Variability Emulation in Linux/Android Devices

Pietro Mercati1, Francesco Paterna1, Andrea Bartolini2,3, Luca Benini2,3 and Tajana Rosing1.
1University of California, San Diego, Computer Science Department, San Diego, USA
2ETH Zurich, Integrated Systems Laboratories, Zurich, Switzerland.
3University of Bologna, DEI, Bologna, Italy

Abstract—Variability is a major challenge for future CMOS technologies. With scaling, design techniques are less effective and it is required to manage variability at runtime. Unfortunately, at this time there is no tool available to investigate the effect of variability on real devices and to test the robustness of current software stacks. In this work, we present VarDroid, a tool for online power and performance variability emulation on real Android devices, implemented in the Linux kernel. VarDroid enables variability investigation on mobiles while capturing real workload dynamics. We show on Android mobile platform that a variability-agnostic OS can have up to 60% performance degradation in presence of variability.

Keywords—Variability; Android; Emulation;

I. INTRODUCTION

Variability in integrated circuits is a major concern today. CMOS scaling exacerbates static and dynamic variations [1]. State-of-the-art publications show that variability in manufacturing results in transistor-level parameters whose power consumption and operating frequency (i.e. performance) [2] follow Gaussian distribution with a wide spread in values.

Counteracting this issue by adopting larger design margins is extremely costly. For this reason, Underdesigned and Opportunistic computing (UnO) is becoming popular [5]. The key idea of UnO is to expose variability at higher layers of the software stack to enable runtime adaptation. Such dynamic adaptation is called Dynamic Variability Management (DVM).

Implementing DVM requires that we investigate the impact of variability on platforms. This can be done using simulators that extract variability from design parameters and propagate it to the processor level to estimate power and performance [3,4]. However, such simulators cannot capture dynamic variations of a real workload. This can be accomplished by variability emulation, which injects variability in real devices in the form of OS or register-level faults. Unfortunately, the state-of-the-art in variability emulation does not consider mobile devices.

Mobiles are characterized by tight user interaction and are required to work on a variety of environments. Therefore, the investigation of variability and DVM is particularly urgent. The goal of mobiles is to provide good user experience rather than high performance, which requires that the impact of variability be investigated on real devices subject to real interactive workloads. At this time, a tool to evaluate the robustness of real mobile devices with respect to variability is missing.

Android is one of the top mobile operating systems. It is built on top of the Linux kernel and is open source. Android allows us to implement flexible monitoring and management infrastructure, making it perfect for our purpose.

In this work, we present VarDroid, a low-overhead framework to emulate power and performance variability on Android devices. VarDroid can be configured to inject variations at the OS level. We demonstrate how our tool can be used to (i) analyze the robustness of existing applications against variability, (ii) evaluate the impact of variability on user experience and (iii) highlight the limitations of current OS design on heterogeneous architectures (e.g. ARM big.LITTLE). We show that a variability-agnostic OS can have up to 60% performance degradation in presence of variability.

II. RELATED WORK

Recent developments present tools for architectural level variability simulation, such as VARIUS [3] and VAM [4]. Such tools can be integrated with cycle-accurate simulators such as GEMS, but they require detailed architectural description of the target platform. Moreover, simulations are slow, as a few seconds of real time may take hours of simulation.

An alternative approach is variability emulation, which consists in injecting faults in real devices. Previous work exploits architecturally-visible registers [8] and combined hardware-software approaches for the evaluation of resiliency solutions [9]. VESPA [6] is a variability emulation framework for SoC performance analysis that translates component-level characteristics into system-level performance distribution. Work in [7] proposes VarEmu, an extension of the QEMU operating system that extracts timing and cycle count from emulated code. The information is fed into a variability model with configurable parameters that determines energy consumption and fault variations in the virtual machine. VarEmu relies on models for power estimation, which have to be adapted to different platforms. None of the previous work on variability emulation addresses mobiles and the crucial need of capturing interactive workload on variability-affected devices. Recent work proposes online reliability emulation in Android [10,11]. The proposed strategy is orthogonal to [10,11] in that we focus on performance and power variability, rather than emulating degradation.

In this work, we develop and implement a framework for emulating performance and power variability on real Android devices. Thanks to this, our solution naturally captures real mobile workloads.

