Measuring a HTML5 Hybrid Application's Native Bridge on iOS

Matti Paksula

M.Sc. Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, May 19, 2016

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Matti Paksula

Työn nimi — Arbetets titel — Title

Measuring a HTML5 Hybrid Application's Native Bridge on iOS

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
M.Sc. Thesis	May 19, 2016	79 pages + appendix pages

Tiivistelmä — Referat — Abstract

Mobile apps are intended to be created with mobile platforms development tools and programming languages. This native development requires specialized skills and can therefore be prohibitively expensive. HTML5 hybrid app development is a popular alternative for native mobile app development. This development model allows developers to use standard web technologies and the end result can be indistinguishable from a native app by its visual representation. This model enables faster iteration speed, allows any web developer to build apps and supports simultaneous cross-platform development. However, since the web technology is not as performant as native, these hybrid apps have often been criticized for being noticeably "laggy" by the app developer community and end users.

One of the key components that affects HTML5 hybrid apps performance is the native bridge used in the app. This component bridges the embedded HTML5 application to the device features that wouldn't otherwise be available (such as writing to a file on the device's file system). The native bridge is one of the few components that a developer can freely change. Selecting the best native bridge for the app's needs is important as an inefficient native bridge can cause human noticeable delay in the app. The performance of native bridges has been acknowledged in academia and industry, but very little researched systematically.

This thesis introduces a systematic method to evaluate native bridges performance. Along with this method, this thesis also describes a new open source tool implementing this method for benchmarking different native bridges. This tool hosts reference implementation for 32 native bridges. Example results from a test suite that tested all implemented native bridges with two embeddable web view engines (UIWebView and WKWebView) on four distinct iOS devices (two iPads, iPhone and iPod Touch) are evaluated. The results show that the majority of the known native bridge methods can cause human noticeable visual and auditory latency. It is also indicated that the performance is largely affected by app usage patterns. The slowest measured native bridge was over two times slower (from no delay to significant user interface delay) than the fastest one.

ACM Computing Classification System (CCS):
C4 [Performance of Systems]: Performance attributes;
H.5.2 [Information Interface and Presentation]: User Interfaces;
D.3.2 [Programming Languages]: Language Classifications—Concurrent, distributed, and parallel languages, JavaScript;
C.2.4 [Computer-communication Networks]: Distributed systems—Client/Server, Distributed Applications

Avainsanat — Nyckelord — Keywords
Mobile applications, hybrid apps, HTML5, cross-platform, native bridge, JavaScript

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Mobile application (app) development is one of the largest growing areas of software development. In June 2015, there were over 3.5 million apps available for download[1] and the number had accelerated growth by over 150,000 new apps per month.[2] These apps might not have long lifetimes as the app market is extremely fast-paced and competitive.[3]

Traditionally mobile apps are done as *native* apps. Native apps are built with programming languages *native* (intended) to the platform (Swift or Objective-C for iOS, Java or C++ for Android) with the tools supplied by the platforms vendor (XCode for iOS, Android Studio for Android). Currently, the growing app development need and the short-livedness of the apps make it hard and therefore expensive to find developers who have the required specific skill sets and availability.[4] For this and other reasons, HTML5 hybrid mobile app development is one of the most popular alternative methods for native app development [38]. Developers need familiarity only with the standard web technologies such as HTML, JavaScript and CSS. The development time is reduced since the code can be re-used for other mobile platforms. These factors can make HTML5 hybrid apps very cost effective.

Well crafted HTML5 hybrid apps can be seemingly indistinguishable from native apps for the end user [8]. However, especially when the complexity of the application increases, these apps have been criticized for being "laggy" and not up-to-par with native apps in their user experience by the leading app developers like Facebook [14].

There are multiple factors that contribute to indistinguishability of behavior. On the iOS platform, one factor is the *native bridge* that is the mechanism used to add native features (such as audio or GPS data) to the embedded HTML5 app. Developers are able to select from multiple different mechanisms to implement this bridging. Over the last years these native bridges have been implemented with different methods. As technology advances and new versions with new features of the mobile operating systems become available, these methods tend to change.

The differences and impact of native bridge mechanisms have systematically been researched very little, even though multiple academic and industry studies have noted this as a problem [9, 33, 39, 31, 13]. This brings us to the main research questions in this thesis:

- How can the performance of a native bridge be systematically measured?

[1] Statista: Number of Apps Available in Leading App Stores, Jun 2015

[2] Statista: Number of New Apps Submitted to the iTunes Store per Month, Jan 2016

[3] Recode: Mobile Apps have a Short Half Life; Use Falls Sharply After First Six Months, Sep 2015

[4] Gartner: Gartner Says Demand for Enterprise Mobile Apps Will Outstrip Available Development Capacity Five to One, Jun 2015

- What are the properties that affect the performance characteristics of a native bridge method?
- What kind of performance degradation can be noticed by the user?

Additional research questions are:

- Which techniques can be used as a native bridge?
- What is the typical usage pattern of the native bridge in a typical hybrid application?
- What are the best performing native bridges?

The existing work on this subject does not provide answers to these questions. To answer these questions a new performance measurement tool was developed as part of this thesis by the author. With the tool a developer is able to compare different native bridges and see how to implement a bridge in her own application. It also provides a suitable test bed for testing new, not yet discovered bridges.

The requirements for the tool are identified in the beginning of chapter 5 that describes the tool in detail. Through this tool, this thesis provides a thorough list of possible native bridge methods with implementation examples. Additionally, sample test results from four distinctively different iOS devices are also provided.

The next chapter 2 gives a background on mobile devices, apps in general and currently popular mobile platforms. Chapter 3 discusses different app development methods ending with HTML5 hybrid apps. In chapter 4 different factors affecting HTML5 hybrid app performance with explanation of the technologies used in these apps is given. The tool is described in detail along with the listing of supported native bridges in chapter 5. Then the results from a sample test run are explored and compared to other related work in chapter 6. Chapter 7 concludes with the learnings from this work.

2 Background

The introduction of *smartphones* in the early 2000's created a new software development area: 3rd party mobile applications. This generation of mobile applications were not commonly referred to the word "app". These smartphones had an operating system that allowed users to install various 3rd party applications, that extended the built-in capabilities and thus became "smarter" than the older mobile phones with fixed sets of functionalities.

Not only did these applications allow users to do more with their phones, but some of the applications have been very disruptive by changing the old value-creation models in the industry. Examples of such applications are the 3rd party messaging applications that bypass the mobile network operator SMS messages completely. The network operator's revenue streams from messaging and phone calls are diminishing because of applications and cellular data connectivity [37].

This chapter gives background for the general concepts related to mobile devices, applications and the mobile industry. First, we explore the transformation leading to modern touch devices. Then, we have a look at modern mobile applications, and lastly the current devices available in the market.

2.1 From Smart to Touch

Early smartphone operating systems included *WindowsMobile*, *Symbian* (mostly phones made by Nokia) and *BlackBerry* (phones made by Research in Motion). Other mobile device operating systems and devices such as the Palm OS for the Palm PDA (Personal Digital Assistant) existed, but those never became widespread before smartphones replaced the functions of these other mobile devices.[5]

Smartphones had a camera, multimedia capabilities, large displays and other features that 3rd party developers could use in their applications. By allowing the user to install applications to extend and customize the phone's abilities, the phone became very tailored for the user and thus very personal. Cellular data network connectivity enabled phones to become a major way to consume services and media content [37].

The transition from smartphones to touchscreen devices began with the release of the *iPhone* in 2007. The press release titled "Apple Reinvents the Phone with iPhone" describes the difference with "When users need to type, iPhone presents them with an elegant touch keyboard which is predictive to prevent and correct mistakes, making it much easier and more efficient to use than the small plastic keyboards on many smartphones".[6] Almost all of the device's frontal surface area was covered with a display that could accurately

[5] Gartner: Gartner Says Worldwide Smartphone Sales Grew 16 Per Cent in Second Quarter of 2008, Sep 2008

[6] Apple: Press release "Apple Reinvents the Phone with iPhone", Jul 2007

receive user touch input. For the applications this meant more freedom in customization of the user interface than in traditional smartphones.

Other companies quickly followed and started to make touchscreen phones as well. Google released its touchscreen capable operating system *Android* and partnered with former smartphone makers, such as HTC to produce phones running Android in 2008.[7] At the same time BlackBerry launched a touchscreen device.[8] Microsoft also re-aligned its older generation Windows-Mobile smartphone operating system to match the new competition renaming it *WindowsPhone*, along with other improvements related to touchscreens in 2009.[9]

The iPhone operating system (originally called "iPhone OS") was designed to be used in other devices than just in phones. In 2010 Apple used the same operating system in their touchscreen tablet computer, *iPad*, and renamed the operating system to *iOS*. Apple had worked with iPad prior to iPhone, but decided to release iPhone first as the phone market was more important for them [36].

After the launch of the iPad, Android phone makers followed again and started to make tablets. The first Android tablet to launch arrived 5 months after the introduction of the iPad. The device came with Android 2.2, an operating system that was still intended for phones instead of tablets. To address this, Android 3.0 was released in 2011 and it was "designed from the ground up for devices with larger screen sizes, particularly tablets" as described in its release notes.[10]

For application developers, the unification of operating system for both phones and tablets meant a bigger potential installation base for the same application that could adapt itself to different screen sizes.

2.2 Apps

The first iPhone version supported only limited web applications. Only with the second version of iPhone in 2008, could users actually download and install applications as we know them today. From their iPhones users could go to the Apple *App Store* and discover, pay and wirelessly download various applications in one single place.[11] For Android, Google built a similar concept called the *Android Marketplace* (later renamed *Google Play*).[12] In contrast to the restrictive Apple App Store, Android platform also allowed

[7] Google Mobile Blog: Google on Android, Sep 2008

[8] BlackBerry: Press release "BlackBerry Takes The World By Storm With Verizon Wireless and Vodafone, Oct 2008

[9] The Inquirer: Windows Mobile becomes Windows Phone, Oct 2008

[10] Android Developers Blog: Android 3.0 Platform Preview and Updated SDK Tools, Jan 2011

[11] Apple: Press release "iPhone 3G on Sale Tomorrow", Jul 2008

[12] Android Developers Blog: Android Market: a user-driven content distribution system, Aug 2008

other app stores to be used, such as *Amazon Appstore*.[13] Later Microsoft followed and announced its *Windows Marketplace for Mobile* (later renamed *Windows Phone Store*).[14]

While previous attempts to create an "app store" existed in the era of smartphones, they had very minimal success. Often mobile application developers needed to host their content, possibly implement payment mechanisms and promote their content in various channels. The new generation app stores made it simpler for the developer to distribute and consumer to install apps [36]. Reliability and quality increased as stores had feedback mechanisms (comments, ratings) and provided other services for the developers such as analytics.

