GreenBundle: An Empirical Study on the Energy Impact of Bundled Processing

Shaiful Alam Chowdhury
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
shaiful@ualberta.com

Abram Hindle
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
abram.hindle@ualberta.ca

Rick Kazman
Information Technology Management
University of Hawaii
Honolulu, HI, USA
kazman@hawaii.edu

Takumi Shuto
Information Sc. & Electrical Eng.
Kyushu University
Fukuoka, Japan
shuto@posl.ait.kyushu-u.ac.jp

Ken Matsui
Information Sc. & Electrical Eng.
Kyushu University
Fukuoka, Japan
matsui@posl.ait.kyushu-u.ac.jp

Yasutaka Kamei
Information Sc. & Electrical Eng.
Kyushu University
Fukuoka, Japan
kamei@ait.kyushu-u.ac.jp

***Abstract*—Energy consumption is a concern in the data-center and at the edge, on mobile devices such as smartphones. Software that consumes too much energy threatens the utility of the end-user's mobile device. Energy consumption is fundamentally a systemic kind of performance and hence it should be addressed at design time via a software architecture that supports it, rather than after release, via some form of refactoring. Unfortunately developers often lack knowledge of what kinds of designs and architectures can help address software energy consumption. In this paper we show that some simple design choices can have significant effects on energy consumption. In particular we examine the Model-View-Controller architectural pattern and demonstrate how converting to Model-View-Presenter with bundling can improve the energy performance of both benchmark systems and real world applications. We show the relationship between energy consumption and bundled and delayed view updates: bundling events in the presenter can often reduce energy consumption by 30%.**

***Index Terms*—energy consumption, MVP, MVC, architecture**

I. Introduction

Energy consumption is an important quality requirement for mobile devices [1] and for mobile software such as apps [2], [3] that affects availability, battery life, and sales. Unfortunately, and often, app developers are addressing energy consumption when it becomes a problem [2], [4], rather than at design time before coding starts. There is evidence that developers simply are not trained enough in the topic of energy consumption at the application level to be able to address energy consumption effectively [4], [5]. Also, available optimization tips do not impact energy consumption in real-world apps, demanding higher-level changes for efficient accesses of energy hungry components [6]. Unfortunately, developers have little idea about what design choices are even available that will affect energy consumption, as well as the consequence and tradeoffs of those design choices. Yet interest exists as Manotoas et al. [4] show: "experienced practitioners are often willing to sacrifice other requirements for reduced energy usage". This paper discusses the kinds of design choices and tradeoffs that architects face, and seeks to illustrate how we can improve energy consumption of mobile applications (and, indeed, of any application) by relatively small changes in an architecture. Specifically, we show how a change to Model-View-Presenter (MVP) with "bundling" or "dropping" strategies can improve the energy performance of apps.

But why focus on mobile? By 2019, the number of global smart-phone users is expected to reach 2.7 billion [7]. Smartphones are essentially portable networked pocket computers powered by batteries [8]. Smartphone apps range from email apps to games, to notifications, prompts, reminders, stock tickers, etc. This wide variety of software and uses is ever present as the network bandwidth demands on mobile networks starts to eclipse PC network bandwidth [9]. This pressure on functionality and portable computing puts a huge strain on a mobile device's battery, which unfortunately has not seen much technological improvement [10]. If the device's battery energy is consumed, the device is typically unusable. The importance of energy consumption on mobile devices has immediate consequences: app developers quickly learn that their apps that use lots of energy suffer in ratings [2] as consumers highly value battery life for their mobile devices [1].

We seek to aid developers in addressing energy consumption at design time, *before* runtime. Our concern is that developers do not have good guidelines or evidence-based models of the costs and benefits of the design choices they make in the design of energy efficient apps. Developers lack knowledge of architectures, patterns, and tradeoffs that are potentially "green" (energy efficient), or the parameters that can make an architecture "green" at runtime. So in this work, we demonstrate how the Model-View-Presenter pattern (MVP) can be modified to reduce the event processing overhead of model objects notifying view objects. We discuss how to modify the presenter of MVP into a proxy that bundles requests or drops redundant requests by delaying notifications—thus avoiding frequent expensive intermediate notifications or

context switches that update views. Our research questions and contributions include:

RQ 1: What is the impact of the number of event sources and event generation rates on software energy consumption?

RQ 2: Can bundling and dropping events help in saving energy while varying number of sources and rates?

Contribution 1: we developed a benchmark Android app that follows Model-View-Presenter (MVP) architecture to understand the impact of bundling and dropping on Model-View-Presenter architecture. We implemented the presenter in three different forms: no bundling, bundling, and dropping. The number of event sources and the rate of event generation (i.e., number of events/second) are determined at runtime with user input. Using the benchmark app, we answer RQ 1 and RQ 2.

RQ 3: What are the energy impacts of bundling and dropping on real-world applications?

RQ 4: With bundling and dropping, can developers save energy without harming user experience?

Contribution 2: We confirm the realism of the findings from the benchmarks with four different real-world Android apps to answer RQ 3 and RQ 4. Because benchmark apps, although good for conducting controlled tests, do not necessarily reflect real-world scenarios and performance [6].

RQ 5: Why do bundling and dropping save energy?

Contribution 3: We investigate the cause of performance changes by analyzing resource access patterns (e.g., CPU use) of apps with bundling and dropping to answer RQ 5.

RQ 6: What are the maintainability consequences of implementing bundling and dropping on Android apps?

Contribution 4: We analyze the difficulty of incorporating bundling/dropping in Android apps, and the consequences of these changes on maintainability, to address RQ 6. Decoupling Level (DL) metric [11] is used for analyzing the maintainability cost of bundling and dropping versions.

We show that a small change to an architecture like MVP allows for making energy consumption tradeoffs, allowing for energy-aware decision making during design and maintenance phases. In general, developers can save significant amount of energy by adopting the proposed bundling and dropping mechanisms and that without harming user experience and without materially affecting the maintenance cost. To support reproducibility and extension, our energy measurements and the open-source benchmark app are shared publicly [12].

II. Background

A. Energy Efficiency is Difficult to Achieve

Power (P) is the rate of work expressed in *watts*. Energy (E), expressed in *joules*, is the total amount of work in a given time (T): $E = P \times T$. Energy consumption is linearly proportional to the run-time of a component, but only when P is constant. A reduced time T can save energy, but what if the CPU switches to a higher power consuming state for a reduced time T? Without actual energy measurements or estimates, this is hard to answer. Moreover, CPU access patterns are just one of the many considerations that affect energy consumption in modern devices [13], [14]. Studies have recommended energy efficient Java collections [15], [16], energy-efficient communication protocols [17], [18], locating and finding energy bugs [19]–[22], and building models and tools for energy estimation [14], [23]–[30]. Despite the increasing amount of energy efficiency research, it is often unclear how software design decisions impact energy consumption [31], and what tradeoffs developers should be aware of [4], [5].

B. Model-View-Presenter

Model-View-Presenter (MVP) is a form of Model-View-Controller (MVC) [32], [33]. MVC often uses a design pattern such as the observable pattern to ensure synchronization between data in the *model*, and visual or concrete representations in the *views*, while shielding the model from direct manipulation from view objects via a *controller*. MVC has many variants. Some have different purposes. One popular variant of MVC is called *active MVC* [34], that is typically implemented with a single process whereby the observer pattern is used to allow interactions between the *model* objects and the *view* objects. In Active MVC, model objects are observables that notify observers (views) when their representation or data is updated. This is done by keeping a list of observers and then notifying each via a method call that the observable they are watching has been updated. It is then up to the observer to query the model objects for the information they need. This can be quite cumbersome as every change can cause a cascade of observers to react and deal with each change, regardless of the granularity or usefulness of the change. Another problem with this pattern is that it puts the notification and listener logic into model objects.

