Making Web Applications More Energy Efficient for OLED Smartphones

Ding Li, Angelica Huyen Tran, William G. J. Halfond
Department of Computer Science
University of Southern California
Los Angeles, California, USA
{dingli, tranac, halfond}@usc.edu

ABSTRACT

A smartphone's display is one of its most energy consuming components. Modern smartphones use OLED displays that consume more energy when displaying light colors as opposed to dark colors. This is problematic as many popular mobile web applications use large light colored backgrounds. To address this problem we developed an approach for automatically rewriting web applications so that they generate more energy efficient web pages. Our approach is based on program analysis of the structure of the web application implementation. In the evaluation of our approach we show that it can achieve a 40% reduction in display power consumption. A user study indicates that the transformed web pages are acceptable to users with over 60% choosing to use the transformed pages for normal usage.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Optimization

General Terms

Performance

Keywords

Energy optimization, Web applications

1. INTRODUCTION

Smartphones provide end users with a range of sensors that can be combined with applications and data via the Internet. This makes the capabilities of smartphones almost boundless and very popular with end users. However, one of the primary limitations of smartphones is that they depend on battery power. Smartphones are energy constrained devices and the use of these capabilities is very expensive. In particular, the energy to drive a smartphone's display is one of the dominant energy consuming components in a smartphone [9].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '14, May 31 - June 7, 2014, Hyderabad, India
Copyright 14 ACM 978-1-4503-2756-5/14/05 ...$15.00.

OLED screens [36] are increasingly popular in different smartphones, such as the Samsung Galaxy, Sony Xperia, and LG Optimus series. These screens are more energy efficient than previous generation displays, but also have very different energy consumption patterns. In particular, darker colors, such as black, require less energy to display than lighter colors, such as white. Unfortunately, many popular and widely used web applications use light-colored backgrounds. This means that, for many web application, there is a significant opportunity to improve the battery life of smartphones by improving the color usage of a web application's pages.

Researchers and engineers have long recognized the need to reduce a smartphone's display energy. A well-known and widely used smartphone technique is to dim the display to conserve energy [15]. For example, when the smartphone is idle. This technique is useful, but there is room for additional improvement by exploiting the OLED screen's unique energy color relationships. One simple approach that has been suggested is to invert colors, switching light colors to dark and vice versa [14]. The primary problem with this approach is that it distorts the color relationships of the user interface because color difference is not an invertible relationship. Another approach is to create an alternate color scheme for mobile web applications. Chameleon proposes a browser extension that retrieves and applies a more energy efficient color scheme when displaying a web application [14]. The drawback of this approach is that it requires a customized browser, additional servers on the network to handle the color scheme, and the color scheme itself must be manually generated.

Given the state of the art, a technique that can automatically transform a web application to make its web pages more energy efficient is desirable. However, there are several significant challenges to providing such a solution. The first of these is to identify colors generated by a web applications. Most modern web applications combine dynamically generated pages and cascading style sheets in a way that makes it complicated to determine which colors will be used in which parts of a web page. Second, it is important to model the color relationships in the web page. Here, it is necessary to know what kind of visual relationships the colors have with each other, i.e., whether they are contained or adjacent. Third, given this information, it is challenging to find a new color scheme that maintains, as much as possible, the color differences and aesthetics of the original web page, while also being more energy efficient.

In this paper we propose a new technique for automatically transforming the color scheme of a mobile web application. The approach rewrites the server side code and templates of a web application so that the resulting web application generates pages that are more energy efficient when displayed on a smartphone. The rewritten web application can then be made available to OLED smartphone users via automatic redirection or a user-clickable link. Our approach employs program analysis to model the possible pages that can be generated by the web application. Using this information, it models the potential visual relationships among the colors of the pages' elements and defines a set of constraints for the new color scheme. Our approach then defines a minimization problem whose solution represents a new color scheme in which the color differences are similar to those in the original web application. Finally, we define an efficient simulated annealing based algorithm to solve the minimization problem and produce a new color scheme that is both energy efficient and visually appealing.

We have implemented our approach in a prototype tool, Nyx, and performed an extensive empirical evaluation on a set of seven web applications. The results of our evaluation show that our approach is successful at automatically rewriting web applications to generate more energy efficient web pages that will be acceptable to end users. In particular, our approach achieved an average 40% reduction in the display's power consumption. Via a user study, we found that users rated the attractiveness and readability of the transformed pages as lower, but still close, to the original. Importantly, over 60% of users indicated that the transformed version would be acceptable for general use given the energy savings, and over 97% said it would be acceptable for use if the battery power was critically low. Overall, we consider these results to be a strong indication that our approach can provide efficient and visually-acceptable transformations for mobile web applications.

The other parts of this paper are organized as follows: In Section 2, we use a simple example to illustrate the basic problem and provide background knowledge. Section 3 provides an overview of the main process of the approach. We introduce how to analyze relationships in the HTML output of web applications in Section 4 and how to identify color relationships and transform colors in Section 5. We discuss the rewriting of the web app in Section 6. Our empirical evaluation for the approach is presented in Section 7. Section 8 describes related work. Finally, the conclusion and future work are discussed in Section 9.

2. MOTIVATING EXAMPLE

In this section we introduce a motivating example to illustrate the challenges our approach must address. Our simple example is shown in Program 1 and its output is shown in Program 2. For explanatory purposes, we inline the CSS properties used by the code in Program 1. As mentioned in Section 1, we have three main challenges to address to automatically transform the colors of a web application.

The first challenge is to extract color information from the implementation of a web application. The color information includes two types of information, the colors generated and the structural relationship they have with other colors. For example, in Program 1, we need to know that (1) the $< body >$ (line 3) tag has white as its background color and the $< td >$ tag has red as its background color, and (2) the

```
1  public void print_html()
2  {
3   print("<body bgcolor=\"white\" style=\"color:black;\">");
4   println("<table><tr>");
5   int a=1;
6   if(a==0){
7     println("<td>hi</td>");
8   }
9   else{
10    println("<td style=\"background-color:red; color:yellow
          ;\">ha</td>")
11  }
12  for(int i=0;i<2;i++){
13    println("<td style=\"background-color:green; color:blue
          ;\">usc</td>");
14  }
15  println("</tr></table>");
16  println("</body>");
17 }
```

Program 1: Sample code of web app

```
1 <body bgcolor="white" style="color:black;">
2     <table><tr>
3         <td style="background-color:red; color:yellow;"">
              ha</td>
4       <td style="background−color:green; color:blue;">usc
            </td>
5         <td style="background−color:green; color:blue;">
              usc</td>
6     </tr></table>
7 </body>
```

Program 2: Output of Program 1

red color area is surrounded by the white color area. This information is obtained by analyzing the code and identifying the strings that define the page's HTML structure and colors. In general this requires us to model the output of a web application and then build more detailed models of the relationships among its HTML elements. We discuss how to extract color and structural information in Section 4.

