cycle to reload a block.

In a direct-mapped cache as we mentioned earlier, a given memory block (memory block m) is **always** stored in a cache line (cache line n) and the consecutive blocks (m + 1) are stored in cache line n + 1. Since the cache size is much smaller than the main memory size the memory block m is stored in the cache line n and this process is repeated until the entire memory is mapped to the cache (Figure 2.6). Direct-mapped cache has one major advantage and that is fast access time on a hit. However, since in a direct-mapped cache a large number of memory blocks

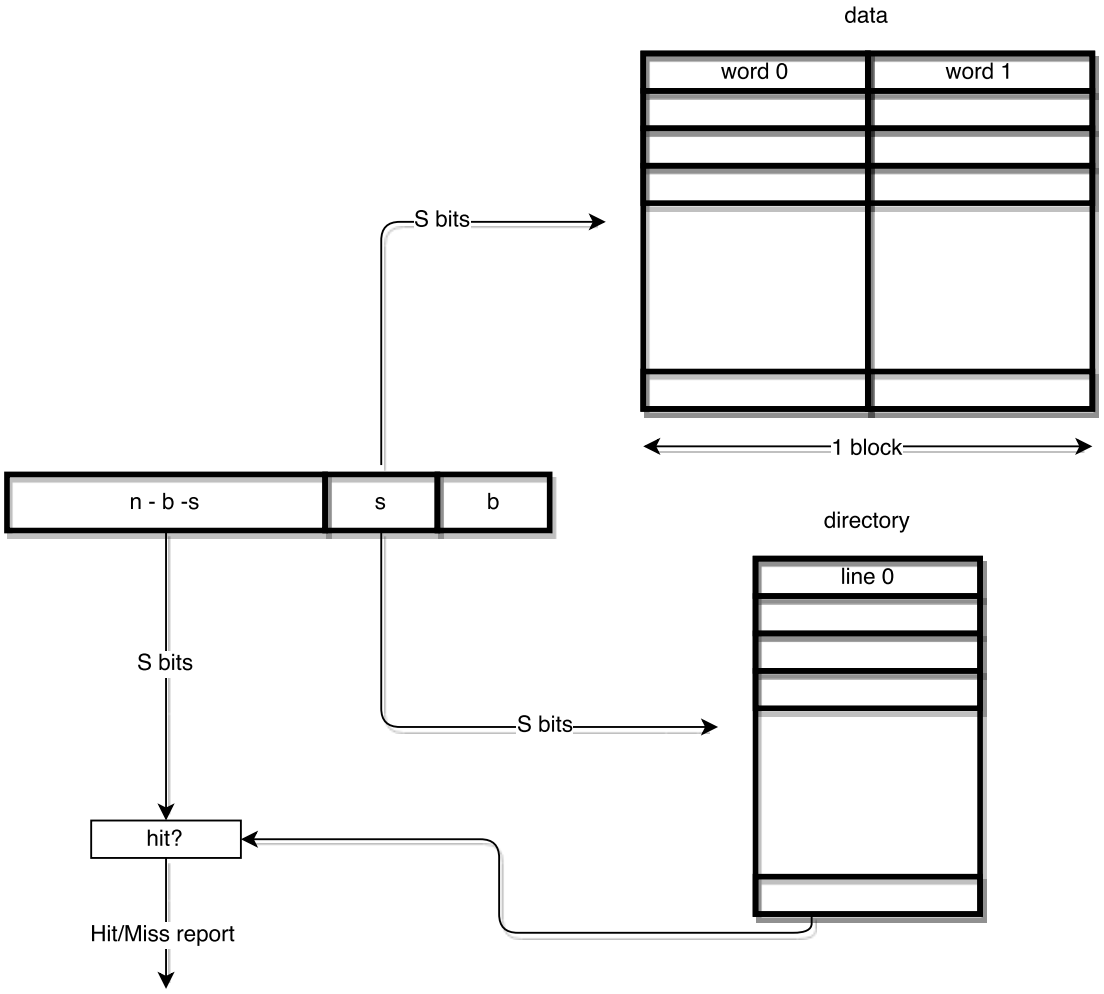

Figure 2.5: Memory structure of the wide cache

are restricted to map to the same cache line, the miss rate could significantly rise when several blocks map the same line simultaneously.

Set-associative Caches: To eliminate the problem with blocks competing for the same cache line while maintaining the fast access time, set-associative caches are introduced. A set-associative cache is separated into sets of lines and each set is direct-mapped but the main difference is that blocks can be stored in any line within the set.

A set-associative cache is usually followed by the name N-way, where N is the number of lines within each set. Notice that for every set the mapping is exactly the same as in the direct-mapped. Each set has its own directory and data memory and therefore, sets have independent miss or hit. In other words a direct-mapped

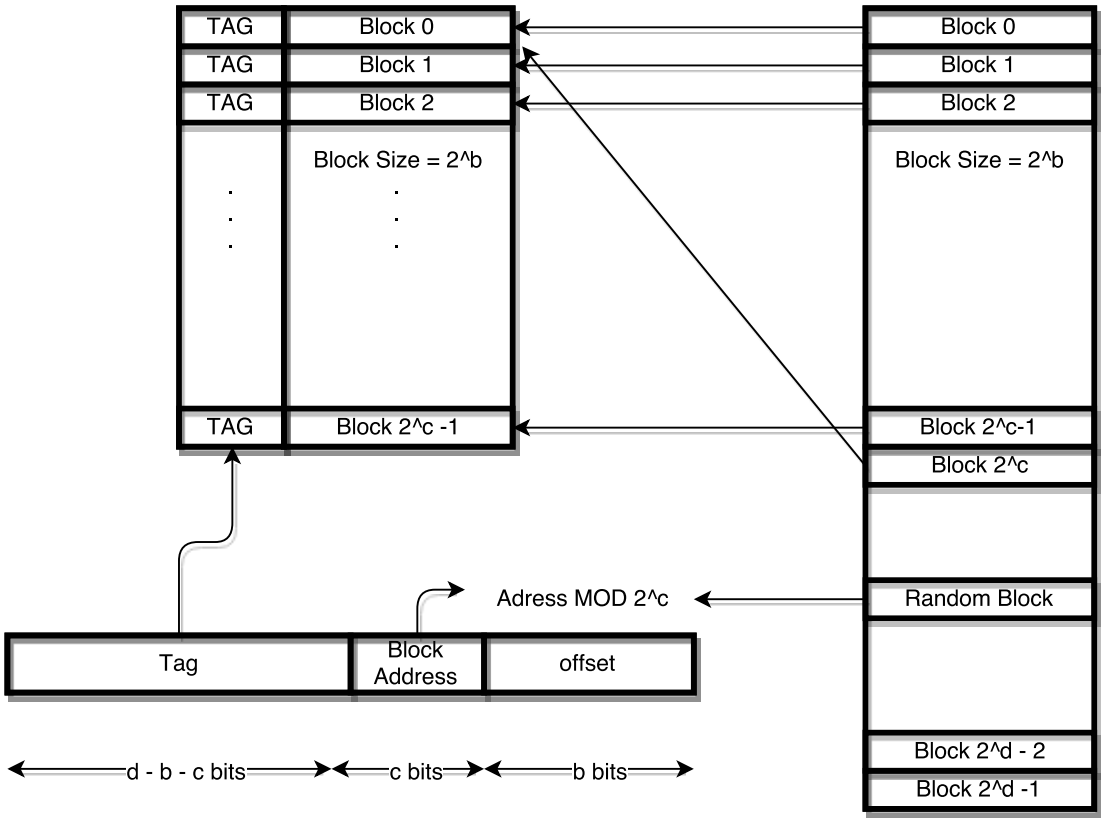

Figure 2.6: Memory address and memory blocks in a direct-mapped cache

cache is a 1-way set-associative cache. In a set-associative cache the width size of data memory is calculated as following:

$$\textbf{number of sets} \times \textbf{number of blocks per set} \times \textbf{block size} \qquad (2.1)$$

Typically the size of a set-associative cache is between two and eight lines. Beyond eight lines , the cache will become slow with insignificant hit rate.

Fully Associative Caches: In a fully associative cache a memory block can be mapped to any cache line. The cache tag in the directory memory of a fully associative cache contains the entire block address. In this way, all the directory lines must be checked in parallel to locate a block address. A *read* or *write* proceeds in the following order: The block tag is matched against all previous tags. A tag bus line looks through all directory entries and the bus value is compared with all tags stored in the directory. In case of a match the data is returned. The directory memory of a fully associative cache is made of CAM (content-addressable

memory) for the purpose of parallel tag matching. Therefore, access time in a fully associative cache is slower than a set-associative cache made out of RAMs but the hit rate should be better since the mapping is more flexible in fully associative cache. Thus, fully associative caches are preferred in designing small caches. [31]

2.3 Android Software Stack

The Android software stack consists of five layers (Figure 2.7). On top of the Android software stack lies the application layer that contains end-user applications (often referred to as *"apps"*). These apps are either delivered with the Android device or downloaded from digital sources such as Google play. Beneath the application layer there is the Android application framework which is a set of Android APIs that help developers write apps. The application framework contains design tools for developing user interface and system tools.

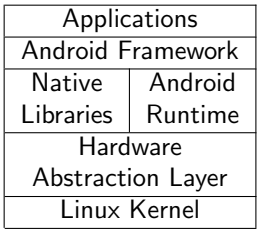

Figure 2.7: Android Gingerbread Software Stack

Android Native Libraries (common C libraries) and Android Runtime, which contains common Java libraries and Dalvik Virtual Machine (DVM), are the third layer in the Android software stack. The Hardware Abstraction Layer (HAL) helps developers to add functionalities without having to change the higher level system. The HAL implementations are usually built into shared object modules (.so). The Linux kernel at the bottom delivers low-level services such as process management, memory management, and device drivers. Some of these features are tailored specifically for mobile embedded platforms.

2.4 gem5 Simulation Environment

As our main open source tool in both understanding and simulating the Android devices, gem5 helped us in developing Agave benchmark suite and continues to be one of the major utilities for understanding the cache system in Android devices. The ability to build different architectures as well as the possibility of modifying different architectures in detail (sim-objects) allows us to improve the cache system in memory and study the overall performance of the system. We will discuss these sim-objects in detail and the changes we have made in the gem5 to study the Android memory behavior in the section 3.1.1.

gem5 is a combination of M5 and GEMS simulators. Inheriting the M5 properties allows the gem5 to have a configurable simulation environment where users can configure multiple ISAs and CPUs. The GEMS aspect of gem5 delivers a flexible memory system that supports cache coherence protocols. The gem5 simulator supports many architectures such as : ARM, ALPHA, MIPS, Power, SPARC, and x86. gem5 has two important features, first it is open source which allows researchers to collaborate in both industry and academia. The second benefit of gem5 is that it allows deeper exploration of multicore systems and complicated cache hierarchies while providing OS facilities such as IO and networking. The gem5 simulator offers a variety of CPU models, execution modes, and memory models and allows us to focus on the one aspect that is important to this project, namely the cache organisation. [10]

The CPU models available in gem5 are *Atomic-simple*, *Timing-simple*, *In-order* and *O3*. The atomic model is a minimal IPC model that we used in this project. The timing model is similar to the latter but it does simulate the memory reference of the memory as well which is not a subject of study in this project. The in-order and O3 are useful in pipeline models.

The system execution mode in gem5 is scripted in the configuration in gem5 and offers two modes, the system call emulation (SE) and full system (FS). The SE mode does not model the devices or OS and only simulates the system services while the FS mode emulates both user and kernel instruction and models a complete system which was favourable in this work. It goes without saying that the

full system simulation has the time trade-off.

gem5 offers two memory systems, classic and RUBY. The classic memory offers a fast and easy configurable system while the RUBY model provides a more complicated structure that allows simulation accuracy for plenty of cache coherent memory systems, as cache coherence was not a subject to be studied in this work, we chose the classic memory model.

