Green Mining: A Methodology of Relating Software Change to Power Consumption

WEB EDITION

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

Abstract

Power consumption is becoming more and more important with the increased popularity of smart-phones, tablets and laptops. The threat of reducing a customer's battery-life now hangs over the software developer who asks, "will this next change be the one that causes my software to drain a customer's battery?" One solution is to detect power consumption regressions by measuring the power usage of tests, but this is time-consuming and often noisy. An alternative is to rely on software metrics that allow us to estimate the impact that a change might have on power consumption thus relieving the developer from expensive testing. This paper presents a general methodology for investigating the impact of software change on power consumption, we relate power consumption to software changes, and then investigate the impact of static OO software metrics on power consumption. We demonstrated that software change can effect power consumption using the Firefox web-browser and the Azureus/Vuze BitTorrent client. We found evidence of a potential relationship between some software metrics and power consumption. In conclusion, we explored the effect of software change on power consumption on two projects; and we provide an initial investigation on the impact of software metrics on power consumption.

Keywords

power; power consumption; mining software repositories; dynamic analysis; sustainable-software; software metrics

I. Introduction

Change can be scary, but software often needs to change. With change comes new opportunities and new dangers. Given a mobile context, such as software on a smart phone or laptop, new dangers may present themselves as power-consumption regressions. A change to the power consumption profile of an application can have a significant impact on the end-users' ability to use and access their mobile device.

Power consumption in a mobile setting limits the length of time that a device can operate between charges. When there is no power left, the user cannot use their device and is often left stranded without the aide of their mobile device, even in an emergency.

In this paper we will investigate the effect of software evolution on power consumption. Ask yourself, when a developer changes an application, will that application consume more or less power? Will new events and notifications quickly use up the battery? Will improving one quality, like responsiveness, negatively affect power consumption? In many of these cases the developer will not know until they test their application, or release it to unsuspecting users. Testing an application for change in power consumption is time consuming, one has to design a regression test and run it multiple times for reliable readings. Furthermore as these tests are running, the system needs to be monitored in terms of power consumption or resource usage. Thus testing for power consumption is expensive and time consuming [1].

Our long term goal is to save developers time, and help alert them before they make a software change that negatively affects power consumption. In this paper we take an initial step to see if changes in OO metrics between software versions could be related to software power consumption.

This paper works towards the idea of *Green Mining* [2], an attempt to measure and model how software maintenance impacts a system's power usage. *Green Mining's* goal is to help software engineers reduce

the power consumption of their own software by estimating the impact of software change on power consumption. Concretely, we can give recommendations based on past evidence extracted by looking at each change in a *version control system* (VCS) and dynamically measure its effect on power consumption. *Green Mining* mixes the non-functional requirement of software power consumption with *mining software repositories* (MSR) research. *Green mining* is an attempt to leverage historical information extracted from the corpus of publicly available software to model software power consumption.

One might ask, why bother measuring power when you can measure CPU utilization? Most research [3]–[5] about software power consumption has focused on the CPU, but other resources such as memory, disk I/O [6], heat, and network I/O all impact software power consumption [1]. In a mobile setting, such as smart-phones and laptops, disk and network I/O can severely affect power consumption. Furthermore, mobile vendors such as Apple, Microsoft, Intel, and IBM, have demonstrated an interest in software power consumption by providing power-management documentation and tools [7]–[11].

Our research questions include:

- RQ1: What are the effects of a performance oriented branch on the power consumption of Firefox?
- RQ2: What software metrics are relevant to power consumption?

In this paper we will address some of the initial steps towards *green mining*: measuring the power consumption of multiple versions and multiple contiguous commits/revisions (commits in version control) of a software system and leveraging that information to investigate the relationship between software change and power consumption. Small gains in power consumption can have a large effect when multiplied across multiple users, systems, or hours.

This paper's contributions include:

- A clearly defined general methodology for measuring the power consumption of snapshots and revisions;
- A comparison of the power consumption of Firefox branches;
- A revision-by-revision analysis of Azureus/Vuze power consumption;
- An attempt to relate power consumption and software metrics.

This paper differs significantly from our ICSE NIER paper, *Green Mining: Investigating Power Consumption Across Versions* [2], which served as a call to arms for *Green Mining*. We do share some Firefox data and tests. In this paper we evaluate different branches of Firefox and investigate per revision power-consumption of the Vuze BitTorrent client. The previous work [2] did not analyze source code, or relate software metrics to power consumption.

II. Previous Work

Throughout this paper we will discuss power consumption in terms of watts. Watts are a SI unit of energy conversion, or power, equal to 1 Joule per second, named after James Watt. A useful reference point is that a 40W incandescent light-bulb consumes 40W when it is operating. To talk about cumulative consumption, we use a unit of watts over time, such as a watt-hour: a 40W bulb operating for 1 hour takes 40 watt-hours. We report mean-watts rather than watt-hours because per study each test takes an equal amount of time.

Industrial interest in software power consumption is apparent [1], [7]–[10]. Companies who produce mobile devices and mobile software have expressed their interest in terms of research funding, documentation, and tools. Google has funded Power Tutor [1], an Android power monitor. Microsoft provides Windows Phone documentation and research [7], [9] about how to reduce power consumption on the Windows Phone. Apple provides advice for improving iOS battery performance of applications [8]. Intel [10] has contributed to Linux in terms of PowerTop, a power-oriented version of UNIX `top`. IBM publishes documentation and sells tools that help reduce server room power consumption [11]. Industrial interest in software-based power consumption is evident as many of these companies have formed and joined the GreenIT [12] group.

Measurement: Power consumption must be measured and related to the software that induces this consumption. Gurumurthi et al. [13] produced a machine simulator called SoftWatt meant to simulate a power consuming device. This kind of virtualization is valuable as hardware instrumentation is expensive. Amsel et al. [4] have produced tools that simulate a real system's power usage, and benchmark individual applications' power usage. Gupta et al. [9] describe a method of measuring the power consumption of applications running on Windows Phone 7.

Tiwari et al. [3] modeled the power consumption of individual CPU instructions and were able to model power consumption based on traces of CPU instructions.

As power consumption is not just the CPU's fault, Lattanzi et al. [14] investigated the power consumption of WiFi adaptors and was able to produce an accurate model of WiFi power usage from a synthetic benchmark. Greenwalt [6] measured and modelled the power consumption attributes of hard-drives (HD), especially the HD seek times and power management timeouts.

In the mobile space PowerTutor [1] is an Android power monitor that augments ACPI power readings with machine learning to improve accuracy. Dong et al. [1] confirm that measuring power usage is intensive.

Kocher et al. [15] used power measurement to execute side-channel attacks on crypto-system implementations in order to expose key-bits. Other uses of measurement is to minimize power consumption.

Optimization: One goal of measuring power consumption is to optimize systems for reduced power consumption. Li et al. [16] applied the idea of load balancing to server-room heating and cooling. Fei et al. [5] used context-aware source code transformations and achieved power consumption reductions of up to 23% for their software. Selby [17] investigated methods of using compiler optimization to save power by reducing the load on the CPU by reducing branch prediction and staying in cache to avoid memory bus access. But power usage research has not leveraged the big-data corpus-based approaches used in MSR research.

