Green Mining: A Methodology of Relating Software Change and Configuration to Power Consumption – WEB EDITION

Abram Hindle

March 3, 2014

Abstract

Power consumption is becoming more and more important with the increased popularity of smartphones, tablets and laptops. The threat of reducing a customer's battery-life now hangs over the software developer, who now 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 OO software metrics and churn 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. We also investigate the effect of library versioning on the power consumption of rTorrent. In conclusion, we investigate 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.

1 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, whether it be a phone or a laptop.

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. Questions that should be addressed include: 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 [3].

This paper works towards the vision of *Green Mining* [11], 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 based on empirical evidence. Concretely, we seek to 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. Thus *Green mining* is an attempt to leverage historical information extracted from the corpus of publicly available software to model software power consumption. This paper explores the feasibility of power consumption prediction based on OO metrics and churn metrics.

One might ask, why bother measuring power when you can measure CPU utilization? Most research [1, 4, 27] about software power consumption has focused on the CPU, but other resources such as memory, disk I/O [7], temperature and cooling, and network I/O all impact software power consumption [3]. 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 [2, 6, 8, 12, 13].

Our research questions include:

- RQ1: Does power consumption evolve over time?
- 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. We will discuss 2 of our previous cases [10] and 1 new case study from the perspective of the pitfalls faced when one has to investigate software change and software power consumption together.

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.
- An investigation and analysis of the pitfalls relevant to measuring the power consumption of software change via three case studies.
- A case study of multiple versions of a library (`libTorrent`) and its effect on power consumption of its client application (`rTorrent`).
- The first study of the effect of churn on power consumption.

This paper discusses 3 case studies, 2 of which have already been published, but with less emphasis on the difficulty of measurement. This paper differs significantly from our ICSE 2012 NIER paper, *Green Mining: Investigating Power Consumption Across Versions* [11], which served as a call to arms for *Green Mining*, as we have more case studies and a more thorough analysis. This paper extends our MSR 2012 paper, *Green Mining: A Methodology of Relating Software Change to Power Consumption* [10], that illustrates our methodology and depicted power consumption tests across Firefox 3.6 and Azureus Vuze. As well we discuss difficulties faced during that work. We also cover a new case-study on library variation, churn, and power consumption.

2 Power

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. Thus watt is an instantaneous rate of power consumption. 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 (average power), the mean of watt measures over time rather than watt-hours because per each study the tests takes an equal amount of time and thus will have only differ by their average power consumption (e.g, 32 measures of 40 watt each taken 1 second apart will be 1280 watt seconds and 40 mean watts).

If these tests were on tasks that do work and immediately terminate, such as compilation, or rendering, that did not have idle behaviour, such as a user reading a screen worth of text on a web browser, then reporting Joules or watt-hours would be more appropriate.

3 Previous Work

Industrial interest in software power consumption is apparent [2,3,6,8,13]. 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 [3], an Android power monitor. Microsoft provides Windows Phone documentation and research [6,8] about how to reduce power consumption on the Windows Phone platform. Apple provides advice for improving iOS battery performance of applications [2]. Intel [13] 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 [12]. Industrial interest in software-based power consumption is evident as many of these companies have formed and joined the GreenIT [19] group.

Measurement: Power consumption must be measured and related to the software that induces this consumption. Gurumurthi et al. [9] produced a machine simulator called SoftWatt meant to simulate a power consuming device. This kind of virtualization is valuable as hardware instrumentation is expensive. This kind of simulation also avoids the observer effect because the system under test is not self-instrumented, so the load of the instrumentation is not felt within the system, but outside the system in the simulator. Amsel et al. [1] have produced tools that simulates a real system's power usage based on CPU measurements, and benchmark individual applications' power. They do not avoid the observer effect because they have to measure the system under test as it is running. Gupta et al. [8] describe a method of measuring the power consumption of applications running on Windows Phone 7. They avoid the observer effect though hardware instrumentation while repeatedly running benchmarks of calls.

Tiwari et al. [27] 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. [17] investigated the power consumption of WiFi adapters and were able to produce an accurate model of WiFi power usage from a synthetic benchmark. Lattanzi derived and compared their model to power measurements of synthetic benchmarks. They instrumented the wireless network interface and recorded its power use externally. This kind of work is especially important in the context of networked mobile computing because it allows modelling of the environment before deployment. Greenwalt [7] modelled the power consumption attributes of hard-drives (HD), especially the HD seek times and power management timeouts. Greenwalt [7] derived his model from the properties of disk accesses and harddrives but did not measure actual harddrives them.

In the mobile space PowerTutor [3] is an Android power monitor that augments ACPI power readings with machine learning to improve accuracy. Dong et al. [3] confirm that measuring power usage is resource and effort intensive.

Industrially, Powertop [13] is popular for power consumption monitoring because it reports the processes and drivers that produce events and wakeups, as well it estimates power use from the ACPI (the power management system interface used on modern computers). Powertop is recommended by Michael Larabel [16] who also benchmarked multiple versions of Ubuntu against each other given idle and load tests.

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

Optimization: One goal of measuring power consumption is to optimize systems for reduced power consumption. Li et al. [18] applied the idea of load balancing to server-room heating and cooling. Fei et al. [4] used context-aware source code transformations and achieved power consumption reductions of up to 23% for their software. Selby [21] 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: This work is relevant to the field of mining software repositories (MSR) because it investigates the changes in a resource usage or performance, namely power consumption, over revisions and snapshots. These revisions and snapshots come from version control systems, continuous integration archives, and other kinds of software repositories. Furthermore this work hopes to motivate the production new software repositories of power-traces in order to enable us to reason about software based power consumption without the heavy lifting of the testing we executed in this paper. Thus any MSR work that is relevant to performance via traces is relevant to this work. For instance Shang et al. [22] 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 [8] 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.

4 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.

4.1 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:

1. Choose a software product to test and what context it should be tested in.
2. Decide on the level of instrumentation and the different kinds of data recorded, including power measurements.
3. Choose a set of versions, snapshots or revisions, of a software product to test.
4. Develop a test case for the software that can be run on the selected snapshots and revisions of the software.
5. Configure the testbed system to reduce background noise from other processes.

6. For every chosen version and configuration:
    - Run the test within the testbed and record the instrumented data.
    - Compile and store the recorded data.
    - Clean up the test and the testbed.
7. 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 with or 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 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, which logs the readings from a power monitor of computer under test, 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 might have to be compiled or built . This kind of testing requires an executable binary or package, thus one has to account for building these VCS snapshots when they decide to use these snapshots and revisions.

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

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 or require a running, 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: disk defragmentation, virus scanning, CRON jobs, automatic updates, disk indexing, document indexing, RSS feed updates, Twitter updates, 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 and configuration: the version of the software should be unpacked, configured, 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 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 meta-data 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 energy consumed (watt-seconds) or mean watts.

4.2 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 3 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. The third product is `rTorrent`, much like Vuze, it is a BitTorrent client but it is meant to run on UNIX shells. `rTorrent` allowed us to share some of the BitTorrent testing framework built around Vuze. 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, the Vuze *leech* (a BitTorrent downloader) test, tests the cost of Vuze downloading a file from a seeder. We test `rTorrent` using the Vuze *leech* test.

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* [1], a hardware device that measures wall socket AC power consumption (watts, kWh, amps, power-factor, volts, etc.), and reports power measures per second. The *Watts Up? Pro* is a general purpose device, unaware that is measuring a computer. SAR [5] was used to record system activity information (CPU, Disk, Memory, Network, etc.) because we wanted to mine for information that would allow us to model power consumption and avoid hardware instrumentation. 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 versions ranging from 2.0 to 3.6, focusing mostly on 3.6 compilations for the main Mozilla Firefox branch and the Electrolysis branch (a branch meant to explore improving UI latency). 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,

[1]For more information on the Watts Up? Pro see `https://www.wattsupmeters.com/secure/products.php?pn=0&wai=837&spec=4`

2011 (3 months); between those 2 revisions we successfully compiled 45 versions. For `rTorrent` we chose 18 snapshot versions that we could compile between `rTorrent` version 0.3.0 (2005) to `rTorrent` 0.8.9 (2011), and `libTorrent` versions 0.6.4 (2005) to 0.13.0 (2011).

Developing a test case: Within each case study's section we will explain each test case in more detail. The test case is the setup and actions executed again an application. 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. `rTorrent` and `libTorrent` were run using a leech test similar to the Vuze tests.

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 (enable CPU Governor and `POWERSAVE` mode in ACPI daemon) 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 and 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.

Per each version and configuration: 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 5.1. For the versions of Vuze we compiled, we unpacked and ran the tests multiple times on the executable, see Section 5.2. For `rTorrent` 5.3 we compiled every available combination of snapshots we could (40 versions) and ran the tests on each combination, these combinations are an example of different configurations. 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 we can estimate the power consumption induced by software changes without compilation or testing. In the next section we discuss our case studies which use this methodology.

5 Case Studies

Our case studies demonstrate an investigation of Firefox 3.6, Vuze and `rTorrent`. 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. The `rTorrent` case study investigates the effect that linking to different versions of the `libTorrent` library has on power consumption. These case studies took more than 30 days to run. Table 1 summarizes the case studies used in this study and the general findings, as well as the number of tests per each scenario.

5.1 Firefox 3.6

Firefox is a popular open-source web-browser developed and published by the Mozilla Foundation. Firefox is a very large software application, thus instead of compiling each version ourselves, we relied on nightly

	Firefox	Firefox: Electrolysis	Vuze: Idle	Vuze: Leech	rTorrent
Binaries	509	482	45	45	18
Releases	43	11	3	3	18
Lib/App Combos	N/A	N/A	N/A	N/A	40
Tests	2131	1500	900	500	294
Source	Nightlies	Nightlies	Subversion	Subversion	Tarballs
Change in Power	Decrease	Decrease	Similar	Decrease	Similar
Focus on Performance	Some	Yes	Minimal	Minimal	Some
Networked	Yes	Yes	Yes	Yes	Yes
Remote Network	Yes	Yes	Maybe DHT	Maybe DHT	No
Stores Significant Configuration Information	Yes	Yes	Yes	Yes	No
Location of Configuration Changed Over Time	Yes	Yes	Yes	Yes	No
GUI Popups	Yes	Yes	Yes	Yes	No
Compiler Incompatibility	N/A	N/A	No	No	Yes
Build System Changed	N/A	N/A	Yes	Yes	No
External Library Dependencies Changed	N/A	N/A	Could Have	Could Have	Yes
Could Not Compile All Versions	N/A	N/A	Yes	Yes	Yes
Language	C++ and JavaScript	C++ and JavaScript	Java	Java	C++

Table 1 – A Summary of the Case Studies and issues encountered. "Maybe DHT" means that the BitTorrent client could call out to the internet, but our file was unique so there will be no seeders. "Tarballs" are .tar.gz archives of source code. "Nightlies" are evening compilations of Firefox from their version control system. "Releases" are publicly named versions such as 3.0b1 and 5.1.1 while "Binaries" are successful builds that were tested. Focus on performance indicates if we observed any attempts to improve performance.

