A Study of Branch Prediction in Android

Adarsh Patil
Computer Science and Automation
M.Sc (Engg), IISc
Email: adarsh.patil@csa.iisc.ernet.in

I. Introduction

The worldwide popularity of inexpensive smart phones, tablets and low power pocket devices has caused a major shift in computing paradigms. Reliability, availability, connectivity, as well as performance related concerns has become paramount. Mobile platforms have different types of application physiognomies than those of traditional desktop applications and pose unique challenges to the architecture community.

Applications for mobile devices differ from desktop applications in significant ways. Desktop applications typically tend to run for long sessions (hours, days) and are much more feature rich. Hence, these sessions contain a great deal of iterating and the locality of reference is much more contained. The feature-richness of these applications means large working-set sizes of instructions and data. Mobile device applications on the other hand tend to be focussed for one purpose rather than being feature-rich and are used in short spurts, frequently, but for short session durations. Speeding up startup time (or wakeup time), responsiveness and getting users back to the activities they were doing when they last shut down is of paramount importance as today's modern mobile Operating Systems offer multi-tasking capabilities just as on desktop. This leads to low locality of reference, high context switches and smaller working-set sizes. Also, mobile devices and applications are mostly used for consumption of content as compared to authoring content. The 4 key requirements for mobile devices are: high performance for multimedia functions, energy/power efficiency, small size and low design complexity.

We focus on one specific issue that has long been considered an important issue for performance optimization of state-of-the-art processors - *control flow prediction*. Improving accuracy of branch prediction improves performance due to correct speculative path execution and reduces power consumed due to fewer squashing of mis-predicted speculative executions. The Operating System affects control flow predictability as it introduces additional user/OS branch aliasing in branch predictor tables. We study this negative impact of kernel branches on branch prediction. As a motivating example, Figure 1 shows distribution of Kernel and User Instructions in typical mobile applications. The label on each bar indicates the percentage of Kernel instruction executed in the duration of the application. Android application take user interactions to all new levels. The unpredictable user behaviour for interactive mobile applications can further exacerbate the branch misprediction rates.

A. *New features in Mobile Devices stress branch prediction*

Today's mobile devices are equipped with several new features that make them attractive and more capable devices.Firstly, the ability to multi-task and switch between applications without closing their state, similar to the minimize in desktop applications. Such context switches can rarely be predicted and incur heavy squashing of instructions in the pipeline due to context switch. This frequent context switch triggers execution of kernel code which leads to poor performance of branch prediction and may need invalidation of branch prediction tables. Such new capabilities aggravate an already existing problem. Secondly, recent releases of android allow applications to run in background. This introduces further complexities in branch prediction and fragmentation of the predictor tables which are indexed with Program Counters. We do not study this type of application switching behaviour and background application activity as it is difficult to simulate and existing benchmarks have not incorporated such behaviours yet. We will make the case with running single application simulation in foreground without any other applications running in the background.

B. *Related Work*

In [1] Huang et. al present a diverse application set as the Moby benchmark to study architectural properties of android applications and evaluate Mobys micro-architecture independent features (instruction mix, working set size, data and instruction locality, and binary execution behaviour) However we focus on branch prediction specifically and aim to understand the interplay for User and Kernel mode branches in Android ecosystem. In [2] Li / Sivasubramanium et. al studied the branch history interference between user/OS branches for desktop applications. They propose new OS-aware control flow prediction techniques to alleviate the destructive impact of this interference for desktop applications. As described above the characteristics of mobile applications differ from desktop applications and the hardware used in both has different characteristics and design properties. To the best of our knowledge a similar study for mobile applications has not been done previously.

II. Keywords

Branch Prediction, Mobile Applications, Android ecosystem, gem5, Moby, OS-aware Branch Prediction

III. Background

A. Android and Dalvik VM

Android is described as a mobile operating system, initially developed by Android Inc. Android was sold to Google in 2005. Android is based on a modified Linux 2.6 kernel. Google, as well as other members of the Open Handset Alliance (OHA) collaborated on Android design, development and distribution. Currently, the Android Open Source Project (AOSP) is governing the Android maintenance and development cycle

Compared to a Linux 2.6 environment though, several drivers and libraries have been either modified or newly developed to allow Android to run as efficiently and as effectively as possible on mobile devices. The focus has always been on optimizing the infrastructure based on the limited resources available on mobile devices. To complement the operating environment, an Android specific application framework was designed and implemented.

