Fine-Grained Energy Modeling for Mobile Application Source Code

ABSTRACT

Energy efficiency has a significant influence on user experience of battery-driven devices such as smartphones and tablets. The goal of an energy model for source code is to lay a foundation for the application of energy-saving techniques during software development. The challenge is to relate hardware energy consumption to high-level application code, considering the complex run-time context, such as user inputs and the virtual machine. Traditional techniques build the energy model from bottom to top, mapping a hardware energy model onto software constructs; this approach faces obstacles when the software stack consists of a number of abstract layers. Another approach that has been followed is to utilize hardware or operating system features to estimate software energy information at a coarse level of granularity such as blocks, methods or even applications. In this paper, we propose a fine-grained energy model for the source code, which is based on "energy operations" identified directly from the source code and able to provide more valuable information for code optimization.

1. INTRODUCTION

In February 2015, the penetration of smartphones was about 75% in the U.S. This figure is expected to reach 85% by December 2015 [4]. With the improvement of hardware processing capability and software development environment, applications are becoming much heavier and more PC-like. At the same time, users are frustrated by limited battery capacity – applications running in parallel could easily drain a fully-charged battery within 24 hours.

Furthermore, current software development is performed in an energy-oblivious manner. Throughout the engineering lifecycle, most developers and designers are blind to the energy usage of code written by themselves. On the other hand, it has been estimated that energy saving by a factor of as much as three to five could be achieved solely by software optimization [8]. To realize this, the first step is to understand the energy characteristics of the source code at different levels of granularity and from different points of view.

Energy modeling of software needs to bridge the gap between high-level source code and hardware, where energy is consumed, in order to enable energy accounting of code. However, traditional bottom-to-top modeling techniques [6, 22, 26, 27] face obstacles when the software stack of the system consists of a number of abstract layers. On the Android platform, say, the source code is in Java and then translated to Java byte-code, further to Dalvik [3] byte-code, native code and machine code and finally has chance to execute on the processors and consume energy. Consequently, the modeling task has to characterize the links among all the layers.

Instead of building a software energy model layer by layer, another approach to acquiring software-level energy information is to use the hardware readings, like CPU state residency, CPU utilization, L1/L2 Cache misses and battery trace, as predictors of software energy use [7,21,28,29]. However, they are only capable of obtaining energy information at a coarse level of granularity such as blocks, methods or applications. Two pieces of work [9, 13] result in source-line energy information. The former requires low-level energy profiles. The latter employs accurate measurement to acquire the energy consumption of source lines.

The main contribution of this paper is to develop a fine-grained software energy model based on "energy operations", which are finer-grained than source lines. Rather than building the software energy model from bottom to top, we identify the basic energy operations (such as multiplications, comparisons and method invocations) directly from the source code and find their correlations to the energy cost by analyzing a diversity of well-designed execution cases. The resulting energy model implicitly includes the effect of all the layers of the software stack down to the hardware.

The key contributions of this work are the following:

- A source-code energy model based on fine-grained energy operations, from which energy operations of source lines, blocks and so on can be derived.
- An approach in which an explicit low-level energy model or hardware profile is not required, since comprehensive information is identified directly from the source code and statistical analysis of a wide range of execution cases.

Our target platform is an Android development board with two ARM quad-core CPUs, and the source code in our study is a game engine used in games, demos and other interactive applications. The result shows that the model's inference accuracy achieves about 85.0%. Based on this model, we are able to obtain more comprehensive energy information than coarse-grained models or techniques could provide.

Firstly, in Section 2, we introduce the approach of identifying energy operations from the source code. The architectural setup and the design of execution cases are detailed in Section 3. We elaborate the data collection and the model construction separately in Section 4 and Section 5, based on which the fine-grained energy accounting is shown in Section 6.

2. BASIC ENERGY OPERATIONS

There are two reasons why we choose to build the source code energy model based on "energy operations". Firstly, an energy operation is "atomic", by which we mean that all the statements, source lines, blocks and methods are made up of a certain number of kinds of operations (in the experiment, we have 120 operations). Secondly, it is fine-grained. Energy information at the level of source lines or methods is useful; however, information at source line level could not distinguish energy consumption of two operations in the same source line, for example.

