Energy Optimization in Android Applications through Wakelock Placement

Faisal Alam*, Preeti Ranjan Panda*, Nikhil Tripathi†, Namita Sharma*, and Sanjiv Narayan†
*Indian Institute of Technology Delhi, New Delhi, India
†Calypto Design Systems (India) Pvt. Ltd., Noida, India

***Abstract*—Energy efficiency is a critical factor in mobile systems, and a significant body of recent research efforts has focused on reducing the energy dissipation in mobile hardware and applications. The Android OS Power Manager provides programming interface routines called *wakelocks* for controlling the activation state of devices on a mobile system. An appropriate placement of wakelock *acquire* and *release* functions in the application can make a significant difference to the energy consumption. In this paper, we propose a data flow analysis based strategy for determining the placement of wakelock statements corresponding to the uses of devices in an application. Our experimental evaluation on a set of Android applications show significant (up to 32%) energy savings with the proposed optimization strategy.**

I. Introduction

Mobile systems are usually composed of several different devices with sophisticated functionality, whose efficient energy management is key to achieving reasonable battery life. Care must be taken to ensure that the power and energy behavior of modern smartphones are in line with expectations. Smartphone users often experience applications executing in the background more frequently than expected, keeping the devices awake for unexpectedly long durations. Frequent wake-ups, in turn, wake up other applications, leading to a cascading effect where unnecessary battery drain is caused by the power hungry resources such as WiFi and GPS.

Mobile operating systems such as Android offer software power management through programming primitives called *wakelocks* that enable the acquiring and releasing of control over different system components. Wakelocks are useful when a task needs to be executed without interruption. If the application needs to prevent a device from being put to sleep because of long periods of inactivity, wakelocks are typically acquired at the start of a task execution and released on completion. However, inappropriate use of these APIs may lead to power being wasted in the devices, thereby affecting battery life. These situations are sometimes difficult to trace because they do not affect the application functionality, but only lead to quicker battery discharge. In this paper, we propose a strategy that places the wakelock acquire and release function calls at appropriate positions in the application keeping energy efficiency in view. The wake up time of different components is minimized during application execution, thereby reducing the overall energy consumption.

II. Related Work

Early research in the field of mobile phone energy optimization includes backlight brightness adjustment [8]. Pathak et al. [7] reported on issues related to energy efficiency at various levels in the mobile systems such as firmware, application, and OS. Recent years have witnessed significant activity in this domain, and several tools have been developed to help application developers test their designs for energy efficiency.

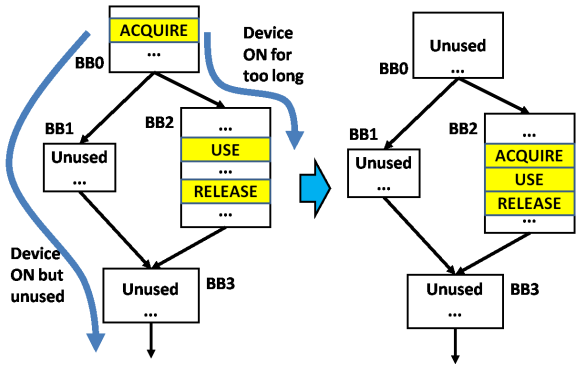
Several optimizations have been proposed at the Dalvik Virtual Machine (DVM) level, which is the software responsible for the execution and management of applications in the Android platform [1], [2]. AccelDroid [3] attempts to improve byte code execution by reducing the two level translation process in a co-designed processor to one step, thereby achieving performance improvement and energy savings. At the application level, researchers have identified sources of energy loss resulting from wakelocks being acquired in Android Java applications without being released [4]. Mandatory wakelock releases were suggested to be placed in the *try-finally* Java blocks to rectify such situations. Jindal et al. [9] addressed *sleep conflicts* that arise when the system is suspended with some components in high power state.

None of the earlier works on energy optimization in mobile devices focuses on automatic code transformations that intelligently place wakelock acquire and release function calls. Works that detect possible sources of energy loss [4], [9] are the closest to our proposed technique. The programmer is expected to correct the inefficiencies manually. In contrast, our work advances the state-of-the-art by providing a strategy to minimize system power and energy by automatically determining the best placement of wakelock acquire and release statements.

