Static Program Analysis for Identifying Energy Bugs in Graphics-Intensive Mobile Apps

Chang Hwan Peter Kim, Daniel Kroening, and Marta Kwiatkowska
Department of Computer Science, University of Oxford, UK
chpkim@gmail.com, {kroening, marta.kwiatkowska}@cs.ox.ac.uk

***Abstract*—A major drawback of mobile devices is limited battery life. Apps that use graphics are especially energy greedy and developers must invest significant effort to make such apps energy efficient. We propose a novel static optimization technique for eliminating drawing commands to produce energy-efficient apps. The key insight we exploit is that the static analysis is able to predict future behavior of the app, and we give three exemplars that demonstrate the value of this approach. Firstly, *loop invariant texture analysis* identifies repetitive texture transfers in the render loop so that they can be moved out of the loop and performed just once. Secondly, *packing* identifies images that are drawn together and therefore can be combined into a larger image to eliminate overhead associated with multiple smaller images. Finally, *identical frames detection* uses a combination of static and dynamic analysis to identify frames that are identical to the previous frame and therefore do not have to be drawn. We implemented the technique against LibGDX, an Android game engine, and evaluated it using open source projects. Our experiments indicate savings up to 44% of the total energy consumption of the device.**

I. Introduction

Mobile devices are ubiquitous: in 2014, over 85% of the computers sold were either smartphones or tablets and that number is expected to increase in the coming years [1]. As the popularity of mobile devices grows, the complexity of their apps grows as well. Moreover, as apps become more complex, they require more computing power, which is especially problematic for mobile devices whose batteries are limited. Most smartphones are equipped with battery-saving features, which typically restrict either software, hardware or a combination of both. Mainstream magazine articles written in 2015 provide tips for improving smartphone battery performance [2], [3]. These tips are largely based on sacrificing non-essentials, such as reducing the frequency of notifications and lowering backlight brightness.

But what if the apps themselves are not energy efficient? Users will avoid using the apps or replace them with more energy-efficient alternatives. Thus there is a strong incentive to develop energy-efficient apps. Yet, a recent study of over 100 programmers shows that programmers have limited knowledge about energy efficiency [4] and how to avoid so called *energy bugs* [5].

A number of papers have been published that aim to help developers produce energy-efficient apps. Examples include a framework for supporting energy-conscious programming using controlled approximation [6] and so called loop perforation [7], a general technique to trade accuracy for performance by transforming loops to execute a subset of their iterations. Dynamic analysis tools such as `Carat` can gather smartphone usage data from a community of users and provide analysis and recommendations on energy consumption [8]. Another strand is work that employs static analysis to check proper API usage, which can save energy [9], [10], identify energy-consuming locations in source code [11], [12], and change an app's color scheme to be more energy efficient [13].

However, none of the techniques mentioned above consider *graphics-intensive* apps, which can arguably benefit the most from energy optimization. A recent study shows that graphics-related API usage accounts for nearly 40% of energy consumption for 55 mobile apps from different domains [14]. For graphics-intensive apps such as games this number is likely to be much higher, as the apps spend most of the execution time in the *render loop*, the loop that draws the graphics. Most graphics-intensive apps rely on *hardware acceleration*, i.e., use the Graphics Processing Unit (GPU) rather than the general-purpose CPU to draw graphics faster. While hardware acceleration can boost performance, it can be used suboptimally, leading to lost performance.

In this paper, we present and evaluate a method for optimizing energy efficiency of graphics-intensive apps, which focuses on GPU operations. Rather than relying on runtime analysis, which explores individual executions, we provide a static analysis method that enables analyzing program code based on future use of data (such as definition-use chains) and can be largely automated. More specifically, we introduce a suite of program analyses that help eliminate unnecessary GPU usage. This can reduce energy consumption of a graphics-intensive app considerably, not only by reducing the amount of work done by the GPU, but also by reducing the amount of data transfer to the GPU. To this end, we propose three techniques to address the following energy bugs. Firstly, *loop invariant texture analysis* statically identifies texture transfers done repeatedly in the render loop, so that it can be done outside of the render loop just once. Secondly, *packing* statically identifies groups of images that may be drawn together, such that each group's images can be "packed" into a larger image that is transferred over to the GPU together, which saves the overhead of transferring and preparing multiple smaller images for draw. Images from different groups can never be displayed together, so they should not be packed together. Lastly, *identical frame detection* uses a combination of static and dynamic analysis to efficiently identify frames that are identical to previous frames and therefore do not have to be drawn.

To summarize, the contributions of this paper are:

- A novel suite of program analyses to reduce energy consumption for graphics-intensive apps.
- An implementation of the analyses using the Soot static analysis framework [15] and an application to OpenGL ES Android apps developed on top of the LibGDX game engine.
- Experiments that show that our analysis noticeably reduces energy consumption without sacrificing performance.

The techniques proposed in this paper use a combination of intraprocedural and interprocedural analyses and require some manual intervention. They are intended to provide feedback to the programmer in order to obtain better (energy-wise) performance. The methods are not limited to the LibGDX game engine and are applicable more generally to visualization software.

II. Preliminaries

We assume familiarity with compiler optimization and static analysis techniques, as described, for example, in [16]. A survey of the most relevant research papers in static analysis can be found in [17].

A. Graphics API

OpenGL is a language-independent and platform-independent API for rendering vector graphics, which uses geometrical primitives such as points, lines and shapes to represent images [18]. It is typically used for rendering graphics using a GPU. *OpenGL ES* is the embedded systems version of OpenGL. From an application developer's point of view, OpenGL can seem fairly low-level, so several frameworks, such as game engines, have been developed on top of OpenGL ES to facilitate its proper and optimal usage. *LibGDX* is such a framework that provides a unified API that works across both desktop and mobile platforms, including Windows, Linux, Android, iOS and Javascript.