Active MVC is cumbersome and requires many model objects to keep track of observers. *Model-View-Presenter* is a variant that uses the observer pattern, but it provides a *proxy* (presenter) between the model and the views. The model objects, when modified, updates the presenter. The presenter notifies views and provides them with the information they need to update. The views do not necessarily need the model objects as the presenter is in the way, thus isolating the model objects further from views, while removing the responsibility of model objects to notify views for updates. The controller part of MVP is often folded into the presenter object itself. Using this presenter as a *proxy* allows one to put delegation logic into the presenter and keep that logic out of the model objects. This means that a presenter could, for example, bundle updates or drop updates that were deemed irrelevant. Both of these choices could improve the runtime behaviour of an application. Because of its simplicity, the MVP pattern has been recommended in several developers' blogs and discussions [35]–[37]. Our approach, however, can also be adopted with other architectures besides MVP.

C. Events, Bundling and Dropping Presenters

An event can be a database update request, a packet transmission request, a view update request and so on. *Bundling* is the act of storing and queuing incoming events such that they can be processed together, even periodically. *Dropping* is the

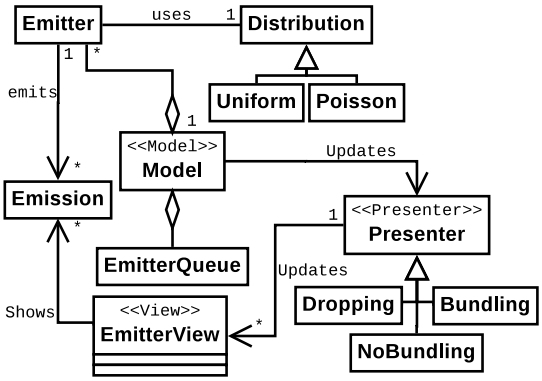

Fig. 1: UML class diagram of the benchmark app.

act of keeping only the last incoming event to be processed—periodically, or on demand, it processes only the most recent event. A *bundling presenter* stores incoming events and send them later in a single batch for processing them together. On the other hand, a *dropping presenter* discards previous events and processes only the most recent one. Bundling is applicable when a delay in processing is acceptable, whereas dropping is relevant when, along with the delay tolerance, the most recent event nullifies the importance of previous events.

Bundling has been found energy efficient in earlier studies. Pathak *et al.* [38] proposed I/O bundling for reducing tail energy leaks in mobile apps. The authors found that some hardware components, such as the network interface card or SDCard, suffer from the tail energy phenomenon. Tail energy is the wasted energy by a component while transitioning from the active to the inactive state. Bundling operations that involve such hardware devices reduces the tail energy phases significantly, and thus save software energy consumption. For the same reason, Chowdhury *et al.* [39] found that writing log messages in batches can reduce energy consumption. Other research found that `HTTP/2` servers can reduce clients' energy consumption by enabling a form of bundling, compared to the `HTTP/1.1` servers [18]. Lyu *et al.* [40] has shown that energy efficiency can be significantly improved by grouping multiple database auto-commit transactions into a single transaction.

In this paper, however, we focus on modifying an existing architectural pattern (MVP) with dropping and bundling. We show that by adopting this modified architecture one can gain the maintainability and architectural benefits of MVP. Yet developers may still decide how to balance event-based energy consumption against other qualities such as latency.

III. Methodology

This section describes the benchmark app we developed for our experimentation, along with our energy measurement process and test scripts for driving the subject apps.

A. The Benchmark App

This app has three major components including Model, View, and Presenter in compliance with the MVP pattern. Figure 1 illustrates the benchmark app with a class diagram.

The benchmark app, with an UI, allows a tester to choose a configuration of parameters to test. For example, the number of emitters (i.e., event sources), event generation rate, test run duration, and the version of the presenter—no bundling, bundling, or dropping—can be selected at run time. For bundling and dropping, a delay parameter is also provided, i.e., how long the app should wait for collecting the incoming events before processing all the saved events in a single batch? This UI, with a button click, can then spawn a new experiment running on a thread separate from the UI thread [41]. The new experiment will have emitters and views of emitters' emissions instantiated.

1) Model: The model is a collection of emitters objects (i.e., event sources) based on the user's input. The model is responsible for dealing with the emitters and forwarding their emissions to the presenter. The model has a *registerObserver* method which can add any number of presenters that can be interested in an update from this model. However, for simplicity, we used only one presenter in our experiments. The model is an observable from the Observer pattern. The model component uses four different sub-components: Emitter, Emission, Distribution, and EmitterQueue.

a) Emitter: An emitter is an event source that emits events at a given rate (i.e., number of events/second). Each emitter creates emission objects that contain all the data of the next scheduled transmission from that emitter. The emitter is also responsible to notify the model about emissions, so that the model can notify the interested presenter. Emitters are meant to simulate event sources like stock prices, weather information, or sensor output.

b) Emission: An emission is an event that contains some data, usually a message. These messages are randomly generated and timestamped.

c) Distribution: Each emitter produces emissions following a probability distribution function (PDF) for scheduling the next emission. The benchmark app is designed to accommodate any PDF at run-time. For simplicity and low variability in our energy measurements, we used only the uniform distribution.

d) EmitterQueue: The EmitterQueue uses a priority queue (Java's PriorityQueue) to schedule emitters for emitting and transmitting the next emissions. The priority queue enacts an efficient algorithm for scheduling the emitter. The EmitterQueue sorts all the emitters based on their next waiting time. The model then removes the first emitter from the priority queue, finishes its transmission, and then insert it again based on its next scheduled emission time. This process continues until the test run duration expires.

2) Presenter: The presenter is an observer of the model component. It is notified whenever one of the model's emitters transmits. The presenter maintains a mapping of emitters and views, which the presenter uses to notify the view of the corresponding emitter with the emission. There are 3 kinds of presenter used in both the benchmark App and the study: i) No bundling—forwards the update immediately; ii) Bundling: waits for the given bundling time, saves all the incoming

updates, and forwards each of them all together; iii) Dropping: same as the bundling except the presenter forwards only the most recent update and discards all the previous updates.

The presenter runs in a separate thread than the UI thread. When the presenter receives an update, it decides which view to notify and passes off the necessary information to the view in the view's thread—such as the UI thread if the view has a UI. The bundling presenter, for example, sleeps inside a timer thread and stores the incoming events in parallel. It then forward all the stored events to the interested views once the sleeping time is over (i.e., the bundling time provided by the user). The dropping presenter is identical except it discards previous events and only forwards the most recent one. To help practitioners for implementing bundling/dropping presenters, we made the benchmark app public and open-source [12].

3) View: Views are meant to receive updates from the presenter. What they do with the update is up to them, but typically they only talk to the presenter and updates them with the emission objects they receive. They are observers of the presenter but might be associated with a particular object. For simplicity, the benchmark app maintains a one-to-one relationship between the emitters and view components—a single view, a *textfield*, is interested in a single emitter. A view in the benchmark app is thus responsible to display the received emission data from a emitter through the presenter.

B. Energy Measurements and Test Scripts

We used two implementations, to verify the generalizability of our proposed approach, of the *GreenMiner* [42] software energy measurement platform to measure the energy consumption and resource usage of the apps used in this paper. The *GreenMiner's* tests and measurements can be accessed remotely and *GreenMiners* have been used extensively in a variety of software engineering energy consumption research [15], [18], [30], [42]–[47].