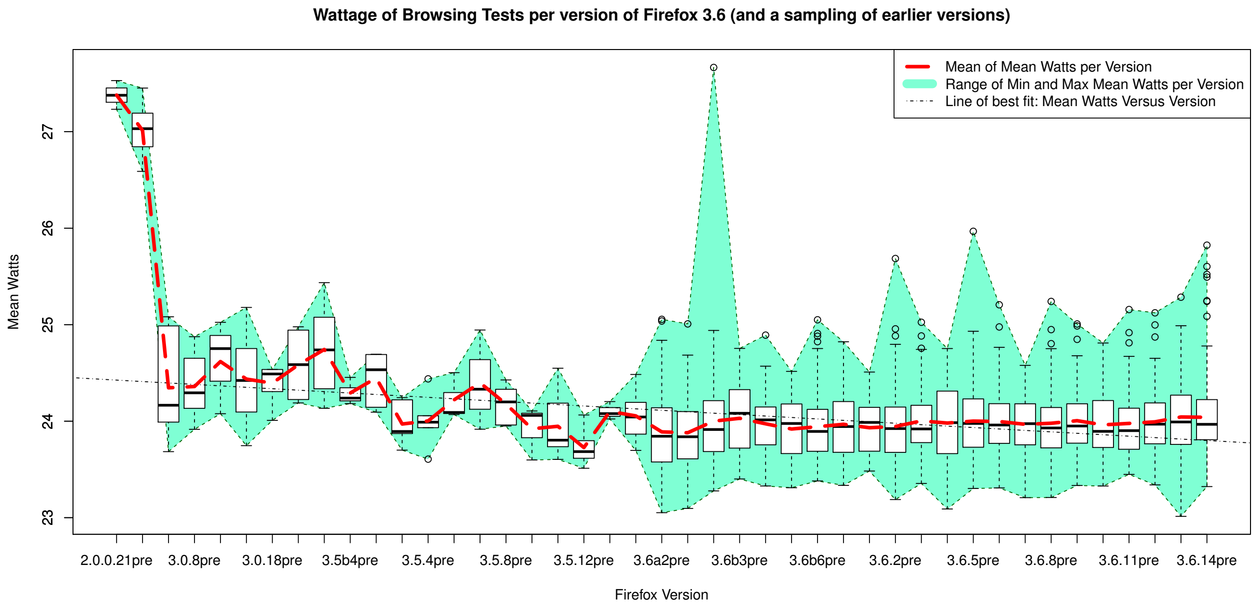

Figure 1 – This is a graph of the distributions of mean wattage (power consumption) of different versions of Firefox. The green-blue area is the range between the minimum and maximum mean wattage for that version. The red line is the mean of mean wattages and the box-plots depict the distribution of 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 2131 test runs. This figure shares data with our previous work that promoted and motivated Green Mining [11].

builds. 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* in terms of OO-metrics on Firefox because we used "nightly builds" and did not have a similar metric suite for C++ and Javascript code as we did for Java code. Near the end of Firefox 3.6 the most common type of source file in the repository was Javascript. This was further complicated by multiple branches of Firefox being developed at once so we focused on the main branch of Firefox 3.6 (depicted in Figure 1 [10]).

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.

Firefox is a large desktop application and it has a GUI. This GUI needs to run in order to test Firefox, thus our first task was to make Firefox visible to the windowing system. We would start Firefox with all the display variables set so it would start on the appropriate display. We encountered numerous GUI related issues that the GUI driver had to address, these are discussed in Section 8.3.

Firefox also creates, caches and saves a lot of information about the user's preferences, settings, cookies, plug-ins and extensions, history, and browser cache. Thus to be fair, each run of Firefox had to delete the `/.mozilla` data directory created by Firefox.

Figure 1 displays the results of 2131 runs of 43 different distinct releases 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. The number of success runs vary because of random failures such as failure to connect to the watts up? pro, old tests that did not get shutdown, or an erroneous run.

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 2 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 2, 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 causing their pages to produce more events. Sources of events include Javascript with timers, and animations from gif files, flash animations and Javascript mouse-overs.

To summarize, Firefox was becoming more efficient over time and was consuming less power. Power consumption was relatively stable during 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 as discussed in the next section.

5.1.1 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) over 482 different binaries consisting of 11 distinct releases, and compared them to Firefox 3.6. Figure 3 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 4, 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 ($0.27W$), showing that power consumption can indeed change over time, and often as a side effect of other behaviour. We can see that intent behind Electrolysis was slowly being achieved, performance gains were happening and this was evident in reduced power consumption even without explicitly addressing power consumption. Small savings in power consumption for popular software such as Firefox can quickly multiply and result in worldwide power savings.

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

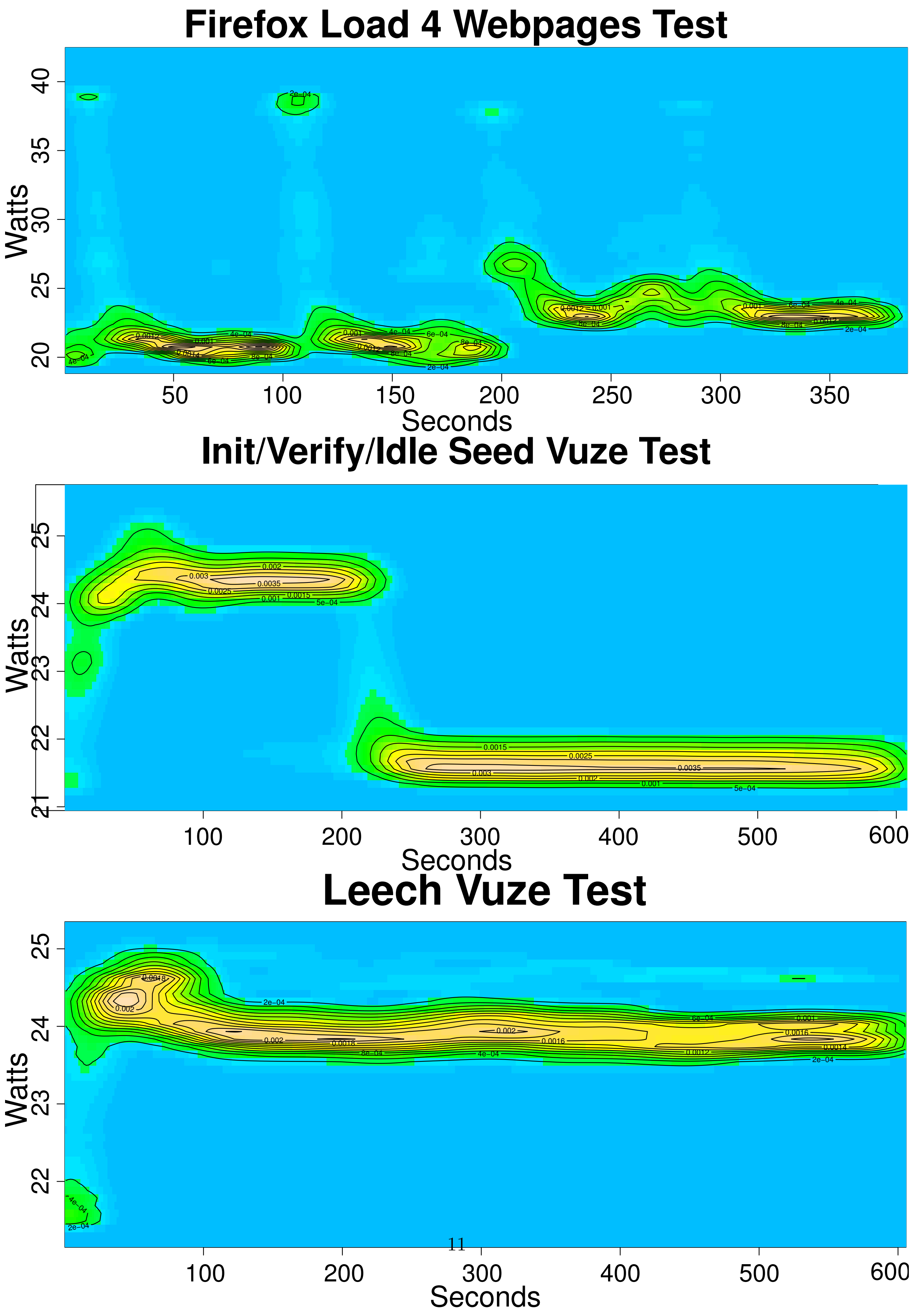

Figure 2 – Density of wattage measurements from Firefox tests, Idle Vuze tests and Leech Vuze tests. The density indicates how often a measurement was taken at a certain wattage at a certain second. This plot is effectively a trace of how thousands of tests ran in terms of watts.

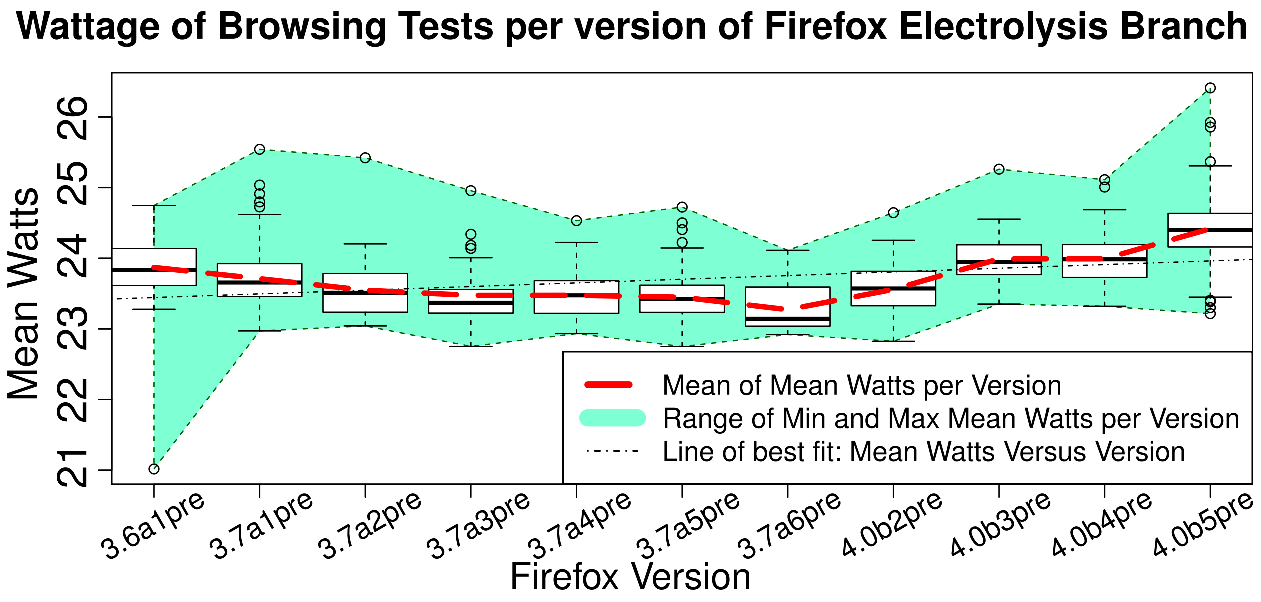

Figure 3 – Firefox Electrolysis Tests showing the distribution of mean wattage of different nightly builds and versions of Firefox Electrolysis. Following the same legend as Figure 1, this diagram shows a slow increase in power consumption but the mean wattages are less than the Firefox 3.6 main branch in Figure 1. This diagrams depicts 1500 test runs.