B. Graphics-intensive Apps

We motivate the problem using *Freegemas* [19], an open-source LibGDX version of the popular game *Bejewelled*. The player swaps adjacent jewels to form a horizontal or vertical chain of three or more gems of the same color. Figure 1 gives a simplified version of the game's code. A LibGDX game can consist of multiple `Screens`. `Screen`'s `render()` method is called back by the library when the screen needs to be displayed. For our purposes, the control-flow graph can simply be thought of as the screen constructor being called once and then the render method on the constructed screen being called in an infinite loop (lines 23–29). Anytime the code constructs a texture or texture region (texture region just refers to an area within a texture), there is data transfer from CPU to GPU and the GPU performs work, which takes time and consumes energy.

```
1  class GameScreen extends Screen {
2    void render(float delta) {
3      TextureRegion background = new TextureRegion
4        (new Texture(Gdx.getFile("background.png")));
5      draw(background, 0, 0);
6
7      for(Cell cell: cells) {
8        if (state == InitialAnimation)
9          cell.updateY();
10       draw(cell.gem, cell.x, cell.y);
11     }
12
13     background.getTexture().dispose();
14   }
15 }
16
17 class Cell {
18   void updateY() {
19     y = y + 2;
20   }
21 }
22
23 class Game  {
24   static void main(String args[]) {
25     Screen screen = getCurrentScreen();
26     while(true)
27       screen.render();
28   }
29 }
```

Fig. 1. Simplified Freegemas Code

C. Energy Bugs

The example has a number of *energy bugs*. For starters, note that background is constructed in *every* iteration of the render loop, i.e. every *frame*, and disposed (lines 3, 4, 5, 13), when the construction can be done outside of the render loop just once. The render loop can iterate up to 80 times per second and, in each iteration, an image is constructed and transferred from CPU to GPU, which not only increases energy consumption but can also decrease frame rate if the image is large. Unfortunately, LibGDX or any library/framework cannot identify this bug since the bug requires knowledge about future execution, which can only be obtained by statically analyzing the application code.

Before a texture on the GPU can be drawn, the texture must first be *bound*. Since binding is expensive, it is ideal to group or pack many smaller textures into a larger texture, bind the larger texture once, then draw regions of it multiple times. LibGDX can pack textures, but the developers must specify the textures to pack, which leaves room for inefficiency. For example, the developer can pack two images that will never be drawn together. In Figure 1, it is possible for the developer to pack textures in `GameScreen` and menu screen (not shown) together, even though they can never be displayed together (since two screens cannot be displayed at the same time). Only a static analysis can be used to determine which images can never be drawn together, which requires knowledge about future execution.

Consider a board game like the running example or any game that spends a long time waiting for user input. In such cases, the screen should not be redrawn if it does not change, as doing so would waste energy. By default, LibGDX's rendering thread calls the `render()` method continuously, meaning

the same screen can be redrawn up to 80 times per second. A simple solution would be for the library to record drawing commands issued in one execution (frame) of `render()`, and compare the list to that in the next execution of the method to see if they are identical, in which case no drawing command would have to be sent to the GPU. Unfortunately, this is expensive and can even reduce the frame rate. Also, this naive approach does not take advantage of the fact that the appropriate monitoring depends heavily on the application. For instance, applications with continuous animations, such as a moving background, do not benefit from monitoring at all. Unfortunately, because a library is oblivious to the application code, it cannot tailor monitoring to the application.

The code can be improved in several ways by using the LibGDX API, but this must be done manually by the developer and is therefore error prone and time consuming. The improvement is also not something that can be made automatically by *any* framework because it requires analysis of the application code, which the framework does not have access to. The only way these improvements can be automated is through *static analysis*.

III. Methodology

Since a graphics-intensive app renders at a high frequency, energy optimization must incur a small or no overhead at runtime. Also, as mentioned previously, energy optimization often requires knowledge about future execution. These reasons are what make static analysis-based energy optimization highly appropriate. We present a suite of static program analyses that address the three types of energy bugs introduced in the previous section:

1) *Loop invariant texture analysis*. Transferring data from CPU to GPU consumes a considerable amount of energy. Instead of monitoring, which has a runtime overhead and cannot determine what textures will be displayed in the next loop iteration, we present an analysis that can identify loop invariant textures. This enables the programmer to hoist them out of the loop.
2) *Packing*. We propose a method to identify which images can/cannot be drawn together so that images are packed into a larger one to reduce texture bindings.
3) *Identical frames detection*. It is possible to save on drawing if the current frame is identical to the previous one. The proposed method employs static analysis to identify statements that are involved in an animation. Once such statements are reached during execution, monitoring can be stopped. In the best case, monitoring can be eliminated completely.

We implemented our static analyses using Soot [15] static analysis framework. Soot converts Android application binary file into Jimple, a three-address intermediate representation against which our static analyses are applied. Our static analyses use a combination of interprocedural and intraprocedural analyses, and specifically points-to analysis, for which we use *SPARK*, which is interprocedural, context-insensitive, flow-insensitive and path-insensitive. We configured Soot to use this points-to analysis for construction of an interprocedural call graph. The call graph does not include edges in Android and LibGDX library code because it would be prohibitively large (orders of magnitudes larger). Our static analyses rely on off-the-shelf analyses provided in Soot, including intraprocedural reaching definitions and reaching uses analyses.

Our static analyses are not fully automatic and instead aim to provide feedback to the programmer to obtain a more energy efficient app. This is justified as follows. Firstly, and most importantly, two of our three static analyses can produce false positives that must be manually eliminated by the user. Since we are in the domain of performance optimization, we could have made the analysis automatic by only reporting cases that can for sure be optimized. But we felt that the benefit of identifying potentially many more optimization opportunities outweighed the downside of having to manually verify them. False negatives are also possible with our static analyses, meaning we can miss optimizations, which is not critical (e.g. compared to missing security flaws). Despite allowing false positives and negatives, our technique is useful for real graphics-intensive apps. Secondly, at the time of writing, at least for our subject LibGDX apps, Soot was not able to produce working Android binary application file from Jimple (even without changes to the Jimple code).