The same or a slightly modified version of the application could run on many different devices from phones to tablets. For example, at the time of the iPad's launch there were over 140,000 applications already available for it (most of the apps being originally made for the iPhone), increasing the success of the product launch. These app stores also expanded to other digital content, where the same mechanisms could be used to purchase music, movies and TV series.[15]

At this phase, mobile applications became more commonly known as *apps* as in an advertising slogan for the iPhone "there's an app for that." describing one of the most important selling points for the iPhone. The word *app* was selected as the word of the year in 2010 by the American Dialect Society, implicating how mainstream these applications had become.[16]

In that year there were less than 250,000 apps available. The total number of apps has grown significantly since: four years later in the beginning of 2015 there were over 3 million apps (1.5M in iOS, 1.6M in Google Play and 0.3M in Windows Phone Store). The total download number of these apps from all platforms was around 140 billion in 2014.[17] While the total app numbers are almost equal in App Store and Google Play, the latter had over 60% more downloads in 2014. Although Google Play had more downloads, the eiOS App Store still created 70% of the yearly app revenue.[18] In 2014 Apple distributed over $10 billion in revenue to developers and app billings rose 50 percent over 2013.[19]

Some of the apps have been successful also by other means than in paid downloads or content. For example, a messaging app "WhatsApp" was acquired by Facebook for $19 billion in the 2014 and many apps have grown

[13]Amazon: Press release 'Amazon Appstore for Android Now Open in Nearly 200 Countries Worldwide", May 2013

[14]Microsoft: Press release 'Microsoft Reveals New Windows Phones With Marketplace and My Phone Services", Feb 2009

[15]Apple: Press release "Apple Launches iPad", Jan 2010

[16]American Dialect Society: All of the Words of the Year, 1990 to Present, 2010

[17]Statista: Statistics and facts about App Stores, Jan 2015

[18]App Annie: 2014 Retrospective

[19]Apple: Press release 'App Store Rings in 2015 with New Records", Jan 2015

to over half a billion monthly active users.[20]. The estimated total number of app developers was over half a million in 2014.[21]

The biggest single category in iOS app store was games (around 23%). The rest of the apps are divided in various categories such as business, education, utilities, entertainment and social.[22]. In addition to apps in the public app stores, apps can also be made available for internal company use. Internal apps are made to serve the workforce and different stakeholders of a company. Some functions of internal apps are internal communications, business intelligence, collaboration, sales and inventory management. Most of these internal apps are distributed through corporate intranets, email or private internal app stores for security reasons.[23]

2.3 Current Mobile Devices

The mobile device industry has been in a state of rapid change in recent years. Changes in mobile device sales are important for app developers as that defines the available installation base for their apps. For consumers the number of apps tells how likely the next popular apps are going be available for that platform.

At the end of the smartphones (without touch capabilities) era in 2008 when Android and iPhone had just entered the market, Nokia's market share was at 40.8% and BlackBerry followed with 19.5%.[24] A few years later in 2014, Nokia sold its mobile device business to Microsoft[25] and BlackBerry sold less than 1% of all smartphones in the last quarter of the same year. These changes can be seen in the figure 1.

The device, operating system and app store are used together. The term *mobile platform* to is used to describe this combination. At the time of writing there are three mobile platforms: iOS (Apple), Android (Google, Samsung, Lenovo, LG, Sony, HTC, others) and WindowsPhone (Microsoft, HTC, others). Touchscreen phone shipments are at their peak. In 2015 global smartphone shipments exceeded 1.4 billion units. Apple currently having around 15% of the market, published record revenues of $75.8 billion for the first fiscal quarter of 2016. The majority of that revenue ($58.6 billion) became from iOS devices. Only a small percentage ($6.7 billion) of the revenue had come from their traditional business, selling desktop and

[20] AppFigures: State of mobile advertising Q4 2014, 2015

[21] AppFigures: Mobile App Developers Stick With one Store, Jul 2014

[22] Statista: Most popular Apple App Store categories in March 2016, by share of available apps, Mar 2016

[23] Gartner: Gartner Says That by 2017, 25 Percent of Enterprises Will Have an Enterprise App Store, Feb 2013

[24] Gartner: Gartner Says Worldwide Smartphone Sales Reached Its Lowest Growth Rate With 3.7 Per Cent Increase in Fourth Quarter of 2008, Mar 2009

[25] Nokia: Press release "Nokia completes sale of substantially all of its Devices & Services business to Microsoft", Apr 2014

Period	Android	iOS	Windows Phone	BlackBerry OS	Others
2015Q2	82.8%	13.9%	2.6%	0.3%	0.4%
2014Q2	84.8%	11.6%	2.5%	0.5%	0.7%
2013Q2	79.8%	12.9%	3.4%	2.8%	1.2%
2012Q2	69.3%	16.6%	3.1%	4.9%	6.1%

Source: IDC, Aug 2015

Figure 1: Smartphone operating system market share from 2012 to 2015 (IDC)

laptop computers.[26] Android's market share of the mobile operating systems has been growing significantly. In 2015 around 80% of all smartphones sold had some version of Android. The market share of iOS in smartphones, on the other hand, has been rather steady, around 13% of the devices for the last four years. Microsoft's WindowsPhone holds a smaller shrinking share at around 3%.

At the same time, as smartphone shipments have been continuously rising, tablet sales have not been growing as fast. Smartphones have become bigger and more diversified in their pricing and are replacing tablets. Due to the success of iPad devices, the operating system market share in tablets is slightly different than in the smartphones. There, iOS has around 25% of the market.[27]

In the coming years, mobile devices are becoming the first "go-to device" for communications, content consumption and business. Gartner predicts that more than 50 percent of users will go to a tablet or smartphone first for all online activities, while the PC will be increasingly reserved for complex tasks.[28]

[26] Apple: Press release "Apple Reports Record First Quarter Results", Jan 2016

[27] Statista: Global market share held by tablet vendors from 2nd quarter 2011 to 4th quarter 2015, 2016

[28] Gartner: Gartner Says By 2018, More Than 50 Percent of Users Will Use a Tablet or Smartphone First for All Online Activities, Dec 2014

3 App Development

Developers use various methods and tools to develop apps. In this chapter we will explore the primary categories to develop apps and compare them to each other. First we explore the traditional ways of creating applications and at the end of this chapter we will focus on HTML5 hybrid app development.

3.1 Web Apps

When the iPhone was introduced in 2007, developers were not able to create applications that could be installed to the device. Instead Apple had enhanced the built-in web browser (Safari) to provide an "app-like" mode for mobile optimized web sites to work as "apps". The developers could register their web apps to a directory hosted by Apple from where users could "install" the apps by creating a bookmark icon to the device's home screen next to the built-in apps.[29] Web apps are still used today as a light-weight version of an app or as a more app-like mobile web site.

Web apps, much like mobile web sites, are developed with the same standard web technologies such as HTML, CSS and JavaScript. In iOS the Safari browser extends these standards allowing some of the device capabilities to be used. Hardware capabilities such as the accelerometer (measuring using the device's movement data) are available. The camera, contacts and other more sensitive data are not available. Limited interaction with other apps is possible. For example, a phone call can be invoked, a map locations can be shown in the built-in maps app or the email composition view can be opened. The multitouch capabilities are also exposed to web apps. A web app knows if the user uses one or more fingers to interact with the web page. Apple also enhanced CSS transitions to make animations smoother to match with the visual quality of their own built-in apps. A web app can also store a limited amount of data in the device to provide a degraded version of the app when network connectivity is not available.[30]

Many of the extensions introduced with the mobile browsers by Apple, Google, and Microsoft have been accepted to the web standards maintained by the World Wide Web Consortium (W3C).[31] This standardization ensures that web apps behave similarly all mobile platforms.

In addition to the access to the device capabilities, the app also needs to mimic the look and feel of an installable app. Various open source libraries have been released to make it easier to create consistent user interfaces that look like native apps [39].

[29] Apple: Press release 'iPhone to Support Third-Party Web 2.0 Applications", Jun 2007
[30] Apple: Safari Developer Library - Getting Started with iOS Web Apps
[31] W3C: Technical Rport: The Screen Orientation API, Oct 2014

3.2 Native Apps

One year after the introduction of the iPhone, Apple released the iPhone SDK (Software Development Kit) for developers. The iPhone SDK is a subset of the Apple's internal private SDK that is used to develop the apps that come pre-installed with the iPhone. Originally, the SDK was a of collection *Objective-C* libraries, the *native* language of the iOS platform. An app written in Objective-C was compiled as a native app that could be submitted to the iOS App Store for distribution. Later in 2014, Apple released a new programming language called *Swift* as an alternative programming language for creating native apps.[32] Today both of these languages can be used to produce code that runs natively on the iOS operating system.

Android apps run in a virtual machine called *Dalvik*. Dalvik is a clean room rewrite from Java VM to be better suited for mobile apps [30]. Applications for Dalvik are written with *Android SDK for Java* or *Android NDK for C/C++* (Native Development Kit). Applications written in both languages are fully native to the Android platform [10]. In WindowsPhone platform the native apps are written with *Windows Phone SDK* with *.NET* or C++.[33]

A mobile platform SDK provides a complete set of tools required for development. The tools typically include an *IDE* (Integrated Development Environment), compilers, libraries, instrumentation analysis and testing tools. Apps can be compiled and run in a real device or in a device simulator for faster development. As the application's code utilizes the SDK's libraries directly, the developer often needs to update the app's source code whenever a new version of the SDK is released to be compatible with the new operating system version.[34]

Native app development guarantees the fastest possible performance for the app. The performance is especially important in games, but also in other areas such as data processing. Native performance has also drawbacks: having direct or near-direct access to the device's hardware makes harder to debug [10].

3.3 Interpreted Apps

An interpreted app is packaged inside a *native shell* application that hosts a language interpreter. This method allows developers to write the app with another language than the platform's native language. Typically these languages are dynamic scripting languages, such as JavaScript (frameworks include Appcelerator Titanium, React Native, Telerik Platform) and Ruby

[32] Apple: Press release "Apple Releases iOS 8 SDK With Over 4,000 New APIs", Jun 2014

[33] Microsoft: The Visual Studio Blog: Introducing Windows Phone SDK 8.0, Oct 2012

[34] Apple: iOS SDK Release Notes for iOS 8.2, Mar 2015

(Rubymotion) that allow more flexibility as the code can be changed without recompilation of the whole application binary. As the resulting app is fully native, developers can use most of the same development tools (debuggers, device simulators) as with fully native development. The abstraction layer might make it more complicated to debug and inspect applications. The finished app is submitted to the specific app store as any native app would [39].

Typically these apps implement the user interface with the same native user interface components (buttons, text fields, image views) that are controlled with the interpreted language. Developers don't need to learn all the platform specific details as the native shell works as a bridge between the developer's code and the native part of the app. The execution speed of interpreted apps can be slower than native apps and new user interface features may only be available when the native shell is updated to support those [39].

3.4 Cross-platform Apps

Usually a developer wants to support multiple platforms to maximise app growth, revenues and reach to customers. Making apps for multiple platforms is expensive and time consuming. It has been a trend that even the top downloaded apps that are available for iOS and Android, are missing for other platforms due to their smaller user base [7]. In addition to development costs, it also takes more effort from developers to learn all different target platforms and follow their changes [39].

As interpreted app frameworks have abstracted the target platform from the app source code, it's possible for them to support simultaneous development for multiple platforms. The development environment packages the developer's source code as different native apps that the developer can submit to different app stores. The developer can re-use some or all of the source code since the shell application does the fitting for the target platform. Examples of cross-platform interpreted app development environments are the aforementioned Appcelerator Titanium and also Xamarin that allow developers to develop iOS, Android and WindowsPhone apps with C#.

As mobile platforms differ from each other in user interface and feature sets, the amount of actual code re-usability varies depending on the app. In order to make the app behave consistently with the target platform (share the same look and feel with other apps), it's required to write the user interface layer separately for each platform. The developers are also dependent on the maintenance and updates for the development environment [39].

Model-driven software development (MDSD) can also be used to develop cross-platform (and single platform) mobile apps. This *generated app* model allow developers to use a modelling language to describe the app and its functionalities. The resulting app is generated either as a native app or

an interpreted app [39]. This approach to develop software is not specific to mobile apps and therefore not listed as its own category. Applause[35] (generates native apps) and iPhonical[36] (generates interpreted Appcelerator apps) are examples of this approach.

3.5 Hybrid Apps

Native SDKs include various user interface components, such as lists and buttons. All SDKs include a native view component, *web view*, that is intended for presenting web content inside the app without opening a full web browser. This view control is *chromeless*: it implements no browser controls such as the address bar or navigation controls (go back, forward). A developer can use the component to load and display any web page and embed it seamlessly to the app's user interface.

