NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
NU-EECS-13-02

April 10, 2013

Making JavaScript Better By Making It Even Slower

Maciej Swiech, Peter Dinda

Abstract

On mobile devices, such as smartphones and tablets, client-side JavaScript is a significant contributor to power consumption, and thus battery lifetime. We claim that this is partially due to JavaScript interpretation running faster than is necessary to maintain a satisfactory user experience, and we propose that JavaScript implementations include a user-configurable throttle. To evaluate our claim we developed a web proxy system, named JSSlow, that reduces power consumption by transcoding client-side JavaScript and injecting "sleep" invocations. This can be done safely, even given JavaScript's single threaded nature, through the use of continuation passing, and the proxy model requires neither server nor client-side changes. Using JSSlow we studied the 120 most popular sites and found that the technique could reduce power consumption by an average of 6% on Android phones. We also considered buggy code (52% reduction) and advertising (10% reduction). To evaluate the system's impact on the user experience, we conducted a user study consisting of interactive tasks the user carried out on. The perceived performance impact varies by user and site, with the variation being highest on the most interactive sites, such as games. This argues for making the throttle user-configurable in some cases.

This project is made possible by support from the United States National Science Foundatation (NSF) via grant CNS-0720691

Keywords: mobile environments, user studies, application studies, power management, empathic systems

Making JavaScript Better By Making It Even Slower

Maciej Swiech Peter Dinda
Department of EECS, Northwestern University
{m-swiech,pdinda}@northwestern.edu

ABSTRACT

On mobile devices, such as smartphones and tablets, client-side JavaScript is a significant contributor to power consumption, and thus battery lifetime. We claim that this is partially due to JavaScript interpretation running faster than is necessary to maintain a satisfactory user experience, and we propose that JavaScript implementations include a user-configurable throttle. To evaluate our claim we developed a web proxy system, named JSSlow, that reduces power consumption by transcoding client-side JavaScript and injecting "sleep" invocations. This can be done safely, even given JavaScript's single-threaded nature, through the use of continuation passing, and the proxy model requires neither server nor client-side changes. Using JSSlow we studied the 120 most popular sites and found that the technique could reduce power consumption by an average of 6% on Android phones. We also considered buggy code (52% reduction) and advertising (10% reduction). To evaluate the system's impact on the user experience, we conducted a user study consisting of interactive tasks the user carried out on. The perceived performance impact varies by user and site, with the variation being highest on the most interactive sites, such as games. This argues for making the throttle user-configurable in some cases.

1. INTRODUCTION

Modern web sites and web applications include a significant client-side component written in the JavaScript language and interpreted by the browser. The client-side code can manipulate the the HTML document via the Document Object Model (DOM), allowing, for example, dynamic page updating, page animations, enhanced controls, and other interactive elements. More generally, JavaScript provides a portable Turing-complete execution model for a high-level scripting language augmented with the capabilities of the browser. The popularity of systems such as Google Documents shows that even complex applications, such as office tools, can be highly effectively implemented in this model. YouOS demonstrated the extremes the model enables, creating an extensive desktop environment within the model.

Because of the centrality of JavaScript, an important focus of research and development has been on how to execute it faster, including projects such as the Google Closure Compiler [5], which tries to increase JavaScript efficiency and eliminate potential bugs. This is challenging because JavaScript is a dynamically typed language with high-level features, such as eval, that generally require interpreted execution. However, it is also important, as the execution model from the perspective of a single page is an event-driven model without threads. While this may simplify application programming because concurrency is not exposed to the developer, it means that a badly-written event handler can block others, degrading the user experience, and in the worst case can result in the browser presenting an "unresponsive script" warning to the user. A well-written event handler running too slowly can result in similar issues.

On a mobile platform, the questions of power, energy, and battery lifetime complicate matters. Slower execution of proper JavaScript may reduce power, and given that internal event generation drives JavaScript execution together with external events, slower execution might also reduce energy and enhance battery lifetime due to less work overall. Finally, slower execution of buggy or peripheral JavaScript might reduce its ability to waste energy and degrade the user experience. Previous work [16] has shown that total JavaScript energy (from transmission and rendering) constitutes a significant portion of energy used during mobile browsing, e.g. 16% on Amazon.com or 20% on YouTube.com.

We claim that on mobile platforms JavaScript interpretation is generally faster than is necessary to maintain a satisfactory user experience, and we propose that JavaScript implementations include a user-configurable throttle. For many sites and users, the throttle can simply be set to a uniform level that will reduce power compared to today's open-throttle setting, while not affecting the user experience. For some sites and users, the throttle is needed so that the user can determine his own trade-off between power and experience.

To evaluate our claims we developed a web proxy system, named JSSlow, that rewrites JavaScript passing

through it using the continuation-passing style. Leveraging this, our system introduces what are effectively "sleep" calls into the JavaScript interpretation process, despite the fact that JavaScript has no native sleep functionality. The occurrence of these sleep calls and their arguments constitute the JavaScript throttle mechanism. The throttle is itself set via a global variable.

JSSlow is intended as a proof-of-concept for a JavaScript throttle, and certainly other alternatives, such as changes to the JavaScript interpreter itself, may be simpler. JSSlow does have the benefit of allowing any user direct access to the throttle functionality simply by using our proxy. It also doesn't require any client or server changes, making it easier to study our overall claim.

Using JSSlow, we conducted three studies with an Android mobile phone as the client device. In the first study, we evaluated the power and energy savings that the throttle can provide when faced with buggy JavaScript and advertising JavaScript. Not surprisingly, the benefits are quite substantial. In the second study, we considered the power and energy savings that the throttle, with a default setting, can provide for the top 120 web sites. We found an average power savings of 6%, with some sites showing significantly higher savings. That is, the introduction of JavaScript throttle reduces power across the board and has particularly strong effects for buggy or exploitative JavaScript.

Our final study was a user study in which we had participants come to our lab and carry out tasks using several common web sites/applications on an Android phone, both with and without JSSlow throttle (with the default setting) active. During the tasks, we measured both articulated user satisfaction and power. We found that for common tasks like commenting on articles or Facebook, there was little difference in satisfaction between the two cases. For fine-grain interactive tasks, such as a game, the impact on user satisfaction varied considerably, suggesting that making the throttle visible in such cases would be preferable.

The primary contributions of this work are as follows.

- We identify an opportunity for reducing mobile device power by throttling JavaScript execution.
- We present the design and implementation of a mechanism for such throttling, the transcoding JSSlow web proxy.
- We show that JavaScript throttling can have a major effect on reducing the power of buggy and advertising JavaScript, reducing these by 52% and 10%, respectively.
- We show that JavaScript throttling, at a default, non-aggressive level, leads to significant power reduction for the top 120 web sites. The average power is reduced by 6%.
- We show that JavaScript throttling, at a default, non-aggressive level, has little effect on user satisfaction for several common tasks. It does have an effect on highly interactive tasks, but the effect appears to be highly user dependent, suggesting that the throttle setting needs to be exposed to the user in such cases.
- We discuss alternative methods for implementing JavaScript throttling.

