An Energy-Aware Programming Approach for Mobile Application Development Guided by a Fine-Grained Energy Model

Xueliang Li   John P. Gallagher

Abstract—**Energy efficiency has a significant influence on user experience of battery-driven devices such as smartphones and tablets. It is shown that software optimization plays an important role in reducing energy consumption of system. However, in mobile devices, the conventional nature of compiler considers not only energy-efficiency but also limited memory usage and real-time response to user inputs, which largely limits the compiler's positive impact on energy-saving. As a result, the code optimization relies more on developers. In this paper, we propose an energy-aware programming approach, which is guided by an operation-based source-code-level energy model. And this approach is placed at the end of software engineering life cycle to avoid distracting developers from guaranteeing the correctness of system. The experimental result shows that our approach is able to save from 6.4% to 50.2% of the overall energy consumption depending on different scenarios.**

I. Introduction

The smartphone, the most popular mobile device, has been considered as one of the most important invention in the contemporary age. In February 2015, the penetration of smartphones was about 75% in the U.S [4]. This figure is still growing. With the improvement of hardware processing capability and software development environment, the smartphone is no longer a handset only to make phone calls, also play entertaining games, watch movie videos, browse web pages and so on. On the other hand, users are meanwhile frustrated by the limited battery capacity – applications running in parallel could easily drain a fully-charged battery within 24 hours.

Software optimization barely by the compiler achieves very little energy-saving for mobile devices since besides energy-saving the compiler on a mobile device has to think of many other important factors, such as limited memory usage and quick responses to user interactions. The Android platform, say, employs the Just-In-Time (JIT) compiler [6], also known as the dynamic compiler. Its optimization window is generally as small as one or two basic blocks in order to use less memory and quicken the delivery of performance boost. However, the small window largely restricts the space of energy-saving strategies. Eventually, the refactoring of code should rely more on developers.

Unfortunately, current software development is performed in an energy-oblivious manner. Throughout the engineering life cycle, most developers and designers are blind to the energy usage of code written by themselves. However, developers are desperate for the knowledge on energy-aware programming techniques. In the most popular software development forum StackOverFlow [40], energy-related questions are marked as favorites 3.89 more often than the average questions [34]. And among the energy-related questions, code-design-related ones are prominently more popular. Moreover, it has been estimated that energy-saving by a factor of as much as three to five could be achieved solely by software optimization [12]. To realize this, the first step is to analyze the energy accounting of source code at different levels of granularity and from different points of view.

In order to enable energy accounting of code, energy modeling of code is needed to bridge the gap between high-level source code and low-level hardware, where energy is consumed. However, traditional bottom-to-top modeling techniques [8], [36], [42], [44] 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 [11], [33], [45], [46]. However, they are only capable of obtaining energy information at a coarse level of granularity such as methods or applications. Two pieces of work [14], [20] 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 energy information on blocks or more coarse-grained units could identify the hot spots in the code, but it gives few clues about how to make changes to the code. The source line is also not an appropriate level of granularity to provide energy information. For instance, the header of `for` loop contains three segments which are *initialization*, *boolean* and *update* at the same source line, but usually have distinct numbers of executions. So the energy information about the source line of the header is not very sensible for developers.

Li et al. [21] propose a source-level energy model based on "energy operations", which is more fine-grained and able to provide more valuable information for code optimization. Compared with coarse-grained techniques, there are several advantages of the operation-based model in guiding the energy-aware programming techniques:

- The energy operations are basic units that constitute the energy consumption of entire software. Thus using the energy estimate of operations, the developers can assess the effects of code changes on energy consumption of code.
- It provides more valuable information on how to make changes. For example, the experiment shows that the "methode invocation" is the most expensive operation, suggesting that in some case we may inline some thin methods at the cost of losing the integrity of the structure of code.

In this paper, we propose an energy-aware programming approach guided by fine-grained energy model of source code. The generic procedures of the approach are as following:

- We utilize the methodology described in [21] to construct the operation-based source-code-level energy model, which is achieved by analyzing the data produced in a range of well-designed execution cases .
- The model generates energy accounting at operation and block level, which captures the energy characteristics of the code.
- We put efforts on the most costly blocks, where we refactor the code to remove, reduce or replace the expensive operations, meanwhile maintain its logical consistency with the original code.

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. We evaluate the approach in three game scenarios, and the experimental result shows that it can save energy consumption by from 6.4% to 50.2% depending on different scenarios.

In the rest of this paper, we firstly introduce the identification of energy operations in Section II. The architectural setup and the design of execution cases are detailed in Section III. We elaborate the data collection and the model construction separately in Section IV and Section V, based on which we are able to capture the energy characteristics and optimize the source code in three different scenarios, `Click & Move`, `Orbit` and `Waves`, as respectively seen in Section VI, VII and VIII.

TABLE I: 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*

II. Basic Energy Operations

There are two reasons why Li et al. [21] choose to build the source code energy model based on "energy operations". Firstly, an energy operation is "atomic", which means 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 [7], 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 I lists 14 representative operations out of a total of 120 in the experiment. 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 VI-A.