A *HTML5 hybrid app* combines the native and web technologies so that the app and its user interface are primarily implemented with an embedded web view. It mimics a native app's user interface with standard web technologies: a well made HTML5 hybrid app is not easily distinguishable by its looks from an app done with fully native UI controls. Figure 2 has a comparison of views from both a HTML5 hybrid app and a similar native app that are seemingly indistinguishable from each other.

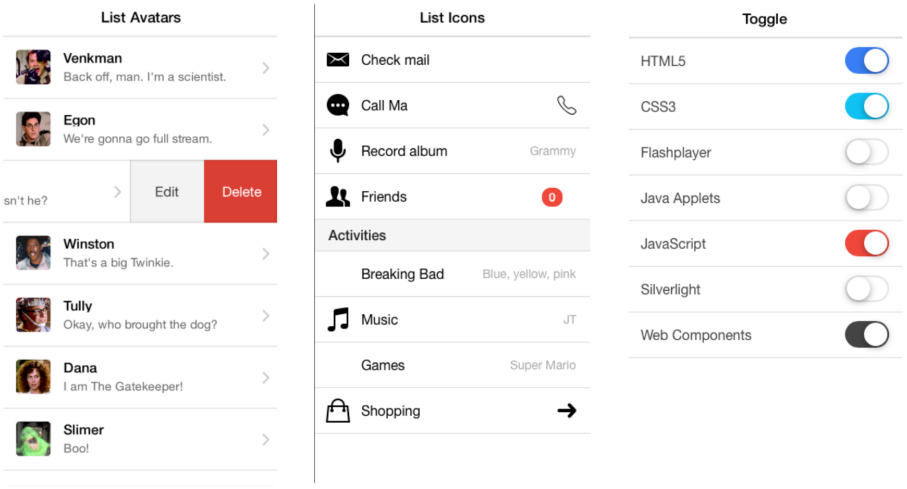
The previously introduced interpreted apps can also be used to produce HTML5 hybrid apps. This kind of combination of different paradigms is supported for example in Xamarin Razor: the web content is generated from a mix of HTML and C# that is then displayed inside a shell app running the interpreter. The term "hybrid app" is sometimes used for interpreted apps since for example "Ruby and native" is a "hybrid approach". In this thesis the term is used to describe HTML5 hybrid apps.

A hybrid app can be implemented in two distinct styles: as a *hybrid web app* or as a *hybrid mixed app* [39].

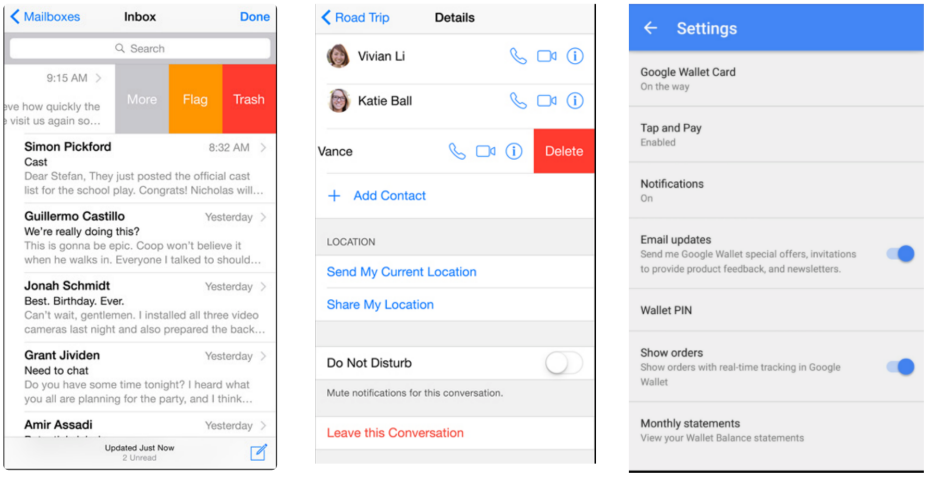
In a hybrid web app, the user interface consists of one full screen web view that loads the HTML, JavaScript and CSS files from the files stored in a shell app. Basically, hybrid web apps are like the mobile web apps, except that instead of a browser, they reside in a native shell app, like interpreted apps. The native shell app contains the required files, enabling the web view to load the user interface instantly and also work without a network connection. Unlike web apps that are limited to the browser's security model and can not access all device capabilities, the web view can have lesser security constraints and it can be given access to the same capabilities that a native app has [39]. While Apple allows hybrid apps in the App Store, it requires that apps have their own content and not just load a remote web

[35]GitHub: Applause https://github.com/applause/applause

[36]Google Code: iPhonical https://code.google.com/archive/p/iphonical/



Case 1: Hybrid apps with a full screen web view and HTML5



Case 2: Native apps with native UI controls (iOS, iOS, Android)

Figure 2: Similar hybrid app vs native app views

page in a native shell.[37]

A hybrid mixed app combines one or more web views with a native user interface and navigation controls. The main idea in a hybrid mixed approach is that an app's user interface can be fully native, but contain one web view to display some information, or, most of the app can be implemented with web views and contain a few native views or some view controls like buttons. The hybrid mixed approach can be described as a "web-native continuum" that starts from mobile web apps and ends in native apps as illustrated in

[37] Apple: App Store Review Guidelines

figure 3 [17].

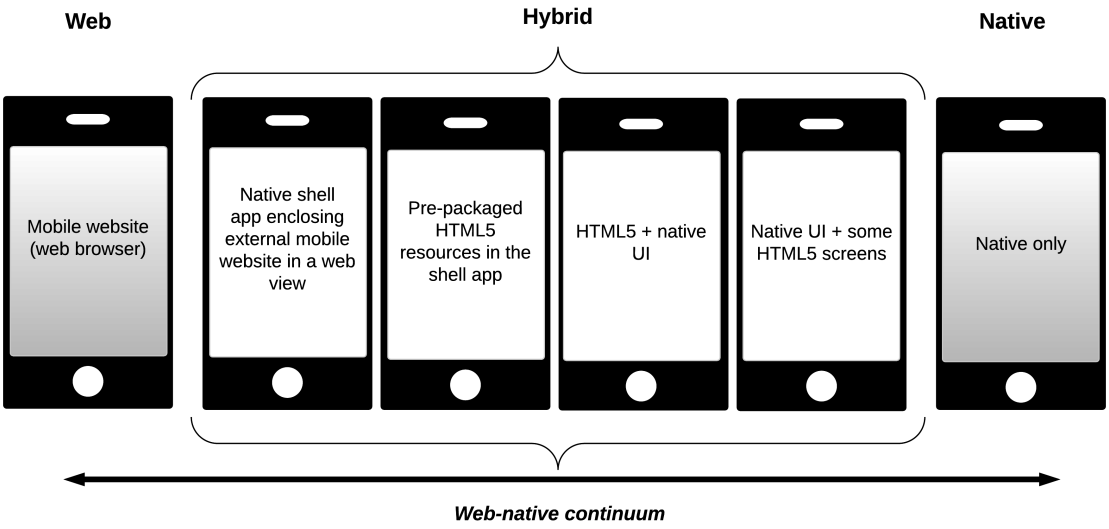


Figure 3: The "web-native continuum" - spectrum of hybrid mixed app development approaches

This *wrapping* technique of enclosing a web view within a native shell app was popularized by an open source project called *PhoneGap* that has been developed from 2008 [8]. The *gap* suffix in the name refers to "bridging the gap" between browser and native SDK. The company behind the project (Nitobi) was acquired by Adobe in 2011. In the acquisition, the project source code was given to Apache Software Foundation and renamed Apache Cordova.[38] The Cordova project provides the source code and tooling for building apps for iOS, Android, WindowsPhone and others. The ability to develop for many platforms makes Cordova a *hybrid cross-platform framework*.

After PhoneGap there have been a number of hybrid web and hybrid mixed app frameworks as both closed and open source (for example IBM WorkLight, AppGyver Steroids, Trigger.io Forge, Ionic, Monaca). Some of the frameworks allow developers to use multiple web views and/or native user interface controls in addition to a single web view. An example of this hybrid mixed approach is given in figure 4 where native navigation controls are placed at the top and bottom of the screen and the app's actual user interface is implemented with a web view.

Internally, all of these hybrid frameworks use the same basic principle: the app has embedded a web view that sends and receives messages with the native shell app to execute code that is not possible to execute inside the

[38] Adobe: Press release "Adobe Announces Agreement to Acquire Nitobi, Creator of PhoneGap", Oct 2011

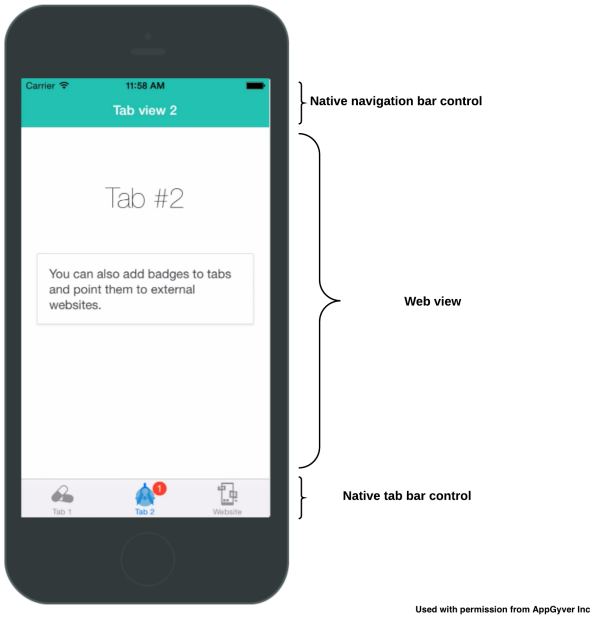


Figure 4: An example of a hybrid mixed app with native UI controls on top and bottom.

browser engine. The native code implementation of the actual interaction with the SDK can be packaged as a library and shared and thus re-used among different hybrid apps. For example, the Cordova project provides a *native plugin* architecture that makes it simple for developers to add native capabilities to their hybrid apps [8].

Hybrid apps have seen significant traction in terms of developer adoption. In 2013 Gartner estimated that by 2016 50% of all apps deployed (especially in the context of business apps) will use some hybrid app solution. This is due the trend of *BYOD* (Bring Your Own Device) where employees are given freedom to select their preferred mobile device, which is increasing device fragmentation.[39] Also Canalys says that HTML5 this hybrid approach is attractive for cross-platform and developer skill set reasons (requiring mostly only web skills). Hybrid cross platform is also expected to make more apps available for WindowsPhone and other non-mainstream platforms [7]. VisionMobile's 2014 developer survey suggests that HTML5 is the most widely used technology at 40% of the developers (native languages such as Java and Objective-C being at around 20%). Almost half of the iOS and Android developers use some other language than the native languages to develop their apps [38]. The actual number of hybrid apps in 2016 is currently unknown. As hybrid apps are packaged like real native apps, therefore there

[39]Gartner: Gartner Says by 2016, More Than 50 Percent of Mobile Apps Deployed Will be Hybrid, Feb 2013

are no statistics available. It would be a significant operation to crawl through the app stores, download all applications and reverse engineer the application binaries to see if they contain HTML5. The number of internal apps that are distributed outside of the public app stores would still be not available.

Hybrid apps with web technologies have been a powerful way for fast iteration of the first versions of the app. Later "more native" versions still tend to contain some HTML parts as *mixed*. A web view component is commonly used in almost all apps that are considered fully native [27]. One example of such app is the Facebook app (one of the most downloaded and used apps in the world) that started mostly hybrid and then moved to more native over time [14].

None of the aforementioned ways to develop an app can be considered the best solution to develop apps. Development resources, developer skill set and the requirements must be considered when choosing a technology [39].