We believe that the automated assistance that our technique provides is a considerable improvement over the state of the art approach of manual inspection and purely dynamic instrumentation, and can be extended to include more sophisticated analyses. We discuss each static analysis in turn.

IV. Loop Invariant Texture Analysis

Hoisting, or loop invariant code motion, is a classic compiler optimization that moves computations that are identical across loop iterations outside of the loop so that they are performed just once. The technique can be adapted to identify redundant texture transfers. Unlike conventional hoisting, which attempts to identify program computations that are loop invariant, our technique targets large data transfers between CPU and GPU, which is much more likely to result in noticeable energy savings. Also, since data transfer to the GPU is a side effect, existing loop invariant techniques, as they are, would not identify the texture transfer as being loop invariant.

Our analysis takes a `Screen`'s `render()` method and returns a set of loop invariant draws. The aim of the algorithm is to identify texture construction calls that are only dependent on data that are 1) written to outside the control-flow of the render method or 2) written to in the control-flow of the render method but the writes are all consumed by the end of the method. This goal can be achieved to varying degrees of precision. The algorithm we present identifies texture construction calls that are not transitively data dependent on application class fields. Resulting textures cannot be persisted across calls to `render` (loop iterations in our setting) since the dependent values, which are locals, will expire when the method in which the construction is done returns. We do not check control dependencies of construction calls because we also want to

identify construction calls that happen conditionally. Texture construction should rarely be done in the control-flow of the `render` method (lazy construction being the exception). While such construction calls will not be hoistable, they can still be manually refactored to be done once. Therefore, our analysis identifies potential loop-invariant texture constructions but the user must manually hoist them.

Algorithm 1 gives the details of the static analysis. In each method reachable (transitively called) from the `render` method, statements with texture construction are checked for loop invariance. A statement is not loop invariant if it is a call to a method that is not *trusted*. A trusted method is a method we have manually determined to not affect invariance (such as a library function used for object construction and draw calls).[1] A field, such as a library static field, can be trusted as well. Each variable reference in a statement is checked. If the reference is a parameter reference or an untrusted field reference, the statement is not invariant. Otherwise, if the reference is not trusted, then its reaching definitions and statements that use it (if it is a reference to the variable being defined in the statement) are checked for invariance if they have not been already.

Let us apply the algorithm against the running example. As the variables involved in `background` construction are locals that use locals and are only used by locals and trusted methods (`Texture` constructor, `TextureRegion` constructor, `Gdx.getFile()`, `draw()`, `getTexture()` and `dispose()`), the texture construction is loop invariant. Being a static analysis method, our algorithm can miss texture constructions that are actually invariant, e.g., if `background` were a field. To handle this case, the analysis of data and control dependencies would have to be performed across methods, rather than just at the construction site's method. Since the aim is performance optimization, missed opportunities are acceptable. Because our algorithm does not check for control dependencies, it can yield false positives as well. However, a drawing command can be performed repeatedly within a conditional, and we believe that sacrificing precision is worth the opportunity to catch these cases (although they have to be checked by the user manually). In fact, even a simpler analysis that checks that texture construction is not done in the control flow of the `render` method would be useful.

Algorithm 1: Loop Invariant Texture Analysis

```
1  global visited, invariants;
2  function findInvariants(renderMethod)
3      foreach m ∈ reachableMethods(renderMethod) do
4          foreach texture construction t ∈ statements(m) do
5              if isInvariant(t) then
6                  invariants ← invariants ∪ t;
7  function isInvariant(stmt)
8      visited ← visited ∪ stmt;
9      if isUntrustedMethodCall(stmt) then
10         return false;
11     stmts ← ∅
12     foreach v ∈ variableReferences(stmt) do
13         if isParamRef(v) or isUntrustedFieldRef(v) then
14             return false;
15         else if not trusted(v) then
16             stmts ← stmts ∪ reachingDefs(v) ∪ uses(v);
17     foreach stmt ∈ stmts do
18         if stmt ∉ visited then
19             if not isInvariant(stmt) then
20                 return false;
21     return true;
```

V. Packing

We present a static analysis method that identifies which images may be drawn together and, consequently, which images are never drawn together in an app. This information can help the user in packing images into a larger image to reduce texture binding. For each screen in the app, which can be identified as classes that implement the `Screen` library interface, methods reachable from screen's constructor are collected. Then, in each reachable method, we look for image file names. Image file names collected for a screen are grouped. Since two screens can never be displayed together, images from different screens should not be packed together. Algorithm 2 gives the pseudo-code for this analysis.

Algorithm 2: Packing Analysis

```
1  function findTextureFiles(screenClass)
2      textures ← ∅
3      foreach m ∈ reachableMethods(constructor(screenClass)) do
4          foreach statements ∈ statements(m) do
5              foreach string constant c ∈ constants(s) do
6                  if isTextureFileName(c) then
7                      textures ← textures ∪ c;
8      return textures;
```

Recall our example given in Figure 1. For this example, the static analysis will place the background, gems, and the help image into one group since they are used in `GameScreen`. Suppose we have another screen, called `MenuScreen`. That

[1]Note that, while using manual classification into trusted and untrusted methods may seem like "cheating", manual classification is necessary because such methods can have side effects. For example, texture constructors transfer an image from the CPU to GPU. It is because we know the semantics of this side effect (i.e., transferring the image multiple times is semantically equivalent to transferring it just once) from domain knowledge that we can consider these statements to be loop invariant.

screen's images will be placed into a separate group.