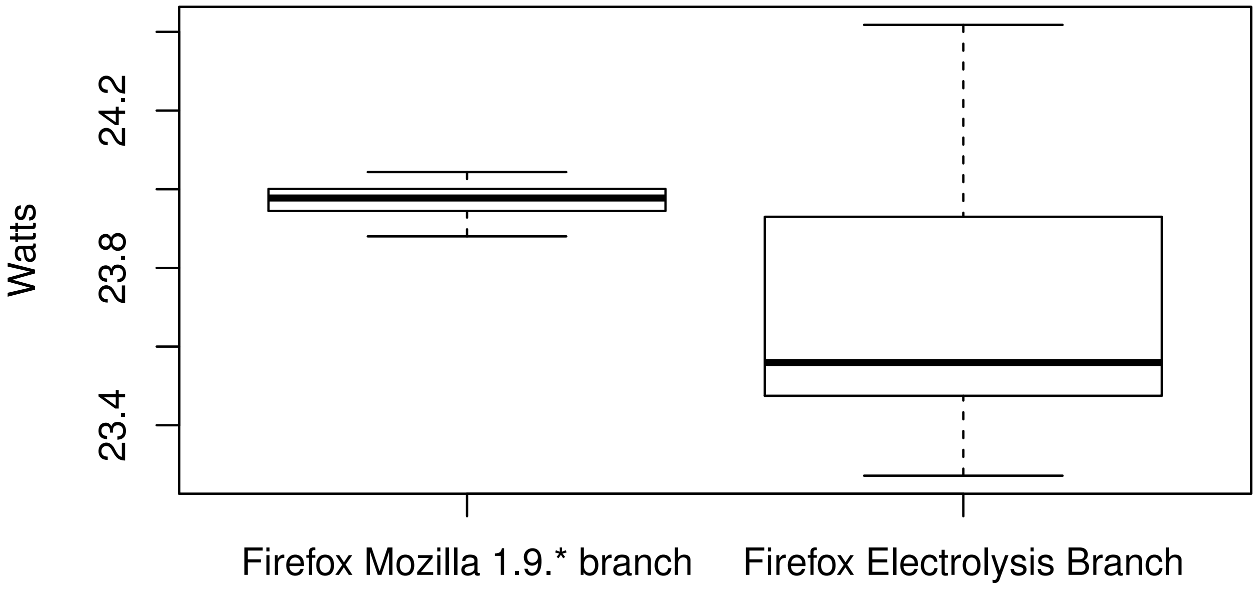

Figure 4 – Comparison of distribution of Firefox 3.6 mean watts and Firefox Electrolysis mean watts. Note that Firefox 3.6's mean is higher but the distribution is tighter.

5.2 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 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 to verify 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 is 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 as not all commits are compilable, and changes in build systems can affect ability to compile. 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 uniformly 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 100Mbit Ethernet with a cross-over cable.

5.2.1 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 distributed hash table (DHT) function, but no one randomly generated the same file so this did not cause a problem. DHT allows BitTorrent clients to connect and find each other without a single centralized tracker.

Based on this test setup we ran the test between 17 and 21 times (median 20) for each version of Vuze that we compiled. This range was chosen as a function of available test time, test runs that did not fail, and minimal test runs to achieve usable statistical power. 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

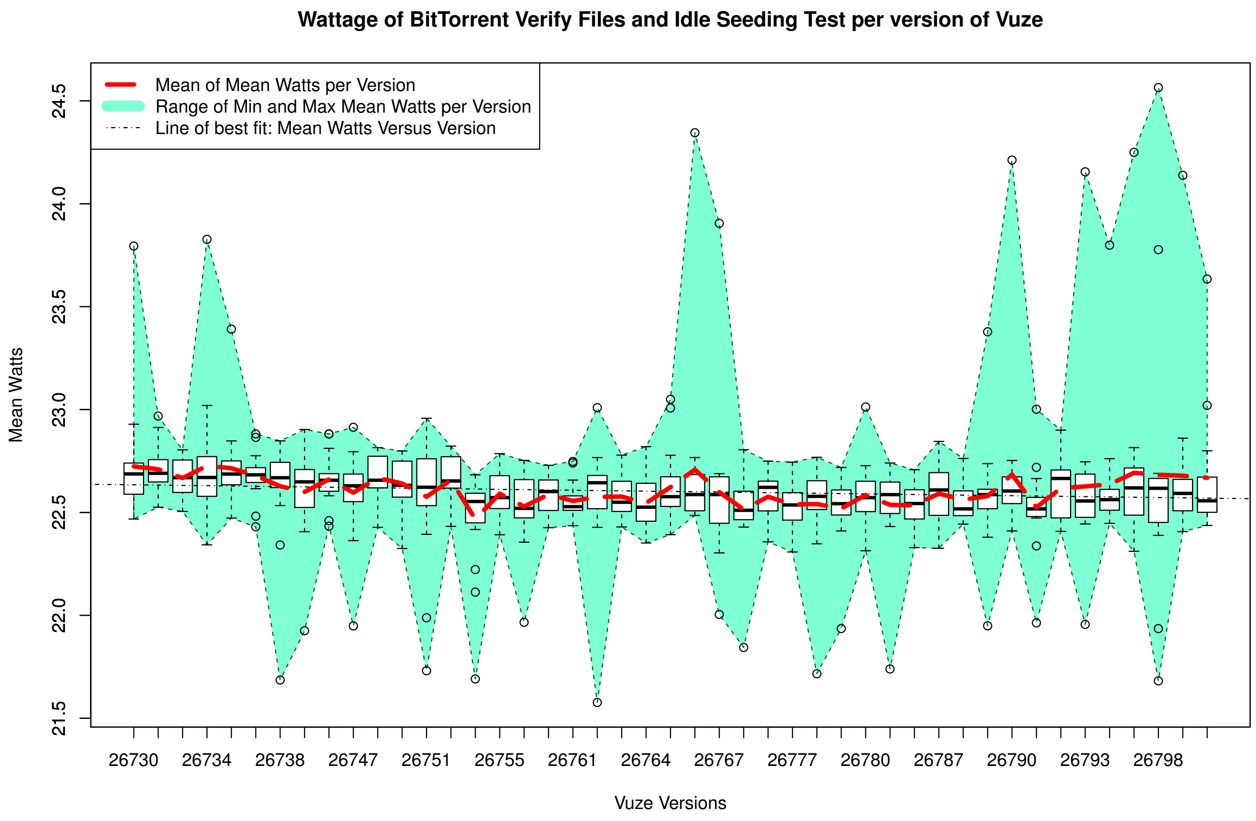

Figure 5 – Vuze Init, Verify and Idle test. This test shows the distribution of tests run on 45 revisions of Vuze, totaling 900 tests. The line of best fit has a weak negative slope across the versions. The test consists of Vuze starting up, checking file integrity, seeding the file and idling.

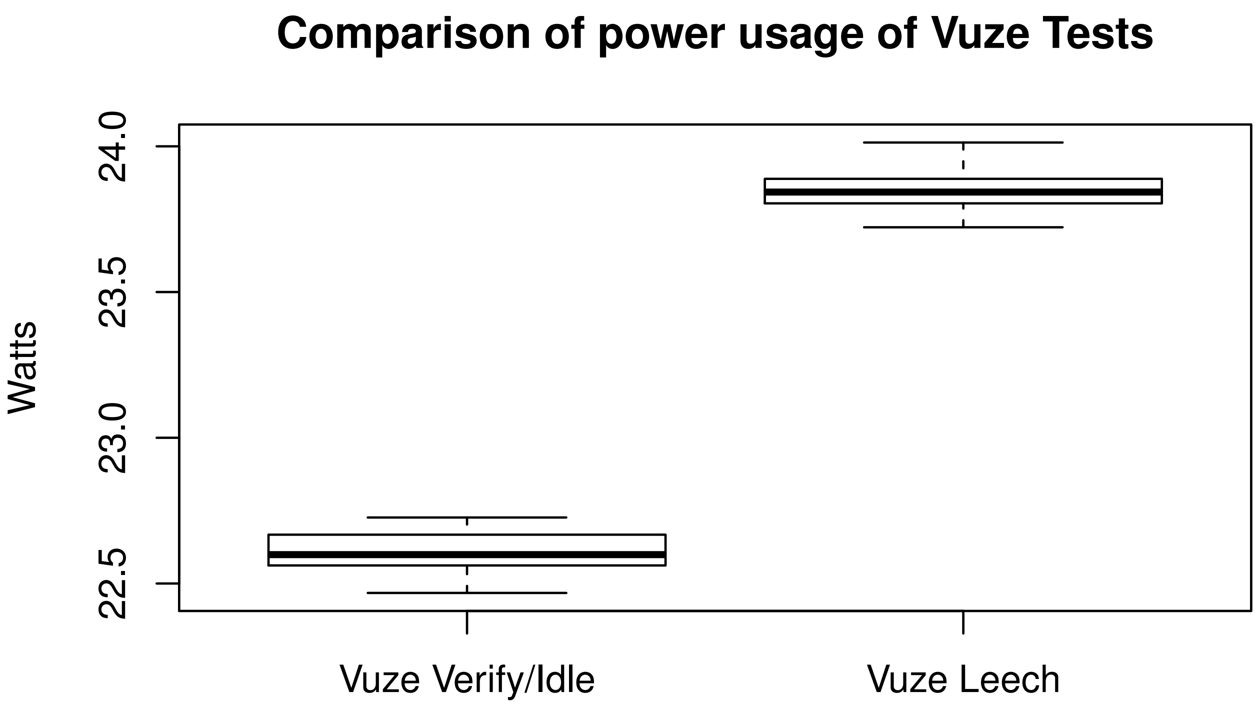

Figure 6 – Comparison of distribution of the Idle Vuze (Figure 5) tests mean watts and Leech Vuze test mean watts (Figure 7).

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. This could suggest that careful attention to UI APIs could result in power savings, especially anything that produces events. Furthermore attempts to improve performance might have power saving benefits, as demonstrated by Firefox Electrolysis.

This test's trace of wattage measurements is in the second sub-figure of Figure 2, 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.

5.2.2 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. The X31 with instrumentation, while idle, ran at approximately 19.5 watts.