Energy operations are identified directly from source code. The enumeration of the operations is inspired by Java semantics [5], which specifies the operational meaning, or behavior, of the Java language, which is the target language in the experiment. We intuitively identify semantic operations that perform operations on the state and may be energy-consuming, and let them be our energy operations. Ones that have little or no energy effect will automatically be identified by the regression analysis in the later stage of the analysis. Table 1 lists 14 representative operations out of a total of 120 in the experiment.

Table 1: Examples of Energy Operations

Operation	*Identified where:*
Method Invocation	*one method is called*
Parameter_Object	*Object is one parameter of the method*
Return_Object	*the method returns an Object*
Addition_int_int	*addition's operands are integers*
Multi_float_float	*multiplication's operands are floats*
Increment	*symbol "++" appears in code*
And	*symbol "&&" appears in code*
Less_int_float	*"<"'s operands are integer and float*
Equal_Object_null	*"=="'s operands are Object and null*
Declaration_int	*one integer is declared*
Assign_Object_null	*assignment's operands are Object and null*
Assign_char[]_char[]	*assignment's operands are arrays of chars*
Array Reference	*one array element is referred*
Block Goto	*the code execution goes to a new block*

Table 2: Examples of Library Functions

Class	Function
ArrayList	*add, get, size, isEmpty, remove*
	glBindTexture, glDisableClientState
	glDrawElements, glEnableClientState
GL10	*glMultMatrixf, glTexCoordPointer*
	glPopMatrix, glPushMatrix
	glTexParameterx, glVertexPointer
Math	max, pow, sqrt, random
FloatBuffer	*position, put*

They include arithmetic calculations like *Multi_float_float*, *Addition_int_int*, in which operands types are explicit, as well as *Increment* whose operand is implicitly an integer. Boolean operations and comparisons, such as *And*, *Less_int_float* and *Equal_Object_null* also form one major part. *Method Invocation* and *Block Goto* are important for the control flow which plays a key role in the execution of the code. Assignments and *Array Reference* will unexpectedly take a significant amount of the application's energy consumption, as will be shown in Section 6.

The application also employs a diversity of library functions that may be written in different languages and at lower levels of the software stack. On the other hand, usually a limited number (67 in the experiment) of library functions are frequently called in one application. So we treat them as basic modeling units. The examples of highly-used library functions in the experiment are shown in Table 2. For instance, the functions in the class of *GL10* are responsible for graphic computing.

3. EXPERIMENTAL SETUP

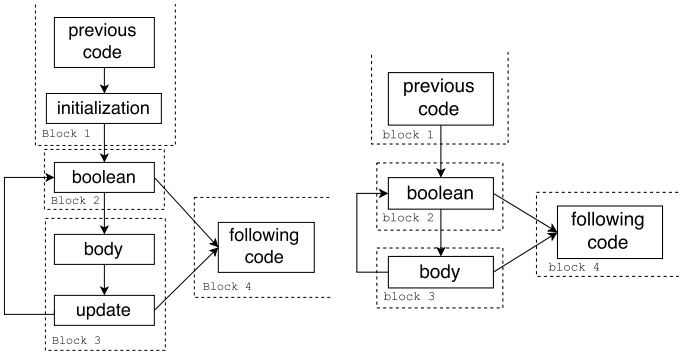
In this section, we will introduce the setup of the target device and the employed source code. We also explain the design principles of the execution cases.

3.1 Target Device

Experimental target: we employ an Odroid-XU+E development board [19] as the target device. It possesses two ARM quad-core CPUs, which are Cortex-A15 with 2.0 GHz clock rate and Cortex-A7 with 1.5 GHz. The eight cores are logically grouped into four pairs. Each pair consists of one big and one small core. So from the operating system's point of view there are four logic cores. In our experiment, we turn off the small cores and run workload on big cores at a fixed clock frequency of 1.1 GHz. We do this in order to remove the influence of voltage, clock rate and CPU performance on the power usage.

Power Reading Script: Odroid-XU+E has a built-in power monitoring tool to measure the voltage and current of CPUs with a frequency of 30 Hz and updates the samples in a log file. We wrote a script to obtain the samples from the file. During execution we run the script on an idle core to minimize its influence on the application.

Note that the power monitor gives two sequences of power samples: one is for the big cores and the other is for the small cores. We pick the sequence of power samples of the big cores, because we only run workload on them.



(a) `for` loop (b) `while` loop

Figure 1: Block division of `for` and `while` loops in control flow graph.