TABLE II: Examples of Library Functions

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

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 II. For instance, the functions in the class of *GL10* are responsible for graphic computing.

III. Experimental Setup

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

A. Target Device

Experimental target: we employ an Odroid-XU+E development board [30] 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.

B. 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.

C. 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 three scenarios which are `Click & Move`, `Orbit` and `Waves`.

In the `Click & Move` scenario, the sprite (the character in the game) moves to the position where the tap occurs. In the `Orbit` scenario, the sprite together with the grid background spins in the three-dimension space. In the `Waves` scenario, the sprite scales up and down, meanwhile the grid background waves like flow. In both the `Orbit` and `Waves` scenarios, the animation will restart from the starting point whenever and wherever 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.

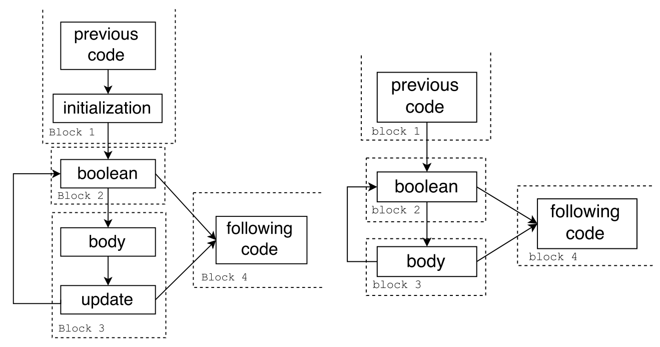
In order to obtain a more varied set of execution cases, we vary the executions of individual basic blocks in the code. This is achieved by systematically removing a set of blocks for each execution case, using the control flow graph obtained using the Soot tool [38]. We ensure that each block could be removed in some execution case. Thus an execution case is made up of one user input sequence and one set of basic blocks.

IV. 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.

A. 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.



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

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

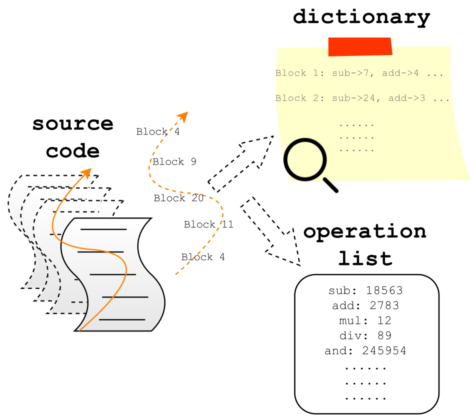


Fig. 2: The flow of the operation-execution data collection.

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 I. 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.

B. 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.

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

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

C. 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 [17], 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.

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.

V. 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 $op_i \in EnergyOps$. $EnergyOps$ 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 $func_i \in LibFuncs$. $LibFuncs$ 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.

$$E = \sum^{op_i \in EnergyOps} Cost_{op_i} \cdot N_e(op_i) \tag{2}$$
$$+ \sum^{func_i \in LibFuncs} Cost_{func_i} \cdot N_e(func_i) + Idle\,Cost$$

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 cases, acquired as shown in Section IV. 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.

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

Inevitably, the 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 [29] 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

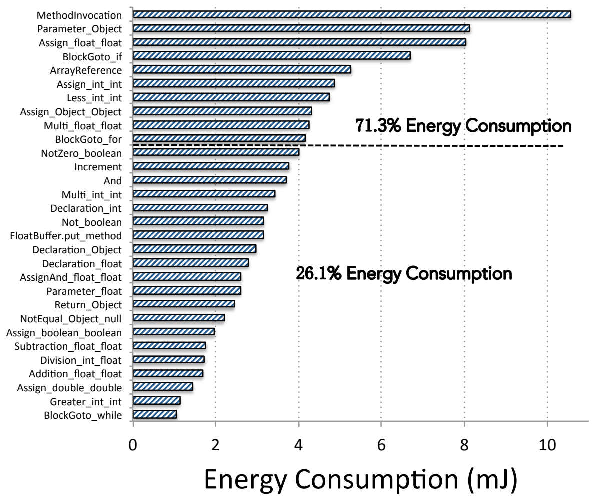


Fig. 3: The top 30 energy consuming operations in `Click & Move` scenario.

value.

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

$$cost_j := cost_j - \alpha\frac{\partial J(cost_1, ...cost_j, ...cost_l)}{\partial cost_j} \tag{5}$$
$$= cost_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(\vec{n^{(i)}} \times \vec{cost}) \cdot n_j^{(i)}$$
$$j = 1, 2, ...l$$

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 the global optima.

In the experiment, the three scenarios (`Click & Move`, `Orbit` and `Waves`) separately have their own processes of data collection and model construction since different scenarios may have different sets of parameters (costs of operations) for the model (Equation (2)). The cost of the same operation is not absolutely constant in certain cases, one of the reasons is that the values of operands influence the energy consumption of operations, as seen in [31]. Our modeling approach is trying to make a good approximation of the costs of operations for individual scenarios.