In total for the 45 versions of Vuze, we ran each test 10 to 15 times with a median of 12 times. This range was chosen as a function of available test time, test runs that did not fail, and minimal test runs to achieve usable statistical power. The mean-watts measurement for the Vuze Leech tests was 23.8 watts. Figure 7

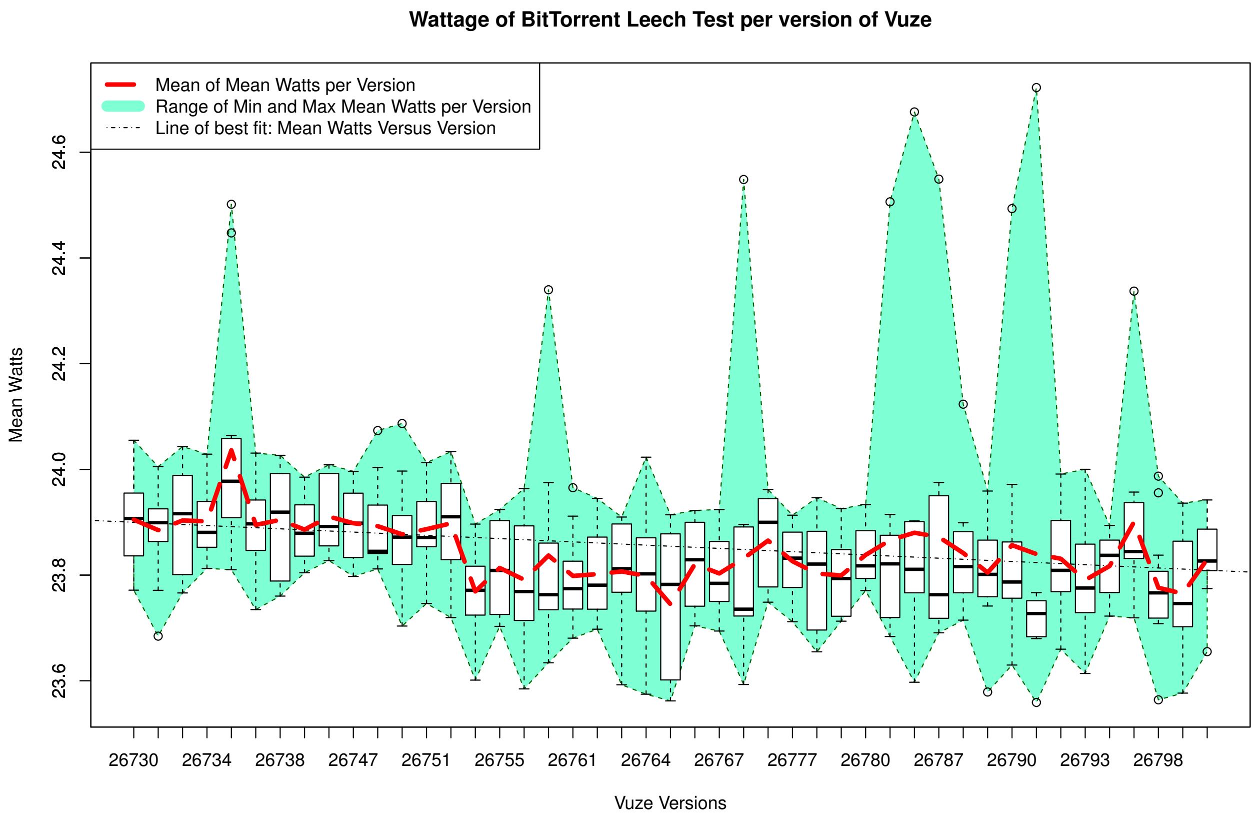

Figure 7 – Wattage of Vuze Leech Tests. This test shows the distribution of tests run over 45 revisions of Vuze, totaling more than 500 tests. The line of best fit has a stronger negative slope than Figure 5, across the versions. The test consists of Vuze starting up, initializing a file, and leeching the blocks from a seeder.

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 2, 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 adjusted-R^2 of 0.038 for this test with very tiny p-values, by adding transactions per second we achieve an adjusted-R^2 value of 0.046, so some information is being provided by the disk usage (AIC also drops slightly). These R^2 values indicate 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 resource usage metrics such as CPU utilization.

5.3 `rTorrent` and `libTorrent`

`rTorrent` [24] is a popular Linux C++ console-based BitTorrent client. `rTorrent`'s popularity comes from its lack of a graphical UI, instead it has a textual UI implemented in ncurses, which allows users to often couple GNU/Screen with it in order to run it remotely and in the background. `rTorrent` can leech (download) and seed (upload) files shared over BitTorrent.

`rTorrent` is the front-end client that makes calls to `libTorrent` in order to download a torrent from a BitTorrent cloud of peers (seeders and leechers). `rTorrent` and `libTorrent` are generally developed together and not all versions of `libTorrent` are compatible with `rTorrent` and vice-versa.

Out of 60 `rTorrent` tarballs we managed to successfully build 18 of them with `libTorrent`, and out of 65 `libTorrent` tarballs we could build 18 that would link with `rTorrent`. The 40 combinations of what was testable are depicted in Figure 11. See section 8.1.2 for a discussion of issues facing building `rTorrent`.

5.3.1 Testing `rTorrent` and `libTorrent`

Our test for `rTorrent` and `libTorrent` was similar to the test for the Leech test for Vuze described in 5.2. The test plan was that a seeder would run on a separate machine which also ran the tracker (the service that tells BitTorrent clients who is available in the swarm). This seeder would seed a file at a maximum rate of 5 to 6 Mb/s. Our testbed would checkout a pre-compiled version of `rTorrent` and `libTorrent`, that we had built ourselves, and then run that `rTorrent` in order to download a high-entropy 2GB file. We ran each combination of `rTorrent` and `libTorrent` multiple times.

Our tests did not need a GUI driver, as `rTorrent` could be started from the command-line. Yet `rTorrent` is ncurses-based, and requires a terminal. To provide a terminal without the actual need for a screen we used GNU/Screen to start a new session with `rTorrent` started within it. This would allow control of `rTorrent` and allow `rTorrent` to output to a terminal. Unfortunately Screen adds more complexity to the framework and can have some effect on power consumption.

We found that the behaviour in terms of power consumption of `rTorrent` was quite stable. Over time and over versions there was not much change in behaviour. This is evident in the slope of the line of best fit on the Means Watts per version of `rTorrent` as seen in Figures 8 and 9. We can see that `rTorrent` versions have a slight downward slope over time, indicating perhaps an increase in efficiency causing an decrease in power consumption.

Figure 8 depicts the power usage of `rTorrent` over all tests for any combination of `libTorrent` and `rTorrent` that was built. We can see there is some variability between versions but there is no clear gain or drop in power consumption across the versions.

Figure 9 depicts the power usage broken down by version of `libTorrent` over all tests for any combination of `libTorrent` and `rTorrent` that was built. We can see if we compare with Figure 8 that there is some change. But it is not clear if changes in `libTorrent` or `rTorrent` really have any affect. There is a lot of variability near the end of `libTorrent` 0.12.x series and it settles down at 0.13.x.

Later `libTorrent` 0.12.x series changes added features such as relying on operating system specific syscalls to preallocate files [26], as well as refactoring out some of the polling calls (to `libevent` and `libepoll`) into a new scheduler (verified in diffs and changelog) [25]. Many other new features were added but these two features could lead to a lot of variability.

Figure 10 is a density plot of our measurements showing per second wattage measurements. We can trace our test based on the power consumption: first there is a slow gradual ramp up as more blocks are downloaded and traded from second 0 to around second 360. Then once the file is complete `rTorrent` cryptographically hashes all blocks and checks them against the hashes in the `.torrent` file which describes the hashes of the blocks of the files, after-wards `rTorrent` goes idle waiting for connections.

The checking blocks requires disk usage (reading the blocks) and CPU usage (hashing the blocks). We created 3 linear models to explain the interaction of CPU and IO on the power consumption (watts) of `rTorrent`. A model of just CPU (% user time) had an adjusted R^2 of 0.6789, a model of just IO (transactions per second) had an adjusted R^2 of 0.3085. Combined, the model had an adjusted R^2 of 0.6856 indicating that more variance was explained and the combination of CPU and IO tells us more than CPU alone. Since this is an adjusted R^2 it already penalizes the model for adding another variable, thus this small improvement is an improvement nonetheless. This is visually apparent in Figure 10.

5.3.2 Version by Version Variation

Figure 11 depicts the combinations of `libTorrent` and `rTorrent` that were successfully built together. We tried all pair-wise combinations and found only these 40 combinations worked together. We can see that certain versions are more compatible than others. The range of this plot is not much and thus we can see that software change in `libTorrent` and `rTorrent` did not have much effect on the tests that we ran.

Thus to address *RQ1*, we can clearly see differences in power consumption based on Figure 11, between libraries and versions but no real trend is apparent. We can see that as the system evolves its power consumption evolves as well, but it does not follow, in this case study, a clear trend with respect to the libraries (`libTorrent`) linked to it. Between all pairs of `rTorrent` versions and between all pairs `libTorrent` versions there were no significant differences found by T-tests at a threshold of 0.05. This could be due to low number of samples, but nonetheless there is no clear trend in terms of difference of means. The take away is that different configurations of libraries and library-clients react differently and a change in one can cause a change in the other. The maximum difference in means between all compilable combinations of `libTorrent` and `rTorrent` is about 0.275 watts, with a mean usage of 22.69 watts. `rTorrent`'s leech test used 1.15 watts less than Vuze's leech test.

6 RQ1: Does power consumption evolve over time?

In this section we address our first research question, **RQ1**, "Does power consumption evolve over time?" Based on the results from our 3 case studies presented in the previous section we can clearly state that the power consumption of a system is not constant over time. There is variation and sometimes a clear positive or negative trend, but this often depends on the context, the configuration and the test being evaluated.

In Table 1 we can see that most of the cases resulted in a reduction but for some tests such as Vuze Idle tests and `rTorrent` tests the power consumption varied but did not change much over time.

Why did the power consumption change? For Firefox we found that there was a general focus on performance optimization, due to competing browsers such as Google Chrome, this eventually resulted

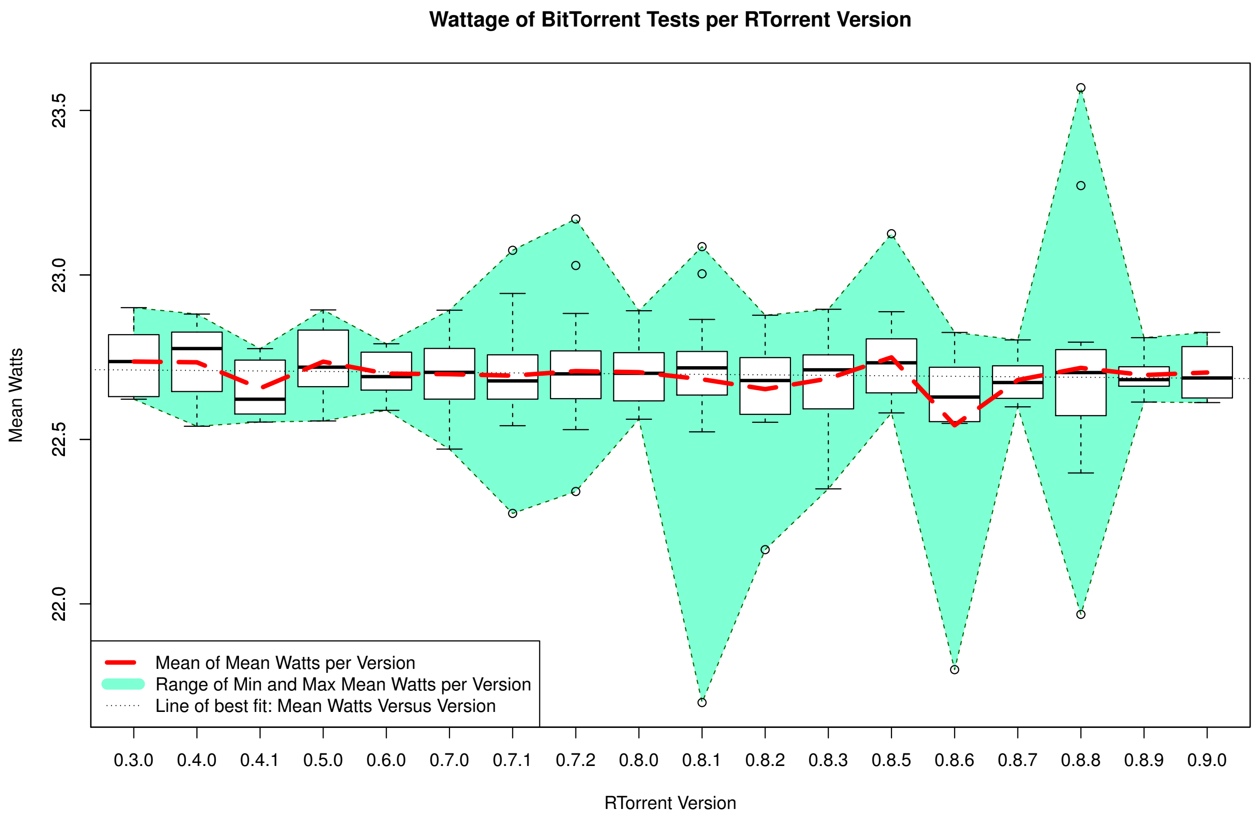

Figure 8 – Wattage per `rTorrent` Version (294 tests). Each boxplot depicts multiple tests of different versions of `libTorrent` used to build `rTorrent`.