The second challenge is to model the relationship between colors that have a structural relationship. In general, a transformation must maintain this relationship to improve the readability and aesthetics of a new color scheme. For modeling this relationship, we use color distance, which is a function that accepts two colors and returns a numeric value to indicate the degree of difference between the two colors [35, 34]. Colors have a larger color distance if they are more different. This modeling is complicated by the fact that there are generally multiple colors used in a web page. For example, in Program 2, we have six colors: white, black, green, yellow, red, and blue. All of these colors have different relationships: white surrounds red and green, green and red are next to each other, etc.. Furthermore, not all color relationships in a web application are equally important. For example, in Program 2, the relationship between white and black (the background color and text color in line 1) is more important than the difference between black and yellow (text color in line 1 and line 5). We address this modeling problem in Section 5.1 and Section 5.2.

The third challenge is that given a model for the relationships between colors, we must find the best color scheme that saves display energy and, at the same time, maintains the attractiveness and readability of web application. From prior research studies we know that the energy consumption of OLED screens is related to the RGB value of each

pixel [13]. Energy consumption of a pixel ranges from black, the least energy consuming color, to white, the most energy consuming color. Therefore, we would want to change the background color of the < *body* > tag in Program 2 to black to save energy. However, to maintain readability we also need to change the background colors for table cells (line 3, 4, and 6) and the text color (line 1, 3, 5, and 7). Otherwise, the content of the web may become unreadable and the appearance of the web application may be degraded. A brute force method to find a new color scheme that satisfies these constraints is inefficient because of the large color space as there are 256^3 colors in total to choose among. We address this problem in Section 5.3.

3. OVERVIEW OF APPROACH

The goal of our approach is to reduce the energy consumption of the HTML pages displayed by a mobile web application. To do this, we automatically rewrite an application so that its generated HTML pages use more energy efficient color schemes and layouts. Our approach can be described as having three phases. An overview of these phases is shown in Figure 1. The first phase is HTML Output Analysis (Section 4). In this phase, the approach builds a model, the HTML Output Graph (HOG), of the HTML pages that can be generated by the application. Then using the HOG, the approach builds the HTML Adjacency Relationship Graph (HARG), which captures the visual relationships, such as adjacency or containment, between pairs of HTML elements. The second phase, Color Transformation (Section 5), builds a Color Conflict Graph (CCG) that describes the relationships between the colors of HTML elements that have a visual relationship. Using the CCG, the approach generates the Color Transformation Scheme (CTS), a new energy efficient color scheme for the application. This is done for an application by calculating a new color scheme that maintains the color distances represented in the CCG, but whose primary colors will consume less energy during display. The third and final phase is Output Modification (Section 6). The result of this phase is that the approach rewrites the application so that the generated HTML pages use the colors contained in the CTS.

4. HTML OUTPUT ANALYSIS

In the first phase, the approach builds models of the application that describe the structural relationships among the HTML elements of the application's web pages. We call this model the HTML Adjacency Relationship Graph (HARG) and it shows which HTML elements can be adjacent to each other or contained by one another. To generate the HARG, the approach first builds another model, the HTML Output Graph (HOG), which describes the HTML pages that can be generated by the application. We explain the two models in more detail below.

4.1 The HTML Output Graph

The HTML Output Graph (HOG) represents the potential HTML output of a web application and is based on an abstraction proposed by Møller and Schwarz [30]. Intuitively, the HOG is a projection of the web application's control flow graph (CFG) where all of the nodes are instructions that generate HTML output. An HOG is represented as a tuple $\langle V, E, v_0, v_f \rangle$. V is the node set where $v \in V$ if the node is in the application's CFG and prints to the application's output stream. In the Java Enterprise Edition (JEE) framework, an example of such nodes would be invocations to `JspWriter.println`. $E \subseteq V \times V$ is the edge set where an edge $(v_i, v_j) \in E$ if there is a path from v_i to v_j in the CFG of the web application with no other node $v_k \in V$ along that path. $v_o \in V$ and $v_f \in V$ are, respectively, the entry and exit nodes of the HOG. The approach builds an HOG for each method of the web application by analyzing the method's CFG. The HOG for the entire application can be obtained by treating each node in the HOG that represents an invocation as a transition to the entry node of the target method's HOG.

To identify the HTML strings produced by each output generating node $v \in V$ we also define a string analysis, S, that maps each v to a finite state automaton (FSA). Our approach assumes that strings defined external to the application, such as user input, will not influence the color or the structure of the HTML tags. For the string analysis, we used a technique proposed by Yu and colleagues [40]. We selected this method because it defines mechanisms for handling common string related operations that appear in web applications but are difficult to analyze in traditional string analyses. In particular, it handles strings generated within loops via concatenation and string replacement by defining a `widen` operation that abstracts repeating portions of the loop so that the FSA representing a string variable within a loop can converge on a safe approximation.

The HOG is suitable for modeling the pages produced by dynamic web applications that directly generate HTML using mechanisms such as JSP, Servlets, or Struts. However, many modern web applications also contain HTML template files that are then filled in by application logic to generate the final HTML content. This is very common in web application frameworks that implement the Model-View-Controller (MVC) pattern, such as Apache Velocity and WebMacro. For these types of applications, the approach builds the HOG directly from the template files. To do this, the approach opens all macros in the template files and identifies the entire HTML frame. Then, the approach defines each line in the HTML frame as a node in the HOG and defines edges based on the order of the lines in the template file. The process for construction of a HOG for template based applications can vary based on the framework, but in general, the process requires a mixture of the string analysis based approach and the template parsing discussed here.

4.2 The HTML Adjacency Relationship Graph

The HTML Adjacency Relationship Graph (HARG) models the visual relationship between pairs of HTML elements in the HOG. The type of information present in the HARG is similar to the Document Object Model (DOM), in that it shows parent/child and sibling relationships. However, since the HARG is built from the HOG, it also contains relationships that could be derived from loop generated HTML elements. The HARG is defined as a tuple $\langle V, E \rangle$ where each $v \in V$ is a node in the graph that corresponds to an HTML element that can be generated by the application. $E \subseteq V \times V$ is the edge set where $(v_i, v_j) \in E$ means that v_i is a parent HTML element of v_j. To illustrate, the HARG for Program 1 is shown in Figure 2.

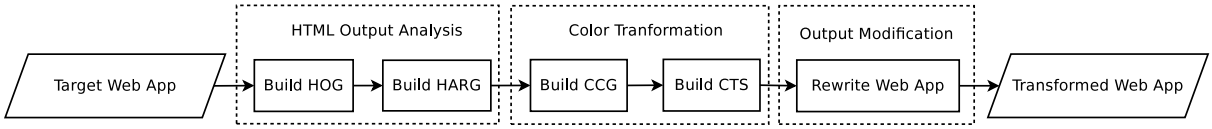

Figure 1: the architecture of Nyx

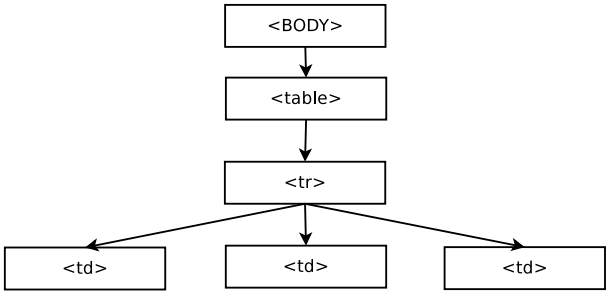

Figure 2: Example HARG for Program 1.