III. VARDROID ARCHITECTURE

A. Variability Models

Variability in integrated circuits manufacturing has two main components: D2D (die-to-die) and WID (within-die). WID variations, then can be further separated into systematic and random variations. At the microarchitectural level, the
elements of key importance for variability are the transistor threshold voltage $V_{th}$ and the effective channel length $L_{eff}$ of transistors. An alternative granularity at which process variations may be considered is the one of C2C (core-to-core) variations [2]. Following this model, low level-variations in $V_{th}$ and $L_{eff}$ translate in operating frequencies and power consumption which are distributed among different cores. Since $V_{th}$ and $L_{eff}$ can be considered normally distributed with good approximation, also power consumption and operating frequency among different cores can be considered normally distributed.

B. VarDroid Framework

Figure 1a shows the architecture of VarDroid. It has two main components: the VarDroid Engine, which can select between three configuration inputs and the VarDroid Monitor. VarDroid’s inputs are: the percentage of frequency degradation over $f_{MAX}$ for PERT, the nanosleep duration for SCHED and the bubble length for INJ. They inject variability at different layers of the software stack, and consequently at different time granularities. This is shown in Figure 1b. PERT sets an upper bound on operating frequency. SCHED has the effect of occupying some scheduling intervals with spurious applications. Finally, INJ runs during the scheduling interval with spurious instructions. More details are below.

Operating conditions perturbation (PERT): PERT limits the operating frequencies of cores to emulate performance variability. We extract $f_{MAX}$ from a Gaussian distribution and assign randomly one value to each core. PERT is implemented at the userspace level, and sets an upper bound on frequency by exploiting the sysfs interface exposed by the cpufreq driver.

Extra scheduling (SCHED): SCHED interleaves the execution the normal execution of applications with the scheduling of extra tasks. It is essentially a program that alternates periods of execution and sleep (implemented with nanosleep). SCHED can choose between an “Idle” application (a long sequence of NOPs) and a “Power” application (a sequence of cpu-intensive operations). The first emulates variability in performance, the second in power. Both PERT and SCHED are C programs, which are cross-compiled with Android NDK tools to run on the target ARM platform.

OS Fault Injection (INJ): INJ is implemented with bubbles in the OS kernel. A single bubble in our design is a sequence of 20 NOPs which are inlined in the scheduler_tick() function of the scheduler source code (core.c). The impact of variability is tuned by selecting the number of bubbles (referred to as “bubble length”).

The VarDroid Monitor has been implemented by modifying a reliability manager described in [11]. It has a kernel sampling function, a monitor driver to transfer data to the user space and a monitor daemon to regulate the data transfer. The monitor infrastructure is able to get one sample per each scheduling tick (10ms in our setup) and transfers data to the userspace at low overhead.

IV. EXPERIMENTAL EVALUATION

VarDroid has been implemented on the Odroid XU3 board. It has a Samsung Exynos 5422 Octa core with ARM big.LITTLE architecture, with a Cortex™-A15 2.0Ghz quad core cluster and Cortex™-A7 quad core cluster [12]. The device has a 2MB L2 cache and a 2GB RAM. The OS is Android 4.4.4 with Linux kernel 3.10.9.

A. VarDroid Validation

The key idea for validating VarDroid is to show that when varying the VarDroid inputs we obtain a Gaussian distribution of frequency and power consumption across cores that follows the low level models for transistor parameters such as channel length $L_{eff}$ and voltage threshold $V_{th}$.

We conduct a series of experiments with single-threaded microbenchmarks. The cpu-bound executes a series of ALU operations without accessing memory, the L2-bound executes a series of loads from the L2 memory and the mem-bound similarly executes a series of loads from the main memory. To avoid the effect of migrations, microbenchmarks exploit the set affinity mechanism to run on a specific core.

Performance Variability: In the following experiments, we keep only one LITTLE core active, running at fixed maximum frequency. Analogous results can be obtained for big cores. Figure 2 shows the normalized execution time of the three microbenchmarks for the three VarDroid configurations. The execution time of the benchmarks increases as the VarDroid input increases. This means that VarDroid effectively makes the processor behave as if it had a maximum frequency lower than the nominal.
To further validate this behavior, we run the *cpu-bound* benchmark with random VarDroid input extracted from a normal distribution, in the three configurations. In Figure 3, the plots on the left show the histogram of VarDroid input. Then we collect the execution times and build a histogram, shown in the plots on the right. In the case of SCHED, we employed a smaller number of samples and a version of the *cpu-bound* benchmark of longer duration. In this way, VarDroid emulation shows results as if the benchmarks were executed by different instances of the same processor, affected by performance variability. No variation in execution time is present when VarDroid is disabled.