In addition to the options mentioned above, gem5 offers execution in both ARM and X86 ISAs, since we are modelling ARM cortex A9 and A15 in this work we chose the ARM ISA model for our simulations. [10]

Chapter 3

Method

We divide this chapter into four sections. First we will explain our experimental set up where we discuss the Android memory management and the preliminary experiment which is the foundation of our research. In the second part, we are going to describe our approach for understanding Android memory regions and Android memory behavior. Third, we explain our simulation on gem5 for categorizing Andriod applications' memory management. Finally, we are going to describe steps we took to mount Android on gem5 as well as our approach to gathering and in-depth analysis of data.

3.1 Experimental Setup

3.1.1 Android Memory Management

Unlike the typical C/Linux program's virtual memory with only four core regions, the simplest Android apps use dozens of instruction and data regions because of the Androids unique software stack. Therefore, it is important to understand how these regions are managed and accessed. Understanding the virtual memory layout and the memory management is the first step in this project to find out more about memory regions that are accessed during app execution. For instance, Zygote is a daemon (computer program that runs as a background process) responsible for launching the Android applications. Zygote starts up after the service manager

(an information director for all available services) but it is actually triggered by *app_process*.

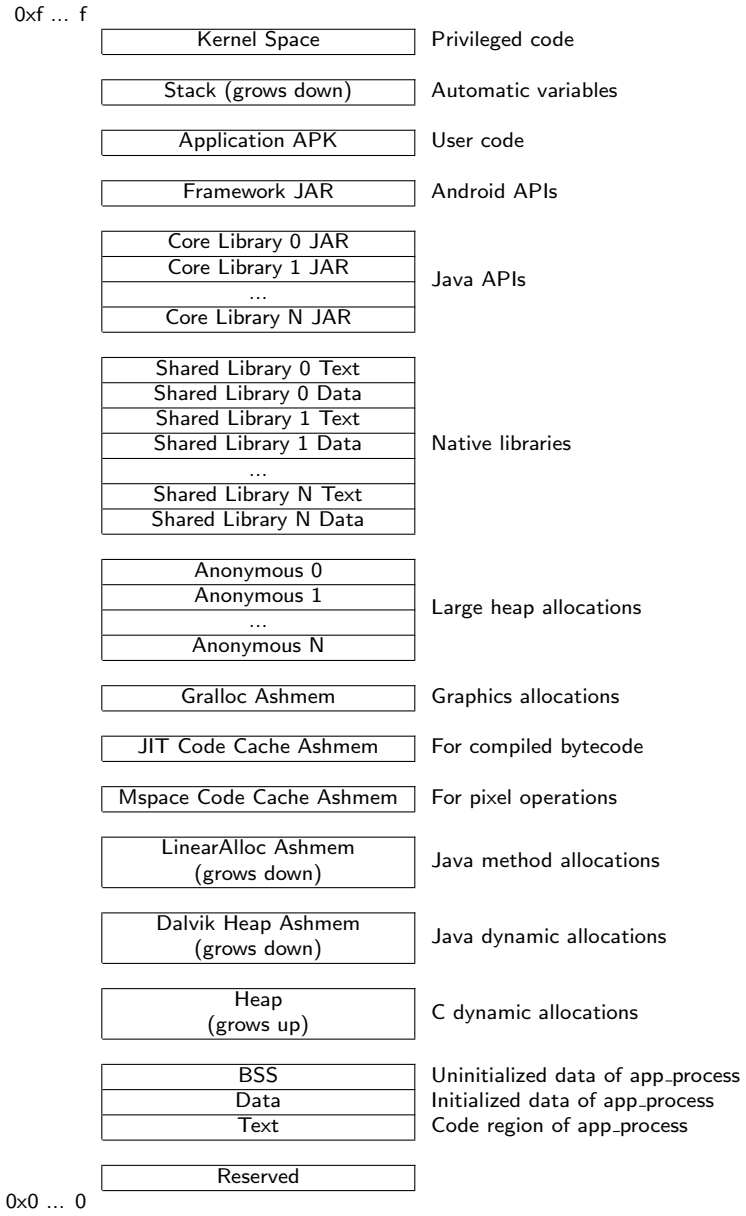

Figure 3.1: Simplified illustration of Virtual Memory Layout for an Android process in a 32-bit system. [1]

Figure 3.1 shows a simplified version of the app_process virtual memory layout. The app_process uses the standard C regions (code, stack, heap, and global) as well as Linux MMAP (responsible for mapping files and devices to memory). JIT compiler (Just-In-Time compiler improves the performance of Java applications at run time) reads User code (APK files), Android framework, and core libraries

(JAR files) as data before converting them into the Dalvik executable files (.dex).

Android places dynamic allocations on the MMAP segment to handle the resource constraints. This is achieved using Android's shared memory (*ashmem*). For instance, graphics buffer allocations are created for driving graphical data to the framebuffer by image producers. In C programs, the user has a large heap region to utilize when needed; In Android however, there are more regions that are designed as heap and the user has less or no influence on which heap region needs to be used. [1]

3.1.2 Virtual Memory Regions Miss Rate: The Preliminary Experiment

Brown et al. have modified an older version of gem5 cache objects to count the level one cache miss rate and logged them at the end of the simulation. Furthermore, we have added a way to log cache accesses in real time in the *cache_imp.cc* (responsible for implementing the cache). The purpose of such cache-logger is to document all accessed memory regions and check if it is a hit or a miss in the cache. We looked at two scenarios: The first scenario is when each region has its own dedicated (virtual) cache with sizes varying from 64B upto 32KB and the second scenario is when a unified 32KB cache is shared among all memory regions. Please note that these scenarios are different than the final goal of this project which suggests a design for a region based cache. This project was designed to understand the basic behavior of the system during the run-time and does not present actual results of a region cached system.
We observed that there is a miss rate in smaller dedicated cache sizes. It suggests that some regions might require larger dedicated cache size to avoid memory conflicts with themselves (these are referred to as a 'self-miss'). The unified cache miss rate suggests that memory regions will have more conflict with each other even if the cache size is relatively large, and some regions would benefit from dedicated cache. Figures 3.2 to 3.5 show graphed miss rate during the run-time for some

memory regions in a direct mapped level one instruction cache simulated in gem5 while running 19 different benchmarks. [1]

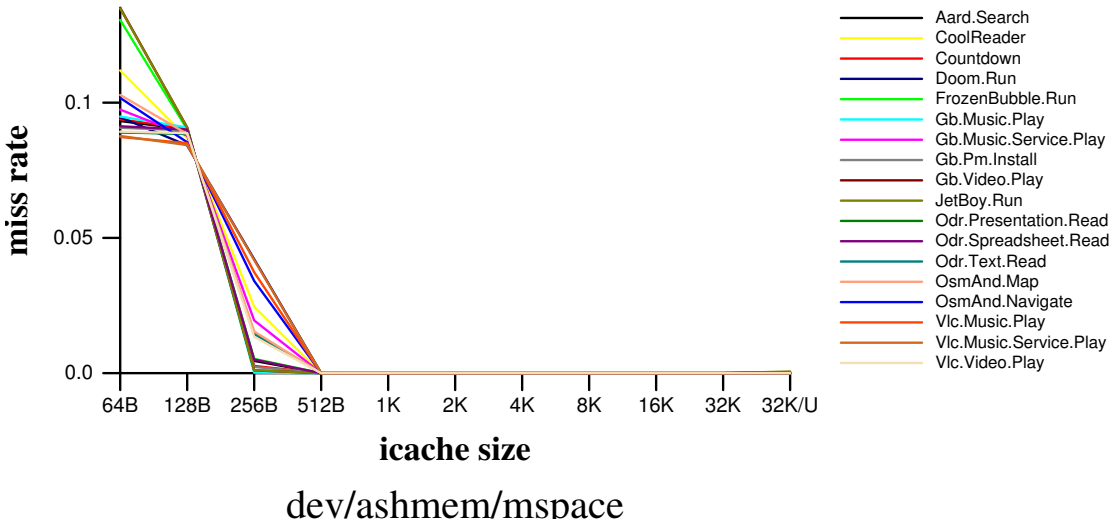

Figure 3.2: Cache miss rate, Simulated for mspace (graphical library) in 19 benchmarks

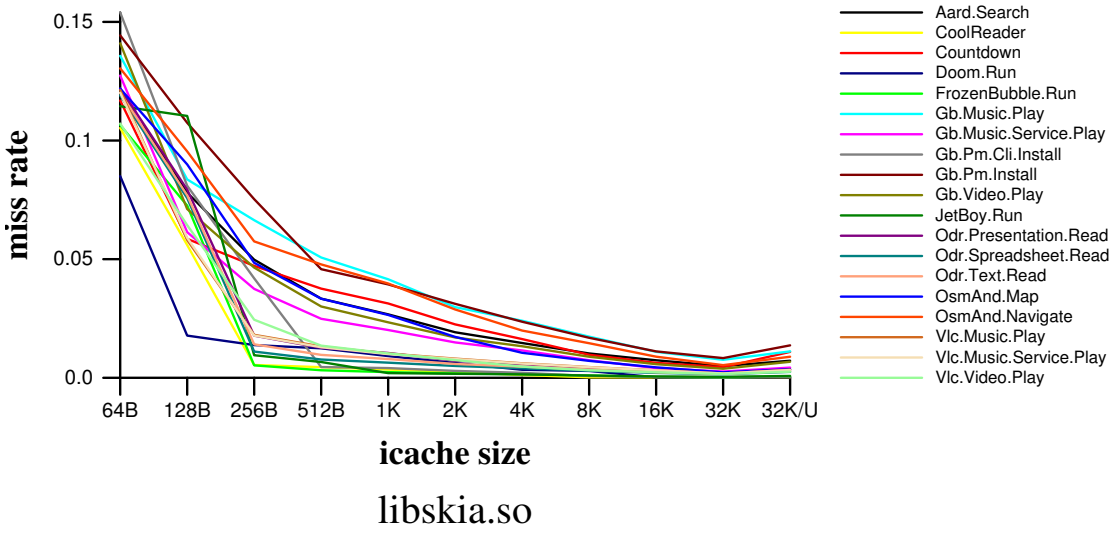

Figure 3.3: Cache miss rate, Simulated for libskia in 19 benchmarks

As we can can see in the figures 3.2 to 3.5, the miss rates increase with the unified cache even if it is a relatively large i-cache (32KB). This is because as number of regions grow, memory regions will compete for cache resources and it is possible that a cache line that is requested in a couple of accesses ahead has already been replaced by the current request and therefore the memory conflict increases. The miss rate is lower if some regions have a dedicated cache, even in a smaller sized cache. To study memory access within each region, we conducted an experiment