2. WHY SLEEPY JAVASCRIPT?

JavaScript is a dynamically typed, object-oriented scripting language developed by Brendan Eich. It is implemented by modern web browsers in order to create rich, dynamic web pages. The syntax and semantics of the language are a superset of ECMAScript Edition 3 [3], although each browser implements its own interpreter, and therefore, version of the language. The typical model for JavaScript execution is event-driven, allowing web pages and applications to respond to user input. JavaScript follows the common model of registering event hooks such as `onClick()`, and assigning event handlers to be invoked as a result of events coming in.

We argue that JavaScript is being run faster than necessary on a mobile platform, and that it is possible to reduce power with little to no effect on the user experience simply by slowing down this execution. While our focus is on reduced power, we feel that reducing power may also lead to reduction in energy (and thus an increase in battery lifetime). This is counter intuitive given that the energy of a task,

$$Energy = Time_{run} \times Power_{run}$$

is often believed to have the property that the power grows more slowly than the time decreases with increasing execution rate. The implication is that given fixed tasks, the best strategy is to execute them with as high a rate as possible. Computational sprinting [12], where execution rate is raised even beyond sustainable thermal limits for brief periods of time, is the extreme example of this "race to finish" approach.

This analysis makes sense for some workloads and tasks, for example, the image recognition kernel in [12], and perhaps the UI controls on the platform (Section 6.5), but it may not capture the event-driven nature of JavaScript execution for a web page or application. Here, the workload is likely to be continuous, and in many cases, users may not be concerned with the speed at which individual tasks complete, or even if they complete. Most importantly, fast execution of JavaScript workload may well introduce additional tasks, which themselves require more energy to execute.

Consider a user visiting a page. The user will interact with the page for a given interval of time, which we call a *dwell time*. Given a fixed perceived performance of

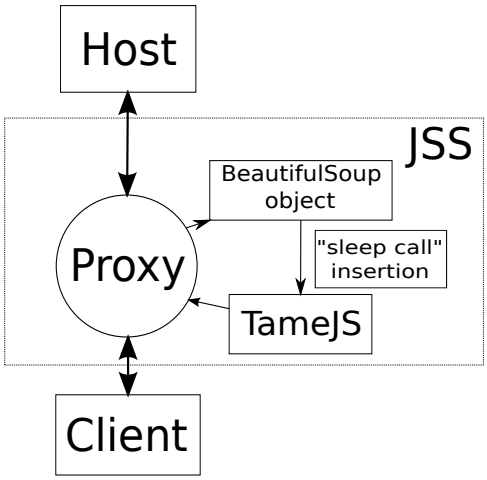


Figure 1: JSSlow system design.

the page, the user's interaction with the page will not change. However, if we lower the execution rate of the JavaScript, we will reduce the power, and reduce the amount of work that needs to be done over the dwell time. Suppose there is an advertising animation on the page. Slowing down the animation's execution will be unlikely to change the user's interaction or satisfaction, but we will have fewer updates to perform over the dwell time, and each update will be done at lower power.

Given this, our goal is to convince the JavaScript interpreter to slow down in such a way as to reduce power. To do the latter, we need it to yield to the kernel, which, if there is no other work available, will put the processor into a lower power state. In other words, we need to convince the JavaScript interpreter to go to sleep.

3. JSSLOW PROXY

The proxy that we have developed, JSSlow, works by examining the body of HTML pages that pass through it, identifying key structures of interest in both embedded JavaScript code as well as in referenced JavaScript code. Within these structures, for example loop bodies, the proxy inserts the equivalent of calls to a "sleep" function. In this way, we are able to slow down the execution of scripts by ensuring that any code likely to be run repeatedly will have to run our sleep call as well. There were two key difficulties in applying this transformation to scripts, (1) JavaScript does not contain a native sleep call, and (2) the JavaScript code is run a single-threaded context. As we noted in the introduction, this is in keeping with JavaScript's event-based execution model.

3.1 Simulating sleep

A sleep call could be simulated in JavaScript's event-based execution model through the use of `setTimeout` timer mechanism, but, as we will show, this approach requires code size comparable to execution steps and thus is unsuitable for looping/recursive code, which is exactly the kind of code we want most to improve. Our adopted approach is to use the continuation passing programming model, explained in Section 3.2, which is supported within JavaScript, albeit not widely used. In essence, our invocations of sleep involve creating a continuation, and then passing that continuation as a timer handler that will be invoked once the desired sleep period has expired. Until the timer event fires, the JavaScript engine can execute independent threads for other pages, and, if it is nothing to do, it will internally await a timer event, using an OS-level `sleep()` or `select()` call. It is the time our page spends in these system calls that reduces the power consumption of the page.

Transforming ordinary JavaScript code into the desired continuation-passing form is challenging. To simplify the process, we leverage TameJS, which has been designed to augment the *server-side* JavaScript language with more straightforward ways of using continuation-passing. By leveraging TameJS on the *client-side*, we can achieve our goal in several steps. First, we parse the JavaScript and identify where we want to inject sleep calls. Next, we inject those calls using the TameJS continuation passing syntax. Finally, we use TameJS to translate back to standards-compliant JavaScript which we then hand to the client. Figure 1 illustrates the process.

3.2 Continuation passing

Continuation passing, a term coined by Steele and Sussman in their paper on the Scheme programming language [15], is a style of programming in which control flow of a program is explicitly managed with continuations.

The core idea of a continuation is that it captures a complete execution state in such a way that it can be restored at a later time. The continuation is made available at the programmatic level; that is, the program can itself operate on its own continuations, for example explicitly restoring them. Of particular note is the so-called "current continuation", which represents the current execution state. This is often augmented with the notion of "call with current continuation", which allows for control-flow parallelism within the program. As an example, an iterator that operates over tree might be written recursively. The iterator's `next()` function would use a call with current continuation to restore the recursive execution context, make one step on the tree, and then return that value with another call with current continuation back to the caller of `next()`. In this example, continuations allow us to marry the straightforward implementation of a recursive traversal of the tree with in-line iteration over the results of this process. The

sharp-eyed reader might note that this process looks a lot like a multithreaded process, and there is in fact an equivalence between cooperative multithreading and continuation-passing.

In the continuation passing style (CPS) of programming, each function takes an additional argument, namely the continuation that the result of the function will be passed to. CPS makes several things more explicit than in "direct style"—the flow of code is immediately obvious since the return of a procedure is directly defined, order of evaluation is explicated since all expressions must be evaluated from the innermost part out, and all calls become tail calls which allows for optimizations.

Since each function must be augmented to include an extra parameter, trying to write code in CPS can be error-prone. This approach is sometimes implemented in compilers such as in Appel and Andrew's book "Compiling with Continuations" [1] or in TameJS.

3.3 TameJS

TameJS [11] is a "source-to-source translator that outputs JavaScript" developed by OKCupid as a JavaScript implementation of the C++ Tame framework [7]. It produces standards-compliant JavaScript that can then be run by any JavaScript engine. TameJS was developed with the NodeJS platform [6] in mind. NodeJS attempts to extend JavaScript to the server side, allowing an application developer to build web applications with a single language and execution model. In the server context, especially due to network delays and concurrency among many users, the limited concurrency and cooperative, event-driven execution model of JavaScript can be particularly limiting. TameJS attempts to address these limitations by making the continuation passing style much easier to use within JavaScript code.

TameJS adds two new primitives to JavaScript to facilitate programming with continuations:

- the `await` primitive, which is essentially "call with current continuation", and
- the `defer` primitive, which essentially invokes the continuation that was passed during the call, returning execution state back to the caller.