The system under test is typically an Android smartphone. Energy is measured using current sensor INA219 and INA159 chipsets that report to an Arduino Uno microcontroller. The microcontroller processes and aggregates measurements, sending to the test computer—a Raspberry Pi model B computer. The current sensors and the Pi are connected to the phones under test. The first *GreenMiner* is connected to 4 Galaxy Nexus phones (system-under-test) running Android 4.4.1 with an INA219, while the *GreenMiner-2* is connected to an ASUS ZenFone 2 running Android 5.0.2 with an INA159. For a given app and test, the Pi acts as the test-runner which pushes, runs, and collects measurements for a given test script. The first *GreenMiner* system has four identical settings with four Galaxy Nexus phones. Running different tests in parallel helps accelerate the measurement process. *GreeMiner-2* has only 1 ASUS Zenphone 2. *GreenMiner* test framework cleans any previously installed apps before running a new test. This is to ensure the same system state for each test; energy consumption of a particular test is therefore unaffected by previous tests. We ran the real-world app tests on the *GreenMiner* and *GreenMiner-2*; the benchmark app was tested solely on the *GreenMiner* to simplify comparison of parameters and energy consumption.

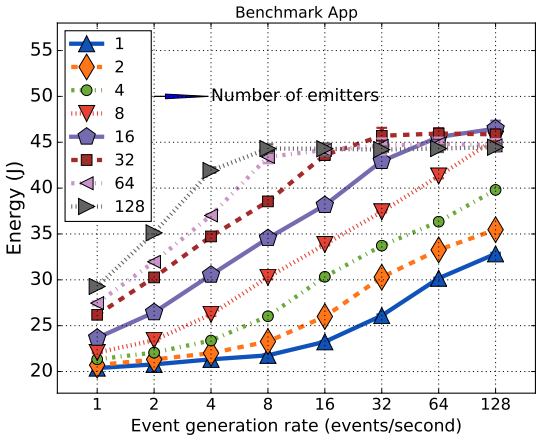

Fig. 2: Energy consumption of the benchmark app with different numbers of emitters and event generation rates. Bars indicate the 99% confidence interval.

Previous *GreenMiner* based works [15], [18], [30], [44], [46] recommended running the same test multiple times, for the observed variability in energy consumption. We ran all versions of our subject apps (benchmark and real world apps) **10** times. To minimize outlier effects, we show the 99% confidence interval of our measurement distribution. In addition, whenever necessary, we used the Kruskal-Wallis test [48] to verify if two energy measurement distributions are statistically different. The advantage of Kruskal-Wallis test is that it does not assume any distribution of the data as it is non-parametric.

To measure the energy consumption of an app, we need to run the app multiple times, which is infeasible with manual testing. We used the Android's `adb shell` [49] script for writing the test cases. For the benchmark app, writing the test script was straightforward. The script selects the number of emitters, the presenter type, the bundling time in case of bundling and dropping, and provides the test duration and event generation rate before clicking the start button. For the real-world apps, however, the tests were written with two of the authors' consensus that these tests represent an average user's interaction with these apps.

IV. Results: Benchmark App

In this section, we show the energy consumption of the different versions of the benchmark app with different settings. To select the number of emitters, and event generation rate, we use powers of 2: 1, 2, 4, 8, 16, 32, 64, and 128. This large range is able to show the big picture: the impact on energy consumption with the increase in the number of emitters and event generation rate. All versions (at all settings) of the benchmark app were run for a fixed period of 20 seconds, repeating 10 times each.

RQ 1: What is the impact of the number of event sources and event generation rates on software energy consumption? Figure 2 shows the energy consumption of

the benchmark app with different numbers of emitters (i.e., event sources) and event generation rates. Clearly, the energy consumption goes up when we increase the number of emitters and/or the event rate. The Spearman [50] correlation coefficient is 0.66 ($p \approx 0$) between the number of emitters and energy consumption in *joules*. The coefficient is 0.69 ($p \approx 0$) between the event rates and energy consumption. The coefficient would have been higher if the phones were able to process high numbers of events and emitters. The variations in energy measurements among multiple runs for each setting are small; Figure 2 shows that the 99% confidence intervals do not overlap until performance is saturated. This is because we kept the benchmark app as simple and deterministic as possible, which is harder to control in real-world apps. We also observe that for high number of emitters (i.e., $\geq$32 emitters) the energy consumption does not change with the increase in the event rate after a threshold. This is because of the limited capacities of the phones we used for our measurements; these phones can process a certain number of events within the allotted 20 seconds test duration. Producing more events than this threshold does not impact the energy consumption, for the phone can not process the extra events within the allotted time. In fact, with the Kruskal-Wallis test, we found statistical differences between the energy measurements until the numbers of emitters and rates are high. For example, with 8 emitters, $\alpha = 0.05, p = 0.0001$ between 32 events/sec and 64 events/sec. However, for $\alpha = 0.05$, p is statistically insignificant (0.6242) between 64 events/sec and 128 events/sec when the number of emitters is fixed to 64.

Table I shows the percent increase in energy consumption in the number of emitters and event generation rates compared against the energy consumption of one emitter with one event per second. For example, even with a single emitter, the energy consumption can go up 38% when the event generation rate is high (128 events/second). And note that the percentage increase with high numbers of emitters and rates would have been much higher than the reported values if the phones were to able processing more events.

Findings: Many modern applications deal with large numbers of event sources with high numbers of incoming events [51]–[53]. Our results show that energy consumption is correlated with both the number of event producers and the rate of event production.

RQ 2: Can bundling and dropping events help in saving energy while varying number of sources and rates? To answer this question, we considered three different waiting times for both bundling and dropping: 0.1 second—the corneal reflex time of human eyes; 0.5 second—half of user-acceptable latency; and 1 second—the broadly used acceptable latency target for interactive applications [54]. Unlike the real-world apps presented later, a wider range of waiting times were not considered for the benchmark app so the graphs are readable.

Figure 3 shows the energy savings of different bundling and dropping rates compared with no bundling or dropping (presented as Nobundling in the figures). Each graph shows the results for all the possible scenarios for a fixed number of emitters. Except for very high numbers of emitters or rates, the energy consumption goes up monotonically for the Nobundling version. While this trend is true for the bundling versions as well, for the dropping versions we do not see such clear trends. This is because the number of events processed by a dropping version (transferring events from the presenter to the views) is to some extent independent of the number of events generated. The small energy increase for the dropping versions with increased rates is due to the cost of producing more events by the model component in our benchmark app. Not surprisingly, dropping is more energy efficient than bundling; the dropping versions process fewer events than the bundling versions. The bundling versions, in spite of processing the same number of events as the no bundling version, can save significant energy.

It is encouraging that, with bundling, we can process and deal with the same amount of workload, and yet can make apps significantly more energy efficient. Table II shows the energy savings by different bundlers and droppers with fixed one emitter. This is to ensure that the energy consumption is not affected by resource limitations, thus enabling accurate comparison. It shows that with bundling (doing all the work without dropping anything) and maintaining a latency such that a user does not notice any change (0.1 second), we can still save up to 25% of the energy (Bundling-0.1s). With user-acceptable latency (Bundling-1s), bundling can save up to 31% in a simple app like our benchmark, with just one event source. We verified with the Kruskal-Wallis test that these differences are indeed statistically significant (with $\alpha = 0.05, p \leq 0.01$).

TABLE I: Percent increase of energy consumption compared with the energy consumption of 1 emitter and 1 event/second. For readability, nearest integer values are presented.

	Rates							
Emitters	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
1	0	2	4	6	12	22	32	38
2	2	4	7	12	22	33	39	43
4	5	8	13	22	33	40	44	49
8	8	13	22	33	40	46	51	55
16	14	23	33	41	47	53	55	56
32	22	33	41	47	53	55	56	56
64	26	36	45	53	54	54	55	54
128	30	42	51	54	54	54	54	54

TABLE II: Energy savings (in percent) by different bundlers and droppers when compared with no bundling or dropping. Results are presented for just one emitter.