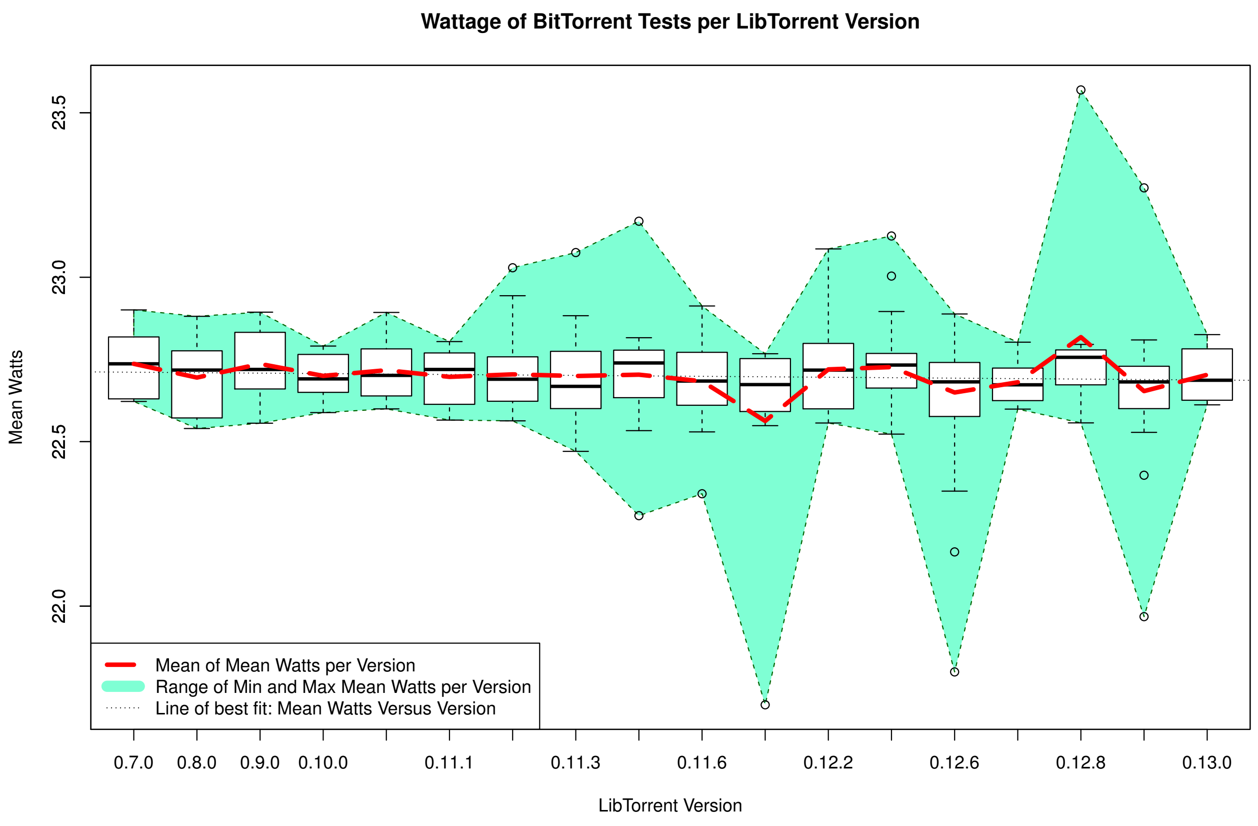

Figure 9 – Wattage per `libTorrent` Version (294 tests). Each boxplot depicts the multiple tests run against various versions of `libTorrent` and combined with `rTorrent`.

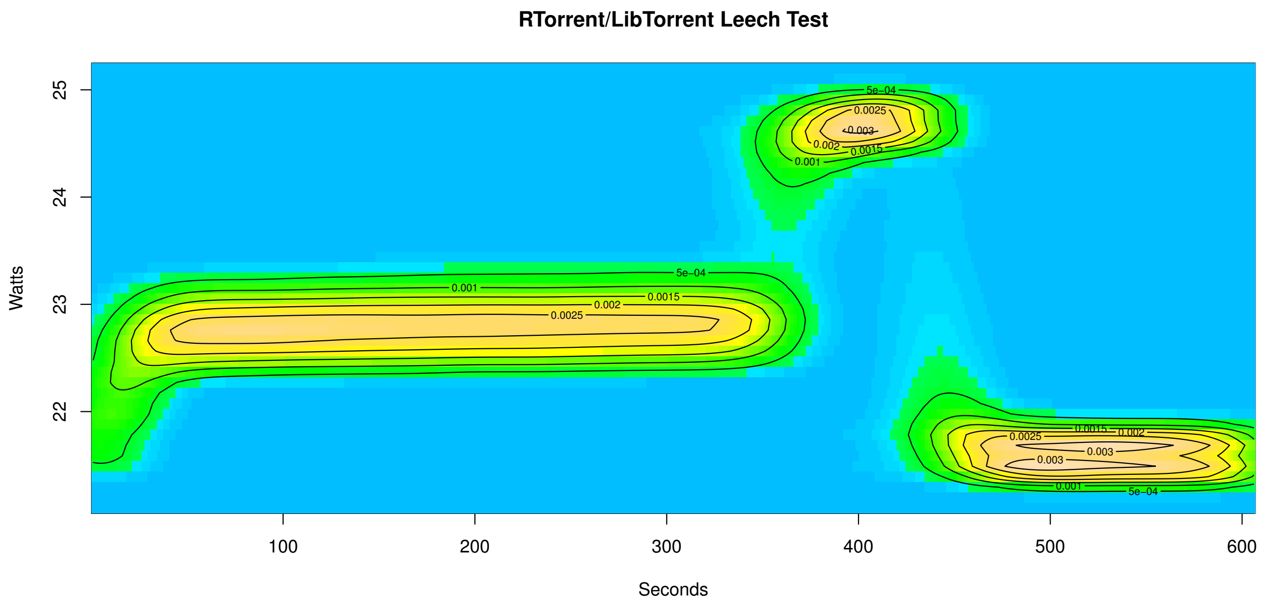

Figure 10 – Density of wattage measurements while running `rTorrent` leech tests. One can clearly see the file being downloaded until around second 360 where the file was validated by `rTorrent` and then `rTorrent` idled waiting for leechers. This plot summarizes 294 tests.

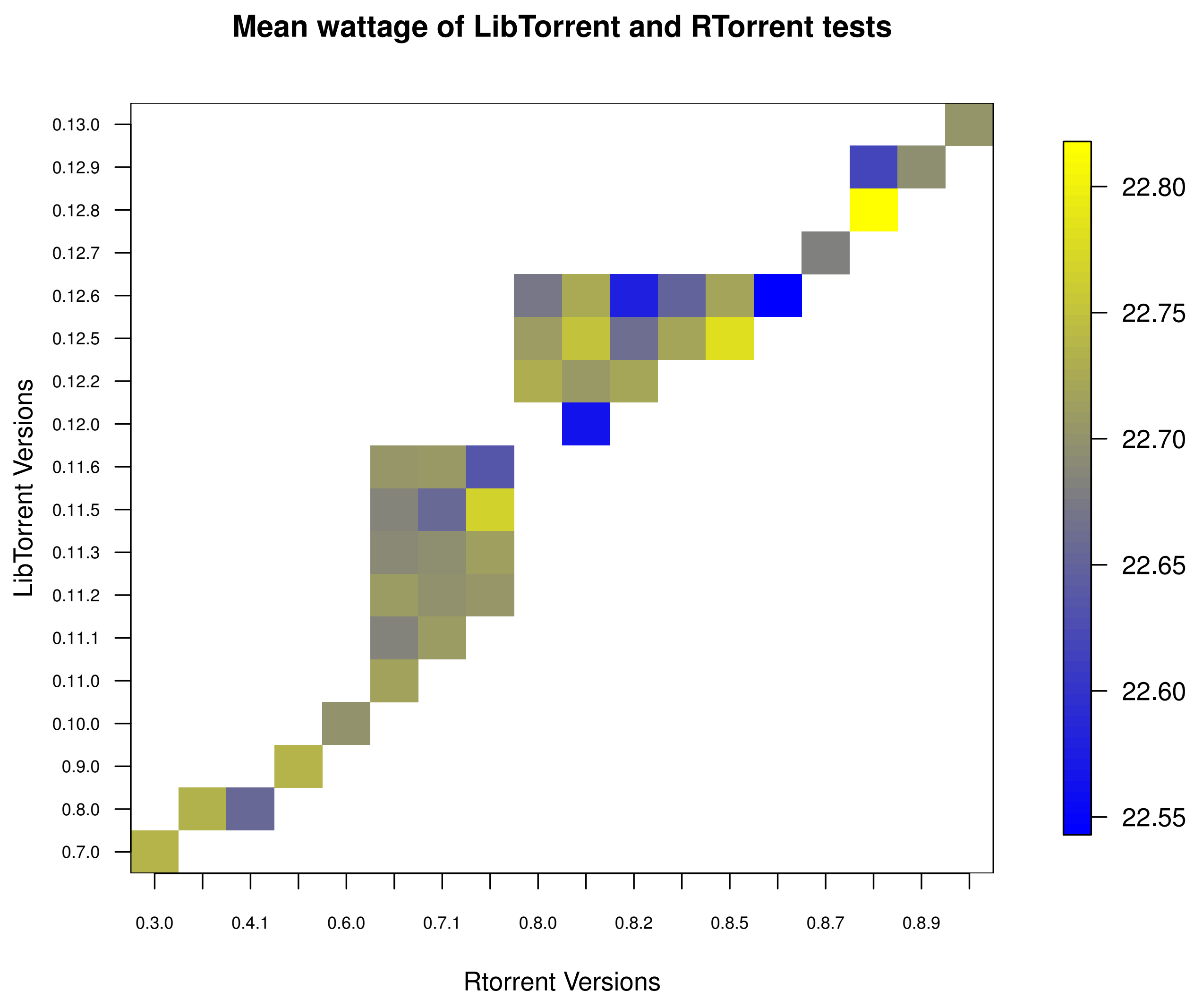

Figure 11 – Configurations of `rTorrent` and `libTorrent` tested. X and Y axis are the versions of `rTorrent` and `libTorrent`. Colors indicate mean wattage of multiple tests for those test runs. 40 combinations of 18 versions of `libTorrent` and 18 version of `rTorrent`.

in Mozilla attempting to re-architect Firefox in order to improve user-facing performance. For Vuze the emphasis was less clear, we attributed one drastic drop to changes in how the application used the SWT's GUI API, as well as a mild performance improvements.