The combination of `await` and `defer` allow for asynchronous callback code to operate like regular sequential code. The following code illustrates how `await` and `defer` are used to introduce pauses in execution:

```
for (var i = 0; i < 10; i++) {
  await{setTimeout(defer(), 100);}
  console.log ("hello");
}
```

The result of the above code is that "hello" is written to the console 10 times, with a delay of 100ms between each write. The block of code handed to `await` immediately executes, setting a timer that will fire in 100 ms. The `await` call also packs up the current continuation, and passes it to the code. The handler the timer event will invoke consists of the invocation of `defer`. It is not until the `defer` is invoked that the previously packed up continuation is unpacked and restored. At this point, execution continues at the next statement following the `await` statement. Most JavaScript engines will implement the timer delay through `select()` system calls, giving the kernel the opportunity to put the processor in a low-power state if there is no other work to do.

If this code were written without the `await/defer` combination, "hello" would immediately be printed to the console 10 times. Because `setTimeout` is a non-blocking function, the JavaScript engine does not wait for it to return before going on to run the rest of the code. The only way to achieve the desired pause would be to pass `console.log` as the callback function to `setTimeout`. This approach can be applied easily in places where we only want to introduce a single delay, but in order to apply it even to the simple code example above, each iteration of the loop would need to be its own successive callback to a deeper `setTimeout`. That is, we would have to code the 10 iterations of the loop nest separately.

In general, to implement iterated, or nested continuation-passing as in the above example in standard JavaScript would require that we syntactically unroll the iteration or nesting. The beauty of the TameJS `await/defer` extensions is that this is not needed, which allows the implementation of continuation-passing programs in the straightforward style available in other languages that support continuations.

3.4 JSSlow

The JSSlow proxy is an extension of a proxy we previously developed to inject JavaScript-based user interfaces that overlay themselves on existing web sites. The initial proxy, which itself was an extension of the Tiny HTTP proxy for Python[1], was designed to support studies of user-centric network scheduling for the uplink of home routers and is described in more detail elsewhere [8]. In this paper, we generally do not make use of its ability to to inject new user interface components into the user experience, but rather as a framework within which we we can carry out transformations of existing JavaScript code. It implements standard SOCKS proxy semantics and so can be used with any web browser simply by configuring the browser's proxy configuration to use it.

JSSlow observes the HTML body of any response that goes through it, and then analyzes and alters that body before returning it to the requesting client. When it has received the body of the page, it parses it using Python's BeautifulSoup library, resulting in a Beauti-

[1] `http://www.oki-osk.jp/esc/python/proxy/`

fulSoup object which is an abstract syntax tree (AST). The AST representation and the library then allow us to search and transform the tree. The core operation we do is identify JavaScript code blocks into which we will inject `sleep` invocations.

Sleep macro implementation.

Our `sleep` is implemented assuming the existence of TameJS, and takes the following form

```
await { setTimeout(defer(), g_slow); }
```

where `g_slow` is a global variable indicating the sleep duration. We generate a special configuration block to set `g_slow` at page load time. It can be adjusted at any point later, for example via a user interface that can also be injected by the JSSlow proxy. It is important to realize that while `g_slow` is a throttle, it is a very course grain one. Any significant `g_slow` value will typically cause the engine to do a yielding system call, which has a significant effect on power, even if `g_slow` is a quite small, nonzero value. Also note that it is the combination of `g_slow` and the locations at which the `sleep` macro is introduced that constitute the overall throttle. The sleep macro can also be changed to allow for complete deactivation of the sleep functionality, resulting in behavior virtually identical to the unmodified code. We discuss alternative methods of throttling in Section 8.

Sleep macro injection.

The following pseudocode illustrates the process of transforming the AST to include invocations of our sleep macro:

```
// create AST of the incoming html
html-copy = BeautifulSoup(incoming-html)

sleep = "await { setTimeout(defer(), g_slow); }"

// iterate over all <script..>..</script> fields
for script in html-copy:
    script-copy = script

    // fetch local scripts
    if script.has_tag("src") && src.is_local():
        script-copy = fetch(src.address)

    insert-at(sleep, "while")
    insert-at(sleep, "if")
    insert-at(sleep, "for")
    insert-at(sleep, "function")

    try:
        script-copy = tame-compile(script-copy)
    except:
        // if compilation failed, just skip
        continue

    script = script-copy

return html-copy
```

Figure 2: Testbed as seen by user.

In essence, we transform all scripts that are inlined in the page or that are referenced by the page and exist on the same site. Each script has sleep macros introduced into the bodies of its of its control structures (while, if, for, and the entry points of functions). The transformed scripts (including the referenced scripts) are then inlined into the page. If a transformation or inlining of a script fails, we use the original script.

We currently avoid transforming referenced scripts that are not local to the originating page's site. This is because we found that many such scripts would require more subtle transformations in order to be successfully inlined. For example, if a script containing the line `document.write(''</script>'')` were fetched, escaping would be needed to avoid having the string be interpreted by the HTML parser as a script tag. Because of this limitation, we currently under report the potential effect of JavaScript throttling, particularly for throttling advertising and marketing JavaScript, which is almost all non-local. We say more about this missed opportunity in Section 7.4.

Once we have finished adding `sleep` invocations to a script, we run the modified script through the TameJS compiler. TameJS transforms all of our uses of the `sleep` macro back to standard JavaScript, but also includes a `require` call that brings in the TameJS library code. We then replace this with the TameJS library code itself (252 lines of JavaScript), resulting in a script that is as self-contained as the original script.

The transformed script is then inserted back into the body of the page and the proxy continues on to the next script. If any error occurred in the compilation of the transformed script, the original script is instead left on the page. Once all scripts had been processed, the script prepends the initialization code, including the global declaration of the sleep duration.

4. TESTBED

To carry out the studies described in this paper, we developed a simple testbed centered around a Google

Galaxy Nexus Android phone. The purpose of the testbed is to enable instantaneous power measurement of the phone while allowing the phone to be used either directly by a user or in an automated fashion, following a script. The phone is used on battery power in all cases. The phone is configured to use the JSSlow proxy, which is also part of the testbed. Figure 2 shows the appearance of the part of the testbed seen by the user. The box under the phone is a current clamp. The phone uses a WiFi for connectivity, attaching to an access point under our direct control.

4.1 Power measurement

Instantaneous power is measured by connecting a conductor directly from the positive terminal of the battery to the positive terminal on the phone, with an insulator directly between the two ends of the conductor, creating a loop external to the phone that current can run through. In order to allow the battery to still fit into the phone (allowing easy connections between the other 3 terminals) a thin conductor was needed. We use layers of metal foil as the conductor.

With the external loop directly accessible, we are able to use a current clamp to measure the current flowing out of the battery. Since this method uses an indirect method of measuring current flow, it has a very low impact on the current itself, which results in accurate readings with little perturbation.

The current clamp we use is a FLUKE i30, which has a maximum current of 20A RMS, and an output of 100 $\frac{mV}{A}$. The device has an accuracy of $\pm 1\%$ at ± 2mA and a resolution of ± 1mA. The output of the current clamp is fed into a RadioShack 22-812 digital multimeter (DMM), which provides access to readings via an RS232 serial port. This port is wired via a RS232 to USB converter to a monitoring PC. We record DMM readings with QtDMM, software designed for communication with various DMMs. Readings are taken at 10 Hz.