	Rates							
Versions	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
Bundling-0.1s	0	0	0	0	3	13	22	25
Dropping-0.1s	0	0	0	0	4	14	25	30
Bundling-0.5s	0	0	2	3	8	18	27	30
Dropping-0.5s	0	0	2	3	9	19	29	34
Bundling-1s	0	1	4	5	10	19	29	31
Dropping-1s	0	1	4	5	11	20	31	36

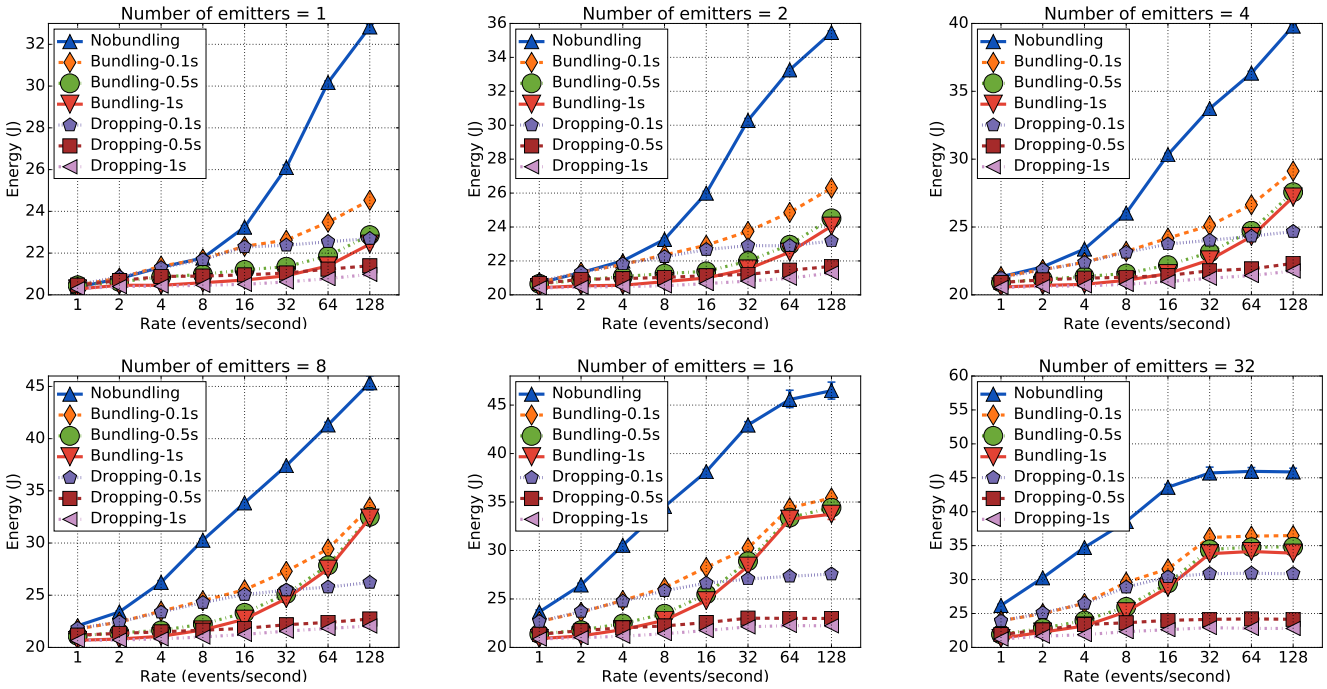

Fig. 3: Energy consumption of bundling and dropping compared with the No bundling versions for different number of emitters. Results for more than 32 emitters are not presented; due to resource limitations, the energy consumption is inconsistent for very high numbers of emitters and rates. Bars indicate the 99% confidence interval.

Findings: Dropping is the most energy efficient approach compared to bundling and no bundling. However, dropping might not be an acceptable alternative when accuracy is important. Application developers can consider bundling in such scenarios, which still saves significant energy over no bundling.

V. Real world Apps

Results from the benchmark app shows that bundling and dropping can save significant energy in Android apps. And this saving is larger with increased numbers of event sources or event generation rates. It is, however, not obvious how such optimization approaches would perform in real-world apps [6]. In this section, we evaluate bundling and dropping in four selected real-world Android apps.

A. Selection of Applications

The apps we selected had to be open source so that we can implement bundling and dropping. We explored the F-Droid repository [59] to find suitable apps. F-Droid contains source code for all the posted Android apps and was used in earlier mining software repositories research [60]–[62]. Finding suitable apps with reasonably small code size (so that we could easily identify where to implement bundling and dropping) was challenging, which hindered us from analyzing more apps. Table III shows the characteristics of the four selected apps.

The different types and code sizes of these four apps enables reliable evaluation of bundling and dropping. The Sensor Readout app is also available on Google Play [63] and has been downloaded more than 50,000 times (as of writing). This app has received 540 reviews with an average rating of 4.3/5. This allows evaluating energy optimization techniques for apps that are already popular. AcrylicPaint, a finger painting app, represents apps where users might spend more continuous time, making energy optimization more crucial.

We have implemented the bundling and dropping versions of these apps, except for Sensor Readout, following the approach presented in section III-A2. The original Sensor Readout app, uses a timer function, and processes only 10 measurements per second, although the app samples measurements continuously. As a result, we did not have to implement our own timer for the bundling and dropping variants. The apps, however, did not follow a clear MVP pattern. Instead, their designs were closer to the MVC pattern. We have identified which classes contained the actual processing code, and refactored those classes to accommodate our bundling/dropping presenters. Our intention was to convert the existing design as close to the bundling MVP pattern as possible. For ensuring correctness, two of the authors were involved in refactoring and testing the apps afterwards.

B. *RQ 3: What are the energy impact of bundling and dropping on real-world applications?*

The energy savings from bundling and dropping are of course impacted by the bundling/dropping time—the time these two variants wait before processing a batch of events. We selected six different times: 0.01s—fastest human time perception; 0.03s—animation speed; 0.1s—the corneal reflex time of human eyes; 0.2s—double the corneal reflex time; 0.5s—half of user-acceptable latency; 1 second—acceptable

TABLE III: Description of the selected four real-world Android apps from F-droid.

App	Type	# Classes	ULOC	Test scenario	Test duration (s)
Sensor Readout [55]	"Real-time graphs of sensor data"	56	6009	Measure the Gyroscope sensor	70
ColorPicker [56]	"Pick colors and display values"	12	908	Move the scroll bars for R,G,B	50
Angulo [57]	"Angle and Distance Measuring"	4	497	Start the measurements and wait	55
AcrylicPaint [58]	"Simple finger painting"	7	936	Draw a hexagon	27