To build the HARG, our approach parses the HOG to aggregate the individual characters in each node's FSA into HTML tags. The traversal begins by traversing all of the edges in the FSA associated with the root node of the HOG and then following all of the outgoing edges of the root node and repeating this process until all nodes in the HOG have been traversed. During the traversal, the approach maintains a parse state that allows it to determine if it is currently parsing an HTML tag, attributes, or text. When the parsing discovers an HTML tag, it creates a corresponding node in the HARG if it is an opening tag or a self-closing tag. There is an edge (v_i, v_j) in the HARG if and only if all of the following four conditions hold: (1) v_i is an opening HTML tag; (2) there is a path P in the HOG from v_i to v_j; (3) the closing tag of v_i is not in P; and (4) along path P, if there is a node v_k that meets conditions (1) – (3), v_k is equals to v_i. Basically, these conditions enforce that v_j will be a child of v_i, contained within v_i's opening and closing tags, and that v_i is the most immediate predecessor that satisfies these condition.

Loops in the FSA will generate infinite strings. When the parsing encounters a loop in the FSA, the approach simulates its unraveling n times. This unraveling may be unsafe because it is possible that the $n+1$ traversal introduces new strings that are not included in the previous n unravelings. However, we have found that for the purpose of identifying the color attributes assigned to each tag, n has a reasonably small bound. In the analysis, we employ the following heuristic: n is assigned the maximum of either the integer value 6 or one more than the largest iteration of repeating substrings in the CSS file. For example, in the string $NyxNyxNyx$, Nyx is a repeating substring. We use the value 6 since this is the maximum iteration of repeating substrings of a hexadecimal string that can be defined as the value of color in an HTML attribute. Case in point, a color is defined by a six digit hexadecimal number (e.g., #000000) and each iteration of a loop could provide one character of this string. Since our goal is to capture potential color information, this gives the approach a reasonable upper bound on the loop unraveling. In practice, we found that 6 was always sufficient and there was no incompleteness in the HARG due to unraveling the loop in this way.

More broadly, the techniques for obtaining the HOG can lead to an over approximation of an application's possible HTML pages. In turn, this can lead to the identification of spurious visual relationships that correspond to infeasible paths. This does not cause a problem for the approach, as this merely introduces additional color relationship constraints that must be accounted for while generating the Color Transformation Scheme in Section 5.3.

5. COLOR TRANSFORMATION

In the second phase, the approach calculates the new energy efficient color scheme for the application – the Color Transformation Scheme (CTS). There are two requirements for the CTS, it must: (1) use energy efficient colors as the basis for the new color scheme, and (2) maintain the color relationships between neighboring HTML elements. The first requirement serves the general goal of the approach and the second ensures that the color-transformed pages are readable and, ideally, as visually appealing as the original pages. To address the first requirement, the CTS should replace large, light colored background areas with dark colors (preferably, black), as mentioned in Section 2. To address the second requirement the approach must transform the other colors of the HTML elements so that their color relationship with the new dark-colored background is similar to their color relationship with the previous light-colored background.

Our approach produces a CTS that meets both requirements. To do this, our approach first builds a Color Conflict Graph (CCG), which describes the color relationships between pairs of HTML elements that have a visual relationship. To begin, our approach changes the background color of the root node of the CCG to black. Generally, the root node of the CCG corresponds to the <body> tag, but can differ for certain layouts. Then the approach calculates a new recoloring of the CCG so that the color distances between adjacent nodes in the recolored graph are the same as color distances in the original graph. This mapping of old colors to the transformed colors is the output of the second phase. Our approach operates on three different types of CCG, the Background Color Conflict Graph (BCCG), which models the relationship between the background colors of neighboring HTML elements; the Text Color Conflict Graph (TCCG), which models the relationship between text colors and their corresponding background HTML element colors; and the Image Color Conflict Graph (ICCG), which models the relationship between an image and its enclosing HTML tag. In the remainder of this section, we first give a formal description of the CCG and its three variants, then introduce how we derive them from the HARG, and finally discuss the calculations that generate the CTS.

5.1 The Definition of Color Conflict Graph

The CCG and its three subtypes, the BCCG, TCCG, and ICCG, show the color relationships between the HTML elements of a page generated by the application. Formally, the

CCG is represented as a weighted complete undirected graph $\langle V, v_0, W \rangle$. The set V represents the graph's nodes, where each node represents a color in the HTML page. $v_0 \in V$ is the root-node of the graph, which is the color that will be transformed to black. Typically, v_0 is the background color of the $< body >$ tag, but users can specify a different root node for unusual HTML layouts. W is a weighting function $W : V \times V \rightarrow I$. Since the CCG is a complete graph, there is an integer edge weight defined for every pair of tuples in V. The weighting function is used to give priority to certain types of visual relationships. The BCCG, TCCG, and ICCG vary in the weights attached to certain definitions.

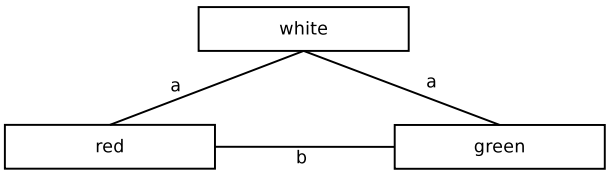

Figure 3: Example BCCG for Program 2.

The BCCG models the relationship among all background colors in the web page. There are three types of relationships modeled in the BCCG: (1) parent and children nodes, (2) sibling nodes, and (3) all other nodes. The weights for these edges are assigned as the constants a, b, c, respectively, where $a > b > c > 0$. We rank the parent-child relationship as the most important. The reason for this is that a parent element's color generally surrounds their children node in the rendering of the HTML page, which means that the color distance for these elements must be maintained to visually distinguish between the elements. We rank the sibling relationship as next important because, generally, siblings are rendered close to each other on a page and therefore their color difference is more important to maintain than that among the remaining elements. Finally, we attach a weight to c because maintaining an element's color difference relationship with the other elements on the page helps to preserve the overall aesthetics of the original color scheme, but is not as important as the other two relationships. An example BCCG with edge weights for Program 1 is shown in Figure 3. In our example, white is the background color of the $< body >$ tag at line 3. The $< table >$ and $< tr >$ tags at line 4 inherit this color. Red and green are the background colors for the $< td >$ tags at line 10 and 13, which are children of the $< tr >$ tag at line 4. Thus, the weight between white and red, or white and green, is a, while the weight is b between red and green.

The TCCG models the relationship between text colors and the background color of the enclosing HTML element. Therefore, only edges that represent a link between these two types of colors are given a non-zero weight a with the remaining edges being assigned a weight of 0.

The ICCG models the relationship between colors in an image and the background color of the HTML element that surrounds the image. Therefore only edges in the ICCG that connect an image's colors to the background color of its enclosing HTML elements are given a non-zero weight of a with the remaining edges being assigned a weight of 0. In some cases, it is not desirable to transform every image in the web application. For example, it may be preferable to not alter a photo in a news article since the original appearance relates to the integrity of the story. Developers can specify a list or pattern of image file names that should not be transformed. When the approach finds an image tag including one of the excluded image tags, it does not construct an ICCG or CTS for the image.