It is important to note that the combination of the phone, current loop, current clamp, etc, remains hand-portable—the user can still hand-hold the phone.

In order to arrive at power, we must multiply the current read from the current clamp by the voltage of the battery. The battery Voltage is subject to some noise, but is relatively stable at 3.8V. This gives us a conversion from voltage read by the current clamp to power of

$$W = V_{clamp} \times \frac{A}{100mV} \times \frac{1000mV}{V} \times 3.8V$$

4.2 WebProxyAutomator

Our testbed can be used non-interactively, consecutively visiting and measuring the power of a list of sites with and without the proxy enabled. To facilitate such studies, and other non-interactive experimentation, we created an automated testing application, WebProxyAutomator (WPA) for use with the Google Android operating system. WPA creates a `WebView` [4], which is an instance of the built in Android web browser, and then makes successive calls to load a new page from the list at selected intervals, tarrying at each page for a user-selected visit time.

Scenario	Proxy On [W]	Proxy Off [W]	Diff [%]
Bugs	1.599	3.325	-52%
Ads	1.332	1.472	-10%

Figure 3: Power reduction for infinite loop bugs and advertising. Average power over 10 s.

5. OPPORTUNITY

To evaluate the potential power savings that are available through JavaScript throttling, we used the testbed to study a range of sites without considering user interaction. This work included a study of the effect on buggy JavaScript and advertising, and a study of the effect on the top 120 most visited web sites.

5.1 Buggy JavaScript and advertising

The effects of JavaScript throttling on power are most extreme in cases where the JavaScript in question is buggy, typically resulting in an event handler being especially long, or going into an infinite loop. To evaluate this, we crafted an intentionally buggy test site with the following JavaScript,

```
var i = 1; while(1) { i *= -1; }
```

which is run before any text is displayed on the page. The result is that the browser becomes unresponsive and the text never appears. When viewing the same page through JSSlow, however, the text immediately renders, the browser is responsive, and the power consumption was less than half that of the untransformed site. This represents the upper bound of power savings that are possible with JavaScript throttling as implemented in JSSlow. This result can be seen in the first row of Figure 3. The power is reduced by 52%, which bounds the opportunity for JavaScript throttling.

Advertising makes extensive use of JavaScript, and because ad code is essentially throw-away code, it is more likely to have bugs. To evaluate the effect of JSSlow JavaScript throttling on advertisements, we manually extracted 50 ads from visits to a random subset of 10 sites from the Google's top-1000 sites (more details on this list in Section 5.2) and ran them through JSSlow, the results of which can be seen in the second row of Figure 3.

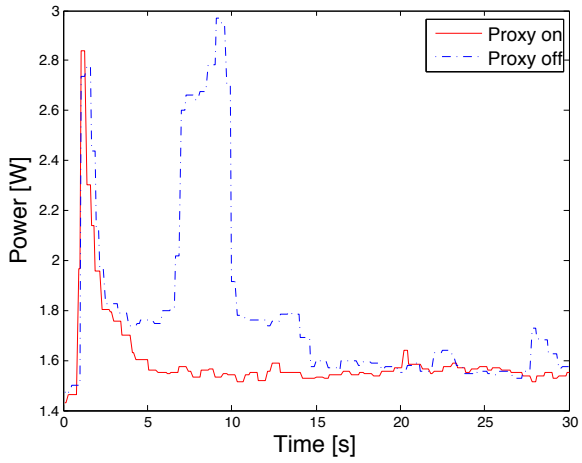


Figure 4: Example power signatures with and without JSSlow.

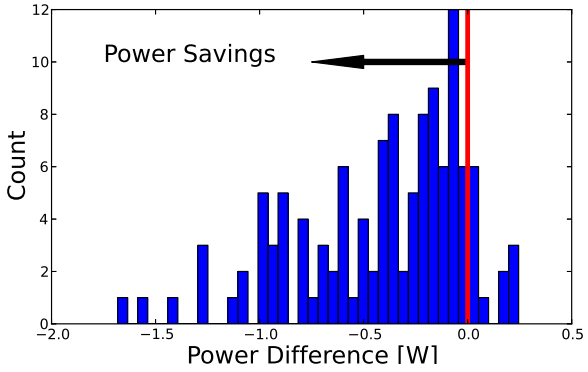


Figure 5: Distribution of absolute power difference between JSSlow-throttled sites and native sites. Average power over 10 s.

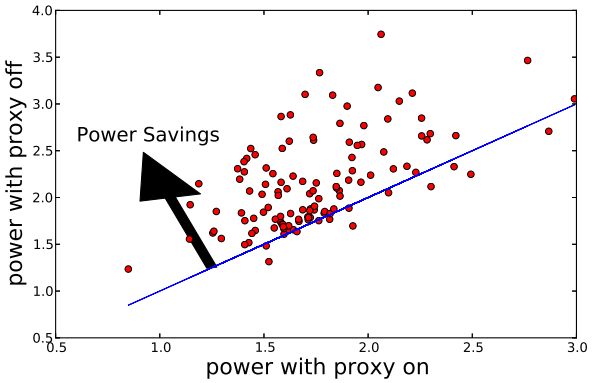


Figure 6: Comparison of absolute power with and without the use of JSSlow. Average power over 10 s.

5.2 Top 120 most visited sites

To evaluate the effects of JSSlow-based JavaScript throttling in a real world environment, but without user interaction, we used the WebProxyAutomator (WPA, described in Section 4.2) to programmatically load popular web sites with and without JSSlow enabled and measured the difference in power. The specific set of sites we we loaded were selected from the Google AdPlanner list of the top 1000 sites[2]. Because we ran the phone on battery power, we were only able to visit the top 120 sites before battery depletion.

Each site was visited an average of four times with the proxy on, and four times with the proxy off. A visit included a load and dwell time of 10 seconds, that is, from the time the site started loading to when we advanced to the next site was 10 seconds. The suite of tests was covered in about 2.7 hours, at which point the battery was depleted. During the load and dwell time, the testbed measured the power at 10 Hz. Thus, for one visit, we have 100 measurements. We refer to this as a *power signature*. We can compare power signatures with and without JSSlow, with an example shown in Figure 4, but we generally compare averages over the duration of the power signature.

It is important to point out that this automated testing involved no interaction with the contents of the page, thus removing the possibility of running JavaScript from user-interaction driven events such as `onClick()`

Across our sample set, we found an average power reduction of 6% when using JSSlow. However, there is considerable variation across sites. Figure 5 shows the distribution of the difference between the throttled and non-throttled runs for the set, while Figure 6 plots the absolute power measured for the paired runs. We note that even though there is an overall power savings, there are sites for which we actually increase power. It is important to point out that JSSlow can be disabled in these cases, simply by setting the `g_slow` parameter to zero.

It is important to note that the power savings also include the additional power used to fetch the larger content in the case of the JSSlow transforms being active. For each script, the JSSlow transformation expands the script's size by a factor of two, and adds about 252 lines of library code. In other words, a script of n lines expands to one of $2 \times n + 252$ lines. These results in more energy spent in receiving the script.