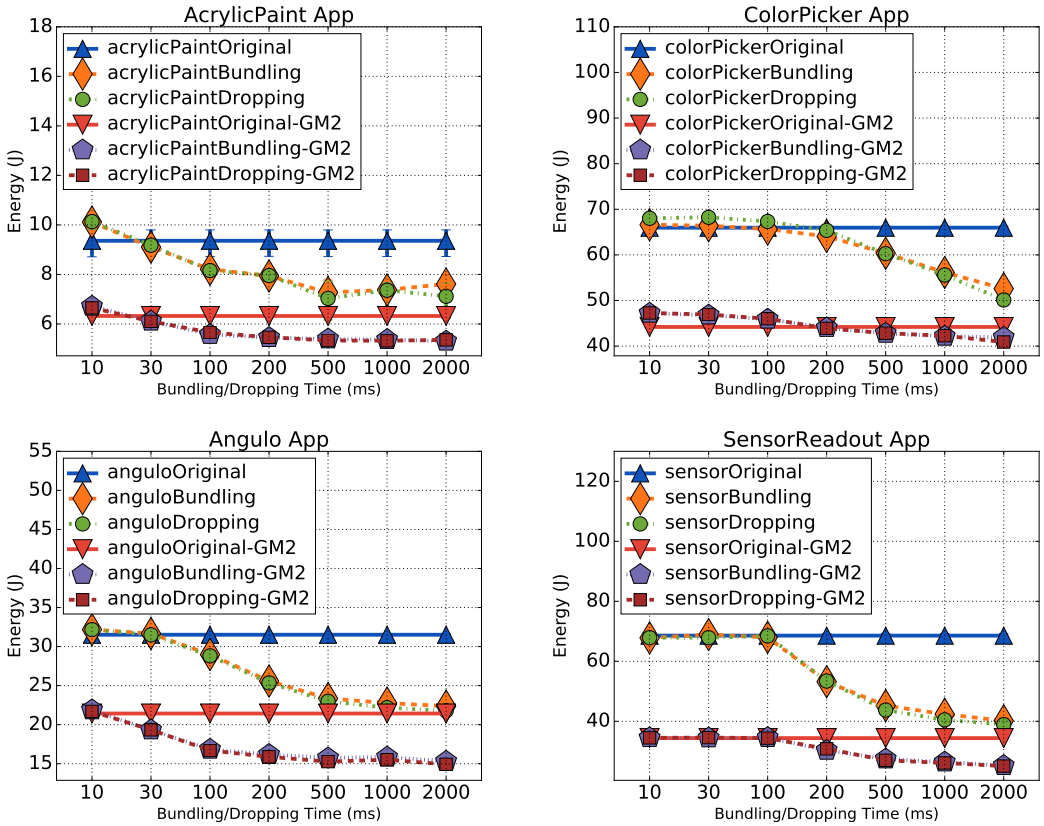

Fig. 4: Energy consumption of bundling and dropping compared with the original versions of four real-world apps. Bars indicate the 99% confidence interval.

user latency [54]; and 2 seconds—double user-acceptable latency. Figure 4 shows the energy consumption of bundling and dropping compared with the original versions of the four selected apps. Here, we also show the results for the *GreenMiner-2* (GM2) on the Asus Zenphone 2.

Except for the Sensor Readout app, we observe more energy consumption for the bundling and dropping versions for very low bundling and dropping times (e.g., 0.01s). This suggests that running a timer thread for bundling or dropping incurs energy consumption overhead. As we mentioned before, Sensor Readout did not require a separate timer thread and does not have this overhead when bundling or dropping is used. For all the apps, however, the energy consumption of bundling and dropping improve significantly when the waiting time is reasonably higher. For example, even for 0.1s latency which is difficult for users to perceive, the bundling and dropping versions of AcrylicPaint and Angulo can save 12% (12% with GM-2) and 9% (8% with GM-2) energy consumption respectively, when compared with their original versions. The energy savings become significant for larger latency. For example, we can save 37% (24% with GM-2) energy consumption for the Sensor Readout app with a 1s latency in drawing the measurements graphs, and that without losing any measurements (i.e., with bundling).

GreenMiner-2 (with the ASUS ZenFone 2 phone) consumes less energy than the *GreenMiner* (with the Galaxy Nexus phone) for all apps, and thus the energy savings are generally lower. The trends in percent of energy consumption reductions, however, are similar across the apps for both the GreenMiners.

C. *RQ 4: With bundling and dropping, can developers save energy without harming user experience?*

Answering this question might require analysis from multiple perspectives. However, with one case study, we show that there are scenarios where the developers can adopt our bundling approach to address user feedback that involves energy expensive modifications. For this study, we selected

the Sensor Readout app—the only app available on Google Play with a significant number of reviews.

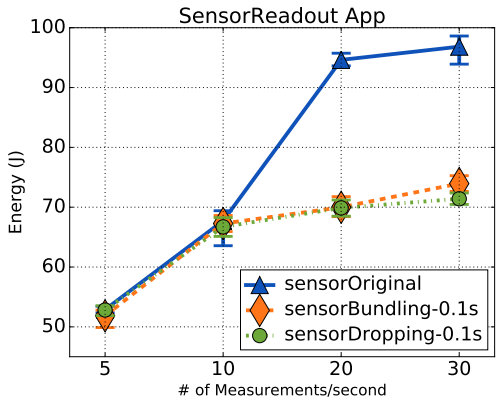

Fig. 5: Energy savings of bundling and dropping for the Sensor Readout app with higher sampling rates. Bars indicate the 99% confidence interval.

In general, this app is praised by users. However, there are reviews suggesting some user dissatisfaction. For example, one user desires to see colour changes with different measurements while updating graphs [55]. Another user is unsure why the sampling rates from the sensors are low—10 samples per second. Changing colors continuously and sampling at higher rates might increase the energy consumption of the app significantly. And yet, developers need to carefully address user feedback for their apps to stay popular [64]. In fact, there are reviews on the Sensor Readout app suggesting similar apps supposedly better than Sensor Readout. We show that Sensor Readout can benefit by applying bundling—with much higher sampling rates—without losing any measurements and without harming latency in updating graphs.

Figure 5 shows the energy consumption of the original version compared with the bundling and dropping versions, with different sampling rates. The bundling/dropping time (i.e., the latency in updating the graphs) is fixed to 0.1s. The average energy consumption of the original version is high (≈94 joules) when the sampling rate is 20/second compared with the original 10/second (≈69 joules); a 36% increase in energy consumption. The difference is also statistically significant (Kruskal-Wallis test, $\alpha = 0.05, p < 0.01$). However, bundling with 20 samples/second consumes similar energy to the original version with just 10 samples/second. With sampling rate higher than 20, the energy consumption of the phones does not increase as expected. The Galaxy Nexus phones are unable to process more than a threshold number of samples.

Findings: Real-world Android apps can save significant energy with bundling and dropping. With more sacrifice in latency, the energy saving is higher. But bundling with almost imperceptible latency (0.1s) is a complete win, saving energy without affecting user satisfaction. In addition, in some scenarios, bundling and dropping can help developers address user concerns with no meaningful sacrifice in usability.

VI. Understanding Resource Utilization Patterns With Bundling and Dropping

It is unsurprising that dropping saves energy; in dropping the presenter only sends the most recent event for processing. This requires less CPU slots for the process (also known as CPU jiffies [30]). Bundling, however, does the exact same amount of work as on-time processing. Thus the question arises: *why does bundling save energy in spite of processing all the events?*

Assumption: A CPU jiffy is an assigned CPU time slot for a process in Linux [30], [46]. More CPU jiffies for a process causes more CPU jiffies for the kernel, because of more context switches between the user space and the kernel space. In on-time event processing system, each event requires at least one user CPU jiffy. This incurs at least one context switch and one kernel CPU jiffy. This effect, however, can be minimized with bundled processing. Batching of events minimizes the number of context switches and the number of CPU jiffies. If our assumption is correct, the energy efficiency of bundling is explainable. The number of CPU jiffies and context switches are almost linearly correlated with software energy consumption [30].

A. *RQ 5: Why do bundling and dropping save energy?*

To verify the above assumption, we have analyzed the AcrylicPaint app. Similar to Chowdhury *et al.* [30], we used the Linux proc file system. To capture the CPU jiffies used by an app, we used `/proc/pid/stat`. This also includes the kernel CPU jiffies used for that app. However, app (process) specific context switches can not be captured using such a file system. We captured the number of context switches from `/proc/stat` before and after running a test for an app. The difference is thus approximately the number of context switches for the app.

