A Linux-Governor Based Dynamic Reliability Manager for Android Mobile Devices

Pietro Mercati
UCSD
pimercat@eng.ucsd.edu

Andrea Bartolini
ETH Zurich
University of Bologna
a.bartolini@unibo.it

Francesco Paterna
UCSD
fpaterna@ucsd.edu

Tajana Simunic Rosing
UCSD
tajana@ucsd.edu

Luca Benini
ETH Zurich
University of Bologna
lbenini@iis.ee.ethz.ch

Abstract—**Reliability is a major concern in multiprocessors. Dynamic Reliability Management (DRM) aims at trading off processor performance with lifetime. The state-of-the-art publications study only the theory supported by simulation. This paper presents the first complete software implementation, working on a real hardware, of a low-overhead, Android-compatible workload-aware DRM Governor for mobile multiprocessors. We discuss the design challenges and the run-time overhead involved. We show the effectiveness of our governor in guaranteeing the predefined target lifetime and show that it achieves up to 100% of lifetime improvement with respect to traditional governors, while providing comparable performance for critical applications.**

I. Introduction

Reliability issues worsen with technology scaling due to the impact of degradation phenomena such as Time Dependent Dielectric Breakdown (TDDB) and Bias Temperature Instability (BTI). Degradation depends on voltage and temperature stress, environmental conditions and workload variations. With scaling, device lifetime becomes more difficult to predict [5], impacting on warranty costs, trust and reputation. Design techniques are not effective to counteract this problem, due to high variation in workloads run and changing environmental conditions.

Smartphones and tablets work in a variety of environments and run workloads with different performance requirements [7]. Therefore, they are subject to variable voltage/frequency stress [16]. Since reliability depends on temperature and voltage, a runtime control is needed to correctly manage the reliability of a device over time [8].

Reliability can be determined by monitoring voltage and temperature. Recent work also presents sensors for monitoring degradation [13], and embeds them in prototypes [14], [17].

Dynamic Reliability Management (DRM) is a set of techniques trading off processor degradation and performance at runtime [10], [15], [18]. Reliability is periodically assessed, and processor operating conditions are controlled to limit the degradation source (i.e. temperature and voltage). The goal of DRM is to not exceed a predefined target reliability within a predefined target lifetime.

Modern operating systems have dedicated software components for power management. Linux uses *Governors* to control operating conditions. Governors are kernel modules interfaced with hardware regulators [11]. Android, which is based on the Linux kernel, can select between different Governors, targeting goals such as providing maximum performance or saving energy. However, there is no governor for reliability management yet.

A. Related Work

DRM is introduced by Snirivasan et Al. in [15] as a technique to respond to changing reliability by limiting chip operating conditions through Dynamic Voltage Frequency Scaling (DVFS). In [18], an accurate voltage and temperature dependent model for TDDB degradation is part of a control loop that uses DVFS to guarantee the target reliability. These papers present simulation results assuming platform and workload models which may not be available for general purpose architectures. This limits the implementation feasibility. These publications target single core scenarios.

These publications effectively counteract degradation, but they do not consider the existence of different workload requirements to obtain good user experience. This is achieved by the technique in [10], which is a workload-aware DRM technique for multiprocessors based on a two-level controller. This technique monitors system reliability on a long time scale and adapts operating conditions to workload quality requirements on a short time scale. This is shown to outperform the state-of-the-art, as it provides full performance to critical applications. It focuses on TDDB, exploiting the model presented in [18] for simulating the presence of degradation sensors. It does not assume a priori knowledge of workload, but only leverages the runtime characterization of workload requirements. Thus it is feasible to implement.

In this work we present the design and the implementation of the *Reliability Governor* on a real Android device, leveraging multiple levels of its software stack. It exploits the Linux governor-based architecture. Therefore it is fully compatible with the Linux power management and could be adapted to other Linux-based systems. We demonstrate that our implementation has low overhead. Our results show the effectiveness of our governor in guaranteeing the predefined target lifetime and it achieves up to 100% of lifetime improvement with respect to traditional governors, while providing high performance for critical applications.

This is the first time a DRM technique has been engineered and integrated into the Linux environment, and demonstrated on a real Android device, the Qualcomm Snapdragon S4 apq8064-based mobile development tablet.

II. Reliability Governor

We developed our DRM framework on the Snapdragon S4 apq8064-based mobile development tablet, which has an asynchronous quad Krait CPU with frequency from 380MHz up 1.67GHz and voltage from 0.95V to 1.25V. Cores have independent voltage/frequency (V/f) settings and fixed operating V/f points. Therefore, changing frequency automatically changes voltage. The operating system is Android 4.1.2 (Jelly Bean), with Linux kernel 3.4.0. The rate of the scheduling tick is 10ms. This corresponds to one *jiffy*, the kernel time unit. In Android, software power management is implemented in the kernel by modules called *Governors*. Governors change operating conditions with different aims, such as energy saving or maximum performance.