5.2 Building the CCG

The CCG is built using the information contained in the HARG. The general intuition of this transformation is that the approach identifies color definitions in the HARG and propagates the definitions along the graph to elements that may inherit the color. The propagated information is then used to identify colors that have a relationship with each other and construct the CCG.

The first step is to identify the color definitions (CDs) generated by each node in the HARG. A CD is generated when an HTML element contains an attribute that defines either the text/background color of the element, or the HTML element is an image tag. For example, background colors of some elements can defined by the `bgcolor` attribute and the color of text or links can be defined by the `text` or `link` attribute. For pages that use CSS, the approach identifies the set of CDs that an HTML element generates based on its ID, class name, or type, which can be determined using a standard CSS parser. An image tag generates a CD for every color used in the image.

The second step is to propagate all of the background CDs to the other nodes in the HARG. This is done to determine which background colors will be adjacent to each other and which image and text colors will appear over a particular background color. The approach propagates the color information using standard iterative data flow analysis [7]. The `Gen` set of a node is comprised of the CDs generated at that node. The `Kill` set kills all CDs that flow in to the node if the node generates a CD. For example, for a node v_i in the HARG, the approach propagates all of its background CDs to a child v_j if v_j does not generate any background CDs. Standard equations are used for the `In` and `Out` sets. Note that CDs originating from images and text are ignored during the propagation.

The final step is to derive the CCG from the colors propagated over the edges of the HARG. Nodes are created slightly differently for each CCG variant. In the BCCG there are nodes for each unique background CD generated in the HARG. Nodes in the TCCG include those in the BCCG plus nodes for the CDs originating from text colors and the ICCG includes the nodes in BCCG plus nodes for the CDs originating from image colors. Since the CCG is a complete graph, there is an edge defined between each pair of its nodes. The edges of the CCG are assigned weights based on the different types of visual relationships discussed in Section 5.1. In general, the weighting is done by iterating over each node v in the HARG and identifying v's set of corresponding nodes in the CCG, N_c. For each edge in the set $\{(n, n_i) | n \in N_c \wedge n_i \in N\}$ where N is the CCG node set, the appropriate weight is assigned based on the relationship it represents and the variant of the CCG.

The construction of the CCG does not take into account the effects of embedded JavaScript. This would result in an incomplete model of the color relationships if JavaScript was used at runtime to modify the colors of a web page. To determine if this would impact our approach, we conducted a small scale study on the use of JavaScript to modify colors in a web page at runtime. In this study, we examined the top

50 web sites, as ranked by `http://mostpopularwebsites.net/`, to determine how JavaScript affected the colors of a website. In all 50 we found that JavaScript was not used to modify the colors. We believe that this result generalizes beyond the top 50 websites and indicates that accounting for JavaScript in the construction of the CCG is not necessary.

5.3 Generating the CTS

To generate the CTS, the approach analyzes each variant of the CCG and computes a recoloring that is more energy efficient, but maintains, as closely as possible, the color differences between nodes in the graphs. In this section we explain the analysis of the BCCG in detail and then briefly describe the analogous process for the ICCG and TCCG.

To transform the background colors, the approach converts the color of the root node of the CCG to black and then transforms the other background colors to maintain their color distances. To state this more formally, let $S = \{C_0, C_1, C_2, C_3....C_k\}$ be the set of colors of each node in the CCG where C_0 is the background color of the root-node (v_o) and each $C_i, i > 0$ is the color for the remaining k nodes. The approach creates a new color scheme S' in which C'_0 =black and then finds color mappings for $C'_1, C'_2...C'_k$ that minimize the overall difference in the color differences of S and S'. The function to be minimized is

$$H = \sum_{i=0}^{k} \sum_{j=0}^{k} w_{ij} |Dist(C_i, C_j) - Dist(C'_i, C'_j)|$$

Where w_{ij} is the weight of the edge between colors i and j in the BCCG. Basically, this minimization function is closer to zero the more closely each of the color difference distances in S' match the color difference distances in S.

Our approach maps this minimization problem to the Energy Minimization Problem[1] (EMP), which is a well-known pixel recoloring problem in the computer vision field [39]. The EMP minimizes the cost of a set of pixels and their labels. Given a set of pixels $P = \{P_0, P_1, ...P_k\}$, and a set of labels $L = \{L_0, L_1.....L_n\}$, and a cost function $Cost(P, L, f)$, where f is a mapping from L to P, the EMP finds a transformation f that minimizes the cost function $Cost$. In our mapping, all nodes in the CCG, the whole color space, and the H function are the pixel set, label set, and cost function respectively. Our problem is then to find a mapping from the whole color space to all nodes in the CCG that maps the root-node to black and minimizes the H function.

This minimization problem is NP-hard, but an approximation can be solved for in a reasonable time using a Simulated Annealing Algorithm (SAA) [39]. For our approach, a close to optimal solution for S' satisfies our two requirements for the CTS and, as we show in Section 7, allows the approach to compute the CTS in a reasonable amount of time. An SAA is a technique for finding a good approximate solution in a very large search space and works by probabilistically exploring states until a good enough solution is found or the computation budget is fully consumed [24]. SAAs are a well-known technique utilized in search-based software engineering and are considered a good fit for problems where identifying an approximate solution in a large search space is sufficient [29, 19]. Our approach's adapted SAA is shown in Algorithm 1. The input of Algorithm 1 is the original color scheme S, and the CCG. The output is the transformed color scheme, S', with the background color of the root node transformed to black.

The SAA begins with an initial color scheme $Current$ (line 1), which is generated by a greedy algorithm $GreedyInit$. The purpose of $GreedyInit$ is to identify a reasonable starting point for the SAA. The basic approach of $GreedyInit$ is to first flip the color of the root node, which is C_0 in S, to black. The algorithm then traverses over each of the remaining nodes in the CCG in order of decreasing edge weight. For each node, $GreedyInit$ assigns a color that minimizes the cost function H for all nodes that have been visited.

Our search-based algorithm needs a time budget T in case the algorithm does not converge on the optimal solution. A counter T that represents the allocated time or computational budget is initialized with an integer at line 3. The SAA iterates until T reaches 0 (lines 4-15). In each iteration, the approach identifies a possible new color scheme $Next$. This is done by calling $Random\_Successor$, which generates a new color scheme by modifying each color in the current scheme, except for C_0, by a random amount (line 6). If the new color scheme minimizes the cost function, H, more than the current one, then the new scheme $Next$ replaces the current scheme (lines 10 - 12). If the new scheme is not an improvement, then the current scheme may still be changed with some small probability (line 13) to prevent the algorithm from getting stuck at a local optimum. The probability function is based on the size of the counter T and the most recent difference of H. Finally, after the counter expires, the current best solution is returned. This represents the S' for the CCG.