Android based systems utilize their own virtual machine known as the Dalvik Virtual Machine (DVM). The DVM uses special byte-code, hence native Java byte-code cannot directly be executed on Android systems. The DVM implementation is highly optimized in order to perform as efficiently and as effectively as possible on mobile devices that are normally equipped with a rather modest CPU subsystem, limited memory resources, no OS swap space, and limited battery capacity. The DVM has been implemented in a way that allows a device to execute multiple VMs in a rather efficient manner. It also has to be pointed out that the DVM relies on the modified Linux kernel for any potential threading and low-level memory management functionalities.

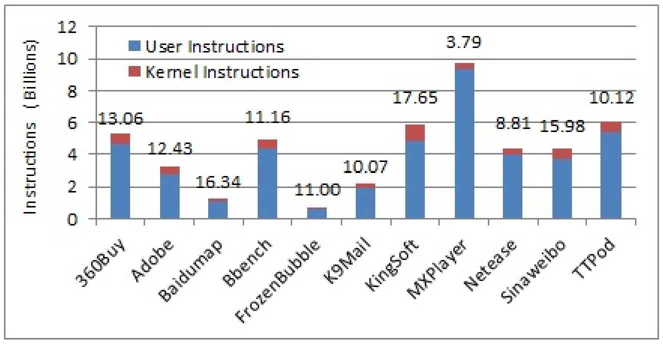


Fig. 1: User and Kernel Instructions

B. Target Platform and ISA

ARM represents a 32-bit reduced instruction set computer (RISC) instruction set architecture, x86 systems are primarily based on the complicated instruction set computer (CISC) architecture. In effect we can say that ARM (RISC) is executing simpler (but more) instructions compared to an x86 (CISC) system. In mobile devices memory is at a premium due to size, power and cost constraints. ARM addresses these issues by providing a 2nd 16-bit instruction set (labelled thumb) that can be interleaved with regular 32-bit ARM instructions. This additional instruction set can reduce the code size by up to 30% with some performance limitations. Hence this thumb instruction set is a compromise between performance and power.

Today, there are basically 4 major chip sets being deployed in Android devices. On the one side there is ARM Cortex-A8 architecture based processors with Vendor tweaks like Qualcomm Snapdragon, Texas Instruments OMAP, Samsung designed Hummingbird chipset and on the other side the Intel x86 based low power Atom processors. However, the competition and trend in the market has seen number of ARM devices explode compared to traditional x86 based devices. This is because compared to x86 processors, the ARM design reveals a strong focus on lower power consumption, which again makes it suitable for mobile devices. Hence, throughout the rest of this work we primarily focus on ARM based processors and study the components in a typical ARM (RISC) processor.

C. Branch Predictors

The processor contains program flow prediction hardware, also known as branch prediction. The processor simulated implements detailed ARMv7-A architecture profile [10]. It contains program flow prediction hardware, also known as branch prediction. With program flow prediction disabled, all taken branches incur a 13-cycle penalty. With program flow prediction enabled, all mispredicted branches incur a 13-cycle penalty. To avoid this penalty, the branch prediction hardware operates at the front of the instruction pipeline. An unpredicted branch executes in the same way as a branch that is predicted as not taken. Incorrect or invalid prediction of the branch prediction or target address causes the pipeline to flush, invalidating all of the following instructions. Within this definition there are certain instructions that will be predicted and some which will not be predicted. The following are some details of the branch prediction methodology used in evaluation later.

1) Predicted instructions: The following are the instructions that the Branch Predictor predicts:
B, BL, BLX, BX, LDR with PC destination, LDM with PC in the register list, PC-destination data-processing operations i.e.ADD, MOV, CPY with being PC-destination register

2) Nonpredicted Instructions: The following are the instructions that are not predicted:

- Instructions that can be used to return from an exception
- Instructions that restore the CPSR from memory or from the SPSR
- PC-destination data-processing instructions with immediate values
- BXJ

3) Tournament Branch Prediction: To minimize the branch penalties, the branch predictor in the processor uses a tournament branch prediction algorithm as proposed by Scott McFarling [11]. The algorithm maintains two history tables, Local and Global, and the table used to predict the outcome of a branch is determined by a Choice predictor. The local predictor is a two-level table which records the history of individual branches. It consists of a 2048 entries. Each entry is a 2-bit saturating counter. The value of the counter determines whether the current branch is taken or not taken. The global predictor is a single-level, 8192-entry branch history table. Again, each entry here a 2-bit saturating counter; the value of this counter determines whether the current branch is taken or not taken. The choice predictor records the history of the local and global predictors to determine which predictor is the best for a particular branch. It has a 8192-entry branch history table. Each entry is a 2-bit saturating counter. The value of the counter determines if the local or global predictor is used.