Note that our static analysis can be improved in a number of ways. For example, since an image file may not be used for texture construction (but this is unlikely), we can actually do def-use analysis to only pick out those images used for texture construction. Also, grouping can be done at a more fine-grained level. Suppose that images are always drawn together – they should always be grouped together. But since it is possible for images to be drawn together most of the time, but not all the time, it would be too restrictive to require a group to only contain images always drawn together. Also, our technique currently cannot check if an existing atlas (which is the result of packing) has been packed in an optimal way – the static analysis result must be used prior to packing. We leave these improvements for future work.

VI. Identical Frames Detection

The basic idea behind identifying identical frames using monitoring is to record screen commands (e.g., draw, clear screen) occurring in the control-flow of the `render` method and, if the list is identical to the previous list of commands, then the current frame is considered identical to the previous frame and therefore does not need to be drawn. We improve on this basic idea through a static analysis that can reduce monitoring by identifying statements that perform animation. Once such a statement is encountered, monitoring can be stopped for the current frame. In the best case, monitoring can be eliminated completely.

Due to its simplicity, we do not give details of the monitor. We assume that the screen's state after a frame has been drawn can be completely captured using a list of commands. In addition, we record changes to any object, such as a texture, that is part of a command that persists across frames and can have its state changed across frames. Comparing frames can also be done by comparing both commands and object state (deep value comparison of objects). We assume that objects that are arguments to screen commands do not have their state changed across frames.

We call a statement that performs an animation an *animation statement*. We manually studied code across apps that performs animation and found that animations typically write to a variable using the variable's value from the previous frame. For example, coordinates and stills of an animation will be updated using their previous values. Based on this observation, we identify an animation by a write to a field that gets its value from the same field. We ignore locals since they cannot persist across frames in our setting (since each frame corresponds to a new render call).

Algorithm 3 gives the details of our static analysis. For each method that is reachable from the `render` method, each definition statement in the method is checked to determine whether it is an animation statement. The definition statement must write to a field. For each variable reference on the right-hand side of the statement, we compute the definition-use chains. Then for each field read in the chains, we check to see if that field is the same as the field written. The algorithm

Algorithm 3: Identical Frame Detection

```
1 function isAnimationStmt(def)
2     if leftOp(def) isnot FieldRef then
3         return false;
4     foreach v ∈ variableReferences(rightOp(def)) do
5         duChains ← defUseChains(v);
6         foreach r ∈ fieldReads(duChains) do
7             if field(r) = field(leftOp(def)) then
8                 return true;
```

can be implemented to varying degrees of precision. Our definition-use chains are intraprocedural, meaning the field write and read must occur in the same method. Also, we do not perform points-to analysis to see that the object of the field written to is the same as the object of a reaching field read. However, our implementation is able to handle cases where the field is an array and writes are to elements within that array (e.g., `field[x] = field[x]+1`).

For example, in Figure 1, `cell.y` is updated through a method reachable from the `render` method (Figure 1, line 9). Since the write to the field obtains its value from a read of the field (`y=y+2`), the write is identified as an animation statement (Figure 1, line 19).

Note that our analysis yields both false positives and negatives. A field that keeps track of time or score can be incremented but never displayed. This will be a false positive for our analysis. To address this, interprocedural dataflow analysis could be used, focused on only analyzing field writes that flow into a draw statement. We leave this improvement for future work. Our technique can yield false negatives also. For example, field writes connected to reads across functions will not be picked up but, from what we have seen, fields involved in animations typically are updated in the same function.

Our technique can be further improved in several ways. Firstly, the implementation could be extended to analyze library code. Secondly, an animation statement dominated by another can be eliminated. Animation statements can be moved up to the earliest point in program execution, from which we know the statements are guaranteed to be reached. Draw commands dominated by animation statements do not have to be instrumented. Thus, in the best case, the monitor can be entirely eliminated statically. Our current implementation just identifies the animation statements and, immediately before each animation statement, a call that stops recording is inserted.

VII. Evaluation

Our evaluation addresses the following research questions:

- Can the proposed static analyses detect energy bugs and enable optimizations that would not be possible by the LibGDX framework or any other runtime approach?
- What is the quantitative effect of these bugs on the energy consumption of the device?

A. Implementation Details

We implemented our static analyses using Soot [15]. Our analyses build on off-the-shelf analyses provided by Soot, including intraprocedural reaching-definitions and reaching-uses analyses. We used the SPARK points-to analysis, which is context-insensitive, flow-insensitive and path-insensitive. Soot uses the points-to analysis to construct an interprocedural call graph. As mentioned before, the call graph does not include edges in Android and LibGDX library code because it would be prohibitively large. Unfortunately, without library code, an app's call-back methods are excluded from the call graph since they are not called from anywhere. We manually identified the call-back methods and created an artificial main method that calls them, which is similar to the approach in [20]. Alternatively, an approach that automatically discovers call-back methods could have been used [21].

The Android application binary file is converted into Jimple, a three-address intermediate representation used by Soot, against which analysis and transformation can be applied. Jimple could be converted back to an Android application binary file. Unfortunately, at the time of writing, at least for our subject LibGDX apps, the binary files produced by Soot crash (even without any transformation applied to the Jimple code). We therefore had to apply the results of our static analyses against the source code of the subjects manually. Also, due to Soot's problem with transforming Android bytecode, we used AspectJ to implement the monitor needed for identical frame detection. The aspect intercepts calls to LibGDX using an around advice, which prevents the calls from being sent to the GPU, records them, and only sends them to the GPU if the list is different from the previous frame's list of commands. We went through each animation statement and manually inserted a call to stop the monitor.