3.2 Target Source Code

The target source code is the Cocos2d-Android [2] game engine, a framework for building games, demos and other interactive applications. It also implements a fully-featured physics engine. Games are increasingly popular on mobile phones, and the applications include more and more fancy and energy-consuming features, requiring high CPU performance. Energy modeling and accounting, explained in the rest of this paper, will present opportunities to guide software development towards energy efficiency.

3.3 Design of Execution Cases

The execution cases whose energy usage is measured and analyzed represent typical sequences of actions during game, including user inputs. We focus on a `Click & Move` scenario, in which the sprite (the character in the game) moves to the position where the tap occurs. To simulate the game scenarios under different sequences of user inputs, we script with the Android Debug Bridge [1] (ADB) , a command line tool connecting the target device to the host, to automatically feed the input sequences to the target device.

In order to obtain a more varied set of execution cases, we vary the executions of individual basic blocks in the code. This is achieved by systematically removing a set of blocks for each execution case, using the control flow graph obtained using the Soot tool [24]. We ensure that each block could be removed in some execution case. Thus an execution case is made up of one user input sequence and one set of basic blocks.

4. DATA COLLECTION

In this section, we describe the collection of data on the number of times each operation executes and the energy consumption of an execution case, based on which we construct the energy model.

4.1 Number of Executions of Operations

To obtain the number of times that each operation executes in an execution case, we need to determine at which level of granularity to track the execution. We choose the level of "blocks". A block is a sequence of consecutive statements, without loops or branches. It is sufficient to track block executions, since if one part of a block is processed, the rest certainly will be processed as well.

We could consider collecting data at other levels of granularity. Tracing individual statements might overload the capacity of the target device. On the other hand, methods or classes are unsuitable execution units, since we cannot determine which parts of the method or class will be active during the execution, and this information about energy operations is lost.

We then divide the source code into blocks. For individual syntactic structures, we deal with block division case by case.

`For` loops and `while` loops are handled as shown in Figure 1. In a `for` loop, the header usually has three segments which are *initialization*, *boolean* and *update*. They are divided into three different blocks. Similarly, we set the `while` header itself as a block ("block 2" in Figure 1b). In order to build the log, we instrument the source code with a log instruction at the beginning of each block.

The generic view of the collection of the operation-execution data is displayed in Figure 2. We build a dictionary showing, for each block, the number of occurrences within it of each energy operation, such as those in Table 1. This dictionary is built using a parser that traverses all the blocks in the code.

Then, using the log file recording the processed blocks, together with the dictionary, we can sum up the number of times that each energy operation is executed during an execution case. To be more precise, let B_i be the number of times that the i^{th} block is executed (this is obtained from the log file). Let $O_{i,j}$ be the number of occurrences of operation j in block i (this is obtained from the dictionary). Then the total number of executions of the j^{th} operation is $\sum_{i=1}^{n}(B_i * O_{i,j})$, where n is the total number of blocks.

4.2 Energy Approximation from Power Samples

We write a script to obtain the power samples from the built-in measurement component with a frequency of 30 Hz. The power samples are the discrete values sampled from the power trace; we approximate energy consumption by calculating Equation (1): $p = power(t)$ is the power trace, that is, the continuous power-vs-time function; $power(t_i)$ is the power sample at time-stamp t_i; Δ_i equals to $t_i - t_{i-1}$, which is the interval between two sequential samples.

$$E = \int_{t_0}^{t_n} power(t)\,dt \approx \sum_{i=1}^{n} power(t_i)\cdot\Delta_i \qquad (1)$$

$$where \quad t_0 \le t_1 \le t_2 \cdots \le t_{n-1} \le t_n$$

4.3 Challenges in Practice

Measurement limitation: the sampling rate of the built-in power monitor is 30 Hz. However, the instruction execution rate is about several million per second. That means, one power sample measures the energy cost of hundreds of thousand instructions. Even though the state of the art of the power measurement can reach a sampling rate of tens of KHz [11], one power sample still includes up to thousands of instructions.

To deal with this problem, we first lengthen the sessions of all the execution cases to above 100 seconds, and then run each case for ten times to calculate their average energy cost. Compared with the execution cases that only run once with sessions around one second, this approach can reduce the error of measuring energy consumption of the code by three orders of magnitude.