As described in Section 3.4, JSSlow does not currently throttle non-local scripts, such as ads. Given that we found that explicitly handling ads lead to power reductions of 10% (Section 5.1), it seems probable that further power reductions will be possible for the top-120 sites once this functionality is complete. Given these results, we believe the baseline average power reduction due to simple inclusion of a JavaScript throttle with a

[2] `https://www.google.com/adplanner/static/top1000/`

Users		
Age	17-25	15
	25-35	4
	35-45	1
Gender	Male	13
	Female	7
Area of Study	Computer Science	10
	Science / Engineering	3
	Liberal Arts	2
	Other	5
Length of smartphone usage	Never	1
	0-1 Months	2
	6 Months - 1 Year	4
	1-2 Years	5
	2+ Years	8
Smartphone Type Owned	None	1
	Android	9
	Blackberry	2
	iOS	7
	Other	1

Figure 7: User study demographics.

default setting, ignoring interaction, is 6–10%.

6. USER STUDY

JSSlow aims to reduce the power consumption of JavaScript interpretation by slowing it down, which could potentially have an inverse impact on user satisfaction with the web sites running those scripts. We designed and ran a user study that would evaluate the effect of JSSlow throttling on both user-perceived satisfaction, as well as measuring the change in power consumption during real-world usage.

6.1 Subjects

Our IRB approval allowed us to advertise our study at various locations throughout the Northwestern University campus. We selected the first 20 students who replied to participate in the study. Demographics of this population are presented in Figure 7. The study was designed to take one hour, each user was compensated for their time with a $20 gift certificate.

6.2 Tasks

The purpose of each task was to approximate the normal usage of a mobile device by a user, as well as to provide various scenarios for which JSSlow might have an effect. The tasks were split broadly among 2 categories: *low interactivity* tasks, and *high interactivity* tasks. What we are trying to capture with these categories is the level of interaction the user will have with JavaScript. The sorts of interaction in low interactivity sites are generally restricted to tasks of the form of leaving a comment, or navigating to a certain part of a site, if the navigation is implemented in JavaScript. High interactivity tasks are ones in which the user is constantly interacting with the script in some way, such as with controls in a game.

CNN and Facebook were selected as low interactivity tasks that are representative of common sites that users visit on their mobile devices. CNN stands in for news and blog-type sites, while Facebook is the exemplar of a social network site.

High interactivity tasks were harder to find, as JavaScript is still relatively new to interactive applications, and browsers do not necessarily implement the same subsets of JavaScript technologies, or with similar performance (Internet Explorer, for example, does not perform well on WebGL benchmarks). Additionally, there are relatively few JavaScript applications that are written to be used with a mobile interface. For these reasons, we chose a very simple application for users—a game of "Snake" written in HTML5 [3]. The game is controlled by a user swiping the screen in the direction they want the snake to move, with the stipulation that the snake can only ever make 90° turns.

6.3 Methodology

We designed an double-blind intervention study in which the subject would either have their code slowed down through the proxy or not, but not be made directly aware of which state they were in. Additionally, the proctor administering the exam would not be aware of the current state either, but only collect information regarding the subject's current satisfaction.

The device that the subject was given to use during the study is the same one described in Section 4. We now consider the flow of the study from the perspective of the subject and the perspective of the proctor.

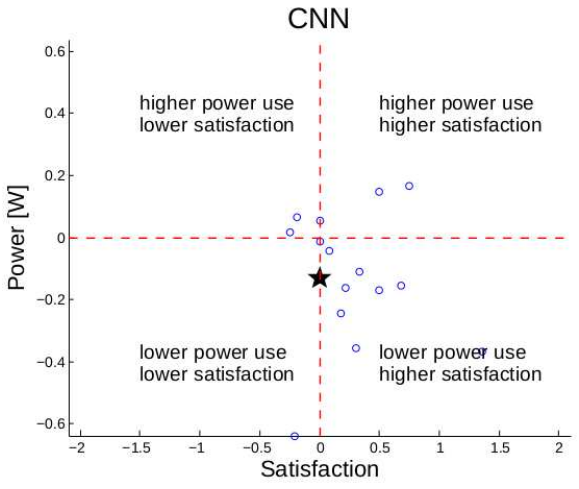
Subject.

When a subject first arrived, we had them fill out a questionnaire designed to determine their level of knowledge and comfort with a modern mobile device. For the duration of the study, the subject only interacted with web sites in the device's browser, which minimized the amount of experience and knowledge needed of the Android platform.

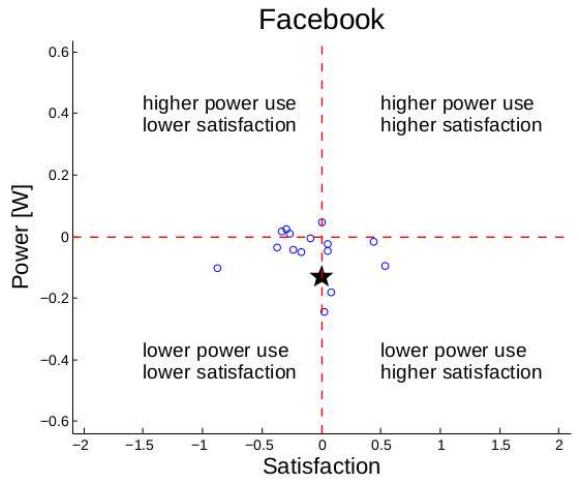
The user was then given 5–10 minutes to browse to any site. The purpose was to get them familiar with the device, as well as to the normal speeds of the network and browser. The device was connected to the proxy for the entire duration of the study, throttling was turned off during each familiarization phase. Once the user finished familiarizing themselves with the device, they were prompted to give his current satisfaction with the performance of the device, in order to establish a baseline for that subject.

At this point the device would be pointed to a landing page, which included links to each page the user would visit over the course of the study. For both of the low interactivity sites, a dummy account was already logged

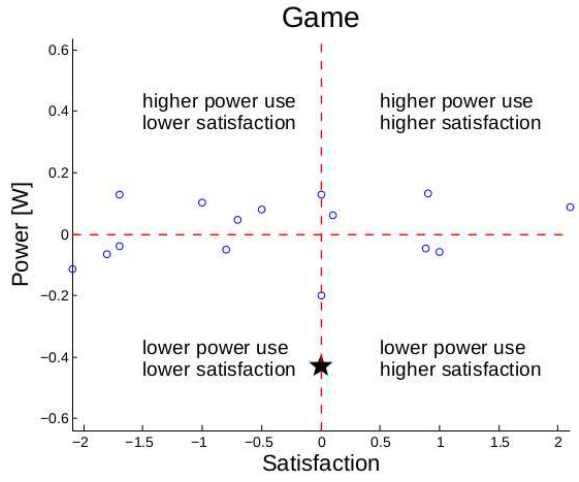
[3] `snake.alexthorpe.com`



(a) CNN



(b) FaceBook



(c) Game

Figure 8: Absolute difference in power versus absolute difference in reported satisfaction. Averaged over task intervals. The star represents power difference without interactivity.

in, so as not to require the user logging into a personal account, eliminating any privacy concerns.

For CNN and Facebook, the user would visit an article or page, respectively, read through the content, and then post a short comment. During the course of this process, the user would be periodically prompted to indicate their level of satisfaction with the site. They were asked to verbally express their satisfaction on a Likert scale of 1–10, where 1 represented complete dissatisfaction, and 10 represented complete satisfaction.