Algorithm 1 Simulated Annealing Algorithm

Input: S, CCG
Output: S'
1: $Current \leftarrow GreedyInit(S, CCG)$
2: $BEST \leftarrow Current$
3: $Initilize(T)$
4: **while** $T > 0$ **do**
5: $\quad Decrease(T)$
6: $\quad Next \leftarrow Random\_Successor(Current)$
7: $\quad$ **if** $H(Current) < H(BEST)$ **then**
8: $\quad\quad BEST \leftarrow Current$
9: $\quad$ **end if**
10: $\quad$ **if** $H(Next) < H(Current)$ **then**
11: $\quad\quad Current \leftarrow Next$
12: $\quad$ **else**
13: $\quad\quad Current \leftarrow Next$ with $P = e^{\frac{H(Current)-H(next)}{T}}$
14: $\quad$ **end if**
15: **end while**
16: $S' \leftarrow BEST$

The approach for computing the CTS for the TCCG and ICCG is very similar to the process described above. A key difference is that the background color transformations identified in S' are substituted into the corresponding colors in the TCCG and ICCG. These transformed colors are treated as fixed and the remaining colors (color of the text and colors within an image) are transformed using the above process. Because a significant subset of the colors are fixed, the cost function H can be optimized further. For space reasons, we omit the description of this optimization. The CTS for the

[1]Here, the term "energy" refers to general cost

TCCG is a transformation for all of the text elements where each new text color maintains the color difference with the transformed background color of its enclosing HTML element. The CTS for the ICCG is a recoloring of each image so that each color in the image maintains its color distance with the transformed background color of its enclosing HTML element. Note that every color in an image is transformed with respect to maintaining the color relationship with the enclosing HTML element, not the other colors in the image. In cases where the image contains color gradients or shadows, this generally results in a less attractive transformation. In fact, our evaluation showed that the attractiveness of applications with more transformed images was generally rated lower than the original. In future work, we plan to investigate more advanced image processing techniques that could improve the aesthetics of recolored images.

6. OUTPUT MODIFICATION

In the third phase, the approach rewrites the web application so that it generates web pages based on the CTS. For our approach we have two different mechanisms for realizing the modifier. For CSS files and HTML templates, our approach simply uses regular expressions to replace all color strings with their corresponding colors. In practice we have found that more sophisticated approaches, such as using CSS parsers to identify style properties to modify is unncessary. For colors that are defined by dynamically generated HTML, the approach inserts instrumentation to perform the rewrite at runtime. The instrumentation replaces the APIs that print HTML to clients (e.g., `JspWriter.println`) with calls to customized printing functions. These printing functions scan the output as its generated and replace printed colors with their corresponding colors in the CTS.

7. EVALUATION

We performed an empirical evaluation of three aspects of our approach, time cost, energy savings, and user acceptance of the appearance of the transformed web pages. We implemented our approach in a prototype tool, Nyx, and used it to address four research questions:

RQ1: How much time does Nyx take to generate the CTS?

RQ2: How much energy is saved by the transformed web pages?

RQ3: What is the runtime overhead introduced into the modified web applications by Nyx?

RQ4: To what degree do users accept the appearance of the transformed web pages?

7.1 Subject Applications

We use seven open source Java-based web applications to evaluate our approach, including applications that have been used in related work, to ensure a broad representation of implementation styles. These applications are implemented using different web application frameworks, include colors defined by HTML and CSS, and employ a varying amount of JavaScript in their user interfaces.

Details of each of these apps are shown in table Table 1. For each subject, the column labeled "Framework" shows the underlying web framework for which the application was implemented. Frameworks included in our study are JSP, a very popular web application framework for Java based web application; Servlet, which describes applications that directly use the Java Enterprise Edition (JEE) framework with no intermediate framework; Struts, a very widely used library and framework for web applications; and, Velocity and Turbine, two popular template based frameworks for developing web applications. The column labeled "SLOC" shows the number of source lines of code in Java for each web application. For applications that are written in JSP, we converted the JSP code into Java, using the Tomcat Jasper compiler, and counted the resulting SLOC.

Our subject applications also represent varying levels of CSS and Javascript usage. JavaLibrary, Portal, and Bookstore define their styles in HTML directly, while ClassRoom, Roller, Scarab, and jForum use CSS to define their style. ClassRoom, Portal, and Bookstore do not make heavy use of JavaScript, while JavaLibrary, Roller, Scarab, and jForum do. Three of the applications, Roller, Scarab, and jForum, use the Model-View-Controller (MVC) architectural style. All applications, except Bookstore and Portal, are publically available from their project web pages. Bookstore and Portal have been widely used in related work [17, 16] and are available via the SQL Injection Application Testbed [3].

7.2 Implementation

We implemented our approach in a prototype tool, Nyx. To generate the Output Graph, we leverage Soot [6] to build the underlying call graphs, control flow graphs and the Jasmin representation for Java classes. For representing the FSAs of each string in the Output Graph and HTML Content Graph, we use the BRICS automaton library [4]. As mentioned earlier, to build the required string analyses, we implemented the concatenation, replacement and widening operations from Yu and colleagues' method [40] and combined them with the BRICS automaton library. We also built an automaton parser for the BRICS library to get the tag name, CSS ID, class name, and color information of HTML tags. We used the SAC CSS parser [2] to identify colors from CSS files. For the Output Modification phase, we used BCEL [1] to modify Java classes and Perl script to modify colors in the CSS files. Our implementation handles HTML 4 and CSS 2, but it is straightforward to extend our tool to support HTML 5 and CSS 3.

7.3 RQ1: Time Cost

To address the first research question, we ran Nyx on all of the subject applications and measured the execution time. The results are shown in Table 1. We separated the runtime into four different parts. The first is the time spent loading all of the Java classes, parsing templates, and building call graphs. This time is shown under the column labeled "Load." The second is the time spent building the Output Graph, HTML Content Graph, and CCG. This time is shown in the column labeled "Analyze." The third is the time spent in calculating the CTS, which is shown under the column labeled "Transform." Finally, the rewriting time is shown in the column "Rewrite." All results were run on a DELL XPS 8100 desktop running Linux Mint 14 with an Intel Core i5@3GHz processor and 8GB memory. Each timing result reported was the average runtime of 10 executions.

As the results show, overall it takes less than three minutes to analyze and transform each subject application. Most of the time cost is incurred in either the Load or Transform

Table 1: Subject Application Information

Application Information			Time cost(s)				Energy saving(%)	
Name	Framework	SLOC	Load	Analyze	Transform	Rewrite	Loading Energy	Display Power
Bookstore	JSP	24305	46.2	9.64	27.5	1.8	26.7	47.2
Portal	JSP	21393	45.1	8.34	53.8	1.7	24.7	44.2
JavaLibrary	JSP&Servlet	73468	45.8	21.7	29.9	2.9	26.1	35.8
ClassRoom	JSP	5127	18.1	5.97	0.385	0.1	35.8	51.6
Roller	JSP&Struts	154065	0.018	1.23	102	0.2	10.4	18.0
Scarab	Velocity&Turbine	145435	0.016	1.84	27.1	0.2	27.1	47.8
jForum	Velocity	31841	0.014	1.94	154	0.1	26.7	47.8

time periods. For the apps with a high Loading time, most of this time was spent by Soot in building the call graphs of the application. Roller, jForum, and Scarab have very small Load times because we can build the Output Graph directly by parsing the templates files instead of analyzing Java classes. Roller, jForum, and Scarab also have a very small analysis cost since the string analysis for templates is much simpler than for Java classes. The length of the Transform time was highly dependent on the structure of the web pages generated in the application. For more complex pages with many colors it took longer to generate a new color scheme.

