Reducing Power Consumption using Improved Wakelock on Android Platform

Joonkyo Kim and Jaehyun Park
School of Information and Communication Engineering
Inha University, Korea
Email: jkkim@emcl.org, jhyun@inha.ac.kr

***Abstract*—The power consumption is one of the most important issues on a mobile device because they usually use battery as a power source. The Android platform, a popular software platform for a hand-held device, also supports various power-saving schemes to reduce power consumption. This power-saving feature sometimes causes unwanted computational disruption. To avoid such disruption, the Android platform provides Wakelock to disable the power-saving mode temporarily. However, since Wakelock can be easily accessed through the user's API, improper use of Wakelock causes a huge extra power consumption. This paper proposes an improved Wakelock scheme that predicts the misuse of Wakelock. This improved Wakelock, called PR-Wakelock (Predict & self-Release Wakelock), does not only detect misuses of Wakelock, but also forcibly releases Wakelock for the system to go into a sleep mode. Several Android apps were implemented and an offline prediction software was also implemented on a linux platform for the simulating PR-Wakelock. The hit ratio of the proposed system obtained through an offline prediction software was close to 86.44%.**

***Keywords*-Energy-aware systems; power management; Android; Wakelock**

I. INTRODUCTION

As the mobile technology evolves, the number of applicable areas of the hand-held devices including a smartphone has dramatically increased during the last decade. The power consumption of a mobile device has become an important issue because of the expanded application area and limited battery life. With an expansion of the application area, the power consumption of a mobile device has become an important issue because most of the hand-held devices use a battery as a power source. To reduce the power consumption and extend battery life, various dynamic power management technologies have been adopted in the hand-held devices.

Dynamic Voltage and Frequency Scaling (DVFS) technique is one of the key technologies used in modern mobile devices. It controls the clock frequency and CPU supply voltage in order to reduce the switching power consumption of the digital components [1], [2]. Since the power consumption(P) is proportional to frequency(f) and voltage(V) as shown in (1), controlling the clock frequency of CPU and supplying voltage to CPU are effective ways to reduce power consumption at a hardware level [3].

$$P \propto f \cdot V^2 \quad (1)$$

Hence, the key idea of DVFS technique is to control both the supply-voltage and operating-frequency of CPU according to the software workload. To predict the workload of software, statistical analysis of the runtime distribution of software workload is usually used [4], [5].

Another power-saving technique at hardware level is Dynamic Power Management (DPM) , which predicts the idle time of the system and controls the power supply to the unused components such as external peripherals as well as CPU's internal modules [6], [7]. To predict the idle time of the system, Program Counter-based Access Predictor (PCAP) algorithm that monitors the applications' function-calls based on the I/O system-calls, was proposed [8]. Previous work showed that DPM using a hardware-based PCAP algorithm can reduce the power consumption up to 29% [9].