Mining Software Repositories: Arguably any MSR work that is relevant to performance via traces is relevant to this work. For instance Shang et al. [18] have investigated performance over versions of software, in particular Hadoop. They evaluated the multiple versions against the stability of the log messages, while not quite a performance measure it required running multiple versions of the system to collect this date. To date there has not been much work, on combining MSR techniques with power performance, other than Gupta's [9] work on mining Windows Phone 7 power consumption traces.

Summary: We have demonstrated that there is much industrial and academic interest in power consumption but not a lot of focus on the effect of software change and software evolution on power consumption. Thus *Green mining* demonstrates novelty by combining MSR research and power consumption.

III. Methodology

In this section we present an abstract methodology for measuring and correlating power consumption of software snapshots and commits. We will also present the concrete methodology we used in the case studies to follow.

A. Green Mining Abstract Methodology

Within this section we will describe how to setup a green mining style power measurement experiment. This methodology is primary for the measurement and extraction of power consumption information relevant to software change. An overview of the general methodology is as follows:

- Choose a software product to test and what context it should be tested in.
- Decide on the level of instrumentation and the different kinds of data recorded, including power measurements.
- Choose a set of versions, snapshots or revisions, of a software product to test.
- Develop a test case for the software that can be run on the selected snapshots and revisions of the software.

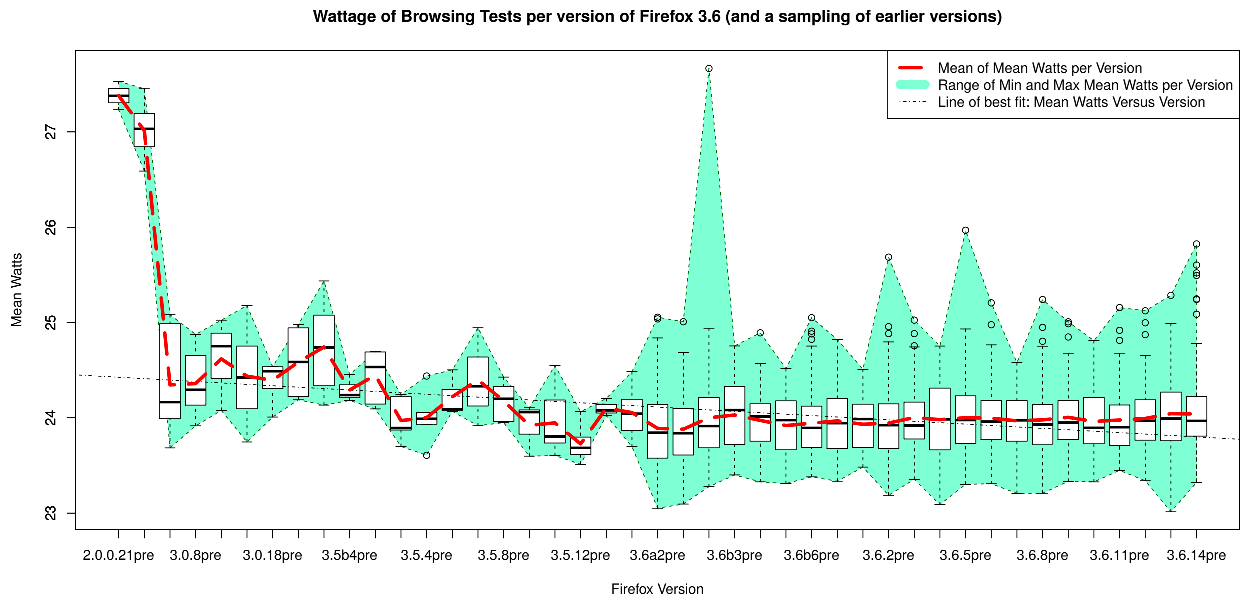

Fig. 1. This is a graph of the distributions of mean wattage (power consumption) of different versions of Firefox. The green-blue area is the range between the minimum and maximum mean wattage for that version. The red line is the mean of mean wattages and the box-plots depict the distribution of mean wattage per test per Firefox version. Note this plot depicts over 509 builds of Firefox 3.6 from alpha to stable versions. The dotted line with a negative slope is a line of best fit on the means; its slope indicates a decrease in power consumption across versions. The ranges for the earlier versions are smaller because we tested less instances of each of the earlier versions. This diagram depicts 2100 test runs. This figure shares data with our previous work that promoted and motivated Green Mining [2].

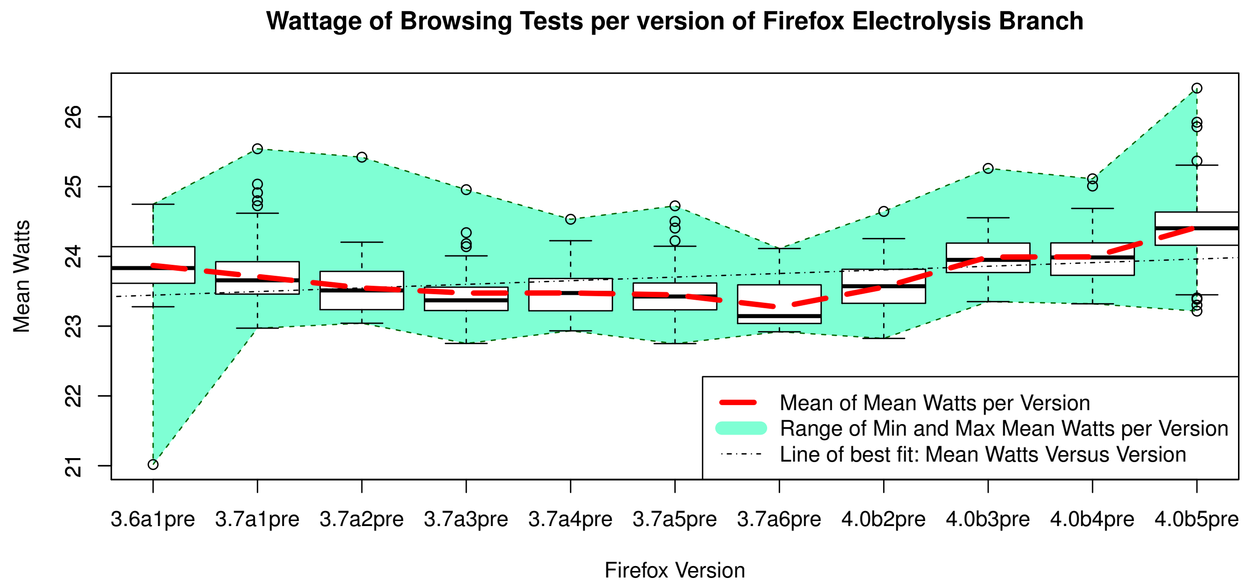

Fig. 2. Firefox Electrolysis Tests showing the distribution of mean wattage of different nightly builds and versions of Firefox Electrolysis. Following the same legend as Figure 1, this diagram shows a slow increase in power consumption but the mean wattages are less than the Firefox 3.6 main branch in Figure 1. This diagrams depicts 1500 test runs.