4 Hybrid App Indistinguishability

Mobile browser vendors have introduced many enhancements to make web technologies more performant. For example hardware acceleration in CSS transitions has been available since the launch of the iPhone. With hybrid apps the development time is significantly boosted as code can be re-used and written without learning the platform specifics or native languages. Over time web technologies and thus hybrid apps are likely to become indistinguishable from native apps [8].

Despite the enhancements implemented in mobile browsers, hybrid apps have been critized for being noticeably *lagging* in user interface performance [14, 33]. The app's user interface being the most visible part of the app has a vital impact on the whole user experience. The user interface issues are not specific only to the hybrid apps. A study by Liu et al. that characterized mobile app bugs identified that most performance bugs are related to user interface lag. The lag reduces smoothness of visual animations or completely prevents users from performing tasks with the app [23].

We will now explore the different factors affecting hybrid app performance and thus the indistinguishability from a fully native app. First what can be perceived by the user is defined. Then individual technologies that affect hybrid apps' performance are discussed. Lastly native bridge latency is identified as one of the key components and then a model for how to measure native bridges is defined.

4.1 User Perceived Performance

Performance measuring in application development can rely on easily quantifiable low level metrics such as *instructions-per-second* of the CPU. The performance that the user perceives might not be affected by these individual hardware metrics. For example, the app's user interface updates are more visible to the user than the execution speed of the CPU. Noticeable delays in the user interface reduce satisfaction, making the application feel less performant even if the instructions-per-second type of metrics show good performance [24].

Users are able to adapt to lagging user interface. Obviously, with the latency the number of errors (for example accidental re-selections of elements) increases and user satisfaction decreases. The exact latencies of what can be noticed, dealt with and what is unacceptable have been broadly researched in various domains. In mobile games Delwadia et al. experimented by adding different latencies to the game play. Latencies tested were clearly noticeable by the participants even if they were just tens of milliseconds. The conclusion was that for mobile games the latency needs to be less than 75ms in order to still be playable. Higher latencies than this significantly worsened the game play [12].

A similar study was conducted by Anderson et al. where participants performed various tasks such as web browsing. As the participants performed the tasks, some consistent latency was added to the interactions. The participants noticed the latency, but were able to adjust to it. Added latency value of 580ms was found to be the last acceptable latency for a mobile app [3].

In 2013 Shengqian et al. analyzed Android app store review comments and the actual app performance. The conclusion was that if the app takes more than 200ms to respond to a user action, it is commonly described as "sluggish". Apps like these tend to be removed from the device and given a bad review in the app store [41].

As described above, the users start to notice delays of tens of milliseconds and are able to adapt to them unless delay reaches several hundred milliseconds. For a mobile app user interface to be described as "smooth" and responsive, the latency needs to be not noticeable by the users. The human eye and brain are able to process static flashing images that are shown for just 13ms each [32], but it is not necessarily the same as what movement can be noticed on screen. This just-noticeable minimum latency that can be perceived by the user has been researched in various domains.

One of the first studies of software latencies was done in 1968 by Miller titled "Response time in man-computer conversational transactions". The paper argues that a latency that is below 0.1s is not noticeable by the user [26]. This Miller's latency definition is generally referenced in various recent publications. Considering that the paper describes "man-computer conversations" that took seconds, this definition should be safely ignored nearly 50 years later.

In a more recent research in 1999 by Regan et al. performed an experiment in virtual reality 3D visualization and movement detection. The users wore a head tracking device that controlled the movement of the graphics on the screen. The latency between the head tracking and screen updates was altered during the experiment. In this experiment the participants were able to detect as low as 7.5ms latencies, although the majority of the participants noticed latency only when it was bigger than 15ms [35].

In the mobile apps domain in 2013 Jota et al. tested the noticeable latency in *taps* (users select items by touching the screen). Participants were tapping static areas on the screen and there was a latency added to a user interface update that resulted from the tap. When using only tap input, most of the participants could not notice latencies smaller than 20ms [19]. Figure 5 shows similar experiment in action.

Similar test for *dragging* (users use a finger to move items) was done by Ng et al. in 2013. In the experiment, participants dragged objects with different added latencies. Figure 6 shows similar test setting in action. When dragging objects the user interface is constantly updated. Therefore the participants noticed significantly lower latencies (from 2.38ms to 11.36ms)

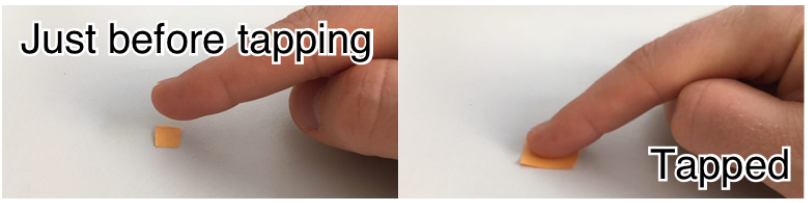


Figure 5: *Tapping* items on touchscreen

than with the tap-events (where latencies could be as high as 20ms) [29]. Deber et al. performed a similar dragging experiment in 2015 that also says that the detectable latency for dragging is around 11ms [11].

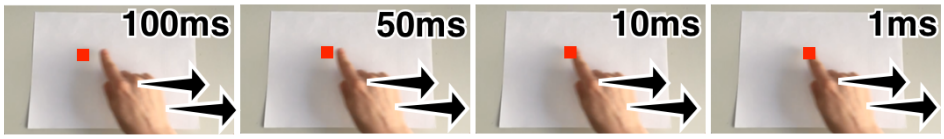


Figure 6: *Dragging* an object from left to right on a touchscreen with the same velocity with different added latencies (from 100ms to 1ms)

In 2014 Ng et al. continued with testing latency perception with stylus interaction. They performed three different tasks with the participants: dragging a larger box (20mm x 20mm), dragging a smaller box (6.25mm x 6.25mm) and scribbling a thin line on the screen. Similar experiments can be seen in figure 7. The median latency noticed was 6ms with the large box dragging and 2ms with the smaller box. The latency in the smaller box was easier to notice, since the tip of the stylus moved proportionally closer to the edge as it was dragged. Scribbling a line had a larger noticeable latency in around 40ms [28].

The research described above has been about visually noticeable latencies. Mobile apps can also give audio-haptic feedback (for example playing a click sound when tapping a button). In a study by Adelstein et al. in 2003 that tested the delay of a visual event versus the expected sound, it was found that the participants start to notice a delay between a visual event and auditory sensation when the asynchrony is around 24ms [1]. Agnew et al. experimented with hearing aid digital signal processing delays in 2000. Results indicated that a delay of 3-5ms was noticeable for most of the participants. The participants were more sensitive to the delay because the sound was their own voice that is easily noticeable [18].

With a touchscreen, users can perform various *gestures* on the screen with their fingers. Latencies also affect the algorithms that are used to detect these gestures. Some of the gestures follow the same movement path, but with a different speed. For example a *drag* is a slower, more consistent movement

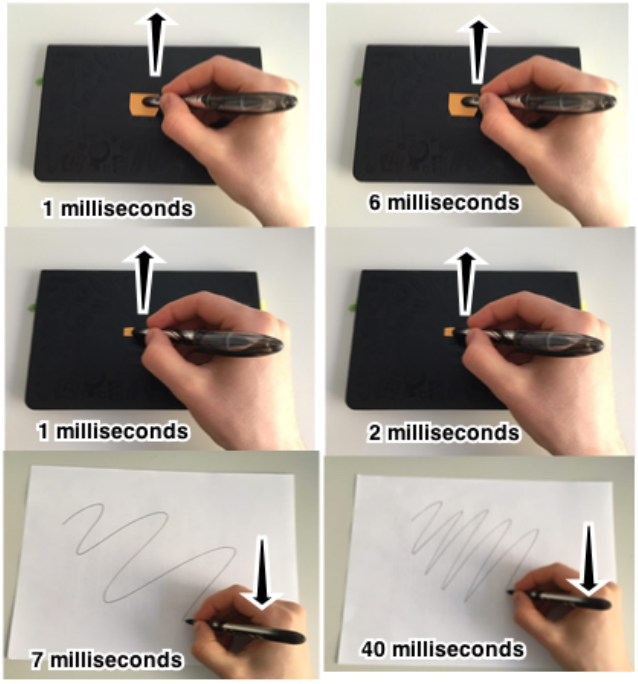


Figure 7: Illustration of how to perform just-noticeable latency testing with a stylus (from 2ms to 40ms)

on the screen than a *flick* that is used for quick browsing of content. The gesture detection engine of the platform's SDK makes decisions about the gestures for the developer. The decision criteria for gesture detection of the SDK's are unknown and therefore Quinn et al. performed an experiment in 2013 where a robotic arm executed consistent gestures on touchscreen devices. The result was that in both iOS and Android, the gesture detection is dependant on the number of samples received. If the sample count is low, the gesture is not recognized correctly [34]. If the user interface is "choppy" or "laggy" the gestures may not be detected correctly.

The latencies noticed depend on the interaction type (drag, tap, stylus or audio-haptic) and on the size of the target interaction element. Simple selecting (tapping with an optional auditory response) can have a higher latency (around 20ms) than when the screen is constantly being redrawn in response to the user events (dragging delay becomes noticeable in around 2-3ms depending on the items dragged). A latency does not completely prevent use. Users are able to adapt to latencies as long as the latencies do not become overly large. Satisfaction in the performance, however, dropped instantly when the latency varies and movement becomes "choppy" [3].

When optimizing for the user perceived performance, other factors can also be taken into consideration such as impact on power consumption and

battery life. Computation can be used to perform other tasks as long as the user interface receives enough computational cycles to have "good enough" latency [2]. For example, in video conferencing, adding a artificial non-noticeable delay frees resources to improve the overall quality of the video call [40].

4.2 JavaScript Execution Speed

Historically native code has been considered the most performant option for app development. However the execution speed comes with a significant increase in development time and requires a larger and more platform specific skill set than with hybrid apps done with JavaScript.

JavaScript performs slower, but improving its performance has been researched a lot during the last years and it is likely to get faster for example with new multicore processors and computational optimizations [25]. Currently JavaScript can be performant enough for most computational challenges and it is constantly being improved [20]. As JavaScript performance is further optimized, more and more of the apps that are dependant on computational speed can be implemented with some sort of hybrid approach. Only very few apps have such executional performance requirements that they can not be implemented with JavaScript [8].

A hybrid app's user interface is controlled with JavaScript in an event-driven style. User interface elements register JavaScript callbacks (functions) that are called when the user interacts with the element. For example when a user taps a button, the button's corresponding `onClick` callback gets called and registered actions are executed. JavaScript is executed in a single-threaded *event loop*. When the callback is executing, the event loop is not running. After the callback has finished, the control is returned to the event loop. In order to keep the application responsive, the callbacks need to execute quickly or split the execution in smaller pieces so that the event loop is not completely blocked [21]. In addition to the programmer defined callbacks, the browser's event loop performs all the layout changes required to update the screen [5].

On both iOS and Android platforms the web view component is run in the UI thread (or main thread) [4, 16]. Therefore, the implications of blocking the JavaScript event loop are also propagated to the UI thread blocking not just the web view, but also the whole user interface including any additional user interface objects.

4.3 Animation in Hybrid Apps

In hybrid apps the user interface can be updated in three ways: by manipulating the HTML element positions with JavaScript directly, by drawing pixels on a `canvas` element with JavaScript, or by using CSS transformations

on the HTML elements. As every method has its own limitations or uses, most hybrid apps require a combination of all of these techniques. CSS transformations have been native enhanced since the launch of the iPhone in 2007.[40] CSS transitions are declarative: a developer describes a transformation and the rendering engine performs the transition independently. While the transitions are performant and comparable to native execution speed, they are not enough to implement the whole user interface [6].