`rTorrent` had a complicated history with its relatively tight co-evolutionary coupling with `libTorrent`. Judging from the changelogs there was also some focus on improving performance as well as a focus on using the optimal operating system call for IO operations. But the focus was not power performance, as bug fixes and new features were added in each revision thus the reasons behind variation `rTorrent` are complex and varied. We investigated the diffs between 0.8.5, 0.8.6, and 0.8.7 of `rTorrent` and observed changes in polling behaviour (the move to a scheduler) but it does not explain why 0.8.7 does not exhibit the drop in mean watts that 0.8.6 did.

Thus it seems that performance, as a non-functional requirement, is often relevant to power consumption but it can modify power consumption in different ways. Vuze and Firefox seemed to consistently gain from attention to performance while `rTorrent` was inconsistent.

In summary, the granularity of what we evaluate affects our ability to determine the fault behind a change in power measurements. These results suggest that power consumption does indeed change, but further motivate the need for more research on explaining why power consumption changes.

7 RQ2: What software metrics are relevant to power consumption?

This section addresses **RQ2:** *what software metrics are relevant to power consumption?* We investigate the relationship between OO metrics, and process metrics and power consumption.

7.1 CKJM Metrics and Power Consumption

The extended CKJM metrics [14, 23] suite relies on compiled Java bytecode to extract classes, exceptions, methods, calls to other methods and classes, call graphs, field visibility and field accesses in order to calculate a raft of OO metrics based around coupling, cohesion, fan-in, fan-out, and size measures.

For each test we evaluated the effect of OO software metrics on power consumption. Using the extended CKJM metrics suite [14, 23] 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 correlates 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.

We did not execute CKJM metrics on Javascript or C++ code (`rTorrent` and Firefox) as the CKJM metrics suite we used did not support those languages and only supported Java byte-code [14, 23].

Table 2 depicts the Spearman-rho correlation (ρ) and p-values (p) of mean OO metrics and mean-watts and the change in mean-watts (DX mean watts). Conversely, table 3 depicts the Spearman-rho correlation and p-values of mean change in OO metrics and mean-watts and the change in mean-watts (DX mean watts).

7.1.1 Vuze Idle Test

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 ($\rho = -0.334$, $p = 0.026$) and a change in mean MOA ($\rho = 0.315$, $p = 0.037$), see Table 3. Data access metric (DAM) is the ratio of private and protected attributes versus the total number of attributes in a class [14]. Measure of aggregation (MOA) is the measure of count of fields in a class that derive from user provided classes [14]. Total mean watts rank-correlated with changes in polymorphic measures such as depth in inheritance tree (DIT) ($\rho = -0.373$, $p = 0.013$), changes in DAM ($\rho = -0.324$, $p = 0.032$), and changes in number of children (NOC) [14] ($\rho = 0.369$, $p = 0.014$), in

	Mean Watts				DX Mean Watts			
	Idle ρ	Idle p	Leech ρ	Leech p	Idle ρ	Idle p	Leech ρ	Leech p
WMC	0.146	0.338	0.184	0.225	-0.094	0.539	-0.040	0.796
DIT	-0.250	0.098	**-0.374**	0.011	0.140	0.360	-0.024	0.874
NOC	0.250	0.098	**0.374**	0.011	-0.140	0.360	0.024	0.874
CBO	0.166	0.274	0.231	0.127	-0.126	0.409	0.019	0.900
RFC	0.159	0.298	0.160	0.293	-0.100	0.512	-0.018	0.904
LCOM	0.137	0.370	0.191	0.210	-0.096	0.529	-0.039	0.801
Ca	0.166	0.275	0.230	0.129	-0.125	0.415	0.021	0.893
Ce	0.175	0.250	0.056	0.712	-0.036	0.812	0.136	0.372
NPM	0.126	0.408	0.183	0.229	-0.098	0.520	-0.059	0.698
LCOM3	0.106	0.489	0.218	0.151	-0.049	0.750	-0.151	0.323
LCO	0.143	0.349	0.139	0.363	-0.090	0.557	-0.003	0.983
DAM	-0.004	0.980	-0.100	0.512	0.108	0.479	-0.120	0.430
MOA	-0.127	0.404	**-0.315**	0.035	0.152	0.319	0.029	0.850
MFA	-0.259	0.086	-0.076	0.621	0.060	0.695	0.064	0.676
CAM	-0.220	0.147	**-0.335**	0.024	0.106	0.486	0.063	0.680
IC	-0.259	0.086	-0.076	0.621	0.060	0.695	0.064	0.676
CBM	-0.259	0.086	-0.076	0.621	0.060	0.695	0.064	0.676
AMC	0.105	0.494	-0.119	0.437	0.043	0.779	0.036	0.816

Table 2 – Spearmean-rho (ρ) correlation and p-value of correlation between Vuze Idle and Vuze Leech test and mean CKJM metrics and mean watts for those tests.

	Mean Watts				DX Mean Watts			
	Idle ρ	Idle p	Leech ρ	Leech p	Idle ρ	Idle p	Leech ρ	Leech p
WMC	-0.058	0.710	0.176	0.254	-0.063	0.686	-0.123	0.426
DIT	**-0.373**	0.013	0.075	0.629	-0.296	0.051	-0.103	0.504
NOC	**0.369**	0.014	-0.073	0.640	0.290	0.057	0.112	0.467
CBO	0.174	0.259	0.142	0.359	-0.008	0.961	0.236	0.123
RFC	-0.045	0.772	0.232	0.129	-0.251	0.101	-0.047	0.762
LCOM	-0.060	0.700	0.152	0.324	-0.050	0.746	-0.120	0.437
Ca	0.121	0.434	0.109	0.482	-0.005	0.973	0.205	0.182
Ce	0.148	0.338	-0.059	0.702	0.124	0.423	0.057	0.712
NPM	-0.021	0.890	0.170	0.269	-0.120	0.438	-0.184	0.231
LCOM3	-0.293	0.054	0.097	0.531	-0.295	0.052	-0.171	0.268
LCO	0.101	0.513	0.240	0.117	-0.032	0.836	0.042	0.785
DAM	**-0.324**	0.032	0.166	0.281	**-0.334**	0.026	-0.168	0.274
MOA	0.257	0.092	-0.041	0.793	**0.315**	0.037	-0.078	0.616
MFA	-0.151	0.328	0.222	0.147	-0.211	0.169	0.051	0.745
CAM	0.073	0.637	-0.253	0.098	0.079	0.610	0.155	0.315
IC	-0.151	0.328	0.222	0.147	-0.211	0.169	0.051	0.745
CBM	-0.151	0.328	0.222	0.147	-0.211	0.169	0.051	0.745
AMC	0.124	0.421	0.153	0.322	0.074	0.631	-0.120	0.438

Table 3 – Spearmean-rho (ρ) correlation and p-value of correlation between Vuze Idle and Vuze Leech test and mean **change in** CKJM metrics and mean watts for those tests.

Table 3. These correlations are interesting and suggest that we need more data and more study to further tease out relationships if they exist.

7.1.2 Vuze Leech Test

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) ($\rho = -0.374$, $p = 0.011$), mean number of children (NOC) ($\rho = -0.374$, $p = 0.011$), mean measure of aggregation (MOA) ($\rho = -0.315$, $p = 0.035$), and mean *Cohesion Among Methods of Class* (CAM) ($\rho = -0.335$, $p = 0.024$) all rank-correlated with mean watts (and each other, see Table 2). Thus there was an interesting correlation between OO structural size metrics and mean watts. In the future, we want to test more Vuze versions, and other systems, in order to improve reliability.

7.2 Process Metrics: Lines of Code and Churn

Process metrics, such as churn, and size metrics such as *lines of code* (LOC) could correlate with power consumption. In our prior work [10] we report a lack of or a weak correlation with LOC, and for the most part that is confirmed in this section, but not for every case. Instead we find that Firefox and Vuze, when evaluated at a fine-grained level do not tend to exhibit churn correlations with power consumption or difference in power consumption, yet `rTorrent` does.

7.2.1 Churn Measures

Churn is viewed as the change in lines and files between two versions of a software system. It can be measured absolutely or relatively against the later or newer version. We leverage the churn measures from Nagappan et al. [20]. These measures are between versions of the systems and we only measure files with source code file extension such as `.java`, `.cpp`, `.js`, etc.

Added lines are the number of new lines added to the new system.

Removed lines are the number of lines removed from the old system.

Churned Lines are sum of number of added and removed lines.

Total Files is the total files in the new system.

File Churn is the total files modified divided by the total files in the new system.

LOC is the total lines of code from source code files in the new system.

ChurnLOC is the Churned Lines divided by the LOC.

We measured these values from each consecutive version of the software that we tested for Vuze, Firefox and `rTorrent`. We sought to find if there was relationship between mean watts and these metrics. We also measured the difference in mean watts between versions and investigated if churn would correlate with those measures.

7.2.2 Firefox and Churn

For Firefox, depicted in Table 4 we can see that the only correlations with a p-value less than 0.05 are *total files* and *LOC*, but these measures are very weakly correlated with mean watts or mean changed watts. The rest of the correlations are very low and have high p-values. Yet for Electrolysis *total files* and *LOC* are negatively correlated with Spearman-rhos less than -0.27. We conclude for Firefox, given our specific tests, that churn does not correlate with mean watts or a change in mean watts, but the high p-values imply we might need more data to make this statement with any sort of statistical significance.

	Firefox				Firefox Electrolysis			
	mean W	p	dx mean	p	mean W	p	dx mean	p
Added Lines	0.004	0.932	-0.055	0.293	0.063	0.202	-0.001	0.990
Removed Lines	0.013	0.812	-0.034	0.521	0.085	0.083	0.047	0.342
Churned Lines	0.009	0.866	-0.053	0.314	0.075	0.126	0.019	0.705
TotalFiles	**0.142**	0.007	0.035	0.502	**-0.295**	0.000	0.035	0.472
File Churn	-0.005	0.930	-0.038	0.467	0.064	0.190	0.020	0.678
LOC	**0.133**	0.012	0.042	0.431	**-0.270**	0.000	0.032	0.508
ChurnLOC	0.006	0.907	-0.053	0.319	0.076	0.122	0.018	0.707
mean Watt	1.000	NA	**0.675**	0.000	1.000	NA	**0.477**	0.000
dx mean	**0.675**	0.000	1.000	NA	0.477	0.000	1.000	NA

Table 4 – Firefox Churn Metrics versus mean watts using Spearman-rho rank correlation

7.2.3 Vuze and Churn

For Vuze, depicted in Table 5 no churn measure or size measure was statistically significantly correlated with mean watts or mean changed watts. The p-values are so high they defy interpretation, in both the idle test and the leech test.