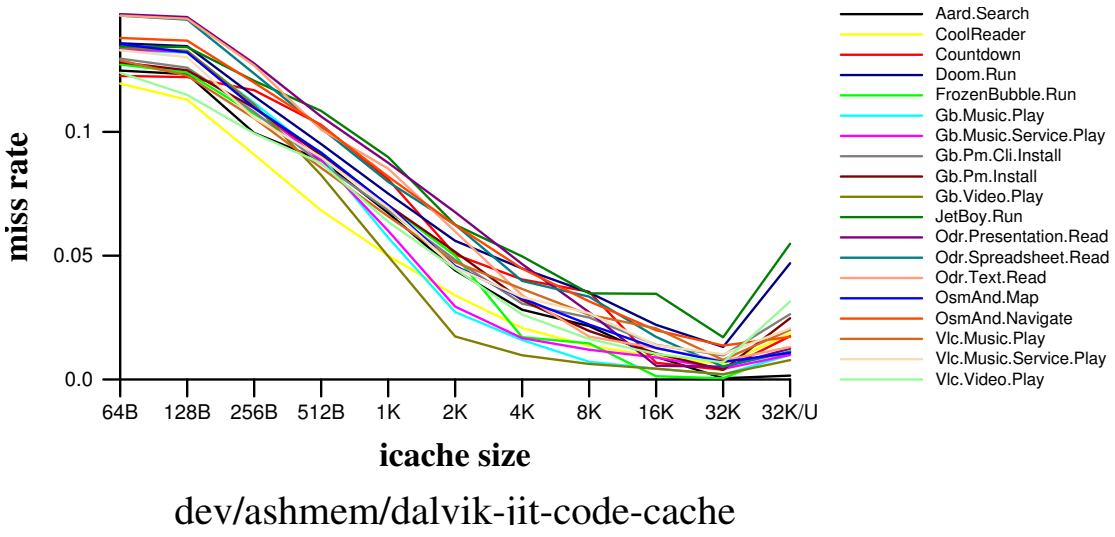

Figure 3.4: Cache miss rate, Simulated for Dalvik JIT in 19 benchmarks

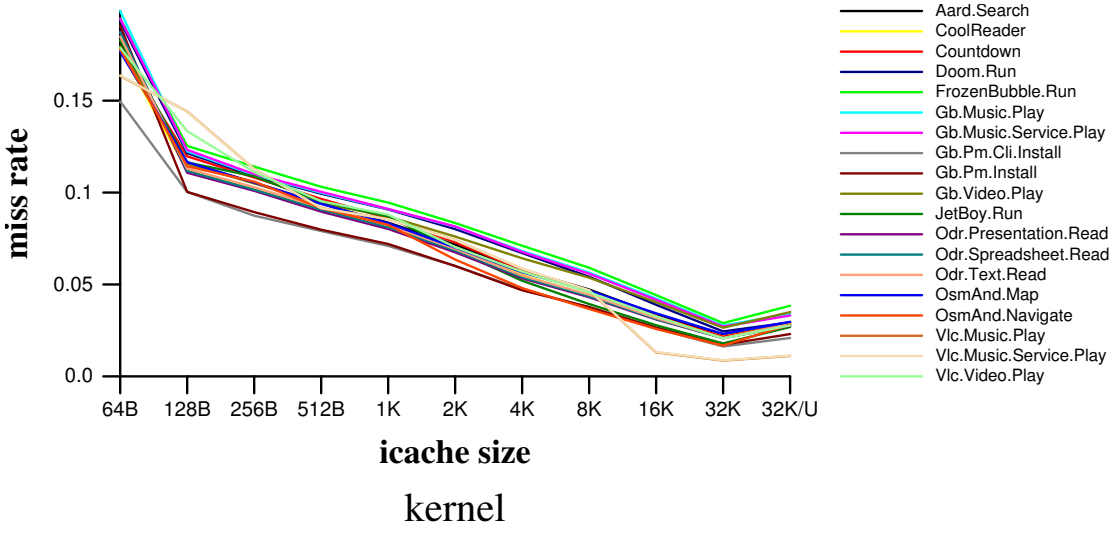

Figure 3.5: Cache miss rate, Simulated for kernel in 19 benchmarks

to look in depth into the cache and log these accesses to show us which regions are more likely to be accessed during the run-time. We will describe these methods in the next section.

3.2 Android Virtual Memory Region Accesses And Profiling Tools

In this section we will discuss our plan for studying and understanding the memory conflicts in the Android virtual memory. We will base our suggestions for region-specific cache memory on what we have achieved from the data presented in the previous section. Here we will introduce these methods and explain why we have picked them for analysing and planning the design of the cache system. In our work we first gathered cache hits and misses, then by parsing the data we created a data base that we use to create two tools for categorising these accesses, namely **conflict matrix** and **reuse distance plot**.

3.2.1 Conflict Matrix

In the cache-logger we can track each cache access and the memory region tagged to that access by looking at the *accesstype()* object in gem5 source code. In this way we will build a matrix to show us how many times each region has has been in conflict with itself or other memory regions. The idea is simple, whenever there is a new access to the cache we check which region is requested. If the requested access is the same region as before, that could potentially be a sell miss, else it is in conflict with another memory region. The only problem here is that we are not tracking the actual cache address that is being called and therefore we don't know which cache line is requested and/or replaced.

3.2.2 Reuse Distance

Once we have access to the *accesstype()*, we can look up the reuse distance between VM accesses. This makes it possible to find the mean reuse distance and this will open many options for adding better cache designs like drowsy cache management or victim cache buffers. Again as previously stated, we can only see which VM is requested and no specific cache line is tracked here which contradicts the actual definition of reuse distance.

3.2.3 Number of Consecutive Accesses to VM

Knowing the number of consecutive accesses within each region will help us to understand the average number of access to each memory region. They help us discover the most accessed regions in different phases during Android applications run time. This can be a key measurement in deciding which regions will most likely benefit from region cache.

3.3 Our Experiment And Data

The latest version of gem5 has a similar structure to the older version. However, the cache is implemented in a slightly different manner from our previous explanation. The CPU port and memory port are implemented within the same part (**cache.cc**) and it is easier to document the cache accesses. The fact that we exported the cache access in the older version by using *ofstream* among other changes in gem5 might have affected the behavior of the simulator (too many changes to the source code could change the accuracy of the simulation). To avoid such error causing method we recommend using the latest version of gem5 with the build-in function *DPRINTF()* that will create a tag for any output information the user is willing to collect. Keep in mind that this function is built-in and therefore might be more reliable in comparison with the method we developed for gathering data. Another difference in the latest version of gem5 is the absence of hard-coded VMA names that Brown et al. had implemented in order to categorize Android memory regions. For instance, almost any memory region without any mapped memory name was tagged anonymous in a previous version which created some difficulties later in our work, specially in the conflict matrix. We do not recommend including anonymous regions in the simulation unless a specific cache block from any of these anonymous regions has a high access rate.

3.4 Building Android on gem5

In this section, we will explain in details how to build an architecture on gem5, boot the Android image on gem5, modify the CPU and memory mapping and finally what we did to generate data and graphs. Please notice that this is a brief explanation of how to build Android on gem5 and we encourage the reader to follow the instructions mentioned in the references for this section to fully understand these steps and pre-requirements.

3.4.1 Build

To build a system such as ARM or X86, one must first compile the desired ISA. Scons is used to set up the *SConscript()* in each sub directory and compile the gem5 source. Scons will automatically generate gem5/build directory and place all files generated by Scons and compiler in that path, it will also determine which component pieces needs rebuilding and rebuild them. Some flags such as **opt** or **fast** can be set to set compilation options, for instance gem5.opt is an optimized binary. The next important option while building gem5 is "-j" flag which is the number of cores on the machine to execute the build (usually number of cores + 1). This will speed up the build time but should be carefully set in order to prevent a crash during the build. The success of the build is normally confirmed by "scons: done building targets." Message at the end of the build. Here is an example of what we build in this project: *scons build/ARM/gem5.fast -j8*

As shown above, we are building a fast ARM system using 7 cores of the Linux machine.

3.4.2 Configuration

The configuration script will allow the user to setup a simple simulation to run on gem5. This script models a system and its parameters such as the number of CPU cores, memory bus connections and memory channel configuration. The script is written in python and defines all system components and parameters for

these components. This config script is later used during the simulation.
gem5 is designed around the*SimObjects* meaning that most system components (CPUs, caches, memory controllers and bus) are sim objects. gem5 implements these objects from C++ classes and implements objects in python. Later, we explain how we modified the C++ class to generate cache data. These objects are all imported from the m5 library in python.
The first sim object is the system. It is the parent for all components and contains much functional information such as CPU, cache, memory size, clock, voltage and the kernel. Later clock and voltage domain is instantiated and the parameters on the clock are set. The memory is normally set to timing mode (as we did in this project), except in some specific cases such as fast-forwarding or checkpoint restore mode, and the memory size is defined. Next, CPU is later created and the CPU model is set. The simplest CPU model is *TimingSimpleCPU* and, as the name suggests, this model executes each instruction in a clock cycle, we used a similar CPU mode in this project since we are not investigating the processor's effect here. Both D and I cache port require a memory bus to connect to the CPU. One important step in the config is connecting the memory components. Each memory object has a master and a slave port. These ports must connect otherwise the request from the master to slave and the response from the slave would not propagate. The final step is to create IO controller on the CPU and memory controller and connect them to the memory bus. (Figure 3.6)

To add the cache to the system, we first create the **Base Cache**. This component is earlier implemented in C++ under *mem/cache/base.hh*, it is later imported as $cxx_h eader$ to the python as a sim object. One advantage of the sim object model is that python can set parameters and create the components from the C class. Here we can set the associativity, hit latency, maximum miss count, mshrs, size, cache level, data vs instruction, LRU tags and many more cache parameters directly in the python object.
In this project, we have instantiated many caches with different sizes and associativity in different cache levels. Once the caches are instantiated, we connect L1

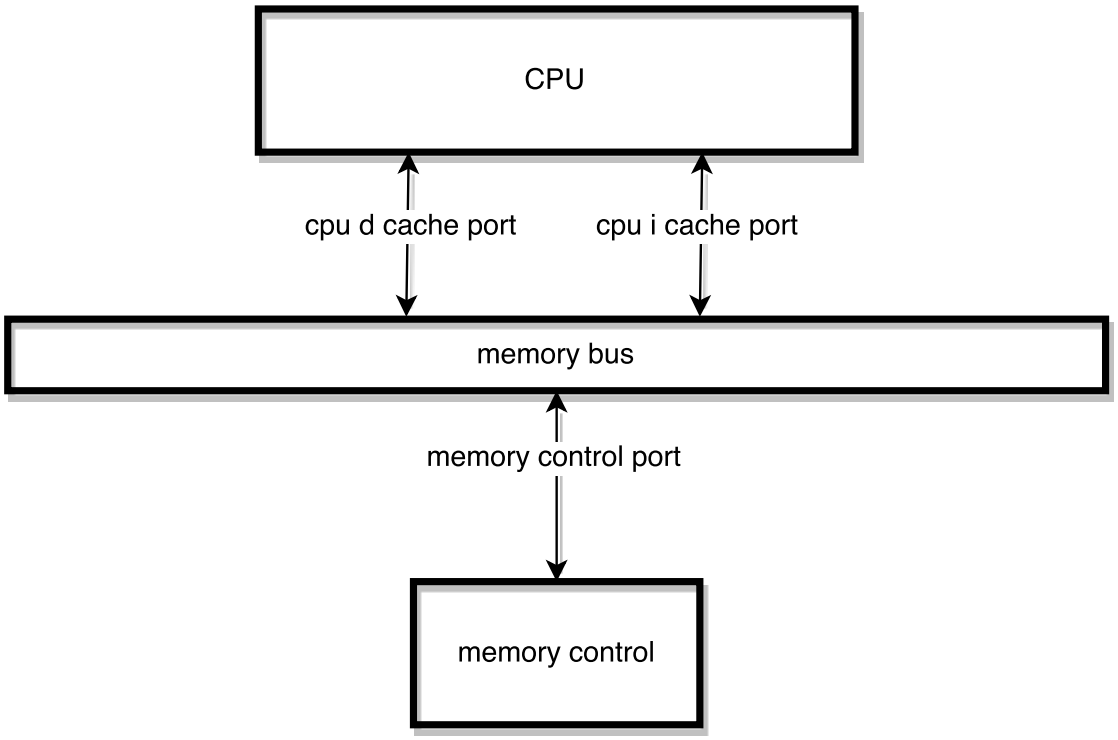

Figure 3.6: gem5 system without the cache

caches to the CPU and an L2 bus (this must be separate for the D and I cache). The final step is to connect the L2 cache to the L2 bus and the memory bus we explained in previous paragraph (Figure 3.7). We added some changes to the C++ class in gem5 to give us the initial data we described in the section 3.1.2. We will discuss these changes later in the section 3.4.