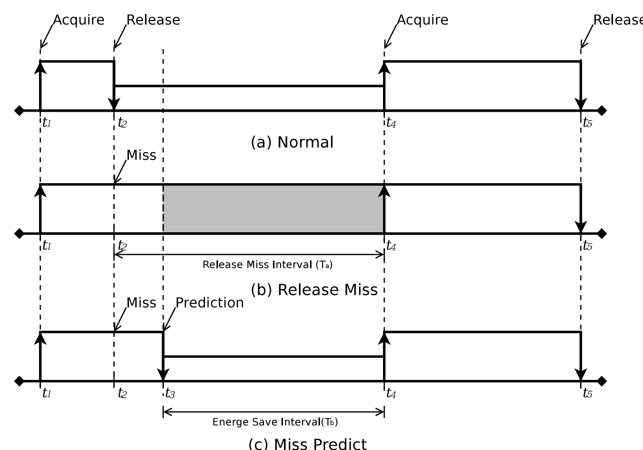

Figure 1. Behaviors of Wakelock

Even though the hardware-based power management techniques described above are effective to reduce the overall power consumption of mobile devices, they may cause unnecessary side effects such as slowing down of an application program or frequently disabling peripherals that may lose connectivity. To compensate these side effects that DVFS/DPM may cause, these power management features are able to be controlled by software explicitly in most mobile platforms. To run a background process without disruption on an Android platform, which is one of the most widely used software platforms in the mobile device market, a special power management module called *Wakelock* was introduced. Wakelock prevents CPU from entering the sleep or power-saving mode even when a mobile device is in an idle state such as a screen-saving mode. Even though Wakelock is a useful feature to manage power-saving feature, it should be handled very carefully because misuse of Wakelock means CPU never goes into power-saving mode, and, in turn, a huge amount of power may be wasted [10]. Fig. 1 shows the timeline of normal and abnormal use-case of Wakelock. Fig. 1(a) shows the timeline of normal case of Wakelock. Since Wakelock is acquired at t_1, which means that the power-saving feature is disabled, and released at t_2 normally, the system may enter into the power-saving mode until t_4 at which Wakelock is acquired again. However, if Wakelock is not released adequately at t_2 because

Copyright (c) IARIA, 2013.

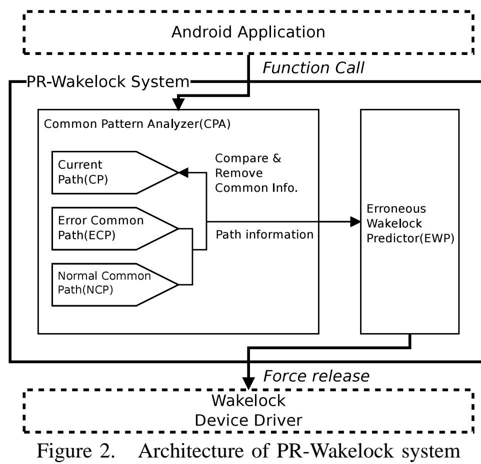

Figure 2. Architecture of PR-Wakelock system

of software bugs, the system remains in an active mode with power-saving mode disabled as shown in Fig. 1(b). In this case, unnecessary power consumption is expected during T_a period. Such inadequate situations of Wakelock have been reported occasionally in many Android applications because Wakelock can be accessed by any application through user's API [11]. Unfortunately, a systematic safeguard to prevent such undesirable situation does not exist in the Android platform. To lessen this kind of Wakelock error, analyzing method of the program at compile time was proposed recently [11]. The compile-time analysis, however, has limitations because every possible software flow cannot be analyzed at compile time without actual running it.

To overcome this limitation, this paper proposes a run-time algorithm to prevent power waste caused by unreleased Wakelock. Fig. 1(c) shows the benefits of the proposed algorithm in this paper. Even if Wakelock is not released at t_2, run-time algorithm proposed in this paper can detect unreleased Wakelock and forcibly release it at t_3, which prevents waste of power consumption during this unnecessary wake-up period, T_b.

This paper describes the key idea and algorithm of the run-time Wakelock predictor. However, since this work is still at a work-in-process stage, the implementation result is not shown in this paper. Instead, offline analysis software will prove the feasibility of the proposed idea.

This paper consists of five sections including this introduction. In Section II, the basic concept of PR-Wakelock system is proposed. For evaluating the performance of PR-Wakelock system, offline simulation results are shown in Section III. Finally, conclusions and future works are described in Section IV and Section V, respectively.

II. IMPROVED WAKELOCK SYSTEM

As described above, since unreleased Wakelock may cause a huge waste of power in the Android platform, this paper proposes an improved Wakelock system, which is called PR-Wakelock (Predict & self-Release Wakelock) as shown in Fig. 2. PR-Wakelock consists of two major modules; *Common Pattern Analyzer* (CPA) and *Erroneous Wakelock Predictor* (EWP).