Figure 6 shows the result (10 measurements for each configuration). The number of CPU jiffies and context switches follow a similar pattern to the energy consumption of the AcrylicPaint's versions (Figure 4). This observation suggests that our assumption is true: bundling indeed reduces the number of CPU jiffies and context switches. This also indicates that bundling and dropping enable efficient resource usage, and thus can potentially provide similar energy savings for platforms other than Android. The mechanisms of context switches between the kernel and user space are similar across different platforms and architectures.

Findings: Bundling and dropping access resources in efficient ways—reducing the need for many context switches, leading to energy efficient software. This observation suggests that the energy efficiency of bundling and dropping is not restricted to Android systems, but also is applicable in other platforms.

VII. Maintainability Analysis

Developers and architects need to be concerned with the maintainability consequences of changes made to enhance any single aspect of a system's quality. Thus it is important

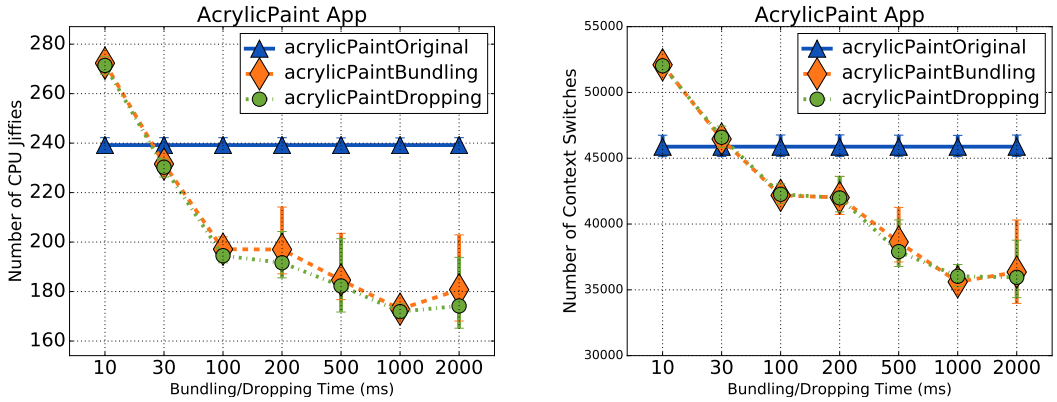

Fig. 6: Numbers of CPU jiffies and context switches for bundling and dropping compared with the original AcrylicPaint app. Bars indicate the 99% confidence interval.

to understand the consequences of the changes made to the apps to implement the bundling or dropping strategies. This motivates our sixth and final research question.

RQ 6: What are the maintainability consequences of implementing bundling and dropping on Android apps?

To assess these consequences we analyzed the before and after versions of the four selected real-world apps. We first reverse-engineered each of the apps using the *Understand* tool [65]. Using this tool we were able to collect code metrics on all versions of the apps. (For the purposes of brevity we only report on the original and bundling versions of the apps here. Results for the dropping versions were very similar.) In each case the modifications to the apps were slight in terms of effort, requiring fewer than 100 additional lines of code and at most three new classes (including one Handler class and one Thread class that runs the timer). But counting the lines of code and classes is just the measure of the required effort for converting a typical app version to a bundled version. Another important question is whether these changes affected the long-term maintainability of the system. If, for example, we added few lines of code but added many new dependencies between classes, this would increase coupling in the system, negatively affect the maintainability of the app going forward.

To determine whether this was the case we analyzed the coupling of the apps, in their before and after versions, using the Decoupling Level (DL) metric [11], a system-wide measure of coupling. The DL metric has been empirically validated [11] and shown to be more reliable than other coupling metrics such as Propagation-Cost [66] and Independence-Level [67] in predicting maintenance effort. DL scores range from 0 to 100, and the higher the number the better, as this indicates that the system's files are more highly decoupled and hence can be independently modified. The purpose of using this metric is to determine if the changes made to address energy efficiency significantly lowered the value of the DL metric. If so, this would mean that the maintainability of the system was negatively impacted by the energy-saving modifications.

The DL values of the before and after versions of four apps are shown in Table IV. While the values of the DL metric varied widely (indicating the inherent maintainability of the apps prior to our intervention) the changes for the apps due to the addition of bundling were small. The observed drops in DL scores were due to new relationships between classes that the bundling and dropping functionality required. But since the DL scores do not change dramatically (decreasing about 6% for AcrylicPaint, increasing 1% for Angulo, and staying the same for ColorPicker), this indicates that the tradeoffs made for energy efficiency were generally good ones—improving the energy efficiency of the apps while sacrificing little, if any, maintainability of the apps for the long term. In fact, in [11], it was noted that small variations in DL ($\leq$ 10%) are typically not meaningful.

TABLE IV: DL values for before/after versions of each app (bundling only).

App	DL Score Original	DL Score Bundling	DL Δ
Angulo	68%	69%	+1%
ColorPicker	17%	17%	+0%
AcrylicPaint	88%	82%	-6%
Sensor Readout	32%	30%	-2%

Findings: Energy efficiency is largely ignored during software maintenance [4]. One reason could be the difficulty in fixing energy bugs [68]. Bundling and dropping, however, are easy to implement and maintain.

VIII. Threats to Validity

External validity is hampered by the single version of the Android OS that we used on four Galaxy Nexus phones. Also, we do not know how many real-world apps can directly take advantage of the proposed bundling approach. The first threat is mitigated somewhat by using the *GreenMiner-2* with a different phone (Zenphone 2). To mitigate the second threat, we tried to select apps from different domains, and with the context-switching analysis we explained why bundling is energy efficient. This might help predicting what other types of apps can adopt GreenBundling.

Internal validity can be criticized for the way we calculated the number of context switches. Unlike the CPU jiffies, process-specific context switches are inaccessible using the `procfs` file system. The difference between after and before when running a process can be affected by other processes (e.g., garbage collection).

The Kruskal-Wallis test, although it does not assume any normality distribution about the data, still assumes that data in each group has similar skewness [69]. These threats are minimized by measuring each configuration 10 times and then showing the means, and confidence intervals. Construct and conclusion validity may also be questioned based on the tests scripts that we created for the real-world apps. It is not guaranteed that typical users of these apps would interact similar to the way our test scripts do. However, our test scripts exercise the main functionality of these apps: e.g., drawing measurement graphs and objects with the Sensor Readout app and the AcrylicPaint app respectively.

IX. Related Work

In recent years, developers have expressed more concerns about software energy consumption [70]. The software research community has been investigating several areas of this issue. Hasan *et al.* [15], Pereira *et al.* [16], and Manotas *et al.* [71] have presented recommendations for selecting energy efficient Java collections. Energy efficient color transformation in Android apps was proposed by Li *et al.* [13] and Agolli *et al.* [72]. Off-loading jobs [73], pre-fetching content [74], and enabling ad-blockers [43] have been found to save energy in some cases. Chowdhury *et al.* [18] suggested that `HTTP/2` servers are more energy efficient, from the clients' perspective, than `HTTP/1.1` servers. Energy efficient logging techniques for Android systems [39] have also been studied.

Other research has shown correcting code smells helps to improve energy efficiency [75]. In a similar vein, the impact of code obfuscation and refactoring on software energy consumption was studied by Sahin *et al.* [76], [77]. The energy change from code obfuscation is too small to notice, whereas refactoring can impact both positively and negatively.

Developers need to measure or estimate their apps' energy consumption. Hao *et al.* [23] proposed an instruction-based energy estimation model. Machine learning based models were proposed by Aggarwal *et al.* (GreenAdvisor [44]), Chowdhury *et al.* (GreenOracle [46] and GreenScaler [30]), and Pathak *et al.* [78]. Nucci *et al.* proposed PETrA [24] to estimate Android apps' energy consumption leveraging various Android tools.