B. Subjects

We experimentally evaluated our techniques against three open-source Android apps written against the LibGDX framework by measuring the reduction of their power consumption, but the methods are more generally applicable. As explained earlier, `Freegemas` is a game where the player swaps jewels to form rows or columns with the same type of jewel [19]. It has 3328 lines of Java code. `Wari` is a board game where each player has a set of seeds and takes turns moving seeds, with the objective of capturing more seeds than the opponent [22]. It is a variation of the ancient game called *Oware*.[2] It has 1183 lines of Java code. `Zxzx` is a space invader-like game, where the player ship trades shots against a larger enemy ship [23]. It has 24687 lines of Java code. We chose these subjects because they exhibit different degrees of graphics usage. As a stationary board game, Wari uses few images and does not use many animations. Also, there is likely to be a lot of idle time between user inputs as it is a strategy game based on counting. Although Freegemas is also a board game, the user is likely to swap gems faster than making a move in Wari, and Freegemas involves more animations, such as those triggered by swapping gems. Zxzx is the most graphics-intensive, with background moving and bullets being fired constantly.

[2]Wari's logic in determining the winner is buggy but this does not affect our results as we were able to exercise the main game play.

C. Experimental Setup

Our static analyses were run on an Intel Core i5-3320M CPU at 2.60 GHz, with 8 GB RAM and 64-bit Windows 7. The JVM on which the static analyses were run was given 2048 KB initial and maximum heap size. Each of the three static analyses ran in under two minutes in all cases. The subjects were executed on a Samsung Galaxy S5 SM-G900F smartphone running Android 4.4.2. We used a Monsoon Power Monitor [24] to measure energy consumption of the smartphone using a USB connection between the smartphone and the monitor.

The basic idea of the experiment is to run a subject (play the game) for a period of time and measure the energy consumption. Then we apply the transformation identified by the static analysis to the game, run the game in an identical way and measure the energy consumption. Each run lasted 25 seconds. The user input sequence, when present, was replayed using a touch replay tool called `FRep` [25]. We found that the readings can vary, e.g., depending on how active the phone has been. To minimize variations, before each run, the energy consumption of the smartphone in standby for 10 seconds was ensured to be between 135 and 150 μAh (this was achieved by leaving the device in standby at least for several minutes before each run). Also, each run was executed several times and each energy measurement we report is the average over these executions. The smartphone's wifi and wireless connection and background processes were disabled. To ensure that games could be replayed consistently, random number generators were changed to have a constant seed. For Zxzx, we also made the enemy spaceship stationary. Although we were not able to completely remove randomness of bullets from spaceships in Zxzx, this did not make a noticeable difference in energy consumption over the multiple executions of each run. Table I shows the results. We now discuss each section of the table in turn.

D. Loop Invariant Texture Analysis

For each app, we consider three runs: a run of the app ("Original"), a run of the app with a "small energy bug" ("Worst", as this is the worst case for our analysis) and a run of the app with a "large energy bug" ("Best"). We were not able to find apps with repeated texture constructions, so we introduced them into the apps and checked if the LibGDX framework could detect them. The framework was not able to detect them. The "small bug" involves creating a `Texture` object for a small image used in the game, which transfers the texture from CPU to GPU, and deleting the texture after drawing, in each iteration of the render loop. For Freegemas, the small image is 2 KB large. For Wari, the small image is 271 bytes large. For Zxzx, the small image is 2 KB large. The "large bug" does the same but for a large image, such as a

TABLE I
EXPERIMENTAL RESULTS

Loop Invariant Texture Analysis								
	Original		Worst			Best		
Subject	*Energy* (µAh)	*FPS*	*Energy* (µAh)		*FPS*	*Energy* (µAh)		*FPS*
Freegemas	1656	60	1696	(2.4%)	60	1894	(14.0%)	14
Wari	1315	60	1482	(12.0%)	60	1899	(44.0%)	58
Zxzx	1407	60	1496	(6.3%)	53	1905	(35.0%)	45
Packing								
	Unpacked		Packed			Wastefully Packed		
Subject	*Energy* (µAh)	*FPS*	*Energy* (µAh)		*FPS*	*Energy* (µAh)		*FPS*
Freegemas	1656	60	1395	(−15.7%)	60	1396	(−15.7%)	60
Wari	1295	60	1315	(1.5%)	60	1294	(0.1%)	60
Zxzx	1584	60	1376	(−13.1%)	60	1407	(−11.1%)	60
Identical Frame Detection – No Input								
	Original		Runtime Only			Static		
Subject	*Energy* (µAh)	*FPS*	*Energy* (µAh)		*FPS*	*Energy* (µAh)		*FPS*
Freegemas	1656	60	1459	(−11.0%)	57	1740	(5.0%)	57
Wari	1315	60	1229	(−6.5%)	59	1204	(−8.4%)	59
Zxzx	1407	60	1810	(28.0%)	49	1568	(11.0%)	54
Identical Frame Detection – Input								
	Original		Runtime Only			Static		
Subject	*Energy* (µAh)	*FPS*	*Energy* (µAh)		*FPS*	*Energy* (µAh)		*FPS*
Freegemas	1626	60	1571	(−3.4%)	54	1849	(13.7%)	58
Wari	1421	60	1419	(−0.1%)	59	1438	(1.2%)	59
Zxzx	1619	60	1620	(0.1%)	49	1548	(−4.3%)	54

background image, used in the game. For Freegemas, the large image is 229 KB large. For Wari, the large image is 203 KB large. For Zxzx, the large image is 141 KB large. Our static analysis was able to identify the bugs. In each run, the subject's game screen is started and left on, without user input, for a period of time (25 seconds as mentioned before). We do not consider user input as repeated drawing and transfer to GPU of the same image can occur regardless of user input. For each run, the energy consumption of the smartphone is reported in µAh (we also give the percentage increase compared to "Original"). Finally, we report the number of Frames Per Second ("FPS"), as this number is important for determining whether animations are smooth.