3.4.3 Running gem5

Once the configuration is done we can run gem5 simulation by running the build and the configuration. In the example, above: we would run:
build/ARM/gem5.fast configs/myConfig.py.
However, in this project we automated most steps by making multiple bash scripts to run the benchmarks for the Android e.g.:
./start_new_checkpoint.sh init
In addition to these changes, we have added a mechanism in the script to run the simulation with different cache sizes which we will explain later. [1]

[1] http://pages.cs.wisc.edu

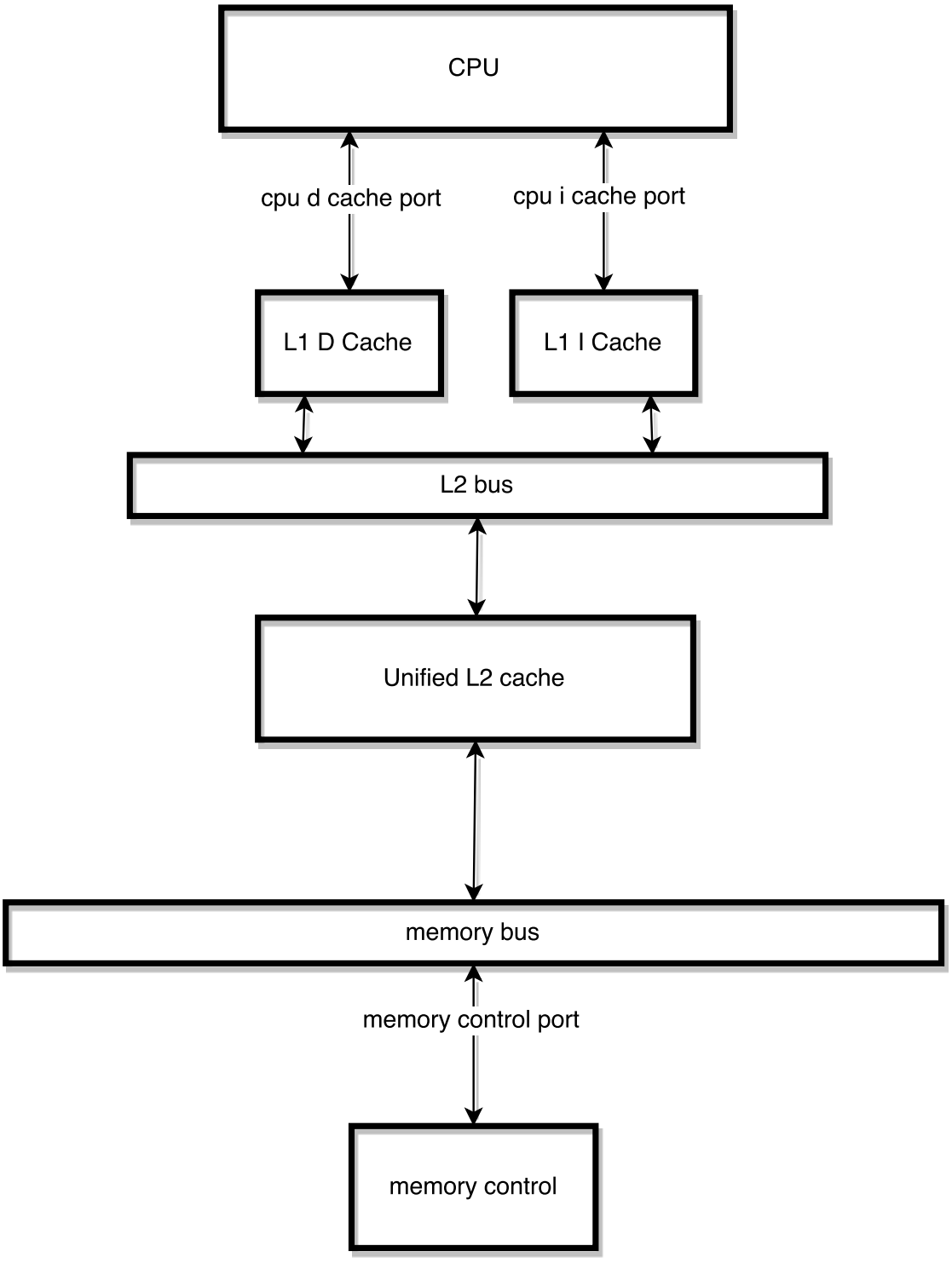

Figure 3.7: gem5 system with two cache levels

3.4.4 Modified gem5

In gem5 source code (structured in C++), every memory request is propagated from the CPU side towards the cache, as explained previously. These requests come from the master port and are responded to by the slave port. In our project,

we modified both CPU requests and cache responses so that the simulation would run two modes: First, the traditional mode which would discard our change to the system and work as explained in the sections above. Second, the VM[2] mode which would iterate through the simulation and recreate the same request while dynamically changing the L1 cache size with each request. This will create a simulation where the cache size would have an impact on the hit rate. The effects of the VM mode are already discussed the section 3.1.2. In addition to the changes to the master port, we added logging functionality to the slave port (cache side) to trace these requests and hits for further investigation. Here we refer to such tool as the *cache logger*. The results from these traces are discussed in the conclusion chapter. Notice that the VM mode and cache logger are not affecting the actual system nor the simulation performance. The one exception is the tick number which is the internal simulation clock; as we are not investigating the memory request delays in this project, this impact is negligible.

In addition to the system changes and gem5 structure, we are not running the benchmarks immediately from the simulation start up unlike most simulations in gem5. Instead, we first boot the Android on the ARM architecture and when the boot time is over we create a checkpoint. from the checkpoint we run the benchmark in the m5 terminal connected to the simulated console interface (see previous section for more information about the benchmarks and their contents). The boot time simulation is therefore not logged. The checkpoint is created in the m5 terminal and is based on the system's internal clock (tick number). The benefit of the checkpoint system is that we can terminate the simulation at any point and continue later from that point without any impact on the system behaviour. See the gem5 manual and gem5 official web page for more information.[3]

[2]Virtual Memory

[3]www.gem5.org

3.4.5 Building Android for gem5

The most common way to build Android for gem5 is to base the build on the emulated Goldfish[4] with some differences in the gem5-specific configuration files. These files include block device naming and some scripts to start gem5. To build Android on gem5 we follow 3 steps.

i. Build Android: The official Android binary can be downloaded from Google's driver page. These binaries contain hardware capabilities. After extracting the files, we set up the environment and choose a target to build using lunch for the Android. Here we used ARM as a target, this will refer to a build for the emulator. Next we built the code using "make" and GNU's parallel task handler with the "-j" flag to speed up the build process.

ii. Preparing a Filesystem for gem5: We create a 2GB empty disk image. Later, we partition the disk into 3 partitions **root** for containing Android system files, **data** for the applications to be installed later and **cache**. Keep in mind that the partitioning can be different depending on the Android version. Up to this point we have created a disk image like the emulator image. To make the image suitable for the gem5 we add the gem5 specific file system and m5 binaries. Some extra scripts are available from the gem5 official web page to unlock the Android lock screen. The disk image is then added to the machines mnt/path and the $M5_PATH$ is added to the bashrc in the Linux machine.

iii. Building the Kernel: ARM cross compiler is used to build the kernel. For this project, we used a kernel developed by Florida State University specifically tailored for the Agave benchmark suite. Here we will not explain the process and instead encourage the reader to see the gem5 official guides for building Android marshmallow and KitKat. After this step, we can run the Android on gem5 as explained in section 3.4.3. [5]

[4]Goldfish is an QEMU based Android emulator.
[5]http://www.gem5.org/AndroidKitKatResources

3.4.6 Plotting the Results

When gem5 finishes the simulation three files are generated automatically, namely the config.ini, config.json and stats.txt. These files can be directed to any preferred path to reside. The output path is also defined in the simulation start up with the -d flag (–outdir). The ini file consists of all parameters and simobject, the json file is like the ini file and the txt file has the statistics. Each gem5 object has its own statistic and at the end of simulation a statistic-dumping command will log all statistics for all objects. However, this statistic is not everything that we required in this work. For instance, the statistics for the L1 cache shows the total number of references as well as hit rate but there is no information on which memory references were accessed nor which clock cycle they may have missed. To solve this problem, we added our own technique to log these details along with the stats gathered by gem5 [6].[7]

Once the memory accesses were logged we parsed them using parsing scripts we developed and created two sources for analysing the logs: The *dict* files and the *pts*. The dict files are simply dictionaries created by python parsing the stat logs (using **ParseVMAStats.py**) and contain name of the top 9 used memory regions, their race to memory resources with other memory regions and their hit rate. The dict files are later used in another script, **MatrixBuiler.py**, to generate the mat file which is our final conflict matrix.

The pts file consists of logged information on when each region was recalled. This information is essential for plotting the reuse distance as a method of analysing the cache behavior and designing a drowsy cache [8]. Finally, we plotted the graphs for reuse distance using **GraphStats.py** developed in python. All the scripts mentioned above are available in Appendix B.

[6] In the latest version, this is not needed since the DPRINTF tracing tool allows us to do as such without requiring these modifications, but at the time of this project this option was not available.

[7] http://pages.cs.wisc.edu

[8] As mentioned earlier here we looked only at which memory region was recalled and not the exact cache line that was requested.

Chapter 4

Findings

The normalized memory access graphs for all simulated benchmarks show that data reads are always higher than data writes and there are no instruction writes, this suggests that basing the memory structure design and our analysis on the data and instruction reads can benefit the system more and will give us a memory structure that is more likely to have increased hit rate in comparison with a design that includes the data writes. Therefore, in the next part of this section we are going to analyse and suggest a cache design on the data and instruction reads. (Figures 4.1 and 4.2)

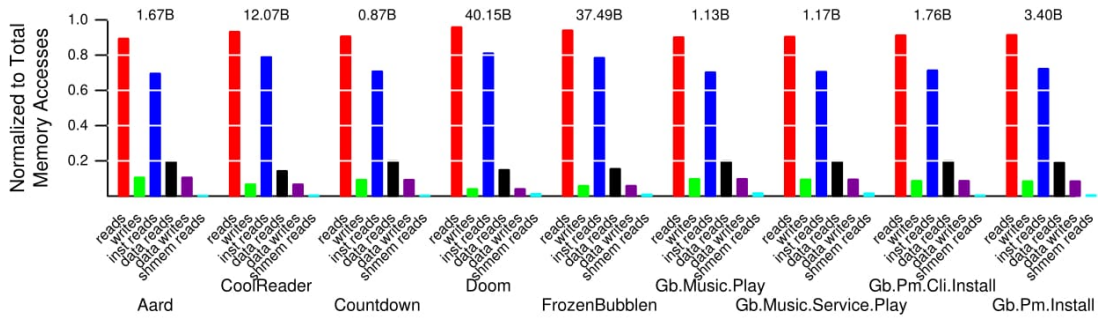

Figure 4.1: Normalized memory access graphs for all Agave benchmarks

considering the normalized **data reads** while running 18 different benchmarks we see that heap and dalvik heap accesses are among the highest (Figure 4.3), this suggests that having a heterogeneous memory structures such as region-based caching to replace the single unified data cache with multiple smaller sized cache and allocating one of these regions solely to heap and dalvik-heap can benefit the

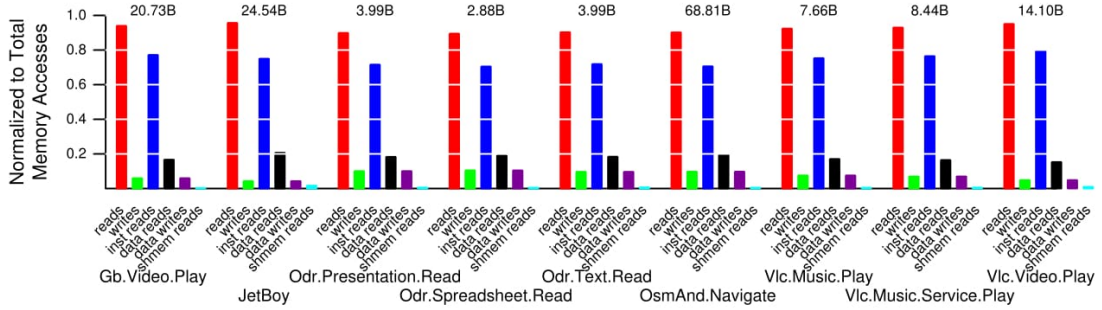

Figure 4.2: Normalized memory access graphs for all Agave benchmarks(continued)

memory structure in terms of memory access and will speed up the memory in Android. Please notice that second to heap (or even in some benchmarks above the heap access) lies the anonymous which refers to the memory regions that were not mapped to the Android libraries and therefore allocating one region to all anonymous regions does not have any research value for us. This problem was caused by the gem5 version that we based our research, since the mentioned version did not have any mapping to the memory access we had to map some memory regions to known libraries and the rest were mapped to anonymous.