Locating software energy bugs and hotspots automatically is another important research area. Wakelock-related energy bugs have been frequently reported by earlier studies [19]–[22]. Developers need to exploit tools and techniques to locate and solve such bugs [21]. Similar to the energy bugs, developers should also resolve energy hotspots [10]. Jabbarvand *et al.* [79] proposed a test-suite minimization approach focusing only on locating energy bugs. In their later work, the authors proposed an energy-specific mutation testing framework with high precision in detecting energy bugs [80].

This paper, however, focuses more on high-level design choices that can help developers writing energy efficient systems. To the best of our knowledge, this is the least explored area of software energy efficiency, and there is still a need for more research on this avenue. The closest to our work is the short study by Sahin *et al.* [31], where the authors investigated different existing design patterns and their energy consumption. In contrast to our work, that study lacks proper guidelines and cost analysis for making a design choice.

X. Conclusion & Future Work

In this work we show that an architectural choice, such as choosing a bundled MVP architecture, can improve the sustainability and energy consumption of a system without negatively impacting system maintainability. The consequence of this research means that architects and developers can (and should) make design decisions to address energy consumption before they start coding.

We have demonstrated the value of a bundled presenter in MVP by first benchmarking a generic MVP architecture and then by demonstrating that the energy improvements demonstrated in the benchmark were in fact realized on real-world apps that were refactored into bundled MVP architectures from more classical MVC architectures. A significant reduction of energy consumption can in fact be achieved. Furthermore we showed that these modified apps did not seriously affect the user experience, nor did the refactored versions suffer in terms of their eventual maintainability. Thus, the energy-savings that we achieved were truly win-win.

Our final message is this: fundamental architectural choices, such as the ones we have investigated in this paper, can have substantial effects on energy consumption. Although we demonstrated our results on MVP-based architectures, it is our hope and belief that developers and researchers can use this study to motivate similar studies, allowing them to address questions of energy consumption, and their consequent tradeoffs, at design time. We do not need to wait until the app is built to make these important design choices. In our future work, we want to evaluate the proposed bundling architectures on other smartphones than Android, and with real end-users for evaluating actual usability. We also want to evaluate other architectural patterns and architectural choices so that architects can predictably translate sustainability requirements into designs and into working systems.

Acknowledgments

Shaiful Chowdhury was supported by the Alberta-Innovates Technology Future PhD Scholarship. Dr. Hindle is supported by an NSERC Discovery Grant. Dr. Kazman was supported by U.S. National Science Foundation grant number 1514561. This research was also partially supported by JSPS KAKENHI Grant Numbers JP15H05306 and JP18H03222 and an "JSPS Invitational Fellowship".

REFERENCES

[1] V. Woollaston, "customers really want better battery life," http://www.dailymail.co.uk/sciencetech/article-2715860/, 2014, (last accessed: 2018-Jul-22).

[2] C. Wilke, S. Richly, S. Gtz, C. Piechnick, and U. Amann, "Energy consumption and efficiency in mobile applications: A user feedback study," in *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, Aug 2013, pp. 134–141.

[3] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Software*, vol. 32, no. 3, pp. 70–77, May 2015.

[4] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause, "An empirical study of practitioners' perspectives on green software engineering," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, 2016, pp. 237–248.

[5] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about software energy consumption?" *IEEE Software*, vol. 33, no. 3, pp. 83–89, May 2016.

[6] C. Sahin, L. Pollock, and J. Clause, "From benchmarks to real apps: Exploring the energy impacts of performance-directed changes," *Journal of Systems and Software*, vol. 117, pp. 307 – 316, 2016.

[7] Statista, "Number of smartphone users worldwide from 2014 to 2020 (in billions)," https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, (last accessed: 201-Aug-06).

[8] P. Poole, "Half of Us Have Computers in Our Pockets, Though You'd Hardly Know it," http://www.huffingtonpost.com/pamela-poole/smartphone-technology_b_2573671.html, (last accessed: 2018-Jul-22).

[9] Cisco, "Cisco visual networking index: Forecast and methodology, 2016-2021," https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html, (last accessed: 201-Aug-06).

[10] Banerjee, Abhijeet and Chong, Lee Kee and Chattopadhyay, Sudipta and Roychoudhury, Abhik, "Detecting Energy Bugs and Hotspots in Mobile Apps," in *FSE 2014*, Hong Kong, China, Novemeber 2014, pp. 588–598.

[11] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level: A new metric for architectural maintenance complexity," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, 2016, pp. 499–510.

[12] "Greenbundle: replication and extension," https://github.com/shaifulcse/GreenBundle-Data-Code.

[13] D. Li, A. H. Tran, and W. G. J. Halfond, "Making Web Applications More Energy Efficient for OLED Smartphones," in *ICSE 2014*, Hyderabad, India, June 2014, pp. 527–538.

[14] 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 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2010.

[15] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy profiles of java collections classes," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, 2016, pp. 225–236.

[16] R. Pereira, M. Couto, J. a. Saraiva, J. Cunha, and J. a. P. Fernandes, "The influence of the java collection framework on overall energy consumption," in *Proceedings of the 5th International Workshop on Green and Sustainable Software*, ser. GREENS '16, 2016, pp. 15–21.

[17] D. Li, Y. Lyu, J. Gui, and W. G. J. Halfond, "Automated energy optimization of http requests for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, 2016, pp. 249–260.

[18] S. Chowdhury, S. Varun, and A. Hindle, "Client-side Energy Efficiency of HTTP/2 for Web and Mobile App Developers," in *SANER '16*, Osaka, Japan, March 2016.

[19] Y. Liu, C. Xu, S. Cheung, and V. Terragni, "Understanding and detecting wake lock misuses for android applications," in *FSE 2014*, Seattle, WA, USA, Nov 2016.

[20] F. Alam, P. R. Panda, N. Tripathi, N. Sharma, and S. Narayan, "Energy optimization in android applications through wakelock placement," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014, pp. 1–4.

[21] X. Wang, X. Li, and W. Wen, "Wlcleaner: Reducing energy waste caused by wakelock bugs at runtime," in *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on*, Aug 2014, pp. 429–434.

[22] P. S. Patil, J. Doshi, and D. Ambawade, "Reducing power consumption of smart device by proper management of wakelocks," in *Advance Computing Conference (IACC), 2015 IEEE International*, June 2015, pp. 883–887.

[23] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating Mobile Application Energy Consumption Using Program Analysis," in *ICSE '13*, 2013, pp. 92–101.

[24] D. D. Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. D. Lucia, "Software-based energy profiling of android apps: Simple, efficient and reliable?" in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 103–114.

[25] A. Carroll and G. Heiser, "An Analysis of Power Consumption in a Smartphone," in *Proceedings of the USENIXATC'10*, 2010.

[26] A. Shye, B. Scholbrock, and G. Memik, "Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures," in *IEEE/ACM MICRO 42*, New York, NY, USA, December 2009, pp. 168–178.

[27] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L. K. John, "Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, ser. HPCA '02, 2002, pp. 141–150.

[28] J. Flinn and M. Satyanarayanan, "PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications," in *WMCSA '99*, New Orleans, Louisiana, USA, February 1999, pp. 2–10.

[29] M. Dong and L. Zhong, "Self-constructive High-rate System Energy Modeling for Battery-powered Mobile Systems," in *Proceedings of the MobiSys '11*, June 2011, pp. 335–348.

[30] S. Chowdhury, S. Borle, S. Romansky, and A. Hindle, "Greenscaler: training software energy models with automatic test generation," *Empirical Software Engineering*, Jul 2018.

[31] C. Sahin, F. Cayci, I. L. M. Gutirrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh, "Initial explorations on design pattern energy usage," in *2012 First International Workshop on Green and Sustainable Software (GREENS)*, June 2012, pp. 55–61.