The user interface is constantly redrawn on the screen as a series of *frames*. *Frame rate* is used to describe the rate of complete updates done. It is measured with an *FPS* (frames per second) number that indicates how many times the display image was redrawn during one second. In iOS the screen refresh rate is limited to 60 FPS: the screen is never redrawn more than 60 times per second [4]. This means that the screen updates every 16.667ms (1000ms / 60). To get a smooth linear movement the object needs to move the same small amount in every update interval (every 16.67ms) [5]. Skipping one screen update (frame) is noticeable for the user as found in previous research where users notice latencies that are clearly below 20ms [29].

The JavaScript function that moves the object might not get called in time when other functions are blocking the event loop. The object doesn't need to move in every frame as long as it moves consistently and does not skip those scheduled movements. This smooth movement is illustrated in figure 8 where the frame rate is first kept consistent in case 1 and then altered in case 2. Animation starts to look "choppy" when the refresh rate varies.

Animation with JavaScript is traditionally implemented with `setTimeout` or `setInterval` callbacks. The problem with these callbacks is that the JavaScript execution context doesn't know the screen refresh cycle (if the screen is currently being redrawn or has just finished drawing). To address this browsers have implemented an animation timing control mechanism to guarantee smooth animation callback called `requestAnimationFrame`. The browser controls the calls of this callback so that it optimizes which animations are visible on the screen and ensures that those callbacks get called more often than the ones that are not visible on the screen. This approach also conserves CPU power as all the animations are not unnecessarily drawn.[41] It is worth emphasising that if the JavaScript event loop is blocked from execution, not even `requestAnimationFrame` callbacks get called and thus no animation occurs.

A recent draft proposal by W3C called "Cooperative Scheduling of Background Tasks" introduces a similar callback to `requestAnimationFrame` called `requestIdleCallback`. This callback can be used by the developer to inform the browser that the work performed in this callback is not time

[40] Apple: Press release "iPhone to Support Third-Party Web 2.0 Applications", Jun 2007
[41] W3C: Technical "Report Timing control for script-based animations"

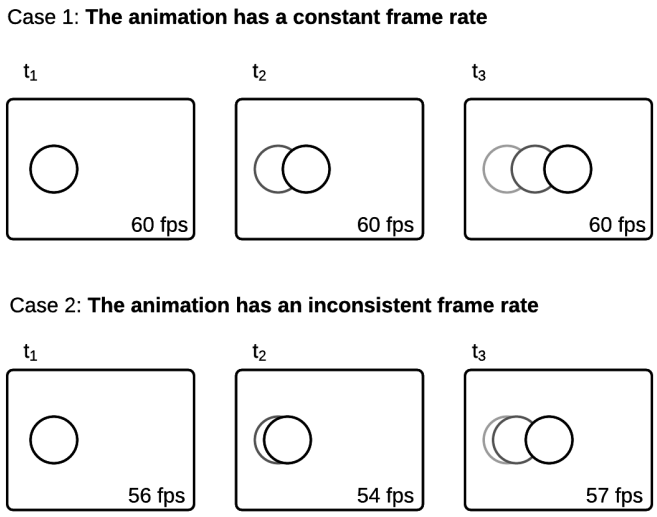


Figure 8: Animation performs consistently or inconsistently causing notable difference in the playback smoothness.

critical and can be performed when the browser would otherwise be idle. The proposal suggests that these idle callbacks will be capped to 50ms execution so that the browser can start responding to events originated from the user in-time. The 50ms is more of a guideline and derived from Miller's 1968 study of Man-computer conversational transactions where the "0.1s" value was identified to appear instant for the user [26]. This and other new features will free up processing time from the event loop in future browser versions, but won't completely eliminate the blocking problem [42].

This blocking problem has recently been addressed by different tools. In 2015 the Safari browser received a rendering frame inspector tool that web and hybrid app developers can utilize to inspect the time spent in each event loop cycle. The tool is is shown in Figure 9 where the x-axis shows the individual frames and the y-axis shows how long a frame took to render and how the time was divided. In the figure all frames render in about 4ms. The tool has support for filters that can highlight the frames that are spending more than the allotted 16ms budget [5].

[42]W3C Editor's Draft "Cooperative Scheduling of Background Tasks" https://w3c.github.io/requestidlecallback/

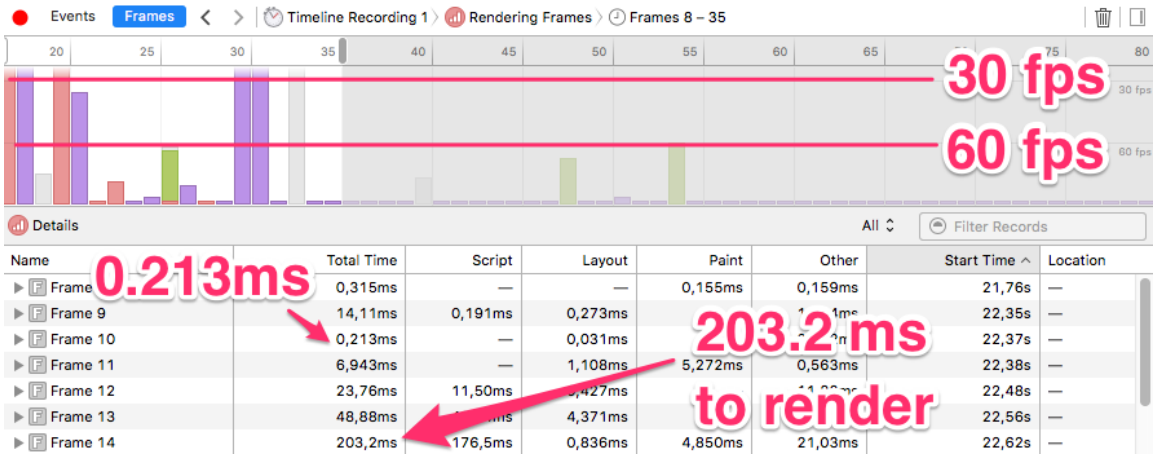


Figure 9: WebKit Rendering Frames timeline showing the 16.67ms budget usage for each frame. Two frames are annotated: one that renders nicely (in 0.2ms) and another that blocks the rendering with total time of 203ms.

4.4 Native Bridge

The web view component where the user interface is presented in a hybrid app is a *sandboxed* execution environment with limited access to the device's capabilities. To allow the JavaScript code running inside the sandbox access to device capabilities like a native app would, a *native bridge* is needed. A native bridge is a mechanism that can pass messages from a web view component to the native code and the other way around. The message is a string, typically in *JSON* (JavaScript Object Notation) format for easy parsing and due to the native support for JSON in the web view.

The message specifies a requested action (e.g. vibrate the device) with parameters (e.g. for 250ms). Figure 10 illustrates communication that is initiated from the web view to the native code. In this example the native shell app receives two different messages that are further passed to the SDK that actually performs the desired actions.

In a hybrid app there are typically two different native bridge mechanisms due to technical limitations: the mechanism used to transmit messages from the web view to native code may not be available for the other direction of communication. The same applies for messages initiated from the native code. Figure 11 illustrates this other direction. In this example the web view requested location updates from the native code. When the native code receives a GPS location update from the SDK and passes it to the web view, there is a message handler for the event. The native keeps sending the updates in requested intervals until it receives a message from web view to stop.

The PhoneGap/Cordova project pioneered native bridging for hybrid

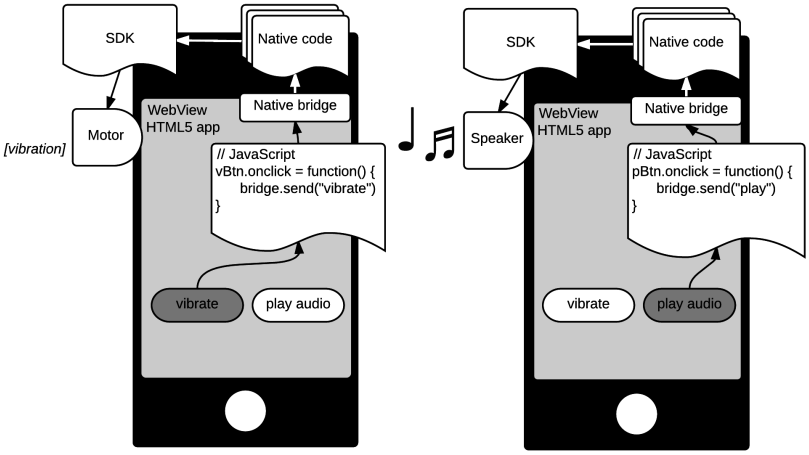


Figure 10: Illustration of a hybrid app native bridge communication initiated from the web view. The app has two buttons that perform different `onClick` callbacks that communicate with the native code through a native bridge.

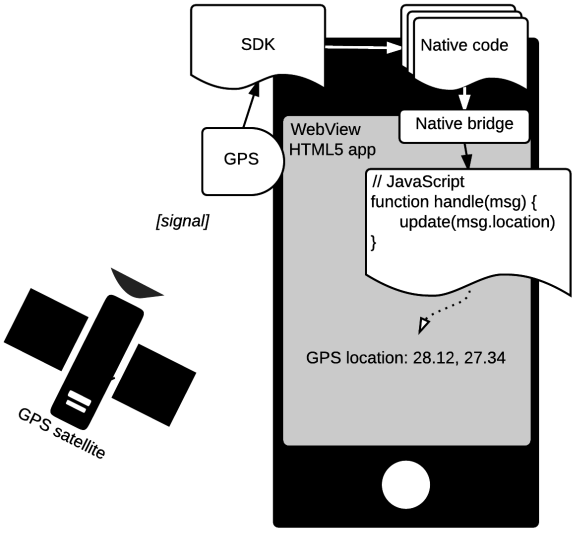


Figure 11: Illustration of a hybrid app native bridge communication flow initiated from the native code. The app has a message handler that receives the location update from the native code.

apps in 2008 by using two mechanisms in the iOS SDK. To receive messages from the web view, a native method `shouldStartLoadWithRequest`

was used that got called whenever the web view started to load a URL. The method determined that if the URL started with `gap:` instead of `http:`, it was a native bridge message. For example, JavaScript started to load a URL with `gap:getloc` to get the last known GPS location from the native side. The location was sent back to the web view by evaluating a JavaScript callback directly in to the web view with the native method `stringByEvaluatingJavaScriptFromString`.[43] Neither of these SDK methods were intended for creating hybrid apps, making this mechanism essentially a "hack" [8]. Today PhoneGap's native bridge implementations can be changed to support different mechanisms.

4.5 Frequency and Size of Native Bridge Messages

In hybrid applications, the number of messages sent and their sizes vary greatly. A hybrid app can use multiple native bridge calls or it can be implemented without a single one. For example, an application where a user can view and edit data from an HTTP API doesn't need to send any native bridge messages, since HTTP API requests can be done entirely from the web view. Even a application where a user selects a photo from the device's photo library and uploads it to a web server can be done entirely without native messages. On the other hand, even the simplest application which has just one single button that plays a sound needs to send a message every single time the button is pressed. Obviously if the button is tapped rapidly then the frequency of messages is high. A application that constantly receives GPS location updates has a lower frequency of updates (location is received every 3 seconds, compared to rapid tapping of the button).

The payload (size) of the messages is usually very small. For example, a typical message to play a sound is just a reference to the sound file "beep.wav" that the native side will read and play, or a message that makes the device vibrate for 3 seconds can just contain word 'vibrate' and 3000. Even retrieving all the contacts from a phone book tends to be very small payload (assuming that the device has, for example, 200 contacts).