TABLE III: NMAE in Cross Validation

Scenario	Set	1st	2nd	3rd	4th
Click & Move	Training	17.7%	15.0%	13.6%	18.9%
	Validation	14.2%	14.2%	19.7%	17.8%
Orbit	Training	19.9%	17.9%	14.4%	16.8%
	Validation	11.7%	17.0%	18.0%	15.0%
Waves	Training	13.9%	14.1%	14.8%	15.0%
	Validation	16.8%	16.7%	16.1%	17.2%

To validate the reliability of model, we apply the four-round cross validation. If the model is proved to be reliable, then we use it for the energy accounting in later stages, otherwise we try other solutions to improve the model. The four-round cross validation procedure is as following: 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.

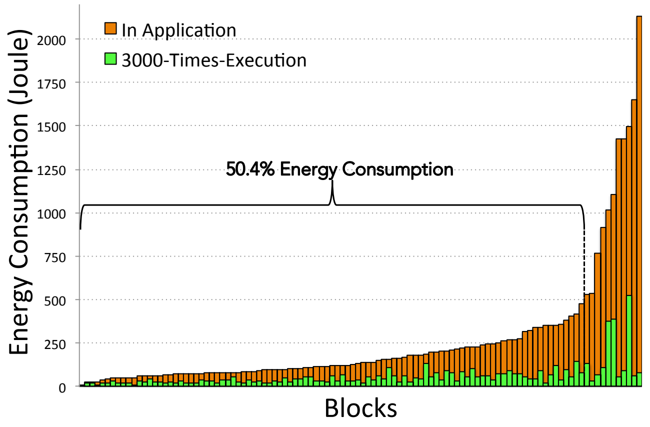
$$NMAE = \frac{1}{n}\sum_{i=1}^{n}|\frac{\hat{e^{(i)}} - e^{(i)}}{e^{(i)}}| \tag{6}$$

In Table III, we can see the Normalized Mean Absolute Error (NMAE) of the model in three scenarios 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 the three scenarios, the NMAE in training sets ranges from 13.6% to 19.9%, and in validation sets from 11.7% to 19.7%.

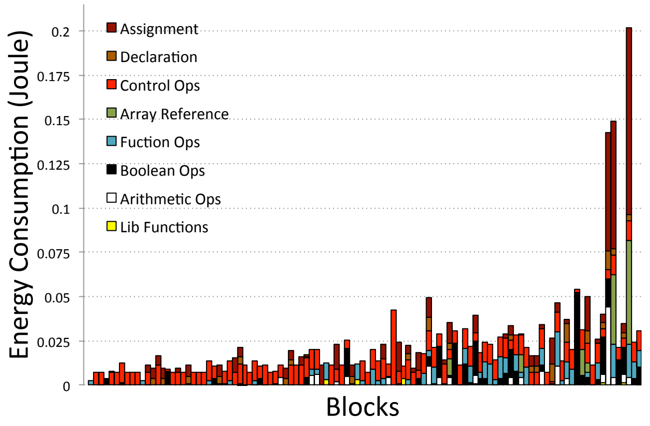
For the three scenarios, the sets of parameters respectively generated in the 2nd, 4th and 3rd rounds of cross validation are chosen to help analyze the energy property of the code in Section VI-A, because they have good balance on both training and validation sets. Their NMAEs are around 15.0%, which means the model's inference accuracy is around 85.0%.

VI. The `Click & Move` Scenario

In this section, we detail energy accounting at operation and block level, according to which we improve the most costly blocks by removing, reducing or replacing the most expensive operations. Later in Section VII and Section VIII, when we talk



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



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

Fig. 4: Energy distribution in `Click & Move`. Blocks are sorted by the order of their run-time energy costs "In Application".

about the `Orbit` and `Waves` scenarios, we will briefly introduce the energy characteristics of the code and use larger part for the code improvements.

A. *Energy Accounting*

The energy model of app 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.

Operation Level:

Figure 3 shows the top 30 energy consuming operations, which are ranked by their single-execution energy costs. "71.3% Energy Consumption" presents the percentage of sum of costs of top 10 operations in the total cost, considering their different numbers of executions in the `Click & Move` scenario. "26.1% Energy Consumption" means the percentage of operations from 11th to 30th. The percentages indicate that the energy-usage of the code is largely determined by a relatively small number of operations. It is because these operations are frequently used and meanwhile expensive themselves. The 30 operations out of 187 (including library functions) take up 97.4% of the whole cost of the code, in which the top 10 consumes the major part with a percentage of 71.3%.

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 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.

Unexpectedly, only one arithmetic operation, *Multi_float_float*, is a member of the top 10. And there are only six arithmetic operations in the top 30. They together cost only 6.1% of the overall energy consumption of the application, which is contrary to our instincts.

Later in block-level energy accounting, 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.

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" here 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 at "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.