[32] M. Fowler, "Gui architectures. 2006," *URL http://www. martinfowler. com/eaaDev/uiArchs. html*, 2007.

[33] M. Potel, "Mvp: Model-view-presenter the taligent programming model for c++ and java," *Taligent Inc*, p. 20, 1996.

[34] C. V. Lopes, *Exercises in programming style*. Chapman and Hall/CRC, 2016.

[35] Bohdan Samusko , "Model-view-presenter: Our choice of architecture for your android app," https://steelkiwi.com/blog/model-view-presenter-our-choice-of-android-app/, (last accessed: 2018-Aug-02).

[36] android-architecture, "Android architecture blueprints," https://github.com/googlesamples/android-architecture, (last accessed: 2018-Aug-02).

[37] Pulkit Sethi, "Xamarin application architecture," https://blog.kloud.com.au/2018/01/17/xamarin-application-architecture/, (last accessed: 2018-Aug-02).

[38] 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 *EuroSys '12*, Bern, Switzerland, April 2012, pp. 29–42.

[39] S. Chowdhury, S. Di Nardo, A. Hindle, and Z. M. J. Jiang, "An exploratory study on assessing the energy impact of logging on android applications," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1422–1456, Jun 2018.

[40] Y. Lyu, D. Li, and W. G. J. Halfond, "Remove rats from your code: Automated optimization of resource inefficient database writes for mobile applications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, Amsterdam, Netherlands, 2018, pp. 310–321.

[41] Android, "Processes and threads overview," https://developer.android.com/guide/components/processes-and-threads, (last accessed: 2018-Jul-22).

[42] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, "GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework," in *MSR 2014*, Hyderabad, India, May 2014, pp. 12–21.

[43] K. Rasmussen, A. Wilson, and A. Hindle, "Green Mining: Energy Consumption of Advertisement Blocking Methods," in *GREENS 2014*, Hyderabad, India, June 2014, pp. 38–45.

[44] K. Aggarwal, A. Hindle, and E. Stroulia, "Greenadvisor: A tool for analyzing the impact of software evolution on energy consumption," in *2015 IEEE ICSME*, Bremen, Germany, Sept 2015, pp. 311–320.

[45] S. Romansky, N. C. Borle, S. Chowdhury, A. Hindle, and R. Greiner, "Deep green: Modelling time-series of software energy consumption," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Shanghai, China, Sept 2017, pp. 273–283.

[46] S. A. Chowdhury and A. Hindle, "Greenoracle: Estimating software energy consumption with energy measurement corpora," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16, 2016, pp. 49–60.

[47] A. McIntosh, S. Hassan, and A. Hindle, "What can android mobile app developers do about the energy consumption of machine learning?" *Empirical Software Engineering*, Jun 2018.

[48] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods*. John Wiley & Sons, 2013.

[49] Android, "Android Debug Bridge," https://developer.android.com/studio/command-line/adb, (last accessed: 2018-Jul-22).

[50] Laerd, "Spearman's Rank-Order Correlation," https://statistics.laerd.com/statistical-guides/spearmans-rank-order-correlation-statistical-guide.php, (last accessed: 2018-May-11).

[51] T. Akidau, "The world beyond batch: Streaming 101," https://www.oreilly.com/people/09f01-tyler-akidau, (last accessed: 2018-AUG-05).

[52] J. Krystynak, "How to serve billions of web requests per day without breaking a sweat," https://www.infoworld.com/article/2868513/database/how-to-serve-billion-web-requests-per-day.html, (last accessed: 2018-AUG-22).

[53] J. Shore, "How many data sources in your apps? let me count the apis." https://searchcloudapplications.techtarget.com/blog/Head-in-the-Clouds-SaaS-PaaS-and-Cloud-Strategy/How-many-data-sources-Let-me-count-the-APIs, (last accessed: 2018-AUG-22).

[54] "Microsoft," https://docs.microsoft.com/en-us/windows/desktop/uxguide/progress-bars, (last accessed: 2018-Jun-02).

[55] F-Droid, "Sensor readout," https://f-droid.org/en/packages/de.onyxbits.sensorreadout/, (last accessed: 2018-Jun-02).

[56] F-droid, "Colorpicker," https://f-droid.org/en/packages/com.enrico.sample/, (last accessed: 2018-Jun-02).

[57] ——, "Angulo," https://f-droid.org/en/packages/eu.domob.angulo/, (last accessed: 2018-Jun-02).

[58] ——, "Acrylicpaint," https://f-droid.org/en/packages/anupam.acrylic/, (last accessed: 2018-Jun-02).

[59] "F-droid: Free and open source android app repository," https://f-droid.org/, (last accessed: 2018-May-22).

[60] L. Bao, D. Lo, X. Xia, X. Wang, and C. Tian, "How android app developers manage power consumption?: An empirical study by mining power management commits," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16, 2016, pp. 37–48.

[61] D. E. Krutz, M. Mirakhorli, S. A. Malachowsky, A. Ruiz, J. Peterson, A. Filipski, and J. Smith, "A dataset of open-source android applications," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15, 2015, pp. 522–525.

[62] X. Xia, E. Shihab, Y. Kamei, D. Lo, and X. Wang, "Predicting crashing releases of mobile applications," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016, pp. 29:1–29:10.

[63] Google Play, "Sensor readout," https://play.google.com/store/apps/details?id=de.onyxbits.sensorreadout, (last accessed: 2018-Jun-02).

[64] D. Pagano and W. Maalej, "User feedback in the appstore: An empirical study," in *21st IEEE International Requirements Engineering Conference (RE)*, July 2013, pp. 125–134.

[65] Scitools.com, "Understand," https://scitools.com/, (last accessed: 2018-Aug-18).

[66] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Management Science*, vol. 52, no. 7, pp. 1015–1030, July 2006.

[67] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna, "From retrospect to prospect: Assessing modularity and stability from software architecture," in *2009 Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture*, Sep 2009, pp. 269–272.

[68] S. A. Chowdhury and A. Hindle, "Characterizing energy-aware software projects: Are they different?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16, 2016, pp. 508–511.

[69] J. McDonald, "Kruskalwallis test: Handbook of biological statistics," http://www.biostathandbook.com/kruskalwallis.html, (last accessed: 2018-AUG-07).

[70] H. Malik, P. Zhao, and M. Godfrey, "Going green: An exploratory analysis of energy-related questions," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15, 2015, pp. 418–421.

[71] I. Manotas, L. Pollock, and J. Clause, "Seeds: A software engineer's energy-optimization decision support framework," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, pp. 503–514.

[72] T. Agolli, L. Pollock, and J. Clause, "Investigating decreasing energy usage in mobile apps via indistinguishable color changes," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, May 2017, pp. 30–34.

[73] A. P. Miettinen and J. K. Nurminen, "Energy Efficiency of Mobile Clients in Cloud Computing," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10, Boston, MA, USA, June 2010.

[74] N. Gautam, H. Petander, and J. Noel, "A Comparison of the Cost and Energy Efficiency of Prefetching and Streaming of Mobile Video," in *Proceedings of the 5th Workshop on Mobile Video*, ser. MoVid '13, Oslo, Norway, February 2013, pp. 7–12.

[75] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 115–126.

[76] C. Sahin, P. Tornquist, R. McKenna, Z. Pearson, and J. Clause, "How Does Code Obfuscation Impact Energy Usage?" in *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014.

[77] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?" in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2014.

[78] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained Power Modeling for Smartphones Using System Call Tracing," in *EuroSys '11*, Salzburg, Austria, April 2011, pp. 153–168.

[79] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware test-suite minimization for android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, 2016, pp. 425–436.

[80] R. Jabbarvand and S. Malek, "μdroid: An energy-aware mutation testing framework for android," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, pp. 208–219.