In some of the benchmarks (mostly game applications) gralloc-buffer is accessed third most accessed after heap and anonymous. *gralloc* is the Android graphic memory allocator. It is used to allocate memory requested by image processor. The buffer queue is the link between the graphic blocks. The high access rate to gralloc suggests a cache region to be allocated to this memory region, however considering the rest of the (none graphical benchmarks) this idea seems unnecessary and since the total amount of access to the rest of the memory regions is higher than gralloc (including anonymous) we would not recommend a specific region for gralloc.

In conclusion considering the data memory access we suggest dividing the single L1 data cache into region-based cache with two regions: heap and common. Figure 4.3 Shows the data accesses to the data cache during the runtime for a game benchmark called Frozen Bubble. This is only an example and the rest of the data access graphs are available in Appendix A.

The conflict Matrix shown in table 4.4 is the sum of conflicted accesses table after

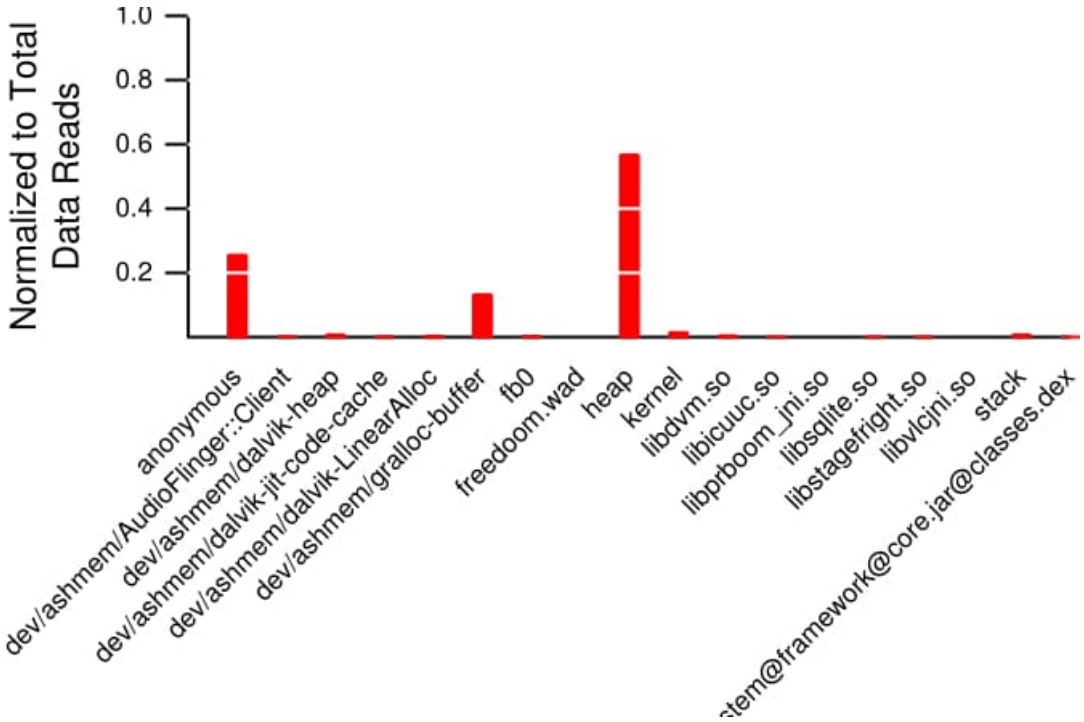

Figure 4.3: VMA data cache accesses during Frozen Bubble run time

running 4 different benchmarks on a direct mapped cache system [1]. This table supports the idea of having a separate cache for the heap and dalvik heap since heap (marked as index 0) and dalivk heap (marked as 2), have the most cache conflicts among the regions. The idea of having a two region L1 data cache seems to be a good design. Notice that here we don't include the **other** since other is a combination of least accessed regions as well as uncharted VM regions that have no accurate information at this point, however, in the latest gem5 version the ability to trace these regions are added but that information is not a part of this project. Also, gralloc is not among the top ten since we did not include graphic heavy benchmarks (such as games) in our conflict matrix and the only graphical benchmark is a video player. and therefore, gralloc was not among the highest.

The **instruction access** charts show a different variety of memory access than the data accesses. Here the libdvm.so and dev/ashemem/mspace win the two most memory regions accessed during the runtime of many benchmarks. libdvm is a part of Dalvik runtime dalvik is the process VM responsible for executing the

[1]Please keep in mind that some of these regions are not used by all benchmarks but since their total access count was the highest among all 4 benchmarks, we have included them in our study

	0:heap	1:libsqlite.so	2: dalvik-heap	3: libc.so	4: libicuuc.so	5: dev/ashmem/dalvik-bitmap-2	6: system@framework@core.jar@classes.dex	7: libandroid_runtime.so	8: libskia.so	9: other
0:heap	842764	0	15807	5957	0	4333	6224	6063	99	8332
1:libsqlite.so	0	76054	0	136	200	0	0	482	0	32379
2: dalvik-heap	4283	0	133458	29	0	38	417	15	0	52
3: libc.so	1151	168	16	50611	94	30	185	399	211	1893
4: libicuuc.so	0	7	0	47	19819	0	9	18	0	172
5: dev/ashmem/dalvik-bitmap-2	1553	0	2175	36	0	26811	590	2263	280	24435
6: system@framework@core.jar@classes.dex	6997	0	248	458	0	314	31794	50	0	964
7: libandroid_runtime.so	11847	17778	0	193	559	2868	282	15378	7721	19038
8: libskia.so	2559	0	0	311	0	0	0	518	25341	215
9: other	2975	9937	2217	6551	855	24950	367	9208	417	155884

Figure 4.4: VMA data cache conflict matrix

applications. mspace is a part of ashmem which is the [2] Android shared memory and is similar to POSIX [3] shared memory with some differences in behavior. It has better performance in low memory devices, since it can discard shared memory if there is a memory pressure. These two memory regions and their reference popularity suggests that a region based L1 instruction cache with three regions: one for libdvm, one for ashmem and one common for the rest of the instruction regions. Figure 4.5 shows the VMA access for Aard benchmark please see the Appendix A for the rest of the graphs made for Instruction memory accesses.

The Android runtime(a part of the libdvm.so), as shown in the conflict matrix table 4.6, is the third highest conflicted region after other and anonymous. Since anonymous is the uncharted regions and other represents a combination of least accessed regions. it seems that allocating a region to Android runtime can improve the hit rate, thus supporting the idea of a region L1 instruction cache. As illustrated in table 4.6, ashmem/mspace is not in the top ten conflicts. This can be a good argument that mspace might not require a separate region cache since the miss rate is already low. However as mentioned earlier this could not be confirmed in this project and requires more experiment.

[2]A shared memory allows multiple programs to simultaneously access the memory to avoid unnecessary copies.

[3]Portable Operating System Interface (for Unix).

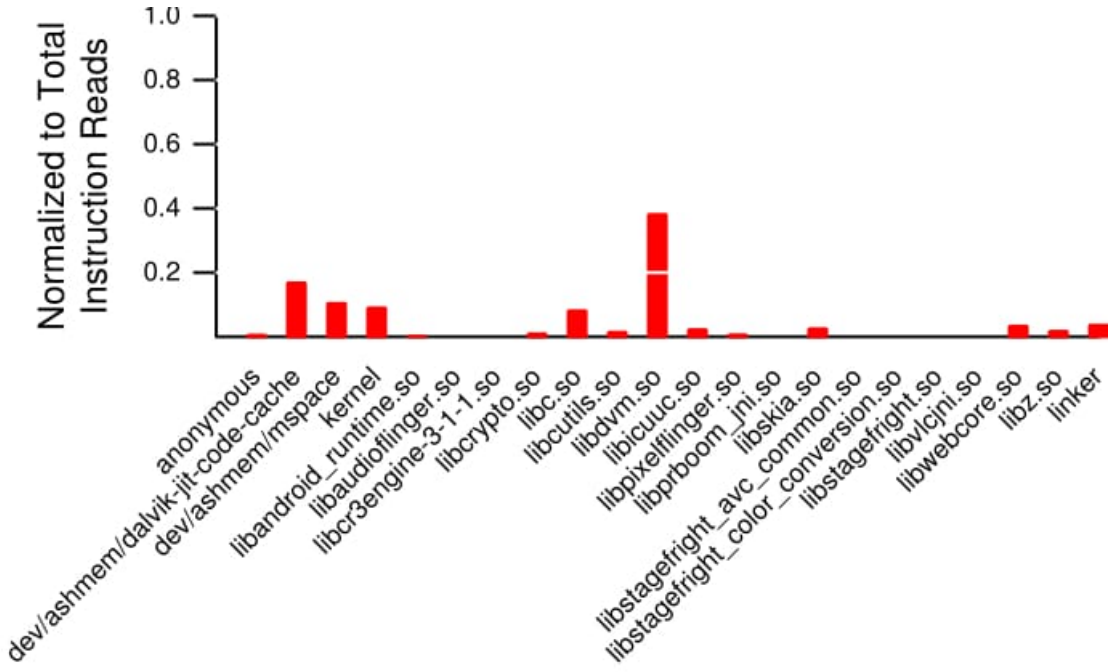

Figure 4.5: VMA instruction cache accesses during Aard run time

	0: linker	1: libicuuc.so	2: libcutils.so	3: anonymous	4: libandroid_runtime.so	5: libutils.so	6: libui.so	7: libm.so	8: libz.so	9: other
0: linker	2193	0	0	10	5	0	0	0	0	63
1: libicuuc.so	0	15132	1	0	0	0	0	0	0	42
2: libcutils.so	0	0	14566	0	18338	184	1	1	0	143
3: anonymous	0	0	330	43645	0	0	0	0	0	18
4: libandroid_runtime.so	0	0	370	1	25309	785	127	127	3	881
5: libutils.so	0	0	390	1	101	9941	72	0	9	677
6: libui.so	0	0	14	0	2	1261	8090	0	0	495
7: libm.so	0	0	2	0	0	0	0	366	0	0
8: libz.so	0	0	0	0	2	0	0	0	772	1
9: other	129	162	165	95	181	989	1069	10	5	50731

Figure 4.6: VMA instruction cache conflict matrix

Both design suggestions for I and D cache mentioned in this chapter are based on our observations from 18 benchmarks running on gem5 simulator with ARM processor model and Android gingerbread image booted. However, the size of these regions and whether if we can benefit from these designs in an actual system was not studied during this project due to the large scope of such work. A more detailed research could be to separate a single L1 Instruction cache into two regions: one

common region and one region with libdvm and ashmem combined. This way it is possible to see the memory hit rate improvement.

At the moment, a research lead by Zachary Yannes is performed in Florida State University Android Lab to evaluate this design on a more reliable and robust version of gem5 with Android marshmallow booted on.

Chapter 5

Discussion

In this project we were able to understand the behavior of the gem5 simulator while running Agave benchmark suite. We obtained data on virtual memory accesses. As planned we were able to simulate many traditional cache systems with different cache size and associativity. We have looked at different innovative memory organisations and for understanding the Android virtual memory conflict we studied the memory references even more in details.

Unfortunately due to the big scope of the project we did not use CACTI to simulate and understand the power consumption in Android platforms. Other works such as Lee et al. [7] have shown that leakage power can significantly be reduced by using the specific memory architectures.

Later in this project we were able to upgrade the operating system from Gingerbread to Kit-Kat, because Gingerbread is out of dated and it lacks many libraries we use in modern Android Smart phones. This showed us many more VMAs that we did not study before and opened opportunities for understanding the GPU behavior in Android mobile platforms. However, due to the complexity of the new gem5 version we were not able to generate cache data and therefore the results from Android kit-kat is not presented here.