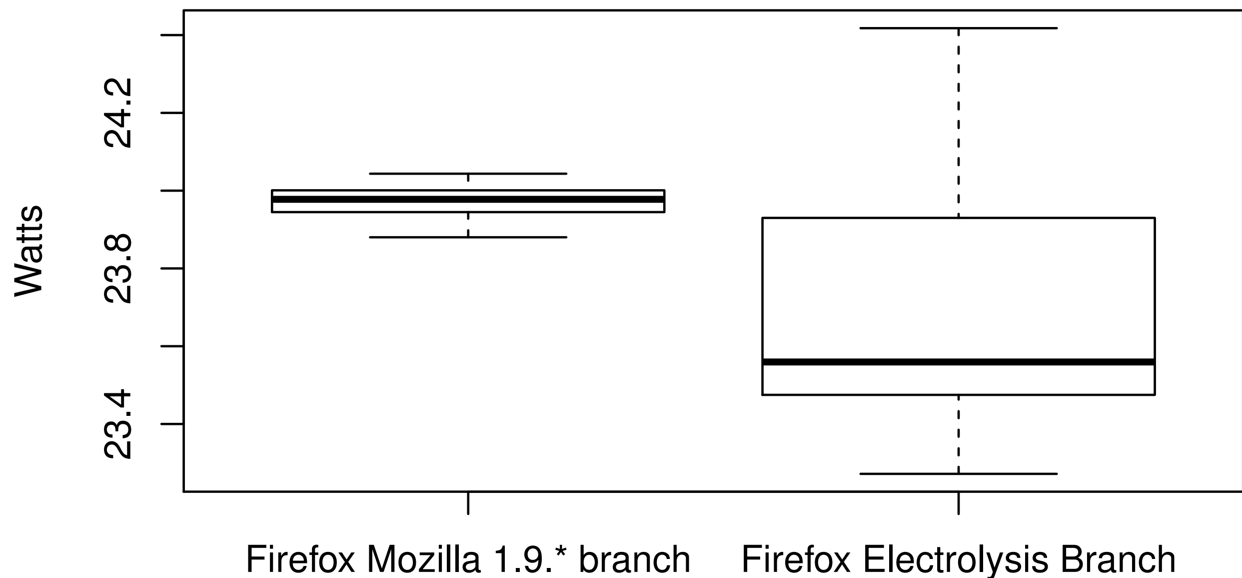

Fig. 3. Comparison of distribution of Firefox 3.6 mean watts and Firefox Electrolysis mean watts. Note that Firefox 3.6's mean is higher but the distribution is tighter.

- Configure the testbed system to reduce background noise from other processes.
- For every chosen version:
  - Run the test within the testbed and record the instrumented data.
  - Compile and store the recorded data.
  - Clean up the test and the testbed.
- Compile and Analyze the results

Choosing a product and a context: The first step is to choose a product and the context in which the product is going to be tested. It is important to consider the purpose of the test: is a particular feature being tested? Is new code being tested? Is the test meant to represent the average user using a product?
Decide on measurement and instrumentation: to decide on instrumentation one has to decide what they want to measure. If the test is going to be deployed on systems without power monitoring, does CPU, Disk I/O and Memory use need to be recorded in order to build models and make a proxy power consumption? On some systems, such as VMs, power cannot always be measured directly measured so other measures might have to serve as a proxy. One should consider the overhead of measurement in terms of CPU, I/O, and events and if any of this affects the power consumption of the system. Too much overhead results in a confounding observer effect which can skew results. Some aspects of a system can be measured without affecting the run-time system. For instance recording power consumption with a separate computer avoids the overhead of recording power use on the same machine.
Choose a set of versions: One must choose a range of versions of the software, snapshots or revisions/commits, to iterate over and test. If one wants to tease out the effects of software evolution one has to monitor the power consumption response of different versions of the software. Snapshots are either source-code or compiled releases of the software. Revisions or commits are the version control's snapshot of the system at a specific time or revision. Furthermore if compilation is needed, candidates chosen must be compiled. This kind of testing requires an executable binary, thus one has to account for building these VCS snapshots into their decision to use these snapshots and revisions.

If a version control system is used, one will have to walk through commits and determine candidate commits. For instance in the case studies we tried to compile every revision in the VCS with varying success.

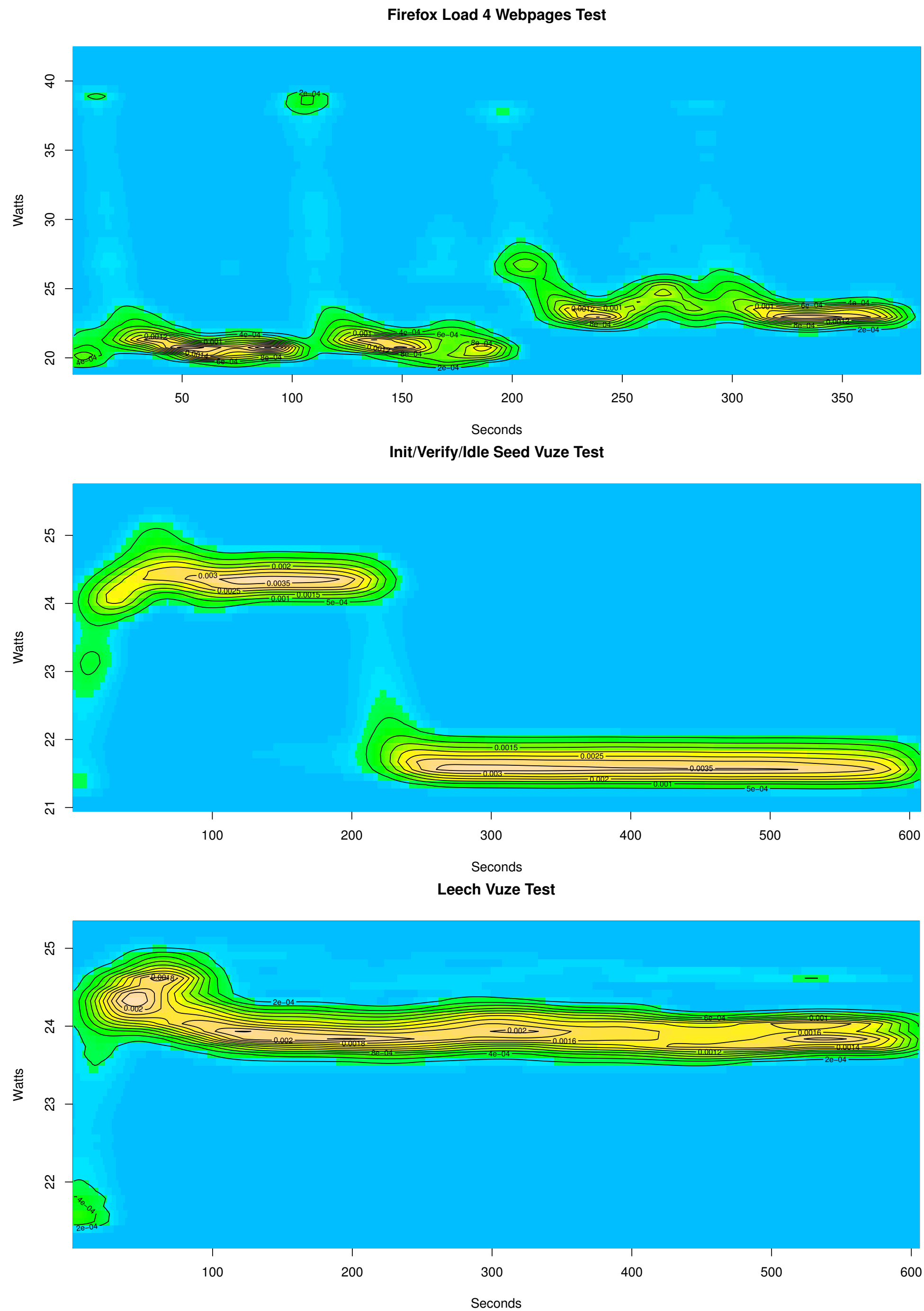

Fig. 4. Density of wattage measurements from Firefox tests, Idle Vuze tests and Leech Vuze tests. The density indicates how often a measurement was taken at a certain wattage at a certain second. This plot is effectively a trace of how thousands of tests ran in terms of watts.