7.2.4 `rTorrent` and Churn

For `rTorrent`, in Table 6, we can see some positive response between churn and size metrics and mean watts and mean change watts, but only in the `rTorrent` version tests and not if the granualarity was `libTorrent` versions. For the `rTorrent` version tests we can see, added lines, removed lines, churned lines, file churn, and ChurnLOC (relative churn over lines) all correlate with meant watts. Relative churn, ChurnLOC correlates the most, and has a strong Spearman rho correlation of 0.71 and a p-value of 0.001. Added lines, removed lines, and churned lines all had p-values less than 0.01 and medium to strong Spearman rho correlations greater than 0.64. Thus for the `rTorrent` tests we observe that churn does matter and does relate to mean-watts of the test.

With such high correlations can we predict the mean watts? Given `rTorrent`'s correlations with some churn measures we created linear models to see if we could predict the mean watts of the system. Using a linear model for mean watts, while controlling for size by including LOC, we can achieve adjusted-R^2 values between 0.1 and 0.26 using $log(0.5 + addedLines)$ ($R^2 = 0.26$), $log(0.5 + removedLines)$ ($R^2 = 0.24$), $log(0.5 + churn)$ ($R^2 = 0.26$), or $ChurnLOC$ ($R^2 = 0.10$). Without log scaling the raw churn values the R^2 value is much less, but this is expected as the linear correlations with these measures and mean watts is low, but they have high rank correlations. In all of the models LOC's coefficient is not statistically significant. Both coefficients of the independent variables in the $ChurnLOC + LOC$ model were not statistically significant. Furthermore our models are simple, as they consist of only 2 independent variables because of heavy multi-colinearity between the size and churn measures, and even between LOC and the other measures. The positive correlation and the positive linear model coefficients indicates that churn measures are positively correlated with mean watts for `rTorrent`.

The `rTorrent` results make it very apparent that the granularity of the testing and the kind of testing definitely matter. This suggests that to better model power we need more kinds of code executed as well more use cases.

7.3 Summary

Our investigation leads us to conclude that there is some promise in OO metrics, but we need more data. Also, we need a wide variety of tests and systems before we can generalize. Perhaps OO metrics are not going to relate to power consumption because they might not represent the dynamic run-time qualities of a system that would affect power consumption. It is apparent that the code that is executed during a test has

	Vuze Idle				Vuze Leech			
	mean W	*p*	dx mean	*p*	mean W	*p*	dx mean	*p*
Added Lines	-0.031	0.843	0.151	0.328	-0.031	0.843	0.151	0.328
Removed Lines	-0.154	0.319	-0.038	0.806	-0.154	0.319	-0.038	0.806
Churned Lines	-0.040	0.795	0.119	0.440	-0.040	0.795	0.119	0.440
TotalFiles	-0.108	0.484	0.006	0.969	-0.108	0.484	0.006	0.969
File Churn	0.085	0.584	0.063	0.686	0.085	0.584	0.063	0.686
LOC	-0.055	0.725	0.123	0.427	-0.055	0.725	0.123	0.427
ChurnLOC	-0.046	0.767	0.105	0.499	-0.046	0.767	0.105	0.499
mean Watt	1.000	NA	**0.556**	0.000	1.000	NA	**0.556**	0.000
dx mean	**0.556**	0.000	1.000	NA	**0.556**	0.000	1.000	NA

Table 5 – Vuze Churn Metrics versus mean watts using Spearman-rho rank correlation

	`rTorrent`				`libTorrent`			
	mean W	*p*	dx mean	*p*	mean W	*p*	dx mean	*p*
Added Lines	**0.688**	0.002	**0.495**	0.043	0.392	0.119	0.279	0.277
Removed Lines	**0.640**	0.006	**0.498**	0.042	0.475	0.054	0.287	0.264
Churned Lines	**0.667**	0.003	**0.505**	0.039	0.395	0.117	0.297	0.248
TotalFiles	-0.159	0.543	0.196	0.452	-0.272	0.291	0.059	0.821
File Churn	**0.588**	0.013	0.306	0.232	0.110	0.673	0.201	0.439
LOC	-0.159	0.541	0.189	0.468	-0.289	0.260	0.049	0.852
ChurnLOC	**0.708**	0.001	0.468	0.058	0.444	0.074	0.267	0.300
mean Watt	1.000	NA	**0.598**	0.011	1.000	NA	**0.784**	0.000
dx mean	**0.598**	0.011	1.000	NA	0.784	0.000	1.000	NA

Table 6 – `rTorrent` and `libTorrent` Churn Metrics versus mean watts using Spearman-rho rank correlation. Note, this shows the same set of tests broken down by `rTorrent` churn first, then `libTorrent` churn.

a significant effect on the power consumption and thus if we only test and study two main code paths we are probably missing what the changes in the system are related to. Thus we need more tests that exercise more code in more systems to definitely say anything more about static OO metrics and process metrics and how they relate to power consumption. The response to churn by `rTorrent` is interesting and suggests that in some scenarios churn might be relevant to power consumption, yet it was not evident in Firefox or Vuze tests.

8 Difficulties Faced While Green Mining

There were patterns that we encountered when dealing with the problem of compiling, building, packaging, configuring, executing, power monitoring, and testing multiple versions of a software application. Table 1 relates the issues discussed in this section to each case study undertaken.

8.1 Compilation

Compiling binaries is a constant issue in MSR studies because of the evolving requirements of a system, as well as changing implicit dependencies. Furthermore not ever commit is made to be immediately compilable.

The change of build environment can cause numerous headaches. This is evident because distributions such as Debian and Ubuntu have many volunteers solely dedicated to building new versions of single packages within the distributions' ecosystem.

For Firefox we avoided compilation entirely because we relied on nightly builds that were distributed by Mozilla to allow people to alpha-test the latest Firefox commits.

Compilation was confounded by evolving compilers and evolving language specifications (C++) as well as changes in build systems.

8.1.1 Compilation of Vuze

In total we built 30% of all versions of Vuze from its Subversion repository. We ran into numerous problems, the build system tended to change over time, Eclipse was popular, *ant* was popular and plain `javac` was popular for a time. Build instructions were sparse and not very trust-able. Thus in our compilation history there are large swathes of Vuze which do not compile given our compilation scripts. In terms of ant, targets (goals) in the ant files would change and get in the way of building, thus we had to check the ant file for particular targets first, before we called them (or build them unsuccessfully).

During the building of Vuze/Azureus we found there was a lot of development that was not being compiled into the main `jar` file. Thus we had to try to address branches, and branches in development. We did not successfully address all of these concerns.

8.1.2 Compilation of `rTorrent` and `libTorrent`

One of the first problems with investigating software changes that requires dynamic tests is building the software. `rTorrent` depends on `libTorrent` and the documentation is not always clear which versions of `libTorrent` are compatible with which versions of `rTorrent`.

Furthermore, as `rTorrent` evolved, so did GCC. The evolution from GCC3 to GCC4 was a major shift, especially in adherence to C++ standards, the C++ standard libraries and name-spaces. Changes in GCC made most of `rTorrent` and `libTorrent` uncompilable without patching.

In order to build many versions of `rTorrent` and `libTorrent` on Ubuntu 11.04 machines we had to produce patches for at least each major version of `libTorrent` and `rTorrent`. Most patches added header includes such as `cstring`, `cstdio`, `cstddef`, `inttypes algorithm`, and `functional`. Our build scripts had to try the relevant patches that we had manually developed for specific versions on versions we did not manually develop patches for. Often this patch application would fail.

8.2 Library Compatibility

Library compatibility becomes a larger concern when one investigates a software ecosystem where different configurations of versions of applications could have drastically different power consumption behaviour. Systems change and so do compilers and development environments. As the environment changes many applications suffer bit rot and become more difficult to build.

The testbed to test for power consumption must be constructed, but the software installed might not be compatible with the intended applications. Library compatibility was not an issue with Firefox because they used statically compiled libraries for many of their dependencies, but in `rTorrent`, `libSigc++` changed during the evolution of `rTorrent` and the older versions of `rTorrent` and `libTorrent` do not reflect this reality.

Vuze tests could be confounded by the underlying widget toolkit used as well as the version of the JDK and JVM used.

`rTorrent` was an interesting case, while we were fortunate not to run into library issues with Firefox or Vuze, `rTorrent` had a kind of development where its UI, `rTorrent`, was closely coupled with the `libTorrent` library. This caused configuration issues, because `rTorrent` would need a specific version of `libTorrent` or better to compile and to run. The need for pairwise testing of valid `rTorrent` and `libTorrent` candidates shows that this could be a serious problem if one was testing an application dependent on more than 1 important external library. Also `rTorrent` relied on `libSigc++`, which changed include-file locations and thus `rTorrent` had to be patched. Sometimes there were more serious errors, but the sources of errors were often not consistent between versions. Worse, some headers, if included would cause errors in modules, thus we could not just include every single header. We tried to address this by applying manually written patches to `rTorrent`, but were not always successful.

As systems evolve so do their libraries, older versions of the system are often difficult to run on newer operating systems. This could be remedied by virtualization or building images of an appropriate operating system. Unfortunately this would add the operating system as another source of variation.

8.3 Spurious GUI Events and GUI Issues

GUIs are difficult to control, hard to read, interpret, and parse automatically and fraught with peril. In all three case studies GUIs caused problems. In Vuze and Firefox numerous modal dialogs needed to be dispatched and canceled, in `rTorrent` a terminal had to be provided for the text-based ncurses UI to run in. Furthermore Vuze would start other applications automatically that would have to be closed.

These issues lead to drastic measures where every few seconds the driver must check the window in focus and check for other spurious windows. This is further compounded by changes in UI behaviour as the program evolves. The GUI driver has to address changes in behaviour over time.

`rTorrent` was had a very stable UI that did not require any interaction. Firefox and Vuze UIs were dynamic and sometimes slightly changed their UIs between versions.

One possible solution to this issue would be a general purpose dialog canceling GUI driver that dismissed any popups or dialogs immediately. We achieved repeatable window placement performance from XMonad, a tile-able window manager, and would recommend that UI window place be taken into account while testing. A maximization keyboard shortcut could be enough to ensure appropriate window placement.

8.3.1 Firefox UI Issues

In the case of Firefox we ran into many GUI related issues. When Firefox starts up for the first time it often asks the user if they want Firefox to be the default browser. Often this dialog is modal and thus must be addressed before anything else can be done in Firefox. When Firefox starts up a second time it might execute the default browser check again, or might ask a question about restoring the previous session. Another kind of pop-up is the check for updates dialog, sometimes Firefox will check for updates or check for extensions. To address these confounding issues which can impede the application starting, we created logic in our GUI driver to detect spurious dialog boxes and press escape to cancel them. While the Firefox tests ran we also had to check to see if the browser was in focus.

8.3.2 Vuze UI Issues

Vuze/Azureus is a GUI application, it is full of dynamic self-updating behaviour. Every-time it would start-up, pop-ups and dialogs would appear. Some pop-ups offered to download "Download Accelerators", other pop-ups would check for updates and offer to update our version of Vuze/Azureus for us. Some of these dialogs might be modal so we had to script our GUI driver to cancel all of these dialogs. This was complicated further by the application opening the Vuze/Azureus blog within Firefox. We had to program the GUI driver to detect if Firefox was open, switch to the Firefox window and close it immediately.