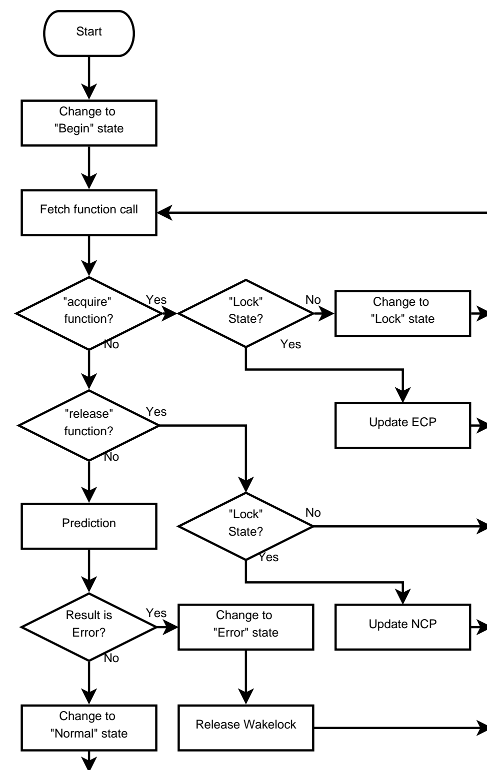

Figure 3. Flowchart of prediction algorithm

In this system, analyzing the sequence of *Wakelock-acquire* and *Wakelock-release* function-call is the key feature for predicting the erroneous Wakelock operation. CPA updates saved common paths when *Common Path(CP)* is revised. It checks sequence of the Wakelock function. If the *Wakelock-acquire* and *Wakelock-release* functions are invoked in order, CP stands for normal path because Wakelock is released normally after acquired. However, if *Wakelock-acquire* function of Wakelock is invoked twice in order, CP is at an erroneous path because Wakelock is locked until the next request. The algorithm of the PR-Wakelock system is presented in Fig. 3. For predicting erroneous Wakelock operation, EWP monitors function-calls at `Lock` or `Normal` state, and it performs prediction whenever function-call is detected. If EWP predicts that the current CP might be a part of an erroneous Wakelock sequence, the current Wakelock is forcibly released by EWP.

A. Common Path Analyzer

To analyze the usage of Wakelock in an application software, it is necessary to monitor and record the executing pattern of application software in real-time manner. The execution patterns are recorded as a unit of *Path*, defined as a "sequence of function calls" in this paper. CPA module in Fig. 2 forms a Path from the executing sequence of function-calls in real-time. Since only the usage of Wakelock is important in this paper, CPA module in Fig. 2 starts recording the executing sequence of function-calls whenever *Wakelock-acquire* function is invoked, and stops recording if another *Wakelock-acquire* or *Wakelock-release* function is invoked. The recorded Path in real-time is saved as CP (Current Path). Besides this CP, in CPA module, Error Common Path (ECP) and Normal Common Path (NCP) are also managed. NCP means that a

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-283-7

	0	A	G	C	A	T
0	0	0	0	0	0	0
G	0	0	1	1	1	1
A	0	1	1	1	2	2
C	0	1	1	2	2	2

Figure 4. Example of Analyzing Subsequence

part of software sequence that has both *Wakelock-acquire* and *Wakelock-release* function. This NCP stands for normal situation of Wakelock system because *Wakelock-release* function is invoked after *Wakelock-acquire* function. On the contrary, ECP represents a part of software sequence that doesn't have *Wakelock-release* after *Wakelock-acquire*. Using these three Paths', CPA analyzes the characteristics of software modules and provides information to determine erroneous situation of Wakelock.

CPA analyzes the recorded CP when another *Wakelock-acquire* or *Wakelock-release* function is invoked again. If *Wakelock-release* function ends the current recording sequence, the path stored in CP is considered as a normal Wakelock pattern and updated into NCP. If another *Wakelock-acquire* function is invoked, on the other hand, this path is considered as an erroneous case of Wakelock and updated in ECP. To extract the feature of executing pattern, *Common Path*, defined as a subsequence of function calls, is extracted from CP after applying Longest Common Subsequence (LCS) algorithm to NCP and ECP that stores the latest common path of normal and erroneous case, respectively [12], [13]. CPA also compares the subsequences between ECP and NCP in order to remove the common path between them because both ECP and NCP may have the Wakelock-related path in common. The LCS-based common path analyzing algorithm used in CPA is depicted in Fig. 4, in which two subsequence, AGCAT and GAC, are compared. LCS algorithms can build the table in Fig. 4 using a dynamic programming technique [12], [13]. Each alphabet in the figure means function name. The circled numbers are common functions, which is related to common path. As a result, CPA extracts AC as a common path from these two sequences. On the other hand, numbers with rectangle are different path position. These can be selected for removing common path from original paths. After removing common path from NCP and ECP, the remaining subsequence, GAT and G, are updated to NCP and ECP, respectively. CPA manages NCP and ECP in this manner and supply to the EWP so that it can detect erroneous situation.