Run-time context: during the running of the application, the Dalvik virtual machine performs garbage collection, which is not part of the application and still could be included in the power samples.

The Dalvik virtual machine produce time-stamp logs when launching the garbage collection procedure. We consider the garbage collection as one library function, so it will be integrated in the model.

Code instrumentation and power reading script: although the instrumentation is at block level rather than statement level, its impact on energy consumption is still not negligible and its cost is as much as 50% of the application's energy consumption itself. Also, the energy cost of the power reading script is up to 5% of the application's consumption.

We followed three experimental principles to address this problem. Firstly, for each execution case, the log of the execution path and of the power samples are separated into two separate runs. In the first round, we record the execution path without reading power samples. In the second round, we only trace power and disable the instrumented log instructions. So for each execution case, the instrumentation for logging the execution path will not influence the power samples.

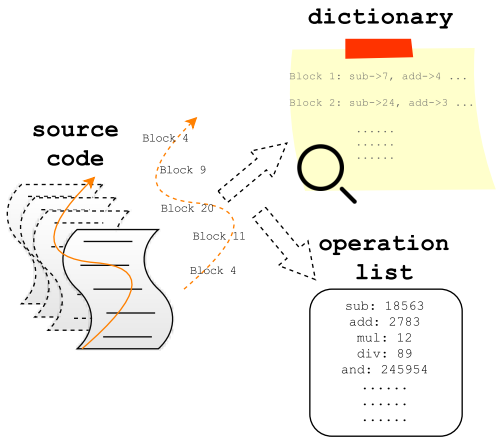


Figure 2: The flow of the operation-execution data collection.

Secondly, in each of the two runs, the main process of the application is allocated to one CPU core, while the thread logging execution path or power samples is allocated to another CPU core, minimizing effects due to interaction of the threads.

Thirdly, we design one "idle execution case" paired with each execution case; this only runs the power reading script without the application. By this means we can get the energy consumption of the main application process by excluding the cost of the "idle execution case" from the execution case. Note that the durations of execution cases are different, so we need to have a distinct "idle execution case" for each execution case.

In summary, each execution case will be run 21 times: once for tracing the execution path; ten times for calculating the average energy consumption of the "idle execution case", and ten times for calculating average energy consumption of the execution case.

5. MODEL CONSTRUCTION

The entire energy consumption is composed of three parts: the cost of energy operations, the cost of library functions and the idle cost. The aimed model is formalized in Equation (2). The cost of energy operations is the sum of $Cost_{op_i} \cdot N_e(op_i)$ (the cost of one operation multiplied by the number of its executions), where $op_i \in Energy\,Ops$. $Energy\,Ops$ is the set containing all the operations. The cost of library functions is the sum of $Cost_{func_i} \cdot N_e(func_i)$ (the cost of one library function multiplied by the number of its executions), where $func_i \in Lib\,Funcs$. $Lib\,Funcs$ is the set of library functions. The $Idle\,Cost$ is the energy consumption of the "idle execution case". The lengths of case sessions are varying, so the $Idle\,Cost$ is different for each execution case.

$$E = \sum_{op_i \in Energy\,Ops} Cost_{op_i} \cdot N_e(op_i) \qquad (2)$$

$$+ \sum_{func_i \in Lib\,Funcs} Cost_{func_i} \cdot N_e(func_i) + Idle\,Cost$$

The model construction is based on regression analysis, finding out the correlation between energy operations and their costs from the data obtained in the execution cases. We set out the collected data in the matrices in Equation (3). The leftmost matrix (N) contains the execution numbers of l operations (including energy operations and library functions) in m execution

Table 3: NMAE in Cross Validation

Set \ Round	1st	2nd	3rd	4th
Training set	17.7%	15.0%	13.6%	18.9%
Validation set	14.2%	14.2%	19.7%	17.8%

cases, acquired as shown in Section 4. Each row indicates one execution case. Each column represents one operation. The vector ($\vec{cost}$) in the middle contains the costs of l operations, which are the values we are aiming to estimate. The vector ($\vec{e}$) on the right of the equal mark contains the measured entire energy costs of the execution cases. So for each execution case, the entire energy cost is the sum of the costs of operations. It should be noticed that the energy costs $\vec{e}$ exclude the *Idle Cost* which is measured when no application workload is being processed.