Figure 1 shows our *Reliability Governor* for workload-aware DRM and the *Debug and Monitor Infrastructure* that exports data to the user space. The diagram refers to the governor for a single core. For a multiprocessor with per core voltage and frequency, this scheme is replicated for each core. The framework has a *Long Term Controller* (LTC) which activates every *Long Interval* (LI = days it takes for reliability to change) and a *Short Term Controller* (STC) which activates every *Short Interval* (SI = 1 jiffy). Tasks are divided in *Highly critical* (H) and *Less critical* (L) as in [10]. For providing good user experience, H tasks must be executed at maximum frequency. The goal of DRM is to let the core reliability R be above a target reliability R_t at the predefined target lifetime.

A. Governor Architecture

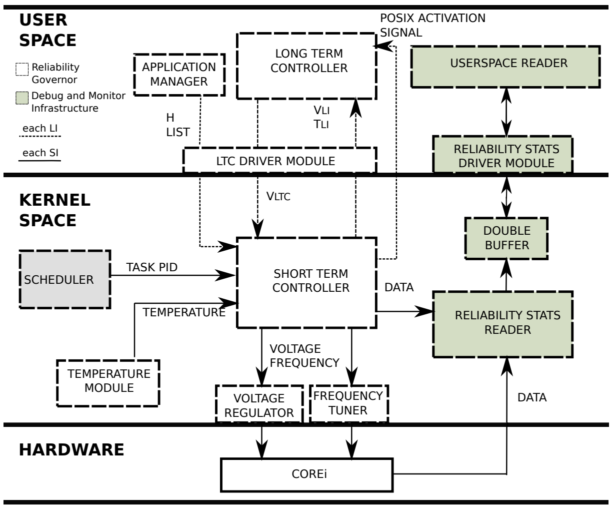

Fig. 1. Block diagram of the implemented governor.

The *Reliability Governor* in Figure 1 (white blocks) operates at two time scales and requires floating point support. It differs from a standard power management governor, which uses only one time scale, compatible with the scheduler rate, and uses simple arithmetic operations. Therefore, our modular implementation leverages both kernel modules and user space daemons. In the kernel space, the *Short Term Controller* at the beginning of each *Long Interval* gets a reference voltage V_{LTC} from the LTC. Then, it assigns new voltage and frequency at each SI based on the *Borrowing Strategy* [10]. The *Long Term Controller*, in the user space, at each LI reads the average voltage and temperature V_{LI} and T_{LI}. Based on that, it calculates the current reliability R with the model presented in [18] for TDDB[1] and computes V_{LTC}, as in [10]. The constraint to reliability is met if at each LI $V_{LI} \leq V_{LTC}$. The *LTC Driver Module* allows LTC and STC to communicate, as they are in different spaces of the software stack. The *Application Manager* communicates to the kernel space the list of H applications (*H list*), so that at each SI the governor knows whether the running task is H or not. The STC acts on the *Voltage Regulator* and the *Frequency Tuner* to change the core operating conditions.

The gray blocks of Figure 1 compose the debug infrastructure that exports data to the user space, similarly to [2]. This is used to validate the proposed solution and to collect experimental results. The *Reliability Stats Reader* reads data both from the STC and from core registers. Data are stored in a *Double Buffer* of memory in the kernel space. The reading is controlled by the *Userspace Reader*, which, once a buffer is full, it flushes it and enables writing on the other one. The *Reliability Stats Driver Module* allows the userspace reader to communicate with the kernel memory space.

B. Design Choices and Implementation

The *Short Term Controller* must be able to gather task requirements (the *H/L flag*) and data from sensors, finally to change V/f before the task is executed. In a time sharing OS, this can be done only in the scheduler, which is triggered at each jiffy. This is the only place in which the beginning and the end of the task execution can be identified. However, in Linux and Android, the scheduler is a critical module, as it contains atomic sections. Thus, calls to external interruptible functions, such those to read sensors and change frequency, may cause kernel panics. Changing the scheduler also affects software portability. Therefore, the STC has been implemented using the *ondemand* governor as starting point [11]. This has a built-in timer and the interfaces towards V/f regulators. We maintained these features, and replaced the ondemand algorithm with the STC one. The sampling rate of the governor can be set at the same rate of the scheduler, which is 1 jiffy (10ms).