Appendix A

VMA Accesses for D and I Cache in four Benchmarks

A.1 VMA Data Access

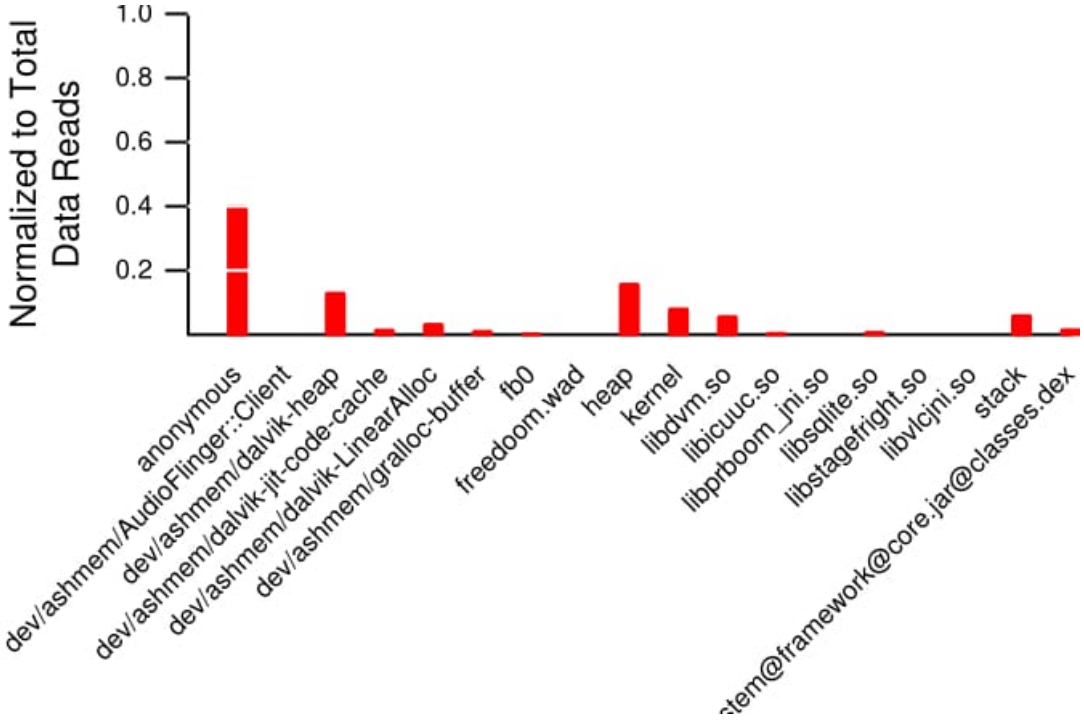

Figure A.1: VMA data cache accesses during Aard run time

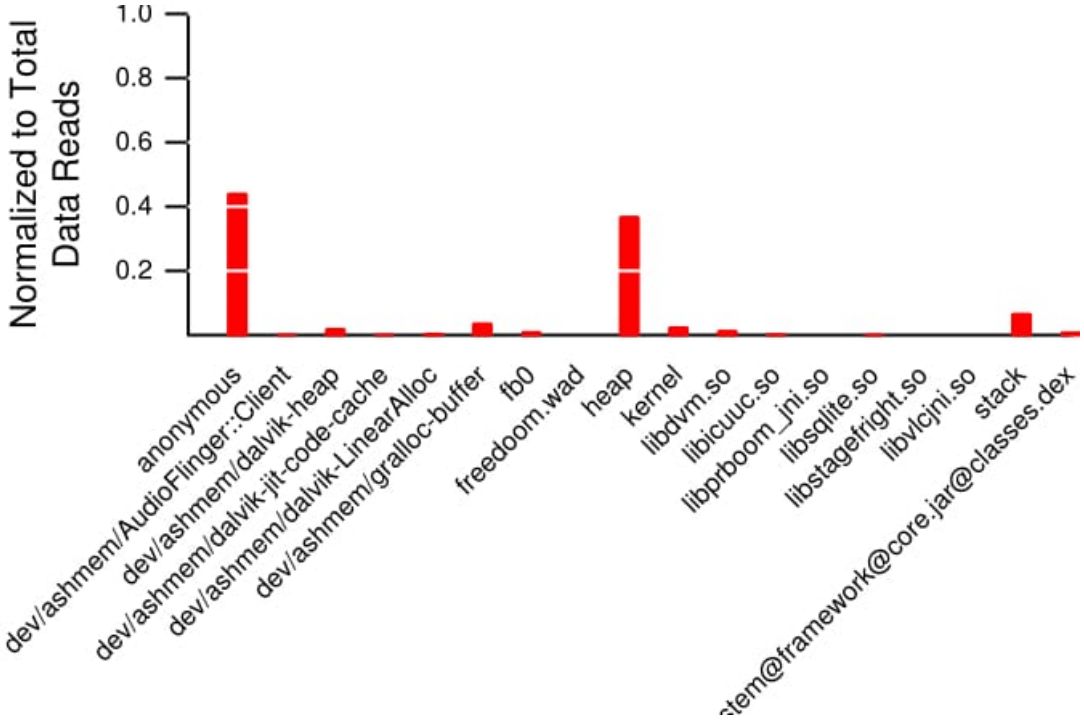

Figure A.2: VMA data cache accesses during Coolreader run time

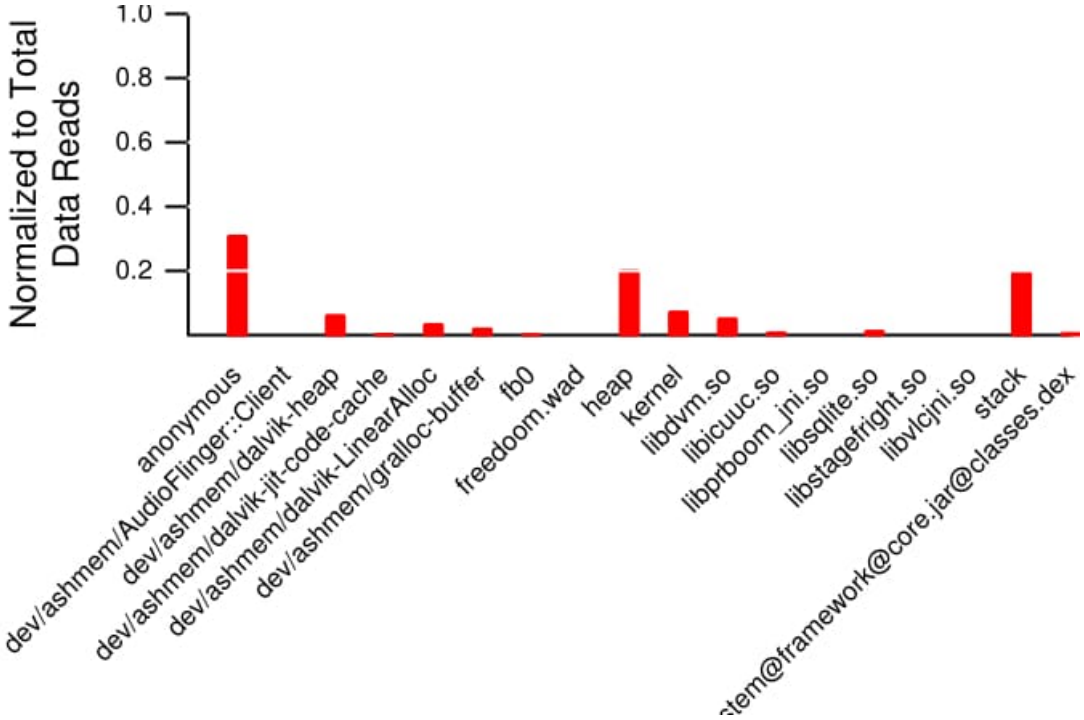

Figure A.3: VMA data cache accesses during Countdown run time

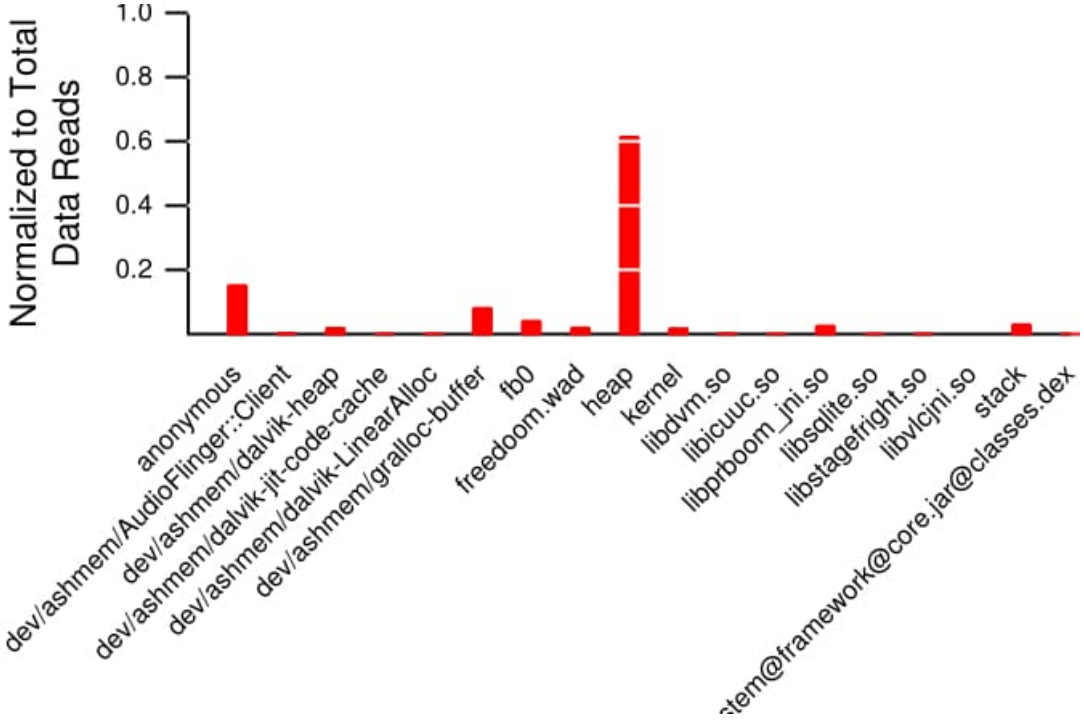

Figure A.4: VMA data cache accesses during Doom run time

A.2 VMA Instruction Access

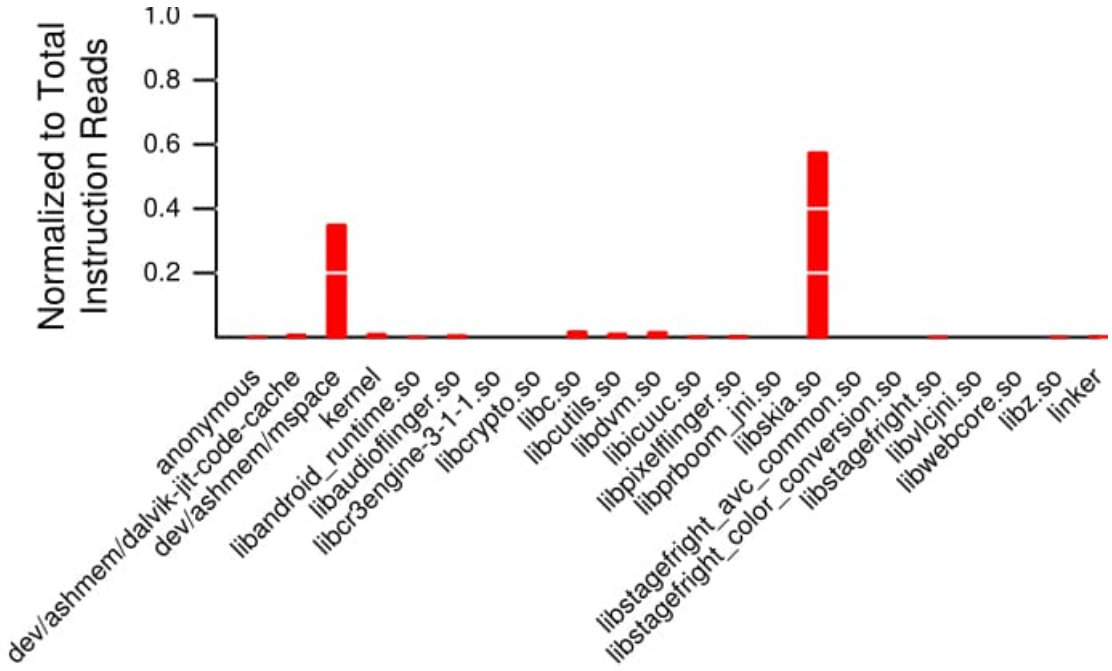

Figure A.5: VMA instruction cache accesses during FrozenBubble run time

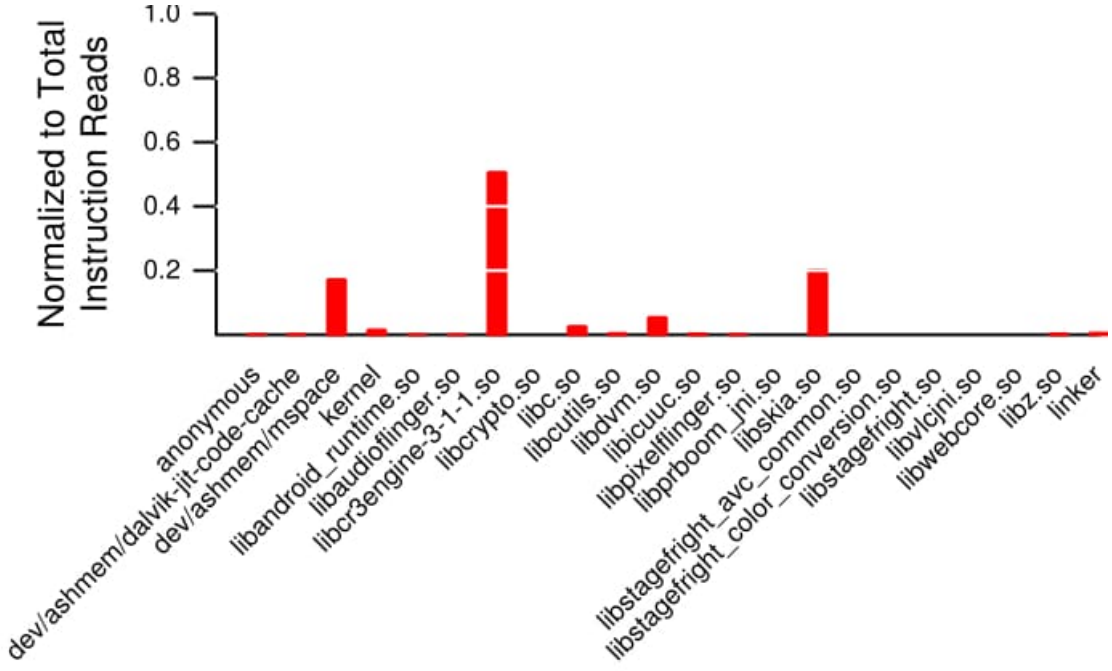

Figure A.6: VMA instruction cache accesses during Coolreader run time

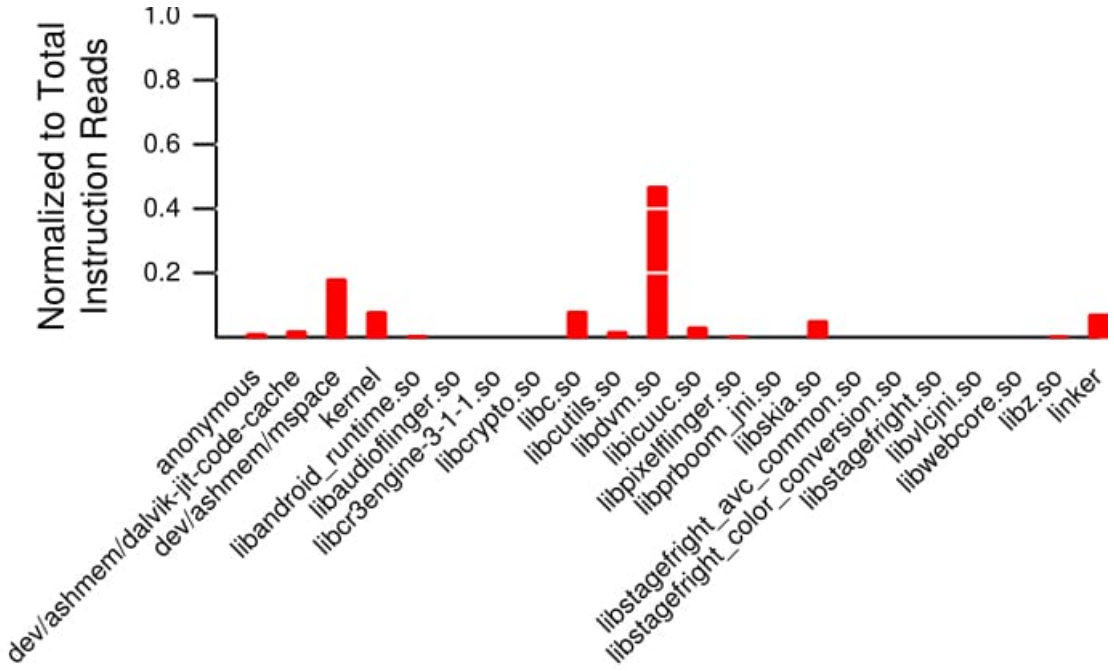

Figure A.7: VMA instruction cache accesses during Countdown run time

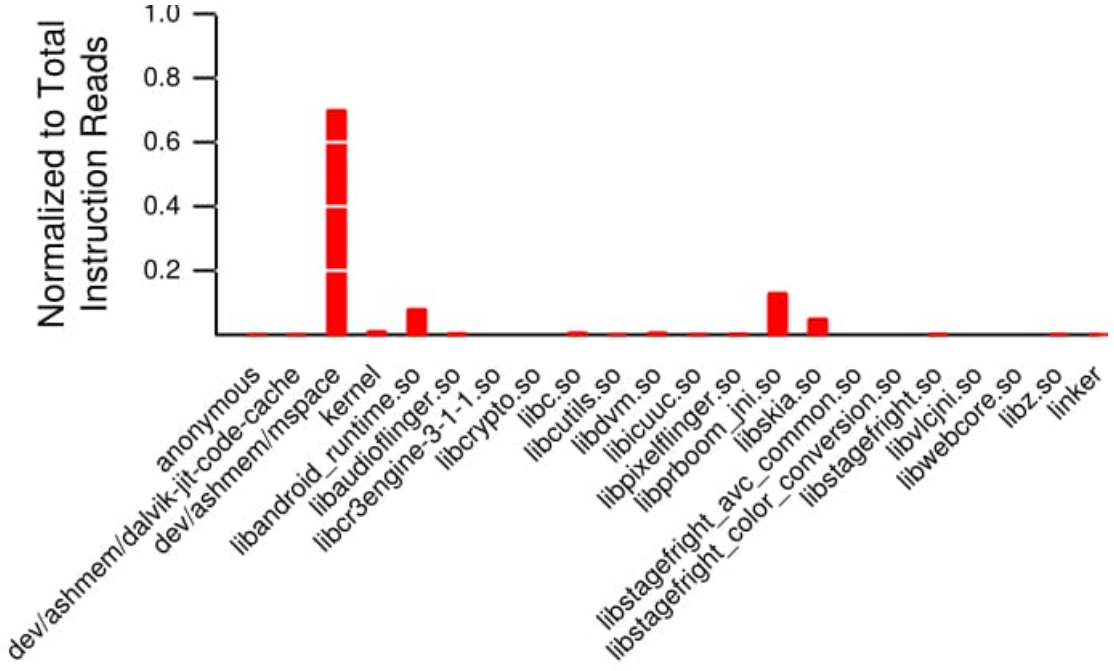

Figure A.8: VMA instruction cache accesses during Doom run time

Appendix B

Scripts

B.1 gem5 Run Script

```
#!/bin/bash

# var="booting"
# if [[ -z "$var" ]]; then
if [ "$#" -lt 1 ]; then
    echo "Usage: $0 <benchmark>"
    exit 0
fi
var="$1"
checkpoint=""
dtb="vexpress-v2p-ca15-tc1-gem5.dtb"
# dtb="vexpress-v2p-ca15-tc1-gem5-gpu.dtb"
kernel="vmlinux" # .unaligned"
# kernel="vmlinux.gpu" # .unaligned"
if [ "$#" -eq 2 ]; then
    checkpoint="-r $2"
fi
export M5_HOME=/home/$USER/gem5_kitkat/gem5_github/gem5_backup
export M5_PATH=/home/$USER/gem5_kitkat/gem5_github/gem5_backup/system

simple_opts="--outdir=benchmarks/$var \
 --stats-file=${var}_stats_region_not_needed.txt \
 configs/example/fs.py \
 --kernel=${kernel} \
 --disk=kk.dalvik.4g.img --cpu-type=atomic \
 --dtb-file=${dtb} \
 --os-type=android-kitkat --num-cpus=1 --mem-size=2GB"

opts="${simple_opts} \
 --caches \
 --l1d_size=32kB \
 --l1i_size=32kB \
 --l1d_assoc=2 \
 --l1i_assoc=2 \
 --l2cache --l2_size=1024kB --l2_assoc=16 \
 --cacheline_size=64 \
  ${checkpoint}"

# opts_debug="--debug-flags=CacheAll --debug-file=${var}_trace.txt"
# opts_debug="--debug-flags=CacheAccess --debug-file=${var}_trace.txt.gz"

cmd="build/ARM/gem5.fast ${opts_debug} ${opts}"
flags="LD_LIBRARY_PATH=/home/$USER/gem5_kitkat/gem5_github/dev/toolchain/x86_64-unknown-linux-gnu/
sysroot/lib /home/$USER/gem5_kitkat/gem5_github/dev/toolchain/x86_64-unknown-linux-gnu/lib/ld-
linux-x86-64.so.2"

benchFile="${M5_HOME}/benchmarks/${var}/system.framebuffer.bmp"
monitor_cmd="/home/$USER/gem5_kitkat/gem5_github/monitor-gem5/monitor-gem5.sh ${benchFile}"
# touch ${benchFile}
# ${monitor_cmd} &

echo -e "Running ${flags} ${cmd}\n"

time LD_LIBRARY_PATH=/home/$USER/gem5_kitkat/gem5_github/dev/toolchain/x86_64-unknown-linux-gnu/
lib /home/$USER/gem5_kitkat/gem5_github/dev/toolchain/x86_64-unknown-linux-gnu/lib/ld-linux-
x86-64.so.2 ${cmd}