B. Code Optimization

The most important consideration of app developers is to guarantee the correctness of software, which should then be followed by energy-efficiency. So our energy-aware programming approach is applied at the end of software engineering life circle when the software system is roughly complete.

The overview of energy-aware programming approach is firstly finding the most costly blocks, where we analyze the energy breakdown among the operations, and make changes to the code to remove, reduce or replace the costly operations.

TABLE IV: The top 10 costly blocks in `Click & Move`.

Block ID	#Executions	Energy Cost (J)
CCNode.visit()	19462	2128.6
CCNode.transform()	18903	1648.4
CCTextureAtlas.putVertex()	2119	1494.4
CCNode.visit().if_4.for_1	16880	1426.8
CCNode.transform().if_1	19664	1426.3
CCTextureAtlas.putTexCoords()	2120	1107.8
CCAtlas.updateValues().for_1	2173	1018.7
CCNode.visit().if_3.for_1	8356	915.7
CCSprite.draw()	8594	766.9
CCTexture2D.name()	13085	537.5

We look into the top 10 costly blocks "In Application" (see Table IV). For example, *CCNode.visit()* is the entrance block of the *visit()* function; *CCNode.visit().if_4.for_1* is the body block of the `for` loop. These 10 blocks are distributed in seven methods, so the code review does not require heavy labor. We find four easy optimization opportunities in blocks, such as *CCNode.visit()*, *CCNode.visit().if_4.for_1* and *CCTexture2D.name()*. There are also other opportunities in other blocks supposed to save energy, but requiring more efforts and gaining little. For example, *CCAtlas.updateValues().for_1* has several busy arithmetic expressions. Usually it is believed that replacing the busy expression with an variable could reduce energy cost, however in this case the overhead of variable declaration counteracts the energy-saving.

The four opportunities to improve the code are very simple and effective, but can only be discovered by the operation-level energy information. The changes will be shown as following.

Program 1 Simplified parts of **original** code in *CCNode.visit()*

```
if (children_ != null) {
    if_body1;
}
draw(gl);
if (children_ != null) {
    if_body2;
}
```

Program 2 The changed Program 1

```
if (children_ != null) {
    if_body1;
    draw(gl);
    if_body2;
} else {draw(gl);}
```

If Combination:

This change is made in the most costly block *CCNode.visit()*, which has two comparisons, two Boolean operations, one *Method Invocation* and one parameter passing. In fact, the two `if` headers make the same comparison, as shown in Program 1. We change the code to Program 2, which combines the two `if` statements and meanwhile keep it logically consistent with Program 1. By the means each execution of the block can reduce one comparison, and when the condition is false, it can additionally reduce one *BlockGoto_if* .

Program 3 Simplified parts of **original** code in *CCNode* class

```
public void visit(GL10 gl) {
      ......
   transform(gl);
      ......
}
public void transform(GL10 gl) {
   tranform_body;
}
```

Program 4 The changed Program 3

```
public void visit(GL10 gl) {
      ......
   transform_body;
      ......
}
public void transform(GL10 gl) {
   transform_body;
}
```

Inner-Class Method Inline:
When "In Application", the *transform()* function is invoked 18903 times and mostly by *visit()* function. We change the Program 3 to Program 4 by inserting the body of *transform()* into *visit()*, meanwhile remaining the original *transform()* function in case that other parts of the code call it. This change can largely decrease the number of *transform()*'s *Method Invocations* that are very expensive. However, it may be at the cost of losing readability of the code, which could also be compensated by adding explanatory comments.

Program 5 The full version of Program 2

```
if (children_ != null) {
for (int i=0; i<children_.size(); ++i) {
   CCNode child = children_.get(i);
   if (child.zOrder_ < 0) {
       child.visit(gl);
   } else
      break;
}
draw(gl);
for (int i=0; i<children_.size(); ++i) {
   CCNode child = children_.get(i);
   if (child.zOrder_ >= 0) {
      child.visit(gl);
   }
}
} else {draw(gl);}
```

Loop-Invariant Code Motion:
CCNode.visit().if_3.for_1 and *CCNode.visit().if_4.for_1* are entrance blocks of the two `for` loops as seen in Program 5. These two loops have a quantity, *children_.size()*, which is computed in each iteration but the value is constant. We thus hoist it outside the loop, as shown in Program 6, which can vastly save the energy of invoking and executing the *size()* function during every iteration. At the same time, we move the declaration of the *child* outside the loop, considering the cost of *Declaration_Object* is about 2.97 mJ and also in the top 30.

Program 6 The changed Program 5

```
CCNode child = new CCNode(); //added
int children_size = children_.size(); //added
if (children_ != null) {
   for (int i=0; i<children_size; ++i) { //changed
      child = children_.get(i); //changed
      if (child.zOrder_ < 0) {
         child.visit(gl);
      } else
          break;
   }
draw(gl);
for (int i=0; i<children_size; ++i) { //changed
   child = children_.get(i); //changed
   if (child.zOrder_ >= 0) {
      child.visit(gl);
   }
}
} else {draw(gl);}
```

Inter-Class Method Inline:
CCTexture2D.name() is the 10th costly block and costs 537.5 Joule "In Application". However, its job is to simply get the value of the private member variable, *_name*, of the class *CCTexture2D*. And this method has only two callers in the code. So we consider to make this variable public and let the two callers directly get access to the variable, which avoids the cost of *Method Invocation*. This change may harm the encapsulation of data, however, only one member of one class is changed. The trade-off between energy-saving and data encapsulation will be at last decided by developers.