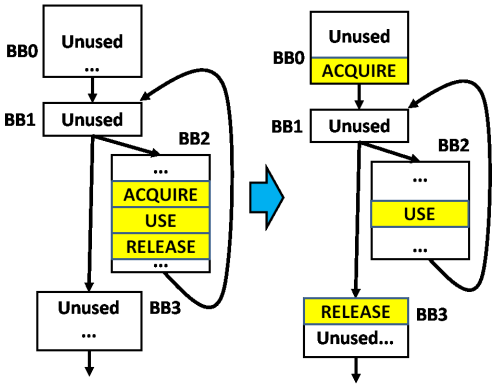
III. Illustrative Examples

Figure 1a shows the flow graph of an illustrative example with four Basic Blocks, defined as maximal sequence of instructions that can be entered at the first statement and exited at the last. The wakelock for a device is acquired in BB0, and released after use in BB2. This may cause two sources of energy inefficiency. First, the wakelock is never released in path $BB0 \rightarrow BB1 \rightarrow BB3$, leading to wasted energy when this path is traversed. Secondly, a significant amount of time lapses between the ACQUIRE and USE in path $BB0 \rightarrow BB2$, leading to wasted energy because the device is ON but unused. The desirable placements of the ACQUIRE and RELEASE are indicated on the right, immediately before and after the USE. However, the placement of ACQUIRE/RELEASE adjacent to the USE might not be always best. Figure 1b shows a loop ($BB1 \rightarrow BB2$) in which the wakelock is acquired and released in every loop iteration, immediately before and after use. The ACQUIRE and RELEASE functions themselves lead

978-3-9815370-2-4/DATE14/©2014 EDAA



(a) Bringing wakelock acquire/release closer to use



(b) Reducing overhead of acquire/release

Fig. 1: Wakelock placement impacts energy efficiency

to an energy overhead, and it is possible that this energy is larger than that incurred when the device is ON during typical instructions in the basic block. Hence, it may not be efficient to release the device in one iteration, only to acquire it immediately in the next. A more efficient solution could be to remove the ACQUIRE and RELEASE operations out of the loop, as illustrated on the right. The actual decision would depend on a quantitative comparison of the energy overhead of the wakelock ACQUIRE/RELEASE for the device, and the energy dissipated by the device during the intervening instructions in basic blocks BB2-BB1. In this paper, our contribution is to determine the energy-efficient positions of the acquisition and release statements.

IV. Optimization Strategy

We outline a Data Flow Analysis formulation for keeping track of the USEs of devices/resources, and deriving the best placements of the wakelock ACQUIRE and RELEASE statements in the code, with the objective of minimizing the energy wasted due to resources being unnecessarily kept active.

A. Data Flow Analysis

Consider a flow graph representing an application with N basic blocks labeled BB_i, $1 \leq i \leq N$. Assume we have M devices/resources $R_1, ..., R_M$. The flow graph is augmented by dummy nodes ENTRY and EXIT representing the start and end of the function being analyzed. Each statement in the basic block may access zero or more resources, and we maintain an array $USES[1..N, 1..M]$, where $USES[i,k]$ is set to TRUE if resource R_k is accessed in BB_i. We assume that estimates are available for power consumption of resources when they are on, and the expected execution durations of each instruction.

1) Basic Block Level Analysis: We maintain two Boolean arrays $WLIN[N,M]$ and $WLOUT[N,M]$ to track the information about the need to acquire resources at the entrance and exit of each basic block. The arrays are initialized by a local analysis at the basic block level. Figure 2 shows a basic block BB_i consisting of statements $S[i,1], ..., S[i,n]$, with $S[i,f]$ ($S[i,l]$) being the first (last) statement using R_k in BB_i. We first compute the energy estimates $E1^k$ and $E2^k$, where $E1^k$ is the energy consumed by R_k during the instructions before $S[i,f]$ ($S[i,1]$ to $S[i,f-1]$), and $E2^k$ is the energy consumed after $S[i,l]$ ($S[i,l+1]$ to $S[i,n]$). Let E^k be the sum of energy overheads for acquiring and releasing resource R_k. Clearly, if $E1^k > E^k$, then we can afford to turn off (release) R_k before entering BB_i, and instead, acquire the resource immediately before its use at $S[i,f]$. Similarly, if $E2^k > E^k$, we can release the resource right after its last use at $S[i,l]$, and acquire it in time for its next use after BB_i. This observation is captured by setting $WLIN[i,k]$ and $WLOUT[i,k]$ respectively to FALSE (or 'F'). If $WLIN[i,k] = F$, there is no need to acquire R_k as we enter BB_i. Similarly, if $WLOUT[i,k] = F$, there is no need to acquire R_k as we exit BB_i. In both cases, the decision on acquire/release of R_k can be taken locally, independent of other basic blocks. If these conditions are not true, we set the *WLIN/WLOUT* vaues to TRUE (or 'T'), indicating that we MAY need to keep R_k alive at the entrance/exit respectively of the basic block. This analysis is performed independently for all resources in all basic blocks.