The small energy bug does not affect Freegemas' performance, but it does affect Wari's energy consumption and Zxzx's energy consumption as well as its FPS. The large energy bug affects all apps considerably, making Freegemas and Zxzx basically unusable due to their low frame rates. The energy consumption of Wari increases by 44%. It could be argued that the drop in FPS might lead the developer to an energy bug without the help of static analysis. However, our static analysis can find the bug more directly and without executing the app. Moreover, not all energy bugs cause a considerable drop in FPS. The drop in Wari's FPS is likely to go unnoticed because its animations are simple. This demonstrates the usefulness of our static analysis.

E. Packing

For each app, we want to compare three versions: 1) one where images are not packed ("Unpacked"), 2) one where images are packed ("Packed"), and 3) one where images are packed with images from another screen, which is wasteful ("Wastefully Packed"). There is no user input as we want to isolate the effect of packing on drawing. Without knowing which images are displayed in a screen, the developer may implement cases 1) or 3). Our analysis helps the developer avoid these mistakes by identifying the images displayed in the control-flow of the render method of each screen. We took each app and created these three versions. For 3), we introduced another screen whose images should not have been packed with the game screen's images. For Freegemas, the game screen's texture atlas is 85 KB large when packed and 473 KB large when packed wastefully. For Wari, the game screen's texture atlas is 216 KB large when packed and 273 KB large when packed wastefully. For Zxzx, the game screen's texture atlas is 148 KB large when packed and 473 KB large when packed wastefully.

The results show that for Freegemas and Zxzx, packing, wasteful or not, saves between 11–15% of energy consumption. But there is no difference between packing and packing wastefully. The reason for this seems to be that, while wasteful packing transfers a larger texture to the GPU, the GPU has enough memory to store the larger texture, meaning there is no noticeable effect on energy consumption. However, wasteful packing does take extra memory so it should still be avoided.

F. Identical Frames Detection

In Table I, we consider two scenarios for the detection of identical frames, one with user input and one without. As user input triggers the screen to change in all the apps considered, in which case frames will no longer be identical, it is the more difficult scenario. Unlike experiments for hoisting and packing, we do not inject bugs, but rather try to reduce energy consumption of an app as is.

Let us discuss the best-case scenario, no user input, first. The reference point is in the column labeled "Original".

In "Runtime Only", a purely dynamic monitor reduces FPS for all apps. This could be due to an inefficient implementation of the monitoring, but even an optimal monitor will incur overhead as every drawing command needs to be intercepted, recorded and compared. Nevertheless, the monitor does reduce energy consumption for Freegemas and Wari, as there are many identical frames in these apps when there is no user input. However, note that the monitor *increases* energy consumption by 28% for Zxzx, a heavily animated game. Also, the drop in FPS is far larger than that for the other apps. The column labeled "Static" gives the results for incorporating our static analysis. Oddly, energy consumption for Freegemas actually increases by 5%. We analyzed Freegemas' code and it turns out that the time that is displayed is updated every frame, rather than every second, which would be more optimal. The update was identified as an animation statement by our static analysis. The monitor only caused additional overhead, which explains why the energy consumption increased. The static analysis does not do much for Wari, but, for Zxzx, it was able to identify the animation statement responsible for the moving background in Zxzx, which was able to reduce its energy overhead to 11% from 28% and improve FPS to 54 from 49.

Let us now discuss the more difficult scenario, with user input. For Freegemas and Wari, the screen was touched every few seconds, triggering an animation that lasted a couple of seconds. For Zxzx, a finger was on the screen all the time and moved constantly to dodge bullets from the enemy spaceship. Any saving in energy consumption that we saw without user input is reduced, as expected. Static analysis for Freegemas causes higher energy consumption for the same reason as for the "no input" case. For Zxzx, static analysis does not reduce energy consumption as much because the baseline also consumes more energy, but it still fares better than runtime only. Also, the static analysis still improves the monitor's FPS considerably.

While our static analysis should ultimately be evaluated based on its ability to reduce energy consumption, just as important is its ability to correctly identify animations. We analyzed each animation statement and determined whether it was actually part of code that performs animation. For Freegemas, 13 of 17 (76.5%) animation statements were involved in animation. The precision was 2 of 3 (66.6%) for Wari and 22 of 41 (53.6%) for Zxzx. The false positives were due to updates to program data such as timers, scores and levels. A more sophisticated analysis that tracks interprocedural data flow into draw statements is needed to eliminate these false positives. We unfortunately were not able to determine recall, which would have required manually analyzing each app exhaustively to find animation statements. But we do know that there are false negatives. For example, Wari uses a third party library for some animations, which our static analysis is not able to identify.

G. Threats to Validity

The main threat is to external validity. We explicitly chose apps with varying degrees of graphics usage and energy bugs of varying degrees of impact to illustrate the range of quantitative benefit our technique can bring. However, we cannot generalize our results to all graphics-intensive apps, frameworks, use cases and energy bugs, since our case studies may not be representative. Our study has the usual internal and construct threats to validity.

VIII. Related Work

A. Energy-efficiency Frameworks

A number of programming guidelines and frameworks have been proposed to achieve energy efficiency, including energy-conscious programming using controlled approximation [6] and program transformation of [26]. Accuracy can be traded off for performance by transforming loops to execute a subset of their iterations by means of a technique known as loop perforation [7].

B. Energy Bugs

The term *energy bugs* for mobile devices was first introduced in [5]. The paper presents a taxonomy of such bugs based on mining online user forums and OS bug repositories. Repeated transfer of images to GPU and displaying identical frames can be considered a *loop bug*, which is part of their taxonomy. They mention the possibility of using symbolic execution to detect loop bugs, which is similar to using static analysis, but they do not present any technique. In [27], a technique is presented to predict workload-dependent loops that impact UI responsiveness. Although this work also looks at performance problems related to loops, the technique is concerned with identifying loops whose performance varies by input size, whereas we aim to remove redundancy within the rendering loop, which lasts for the duration of the app. The runtime testing technique `TODDLER` reports tests whose loops perform computations that are similar across iterations, which can be used to identify performance bugs [28].