8.4 Evolving Application Configuration and Cleaning up State

In all tests we needed to carefully scrub not only the testbed where the programs were run, but other directories including our home directories in order to avoid cached configuration information or cached downloads. Vuze was particularly difficult because it changed its save file behaviour and made its own directories within the current user's home-directory that had to be deleted. The dot files and dot directories that contained configuration information often had to be deleted as well. In all cases configuration directories were automatically created, and thus had to be deleted. If one could re-image quickly this difficulty would be rectified.

The testbed had to be cleaned up as well. Old testbed that were inadvertently running needed to be cleanly and clearly killed off before a new test could run.

In operating systems like Linux caching of executables is a concern because it affects start-up time and if the disk needs to be read. By avoiding peripheral access power is generally saved but at the cost of test accuracy.

One possible solution to the state problem is to re-image the machine under test and reboot after each test.

8.4.1 Firefox

Firefox is a complicated piece of software with many user-facing customization stored within its profile. We had to delete the profile from `/.mozilla` each time to avoid cache effects. This had the side effect of causing Firefox to build a new profile each and every time.

8.4.2 Vuze

Running Vuze/Azureus was not easy, we had to make a script that could invoke the jar file safely with all the necessary extra dependency jar files. We were fortunate not to run into library incompatibilities when trying to run Vuze/Azureus. Furthermore the behaviour of Vuze/Azureus changes over time. Our tests on revisions in the r12000 to r13000 range showed that the download directory would be local, while r25000 and greater revisions tended to create a download directory called `/Vuze Downloads`. Thus per each run we had to delete all the configuration directories and the file download directories. Without deleting the download directory, our client would simply validate a file and start seeding it.

8.4.3 `rTorrent`

For `rTorrent` we could simply delete the download directory over which we had explicit control. `rTorrent` was the least troublesome of the applications in terms of stored configuration.

8.5 Remote/Network Effects

We did not wish to store test data on the testbed, for worry of disk fragmentation and the disk filling up. Thus we transferred the results of a test remotely to another machine. Furthermore we remotely stored many of the compilations and snapshots of a programs we tested. Thus if the network was disconnected and did not reconnect itself valuable test time would be lost.

Some of our tests relied on the network as well, thus network failure had to be monitored as it could affect the test. Imagine a BitTorrent or Firefox test with no accessible content to download or view. Such a test would give spurious results.

Remote content is also a concern. For our Firefox tests we remotely hosted some test content, but dynamic content is hard to mirror and emulate, especially if there are advertisements and AJAX. The more dynamic a webpage is, the harder it is to consistently test it.

Other issues include checking for updates and auto-updating, one should investigate the applications being tested and try to avoid exercising these features unless it is the intent of the test.

Remote network effects can be avoided in some cases by careful routing or by isolating a machine from the network completely, which is not feasible for all tests.

8.6 Appropriate Peripherals

Our BitTorrent tests (both `rTorrent` and Vuze) occurred on a local area network, using Ethernet. This means that the cost of communication would be quite low compared with WiFi. We were not at liberty to use BitTorrent over WiFi but it would have been interesting to observe the effect.

Thus when one builds a test, one has to be aware of the peripherals (network adapters, disks) that might affected such a test.

8.7 Temperature: Heat and Cooling

The temperature of the testbed should be maintained constant, because if the room temperature changes the behaviour of the fans within the testbed can affect the power recorded during a test. A change in exposure to sunlight could cause an idle 1W difference in desktop machines. Since a system under load tends to be hotter, one is often measuring a complex response of the cooling system spinning up to counteract the heat detected by the case and CPU heat sensors. Furthermore if the building in which the testbed is stored has different heating or cooling behaviour during different seasons tests recorded in winter could be very different than tests recorded in summer. One problem with ACPI and batteries is that they often lack temperature circuitry and cannot correct their measurements for temperature.

One possible solution is to place machines under test in a temperature controlled environment, much like an incubator, as it is easier to heat a device than to cool it.

8.8 Mining Software Repositories

One of the issues facing anyone involved in green-mining or the investigation of software power consumption relevant to existing software is that there is a dearth of power trace repositories. In this paper we tested multiple systems and produced numerous power consumption traces from numerous tests, but it is our hope that in the future practitioners who do this kind of work will be record their power consumption traces in a software repository. These power trace repositories would leverage the resources of the stakeholders involved and hopefully would provide mining software repositories researchers with a wider variety of data than they could generate themselves. One of the major issues facing this particular work is the lack of available traces, with the ever increasing focus on software power consumption we suspect future MSR researchers will be able to leverage power consumption traces from performance test repositories for tests they did not write, and for software they did not actually test.

Future avenues of this work relevant to mining software repositories include:

Normalization of power measurements from different operating system images. Based on our findings regarding compilation and libraries How can we combine measurements from different machines or VMs and different operating systems versions in order to get a clear understanding of the effect that software maintenance has on power consumption?

How to model power consumption based on multiple kinds of tests. One test will not fully exercise the system or cover all the relevant tasks, thus when one tests a system and monitor power consumption how can one combine these measurements later into a coherent model? We encountered this problem as we could not clearly combine our Idle Vuze and Leech Vuze tests.

From a diff-level perspective, what structures and tokens affect power consumption. Instead of a purely metric/structural approach, are there code tokens or topics of code tokens that closely related to power consumption. These more semantically oriented models would likely prove of immediate use to practitioners.

Temperature, how to handle inconsistency. If one parallelizes tests then the temperature of the machines under test is unlikely to be constant and could cause a bias.

More tests, more systems, more code. This study suffers heavily from not having enough data in some case studies to make a statistically significant distinction between some measurements. We need repositories of this kind of information.

Exploring process metrics and power consumption. We tested a very small subset of measures, but the process metrics (relevant to churn) are particularly relevant to MSR research. There needs to be more investigation into conveniently extracted measures and their relationship to power consumption.

8.9 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. We also place meaning on software change even though there are obviously many meanings and purposes behind change. Furthermore our tests were not necessarily built to observe or exploit the software in terms of the code that was changed. Future work should address the coverage of the tests that we do run.

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 (reported with a precision of 0.1 W) and wattage measurements are actually estimates, so there can be measurement error. One concern in this work is if it was software change or if it was the environment that induced variability in measurement. Measurement error is a concern on modern systems, as well as the overhead of the testbed, as 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. External validity was also hampered by the lack of variation in terms of hardware. 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.

9 Conclusions

In this paper we proposed a methodology of relating software change to software power consumption and we applied this methodology in 3 case studies on 3 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 [28] per hour!

Our Azureus/Vuze tests were 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 demonstrated the difficulty of building and testing an library that had high coupling with its client application (`rTorrent`), but we could not find any clear power relevant interaction between the library versions and the client version. 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.

We found that power consumption of `rTorrent` tests correlated with churn measures, but for other systems such as Vuze, and Firefox no such correlation between their churn metrics and mean wattage was found.

We discussed our encounters with various common difficulties, that included UI noise such as dialog windows, pop-ups and UI based exceptions, as well as cached configurations, library configuration issues, and build issues compounded by an evolving environment.

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. This research was supported by an NSERC discovery grant.

References

[1] Amsel, N., Tomlinson, B.: Green tracker: a tool for estimating the energy consumption of software. In: Proceedings, CHI EA, pp. 3337–3342. ACM, New York, NY, USA (2010)

[2] Apple Inc.: iOS Application Programming Guide: Tuning for Performance and Responsiveness. `http://ur1.ca/696vh` (2010)

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

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

[5] Godard, S.: Sysstat utilities home page. `http://pagesperso-orange.fr/sebastien.godard/`

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

[7] Greenawalt, P.: Modeling power management for hard disks. In: MASCOTS '94., Proceedings of the Second International Workshop on, pp. 62 –66 (1994)

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

[9] Gurumurthi, S., Sivasubramaniam, A., Irwin, M., Vijaykrishnan, N., Kandemir, M.: Using complete machine simulation for software power estimation: the SoftWatt approach. In: Proc. of 8th Int. Symp. High-Performance Computer Architecture (2002)

[10] Hindle, A.: Green mining: A methodology of relating software change to power consumption. In: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on, pp. 78–87 (2012). `http://greenmining.softwareprocess.es/`

[11] Hindle, A.: Green mining: Investigating power consumption across versions. In: Proceedings, ICSE: NIER Track. IEEE Computer Society (2012). `http://ur1.ca/84vh4`

[12] IBM: IBM Active Energy Manager. `http://www.ibm.com/systems/management/director/about/director52/extensions/actengmrg.html` (2011)

[13] Intel: LessWatts.org - Saving Power on Intel systems with Linux. http://www.lesswatts.org (2011)

[14] Jureczko, M., Spinellis, D.: Using Object-Oriented Design Metrics to Predict Software Defects, *Monographs of System Dependability*, vol. Models and Methodology of System Dependability, pp. 69–81. Oficyna Wydawnicza Politechniki Wroclawskiej, Wroclaw, Poland (2010). `http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/`

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

[16] Larabel, M.: Ubuntu's power consumption tested. `http://www.phoronix.com/scan.php?page=article&item=878` (2007)

[17] Lattanzi, E., Acquaviva, A., Bogliolo, A.: 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, *Lecture Notes in Computer Science*, vol. 3254, pp. 352–361. Springer Berlin / Heidelberg (2004)

[18] Li, L., Liang, C.J.M., Liu, J., Nath, S., Terzis, A., Faloutsos, C.: Thermocast: A cyber-physical forecasting model for data centers. In: Proceedings, ACM SIGKDD. ACM (2011)

[19] Murugesan, S.: Harnessing Green IT: Principles and Practices. IT Professional **10**(1), 24–33 (2008)

[20] Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, pp. 284–292. IEEE (2005)

[21] Selby, J.W.A.: Unconventional applications of compiler analysis. Ph.D. thesis, University of Waterloo (2011)

[22] Shang, W., Jiang, Z.M., Adams, B., Hassan, A.E., Godfrey, M.W., Nasser, M.N., Flora, P.: An exploratory study of the evolution of communicated information about the execution of large software systems. In: WCRE, pp. 335–344 (2011)

[23] Spinellis, D.: Tool writing: A forgotten art? IEEE Software **22**(4), 9–11 (2005). DOI 10.1109/MS.2005.111. URL `http://www.spinellis.gr/pubs/jrnl/2005-IEEESW-TotT/html/v22n4.html`

[24] Sundell, J.: libTorrent and rTorrent Project. http://libtorrent.rakshasa.no/

[25] Sundell, J.: [Libtorrent-devel] LibTorrent 0.12.6 and rTorrent 0.8.6 released (2009). http://rakshasa.no/pipermail/libtorrent-devel/2009-November/002361.html

[26] Sundell, J.: [Libtorrent-devel] LibTorrent 0.12.7 and rTorrent 0.8.7 released (2010). http://rakshasa.no/pipermail/libtorrent-devel/2009-November/002539.html

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

[28] US Department of Energy: Energy explained: Electricity. http://www.eia.gov/energyexplained/index.cfm?page=electricity_home#tab2 (2012)