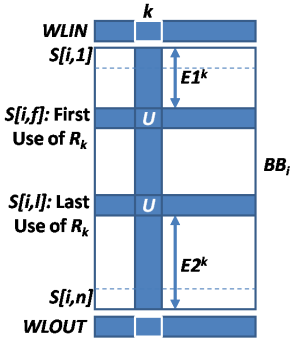


Fig. 2: Local data flow analysis within a Basic Block

2) Forward Propagation: The next step is to propagate the information in the individual basic blocks, beyond the basic block boundaries. Our problem has the characteristic of a *bi-directional data flow analysis*, since in an aggressive optimization, information needs to flow in both directions of the control flow graph—from ENTRY to EXIT (on when to acquire), and from EXIT to ENTRY (on when to release). However, in practice, this bi-directional analysis can be split in our example into two simpler separate passes—a forward propagation and a backward propagation.

In the forward direction, we formulate the following data flow analysis equations to propagate the *WLIN/WLOUT* values.

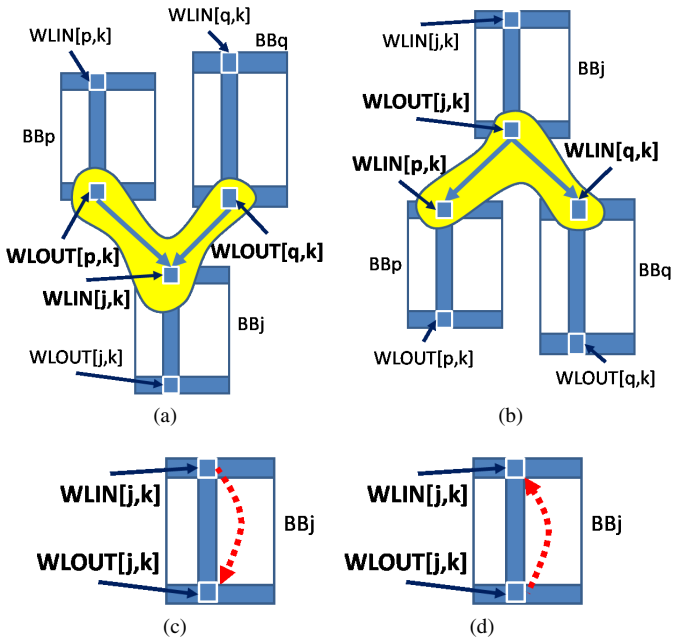


Fig. 3: Data Flow Analysis (a) Forward propagation from predecessors (b) Backward propagation from successors (c) Forward propagation within Basic Block (d) Backward propagation within Basic Block

First, the *WLIN*/*WLOUT* values of the ENTRY node are initialized to FALSE for all R_k since no resource needs to be acquired in the dummy node.

$$WLIN[\text{ENTRY}, k] = WLOUT[\text{ENTRY}, k] = F \quad (1)$$

A key definition is the behavior of the JOIN node with multiple predecessors. Consider BB_j with predecessors BB_p and BB_q (Figure 3a). The value of $WLIN[j,k]$ at the JOIN node is influenced by the *WLOUT* values of its predecessors in the following way:

$$WLIN[j,k] = WLIN[j,k] \bigwedge \left(\bigvee_{i \in Pred(j)} WLOUT[i,k] \right) \quad (2)$$

Clearly, if $WLIN[j,k]$ has already been established to be FALSE by a local basic block level analysis, then the decision stands irrespective of the behavior of any other basic block; therefore we use the AND operator with $WLIN[j,k]$. Similarly, if the *WLOUT* values of ALL predecessors of BB_j are FALSE (in our example, both $WLOUT[p,k]$ and $WLOUT[q,k]$ are FALSE), meaning that it is better to release R_k at the exit of ALL predecessors of BB_j, then we can turn $WLIN[j,k]$ also to FALSE. Even if R_k is used in BB_j, we can acquire it inside the basic block, just before its use. In other cases, we can retain the original value for $WLIN[j,k]$.