Static analyses for quantitative properties of software are rare. The most common static analyses of this kind consider worst-case execution time of real-time software (e.g. [29], [30], [31]), but quantitative properties of probabilistic software have also been analysed [32]. Energy is typically not considered at the code level, though there are frameworks that support software product lines [33] and cardiac pacemaker models [34]; with the exception of [34] none consider real energy measurements. In [35], a static analysis is presented for hoisting loop invariant data structures or objects, which is technically similar to the loop invariant technique we presented. Our setting and aims are different, however. We target graphics-intensive mobile apps with the goal of saving energy consumption. To this end, we only look for method calls that are known to be energy greedy, such as texture constructions, rather than any loop-invariant data structures. Also, our technique targets method calls that would conventionally not be considered loop invariant because they have side effects to

the GPU. Also, we present the packing and identical frame detection optimizations.

C. Energy Estimation

Techniques exist for mapping energy consumption to program entities, such as methods [36] and source code locations [11], and even runtime artifacts such as paths [37]. These can be used to detect bugs where a code portion spends more energy than other portions. Since the problems we tackle, i.e., repeated image transfers, inefficiently packed images, and identical frames, need not necessarily consume more energy than other areas of the app, energy profiling does not seem to be appropriate.

D. Runtime Analysis

There are works that rely on data obtained at runtime to find energy bugs. We tackle problems that are best addressed using knowledge about future execution, which only a static analysis can provide. Regardless, we compare our work to some dynamic analyses. `TODDLER`, mentioned above, is a dynamic analysis technique. `Carat` gathers smartphone usage data from a community of users and provides analysis and recommendations on energy consumption [8]. Since it works at the usage level, rather than code, it is not applicable to our problem. In [38], tests are generated for finding energy bugs. Using a known baseline energy consumption of the device in idle state, failing tests can be identified. Anomaly detection is used to detect energy hotspots. The technique, like other testing techniques, needs to know what is correct. For our problems, this would require knowing what the right amount of energy consumption is for an app without repeated image transfers, unoptimized packing, and identical frames, which would be hard to obtain. `GreenDroid` uses dynamic taint analysis to identify underusage of sensors [39], which is an orthogonal problem to ours.

E. Static Analysis for Mobile Apps

A mobile app can acquire a *wakelock* to keep the phone awake, but it should release it eventually, to prevent energy waste. A static analysis based on reaching definitions can identify cases where a wakelock is never released [9]. Our paper tackles a different problem requiring a different solution. `PerfChecker` provides static analyses to identify list views that do not reuse recycled items and heavyweight calls in main UI thread (which affects responsiveness) [10]. While the idea of applying static analyses aimed at improving graphics is similar to our technique, our technique targets graphics-intensive apps (e.g. games as opposed to apps with largely static screens) and therefore requires more specialized techniques. Also, we quantitatively demonstrate the energy consumption savings that our technique can provide, while their work does not. `Nyx` uses static analysis to replace the large, light-colored background areas of web applications with dark colors to reduce the energy consumed by OLED screens [13]. By contrast, our technique leaves screen output unchanged.

F. Compiler Optimization

The presented suite of static analysis techniques can be seen as a high-level version of classic compiler optimizations. We developed a loop invariant analysis to identify repeated texture constructions. Our analysis for packing relies largely on control-flow reachability. Our identical frame detection analysis is similar to induction variable analysis [40] in that we are looking for statements of the form `x=x+c`. Unlike induction variable analysis, which requires figuring out how the variable is updated (be it for strength reduction or understanding data structure traversal), our analysis only cares that the variable's value will be different from the previous frame's value. Also, whereas compiler optimizations have traditionally been used for optimizing speed of execution, our compiler-like optimizations are for optimizing energy consumption of graphics-intensive mobile apps.

IX. Conclusion

Graphics-intensive apps such as games rely on GPUs for drawing, which can provide a huge performance benefit but at the same time leaves a great deal of room for developer mistakes. While a library or a framework can help prevent mistakes, there are code issues that require knowledge of future execution, which makes static analysis a necessity. In this paper, we presented three such problems and the corresponding static analyses that for the first time tackle graphics-intensive apps: loop invariant texture analysis that identifies repeated texture transfers, packing that identifies images used in the control-flow of a screen, and identical frame detection that identifies redundant frames that do not have to be drawn.

We have designed these methods so that they provide feedback to the programmer, identifying potentially energy-wasteful operations. Our approach is novel in that it provides static analyses that are tailored for energy optimization of graphics-intensive apps. We implemented the analyses using the Soot static analysis framework, targeting Android apps written using the LibGDX game engine. Evaluation on three open source projects using real energy measurements obtained using a power monitor shows that our loop invariant texture analysis can save up to 44% in energy consumption and prevent a drop in frame rate, packing can save up to 15% in energy consumption, and identical frame detection can save up to 11% in energy consumption while surpassing a purely runtime approach in display performance.

There are several directions for future work aimed at improving precision of the static analysis. One possibility is to increase the precision of the static analyses. Another option is to increase automation, for example by employing techniques based on [21], which automatically discovers callback methods. Finally, developing generic energy optimization techniques for GPUs would be a worthwhile direction.

Acknowledgment

This work has been supported by the ERC Advanced Grant VERIWARE and by the ERC Consolidator Grant CPROVER.

REFERENCES

[1] International Data Corporation, "As tablets slow and PCs face ongoing challenges, smartphones grab an ever-larger share of the smart connected device market through 2019, according to IDC," http://www.idc.com/getdoc.jsp?containerId=prUS25500515.

[2] CNN, "The apps that drain your phone's battery the most," http://money.cnn.com/2015/06/03/technology/battery-draining-apps/.

[3] TIME, "15 tricks for getting way better smartphone battery life," http://time.com/3820202/better-smartphone-battery-life/.