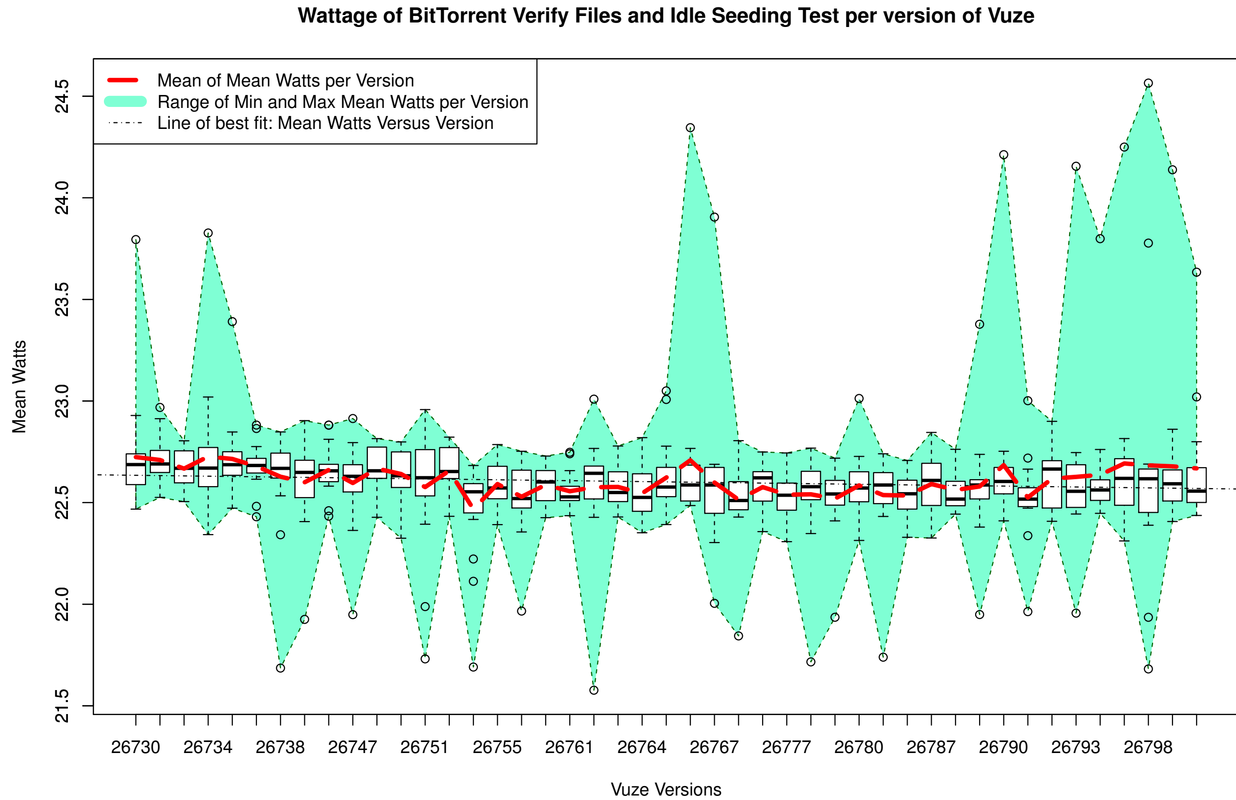

Fig. 5. Vuze Init, Verify and Idle test. This test shows the distribution of tests run on 45 revisions of Vuze, totaling 900 tests. The line of best fit has a weak negative slope across the versions. The test consists of Vuze starting up, checking file integrity, seeding the file and idling.

Developing a test case: Based on the context chosen at the beginning, a representative test case must be built that will exercise the necessary functionality of the target product. A test case might simulate user input, or focus on specific tasks of a system such as initialization. The test case is expected to be independent and clean up after itself. One test case should not affect the next. This can be difficult to achieve and might require monitoring a test case before one can be confident it has been addressed. Furthermore the test cases must deal with the evolution of the software, the same feature might be accessed in different ways depending on the version of the software. Tests must handle situations such as software that is run for the first time and prompts for input to help initialize the software.

Configure the testbed: the testbed will often include a full modern operating system. These modern operating systems have numerous services that execute automatically. These kinds of services add noise to measurements and should be disabled. Such interruptions include: automatic updating, disk defragmentation, virus scanning, CRON jobs, automatic updates, disk indexing, document indexing, RSS feed updates, twitter feeds, etc. Furthermore the system should be isolated in terms of other users and services that the testbed system provides. In terms of user interfaces it is useful to turn off screen blanking and screen savers. Window manager choice can matter as well, if a window manager is used that makes window placement predictable then tests are easier to develop, without UI predictability tests need to include more logic to handle UI exceptions. If the configuration and setup of the testbed can be automated then the results from these tests are far more reproducible.

Per each version: the version of the software should be unpacked and the tests run against this version, multiple times. Before the test starts the testbed must initiate all external instrumentation, such as memory, CPU, and disk I/O monitoring tools. Once the system is ready the test is initiated. Once the test is complete the information from the instrumentation, the power monitoring device, the external instrumentation and the logs are all recorded and packed up. This bundle of compiled information is then locally or remotely