The processor also contains a Branch Table Buffer (BTB) of 2048 entries tagged by 18 bits. The BTB acts as a buffer for the choice predictor and aids quick lookup for the tournament prediction. The complete specifications of this is summarized in
Table I below.

IV. Evaluation Methodology

We use the Moby Mobile Benchmark suite [1] to study architectural characteristics of Android applications. We run this benchmark on gem5 simulator in the Full System Cycle Accurate mode for the ARM architecture. The rest of the section describes the details of the benchmark and modifications made to the gem5 simulator to collect further statistics. Table 1 gives the architectural simulation parameters under which this was studied. These specifications were used to match a typical ARM community supported development platform like the PandaBoard [7]. The study of optimum trade-off numbers for these parameters (cache size, associativity, block size, i/d cache split) used here for simulation of the android ecosystem is beyond the scope of this work. Such work has been carried out in the HPC lab earlier [3] and we derive these numbers from the conclusion of the study.

TABLE I: Simulation Parameters Used in Experiments

Parameter	Value
CPU	O3 CPU [9]
L1 i-cache	32KB / 4 way set assoc / 64 Byte line size
L2 d-cache	32KB / 4 way set assoc / 64 Byte line size
L2 cache	512KB / 16 way set assoc / 64 Byte line size
iTLB / dTLB	128 entries each
Main Memory	256MB LP DDR2
Branch Predictor Type	Tournament
BTB Entries	2048
Global Predictor Size	8192 (2 bit)
Local Predictor Size	2048 (2 bit)
Choice Predictor Size	8192
Choice Counter Bits	2
RAS Size	16

A. Simulator

We use the gem5 simulator and the hardware performance counters provided by ARM processors to evaluate Mobys micro-architectural features. However we found the counters provided were insufficient to perform detailed study as needed. The gem5 simulator was instrumented with several additional counters at various points. This required a rewrite of the Stats class of gem5 and additional helper classes were included. To calculate the branch-prediction effectiveness the following additional counters were added.

- number of continuous instructions executed in user / kernel mode
- number of context switches
- number of branch instructions in user / kernel mode
- number of branches per burst in user / kernel mode
- number of incorrect predictions in user /kernel mode
- number of instructions squashed from pipeline per misprediction in user / kernel mode

The modified gem5 simulator was then complied and built. This instrumented version of gem5 was used throughout for simulations.

B. Benchmarks

The moby benchmark is composed of 10 mobile applications of diverse classes such as web browser, documents, online shopping, email, audio, video, maps, games etc. Moby has been ported onto the gem5 simulator. The benchmark consists of popular application from the Google Play Store except the BBench browser application. We defer the reader to the related Moby literature [1] to note the applications and thier criteria of selection. Another notable reason for choosing this benchmark is that it eliminates the non-deterministic replay and overhead of replaying user interaction through execution of *activities* by specifying their inputs (user-action) manually through command-line arguments.

Moby benchmark suite is provided as a Ice Cream Sandwich (2.6.35) kernel image [8], the raw disk image and a SD card image. The SD Card was auto-mounted into the target environment in gem5 using appropriate scripts as specified by the benchmark. Further, to startup each application benchmark on bootup of the target environment each application .rc script (init script) was passed to the simulator. Given that the benchmark takes several days to complete full system simulation several jobs were queued back to back and the targed system was reset to flush cache, pipelines and counters.

V. Experimental Results

A. Instruction Flow

The flow of instructions executed by most mobile applications exhibit complex behaviour. Mobile applications are heavy Graphics based applications and have high number of user interactions which leads to combinatorially large number of execution paths possible for the instruction flow. Adding to this, android applications are written in high level language (Java) for portability which is then translated by Dalvik VM to produce optimized byte code adding to the complexity. The applications invoke several libraries and binaries during flow of execution. As Huang et. al [1] point out, the phases of execution in Moby applications can be characterized into 3 groups: Java-language related, C-language related, and system related. We categorize the first two of these as *user mode* and the system related phases as *kernel mode* execution. There is different degrees interleaving of execution and burst lengths of these 3 phases in Moby applications. Due to this variable instruction locality, branch prediction accuracy are largely affected. Figure (2) shows the lengths of kernel mode bursts (in millions of instructions) for 3 different applications. The first 100 bursts are perfectly co-related and this corresponds to the boot time of the device.