$$\begin{pmatrix} n_1^{(1)} & n_2^{(1)} & \dots & n_l^{(1)} \\ n_1^{(2)} & n_2^{(2)} & \dots & n_l^{(2)} \\ & \dots & \dots & \\ n_1^{(m-1)} & n_2^{(m-1)} & \dots & n_l^{(m-1)} \\ n_1^{(m)} & n_2^{(m)} & \dots & n_l^{(m)} \end{pmatrix} \times \begin{pmatrix} cost_1 \\ cost_2 \\ \dots \\ cost_l \end{pmatrix} = \begin{pmatrix} e_1 \\ e_2 \\ \dots \\ e_{m-1} \\ e_m \end{pmatrix} \quad (3)$$

Inevitably, the execution path and power samples are not absolutely accurate. Furthermore, the energy model in reality is unlikely to be completely linear. For these reasons Equation (3) may be unsolvable, that is, the vector $\vec{e}$ is out of the column space of N. We thus employ the gradient descent algorithm [18] to compute the approximate values of $\vec{cost}$.

The elements of $\vec{cost}$ are randomly initialized and then improved by the gradient descent algorithm iteratively. We first introduce the error function J (computed by Equation (4)) which indicates the quality of the model. The smaller J is, the better the model is. $\vec{n^{(i)}}$ is the i^{th} row in N, $\vec{cost}$ is the middle vector above. $\vec{n^{(i)}} \times \vec{cost}$ is the estimated energy cost for the i^{th}execution case, $e^{(i)}$ is its observed energy cost. J first computes the sum of the squared values of the estimate errors of all the execution cases, which is afterwards divided by $2m$ to get the average value.

$$J(cost_1, cost_2, ...cost_l) = \frac{1}{2m}\sum_{i=1}^{m}(\vec{n^{(i)}} \times \vec{cost} - e^{(i)})^2 \quad (4)$$

$$cost_j := cost_j - \alpha \frac{\partial J(cost_1, ...cost_j, ...cost_l)}{\partial cost_j} \quad (5)$$

$$= cost_j - \alpha \frac{1}{m}\sum_{i=1}^{m}(\vec{n^{(i)}} \times \vec{cost}) \cdot n_j^{(i)}$$

$$j = 1, 2, ...l$$

The idea of gradient descent is to minimize J by repeatedly updating all the elements in $\vec{cost}$ with Equation (5) until convergence. The partial derivative of the function J on $cost_j$ gives the direction in which increasing or decreasing $cost_j$ will reduce J. Every element ($cost_j$) of $\vec{cost}$ is updated one by one in each iteration. The value α determines how large the step of each iteration is. If it is too large, the extremum value will possibly be missed; if too small, the minimizing process will be rather time-consuming. It needs to be manually tuned. Theoretically, the gradient descent algorithm could only find the local optima. In practice, we randomly set the values in $\vec{cost}$ and restart the entire gradient decent procedure for several times to look for global optima.

To validate the model, we apply the four-round cross validation procedure: the set of execution cases are randomly divided into four subsets; in each round, one of them is chosen to be the validation set and the others together to be the training set. In Table 3, we can see the Normalized Mean Absolute Error (NMAE) of the model in training and validation sets in the four rounds. The NMAE is a well-known statistical criterion that shows how well the estimated value matches the measured one. It is computed by Equation (6), the mean value of normalized difference between the predicted energy cost $\hat{e}$ and the measured cost e. The lower the ratio the better the result. In our cross validation, the NMAE in training sets ranges from 13.6% to 18.9%, and in validation sets from 14.2% to 19.7%. The model produced in the 2nd round is chosen to help analyze the energy property of the code in Section 6, because it has a good balance on both training and validation sets. Its NMAE is around 15.0%, which means the model's inference accuracy is around 85.0%.