When developing a hybrid application, developers often have limited knowledge of native programming and rely on different native plugins that do only one re-usable feature (such as compressing an image and sending it back to the web view). This also keeps the application's logic in the web view (when the web server endpoint needs to be changed, it will only be updated to the JavaScript running in the web view). This creates larger messages, such as, for example, raw photo binary data. Let's take the case of a photo that needs to be compressed before sending to the server. It is more performant to do image manipulation operations in the native code. In this case, after the user has selected the photo, the JavaScript will pass it on

[43] PhoneGap: Github commit to iOS that adds vibration support
https://github.com/infil00p/phonegap-iphone/commit/d5f22109fa6b25c6529e045b62693e4ec258db7c

to the native side with a bridge for compression. This creates one message that has a large payload (megabytes of photo data). Once the image has been compressed, it can be sent directly to the web server from the native code or transferred back to the web view, as fewer megabytes of photo data because of the compression.

In table 1 below there is a list of examples of native features. For each feature, the table shows the originator for the native bridge message as sender, the size of the message, proposed frequency of how often these messages occur in an app and what are example contents of the payload.

Table 1: Examples of native bridge usage

Feature	**Sender**	**Size**	**Frequency**	**Example payload contents**
Play sound	web view	small	When tapped	Filename
Vibrate device	web view	small	When tapped	Time to vibrate
Get phonebook contact names	web view	small	Once	List of names
Create a contact to phonebook	web view	small	A few times	One name and its details
Battery status enquiry	web view	small	Seldom	Current battery percentage
Get a picture from camera	web view	large	Seldom	3MB of photo data
Record sound for 3 seconds	web view	large	Seldom	100KB sound data
Request GPS updates	web view	small	Once	Integer
GPS update	native	small	Every n seconds	Coordinates
Request motion updates	web view	small	Once	Integer
Device motion update	native	small	Every n ms	X,Y,Z coordinates
Push notification message	native	small	Once per day	Marketing message

4.6 Native Bridge Latency

Since 2008 different mechanisms have been used to implement native bridges in hybrid apps. Any mechanism that can be used to transmit a message between web view and native can be used. A thorough list of what can be used as a native bridge will be given in chapter 5.

Some work to compare different bridges and/or hybrid apps to similar native apps has been done in both industry and academia. A common comparison is to find out what is the added device access latency for a hybrid app in comparison to a fully native app. Corral et al. compared a hybrid app built with Cordova to a native app that implemented the same features for accessing device capabilities on Android [9]. They measured the time to complete an action, for example playing a sound or triggering device vibration. Each feature was tested 1000 times on both apps. This experiment inspired the same kind of comparison for Windows Phone.[44] Both experiments concluded that these features trigger slower in a hybrid

[44]Course work: Gustavsson and Kostopoulos "Performance Dissimilarities Between Hybrid and Native Windows Phone Applications"

app than in a native app: depending on the feature tested, the difference was ten to hundreds of milliseconds. The testing methodology that tests features end-to-end without identifying which parts of the hybrid app are slow is not valid. It does not test if it is the callback handling in the web view or the native bridge transmission speed. Or whether there is there something in the feature implementation itself (e.g. in the way of a sound is played) that is different on both code bases (something that is not part of native bridge's performance at all).

Hybrid app frameworks such as Cordova and Marmelade have built-in tests measuring the speed of their native bridge. These tests have been implemented so that the native bridge mechanism can be changed independently from the test. These framework tests are intended to compare the speed of the native bridge among different devices. It is common to test against a "ping" functionality: the native side just acknowledges the message and sends a reply to the web view.[45,46]

A commercial Cordova competitor called Trigger.io Forge published a blog post in 2012 on why they don't use Cordova claiming that their native bridge is up to 5 times faster than Cordova [13]. To support this claim, they have published a test project for comparing the native bridges of Cordova and Trigger.io.[47]

Puder et al. describe a native bridge based on WebSockets, an HTTP protocol extension that allows persistent connections [33]. The shell app implements an HTTP server with the WebSocket extensions. The communication is initiated with HTTP that does not block UI/main thread of the application. This design has been used in some popular hybrid apps. The LinkedIn engineering team shared details of their mobile app redesign that used WebSockets as the native bridge method. They compare WebSockets to the traditional synchronous PhoneGap method and have better results in both transmission latency and user interface locking due to the UI thread locking. The final LinkedIn app used different bridges for different types of messages to get the best performance [15].

In 2013 Mihai Parparita published a native bridge test tool for iOS along with some results on a few iOS devices [31]. The tool takes a framework independent approach and focuses solely on measuring different native bridges (it does not implement actions like playing a sound). With the tool a developer can compare transfer speeds of native bridges. The tool was released as public domain in GitHub under the name "WebView Communication

http://www.idt.mdh.se/kurser/ct3340/ht13/MINICONFERENCE/IRCSE2013-submissions/ircse13_submission_7.pdf

[45]Apache Cordova: Cordova Mobile Spec Exec test https://github.com/apache/cordova-mobile-spec/blob/master/www/benchmarks/exec.html

[46]Marmelade Technologies: Benchmark example in Marmelade documentation http://docs.madewithmarmalade.com/display/MD/Benchmark+Example

[47]Trigger.io: https://github.com/trigger-corp/Forge-vs-Cordova-Performance

Mechanisms".[48] The tool has been regularly updated to include the latest changes in the iOS platform. Figure 12 shows the user interface of the tool.

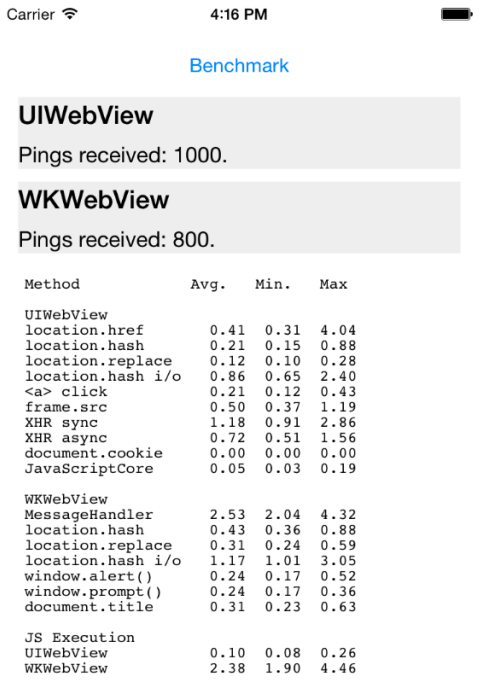


Figure 12: *WebView Communication* tool by Mihai Parparita running and showing results.

4.7 Platform Differencies

Hybrid apps are often considered a lucrative option for their "develop once, deploy everywhere" model [9]. However, each platform has their own implementation of the native web view control. Different web views (being based on different browsers' engines) render and function slightly differently as different desktop browsers do.

Android hybrid (or native) apps can use any web view component available for use. For example, the popular Google Chrome browser's Chromium engine can be compiled and distributed with the app. Crosswalk, an open source Chromium fork by Intel, is a popular alternative implementation with direct support for hybrid apps.[49] Bundling an app with a specific version of the Chromium engine instead of using the standard web view component of the device ensures similar operating across different versions.

[48] GitHub WebView Communication Mechanisms
https://github.com/mihaip/web-experiments/tree/master/webview-communication
[49] Crosswalk: https://crosswalk-project.org

The iOS platform differs from Android by more restrictive model both technically and in legal agreements. Apps in the App Store are not allowed to present any web content (including local) with any other web engine implementation than the ones that are available in the iOS SDK.[50] Therefore, the alternative web browsers like Google Chrome for iOS also use the built-in engine instead of Chromium.[51] Also WindowsPhone has a tighter security model and limits for apps. Efforts to port Google's Chromium browser have not been successful due to these restrictions. Hybrid apps can only use the web view that is based on the Microsoft browser technology.[52]

Apple's iOS SDK has three different web view components for developers to choose from. The original `UIWebView` is still the most widely used as it has the best support but it executes JavaScript slowly. In iOS8 Apple introduced `WKWebView` that does not have all the features required for advanced hybrid apps but it executes JavaScript faster and is a more modern browser implementation than `UIWebView`. In iOS9 Apple introduced `SafariViewController`, that enables developers to embed the full built-in Safari browser that performs as fast as the Safari browser app. `SafariViewController` does not allow browser chrome (address bar, browser UI controls) to be hidden or add any interaction with the native code, so it can not be used in hybrid apps [4].

4.8 Measuring Native Bridges on iOS

While hybrid apps look very similar to native apps, they often fail to *feel* native when the user starts to interact with the app [14]. On iOS platform a hybrid app developer is more locked down by the limits of the platform. The web view can not be changed from web views offered by the SDK, CSS transitions can not be individually enhanced, nor can JavaScript performance be improved. The apps's native bridge is one of the few components that a hybrid app developer can choose freely. The native bridge used is one of the few components where a hybrid app developer can have freedom of choice. The author of the most complete iOS native bridge test suite, Mihai Parparita, writes on the importance of the native bridge "When trying to hit 60 frames per second, spending 3 milliseconds of your 16 millisecond budget [1000ms/60] on pure overhead feels wasteful." [31].

Mallik et al. suggest that more "close to the flesh" (visual or audible delays) than "close to the metal" (CPU cycles used) metrics be selected for measurement of user perceived performance [24]. All of the aforementioned native bridge testing including Parparita's test focus only on the latency of message passing. This approach ignores side effects from message passing, such as UI thread locking when the message is large or there are a lot of small

[50] Apple: App Store Review Guidelines
[51] Chromium: Issue 423444: Use WKWebView on iOS 8
[52] Chromium: Issue 489037: Port Chrome to Windows 10 mobile

messages in a short time. As Mallik et al. suggest, the raw performance seldom reflects on the user perceived performance: message transmission latency is the wrong metric if it has other side effects.

In practice this means that if the time to start a native capability (play a sound) from user interface is low (the bridge is fast and the app is responsive to the user events), but the frame rate suffers, then the app does not feel native. In the same way, if the user interface maintains a high frame rate, but there is a user noticeable delay between the interface touch and feedback (sound played), the app does not feel native.

Table 2: Summary of noticed latency limits

Latency	**Domain**	**Description**	**By**
580ms	App usage	Last acceptable latency	Anderson et al. [3]
200ms+	App usage	Described as "sluggish" in review comments	Shengqian et al. [41]
100ms	Terminal usage	Noticeable computer terminal latency	Miller [26]
75ms	Gaming	Last acceptable latency	Delwadia [12]
20ms-40ms	Gaming	Noticeable latency	Delwadia [12]

Table 2 gathers all previously discussed latencies that were identified as noticeably lagging. In order for a hybrid app to be indistinguishable from the native performance levels, it needs to maintain better latencies than in that table. The previously discovered just-noticeable latency limits in various contexts are gathered in table 3.

Table 3: Summary of just-noticeable latencies

Latency	**Domain**	**Description**	**By**
40ms	Stylus	Scribbling a line	Ng et al. [28]
24ms	Hearing	Audio-haptic difference	Adelstein et al. [1]
13ms	Human eye	Brain processes static images	Potter et al. [32]
20ms	Tapping	Selection latency	Jota et al. [19]
11ms	Dragging	Dragging latency	Deber et al. [11]
7.5ms-15ms	3D VR	Head tracking	Regan et al. [35]
6ms	Stylus	Dragging large box with a stylus	Ng et al. [28]
3-5ms	Hearing aid DSP	Audio Latency	Agnew et al. [18]
2.38ms-11.36ms	App usage	Dragging with finger	Ng et al. [29]
2ms	Stylus	Dragging small box with a stylus	Ng et al. [28]

From that table we can see that a noticeable latency for tap-based input is around 20ms. For dragging it is safe to say that it's around 3ms. In a hybrid app context this means that the time to deliver a message from web view to native (or the other way around) for something to happen should then be below 20ms. Dragging latency is affected by how often the web view

can be redrawn. As the UI is redrawn every 16.667ms (1000ms/60), the UI must be redrawn every 19ms (16.667ms+2.37ms) before the user notices a lag.