[1]Percent of all branch instructions that are conditional

TABLE II: Execution Statistics

Application	**Total Instructions (M)**	**Kernel Instructions (%)**	**Context Switches (M)**	**% of Branches**	**% of conditional branches** [1]	**Conditional Branch Statistics**				
						% overall incorrect	User Mode		Kernel Mode	
							Number of Branches (M / %)	% incorrect	Number of Branches (M / %)	% Incorrect
360Buy	5331	13.06	0.62	22.95	68.89	11.16	728 (86.34 %)	3.87	115 (13.66 %)	57.20
Adobe	3198	12.43	0.58	21.73	69.70	10.03	445 (91.89 %)	6.63	39 (8.11 %)	48.46
Baidumap	1231	16.43	0.58	25.70	67.63	11.34	192 (89.84 %)	6.83	22 (10.16 %)	51.22
BBench	4956	11.16	0.72	23.36	66.71	12.50	686 (88.87 %)	8.42	86 (11.13 %)	45.12
FrozenBubble	660	11.00	0.76	23.67	69.28	8.82	100 (92.53 %)	4.39	8 (7.47 %)	63.67
K9Mail	2179	10.07	0.78	19.72	68.83	11.32	258 (87.28 %)	1.53	38 (12.72 %)	78.46
KingSoft	5854	17.65	0.67	26.03	66.99	12.58	853 (83.53 %)	4.48	168 (16.47 %)	53.65
MXPlayer	9715	3.79	0.72	8.49	74.31	7.14	567 (92.76 %)	2.21	44 (7.24 %)	70.22
Netease	4348	8.81	0.54	21.72	69.02	10.82	610 (93.62 %)	6.89	42 (6.38 %)	68.55
Sinaweibo	4366	15.98	0.64	25.84	66.40	12.60	653 (87.12 %)	6.50	96 (12.88 %)	53.85
TTPod	6059	10.12	0.75	23.70	66.95	12.36	850 (88.43 %)	4.85	111 (11.57 %)	69.72

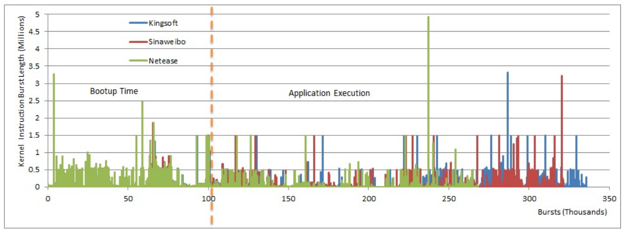
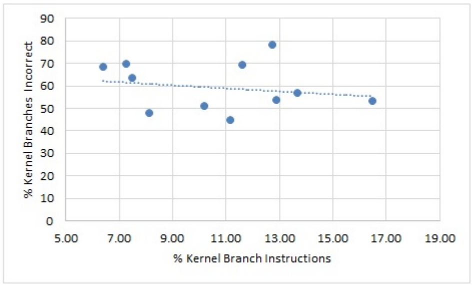
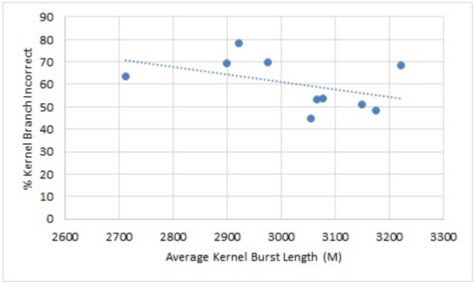