7.4 RQ2: Energy Saving

To evaluate the energy savings of our approach, we deployed the original and transformed subject web applications on a Tomcat web server. We then accessed both versions of the web application using a Samsung Galaxy II smartphone and measured the energy/power consumption of the phone using the Monsoon Power Meter [5].

There are two distinct energy/power phases when a mobile phone visits a web application. The first phase is "Loading and Rendering", in which the browser loads the contents of the web page and renders them on the screen. The second phase is "Display" in which the mobile phone has finished loading and just displays the web page contents on the screen. A key distinction is that the potential time for Display is unbounded, it ends when the user closes the browser or moves to another page. In contrast, the time for the Loading is bounded. Therefore, for fairness, we measure the *energy* consumed during the Loading and Rendering phase and the *power* draw during the Display phase. This is more fair than simply measuring the energy of both phases, since it is possible to inflate energy savings by allowing the Display phase to continue for an extended period of time. (Recall that energy = power * time.)

To differentiate these phases, we leveraged the energy and power measurements provided by the Monsoon Power Meter. Key to doing this was understanding what happens on a smartphone during these different phases. In the Loading and Rendering phase, multiple components in the smartphone, such as CPU, memory, WIFI, 4G network, and the screen, are busy. The Loading Phase has a limited time span, it starts at the point that the browser sends the request to the server and ends when the browser finishes rendering the contents of the web page to the screen. In contrast, during the Display phase, all components except the screen of the smartphone are in the idle state. Therefore, we can figure out the start and end times of the Loading and Rendering Phase by observing the power state of the mobile phone in the power meter. The start point of the Loading and Rendering phase is the point when the phone switches to the high power state and the end point is when the phone switches back to the low power state. The start point of the Display phase is the end point of the Loading and Rendering phase. For both phases, we took measurements of the original and transformed web application 10 times and reported the average percentage decrease in the columns of Table 1 labeled "Loading" and "Display."

On average, there was a 25% decrease in energy consumption during the Loading and Rendering phase and 40% less power consumed during the Display phase for the transformed web applications. Overall, these are strong results and show that our approach can result in significant energy savings for smartphone users.

Of interest to us was the fact that energy decreased during the Loading and Rendering phase. This was puzzling since the transformation did not change the size of the pages in any meaningful way. In our investigation, we learned that in order to speed the display of the web pages, the smartphone begins to display parts of the screen, such as background color, as soon as possible. Therefore, there was energy consumed by the screen even during the Loading and Rendering phase. This difference became more significant during the Display phase, when only the screen was actively drawing energy. Also, we investigated the lower savings incurred by the Roller app. We found that Roller only covers about 60% of the screen. Because of this, we can only change colors for 60% of the screen used by the web application, to save energy. The other 40% is left as white, which is the default color of the web browsers. This suggests that an easy to achieve optimization would be to change the default background color of mobile browsers to black.

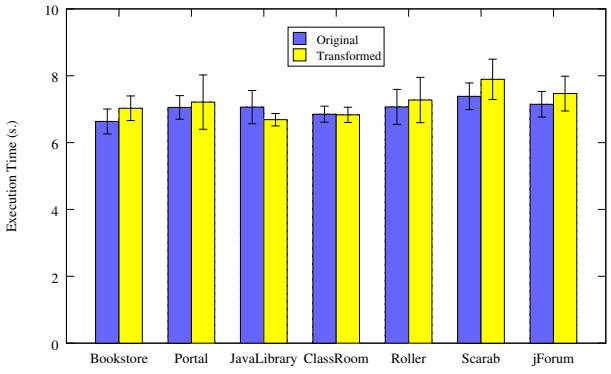

Figure 4: Overhead

7.5 RQ3: Runtime Overhead

The Modifier introduces additional operations into the web application; namely, rewriting the color attribute of HTML strings. Therefore, we are interested in measuring the runtime incurred via this operation. For the experi-

ment, the server was a Core i7@2.8GHz desktop with 8GB of RAM running Linux kernel 3.8 and Tomcat 6. The smartphone was a Samsung Galaxy II running Android 4.0 and connected to the server via wireless. To calculate the overhead, we compared the average time of the Loading and Rendering phase. We used the time for this phase as it represents the time that users need to wait before they can see the contents of the web application. We measured this time on the server and client side over ten executions of the experiment for RQ2.

The results of this experiment are shown in Figure 4. The results show that, on average, the transformed versions take 2.4% more time than the original. However, as can be seen in the figure, sometimes the transformed application is faster. Even for applications where we only transformed CSS files, we saw similar differences. We investigated this further by checking the results across different executions. Our results indicated that average loading time for all versions was about 7 seconds and even the same version of an application would routinely vary by up to 1.2 seconds with a standard deviation of about 5.6%. From this data we concluded that variations in the wireless signal were likely dominating any variation introduced by our modification overhead. To eliminate interference from the wireless, we also measured execution time just on the server side. On average the server side increase was 34ms, which represented about a 22% increase. However, the actual distribution was bi-modal with an average of a 75% increase for apps whose code was modified as opposed to almost 0% for those with only template changes. This result is fairly intuitive, as any modification to a template based web application did not require much additional runtime overhead and in cases where runtime transformations were required, there were relatively few of these operations.

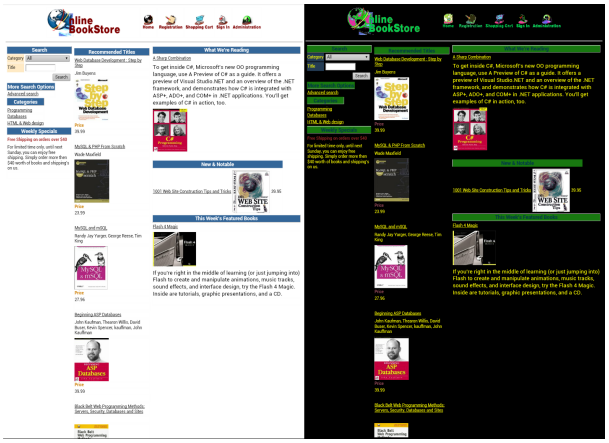

Figure 5: Comparison in questionnaires

7.6 RQ4: User Acceptance

To address the fourth research question, we conducted an end-user survey in which users were asked to compare and rate the appearance of the original and transformed web applications. The survey group was 20 M.S. and Ph.D. students at the University of Southern California who were enrolled in the third author's graduate level testing and analysis class. The students were asked to complete an anonymous online survey on their own time and no incentives were provided to the students to complete the survey. No background on the research project was given to the students and no connection of the work to the third author was suggested.