C. Evaluation

Figure 5 illustrates the energy consumption of the software without and with the changes introduced in the previous section. From left to right, the bars indicate accumulative effects of the changes. For example, "+ *If Comn*" is the energy consumption of the code with "If Combination"; "+ *Inner-Class MI*" is the energy consumption of the code with the changes of both "If Combination" and "Inner-Class Method Inline". Totally, these four simple changes save 6.4% of the entire energy consumption without influencing the functionality of code. These changes are made in the basic part of the game engine, where most applications will base on, so any gain here can have fundamental impact. Furthermore, these changes are made with little knowledge about the algorithm of code, the developers who wrote the code are surely able to improve the code much more and achieve more energy-saving.

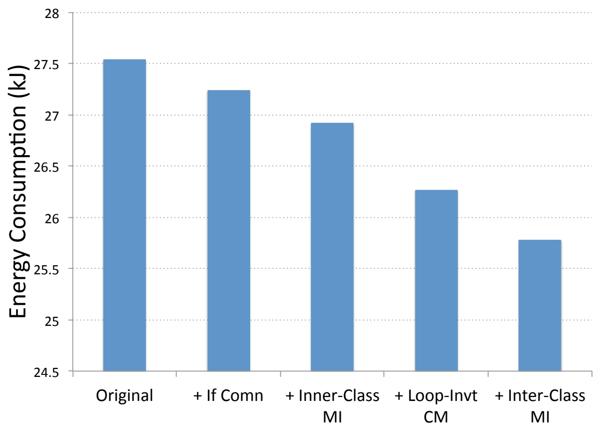


Fig. 5: Energy consumption of the code without and with the changes in `Click & Move`.

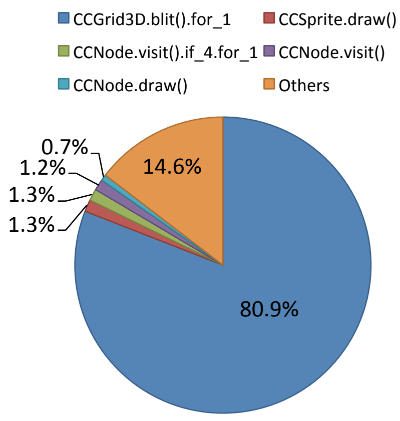


Fig. 6: In `Orbit` scenario, the energy proportions of blocks "In Application".

VII. The `Orbit` Scenario

In this section, we briefly introduce the energy characteristics of `Orbit` scenario. Afterward, we improve the most costly blocks according to the types of expensive operations. In Section VII-C, we can see that the improvement can save as much as 50.2% of the overall energy consumption.

A. Energy Accounting

In the `Orbit` scenario, the block *CCGrid3d.blit().for_1* dominates the overall energy consumption. As shown in Figure 6, 80.9% of the entire cost is consumed by this block. The second costly block consumes only 1.3%. "In Application" here means running the `Orbit` scenario without removing any block. Later in Section VII-B, we will barely put attention on this single block, requiring fairly little effort to achieve improvements.

B. Code Optimization

Program 7 shows the original code of *CCGrid3D.blit().for_1*. In this block, the *Control Ops* (*BlockGoto_for* and *Field Reference*) use up 35.6% energy; *Boolean Ops* use up 20.5%; the assignments use up 16.7%; *Arithmetic Ops* use up 14.0%; *Lib Functions* use up 13.3%. We find three easy changes to reduce or replace the expensive operations.

Loop-Invariant Code Motion:
In this block, the value of *vertices.limit()* is constantly 2112, we thus hoist it outside the loop and replace it with the variable *limit*, as shown in Program 8. This change avoids calls of *vertices.limit()* and at the same time decreases a small amount of *Field Reference*.

Loop Unrolling:
Also as shown in Program 8, we duplicate the loop body eight times, which reduces the times of comparisons, *BlockGoto_for*s, assignments and additions. Note that, we set the value of increment as 24 since 24 is a factor of the *limit*, 2112.

Full-Use of Library Function:
The job of Program 7 or Program 8 is getting all the elements in *vertices* one by one and putting them into *mVertexBuffer* one by one. The whole Program 7 in fact can be replaced by simply one line: *mVertexBuffer.put(vertices.asReadOnlyBuffer())*, which means putting the entire *vertices* into *mVerteBuffer*. This change realizes the same functionality using the already existing library function, which is one of the key library functions already compiled into native code.