Fig. 2: Burst Pattern of Kernel Instructions



(a) Incorrect Prediction vs % of Branch Instructions



(b) Incorrect Prediction vs Burst Length

Fig. 4: Trend of Kernel Incorrect Prediction

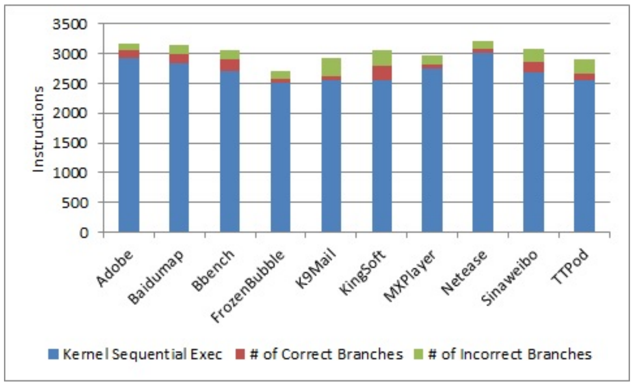


Fig. 3: Burst Distribution of Kernel Instructions

Different types of branch instructions indirectly reveal the complexity of programs and their demands on branch predictors. As shown in Table II, branches account for about 22% of total instructions on an average where about 70% of these branches are conditional branch. Conditional branches may result in executing an erroneous path and consequently require out-of-order processors to roll back and squash several instructions. The high percentages of conditional branch instructions is likely to trigger many mispredictions with large penalties, which will affect the overall performance. To be able to better utilize out-of-order processors and exploit instruction level parallelism branch prediction accuracy plays a pivotal role.

B. Incorrect Branch Predictions

We observe an average of 60% incorrect branch predictions in Kernel Mode as compared to 5.5% incorrect branch predictions in User Mode. This huge skew can be attributed to very low Kernel instruction execution numbers (on average 11.86%). The branch prediction mechanism is unable to observe and maintain correlation in the kernel mode bursts which are few and far spaced in the instruction execution flow. However this high incorrect branch prediction leads to instruction squashes as speculative instructions are executed along the mispredicted path. We also observe the number to be on an average 57 instructions per kernel burst (including the bootup time). This number is significantly high and represents wasted executions along this path. Figure 3 shows the average kernel mode burst distribution between sequential execution, instructions executed on correct and incorrect branch speculations.

Based on the statistics from Moby applications, the plot in Figure 4(a) shows the percentage of Kernel Mode Incorrect Branch Prediction versus the percentage of kernel branches executed. Similarly Figure 4(b) shows the percentage of Branch Prediction versus the average kernel mode burst length. This further supports our claim that as number of kernel branch instruction increase mispredictions decrease. Similarly, as number of kernel bursts size increase the mispredictions decrease. However the decrease is only linear. We expect that this linear trend would be insufficient to cope with additional context switches that will be added due to the new features in upcoming android releases (refer Section I) going forward.

VI. Conclusion

In this work, we have studied the instruction flow in Android ecosystem and the time spent in execution in the User and Kernel mode. We studied the distribution of branches in User and Kernel mode. We also quantified the wasted work in mispredicted branches. Power consumption being a first rate design parameter in mobile devices, this wasted effort can be avoided with better branch prediction techniques. The trend of mispredictions with respect to kernel mode burst duration and time spent in kernel mode execution was also studied. This study can prove invaluable to hardware designers trying to improve performance of Android on ARM devices. Although tournament branch prediction maintain 2 context using the global and local tables it is not effective for mobile applications where the kernel executions are few and far spaced in the instruction execution flow. Thus, the proposed techniques by Li, Sivasubramanium et. al about using OS aware branch prediction could be used to reduce mispredictions rates in Mobile devices.

In the process we authored several new hardware statistics for the gem5 simulator which can be used in a general simulation exercise to understand branch prediction statistics.

Acknowledgment

The author would like to thank Prof. R Govindarajan and the HPC lab for the resources to run the simulations and the timely thoughtful ideas.

References

[1] Huang, Yongbing, Zhongbin Zha, Mingyu Chen, and Lixin Zhang. ”Moby: A Mobile Benchmark Suite for Architectural Simulators.”

[2] Li, Tao, Lizy Kurian John, Anand Sivasubramaniam, Narayanan Vijaykrishnan, and Juan Rubio. ”Understanding and improving operating system effects in control flow prediction.” In ACM Sigplan Notices, vol. 37, no. 10, pp. 68-80. ACM, 2002.

[3] Vishaal Mohan under guidance of Prof. R. Govindarajan . ”Study of Mobile Benchmark characteristics on a full-system simulator”, July 2014.

[4] Bornstein, D., Dalvik VM Internals, Google I/O Developer Conference, 2008

[5] Brady, P., Android Anatomy and Physiology”, Google I/O Developer Conference, 2008

[6] http://gem5.org/

[7] Pandaboard System Reference Manual

[8] http://developer.android.com/about/versions/android-4.0-highlights.html

[9] http://www.m5sim.org/O3CPU

[10] ARM Technical Reference Manual (Coretex-A8) http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf

[11] McFarling, Scott. Combining branch predictors. Vol. 49. Technical Report TN-36, Digital Western Research Laboratory, 1993.

[12] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.