```

Figure B.1: gem5 automated script for running a benchmark and initializing system parameters

B.2 Python Parsing and Plotting Scrpts

```
1  #!/usr/bin/python2.7
2  import os, sys, urllib
3  import re
4  import pickle
5  # import matplotlib.pyplot as plt
6  import numpy as np
7  #from numpy.random import *
8
9  ACCESS_FORMAT_V2 = re.compile('Tick: (?P<tick>(\d+)),Configuration: ,\s*(?P<cachesize>(\d+)[KB])
   \s*,\s*(?P<association>(\d+)-way)\s*,\s*(?P<cachetype>[\w\.]+)\s*,\s*(?P<cachename>\w+),\s*VMA
   Accessed: (?P<vma_name>[^,]+),\s*VMA Stats: (?P<access_result>\w+)(,\s*Evicting: (?P<evicted_vma>
   [\S\.]+))?')
10
11 CACHETYPE_DATA = 1
12 CACHETYPE_INSTR = 2
13 cacheType = 0
14
15 statsKeys = ['index', 'tick', 'prevLib', 'hit', 'othermiss', 'selfmiss']
16
17 def parseAccess(line):
18     m = re.match(ACCESS_FORMAT_V2, line)
19     if not m:
20         print 'Error: cannot parse line "%s"' % (line)
21         return None
22     access = m.groupdict()
23
24     if cacheType != CACHETYPE_DATA or cacheType != CACHETYPE_INSTR:
25         setCacheType(access)
26
27     return access
28
29 def setCacheType(access):
30     global cacheType
31     if 'dcache' in access['cachetype']:
32         cacheType = CACHETYPE_DATA
33     elif 'icache' in access['cachetype']:
34         cacheType = CACHETYPE_INSTR
35
36 def printStats(stats, currentLib):
37     print 'Library %s: hits %d, othermiss %d, selfmiss %d' % (currentLib,
38                                                               stats[currentLib]['hit'],
39                                                               stats[currentLib]['othermiss'],
40                                                               stats[currentLib]['selfmiss'])
41
42 def updateStats(stats, access, currentLib, evictedLib):
43     try:
44         if 'Miss' in access['access_result']:
45             if currentLib == evictedLib:
46                 stats[currentLib]['selfmiss'] += 1
47             else:
48                 stats[evictedLib]['othermiss'] += 1
49         elif 'Hit' in access['access_result']:
50             stats[currentLib]['hit'] += 1
51         else:
52             print 'Error: invalid access result %s' % (access['access_result'])
53     except KeyError as e:
54         print 'Error: invalid libname %s' % (evictedLib)
55     return (stats)
56
57 def updateList(vmaList, libName):
58     if libName in vmaList:
59         vmaList[libName] += 1
60     else:
61         vmaList[libName] = 1
62
63     return vmaList
64
65 def accessListBuilder(statsDict):
66     vmaStats = {}
67
68     ticks = sorted([tick for tick in statsDict])
69     for i, tick in enumerate(ticks):
70         prevLib, currLib, hits, othermiss, selfmiss, runIndex = statsDict[tick]
71         accessCount = hits + othermiss + selfmiss
```