Once finished with the low interactivity tasks, the user would be pointed to the high interactivity task, and given 5–10 minutes to familiarize themselves with the speed and reactivity of the game. During this time, the proxy was again off. Once this time was up, the user would play the game for two 5-minute sessions. As in the previous phase, the user was periodically prompted to verbally provide his current level of satisfaction.

When both sessions had been completed, the user was asked to fill out an exit questionnaire, indicting his overall satisfaction with the performance of the device for the entire study. They were also asked to note any times during which they noticed changes in performance.

Proctor and proxy.

The study was designed to be double blind—neither the user nor the proctor were aware of the state of the proxy. The proxy had a user-study mode that would expose certain controls to the proctor, allowing it to transition to the appropriate state for each phase of the study. During each interactive phase, the proctor would prompt the user to verbally indicate their satisfaction every 30 seconds, and note this figure down.

The proxy would start in a non-throttling state, so as to allow the user to get accustomed to the baseline behavior during the initial familiarization phase. Once this was done, the proctor would send a signal to the proxy, making it transition to the low interactivity state. During this state, the proxy would randomly choose whether or not to throttle each site that was loaded. The proctor would send signals to the proxy for when the user started each site, allowing careful time stamps to be kept. At the end of this phase, the proctor would send the proxy another signal, putting it back into the familiarization state.

For the first game session the proxy would randomly choose whether or not it would throttle JavaScript, and it would then choose the opposite of that choice for the second run.

During the entire study, the instantaneous power draw was being measured and recorded. This information, combined with the time-stamped logs and satisfaction results allowed us to extract the average satisfaction for each site in each task, the average power used during the time spent on that site, and whether or not the site

Task	Average Difference Off-On			Average	
	Avg	StDev	Conf	On	Off
CNN	0.29	0.39	0.12	7.84	7.54
FB	-0.11	0.34	0.11	7.18	7.29
Game	-0.26	1.24	0.39	5.39	5.65

Figure 9: Average absolute difference in satisfaction levels with proxy on or off, and average absolute satisfaction levels with proxy on or off.

Task	Average Difference Off-On [W]			Average [W]	
	Avg	StDev	Conf	On	Off
CNN	-0.12	0.22	0.06	2.08	2.20
FB	-0.05	0.08	0.03	1.88	1.92
Game	0.013	0.10	0.036	2.26	2.25

Figure 10: Average absolute difference in power (over task interval) with proxy on or off, and average absolute power levels with proxy on or off.

had been throttled.

6.4 Results

We conducted our study over a total of 20 subjects, and were able to get satisfaction results from 19 subjects, as well as power readings for 15 subjects. We report on these 15 in the following. To account for any anchoring effect due to the user-based interpretation of satisfaction, we did our analysis based on the differences in satisfaction, paired by user. We looked at the values of the absolute differences, as well as the relative differences.

The results of the study are presented in Figures 8 through 12. Figure 8 plots the differences in power versus the differences in satisfaction for each user between slowed down and normal script execution. The figure is split into 3 plots, showing the results for each task. We also show the average power savings in the case of no user input, indicated with a black star.

The tables in Figure 9 and Figure 10 present the findings in absolute difference. For each task, the satisfaction ratings and power usage for slowed down and normal site performance are compared for that user, and then averaged across all users. We also calculate the standard deviation, and 95% confidence interval for the average value that we found. The last two columns show the average value of satisfaction with the proxy slowing down JavaScript execution across all users, and the average value of satisfaction with JavaScript running normally.

The tables in Figure 11 and Figure 12 present the same data set, but consider the percentage difference in satisfaction and power usage, rather then the absolute difference. For these values standard deviation, and

Task	Avg [%]	StDev	Conf
CNN	4.24	5.24	1.67
FB	-1.10	5.55	1.81
Game	-0.18	33.10	10.52

Figure 11: Relative difference in satisfaction.

Task	Avg [%]	StDev	Conf
CNN	-5.24	10.00	2.92
FB	-2.38	3.61	1.05
Game	0.85	4.55	1.33

Figure 12: Relative difference in power.

95% confidence interval are also calculated.

6.5 Analysis

We consider the data from our study using our two broad categories: low-interactivity and high-interactivity tasks. We consider the data from these categories separately since users may have different expectations for how fast each operates. We are testing whether we can proceed power and energy reduction during interactive use, without effecting the satisfaction of the user.

Figure 9 is telling us that there is little change in user satisfaction when the proxy is applied in the low interactivity tasks. The confidence intervals suggest that for the Facebook score the detected difference is not meaningful, while for CNN, the detected difference is statistically significant but very small. On the other hand, for the high interactivity task, we see large variation, but with both increased and decreased satisfaction, suggesting that response to performance in high interactivity cases is highly user-dependent. While not statistically significant, the difference between the proxy on and off is quite small. Figure 11 tells much the same story. If we average all low interactivity results together, we get a change in satisfaction of 1.6% between the proxy being on versus it being off.

Figure 10 shows statistically significant power savings for the low interactivity tasks. Figure 12 presents the relative differences. Average all of the low interactivity measurements, we get average power saving of 3.8%. This number is worth comparing with the top-120 study (Section 5.2 where we found a 6% savings, without any user interaction.

For the high interactivity task we find a small increase in power consumption, although it does not rise to statistical significance. Nonetheless, this might seem impossible and contradictory with our automated testing-based finds, but we now consider an explanation.

Impact of the interactive governor.

The phone that we used was running Android OS version 4.0.4, which uses the Linux "interactive" CPU

frequency governor, whose goal is to provide as smooth a user interface as possible. The governor accomplishes this by ramping up the processor to the maximum power state on any human input, and then evaluates past load to slowly scale down power state [17]. The assumption is that once user interaction has been initiated, more user interaction is likely to happen, and the system should be able to process / respond to that interaction as quickly as possible.

Figure 13 shows the instantaneous power draw on the battery in 3 scenarios: (1) a site is loaded by the browser, rendered, and allowed to run for 30 seconds, (2) the same site is loaded and rendered by the browser, but after 25 seconds a user begins typing a comment, and (3) the same site is loaded and rendered by the browser, but after 20 seconds the user inputs random swipes (continuous contact with the screen). We can see that the power draw during user interaction is on the order of that of the initial page fetch and load in the case of typing, and about half of that in the case of random swipes.

Because of the interactive governor, there will be increased power usage during any time a user is interacting with the device. We claim that the more frustrated a user became with the decreased responsiveness of the game application, the more often they would press the screen, trying to correct errors that had been committed during gameplay because the interface had applied their previous action too late. This effect will also be noticeable in any situation where a user has to scroll through a page often, or has to follow multiple links to get to the content they are looking for.

Power versus satisfaction.

In Figure 8 we compare the change in satisfaction to the change in power draw for each user, and plot them for each task. We have included dotted lines that separate the space into 4 quadrants. If an increase in power savings came directly at the cost of satisfaction (and vice versa) we would expect clustering of points in the upper-right lower-left quadrants, as a positive difference in satisfaction is good, and a positive change in power usage is bad. These plots include data from all users, places where no data was collected is represented with a difference of 0.

In the CNN results, most of the points are clustered in the lower-right hand of the plot, with a few outliers. This is interesting, in that there were very few users who reported lower satisfaction with slowed down execution. In the Facebook plot, we see a similar distribution of power difference, though at a smaller scale. In this task however, there was much more variance in the change in satisfaction, users reported both positive and negative changes in performance.