B. Erroneous Wakelock Predictor

During recording the path into CP, EWP shown in Fig. 2 predicts a possible Wakelock errors whenever CP is updated. EWP determines whether current usage of Wakelock might be error or not based on the similarity between CP and either NCP or ECP. The similarity between the two paths are defined as the length of common path that can be determined by LCS algorithm. By comparing the similarity, EWP decides if the current executing sequence might be an error if the length of common path between CP and ECP is longer than the length

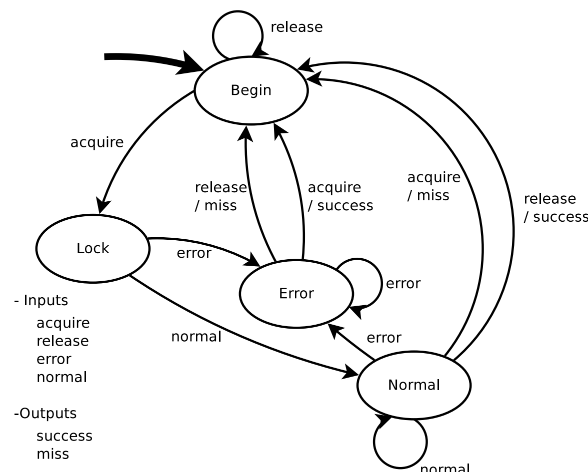

Figure 5. State of PR-Wakelock System

between CP and NCP as shown in (2). In (2) $\mathcal{L}$ denotes length of sequence and $\sqcap$ operator denotes LCS algorithm.

$$\mathcal{L}(\sqcap(CP, NCP)) + \Delta < \mathcal{L}(\sqcap(CP, ECP)) \qquad (2)$$

However, the subsquence stored in CP is not a complete sequence but a portion of it, because CP stores only executing sequence path after *Wakelock-acquire* function is invoked. This means that CP may contain insufficient information to predict accurately. To increase the accuracy of prediction regardless of incomplete information of CP, predictor maintains a threshold value, Δ, in (2) adaptively. To correct current information, EWP monitors function calls continuously until second Wakelock function is invoked, even though EWP decided current state. And EWP decides whether prediction is correct or not after it confirms second Wakelock function. At the beginning, Δ starts from zero. If the prediction is confirmed as wrong, threshold value increases.

Fig. 5 shows the state diagram of predictor. EWP starts from `Begin` state, and returns to `Begin` state when the path from CP is completed recording. If Wakelock is acquired at sometime, EWP moves to `Lock` state and starts prediction to decide whether CP is error or not. EWP can predict `Normal` state despite of CP is error actually, because CP is not enough long to predict correctly. So EWP continues prediction until another Wakelock function is invoked even though current state is `Normal`. On the other hand, if EWP decides CP is error, state moves to `Error` state. And then PR-Wakelock system forcibly releases Wakelock.

III. Simulation Results

To verify the feasibility of the proposed Wakelock system, a simulation system was implemented on Samsung Galaxy Note (SHV-E160K), which is based on Android 4.0.3 (Ice Cream Sandwich). In order to test the algorithm, three Android applications were implemented for simulating erroneous situation. First application was based on a socket application, which connected to a server and periodically sent data through network. In this case, Wakelock was acquired when this application tried to connect the socket, and released when transmission of data was finished normally. Second application was based on I/O operation of file system that tries to write

TABLE I. SIMULATION RESULTS

	Socket program	I/O program	Music player
Average	77.45	86.44	74.58
Maximum	84	90.28	79.31
Minimum	56.5	81.21	68.10
Std. Dev.	10.69	2.72	3.28
Δ(Final value)	15	17	23.2

random data to SD Card. Wakelock was used between the open and close functions of the *Writer* class. Last one was the music player application. That played MP3 music when user requested, and Wakelock was used to prevent stopping of the music when the system entered the power-saving mode. In all three applications, invocation of the *Wakelock-release* function was omitted randomly with uniform distribution.

During the software execution, the function-calls are logged using the *Debug* class provided by Android API. The function name, thread ID, and calling time were logged in order to analyze path information. Since the purpose of the test-bed system was to demonstrate the feasibility of the proposed algorithm, the actual analysis was not performed in real-time yet. Rather, the logged information was analyzed by the separate analysis and prediction software that was implemented on a Linux platform.

The prediction accuracy of the implemented CPA and EWP was evaluated through 10 executions of each application programs and the results were shown in Fig. 6 and Table I. It showed about 86.44% of predictions are valid on flash I/O based application. Because similarity between NCP and ECP is different according to application, Δ in (2) was also different.