Table 2: Subject Application Information

Name	Attractiveness Ori	Attractiveness Trans	Readability Ori	Readability Trans	Preference(%)
Bookstore	6.5	4.2	7.6	5.9	24
Portal	6.9	5.3	7.5	5.6	18
JavaLibrary	6.7	6.9	7.0	6.4	29
ClassRoom	6.8	6.4	7.2	7.1	59
Roller	7.0	6.5	6.9	5.5	24
Scarab	7.4	5.4	6.9	6.5	18
jForum	7.0	5.4	7.0	5.4	12

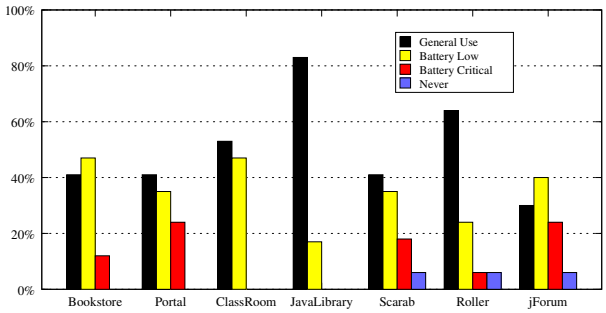

Figure 6: The Acceptance rate of transformed web application

The survey presented users with a series of before/after screenshots of the seven subject applications. An example for the Bookstore application is shown in Figure 5. For each image, the survey group was asked to rate the attractiveness and readability of each version on a scale of 1 to 10, with 10 being the highest. The users were then asked which version they would prefer to use. Finally, the last question asked if the black background version could save them X% energy, at what battery level would they choose to use it. For each app, X was replaced by the energy savings of the Display phase. The available responses were "Always–regardless of battery level," "Most of the time," "Only when the battery level is low," "Only when the battery level is critical," and "Never." The wording of the questions and forms are available via the project web page [20].

We received 17 responses to the survey. The results are shown in Table 2 and Figure 6. In Table 2, the columns Attractiveness and Readability report the related scores of both the original version and transformed version. The subcolumns Ori and Trans represent the original and transformed version receptively. The column Preference reports the percentage of users who prefer the transformed web application over the original one. The In Figure 6, we report when users would choose to use the transformed web application. For space reasons, we merged the option "Always–regardless of battery level," and "Most of the time" into "General Use." The bars show the different time points. The y-axis is the percentage of users who would switch to the transformed version at each time point.

For attractiveness, the original app received an average score of 6.9 and the transformed app a score of 5.7, an average decrease of about 17%. This indicates that generally the users thought the color scheme of the original apps were more visually appealing than the transformed version, but only by a relatively small difference. In fact, for one

app, JavaLibrary, users found the transformed version to be more attractive. For readability, the original apps received an average score of 7.1 and the transformed apps a score of 6.1, an average decrease of about 14%. In general, as we examined the per app results in more detail, we noticed that applications whose screenshots contained a higher amount of transformed images, Bookstore, Portal, and jForum, received significantly lower scores. We hypothesize that our rather crude transformation of image colors, which neglects shadows and gradients, impacted this score significantly. Transformed images, in general, were not as clear or readable as their original versions. In future work, we plan to explore improved image processing techniques for transforming image colors.

For user preference, it was clear that users preferred the original version based on visual appearance and usability along. On average, over 73% of the users preferred the original application. However, when asked to consider the energy savings, there was a dramatic shift in user preference. On average 67% of the users chose to use the transformed version for general usage if it could save them X% of energy during display. Overall, more than 97% of users chose to switch to the transformed version before the battery became critically low.

Overall, we consider the results for user acceptance to be positive. Although users rated the attractiveness and readability lower of the transformed apps, when made aware of the energy savings, they overwhelmingly preferred to use the transformed application.

8. RELATED WORK

The closest work to Nyx is Mian and colleagues' work Chameleon[14]. Their approach modifies the source code of browsers to change the colors of web pages in the rendering buffer. It first manually builds color transformation schemes for each of the top twenty web sites, such as Google, and saves them in a cloud server. When the browser sends request to one of these web sites, it queries the cloud server and downloads the pre-installed transformed color schemes. Then, it renders the transformed web application with the downloaded transformation scheme. Nyx is different from Chameleon in two aspects. First, Nyx builds the color transformation scheme automatically for web applications. Thus, our approach is more easily applied to a broad range of web applications. Second, our approach modifies the web application directly on the server side. Thus it does not introduce the client-side cost of obtaining the transformation or applying it in the browser.

Other approaches to save energy for OLED screens have also been proposed. Kamijoh and colleagues' work [23] is one of the first to optimize energy for OLED screens. It optimizes the energy consumption of OLED screen for the IBM Wristwatch by reducing the number of pixels that are bright. However, this work only considers two colors, the black background color and the different foreground color. As such, this approach is not applicable for color displays.

Choi and colleagues' method [12] reduces the energy consumption of LCD screens by reducing the screen refresh rate, color depth, and luminance. However, this approach relies on changing the hardware circuitry of LCD screens.

Lyer and colleagues [21] also proposed a method for changing colors on OLED screens to save energy. This method saves energy by darkening the areas that are not focused on by users. The drawback of this approach is that the contents in the darkened area are not readable. Compared with this approach, Nyx can better maintain the readability of the entire page.

Energy consumption of mobile devices can also be optimized in other ways. One category of approaches for saving mobile energy is detecting the misuse of sensors [31, 28]. Our approach EDTSO [26] saves the energy consumption of test suites with energy directed test suites minimization. Zhong and colleagues [42] optimized the communication energy of mobile phones by redesigning the communication protocol. Rodoplu and colleagues [32] proposed a deployment algorithm to minimize the energy consumption of ad hoc networks. Chen and colleagues [10] proposed a method to save energy consumption for Java-based mobile applications by offloading workload to a server.

Another related group of work is energy modeling and measuring. Mian and colleagues [13] model the energy consumption of OLED screens. They discovered that the energy consumption of OLED is linear to the RGB value. Our previous works [18, 25] model and measure the energy consumption of mobile devices on a source line level. Tiwari and colleagues [37, 38] model the CPU energy on instruction level. eProf [31] models energy with a state machine. Some other approaches [13, 41, 27] model energy consumption on a system call level. All the approaches mentioned above do not optimize the energy consumption for mobile applications, they only model or estimate energy consumption.

Besides energy saving, transformation techniques for web applications are also used to optimize the user experience for mobile devices. Jones and colleagues [22] improve the readability and attractiveness of web applications on mobile devices by manually redesigning the layout of web applications. Bila and colleagues [8] design a system that enables end-users to adjust the layout of web applications manually. Chen and colleagues [11] improve the readability of web applications on mobile devices by partitioning web pages into segments. Minimap [33] improves the readability by enlarging the contents of the web application.

9. CONCLUSION AND FUTURE WORK

This paper presents a new technique, Nyx, to make web applications more energy efficient for OLED based mobile devices. The basic idea of Nyx is to replace the large, light colored background areas of web applications with dark colors (preferably, black) to reduce the energy consumed by OLED screens. During the transformation of an application's web page colors, Nyx also tries to maintain the aesthetics of the original web application. An evaluation for Nyx on seven open source web applications shows that it can reduce energy consumption by an average of 40% with only a minor reduction in users' rating of the pages' attractiveness and readability. Our user study also shows that 97% of users will accept the transformed web application generated by Nyx if the battery power is critically low.

10. ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation under Grant No. CCF-1321141.

11. REFERENCES

[1] http://commons.apache.org/proper/commons-bcel/.

[2] http://cssparser.sourceforge.net/.

[3] http://www-bcf.usc.edu/~halfond/testbed.html.

[4] http://www.brics.dk/automaton.

[5] http://www.msoon.com/LabEquipment/PowerMonitor/.

[6] http://www.sable.mcgill.ca/soot/.

[7] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[8] N. Bila, T. Ronda, I. Mohomed, K. N. Truong, and E. de Lara. PageTailor: Reusable End-user Customization for the Mobile Web. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, MobiSys '07, pages 16–29, New York, NY, USA, 2007. ACM.

[9] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[10] G. Chen, B.-T. Kang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and R. Chandramouli. Studying Energy Trade Offs in Offloading Computation/Compilation in Java-Enabled Mobile Devices. *IEEE Transaction on Parallel Distributed System*, 15(9):795–809, Sept. 2004.

[11] Y. Chen, W.-Y. Ma, and H.-J. Zhang. Detecting Web Page Structure for Adaptive Viewing on Small Form Factor Devices. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 225–233, New York, NY, USA, 2003. ACM.

[12] I. Choi, H. Shim, and N. Chang. Low-power Color TFT LCD Display for Hand-held Embedded Systems. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, ISLPED '02, pages 112–117, New York, NY, USA, 2002. ACM.

[13] M. Dong, Y.-S. K. Choi, and L. Zhong. Power Modeling of Graphical User Interfaces on OLED Displays. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 652–657, New York, NY, USA, 2009. ACM.

[14] M. Dong and L. Zhong. Chameleon: A Color-adaptive Web Browser for Mobile OLED Displays. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 85–98, New York, NY, USA, 2011. ACM.

[15] M. Dong and L. Zhong. Power Modeling and Optimization for OLED Displays. *IEEE Transactions on Mobile Computing*, 11(9):1587–1599, Sept. 2012.

[16] W. G. J. Halfond and A. Orso. Command-Form Coverage for Testing Database Applications. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 69–80, Washington, DC, USA, 2006. IEEE Computer Society.

[17] W. G. J. Halfond and A. Orso. Improving Test Case Generation for Web Applications Using Automated Interface Discovery. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 145–154, New York, NY, USA, 2007. ACM.

[18] 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.

[19] M. Harman. The Current State and Future of Search Based Software Engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 342–357, Washington, DC, USA, 2007. IEEE Computer Society.

[20] http://www-scf.usc.edu/∼dingli/Nyx. Nyx Project Page.

[21] S. Iyer, L. Luo, R. Mayo, and P. Ranganathan. Energy-Adaptive Display System Designs for Future Mobile Environments. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 245–258, New York, NY, USA, 2003. ACM.

[22] M. Jones, G. Marsden, N. Mohd-Nasir, K. Boone, and G. Buchanan. Improving Web Interaction on Small Displays. *Comput. Netw.*, 31(11-16):1129–1137, May 1999.

[23] N. Kamijoh, T. Inoue, C. M. Olsen, M. T. Raghunath, and C. Narayanaswami. Energy Trade-offs in the IBM Wristwatch Computer. In *Proceedings of the 5th IEEE International Symposium on Wearable Computers*, ISWC '01, pages 133–, Washington, DC, USA, 2001. IEEE Computer Society.

[24] P. Laarhoven and E. Aarts. *Simulated Annealing*, volume 37 of *Mathematics and Its Applications*. Springer Netherlands, 1987.

[25] 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 '13, pages 78–89, New York, NY, USA, 2013. ACM.

[26] D. Li, C. Sahin, J. Clause, and W. Halfond. Energy-directed test suite optimization. In *Proceedings of the 2nd International Workshop on the Green and Sustainable Software*, GREENS '13, pages 62–69, 2013.

[27] T. Li and L. K. John. Run-time Modeling and Estimation of Operating System Power Consumption. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 160–171, New York, NY, USA, 2003. ACM.

[28] Y. Liu, C. Xu, and S. Cheung. Finding Sensor Related Energy Black Holes in Smartphone Applications. In *Proceedings of the 2013 IEEE International Conference on Pervasive Computing and Communications*, PerCom '13, pages 2 –10, Washington, DC, USA, 2013. IEEE Computer Society.

[29] P. McMinn. Search-based Software Test Data Generation: A Survey: Research Articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[30] A. Møller and M. Schwarz. Automated Detection of Client-state Manipulation Vulnerabilities. In *Proceedings of the 2012 International Conference on*

Software Engineering, ICSE 2012, pages 749–759, Piscataway, NJ, USA, 2012. IEEE Press.

[31] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is Keeping My Phone Awake?: Characterizing and Detecting No-sleep Energy Bugs in Smartphone Apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[32] V. Rodoplu and T. Meng. Minimum Energy Mobile Wireless Networks. In *Proceeding of the 1998 IEEE International Conference on Communications*, volume 3 of *ICC '98*, pages 1633–1639, Washington, DC, USA, 1998. IEEE Computer Society.

[33] V. Roto, A. Popescu, A. Koivisto, and E. Vartiainen. Minimap: A Web Page Visualization Method for Mobile Phones. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 35–44, New York, NY, USA, 2006. ACM.

[34] R. Shamey, D. Hinks, M. Melgosa, R. Luo, G. Cui, R. Huertas, L. Cardenas, and S. G. Lee. Evaluation of performance of twelve color-difference formulae using two NCSU experimental datasets. In *Proceeding of the 2010 Conference on Colour in Graphics, Imaging, and Vision*, volume 2010, pages 423–428. Society for Imaging Science and Technology, 2010.

[35] G. Sharma, W. Wu, and E. N. Dalal. The CIEDE2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations. *Color Research & Application*, 30(1):21–30, 2005.

[36] J. Shinar and V. Savvateev. Introduction to Organic Light-Emitting Devices. In J. Shinar, editor, *Organic Light-Emitting Devices*, pages 1–41. Springer New York, 2004.

[37] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, 1994.

[38] V. Tiwari, S. Malik, A. Wolfe, and M. Tien-Chien Lee. Instruction level power analysis and optimization of software. *Journal of VLSI signal processing systems for signal, image and video technology*, 13(2-3):223–238, 1996.

[39] O. Veksler. *Efficient Graph-based Energy Minimization Methods in Computer Vision*. PhD thesis, Ithaca, NY, USA, 1999. AAI9939932.

[40] F. Yu, T. Bultan, M. Cova, and O. Ibarra. Symbolic String Verification: An Automata-Based Approach. In K. Havelund, R. Majumdar, and J. Palsberg, editors, *Model Checking Software*, volume 5156 of *Lecture Notes in Computer Science*, pages 306–324. Springer Berlin Heidelberg, 2008.

[41] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM.

[42] L. Zhong, M. Sinclair, and R. Bittner. A Phone-Centered Body Sensor Network Platform: Cost, Energy Efficiency & User Interface. In *Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks*, BSN '06, pages 179–182, Washington, DC, USA, 2006. IEEE Computer Society.