If R_k is used in BB_j ($USES[j,k] = TRUE$), then the value of $WLOUT[j,k]$ remains unchanged from the initial value. However, if there is no use of R_k, we have an opportunity to propagate a FALSE value from $WLIN[j,k]$ to $WLOUT[j,k]$ — if BB_j does not use R_k, and the wakelock is released at its entry, then we have no need for it in the basic block, and can propagate this state to succeeding basic blocks (Figure 3c). The AND with $WLOUT[j,k]$ follows the same logic as earlier — if the wakelock for R_k is not needed at BB_j's exit according to the local analysis, it will never be needed.

$$WLOUT[j,k] = WLOUT[j,k] \wedge (WLIN[j,k] \vee USES[j,k]) \quad (3)$$

3) Backward Propagation: In the backward propagation step, the corresponding equations are (Figures 3b and 3d):

$$WLIN[\text{EXIT}, k] = WLOUT[\text{EXIT}, k] = F \quad (4)$$

$$WLOUT[j,k] = WLOUT[j,k] \bigwedge \left(\bigvee_{i \in Succ(j)} WLIN[i,k] \right) \quad (5)$$

$$WLIN[j,k] = WLIN[j,k] \wedge (WLOUT[j,k] \vee USES[j,k]) \quad (6)$$

As a pre-processing step, we replace a loop that does not access any of the resources, by a dummy basic block with an estimated energy dissipation for resource R_k equal to the energy dissipation computed for the resource for the whole loop duration. Conditionals not accessing resources are also similarly simplified.

B. Wakelock Placement

Following the data flow analysis, the next step is to insert the wakelock ACQUIRE and RELEASE statements in the appropriate places in the code. Considering the individual basic blocks first, if we have a USE of resource R_k in BB_i, with $WLIN[i,k] = FALSE$, then we insert an ACQUIRE for R_k just before the first USE, since the wakelock won't be acquired at the basic block entry (Figure 4a). Similarly, a RELEASE is placed after the USE when $WLOUT[i,k] = FALSE$ (Figure 4b). Crossing basic block boundaries, a wakelock ACQUIRE/RELEASE statement is necessary when we have an edge in the flow graph with a change in value for a resource. If the value changes from $T \rightarrow F$ for a resource R_k when following an edge (i.e., we have BB_j as a successor to BB_i, with $WLOUT[i,k] = T$ and $WLIN[j,k] = F$), we insert a wakelock RELEASE statement at the entry of BB_j (Figure 4c). To prevent releasing an already released resource (because of some other $F \rightarrow F$ edge ending at BB_j), the release statement can first check if the resource is already acquired. The $F \rightarrow T$ edge is similarly treated (Figure 4d).

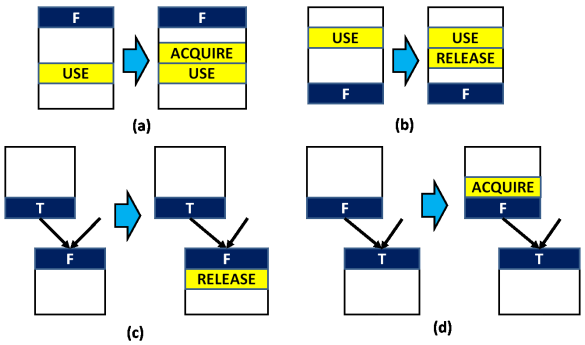


Fig. 4: Wakelock placement after Data Flow Analysis (a) False before USE (b) False after USE (c) T→F Edge (d) F→T Edge

V. Experimental Evaluation

We conducted experiments to evaluate the effectiveness of our proposed power optimization strategy. We used PowerTutor [4], an application that estimates the energy consumption for Android applications, using a power model based on the HTC Dream phone. The application power simulation was performed on the *GingerBread* (2.3.4) Android version on a Samsung GT-S5670 phone (the host platform), which has a configuration similar to the target HTC phone, and would generate representative stimulus for the target. The total energy dissipation was compared between the unoptimized and optimized versions of the applications. The generated power data corresponds to the HTC target. We evaluated the effectiveness of our proposed optimization strategies on five different open source applications [10]. The application characteristics and relevant optimizations are described below.

The *HttpMon* application (Resources accessed: CPU, Display, Network Connection) is used for finding how long a website was up. It establishes contact with a given website at specified time intervals and notifies the user in case of an unsuccessful ping. The wakelock (PARTIAL_WAKE_LOCK) is acquired in the *onReceive*() function which keeps the CPU on for the duration it is acquired. It is acquired much before its actual use for the network connection and released only in the *Finally* block. In the optimized version, the wakelock acquire instruction was moved closer to the network connection use. In a second optimization, slack was introduced in the wakeup timing of an Alarm manager, leading to energy savings.