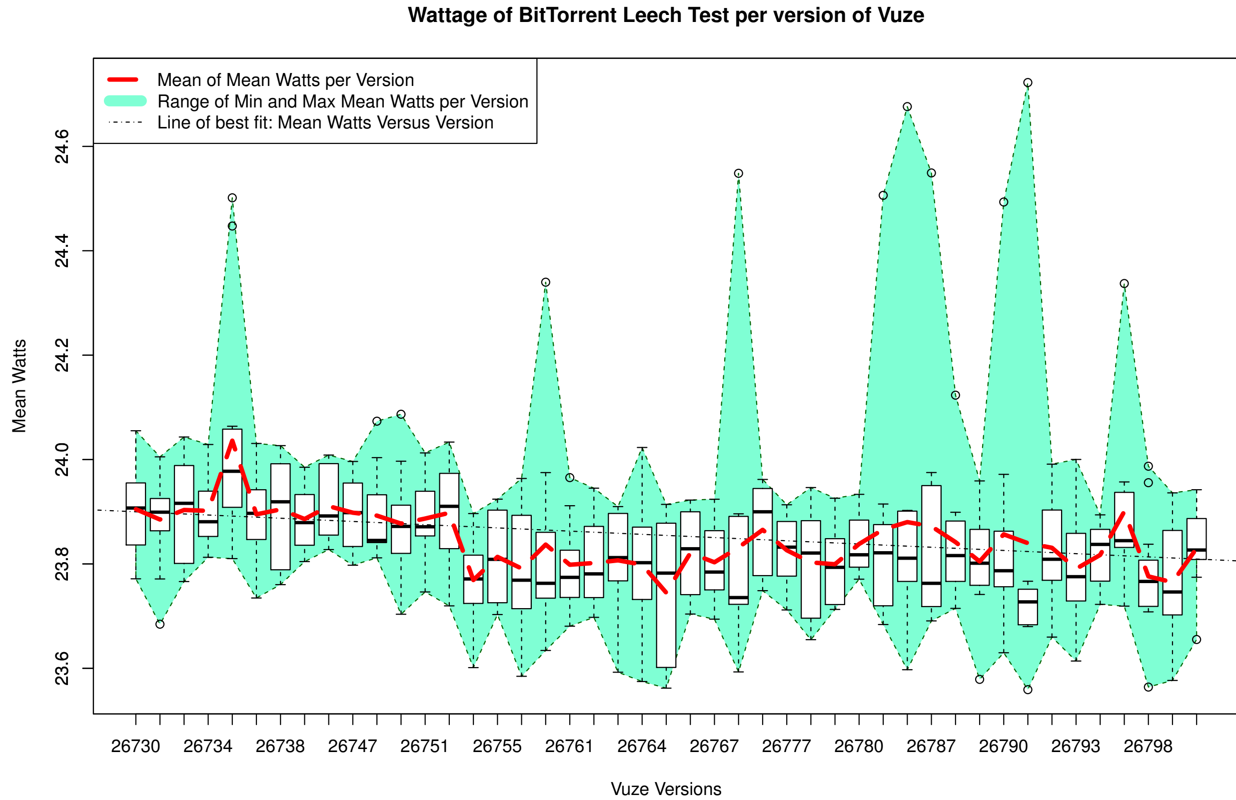

Fig. 6. Wattage of Vuze Leech Tests. This test shows the distribution of tests run over 45 revisions of Vuze, totaling more than 500 tests. The line of best fit has a stronger negative slope than Figure 5, across the versions. The test consists of Vuze starting up, initializing a file, and leeching the blocks from a seeder.

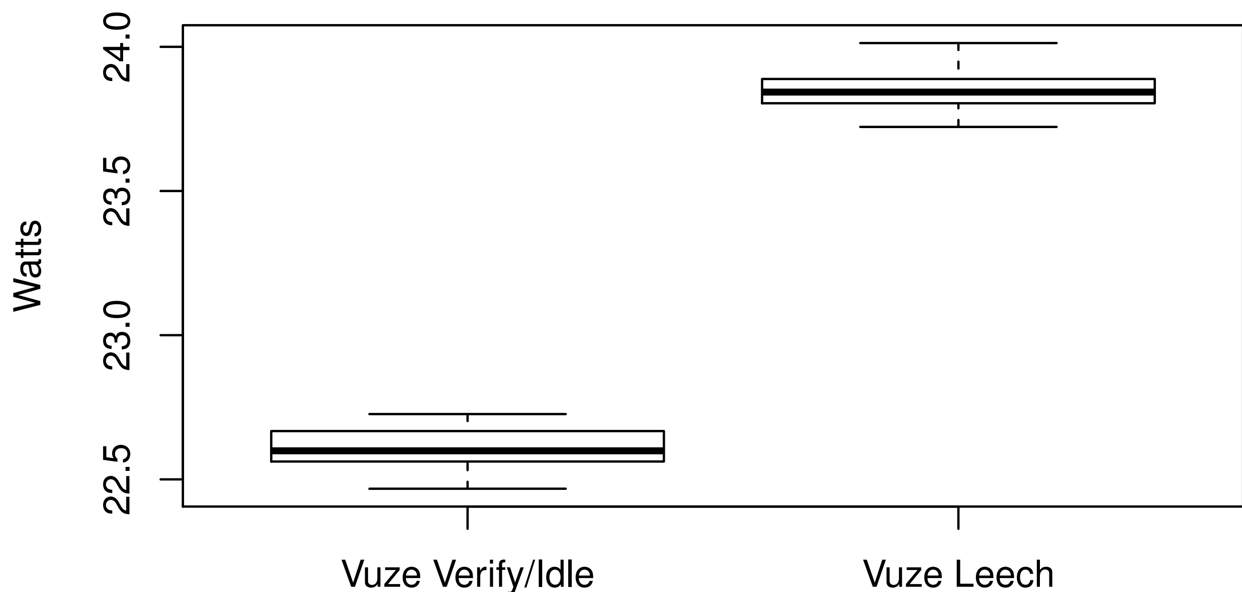

Fig. 7. Comparison of distribution of the Idle Vuze (Figure 5) tests mean watts and Leech Vuze test mean watts (Figure 6).

stored. Once all necessary information is extracted from the remains of the test-case, the test-case or testbed is meant to clean up after itself as to allow another test-case to run independently. Note for reliable results these tests should be run multiple times as it is unlikely that one has full control over every tiny minutia in the system, the testbed, and the tests. Multiple runs ensure reliability, but one should avoid multiple consecutive runs for fear of disk caching (restarting the system can alleviate this issue). When storing the results of each test one should record the necessary metadata which could include the machine-name, testbed identity, the current configuration, start and end time, the power monitor trace itself and summary statistics.
Compile and Analyze the results: once the tests are executed enough times one can analyze the results. It is useful to summarize each run by the number of watts consumed or mean watts.

B. Green Mining Concrete Methodology

In this section we instantiate the methodology of the previous section and we describe the concrete aspects of tests. We demonstrate the applicability of the previous methodology to our case studies.
Choosing a product and a context: in this paper we chose 2 products. The first product is Firefox, a popular C++ implemented consumer-oriented open-source web-browser maintained by the Mozilla foundation. The second product is Azureus, now known as Vuze, a popular Java-based Peer-to-Peer (P2P) BitTorrent client. Our Firefox testing context simulates the browsing behaviour of a mobile-user, and slightly exercises the animation and Javascript features of Firefox in the process. For Vuze we test 2 contexts: the first context tests the start-up cost of Vuze before it seeds a 2GB file, and then the cost of an idle Vuze seeding; the second context tests the cost of Vuze downloading a file from a seeder.
Decide on measurement and instrumentation: we decided to measure the power consumption of the system using an external AC power monitor called the *Watts Up? Pro*, a hardware device that measures wall socket power consumption (watts, kWh, amps, power-factor, volts, etc.), and reports power measures per second. We wanted to mine for information that would allow us to model power consumption and avoid hardware instrumentation, thus we used SAR [1] to record system activity information (CPU, Disk, Memory, Network, etc.). We combine both of these measures and synchronize them with timestamps.
Choose a set of versions: For Firefox we relied on nightly snapshots for the 2009–2010 nightly builds of Firefox with version ranging from 2.0 to 3.6, focusing mostly on 3.6 compilations for the main Mozilla branch and the Electrolysis branch for 3.6. Because the Firefox versions we had were binary snapshots we could immediately run them. For Vuze, we relied on 45 subversion revisions starting from revision 26730 on September 14, 2011 to revision 26801 on December 15, 2011 (3 months); between those 2 revisions we successfully compiled 45 versions.
Developing a test case: Within each case study's section we will explain each test case in more detail. For the Firefox test case we opened 4 different webpages and randomly scrolled through them as if we were a mobile user browsing the web and reading the webpages they downloaded. We had two Vuze tests, the first test checks Vuze's start-up, idling and file integrity check; the second test measures the power consumption of downloading a 2GB file from a seeder.
Configure the testbed: In all test cases we had the same testbed. The hardware we used in the case study included an IBM Lenovo X31 Thinkpad laptop running Ubuntu 11.04, with its battery removed, plugged into a *Watts Up? Pro* device for power monitoring. We did not use the battery because we did not have a method of recharging the battery automatically. We made a script that put the laptop into an aggressive power-saving mode that would simulate mobile use. We turned off screen-savers and screen-blankers for consistency. We turned off automatic software updates and the checking for such updates, we also disabled disk indexers. We disabled spurious cronjobs. We logged into X11 as the `greenminer` user, using the XMonad window manager, which was set to full-screen each window that popped up. The screen was left on during the tests.