A fundamental problem is how to recognize whether tasks are H or L. In the scheduler it is easy to add a field to the task data structure with the *H/L flag*. Changing the task structure, however, is invasive, and it affects the portability of the framework. Our approach does not require to change the task structure. From the user space, the *Application Manager* passes the list of PIDs (process IDs) of applications labeled as H (the *H list*). At each SI, the STC reads the PID of the running task and checks whether it is or not in the *H list*[2]. The list can be changed dynamically.

To reduce reading overhead, temperature sensors are read each second by an independent *Temperature Module*, which writes values to variables shared between the temperature module and the STC. Temperature, in fact, does not need to be observed every 10ms, as it changes at a slower rate. Moreover, the function that reads temperature sensors, *tsens_get_temp*, blocks execution. If called by the STC it can cause malfunctioning and kernel panics. Once the STC has the flag associated

[1]Other degradation phenomena can be included with appropriate models.

[2]This could be further improved with a suitable interface allowing the user to choose H applications.

to the running task, it chooses the frequency to be set, and applies it to the core.

The STC also has an internal counter to keep track of the jiffies elapsed from the beginning of the LI (t_{LI}). All the internal module computations are optimized for integer-based fixed point arithmetic. Note that this modular structure allows for changes and improvements in case the hardware has additional features, such as degradation sensors.

The *Long Term Controller*, once the LI is over, updates R and calculates the new V_{LTC}. This requires floating point operations which cannot be performed in the kernel space. Moreover, the LTC activates at a much slower rate (order of days) w.r.t. the typical kernel times (order of jiffies). Therefore we decided to "expand" the basic Governor structure by implementing the LTC as a user space daemon. In this way, the LTC can sleep until the STC recognizes that the LI is over and wakes it up. To implement this the LTC waits on a particular POSIX Activation Signal (SIGUSR1 in our implementation). In Linux kernel, signals can be sent directly form the kernel to the user space, but not vice versa. When the LI is over, the STC sends the signal to the LTC, which computes the new value of V_{LTC} and goes back to sleep. Then, it notifies to the STC that a new LI has begun by resetting its t_{LI}. In the time elapsed between sending the signal and resetting the t_{LI} shared variable, the STC operates as the LI is not over. As shown in the results, this does not lead to significant errors. To allow the exchange of data between the LTC and application manager in the user space, and the STC in the kernel space, we have developed a device driver called *LTC Module Driver*. Note that this structure allows for changes both to the reliability model and to the way to calculate V_{LTC}.

Due to the complexity of the depicted implementation, we created an ad hoc *Debug and Monitor Infrastructure* similar to the one in reference [2]. The kernel module *Reliability Stats Reader* provides the function *update_reliability_stats*. This function is inlined in the scheduler and executed at each tick. It gathers values from data to be monitored and writes them in the first free line of the *Double Buffer* of memory allocated in the user space. In this way, we are able to read data with a resolution of 10ms. We hide the overhead of exporting data from the kernel to a user space log by using a dedicated kernel double buffer memory. A userspace application (*Userspace Reader*) periodically polls the buffer to see whether it is full. Once full, it flushes it and swaps buffers. This is done through the *Reliability Stats Module Device Driver*, which works as an interface between kernel and user space for data export.

III. Results

Table I shows the average overheads of our governor. Latencies in the kernel space are computed by successively sampling the cycle register and computing the difference. Latencies in the user space, instead, are computed with the function *gettimeoftheday*.

The temperature module takes 5.2us to read the temperature sensors. This has been measured sampling the cycle register before and after the function *tsens_get_temp*. The sensors are sampled once a second, so the overhead is 0.00052%. The governor takes 14.22us to change the frequency. This has been measured sampling the cycle register before and after the function *cpufreq_driver_target*. The execution time of the

Overhead	Measure
Temperature sensors	$5.2us$
Frequency change	$14.22us$
STC algorithm + Frequency change	$14.65us$
Passing *H list*	$3.632ms$
LTC execution	$73.840ms$
Buffer reading	$30ms$
Memory difference w.r.t. original	8.192KB (0.13%)

TABLE I. Implementation overheads

entire STC algorithm inside of the governor, included the changing of frequency, is 14.65us. Since the governor rate is 10ms, this is an overhead of 0.14%. The time for passing the *H list* is 3.632ms. This has been measured executing *gettimeoftheday* before and after calling the procedure that passes the *H list*. The LTC execution time (73.840ms) is calculated from the moment the activation signal is received until the LTC goes back to sleep. Considering invoking the LI once a day, the overhead is negligible. The difference in memory between the original compiled kernel image, which is 6.2464MB, and the version with our governor is 8.192KB (0.13%). These measures tells that our governor has a very low impact.