[4] C. Pang, A. Hindle, B. Adams, and A. Hassan, "What do programmers know about software energy consumption?" *Software, IEEE*, vol. PP, no. 99, pp. 1–1, 2015.

[5] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices," in *ACM Workshop on Hot Topics in Networks*, ser. HotNets-X. ACM, 2011, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2070562.2070567

[6] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *Programming Language Design and Implementation (PLDI)*, B. G. Zorn and A. Aiken, Eds. ACM, 2010, pp. 198–209. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806620

[7] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Foundations of Software Engineering (FSE)*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 124–134. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025133

[8] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma, "Carat: Collaborative energy diagnosis for mobile devices," in *Embedded Networked Sensor Systems*, ser. SenSys. ACM, 2013, pp. 10:1–10:14. [Online]. Available: http://doi.acm.org/10.1145/2517351.2517354

[9] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps," in *Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2012, pp. 267–280. [Online]. Available: http://doi.acm.org/10.1145/2307636.2307661

[10] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 1013–1024. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568229

[11] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan, "Calculating source line level energy information for Android applications," in *International Symposium on Software Testing and Analysis (ISSTA)*, M. Pezzè and M. Harman, Eds. ACM, 2013, pp. 78–89. [Online]. Available: http://doi.acm.org/10.1145/2483760.2483780

[12] M. Wan, Y. Jin, D. Li, and W. Halfond, "Detecting display energy hotspots in Android apps," in *Software Testing, Verification and Validation (ICST)*. IEEE, April 2015, pp. 1–10.

[13] D. Li, A. H. Tran, and W. G. J. Halfond, "Making web applications more energy efficient for OLED smartphones," in *International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 527–538. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568321

[14] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy API usage patterns in Android apps: An empirical study," in *Mining Software Repositories*, ser. MSR. ACM, 2014, pp. 2–11. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597085

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

[16] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[17] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, no. 7, pp. 1165–1178, July 2008.

[18] Khronos, "The standard for embedded accelerated 3D graphics," https://www.khronos.org/opengles/.

[19] David Saltares, "Freegemas LibGDX," https://github.com/saltares/freegemas-gdx.

[20] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Programming Language Design and Implementation*, ser. PLDI. ACM, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[21] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework," in *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2015.

[22] FrodCube, "WariLibgdx," https://github.com/FrodCube/WariLibgdx.

[23] Tim Su, "Zxzx," https://github.com/timsu/zxzx/.

[24] Monsoon Solutions, Inc., "Power Monitor," https://www.msoon.com/LabEquipment/PowerMonitor/.

[25] strAI, "FRep – finger replayer," https://play.google.com/store/apps/details?id=com.x0.strai.frep&hl=en.

[26] S. Misailovic, D. M. Roy, and M. C. Rinard, "Probabilistically accurate program transformations," in *International Conference on Static Analysis*, ser. SAS. Springer, 2011, pp. 316–333. [Online]. Available: http://dl.acm.org/citation.cfm?id=2041552.2041576

[27] X. Xiao, S. Han, D. Zhang, and T. Xie, "Context-sensitive delta inference for identifying workload-dependent performance bottlenecks," in *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2013, pp. 90–100. [Online]. Available: http://doi.acm.org/10.1145/2483760.2483784

[28] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 562–571.

[29] I. Wenzel, R. Kirner, B. Rieder, and P. P. Puschner, "Measurement-based timing analysis," in *Leveraging Applications of Formal Methods, Verification and Validation*, ser. Communications in Computer and Information Science, vol. 17. Springer, 2008, pp. 430–444. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88479-8_30

[30] S. Chattopadhyay and A. Roychoudhury, "Scalable and precise refinement of cache timing analysis via model checking," in *Real-Time Systems Symposium (RTSS)*. IEEE, 2011, pp. 517–562.

[31] S. A. Seshia and J. Kotker, "GameTime: A toolkit for timing analysis of software," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, March 2011, pp. 388–392.

[32] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker, "Abstraction refinement for probabilistic software," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, vol. 5403. Springer, 2009, pp. 182–197.

[33] C. Dubslaff, S. Klüppelholz, and C. Baier, "Probabilistic model checking for energy analysis in software product lines," in *International Conference on Modularity, MODULARITY*. ACM, 2014, pp. 169–180. [Online]. Available: http://doi.acm.org/10.1145/2577080.2577095

[34] B. Barbot, M. Kwiatkowska, A. Mereacre, and N. Paoletti, "Building power consumption models from executable timed I/O automata specifications," in *Hybrid Systems: Computation and Control, HSCC*. ACM, 2016, pp. 195–204. [Online]. Available: http://doi.acm.org/10.1145/2883817.2883844

[35] G. Xu, D. Yan, and A. Rountev, "Static detection of loop-invariant data structures," in *Object-Oriented Programming (ECOOP)*. Springer, 2012, pp. 738–763.

[36] 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*. ACM, 2012, pp. 29–42.

[37] T. Hönig, C. Eibel, R. Kapitza, and W. Schröder-Preikschat, "SEEP: Exploiting symbolic execution for energy-aware programming," in *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, ser. HotPower. ACM, 2011, pp. 4:1–4:5. [Online]. Available: http://doi.acm.org/10.1145/2039252.2039256

[38] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting energy bugs and hotspots in mobile apps," in *Foundations of Software Engineering*, ser. FSE. ACM, 2014, pp. 588–598. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635871

[39] Y. Liu, C. Xu, and S. Cheung, "Where has my battery gone? Finding sensor related energy black holes in smartphone applications," in *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on*, March 2013, pp. 2–10.

[40] S. Calman and J. Zhu, "Interprocedural induction variable analysis based on interprocedural SSA form IR," in *Program Analysis for Software Tools and Engineering (PASTE)*. ACM, 2010, pp. 37–44.