[1] SAR and Sysstat http://pagesperso-orange.fr/sebastien.godard/

Per each version: for all versions of Firefox we unpacked them and ran tests on them multiple times, we describe these details in the Firefox case study Section IV-A. For the versions of Vuze we compiled, we unpacked and ran the tests multiple times on the executable, see Section IV-B. Each test run would upload the monitoring data to another server as to not impact the disk space of the testbed machine.
Compile and Analyze the Results: Our case studies contain the analysis of the results in terms of performance, power consumption and correlation with software metrics. In the end we had thousands of power traces and we had to aggregate this data meaningfully.

We seek to produce a huge corpus of software change correlated with power consumption behaviour by following this *Green Mining* vision and replicating this methodology over a large corpus of available software, version per version, revision per revision. The more measurements we have the better change that we can estimate the power consumption of software changes even without compilation or testing. In the next section we discuss our case studies which use this methodology.

IV. Case Study

Our case studies demonstrate an investigation of Firefox 3.6 and Vuze. The Firefox case study focuses on the power consumption of nightly compiled versions of Firefox, each run and tested many times. The Vuze case study investigates the fine grained version control commits by checking out a range of revisions or commits, compiling them, testing them, and then correlating their power consumption with software metrics. These case studies took more than 30 days to run.

A. *Firefox 3.6*

Our tests for Firefox were meant to simulate a mobile user browsing multiple webpages, and also to catch the cost of Firefox start-ups. For each page viewed we killed and restarted Firefox. Each page that we used was remotely hosted. For each page Firefox would load the page and then our UI driver would simulate a user scrolling through the webpage. To drive the UI we used X11::GUITest, a GUI testing framework. In order to generate the test of the user scrolling, we used X11::GUITest to record our own browsing session of reading and scrolling through a webpage. The intent was to produce a realistic browsing session with navigation keys such as up, down, page-up, page-down and mouse motions to catch messages that pop-up when we mouse over them.

The 4 different web-pages consisted of 2 Wikipedia pages, a mirror of the main-page and a page about the "Battle of Vukovar", and 2 NYAN-Cat pages (http://nyan.cat/) mirrored in different ways (but hosted remotely by us). The NYAN-Cat pages include GIF animations and client side Javascript animation. Testing all 4 pages took about 6 minutes.

While the Firefox tests ran we also had to check to see if the browser was in focus. Sometimes the browser would prompt to see if we wanted Firefox to be the default browser or to restore an older session, thus our GUI driver had to detect this and avoid this situation.

The Firefox binaries that we tested were "nightly builds" provided by Mozilla on their FTP site. Thus each binary tested was the accumulation of revisions for that day on that branch (mozilla-1.9.2 for Firefox 3.6). We did not evaluate *RQ2* for Firefox because we used "nightly builds" and did not have a similar metric suite for C++ code.

Figure 1 displays the results of 2131 runs of 43 different distinct versions of Firefox from version 2.0.0.21pre to version 3.6.14pre. Each version is a box-plot consisting of a set of nightly builds of Firefox, each run 3 or more times.

Figure 1 shows that Firefox 3.6 is relatively stable in terms of power consumption, but it can fluctuate between versions and measurements. The negative slope of the fitted line shows that there is a decrease in power consumption over time. For the 3.6 versions of Firefox it was relatively flat and stable. The difference in means between pre-3.6 Firefox versions and Firefox 3.6 versions is about 0.56 watts (T-test p-value of 0.012). Earlier versions of Firefox had higher mean power consumption, but also poorer general performance.

The first part of Figure 4 shows a density plot of the Firefox tests. Visible at the top of the figure are four yellowish peaks, each indicate a Firefox start-up for a webpage, which are heavy in terms of disk I/O, memory and CPU use. Most Firefox runs looked similar to other runs hence the high density. This plot, Figure 4, effectively summarizes the power consumption over time of 2131 separate runs of Firefox with our test. The last half of this plot shows elevated power consumption due to the GIF animation and Javascript animation used in the NYAN Cat tests. One take away from this is that webpages that are concerned about power usage should avoid Javascript with timed events and animations.

To summarize, Firefox was becoming more efficient over time and was consuming less power. Power consumption was relatively stable during the Firefox 3.6, but was up to 0.5 watts less than prior versions. This correlates with Firefox's pressure to achieve similar performance to that of Webkit-based competitors such as Chrome and Safari.

1) Firefox Electrolysis Branch: In parallel with mainline Firefox 3.6 development, a branch of Firefox was created called Electrolysis [2]. Electrolysis was an attempt to improve Firefox's performance and improve stability by allowing separate pages or tabs to exist as separate processes. Electrolysis also included some Javascript optimizations to deal with the pressure of HTML5 JS performance demonstrated by competing browsers at the time. One reason to test Electrolysis power consumption is to see if improved Firefox UI performance leads to increased Firefox resource usage and more power consumption. For example in an attempt to address UI fluidity and smoothness the Firefox developers could have added even more events and timers in order to improve responsiveness, thus this new version could potentially use more power.

We measured 1500 separate Firefox Electrolysis-branch tests (the same as the Firefox 3.6 tests) and compared them to Firefox 3.6. Figure 2 shows the plot of Electrolysis versions of Firefox compiled as nightlies. We can see that as Electrolysis was developed power consumption slowly increased. Yet when we compare the mean of the Electrolysis tests to that of the Firefox tests we can see that Electrolysis branch has reduced mean power consumption by about 0.27 Watts (T-test p-value of 0.023). We have plotted the difference in means of Firefox 3.6 and Firefox Electrolysis tests in Figure 3, we can see that electrolysis tests are clearly below most of Firefox 3.6's tests.

In response to our *first research question* (**RQ1**) the performance oriented branch, Electrolysis, of Firefox achieved a savings in power consumption ($0.27W$). We can see that intent behind Electrolysis was slowly being achieved, performance gains were happening and this was evident in reduced power consumption. Small savings in power consumption for popular software such as Firefox can quickly multiply and result in worldwide power savings.