![Fig. 2. Microbenchmarks execution times](image)

**Power Variability**: Next, we verified the effect of VarDroid on power variability. In Figure 4 we show the histogram of random input for the SCHED Power configuration (e.g. *nanosleep* duration) and the corresponding histogram of average power consumption. In this experiment, we are measuring only the power consumption of the LITTLE cluster, but a similar result can be obtained for the big cluster. One sample of power consumption is computed by averaging the power over a period of 120 seconds, with a given SCHED power input. Indeed, it is well recognized that FPS in 3D gaming is a good metric for quality of experience. In particular, the closer to 60fps (which is the maximum allowed by Android), the better. The results in Figure 6 show the different FPS traces. As expected, the FPS lowers as the INJ input increases. In particular, in a scenario of severe variability impact (150k), user experience is heavily affected, as FPS drops even below 50.

![Fig. 3. VarDroid performance distribution](image)

**B. Use Cases**

In the following experiments, we present use cases in which we can exploit VarDroid.

**Application Robustness**: VarDroid can be used to investigate the impact of variability on real application performance and power consumption. In Figure 5, we present the scores obtained with the mobile benchmark Antutu v5.6.1 with increasing bubble length of INJ (from 0 to 150000). Scores of mobile benchmarks are a common metric to compare the quality of execution of different devices. As expected, all the scores decrease as the INJ input increases. Therefore, VarDroid can be used by application developers to test the performance of their apps in a real device affected by variability.

![Fig. 4. SCHED Power variability of LITTLE cluster](image)

**User Experience**: Since VarDroid is implemented on a real device, it can capture real workload dynamics and user interaction. Therefore, it can be used to assess the impact of variability on user experience. For this experiment, we record and replay a 10 seconds-long touch event trace of a popular Android 3D game (Temple Run), to reproduce the same workload. Then, we execute the trace with different INJ bubble lengths, and measure Frame per Seconds (FPS) by monitoring the SurfaceLinger service. Indeed, it is well recognized that FPS in 3D gaming is a good metric for quality of experience. In particular, the closer to 60fps (which is the maximum allowed by Android), the better. The results in Figure 6 show the different FPS traces. As expected, the FPS lowers as the INJ input increases. In particular, in a scenario of severe variability impact (150k), user experience is heavily affected, as FPS drops even below 50.
Variability Tolerance of the OS: VarDroid can be used to detect limitations and “bugs” of existing OS design. In this experiment, we focus on the workload migration policy employed in ARM big.LITTLE architectures [12]. In such architecture, a hysteresis mechanism regulates the task migration between LITTLE and big cores, with a high and a low utilization threshold. When the high threshold is exceeded, the task is migrated to the big core, and is returned to the LITTLE core if utilization decreases below the low threshold. The limitation of such mechanisms is that migration thresholds are fixed. Because of this, the execution can incur significant performance penalties.

![Fig. 7. Migration (a) performance and (b) power penalty](image)

To show this, we implemented a single-threaded cpu-bound synthetic workload that periodically alternates between three execution phases, respectively with low, medium and high utilization. Then, we execute it with only one LITTLE core and one big core active. The workload is initially allocated on the LITTLE core and execute low and medium phases. The high phase instead triggers the migration to the big core. We execute the program with increasing INJ input, and measure execution times. Figure 7a shows the normalized execution times of the medium phase. Results show a penalty of up to 60%. A lower migration threshold in this case would have avoided such penalty. The variability tolerance of the migration policy in this case could be improved by changing the migration threshold dynamically. Indeed, in a scenario in which platform can actually detect the variability at runtime thanks to dedicated sensors, thresholds should be adapted dynamically.

As a final example, we make the case in which the LITTLE core can choose to migrate the workload between two big cores, a core X of the big cluster with nominal power consumption (which is 1.5W from our measurements) and a core Y with higher power consumption due to variability. Results in Figure 7b show that a variation-agnostic migration policy can incur in a power penalty of up to 20%.

V. CONCLUSION

This paper presents VarDroid, an emulation framework for power and performance variability, implemented in the Android operating system. With VarDroid it is possible for the first time to investigate variability while capturing real workload dynamics. We validate its behavior with and show use cases to illustrate possible applications.

VI. ACKNOWLEDGEMENTS

This work was supported by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, through TerraSwarm center.

REFERENCES