When measuring the native bridge performance the following properties should be taken to consideration:

- **Direction** of the communication
- **Payload** of the message transmitted. A message can be either a small JSON message, such as:
  ```
  { action: "playAudio", fileName: "click.mp3" }
  ```
  Or a larger message, such as some image data:
  ```
  { action: "writeFile", fileName: "photo.jpg",
  data: "base64encodedimagedata...of5megabytes" }
  ```
- **Interval** of message sending. A stream of device's movements from the accelerometer sensor is sent more often (for example every 10ms) than GPS location updates (every 5 seconds).
- **Render pause:** How long the native bridge blocks the screen refresh updates (JavaScript event loop or the native UI thread). Pausing should be measured in milliseconds, as frames per second has too large a resolution (one frame is 16.667ms).
- **Latency:** How long it takes to transfer one message.

None of the existing work on performance analysis of native bridges implements all of the properties listed above.

5 NativeBridgeBenchmark Tool

The *NativeBridgeBenchmark* is a tool written by the author for measuring native bridge performance. The tool allows tests to be defined in various configurations and is easily extendable: a new native bridge can be added or an existing one modified. Different directions can be tested individually. Message size can be freely set along with the sending interval. It also supports parallel testing with multiple devices to shorten the total time taken for a test run. The full source code of the project has been released as open source under the MIT license and is publicly available for download from GitHub.[53]

In the rest of this chapter the architecture and usage of this tool is described with examples. Then the implemented native bridges are listed along with implementation details of each bridge.

5.1 Requirements

The NativeBridgeBenchmark tool was developed because none of the existing native bridge measurement projects were constructed in a way that took into consideration all the properties previously introduced (direction, payload, interval, render pause and latency), nor did any of the existing measurement projects include all the native bridges that were discovered for this work. This level of granularity of parameter settings is required to properly benchmark bridges in the way described in the previous chapter.

Also, no other tool provided a way to describe a test suite (testing multiple native bridges with certain parameters). A test suite can be re-run multiple times over time as new iOS versions become available. Multiple test suites need to be stored and be runnable at the same time. Over time, new native bridges will be discovered and old ones will stop working. The tool needs to be extendable for future changes in the iOS.

For the user, the usage requirements are a Mac OS X computer with XCode development tools and as many iOS devices as desired. An Apple Developer Certificate is also required for the testing with iOS devices.

5.2 Usage

In NativeBridgeBenchmark the following terminology is used:

- *Client* - The minimal hybrid app that performs tests in the device.
- *Companion* - The web server application that hosts the tests and stores the results.
- *Method* - The native bridge mechanism used in the test.

[53] GitHub: Repositories https://github.com/matti/NativeBridgeBenchmark, https://github.com/matti/NativeBridgeBenchmark-companion

- *Bridge head* - The receiving handler function in the native code or in the web view.
- *Payload* - The message that is transmitted (randomized string of desired length).
- *Interval* - The waiting time between the message transmissions.
- *Samples* - The number of samples to be performed.
- *Render pause* - The longest time the web view was unable to render anything (event loop blocked).
- *Transfer time* - The time it took to transfer the message.

The NativeBridgeBenchmark consists of *client apps* (that are run in mobile devices) and *a companion web app* run in a desktop computer. The companion app can also be run in a public web server. The iOS client is written in Objective-C and has a similar architecture (one full screen web view inside a native shell app) to hybrid apps done with open source hybrid app frameworks (such as PhoneGap/Cordova) to provide comparable results. Clients connect to the companion web app and perform an individual test or a test suite defined in the companion. After a test is completed (all messages have been sent), the client sends results back to the companion before moving on to the next test. The companion has been designed to support different clients, including clients for different mobile platforms. The companion is written in Ruby on Rails.

In a typical testing setup there is one desktop computer, a WLAN base station and different mobile devices selected for the test. The desktop computer runs the companion web app and the mobile devices have the client app installed. All the devices and the desktop running the companion web app connect to the same wireless network. Clients load the test application from the companion app in their web view. This enables faster development as the client's native shell does not need to be recompiled for every change.

The user installs the companion app on her computer and starts it. The companion app is essentially a local web app that has a simple user interface for configuring test configurations. In the user interface, the properties of a single test can be defined and stored in a database. For a single test, the user sets the direction, method, samples, payload and interval.

The client app is an Objective-C project that is opened with XCode. The user compiles the application with XCode and installs it to her iOS device. The app starts and opens the companion web app running on the computer in a web view. The client and companion apps are both essentially the same web app, the difference is just a matter of the device that they are used in. When the user starts the previously defined single test, message sending and

result storing start. Once all the messages have been sent, the activity stops and the user can start another test.

A test suite is a collection of multiple different tests. These tests can be created either with the GUI, or with a script that inserts the test parameters directly in the database. A test suite run is started similarly to a single test. The only difference is that once the test ends, a new test with different parameters is automatically started. Using automatically advancing test suites saves the users time, as testing multiple native bridges with different configurations can take hours, depending on the configuration.

Once the tests have been run, the web interface shows the results (samples) in a table format where one row is one test result sample. For each result sample, the transfer time and render pause are stored. The results can be exported as a CSV file for later analysis with, for example, Excel.

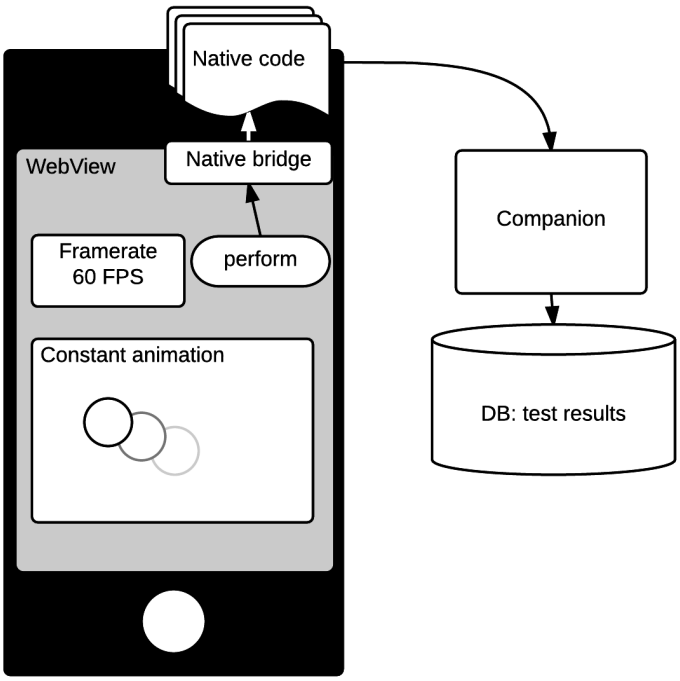


Figure 13: Illustration of the NativeBridgeBenchmark client and the client's communication to the companion web app.

Figure 13 shows an illustration of a mobile device running the client app. In the device the client app has been started and the wrapped web view shows a test page from the companion server. The web view is showing the current frame rate, some constant animation for an immediate visual feedback of the performance and a tappable button (labeled "perform") to

manually start a new test run.

A test run is a series of tests with the same parameters. The parameters define the direction of the test run, the native bridge to be used, the total number of messages to be sent, the message sending interval and the payload (message size) to be transmitted in one test. The web view needs to be selected before the test run is started.

5.3 Architecture

A sequence diagram of a user initiated test from the web view to native is shown in figure 14. In the diagram the client loads the test page, then the user starts the test by pressing a button on the screen. The JavaScript in the web view sends the message to the bridge head component on the native side. There the desired native call is invoked (the desired method in the SDK but in the tool just an empty method, since no actual action needs to perform). Then, an asynchronous record call is initiated so that the native bridge can return control back. The asynchronous recording forwards the message parameters from the call including a timestamp from that moment when the message was received in the bridge head. When the test run has been completed, the client sends results to the companion.

Another sequence diagram in figure 15 shows where the message is sent another way around (from native to the web view). Also here the user starts the sending by requesting messages with certain parameters. In a few seconds the native side starts to send messages to the web view. The JavaScript method `bridgeHead()` gets called for each message. This method does similar recording of the received messages with timestamps. At the end all these recordings get sent to the companion.

5.4 Message Format

The messages that are sent in the tool are represented as URLs. The native bridge requests are identified by their protocol, URLs that start with `nativebridge://` instead of `http(s)://` are processed as native bridge messages and not sent to the networking layer. This custom protocol was done as an additional experiment on which protocols can be used with native bridges. If the method does not support this custom protocol, then HTTP is used. The URL contains parameters for the benchmarking that are stored in the URL as follows:

```
nativebridge://ping?webview_started_at=STARTED&payload=MESSAGE&
    method_name=METHOD_NAME&fps=FPS&render_paused=PAUSED
```

All capitalized constants in the URL are replaced for every message sent with the following values:

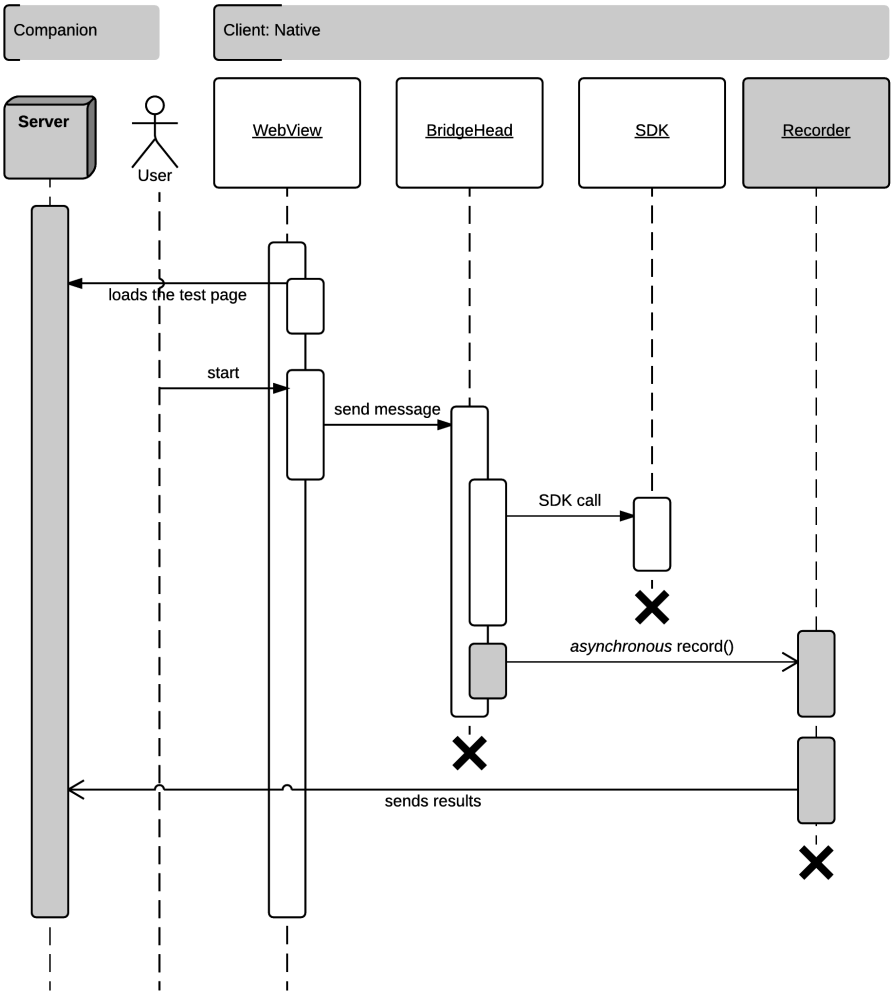


Figure 14: A diagram of a single message sending from the web view to the native code in NativeBridgeBenchmark