B. Azureus/Vuze BitTorrent Case Study

The purpose of this case study is to investigate the relationship between the revision by revision change of Vuze with respect to power consumption. Furthermore we want to investigate the effect of code metrics on power consumption, thus we need coherent chunks of code to experiment with. Commits provide this appropriate coherency. The value of these kinds of tests is they allow us to relate fine grained incremental changes, the code of software revisions, to power consumption.

Vuze is a popular Java based open-source BitTorrent client. BitTorrent is a popular P2P file-sharing protocol often blamed for much IP infringement, but it is also an effective method of distribution for large legal files, such as the Ubuntu Linux CDs, as BitTorrent relies on the bandwidth of volunteers, who act as seeders. Seeders provide pieces of the file being shared to leechers who download these parts. Leechers can be seeders as well: BitTorrent tries to use game theory to make downloaders more willing to upload parts of files to other leechers.

BitTorrent clients are interesting because they are long running background processes, on desktops and mobile PCs, that share files by seeding and leeching torrents. Also, BitTorrent tries ensure random redundancy of its network by seeding blocks of files in random order to leechers. This is meant to protect

[2]Further details can be seen here: https://wiki.mozilla.org/Electrolysis

against the seeders disconnecting, and leaving the network of leechers without enough blocks to achieve 100% file block coverage. BitTorrent clients also use much cryptographic hashing for verifying that blocks are received. Thus the profile of a BitTorrent client is interesting, it mixes intermittent heavy CPU use, with intermittent heavy random I/O usage. Often a BitTorrent client has to verify file block integrity and entire files have to be integrity checked.

Vuze was chosen because it is a popular product, often appearing in the Source Forge top 10; implemented in Java, it often easy to compile, and runs on many machines. In this study we used multiple machines to attempt to compile all versions of Vuze. We achieved approximately 25% to 50% compilation coverage of the versions investigated. Of those we chose a 3 months worth of relatively recent commits that compiled without much issue: 45 compilations of subversion revisions starting from revision 26730 on September 14, 2011 to revision 26801 on December 15, 2011 (3 months). To compile all of these versions we made a flexible build script that tried to handle the different build methods (Apache Ant, javac, Eclipse) used to compile Vuze.

We then developed two sets of tests to be described in the next sub-sections: the first test investigates Vuze's start-up, idling and file integrity check; the second test measures the power consumption of leeching a 2GB file from a seeder. The same file was used in both tests, to avoid bandwidth savings due to compression we generated a 2GB file made of random noise exhibiting an entropy of 8 bits per byte.

Since the tests were about BitTorrent we had a server act as both the tracker and a seeder. The server and laptop communicated via a 100Mbit Ethernet cross-over cable.

For the Vuze tests we had a UI driver that pressed escape every second and closed Firefox if it started up. This was necessary because the versions of Vuze we were running tended to prompt for much input and would prompt to update the tested version of Vuze to a more recent version, or even try to open the Vuze blog in Firefox. All of these issues confounded our ability to run Vuze and thus our UI driver had to handle this. This can affect our testbed overhead since we create keyboard events every second.

For each test we evaluated the effort of OO software metrics on power consumption. Using the extended CKJM metrics suite [3] we measured the metrics of each version of Vuze tested and investigated the effect of various metrics on power consumption. Our primary tool is Spearman-rho rank based correlation. We chose rank based correlation because we hope to provide recommendations to programmers later, thus knowing that an increase in one measure leads to an increase in power consumption is quite valuable.Most of the CKJM metrics did not rank-correlate with any level of statistical significance so we will not discuss them, and those with enough statistical significance are unlikely to be significant after correcting for multiple hypotheses.

1) Azureus/Vuze Init, Verify, and Idle Test: Our first test was a relatively simple idle seeding test. BitTorrent clients upon start-up tend to verify the blocks in their downloaded files. In this test, the file was already placed in the download folder, and thus Vuze would verify the integrity of the file. After the integrity check, Vuze would register with the tracker (the computer serving the file as well) that it can share the file it just verified. Then the Vuze client would open up ports and wait for any communication. The Vuze client would also announce the availability of the file using the DHT function, but no one randomly generated the same file so this did not cause a problem.

Based on this test setup we ran the test between 17 and 21 times (median 20) for each version of Vuze that we compiled. Figure 5, shows a test with some variability but it is primarily centered around 22.6 watts per test. In terms of means, the mean power consumption has a tiny negative slope of -0.0015, but this more pronounced in the medians with a negative slope of -0.0024, arguably this is still quite flat, and the difference in means from the first and last versions is not statistically significantly different (T-test p-value of 0.59). But the tests between earlier revisions and the revisions 26753 to 26783 (the start of the 4700 release) tend to be statistically significantly different than revisions before and after (T-test p-values less than 0.05). This behaviour becomes more prominent in the next test. An investigation of the commit-log shows that many of the changes are UI hints, minor performance fixes and minor bug fixes.

[3]Extended CKJM Metrics: http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

This test's trace of wattage measurements is in the second sub-figure of Figure 4, it clearly shows the busy work at the start where Vuze verified the file, and then the system goes relatively idle after the file is verified. One interesting observation is that the power consumption of the Vuze tests is higher even while idle than the Firefox tests.

Software Metrics and Power Consumption: to address **RQ2** about software metrics for this test, change in mean watts correlated with a change in mean DAM ($\rho = -0.334$, $p = 0.0265$) and a change in mean MOA ($\rho = 0.315$, $p = 0.037$). Data access metric (DAM) is the ratio of private and protected attributes versus the total number of attributes in a class. Measure of aggregation (MOA) is the measure of count of fields in a class that derive from user provided classes. Total mean watts rank-correlated with polymorphic measures such as depth in inheritance tree (DIT) ($\rho = -0.37$, $p = 0.0125$) and number of children (NOC) ($\rho = 0.369$, $p = 0.0138$). These correlations are interesting and suggest that we need more data and more study to further tease out relationships if they exist.

2) Azureus/Vuze Leech Test: The Vuze leech test is meant to simulate a user downloading a file with BitTorrent from 1 primary seeder and no other leechers. In this test, the download directory is cleaned out and the Vuze application starts up ready to download the 2GB file described by the torrent file. Upon discovering there is no file yet, the client builds a sparse file on the file-system and then proceeds to contact the tracker. The tracker tells the client about the seeders and leechers on that torrent and announces this client's new membership. Then the client contacts the lone seeder and starts requesting blocks from it. We did not specify to Vuze to download blocks in order so it downloaded blocks in random order. This test required more network I/O and disk writing I/O than the idle test.

In total for the 45 versions of Vuze, we ran each test 10 to 15 times with a median of 12 times. Figure 6 shows the power consumption distribution of each version of Vuze tested. Interesting features of this test include the drop around revision 26753, that was visible in the previous test as well, shown in Figure 5. That power consumption is reduced in the dip, but it briefly spikes up again later. This test shows that the power consumption of Vuze seemed to be going down over 3 months and maybe that had something to do with fixes addressing resource use (memory leaks) and the mild performance improvements that were discussed in the commit-log. The slope of this plot is -0.0023, similar to the slope of the idle test's median line.