Program 7 The **original** code of *CCGrid3D.blit().for_1*

```
for (int i = 0; i < vertices.limit(); i=i+3) {
   mVertexBuffer.put(vertices.get(i));
    mVertexBuffer.put(vertices.get(i+1));
    mVertexBuffer.put(vertices.get(i+2));
}
```

Program 8 The changed Program 7

```
int limit = vertices.limit(); //added
for (int i = 0; i < limit; i=i+24) { //changed
    mVertexBuffer.put(vertices.get(i));
    mVertexBuffer.put(vertices.get(i+1));
    mVertexBuffer.put(vertices.get(i+2));
                  ...
                  ...
    mVertexBuffer.put(vertices.get(i+23)); //added
}
```

C. Evaluation

Figure 7 shows the accumulative effects of the code changes on energy consumption. Exceptionally, "*Full-Use LF*" does not take previous changes into account and means only replacing Program 7 with the built-in library function as stated above.

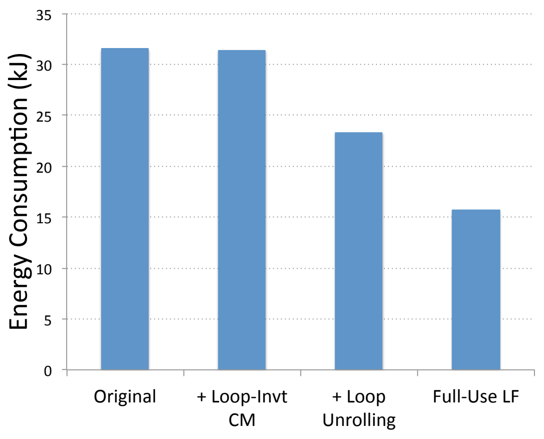


Fig. 7: Energy consumption of the code without and with the changes in `Orbit`.

We can see that loop-invariant code motion does not gain much energy saving because the *vertices.limit()* which is a library function as well uses a very small percentage of energy consumption. On the other hand, loop unrolling achieves 25.8% energy saving due to the reduction of amount of *Control Ops*, comparisons and assignments, which occupy most of the cost. And the most effective change is the replacement to a library function, saving 50.2% energy consumption because this library function has been complied into native code before execution, in contrast the java source code need runtime interpretation which is not free from energy consumption. The result indicates that it is a good idea for developers to make a good use of library functions rather than implementing the same function themselves with java source code.

VIII. The `Waves` Scenario

In this section, similarly, we first analyze the energy characteristics of the blocks in the `Waves` scenario, based on which we modify the code and then evaluate the effects of changes on energy consumption.

A. Energy Accounting

Unlike the `Orbit` scenario where only one block dominates energy consumption, in `Waves` scenario, the costs of top seven blocks are at the same order of magnitude of kJ, as listed in Table V. The *CCGrid3D.blit().for_1* is also employed in this scenario and is the most costly as well among all the blocks. The majority of blocks in Table V are directly or indirectly invoked by *CCWaves3D.update().for_1.for_1*, as shown in Program 9. And their jobs are mostly to set or get the values of member variables, so a large part of energy consumption goes to assignments and *Function Ops*. It was not expected that the code spends such a large amount of energy on simple setter and getter functions.

Program 9 The **original** code in *CCWaves3D.update()*

```
int i, j;
for( i = 0; i < (gridSize.x+1); i++ ) {
   for( j = 0; j <(gridSize.y+1); j++ ) {
      CCVertex3D v = originalVertex(ccGridSize.ccg(i,j));
                     ...
      setVertex(ccGridSize.ccg(i,j), v);
   }
}
```

Program 10 Program 9 after Method Inline & Code Motion

```
ccGridSize ccgridsize = new ccGridSize(0,0); //added
CCGrid3D ccgrid3d = (CCGrid3D) target.getGrid(); //added
CCVertex3D v = new CCVertex3D(0,0,0); //added
int i, j;
for( i = 0; i < (gridSize.x+1); i++ ) {
   for( j = 0; j <(gridSize.y+1); j++ ) {
   ccgridsize.x=i;ccgridsize.y=j; //added
   v = ccgrid3d.originalVertex(ccgridsize); //changed
                 ...
   ccgrid3d.setVertex(ccgridsize, v); //changed
   }
}
```

B. Code Optimization

Full-Use of Library Function:
We have talked about the optimization for *CCGrid3D.blit().for_1* in Section VII-B where we replace the entire Program 7 with the one-line code, which makes use of library functions. We keep this change in this scenario. For other blocks, we come up with one modification as following.

Method Inline & Code Motion:
As shown in Program 9, the three functions called in the inner loop body are *CCGrid3DAction.originalVertex()*, *ccGridSize.ccg()* and *CCGrid3DAction.setVertex()*, which respectively cost 2891.3 Joule, 3769.1 Joule and 3285.4 Joule "In Application". Note that, *CCGrid3DAction* is the parent class of *CCWaves3D*, so *originalVertext()* and *setVertex()* can be directly called without referring to their class names. As seen in Program 10, we unpack these three methods in this block: the first and fourth "added" lines are unpacked *ccGridSize.ccg()*; the second "added" and first "changed" lines are unpacked *CCGrid3DAction.originalVertex()*; the second "added" and second "changed" lines are unpacked *CCGrid3DAction.setVertex()*. This change removes all the *Method Invocations*, parameter passing and value returns related to these three functions invoked by this block. Note that, the first three "added" lines are located outside the loop in order to reduce energy consumption of the process of initializing objects and calling *CCNode.getGrid()*.