This could be partly attributable to the fact that the user had to click through more links to get to the full text of the article. Users were told not to let load time influence their ratings of satisfaction, but they could have taken that factor into consideration anyways.

The plot for the game task has results scattered throughout each quadrant, suggesting that each user's perception of performance is highly varied, and argues for allowing the user to control whether or not a slowdown is applied.

7. LIMITATIONS OF JSSLOW

We now consider some of the limitations of our specific implementation of a JavaScript throttle. The JSSlow proxy is a proof of concept, showing that it is possible to slow down execution with no client or server changes, but, as we show in Section 8, there may be other, finer grain ways of implementing a throttle that avoid JSSlow's implementation issues.

7.1 Continuation passing style

JSSlow's use of CPS to simulate sleep calls introduces limitations related to memory use and performance.

Stack space.

Consider a loop. By transforming the loop body to use continuations, we are basically transforming the loop into a recursion. In the worst case, we then would need to allocate additional stack space for each iteration or block of iterations of a loop, as it is a function call. The TameJS compiler tries to limit stack use via tail call optimization, and, strictly speaking, since the original code is a loop, it must be possible to do this optimization. However, the actual optimizer is unable to detect this situation in some cases. In such cases, the runtime stack grows with number of loop iterations resulting in large memory use, and, of course, the script fails if the stack gets too large.

As an example, using the V8 engine included in Node.js, with a 4 KB runtime stack, and a `for` loop with no additional allocation that TameJS was unable to tail call optimize, the transformed script ran out of memory between 13,000 and 14,000 iterations.

Running out of stack space is not an inherent issue with JSSlow's approach, but is a consequence of unrecognized tail call recursion optimization opportunities.

Performance.

Another consequence of the JSSlow approach of slicing up program execution via continuations is that there is overhead associated with continuation creation and use. To measure this overhead, we compared normal and transformed versions of a `for` and `while` loop. The body of each loop was a non-blocking arithmetic operation, with no allocation. In the transformed version, each loop iteration became a continuation, but there

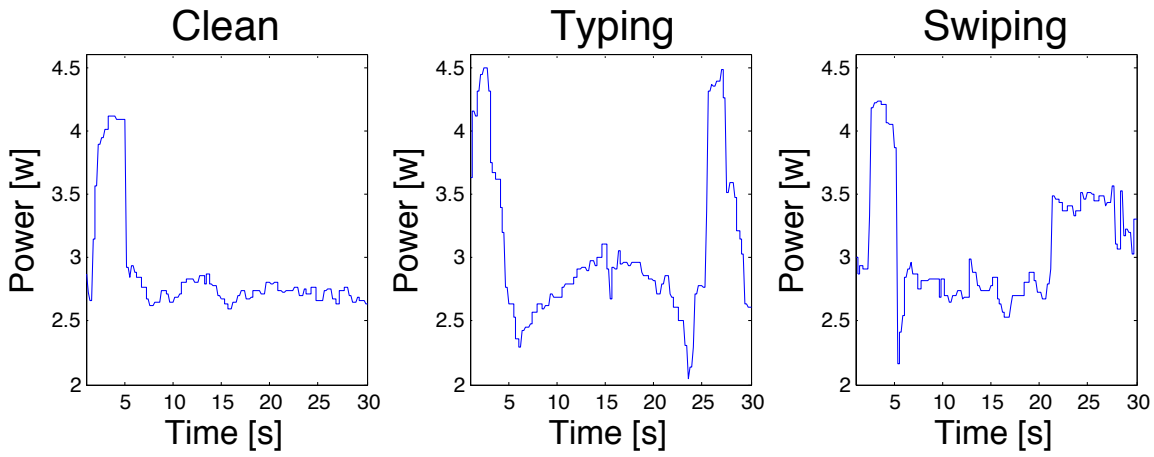


Figure 13: Power signature with and without user interaction. "Clean" indicates the site without any user interaction. "Typing" and "Swiping" indicate these activities are occurring.

was no sleep used during the continuation. We ran 13,000 iterations of the for loop (where tail call optimization was not performed) and 100,000 iterations of the while loop (where it was). For each, we measured the time it took for the loop to complete in the base and transformed code. All code was run on the V8 JavaScript engine included in Node.js using a stack size of 4 KB. The transformed `for` loop run times increased by an average of 2.1%, the transformed `while` loop run times increased by an average of 1.6%.

7.2 Increased content size

Since JSSlow must not only inject additional "sleep" calls into intercepted code, but also transform the resulting code into code runnable by any JavaScript engine, we are increasing the size of the content being delivered to the mobile device. Given that wireless reception of data also affects battery usage, we must be careful to take this into account, as it will have an effect on our energy savings. Our results do this, as we capture the average power over both the fetch and the dwell time. If throttling were implemented internally in the JavaScript interpreter in the client, however, the additional energy cost due to the increased content size could be avoided.

7.3 Energy increases

Our results from the top-120 sites (Section 5.2) indicate that we save energy on most sites by using JSSlow. However, a small fraction of sites actually have increased energy use. Ideally, a tool like JSSlow would be able to deactivate itself for such sites. While we can set the `g_slow` throttle in the JSSlow-transformed code, even when set at zero, the energy costs due to the increased content size and the overhead of continuation-passing execution remain. If throttling were implemented internally in the JavaScript interpreter, however, these cost could be avoided. The interpreter could also adaptively set the equivalent of `g_slow`, testing if a non-zero level actually results in decreased power.

7.4 Missed opportunities in advertising

JSSlow currently inlines scripts, but we have difficulty inlining scripts that come from non-local sites. We therefore have disabled this functionality in most of our studies. However, advertising JavaScript is almost always of this non-local form, and, as we showed in Section 5.1, it is particularly amenable to energy reduction by our transformations. The consequence is that we were unable to apply the JSSlow throttle to most advertising, and thus miss this further opportunity to decrease energy use, and possibly increase satisfaction, for typical sites.

7.5 Course-grain operation

The JSSlow throttle consists of the locations of the injected sleep invocations, and the duration of the sleep intervals. Even with a very small sleep interval, this can have a quite course-grain effect on the performance/power tradeoff. If we inject the sleep macro in every loop iteration, we have the potential of invoking the kernel sleep system call once per iteration. This would slow the loop to one over the minimum sleep interval the kernel provides. In the case of a typical desktop Linux kernel, configured with a 1 ms periodic timer, this would potentially decrease the loop iteration loop to as little as 1 KHz. Having finer grain control over when and how often the JavaScript interpreter thread yields would allow for a more gradual tradeoff between performance and power.

8. RECOMMENDATIONS

Despite its limitations, the JSSlow implementation of the JavaScript throttle, and the results we have derived from using it, are, we believe, sufficient for us to make a set of recommendations about the possible mechanisms and policies of JavaScript throttling.

8.1 Deploy JavaScript throttling

JSSlow and the results we have developed using it show that it is feasible to reduce power and energy on mobile devices via JavaScript throttling with little effect on user satisfaction. Arguably, the limitations of JSSlow's implementation of throttling actually understate the case.

JavaScript throttling could potentially be deployed in an incremental way, using a proxy approach, like JSSlow's, to provide throttling for existing, unmodified client software talking to existing, unmodified sites. Over time, more sophisticated forms of throttling could be rolled out.