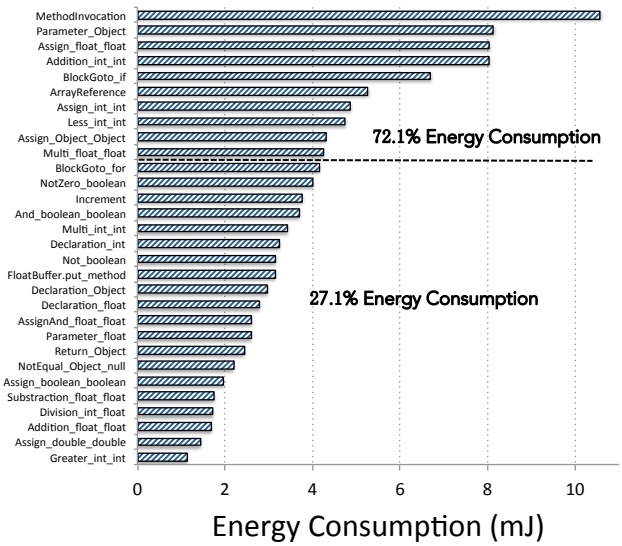


Figure 3: The top 30 energy consuming operations.

$$NMAE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{\hat{e^{(i)}} - e^{(i)}}{e^{(i)}}\right| \quad (6)$$

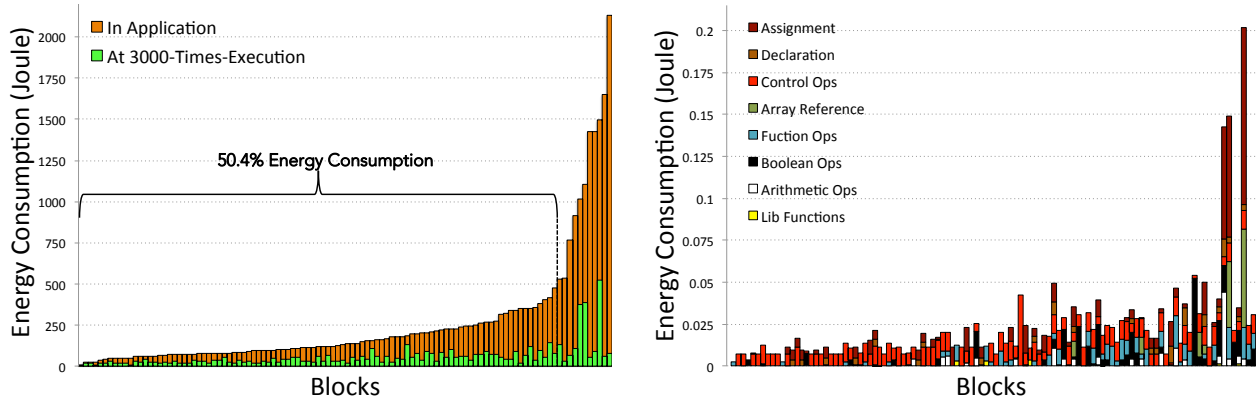
6. ENERGY ACCOUNTING

The energy model for the application source code based on energy operations facilitates comprehensive energy accounting at different levels of granularity and from various viewpoints. In this section, we will see the rank of the most expensive operations, and the contributions of different operations to the energy consumption of each block.

6.1 Operation Level

Figure 3 shows the top 30 energy consuming operations. The figure also indicates that the energy-usage of the code is largely determined by a relatively small number of operations. The 30 operations out of 187 (including library functions) take up 99.2% of the whole cost of the code, in which the top 10 consumes the major part with a percentage of 72.1%. Among all the operations (even beyond the top 30), the costs of them vary in a range of several orders of magnitude.

Usually, it is supposed that the sophisticated arithmetic operations, such as multiplications and divisions, should be the most costly. However, the result shows that *Method Invocation* ranks the highest. This is due to a sequence of complex processes to fulfill *Method Invocation*, such as storing the return address and managing the stack frame. Instance methods are always implicitly passed a "this" reference as their first parameter. It suggests a trade-off between the structure and the energy saving when writing the code. That means, in certain cases, we



(a) Block costs “In Application" and “3000-Times-Execution".

(b) Energy proportions of different kinds of operations in blocks.

Figure 4: Energy distribution on blocks and operations. Blocks are sorted by the order of their run-time energy costs "In Application".

could unpack some thin methods that are highly-invoked in the code, at the cost of losing the integrity of the structure of the code to some extent.

Two arithmetic operations, *Addition_int_int* and *Multi_float_float*, are members of the top 10. Unexpectedly, the addition is twice as expensive as the multiplication. We surmise that this is a result of their operands in the target code, as experimentally shown in [20], the energy cost of the arithmetic computation is operand-dependent.

Later in Section 6.2, we will see that assignments, comparisons and *Array Reference* play significant roles in the overall energy consumption. This is not only because they are frequently used, but also because they are costly as operations themselves, as shown in Figure 3.