The *PocketTalk* application (Resources accessed: CPU, Display, Vibrator) reads aloud a text message received by the phone. It is also configurable to encode each character into Morse code and communicate the message through vibrations. The wakelock (SCREEN_DIM_WAKE_LOCK) controlling the CPU and display is acquired before converting the text to Morse code. If the phone's vibrator is off, leading to an exception, the wakelock is left acquired for a long duration. This is optimized by adding a try-catch block that releases the wakelock on encountering the exception.

The *Pedometer* application (Resources accessed: CPU, Display, Accelerometer) computes the number of steps a user has taken based on the number of shakes experienced by an accelerometer. Here, the wakelock (PARTIAL_WAKE_LOCK/ SCREEN_DIM_WAKE_LOCK) is acquired (or released) when the start (or stop) button is pressed. This leads to unnecessary battery drainage as the wakelock is left acquired even when there is no accelerometer event. This is optimized to acquire and release the wakelock in the listener function for the shaking effect. The device now enters the high energy state only when a shake effect is experienced.

The *SalatTimes* application (Resources accessed: CPU, Display, GPS/WiFi, Alarm) sounds an alarm at the time of the Muslim prayer. It calculates the time of the prayer based on the device coordinates. The application starts the service for displaying the notification and sounding an alarm at the scheduled prayer time. However, after the notification, it does not stop the service, leading to unnecessary energy consumption. The optimized application acquires the wakelock (PARTIAL_WAKE_LOCK) at the service starting point and releases it soon after the notification and alarm instruction. The service is re-started and the wakelock is re-acquired when the *Intent* is next registered.

Figure 5 gives a comparison of the energy consumption by the unoptimized and optimized versions of the applications, with the energy of the unoptimized application treated as 100%. The 100% corresponds to absolute values for the energy ranging from 80mJ to 458J. The applications were executed for durations ranging from 2 to 11 minutes in our experiments. We observe energy reductions between 8% and 32% as a result of applying the optimizations.

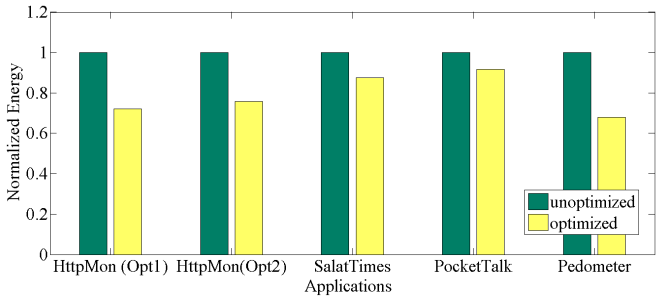


Fig. 5: Optimization results for different Apps

VI. Conclusions

We addressed the issue of energy optimization in mobile applications by carefully controlling the placement of wakelock statements that are responsible for the acquisition and release of devices. We outlined a data flow analysis formulation for determining the exact placement of wakelock statements corresponding to all the uses of devices in an application. Energy dissipation is reduced by ensuring that devices are maintained in active state only for the necessary duration. Our experiments on a set of applications indicate an overall energy reduction of up to 32% when the optimization is applied. Future research includes the investigation of a closer co-ordination between the application, compiler, and operating system on energy efficiency issues.

References

[1] C.-S. Wang, G. Perez, Y.-C. Chung, W.-C. Hsu, W.-K. Shih, and H.-R. Hsu, "A Method-based Ahead-of-Time Compiler for Android Applications," CASES, 2011.

[2] G. A. Perez, C.-M. Kao, Y.-C. Chung, and W.-C. Hsu, "A hybrid just-in-time compiler for Android: comparing JIT types and the result of cooperation," CASES, 2012.

[3] C. Wang, Y. Wu, and M. Cintra, "Acceldroid: Co-designed acceleration of android bytecode," CGO, 2013.

[4] 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," MobiSys, 2012.

[5] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," CODES+ISSS, 2010.

[6] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with Eprof," EuroSys, 2012

[7] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices", HotNets, 2011.

[8] R. Cornea, A. Nicolau, and N. Dutt, "Software Annotations for Power Optimization on Mobile Devices", DATE, 2006.

[9] A. Jindal, A. Pathak, Y. C. Hu, and S. P. Midkiff, "Hypnos: understanding and treating sleep conflicts in smartphones", Eurosys, 2013.

[10] Google Play, http://play.google.com