To evaluate the behavior of reliability control over a target lifetime of 5 years in a reasonable experimental time, we set the LI duration to 500 jiffies (500 jiffies=30days). Figure 2 has two time axes. One is the virtual time in years (for reliability), the other is the experimental time in seconds. The platform executes two applications, one is H and the other is L. We chose the *BurnCortexA9* power virus[3] as test application. This application causes high voltage and temperature stress, thus it accelerates aging. Our aim is to have an easily controllable and high-stress workload for this experiment, and not a representative mobile workload (targeted in the last experiment). Figure 2 shows the reliability of each core over time and the target reliability $R_t = 0.8$ (first plot), frequency traces (from second to fifth plot) and the plot of task allocation.

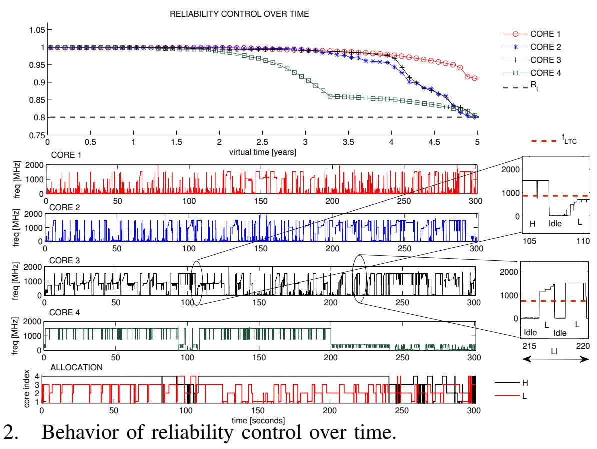

Fig. 2. Behavior of reliability control over time.

Our DRM strategy executes with the native Android scheduling and allocation. Reliability-aware allocation has been proposed in literature [12], but we do not consider it in this work, as it requires to modify the scheduler. This explains

[3]cpuburn power virus by Robert Redelmeier: it takes advantage of the superscalar architecture to maximize the CPU power consumption

why in the reliability plot of Figure 2, core 4 degrades faster[4]. This core is the one in which the H application is allocated for the most part of time, as shown by the last plot. This causes high execution frequency and voltage, thus a faster degradation. The core 4 at 200 seconds (corresponding to 3.3 years of virtual time) consumes all the reliability budget for executing the H tasks, thus it is forced to run at minimum frequency for the rest of time for meeting the target reliability. In this case, a reliability-aware scheduler would avoid to use core 4 for H tasks, and we can see that even the unmodified Linux scheduler does react somewhat in the right direction.

The reliability governor executes H tasks always at maximum frequency while the L tasks are executed at a frequency that meets the LTC reference voltage (V_{LTC}) over the LI, paying also the extra degradation induced by H tasks. This can be noticed in the first zoom of Figure 2. This shows the execution frequency of the 3rd core inside of a LI. The dashed line represents the frequency f_{LTC} correspondent to V_{LTC}. The LI starts with the execution of the H application, which borrows reliability budget allocated for the LI. After a idle period, core 3 executes the L application. The execution frequency is lower than f_{LTC}, as the L application pays the loan of the H application. The second zoom, instead, shows a LI in which only the L application is executing. Here the borrowing allows to leverage idle periods to increase the frequency for execution of L application. When no H tasks are present, L tasks can run at higher performance. The stairs present in the plot are a consequence of frequency/voltage quantization.

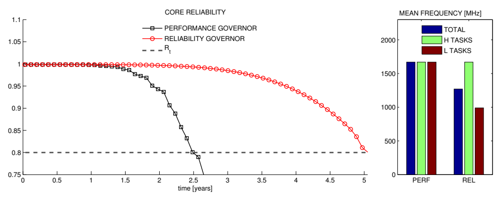

Fig. 3. Comparison against performance governor.

In Figure 3 we compare our governor against the performance governor for the same workload scenario, for a single core. Our governor achieves a 100% lifetime improvement, while providing comparable performance (in terms of mean applied frequency, averaged over system lifetime) for the execution of H tasks. L tasks pay for achieving the target lifetime by executing at a lower frequency.