TABLE V: In `Waves` scenario, the top 10 costly blocks "In Application". And the energy percentages of different kinds of operations in each block.

Block ID	#Executions	Energy Cost (J)	Assi.	Decl.	Cont.	Func.	Bool.	Arit.	Libr.
CCGrid3D.blit().for_1	112193	8094.1	16.7%	0%	35.6%	0%	20.5%	14.0%	13.3%
CCVertex3D.CCVertex3D()	40219	5232.0	27.2%	0%	10.0%	62.8%	0%	0%	0%
CCWaves3D.update().for_1.for_1	34604	4088.7	10.7%	0%	32.1%	0%	14.7%	39.0%	2.2%
ccGridSize.ccg()	42275	3769.1	0%	0%	32.1%	67.9%	0%	0%	0%
CCGrid3DAction.setVertex()	31856	3285.4	14.6%	7.8%	30.9%	46.7%	0%	0%	0%
CCGrid3DAction.originalVertex()	36566	2891.3	19.1%	10.2%	40.3%	30.4%	0%	0%	0%
CCNode.getGrid()	49119	2145.1	0%	0%	58.1%	41.9%	0%	0%	0%
ccGridSize.ccGridSize()	10570	1173.8	30.3%	0%	31.6%	38.0%	0%	0%	0%
CCGrid3D.setVertex()	3944	657.2	10.1%	1.6%	32.8%	28.9%	0%	26.4%	0.2%
CCGrid3D.originalVertex()	2785	374.2	14.0%	1.9%	33.4%	17.9%	0%	32.8%	0%

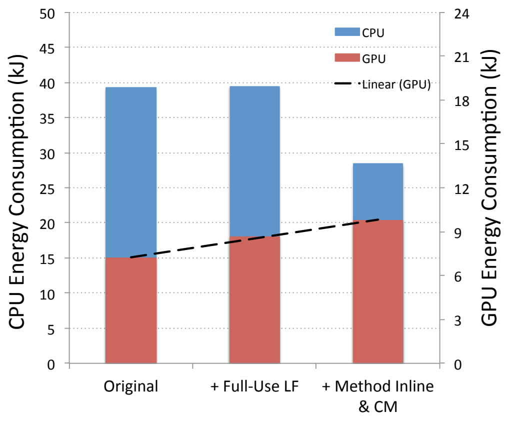


Fig. 8: CPU and GPU Energy consumption of the code without and with the changes in `Waves`.

C. Evaluation

Figure 8 shows the accumulative effects of changes on energy consumption of CPU and GPU (note that previous figures only showed the CPU energy consumption because the GPU energy consumption did not vary noticeably before), and the dashed line indicates the linear trend of the GPU energy consumption. In the case of game, usually the aimed frames per second (FPS) is 60 Hz, when the game overloads CPU, the FPS will decrease, and when the workload is light, even very light, the FPS is generally fixed to 60Hz. The FPS in "*Original*" is around 36Hz; that in "+ *Full-use LF*" is around 50Hz; that in "+ *Method Inline & CM*" is around 60Hz. The change of *Full-Use LF* (full use of library function) does not save energy for CPU since the execution of original `Waves` actually overloads the CPU capacity, so the improvement of code enhances the performance and enables the device to generate more frames every second. Consequently, the GPU does more work and consumes more energy, as seen in Figure 8. After this change, when we apply the method inline and code motion, 27.7% of the overall CPU energy is saved, and for the same reason GPU consumes slightly more. This experimental result shows that our approach not only saves energy but also potentially boosts performance, which benefits users doubly.

IX. Related Work

Energy Modeling:

From the hardware side, initial efforts on energy modeling research have been put on circuits-level (see the survey [28]), gate-level [26], [27] and register-transfer-level [15]. Later, research focus shifted towards high-level modelings, such as software and behavioral levels [25].

Energy modeling techniques for software start with the basic instruction level, which calculates the sum of energy consumption of basic instructions and transition overheads [8], [42]. Gang et al. [36] base the model at the function-level while considering the effects of cache misses and pipeline stalls on functions. T. K. Tan et al. [41] 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, dynamic compilation 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 [11], [18], [33], [37], [45], [46].

Shuai et al. [14] and Ding et al. [20] 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.

Compared with approaches above, Li et al. [21] explore the idea of identifying energy operations and constructing a fine-grained model based on operations which is able to capture energy information at a level more fine-grained than source line.