The trace of wattage measurements is in the third sub-figure of Figure 4, it clearly shows the busy work at the start where Vuze starts up, makes a new file, and then proceeds to download blocks of the file. In the tests the file tends to be entirely downloaded by the 6th to 8th minute and then its integrity is verified. But even when it is downloading, the power consumption remains high, it is has not settled down and perhaps the system has not had time to idle yet. The CPU use of the test was about 46% on average and there were many write transactions. It seems that both CPU and disk usage matter here. If we make a linear model of just percent user CPU utilization, we achieve an R^2 of 0.038 for this test with very tiny p-values, by adding transactions per second we achieve a R^2 value of 0.046, so some information is being provided by the disk usage (AIC also drops slightly). This indicates that the linear model poorly estimates power usage for this test, yet disk writes have an effect on power usage. This test probably lacks enough variation to tease out interesting relationships based on performance metrics.

Software Metrics and Power Consumption: to address **RQ2** for this test we used the same methodology as the last test for investigating OO software metrics related to these results. In this test, no metric correlated with statistical significance with the change in watts between versions. Mean Depth of Inheritance (DIT) ($\rho = -0.374$, $p = 0.011$) and mean number of children (NOC) ($\rho = -0.374$, $p = 0.011$) both rank-correlated with mean watts (and each other). Thus there was an interesting correlation between OO structural size metrics and mean watts. In the future, we want to test more Vuze versions in order to improve reliability.

C. Threats to Validity

This work faces numerous threats to validity, but we hope that our experiences, which we have presented in this paper, will allow others to carry out similar experiments with more reliability, accuracy and fewer threats.

Construct validity is threatened by the assumptions regarding the granularity or meaning of revisions/commits and snapshots. Fortunately measuring power consumption of a system is very concrete, but we suffer from the observer effect of whether or not we are measuring the software that we are testing is a concern.

Internal Validity: our OO metrics results are largely inconclusive because after p-value correction for multiple hypotheses these p-values would be insignificant. The measurements from the power monitor are not extremely accurate and wattage measurements are actually estimates, so there can be measurement error. Measurement error is a concern on modern systems, as well as the overhead of the testbed, often we are measuring its interaction as well.

External Validity: one obvious weakness of this approach is that the tests are very system specific. Generalizability could be improved with more test diversity and more systems; for instance some software has hardware specific optimizations. Unfortunately these tests are time consuming and difficult to setup, thus data is limited. Future work will address this by evaluating more software systems.

Reliability: our methodology is laid out clearly; others could replicate this study with their own equipment. The reliability of the OO metrics correlations are low. The purpose of this paper was to lay out this methodology and hence we feel this aides the reliability of the result.

V. Conclusions

In this paper we proposed a methodology of relating software change to software power consumption and we applied this methodology in 2 case studies on 2 distinct systems. In our case studies we observed the effects of intentional performance optimization within Mozilla Firefox and observed a steady reduction in power consumption of Firefox over time. We showed the the Electrolysis branch of Firefox 3.6, which was dedicated to improving Firefox performance with a multi-process model, achieved lower power consumption than the versions before it. With savings of $0.25W$, if 4 million users upgraded to the electrolysis branch there could be a savings of 1.0 Mega-Watt per hour worldwide; this is equivalent to saving an American household's monthly power use [19] per hour!

Our BitTorrent tests were executed on Azureus/Vuze across the actual revisions of the project, we showed that even with a small number of revisions that power relevant behaviour was visible. We then tried to relate OO software metrics, such as depth of inheritance and number of children to power consumption. We found evidence of some effect, but we found that this relationship depended greatly on the structure of the tests executed. Future work will include a more detailed evaluation of which kinds of changes lead to changes in power consumption.

Our case studies demonstrated the feasibility and the promise of the green mining methodology: measuring the power consumption of tests of multiple versions of a software system by combining power measurement and mining software repositories methodologies.

Acknowledgments: Thanks to Taras Glek of Mozilla, Andrew Wong, Philippe Vachon, and Andrew Neitsch.

References

[1] M. Dong and L. Zhong, "Self-constructive, high-rate energy modeling for battery-powered mobile systems," in *Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys)*, June 2011.

[2] A. Hindle, "Green mining: Investigating power consumption across versions," in *Proceedings, ICSE: NIER Track*. IEEE Computer Society, 2012, http://ur1.ca/84vh4.

[3] V. Tiwari, S. Malik, A. Wolfe, and M. Tien-Chien Lee, "Instruction level power analysis and optimization of software," *The Journal of VLSI Signal Processing*, vol. 13, 1996.

[4] N. Amsel and B. Tomlinson, "Green tracker: a tool for estimating the energy consumption of software," in *Proceedings, CHI EA*. New York, NY, USA: ACM, 2010, pp. 3337–3342.

[5] Y. Fei, S. Ravi, A. Raghunathan, and N. K. Jha, "Energy-optimizing source code transformations for operating system-driven embedded software," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 2:1–2:26, December 2007.

[6] P. Greenawalt, "Modeling power management for hard disks," in *MASCOTS '94., Proceedings of the Second International Workshop on*, Jan 1994, pp. 62 –66.

[7] C. Gray, "Performance Considerations for Windows Phone 7," http://create.msdn.com/downloads/?id=636, 2010.

[8] Apple Inc., "iOS Application Programming Guide: Tuning for Performance and Responsiveness," http://ur1.ca/696vh, 2010.

[9] A. Gupta, T. Zimmermann, C. Bird, N. Naggapan, T. Bhat, and S. Emran, "Energy Consumption in Windows Phone," Microsoft Research, Tech. Rep. MSR-TR-2011-106, 2011.

[10] Intel, "LessWatts.org - Saving Power on Intel systems with Linux," http://www.lesswatts.org, 2011.

[11] IBM, "IBM Active Energy Manager," http://www.ibm.com/systems/management/director/about/director52/extensions/actengmrg.html, 2011.

[12] S. Murugesan, "Harnessing Green IT: Principles and Practices," *IT Professional*, vol. 10, no. 1, pp. 24–33, 2008.

[13] S. Gurumurthi, A. Sivasubramaniam, M. Irwin, N. Vijaykrishnan, and M. Kandemir, "Using complete machine simulation for software power estimation: the SoftWatt approach," in *Proc. of 8th Int. Symp. High-Performance Computer Architecture*, Feb 2002.

[14] E. Lattanzi, A. Acquaviva, and A. Bogliolo, "Run-Time Software Monitor of the Power Consumption of Wireless Network Interface Cards," in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, vol. 3254, pp. 352–361.

[15] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology CRYPTO 99*, ser. Lecture Notes in Computer Science, M. Wiener, Ed. Springer Berlin / Heidelberg, 1999, vol. 1666, pp. 789–789.

[16] L. Li, C.-J. M. Liang, J. Liu, S. Nath, A. Terzis, and C. Faloutsos, "Thermocast: A cyber-physical forecasting model for data centers," in *Proceedings, ACM SIGKDD*. ACM, 2011.

[17] J. W. A. Selby, "Unconventional applications of compiler analysis," Ph.D. dissertation, University of Waterloo, 2011.

[18] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. N. Nasser, and P. Flora, "An exploratory study of the evolution of communicated information about the execution of large software systems," in *WCRE*, 2011, pp. 335–344.

[19] US Department of Energy, "Energy explained: Electricity," http://www.eia.gov/energyexplained/index.cfm?page=electricity_home#tab2, Jan 2012.