Figure 4 reports the performance while varying the target lifetime in the DRM control, to show the trade off between lifetime and performance. The performance metric is derived from the normalized score of the benchmark for mobile devices *3D Mark IceStorm* [1]. This application runs over the entire system lifetime and is characterized as a L application. Performance equal to 1 corresponds to that obtained with the performance governor, while performance equal to 0 is that obtained with the powersave governor. The DRM control with target lifetime from 1 to 2.5 years has performance comparable to the case with the performance governor. This is consistent with results shown in Figure 3. If the target lifetime is 3 years, performance is a little lower. When the target lifetime is 5 years, performance is almost halved. This happens because the benchmark runs as a L application and sacrifices performance to meet the target lifetime. However, as shown in Figure 3, the use of the performance governor reduces considerably the lifetime.

[4]Clearly, modifying the Linux scheduler to account for processor reliability degradation would be desirable and it is technical feasible and we plan to propose reliability-aware schedulers to the Linux community in the near future.

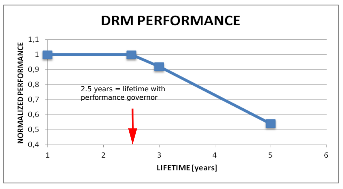

Fig. 4. Performance evaluation with 3D Mark IceStorm benchmark.

IV. Conclusion

In this paper we presented a novel *Reliability Governor* for workload-aware DRM, compatible with the existing Android/Linux software stack, that has been implemented on a real Android mobile device. Our solution has negligible overhead. We show the correctness of operation of the implemented technique and its effectiveness in guaranteeing the system target lifetime. Our solution achieves up to 100% of lifetime improvement with respect to traditional governors, while providing high performance for critical applications.

V. Acknowledgements

This work was supported by the FP7 project PHIDIAS (g.a. 318013) and the FP7 ERC Advance project MULTITHERMAN (g.a. 291125), and by Qualcomm, SRC (Task ID 2169.001) and NSF variability (g.a. 1029783).

References

[1] 3dmark for android , http://www.futuremark.com/benchmarks/3dmark.

[2] A. Bartolini. Dynamic power management: from portable devices to high performance computing. In *Ph.D. dissertation, DEIS., University of Bologna*, Bologna, Italy, 2011.

[3] A. Bartolini, M. Cacciari, A. Tilli, and L. Benini. Thermal and energy management of high-performance multicores: Distributed and self-calibrating model-predictive controller. *Parallel and Distributed Systems, IEEE Transactions on*, 24(1):170–183, 2013.

[4] J. Blome, S. Feng, S. Gupta, and S. Mahlke. Self-calibrating online wearout detection. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 109 –122, dec. 2007.

[5] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10 – 16, nov.-dec. 2005.

[6] K. Bowman, A. Alameldeen, S. Srinivasan, and C. Wilkerson. Impact of die-to-die and within-die parameter variations on the throughput distribution of multi-core processors. In *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, pages 50 –55, aug. 2007.

[7] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 179–194, New York, NY, USA, 2010. ACM.

[8] P. Gupta, Y. Agarwal, L. Dolecek, N. Dutt, R. K. Gupta, R. Kumar, S. Mitra, A. Nicolau, T. S. Rosing, M. B. Srivastava, S. Swanson, and D. Sylvester. Underdesigned and opportunistic computing in presence of hardware variability. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(1):8 –23, jan. 2013.

[9] E. Karl, D. Blaauw, D. Sylvester, and T. Mudge. Reliability modeling and management in dynamic microprocessor-based systems. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 1057 –1060, 0-0 2006.

[10] P. Mercati, A. Bartolini, F. Paterna, T. S. Rosing, and L. Benini. Workload and user experience-aware dynamic reliability management in multicore processors. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[11] V. Pallipadi and A. Starikovskiy. The ondemand governor: past, present and future. In *Proceedings of Linux Symposium, vol. 2, pp. 223-238*, 2006.

[12] F. Paterna, A. Acquaviva, and L. Benini. Aging-aware energy-efficient workload allocation for mobile multimedia platforms. PP(99):1, 2012.

[13] P. Singh, E. Karl, D. Blaauw, and D. Sylvester. Compact degradation sensors for monitoring nbti and oxide degradation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 20(9):1645 –1655, sept. 2012.

[14] P. Singh, E. Karl, D. Sylvester, and D. Blaauw. Dynamic nbti management using a 45 nm multi-degradation sensor. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 58(9):2026 –2037, sept. 2011.

[15] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The case for lifetime reliability-aware microprocessors. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 276 – 287, june 2004.

[16] C. H. K. van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1260–1265, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[17] L. Vincent, P. Maurine, S. Lesecq, and E. Beigné. Embedding statistical tests for on-chip dynamic voltage and temperature monitoring. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 994–999, New York, NY, USA, 2012. ACM.

[18] C. Zhuo, D. Sylvester, and D. Blaauw. Process variation and temperature-aware reliability management. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 580 –585, march 2010.