Energy-Saving Techniques:

A large amount of research efforts on energy-saving for mobile devices have been put on the main hardware components, such as the CPU, display and network interface. The CPU low-power-design techniques involve dynamic voltage-frequency scaling [22] and heterogeneous architecture [13], [23]. Techniques for display contain dynamically dimming the back-light [9], [32], tone-mapping based back light scaling [5], [16]. The network-related techniques try to exploit idle and deep sleep opportunities [24], [39], shape the traffic [10], [35] and so on.

There are many pieces of work relevant to code refactor for energy-saving . Vetro' et al. [43] define the concept of energy code smells that are the code patterns (such as self assignment, repeated conditionals and useless control flow) maybe energy-consuming. However, the code patterns selected in [43] have very little impact (less than 1.0%) on energy consumption. Our experimental result shows that our approach is able to save half of the entire energy consumption in certain scenario.

Ding et al. [19] conducted a small scale evaluation of several commonly suggested programming practices that may reduce energy. Its result shows that reading array length, accessing class field and method invocation all cost noticeable energy. However, this work only provides a small number of suggestions to developers on how to make the code more energy-efficient.

Compared with previous work, our research propose a systematic energy-aware programming approach, which is guided by the operation-based source-code-level energy model. The experimental result shows that this approach is an effective guideline for energy-aware mobile application development.

X. Conclusion

In this paper, we propose an energy-aware programming approach for mobile app development, which is guided by the operation-based source-code-level energy model. The general steps of the approach are as following: 1) we construct the operation-based energy model by mining the data generated in a range of well-designed execution cases; 2) based on the model, we capture the energy characteristics of the code; 3) we improve the code by removing, reducing or replacing the expensive operations in the costly blocks.

We evaluate this approach on a real-world game engine and on a physical Android development board with two ARM quad-core CPUs. The experimental result shows that our approach has a significantly positive impact on energy-saving. For different scenarios, this approach can save energy by from 6.4% to 50.2%. The result also indicates that the performance of code is a byproduct as well of this approach, which potentially improves user experience more.

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] B. Anand, K. Thirugnanam, J. Sebastian, P. G. Kannan, A. L. Ananda, M. C. Chan, and R. K. Balan. Adaptive display power management for mobile games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 57–70, New York, NY, USA, 2011. ACM.

[6] Android. *A JIT Compiler for Android's Dalvik VM*. http://www.android-app-developer.co.uk/android-app-development-docs/android-jit-compiler-androids-dalvik-vm.pdf.

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

[8] 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.

[9] W.-C. Cheng and M. Pedram. Power minimization in a backlit tftlcd display by concurrent brightness and contrast scaling. *Consumer Electronics, IEEE Transactions on*, 50(1):25–32, Feb 2004.

[10] C. Chiasserini and R. Rao. Improving battery performance by using traffic shaping techniques. *Selected Areas in Communications, IEEE Journal on*, 19(7):1385–1394, Jul 2001.

[11] 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.

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

[13] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The greendroid mobile application processor: An architecture for silicon's dark future. *Micro, IEEE*, 31(2):86–95, March 2011.

[14] 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.

[15] 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.

[16] A. Iranli and M. Pedram. Dtm: Dynamic tone mapping for backlight scaling. In *Proceedings of the 42Nd Annual Design Automation Conference*, DAC '05, pages 612–617, New York, NY, USA, 2005. ACM.

[17] 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.

[18] 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.

[19] D. Li and W. G. J. Halfond. An investigation into energy-saving programming practices for android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, GREENS 2014, pages 46–53, New York, NY, USA, 2014. ACM.

[20] 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.

[21] X. Li and J. P. Gallagher. A top-to-bottom view: Energy analysis for mobile application source code. *CoRR*, abs/1510.04165, 2015.

[22] X. Li, G. Yan, Y. Han, and X. Li. Smartcap: User experience-oriented power adaptation for smartphone's application processor. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 57–60, San Jose, CA, USA, 2013. EDA Consortium.

[23] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: Using low-power processors in smartphones without knowing them. *SIGPLAN Not.*, 47(4):13–24, Mar. 2012.

[24] J. Liu and L. Zhong. Micro power management of active 802.11 interfaces. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 146–159, New York, NY, USA, 2008. ACM.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

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

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

[32] S. Pasricha, M. Luthra, S. Mohapatra, N. Dutt, and N. Venkatasubramanian. Dynamic backlight adaptation for low-power handheld devices. *IEEE Design Test of Computers*, 21(5):398–405, 2004.

[33] 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.

[34] G. Pinto, F. Castor, and Y. D. Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 22–31, New York, NY, USA, 2014. ACM.

[35] C. Poellabauer and K. Schwan. Energy-aware traffic shaping for wireless real-time applications. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 48–55, May 2004.

[36] 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.

[37] 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.

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

[39] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: Hierarchical power management for mobile devices. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 261–274, New York, NY, USA, 2005. ACM.

[40] STACKOVERFLOW. http://stackoverflow.com.

[41] 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.

[42] 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.

[43] P. G. M. M. Vetro' A., Ardito L. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. In *ENERGY 2013 : The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 34–39, March 2013.

[44] 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.

[45] 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.

[46] 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.