Figure B.2: Python script for parsing the simulation result and generating the .dict for building the matrix and graphing the VMA Accesses

```
#!/usr/bin/python2.7
import os, sys, urllib
import re
import pickle
import matplotlib.pyplot as plt
import numpy as np
from numpy.random import *

ACCESS_FORMAT_V2 = re.compile('Tick: (?P<tick>(\d+)),Configuration: ,\s*(?P<cachesize>(\d+)[KB])
\s*,\s*(?P<association>(\d+)-way)\s*,\s*(?P<cachetype>[\w\.]+)\s*,\s*(?P<cachename>\w+),\s*VMA
Accessed: (?P<vma_name>[^,]+),\s*VMA Stats: (?P<access_result>\w+)(,\s*Evicting: (?P<evicted_vma>
[\S\.]+))?')


CACHETYPE_DATA = 1
CACHETYPE_INSTR = 2
cacheType = 0
vmaList = {}

def readVMADict(filename):
    global vmaList
    with open(filename, 'r') as f:
        s = f.read()

    vmaList = eval(s)
    print vmaList

def readStatsDict(filename):
    with open(filename, 'r') as f:
        s = f.read()

    statsDict = eval(s)
    return statsDict

def accessFinder(myString):
    matrixIndex = 9
    for key in vmaList.keys():
        if myString and (key.lower() in myString.lower()):
            matrixIndex = vmaList[key]
            break
    return matrixIndex

def getSimpleLibName(libName):
    simpleLibName = 'other'
    for i, lib in enumerate(vmaList):
        if (libName in lib) or (lib in libName):
            simpleLibName = lib
            break

    return(simpleLibName)

def printMatrix(mat, outfile):
    vmas = [(vmaList[key], key) for key in vmaList]
    vmas = sorted(vmas, key = lambda x: x[0])

    print vmas
    print mat

    with open(outfile, 'w') as f:
        vmaStr = ', '.join(['%d: %s' % (index, vma) for index, vma in vmas])
        f.write(vmaStr + '\n')
        f.write(str(mat))

def printStats(stats, currentLib):
    print 'Library %s: hits %d, othermiss %d, selfmiss %d' % (currentLib,
                                                              stats[currentLib]['hit'],
                                                              stats[currentLib]['othermiss'],
                                                              stats[currentLib]['selfmiss'])

def matrixBuilder(filename):
    Matrix = [[0]*10 for i in range(10)]

    stats = {} # dict.fromkeys(['hit', 'othermiss', 'selfmiss'])
    statsDict = {}
```

Figure B.3: Python script for generating the conflict matrix

```
#!/usr/bin/python2.7
import os, sys, urllib
import re
import pickle
import matplotlib.pyplot as plt
import numpy as np
from numpy.random import *

ACCESS_FORMAT_V2 = re.compile('Tick: (?P<tick>(\d+)),Configuration: ,\s*(?P<cachesize>(\d+)[KB])
\s*,\s*(?P<association>(\d+)-way)\s*,\s*(?P<cachetype>[\w\.]+)\s*,\s*(?P<cachename>\w+),\s*VMA
Accessed: (?P<vma_name>[^,]+),\s*VMA Stats: (?P<access_result>\w+)(,\s*Evicting: (?P<evicted_vma>
[\S\.]+))?')

# dvmaList={"anonymous":0,"heap":1,"stack":2,"kernel":3,"gralloc-buffer":4,"dalvik-
heap":5,"fb0":6,"libdvm.so":7,"dalvik-LinearAlloc":8}
# ivmaList={"mspace":0,"libdvm.so":1,"libskia.so":2,"kernel":3,"app
binary":4,"libstagefright.so":5,"dalvik-jit-code-cache":6,"libc.so":7,"libcr3engine-3-1-1.so":8}

CACHETYPE_DATA = 1
CACHETYPE_INSTR = 2
cacheType = 0
vmaList = {}

def readVMADict(filename):
    global vmaList
    with open(filename, 'r') as f:
        s = f.read()

    vmaList = eval(s)
    print vmaList

def readStatsDict(filename):
    with open(filename, 'r') as f:
        s = f.read()

    return eval(s)

def accessFinder(myString):
    matrixIndex = 9
    for key in vmaList.keys():
        if myString and (key.lower() in myString.lower()):
            matrixIndex = vmaList[key]
            break
    return matrixIndex

def getSimpleLibName(libName):
    simpleLibName = 'other'
    for i, lib in enumerate(vmaList):
        if (libName in lib) or (lib in libName):
            simpleLibName = lib
            break

    return(simpleLibName)

def printStats(stats, currentLib):
    print 'Library %s: hits %d, othermiss %d, selfmiss %d' % (currentLib,
                                                              stats[currentLib]['hit'],
                                                              stats[currentLib]['othermiss'],
                                                              stats[currentLib]['selfmiss'])

def updateStats(stats, access, currentLib, evictedLib):
    # hits + othermisses + selfmisses = total_accesses
    # print 'Updating stats for access: ', access
    try:
        if 'Miss' in access['access_result']:
            if currentLib == evictedLib:
                # print 'Self-evict: %s selfmiss++' % (currentLib)
                stats[currentLib]['selfmiss'] += 1
            else:
                # print 'Other-evict: %s othermiss++' % (evictedLib)
                stats[evictedLib]['othermiss'] += 1
        elif 'Hit' in access['access_result']:
            # print '%s hit++' % (currentLib)
            stats[currentLib]['hit'] += 1
```

Figure B.4: Python script for graphing the reuse distance

Bibliography

[1] M. Brown, Z. Yannes, M. Sanati, M. Lustig, A. Sidelnikov, S. McKee, G. Tyson, and S. Reinhardt, "Agave: a Benchmark Suite Addressing Android System Complexity," in *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2016. Accepted as poster.

[2] J. Mander, "80% of Internet users own a smartphone." `http://www.globalwebindex.net`. Online; accessed 15-January-2016.

[3] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating system and multiprogramming workloads," *ACM Transactions on Computer Systems (TOCS)*, vol. 6, pp. 393–431, Nov. 1988.

[4] F.Bodin and A. Seznec, "Skewed associativity enhances performance predictability," in *Proc. 22nd ACM International Symposium on Computer Architecture*, pp. 265–274, June 1995.

[5] A. Agarwal and S. Pudar, "Column-associative Caches: A Technique for Reducing the Miss Rate of Direct-mapped Caches," in *Proc. 20th ACM International Symposium on Computer Architecture*, pp. 179–190, May 1993.

[6] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. 17th ACM International Symposium on Computer Architecture*, pp. 364–373, 1990.

[7] H. Lee and G. Tyson, "Region-based caching: An energy-delay efficient memory architecture for embedded processors," in *Proc. 4th ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 120–127, Nov. 2000.

[8] M. Bhadauria, S. McKee, K. Singh, and G. Tyson, "A precisely tunable drowsy cache management mechanism," in *Proc. IBM T.J. Watson Conference on Interaction between Power/Performance, Architecture, Circuits, and Compilers (P=ac2)*, Oct. 2006.

[9] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *Proc. 28th ACM International Symposium on Computer Architecture*, pp. 240–251, June 2001.

[10] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, and D. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 1–7, Aug. 2011.

[11] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE 4th Workshop on Workload Characterization*, pp. 3–14, Dec. 2001.

[12] S. P. E. Corporation, "SPEC CPU 2006." `https://www.spec.org/cpu2006/`. Online; accessed 15-January-2016.

[13] C. Bienia, S. Kumar, J. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proc. 17th ACM International Conference on Parallel Architectures and Compilation Techniques*, pp. 72–81, Oct. 2008.

[14] Aurora Softworks, "Quadrant Benchmark." `https://play.google.com/store`. [Available at Google Play; accessed 28-September-2015].

[15] "AnTuTu Benchmark." `http://www.antutu.com/en/Ranking.shtml`. [Available at Google Play; accessed 15-August-2015].

[16] Unstable Apps, "CPUBenchmark." `https://play.google.com/store`. [Available at Google Play; accessed 28-September-2015].

[17] Kishonti Informatics, "GFXBench GLBenchmark." https://gfxbench.com/result.jsp. [Available at Google Play; accessed 28-September-2015].

[18] J. Kim and J. Kim, "Androbench: Benchmarking the storage performance of Android-based mobile devices," *Springer Frontiers in Computer Education*, vol. 133, pp. 667–674, 2012.

[19] C. Lin, J. Lin, C. Dow, and C. Wen, "Benchmark Dalvik and native code for Android system," in *Proc. IEEE International Conference on Innovations in Bio-Inspired Computing and Applications*, pp. 320–323, Dec. 2011.

[20] C. Lee, E. Kim, and H. Kim, "The AM-Bench: An Android multimedia benchmark suite," Tech. Rep. GIT-CERCS-12-04, Georgia Institute of Technology, Center for Experimental Research in Computer Systems, 2012.

[21] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-system analysis and characterization of interactive smartphone applications," in *Proc. IEEE International Symposium on Workload Characterization*, pp. 81–90, Nov. 2011.

[22] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. Emmons, and N. Paver, "A structured approach to the simulation, analysis and characterization of smartphone applications," in *Proc. IEEE International Symposium on Workload Characterization*, pp. 113–122, Sept. 2013.

[23] D. Pandiyan, S.Y.Lee, and C. Wu, "Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite — MobileBench," in *Proc. IEEE International Symposium on Workload Characterization*, pp. 133–142, Oct. 2013.

[24] Y. Huang, Z. Zha, M. Chen, and L. Zhang, "Moby: A mobile benchmark suite for architectural simulators," in *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 45–54, Mar. 2014.

[25] J. Poovey, M. Levy, S. Gal-On, and T. Conte, "A benchmark characterization of the EEMBC benchmark suite," *IEEE Micro*, vol. 29, Sept. 2009.

[26] J. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 1–17, Dec. 2006.

[27] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for accurate and efficient simulation," in *Proc. ACM International Conference on Measurement and Modeling of Computer Systems*, pp. 318–319, June 2003.

[28] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "SimPoint 3.0: Faster and more flexible program analysis," in *Workshop on Modeling, Benchmarking and Simulation*, June 2005.

[29] C. Lefurgy, P. Bird, I. Chen, and T. Mudge, "Improving code density using compression techniques," in *Proc. 30th IEEE International Symposium on Microarchitecture*, pp. 194–203, Dec. 1997.

[30] S. Segars, K. Clarke, and L. Goudge, "Embedded control problems, thumb, and the arm7tdmi," pp. 22–30, Oct. 1995.

[31] M. Dubois, M. Annavaram, and P. Stenström, *Parallel computer organization and design*. Cambridge University Press, 2012.

[32] A. Smith, "Cache memories," *ACM Computer Surveys*, vol. 14, pp. 473–530, Sept. 1982.