Block Goto operations are expensive as well. Based on the types of conditionals and loops where "Block Goto" occurs, they are classified into *BlockGoto_if*, *BlockGoto_for* and *BlockGoto_while*. The result shows that they cost different amounts of energy as operations themselves, respectively 6.7 mJ, 4.1 mJ, 1.1 mJ. And together with *Method Invocation*, they take up 37.6% of the total energy consumption of the application.

6.2 Block Level

In the execution cases, we have 108 active blocks with a wide diversity of energy usage. As shown in Figure 4a, "In Application" means running the `Click & Move` scenario with the full set of blocks. The costs of blocks "In Application" are plotted as orange bars. Note that, blocks here obviously have distinct execution times. The cost of a fixed number (3000) of executions of one block are calculated by multiplying its single-execution cost by 3000. This could help us compare the single-execution costs of different blocks. The costs of blocks "3000-Times-Execution" are plotted as green bars.

Similar to energy distribution on operations, only a small number (11 blocks) of all the blocks uses up nearly half of the entire cost, which indicates that putting efforts on optimising a small group of blocks can achieve significant energy-saving.

There are two factors that make one block costly "In Application". The first factor is a large number of executions. For example, the most costly block "In Application" (the rightmost orange bar in Figure 4a) has a large number of execution times. This block takes only 30.6 mJ for single-execution but 2128.6 Joule when running "In Application". The second factor is the energy consumption of the block itself. For example, the three prominent green bars in Figure 4a, whose single-execution costs are 201.5 mJ, 146.9 mJ and 142.8 mJ. We will later zoom in these three blocks to see which operations contribute to their energy costs.

We can further observe the energy proportions of operations in each block in Figure 4b. To illustrate, operations are grouped into eight classes. Specifically, the "Block Goto" operations and *Method Invocation* are gathered in *Control Ops*; the parameter passing and the value returns of methods are in *Function Ops*; the comparisons and Booleans are in *Boolean Ops*; all the arithmetic computations are in *Arithmetic Ops*; all the library functions are in *Lib Functions*.

Most of the blocks cost less than 25 mJ for single-execution. In these blocks, *Control Ops* occupy the major part of the energy consumption, in contrast, *Arithmetic Ops* only take a tiny proportion.

For those three most prominent blocks, assignments and *Array Reference* are the biggest energy consumers. Furthermore one of the three blocks has the largest proportion of *Arithmetic Ops* among all the blocks.

The most expensive block "In Application" consists of three even parts: *Control Ops*, *Function Ops* and *Boolean Ops*. This block is the main entrance of the game engine to draw and display frames, so its works are conditional judgments and method invocations.

7. RELATED WORK

From the hardware side, initial efforts on energy modeling research have been put on circuits-level (see the survey [17]), gate-level [15, 16] and register-transfer-level [10]. Later, research focus shifted towards high-level modelings, such as software and behavioral levels [14].

Energy modeling techniques for software start with the basic instruction level, which calculates the sum of energy consumption of basic instructions and transition overheads [6,26]. Gang et al. [22] base the model at the function-level while considering the effects of cache misses and pipeline stalls on functions. T. K. Tan et al. [25] utilize regression analysis for high-level software energy modeling.

However, the run-time context considered in the above works is unsophisticated, free from user inputs, a virtual machine and so on. Furthermore the software stack below the level that they deal with (such as the level of the basic or assembly instruction) is relatively thin.

When research is focused on the energy of mobile applications, the level of granularity of the techniques is increased as well. An important part of such efforts is the use of operating system and hardware features as predictors to estimate the energy consumption at the component, virtual machine and application level [7, 12, 21, 23, 28, 29].

Shuai et al. [9] and Ding et al. [13] propose approaches to get source line energy information. The former requires the specific energy profile of the target system, and the workload is fine-tuned. The latter utilizes advanced measurement techniques to obtain the source line energy cost.

In contrast to the above approaches, we explore the idea of identifying energy operations and constructing a fine-grained model which is able to capture energy information at a level lower than source line.

8. CONCLUSION

In this paper, we propose a fine-grained energy model for mobile application source code on the basis of energy operations. We first introduce the energy operations that are identified directly from the source code. The energy operations are employed as the basic units that constitute the overall energy consumption of the source code. We then design a wide diversity of execution cases to generate data about the operation executions and the entire energy consumption. Regression analysis is applied to use the data to estimate the energy consumption of each operation. Finally, we show that the model is capable to capture comprehensive energy features that coarse-grained models or techniques could not shed light on.