IV. CONCLUSION

This paper proposes PR-Wakelock, which is an improved Wakelock algorithm for an Android platform. The proposed algorithm analyzes the behavior of Wakelock through the function-call analysis and predicts possible errors of Wakelock. Once improper operation of Wakelock is found, Wakelock is automatically released and CPU enters power-saving mode to reduce power consumption. Three Android softwares are implemented for simulating erroneous Wakelock behaviors, and offline analyzing software was also implemented on Linux platform for predicting error of Wakelock. According to the offline analysis, the prediction ratio of PR-Wakelock system was up to 86.44%. So, result shows that Wakelock error detection from function-call path analysis is feasible enough to apply on a real device.

V. FUTURE WORKS

PR-Wakelock system proposed in this paper has not been implemented yet. However, it can be implemented on Android system in order to monitor the operation of Wakelock in real-time, and, in turn, reduce the power consumption of Android-based mobile system. For this, CPA module can be implemented using *Debug* class device driver to monitor function calls without modifying Android framework itself. With implementing PR-Wakelock, the actual power consumption can be analyzed in the future work.

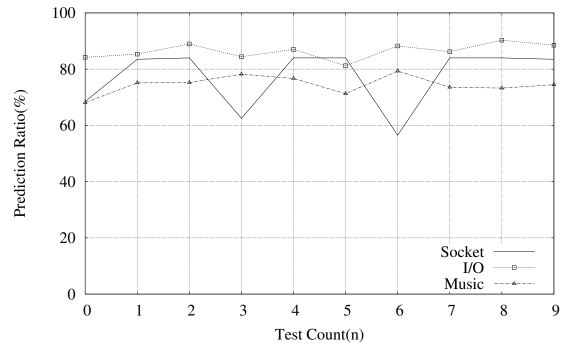

Figure 6. Simulation Results

REFERENCES

[1] J.-B. Lee, M.-J. Kim, S. Yoon, and E.-Y. Chung, "Application-support particle filter for dynamic voltage scaling of multimedia applications," IEEE Trans. Comput., vol. 61, no. 9, Sep. 2012, pp. 1256–1269.

[2] X. Chen, C. Xu, and R. P. Dick, "Memory access aware on-line voltage control for performance and energy optimization," in IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2010, Nov. 2010, pp. 365–372.

[3] W.-Y. Liang, S.-C. Chen, Y.-L. Chang, and J.-P. Fang, "Memory-aware dynamic voltage and frequency prediction for portable devices," in 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2008, pp. 229–236.

[4] J. Kim, S. Yoo, and C.-M. Kyung, "Program phase and runtime distribution-aware online dvfs for combined vdd/vbb scaling," in Design, Automation Test in Europe Conference Exhibition, 2009, DATE '09, Apr. 2009, pp. 417–422.

[5] ——, "Program phase-aware dynamic voltage scaling under variable computational workload and memory stall environment," IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 30, no. 1, Jan 2011, pp. 110–123.

[6] L. Cai, N. Pettis, and Y.-H. Lu, "Joint power management of memory and disk under performance contraints," IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 25, no. 12, Dec 2006, pp. 2697–2711.

[7] W.-K. Lee, S.-W. Lee, and W.-O. Siew, "Hybrid model for dynamic power management," IEEE Trans. Consum. Electron., vol. 55, no. 2, May 2009, pp. 656–664.

[8] C. Gniady, A. R. Butt, Y. C. Hu, and Y.-H. Lu, "Program counter-based prediction techniques for dynamic power management," IEEE Trans. Comput., vol. 55, no. 6, Jun. 2006, pp. 641–658.

[9] Y.-S. Hwang, S.-K. Ku, and K.-S. Chung, "A predictive dynamic power management technique for embedded mobile devices," IEEE Trans. Consum. Electron., vol. 56, no. 2, Jul. 2010, pp. 713–719.

[10] "PowerManager.WakeLock—Android Developer." [Online]. Available: http://developer.android.com/reference/android/os/PowerManager.WakeLock.html. [retrieved: Feb, 2013]

[11] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps," in Proceedings of the 10th international conference on Mobile systems, applications, and services MobiSys '12, 2012, pp. 267–280.

[12] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," Journal of ACM, vol. 24, no. 4, Oct. 1977, pp. 664–675.

[13] T. H. Comen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms, 2nd Ed." The MIT Press, 2001.