8.2 Throttle the engine

The limitations of JSSlow, described in Section 7 could be avoided if the throttle mechanism were embedded into the JavaScript interpreter, or engine, itself. The interpreter main loop could simply decide on whether to yield or sleep on each iteration. This would provide a very fine grain throttle while not changing any JavaScript syntax or semantics. Furthermore, since the throttle would be part of the browser implementation, instead of embedded into the JavaScript code itself, as in JSSlow, there would be no code expansion limitation nor any continuation passing overhead.

We examined two JavaScript engines, V8 and SpiderMonkey, with the goal of evaluating the challenges of such an implementation. In both cases, we were readily able to find good points at which to add throttling.

In the V8 codebase, we added a sleep call to the `Invoke` function, located in the file execution.cc. The `Invoke` function marks a point of entry for execution of code in the V8 virtual machine. Through the use of selective invocation of a `nanosleep()` system call with different parameters, we are able to slow down JavaScript execution with arbitrary precision.

In the SpiderMonkey codebase, we identified the `DO_NEXT_OP` macro, located in the filed jsinterp.cpp, as a location to implement a throttle. `DO_NEXT_OP` is the step in the SpiderMonkey state machine during which the next script opcode is fetched and processed.

To control this mechanism, we could provide programmer-driven policy or a user-driven policy, both of which we describe next.

8.3 Expose the throttle to the programmer

Given a throttle built into a JavaScript engine, as described in the previous section, a natural question is what policy should control this mechanism. It is tempting to simply expose the sleep functionality directly to the developer, but this might conflict with JavaScript's event-driven model, and would probably require the introduction of threading to be effective. Calling sleep in part of the code should not put the entire document to sleep. For example, we would not want to stop user elements from rendering while we wait for non-essential code to run again.

One alternative that we believe would be fruitful is to directly expose the control parameter that the engine is using internally to decide when to sleep and for how long. However, this would have the disadvantage of breaking the abstraction between the JavaScript code and any given interpreter—we would like the programmer to be able to adjust the throttle of any interpreter.

Another alternative, which would maintain this abstraction, would be to codify a set of API functions that can be provided by any interpreter, functions that provide for relative throttle changes, and extremes, such as: `throttle_up()`, `throttle_down()`, `throttle_max()`, and `throttle_min()`. Extending control through an API such as this would allow the application developer to provide input into the throttle-setting process in an engine-independent way, while leaving ultimate control to the engine developer. This approach also has the advantage of not requiring any change to the standard JavaScript execution model.

8.4 Expose the throttle to the user

In the user study that we conducted, throttling applied to the high interactivity task resulted in a high variance of changes in satisfaction. Considering that each user likely has both a distinct perception of performance change, as well as a different expectation of application performance, this makes sense, and argues for exposing a level of control to the user.

In this model, throttling would be turned on by default and set to an initial value that would either be determined by the engine developer, the application developer, or the mechanism we describe in the next section. The user would be presented with a method to change the throttle setting, or turn off throttling altogether, perhaps in the form of a visible button or toggle, or via a physical button, or via a special swipe pattern, or biometrics, or perhaps even via a maneuver that is detectable using an accelerometer. The change in throttle level from the default would be stored locally, so that the user would not have to change this every time they revisited the application. Previous work (e.g. [13, 8, 14, 9, 10]) has shown that by allowing users to set their own level of performance leads to power reductions as well as more satisfied users, and that users are often quite capable of setting such throttles.

8.5 Socialize the default throttle setting

A shared problem with any policy for setting the throttle is how to determine the default setting, which will represent a good tradeoff of satisfaction and energy savings. Users or applications could then modify this setting as needed. One approach might be, for each site, to select a response latency bound for controls (e.g., 15 ms as in [2]), and then select a throttle setting that just barely achieves this. The approach we prefer would be to have the JavaScript engines report user settings per site to a common service. This service would integrate the settings to determine average or median settings for a site that it would then convey to users visiting the site for the first time.

9. CONCLUSION

We investigated the claim that throttling the execution rate of client-side JavaScript can lead to lower power and energy on mobile devices without a significant decrease in user satisfaction with the the sites that employ the JavaScript code. We have generally found this to be the case, particularly for lower-interactivity sites. Our studies were done with an implementation of throttling that is based around JavaScript transformation in a web proxy. While this has the advantage of working with any client or site, its limitations may also understate the case. Nonetheless, it appears that even a simple throttle implemented in this way, with a default, unalterable setting, is able to reduce average energy over a range of sites by 4–10%, with lower interactivity leading to more savings. These results recommend the deployment of JavaScript throttling, and we provided a range of recommendations on how to do so.

10. REFERENCES

[1] Appel, A. *Compiling with continuations.* Cambridge University Press, Cambridge New York, 1992.

[2] Cheshire, S. Latency and the quest for interactivity. In *White paper commissioned by Volpe Welty Asset Management, LLC, for the Synchronous Person-to-Person Interactive Computing Environments Meeting* (1996).

[3] Ecma, S. Ecma-262 ecmascript language specification, 2009.

[4] Google. Android webview. `http://developer.android.com/reference/android/webkit/WebView.html`.

[5] Google. Closure compiler. `https://developers.google.com/closure/compiler/`.

[6] Joyent, I. Nodejs platform. `http://www.nodejs.org/`.

[7] Krohn, M., Kohler, E., and Kaashoek, M. F. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2007), ATC'07, USENIX Association, pp. 7:1–7:14.

[8] Lange, J. R., Miller, J. S., and Dinda, P. A. Emnet: Satisfying the individual user through empathic home networks. In *Proceedings of the 29th IEEE Conference on Computer Communications (INFOCOM)* (March 2010).

[9] Lin, B., and Dinda, P. User-driven scheduling of interactive virtual machines. In *Proceedings of the Fifth International Workshop on Grid Computing* (November 2004).

[10] Lin, B., and Dinda, P. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of ACM/IEEE SC (Supercomputing)* (November 2005).

[11] OKCupid. Tamejs javascript framework. `http://www.tamejs.org`.

[12] Raghavan, A., Luo, Y., Chandawalla, A., Papaefthymiou, M., Pipe, K., Wenisch, T., and Martin, M. Computational sprinting. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on* (2012), IEEE, pp. 1–12.

[13] Shye, A., Pan, Y., Scholbrock, B., Miller, J. S., Memik, G., Dinda, P. A., and Dick, R. P. Power to the people: Leveraging human physiological traits to control microprocessor frequency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2008), MICRO 41, IEEE Computer Society, pp. 188–199.

[14] Sousa, J., Balan, R., Poladian, V., Garlan, D., and Satyanarayanan, M. Giving users the steering wheel for guiding resource-adaptive systems. Tech. Rep. CMU-CS-05-198, School of Computer Science, Carnegie Mellon University, December 2005.

[15] Steele, G., and Sussman, G. Scheme: An interpreter for the extended lambda calculus. *Artificial Intelligence Lab Memo 349* (1975).

[16] Thiagarajan, N., Aggarwal, G., Nicoara, A., Boneh, D., and Singh, J. P. Who killed my battery?: analyzing mobile browser energy consumption. In *Proceedings of the 21st international conference on World Wide Web* (New York, NY, USA, 2012), WWW '12, ACM, pp. 41–50.

[17] Vorontsov, A. Android interactive cpu governor kernel patch. `http://lkml.org/lkml/2012/2/7/483`.