9. REFERENCES

[1] *Android Debug Bridge*. http://developer.android.com/tools/help/adb.html.

[2] *Cocos2d-Android*. https://code.google.com/p/cocos2d-android/.

[3] *Dalvik Virtual Machine*. http://source.android.com/devices/tech/dalvik/.

[4] *Report: U.S. Smartphone Penetration Now At 75 Percent*. http://marketingland.com/report-us-smartphone-penetration-now-75-percent-117746, 2015.

[5] D. Bogdanas and G. Roşu. K-Java: A complete semantics of Java. *SIGPLAN Not.*, 50(1):445–456, Jan. 2015.

[6] C. Brandolese, W. Fomacian, F. Salice, and D. Sciuto. An instruction-level functionality-based energy estimation model for 32-bits microprocessors. In *Design Automation Conference, 2000. Proceedings 2000*, pages 346–350, 2000.

[7] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 335–348, New York, NY, USA, 2011. ACM.

[8] C. Edwards. Lack of software support marks the low power scorecard at DAC. In *Electronics Weekly.*, pages 15–21, June 2011.

[9] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 92–101, Piscataway, NJ, USA, 2013. IEEE Press.

[10] C.-T. Hsieh, Q. Wu, C.-S. Ding, and M. Pedram. Statistical sampling and regression analysis for RT-level power evaluation. In *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*, pages 583–588, Nov 1996.

[11] X. Jiang, P. Dutta, D. Culler, and I. Stoica. Micro power meter for energy monitoring of wireless sensor networks at scale. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 186–195, New York, NY, USA, 2007. ACM.

[12] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 39–50, New York, NY, USA, 2010. ACM.

[13] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for Android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 78–89, New York, NY, USA, 2013. ACM.

[14] E. Macii, M. Pedram, and F. Somenzi. High-level power modeling, estimation, and optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(11):1061–1079, Nov 1998.

[15] R. Marculescu, D. Marculescu, and M. Pedram. Adaptive models for input data compaction for power simulators. In *Design Automation Conference, 1997. Proceedings of the ASP-DAC '97 Asia and South Pacific*, pages 391–396, Jan 1997.

[16] F. Najm. Transition density: a new measure of activity in digital circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 12(2):310–323, Feb 1993.

[17] F. Najm. A survey of power estimation techniques in VLSI circuits. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):446–455, Dec 1994.

[18] A. Ng. *CS229 lecture notes*. http://cs229.stanford.edu/notes/cs229-notes1.pdf, 2012.

[19] Odroid. *Odroid-XUE*. http://www.hardkernel.com/main/main.php.

[20] J. Pallister, S. Kerrison, J. Morse, and K. Eder. Data dependent energy modelling: A worst case perspective. *CoRR*, abs/1505.03374, 2015.

[21] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with Eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[22] G. Qu, N. Kawabe, K. Usarni, and M. Potkonjak. Function-level power estimation methodology for microprocessors. In *Design Automation Conference, 2000. Proceedings 2000*, pages 810–813, 2000.

[23] A. Shye, B. Scholbrock, and G. Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 168–178, New York, NY, USA, 2009. ACM.

[24] Soot. *A framework for analyzing and transforming Java and Android Applications*. http://sable.github.io/soot/.

[25] T. Tan, A. Raghunathan, G. Lakshminarayana, and N. Jha. High-level software energy macro-modeling. In *Design Automation Conference, 2001. Proceedings*, pages 605–610, 2001.

[26] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, Dec 1994.

[27] T. Šimunić, L. Benini, G. De Micheli, and M. Hans. Source code optimization and profiling of energy consumption in embedded systems. In *Proceedings of the 13th International Symposium on System Synthesis*, ISSS '00, pages 193–198, Washington, DC, USA, 2000. IEEE Computer Society.

[28] C. Wang, F. Yan, Y. Guo, and X. Chen. Power estimation for mobile applications with profile-driven battery traces. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ISLPED '13, pages 120–125, Piscataway, NJ, USA, 2013. IEEE Press.

[29] L. Zhang, B. Tiwana, R. Dick, Z. Qian, Z. Mao, Z. Wang, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 105–114, Oct 2010.