- `STARTED` - Time when message sending started (as UNIX epoch timestamp with milliseconds)
- `MESSAGE` - Message contents as a string
- `METHOD` - Native bridge method used (e.g. `http.websockets`)
- `FPS` - Current FPS of the web view (not used in the results)
- `PAUSED` - Largest event loop rendering pause since the last message was sent (resets after every message is transmitted)

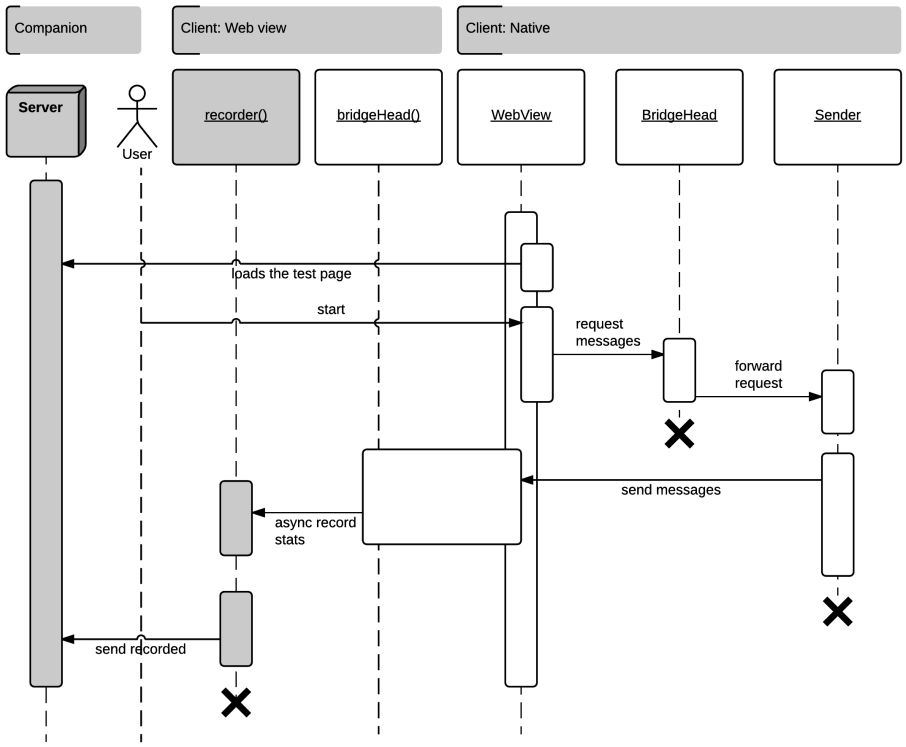


Figure 15: A diagram of a single message sending from the native code to the web view in the NativeBridgeBenchmark

This message format (URL) is used with all the bridges (even the ones that do not use URL loading) for consistency.

5.5 Example Measurement

The web view where the test runs has a loop function implemented with a `setAnimationFrame` that gets called every 16.667ms, if the event loop is able to perform these calls (as described previously). The function compares the current time to the time it was last called, records the difference in a variable, and also prints it to the screen if it was bigger than the last stored difference. If the event loop is free, the function gets called every 16.667ms and the screen is constantly updated with the number "17" (ceiled i.e. rounded to nearest millisecond). If the event loop gets blocked, the screen does not update until the execution resumes, then it is updated again with a larger number. For example, if the event loop was paused for 35 milliseconds it renders "52" (16.667+35). If the event loop resumes normally it doesn't update the number, since 17 is smaller than 52.

Every message sent on the native bridge resets this number to -1 (and

then the next event loop update will set the number to 17). This number is used by the recorder component to store the maximum pausing that happened during a single message sending.

At the start of each test, a payload is generated. The payload is a randomized string of a desired length, stored in the test settings. This payload will be assigned to every test message that was described above.

The test run is a loop that runs until the number of requested messages is sent. For each message, the tool populates the test string with the defined parameters and sends it with the requested native bridge. When sending messages from the web view to the native, the render pause is attached to the message just before the message gets sent (and after that reset to -1). When sending messages from the native to the web view, the rendering pause is recorded upon receiving the message (and then reset to -1).

Once the message is received on the other side (native or web view), the current time is compared to the message's timestamp. The difference is the time it took to transmit the message i.e. the transfer time. The length of the rendering pause is taken either from the message or from the value on the screen (depending on the direction). The result sample is then stored in a temporary location, waiting to be sent to the companion.

When all the messages for the test have been sent, the tool sends the gathered result samples to the companion app, that then stores the results in the database. After the samples have been "flushed", the tool advances to the next test or stops.

5.6 Native Bridges for UIWebView

All the bridging methods implemented in the tool are listed in here. This listing of native bridges was gathered from existing work and additionally by reading the iOS SDK and the W3C HTML specifications. In the listing an identifier is first given in bold (e.g. **html.script**) followed by a brief explanation and a code example. The identifier given is used to refer to the method in the results.

The code example only shows the relevant part or implementation details, omitting the boilerplate code required. For example a fully functional `html.iframe` call would be as follows:

```
<button onclick="callNative('hello')">Say hello to native</button>

<script>
function callNative(msg) {
  var iframeElem = document.createElement("iframe");
  var iframeElem.src = "nativebridge://"+msg;

  document.body.appendChild(iframeElem)
}
</script>
```

In the listing below only `<iframe src="nativebridge://...">` is given as a code example to show the relevant part.

From UIWebView to Native

There are three main categories for an `UIWebView` to send a message: *request*, *connection* and *direct access* based bridges.

Request based bridges

A request based bridge uses a networking request to send the message. This was the original method PhoneGap used to send a messages in 2008. Different kinds of actions trigger a remote resource load that can be intercepted:

- **html.script** HTML script element is added to the page that triggers a network request. Requests must start with HTTP.

  ```
  <script src="http://nativebridge://...">
  ```

- **html.link** CSS document linking is used similarly. Must start with HTTP.

  ```
  <link href="http://nativebridge://...">
  ```

- **html.iframe** Embedded HTML document
  html.img Image element of bitmap types (JPEG, GIF, PNG, ...)
  html.object Embedded multimedia object (video, audio, ...)
  html.embed Embedded external application or interactive content

  The different types of visual elements above can be added as *hidden* elements (with CSS). If images are not set as hidden, they will be shown as `?` icons in the app. All images must be loaded with the HTTP protocol.

  ```
  <iframe style="display: none" src="nativebridge://...">
  <img style="display: none" src="http://nativebridge://...">
  <object style="display: none" data="nativebridge://...">
  <embed style="display: none" src="nativebridge://...">
  ```

- **html.svgImg** SVG (Scalable Vector Graphics) image element. An image element is created inside a SVG document.

  ```
  svgImageElem = document.querySelector("svg image.nativebridge")
  svgImageElem.setAttributeNS('http://www.w3.org/1999/xlink',
                              'href',
                              'http://nativebridge://...')
  ```

- **location.href** Navigate browser to a new URL
  location.replace Same as the `location.href`, but does not maintain

the URL history
location.hash Update the URL anchor part (http://example.com/page/#anchor)
location.replaceHash Update URL anchor part with `location.replace`

Manipulating the browser location with JavaScript causes a new request to trigger.

```
location.href="nativebridge://...";
location.hash="nativebridge://...";
location.replace="nativebridge://...";

// replaceHash, notice the # prefix
location.replace="#nativebridge://...";
```

- **a.click** Triggering a link element click programmatically with JavaScript.

```
var a = document.createElement("a");
a.href = "nativeBridge://...";
a.click();  // simulate a click programmatically
```

- **xhr.sync** `XMLHTTPRequest` synchronous HTTP request
  xhr.async Asynchronous request

  Both must use the HTTP protocol.

```
var async = true//false;
xhr.open("get","http://nativebridge...",async)
xhr.open("get","http://nativebridge...",async)
```

These different ways of triggering the load are worth testing even if it seems irrelevant at this point, because the internal implementations vary greatly (this can be seen in the test results). All of the above methods can be intercepted with either the `shouldStartLoadWithRequest` method of the `UIWebViewDelegate` that gets called when certain types of requests start [54] or with a custom `NSURLProtocol` (for XHR) that decides if the request is a native bridge message or not [55].

Connection bridges

In connection based bridges the request from the web view is not interrupted. A TCP connection is created and the message is sent through it. For these bridges to work, there needs to be a server inside the native shell app.

- **xhrlocal.sync** Like `xhr.sync`, but connects to a web server running in a randomized port (because ports might be taken by other applications). For example the `CocoaHTTPServer` [56] library can be used to implement

[54] UIWebViewDelegate in Apple iOS Developer UIKit Framework Reference
[55] NSURLProtocol in Apple iOS Developer Foundation Framework Reference
[56] https://github.com/robbiehanson/CocoaHTTPServer

a server.
xhrlocal.async Same as above but as asynchronous version (like `xhr.async`).

Naturally these methods need to use HTTP. The message payload is limited to approximately 6*1024 before the web view cancels the request (because the payload is embedded in the URL that has a maximum size).

```
var async = true//false;
xhr.open("get","http://localhost:1234/nativebridge...",async);
xhr.open("get","http://localhost:1234/nativebridge...",async);
```

- **xhrpost.sync** Like the `xhrlocal.sync`, but uses HTTP post method and therefore can deliver larger payloads (because the payload is sent as the request body, that is not limited).
  xhrpost.async Same as asynchronous.

```
var async = true//false;

var xhr = new XMLHttpRequest();
xhr.open('POST', 'http://localhost:1234/nativebridge', async);
xhr.send(messageJSON);
```

- **websockets** WebSockets is an extension to HTTP that enables web applications to maintain a bidirectional connection to the web server.[57] The previously mentioned `CocoaHTTPServer` also implements WebSockets support. WebSocket messages are virtually limitless on the protocol level, but the payload size is limited to the server data structure handling. `CocoaHTTPServer` can handle tens of megabytes and can possibly be extended to handle more.

```
var ws = new WebSocket("ws://localhost:31337/service");
ws.onopen = function() {
  console.log("connection opened");
  ws.send("http://nativebridge...");
}
```

Direct access bridges

These are unofficial ways to expose native runtime directly to the web view.

- **alert** Alert dialog with a message and a dismiss button
  confirm Dialog with a message and confirm or cancel buttons
  prompt Dialog for user to input text

  These popup dialogs can not be officially modified. However, the `WebUIDelegate` from OS X libraries shows that it is possible (and

[57] W3C The WebSocket API http://www.w3.org/TR/websockets/

allowed in desktop applications) to modify them [58]. The idea is to override so that they get called, but don't present the dialog at all. The message for the dialog is a native bridge message. Using these bridges may result in App Store rejection [59].

```
alert("nativebridge://...");
confirm("nativebridge://...");
prompt("nativebridge://...");
```

- **jscore.sync** Since iOS 8 the `JavaScriptCore` framework can be unofficially hooked to `UIWebView`'s JavaScript runtime allowing custom native functions to be added to the web view (e.g. `window.nativeBridge(message)`). This method is the closest to having a *native* native bridge implementation.

```
exposedViewController.nativeBridge("nativebridge://...");
```

From Native to UIWebView

In a hybrid app the native runtime seldom starts sending messages to the native side without an explicit request to do so as most of the logic is executed in the web view. There are considerably fewer options for messaging to the web view. The main categories are *connection*, *observe* and *direct*.

Connection bridges

- **websockets** As stated above, the websockets can be used bidirectionally. The bridge can be used once the web view has opened the connection.

```
[webSocket sendData: JSONAsNSData];

// JavaScript counterpart in the web view:
ws.onmessage = function(event) {
  bridgeHead(event.data);
}
```

Observe bridges

- **location.hash** URL anchor part changes can be observed with JavaScript (e.g. from http://domain/path#something to http://domain/path#else)

```
currentURLComponents.fragment =
  [ NSString stringWithFormat:@"#webviewbridge:%@", [messageJSON]];

NSURL *newURL = [currentURLComponents URL];
```

[58] WebUIDelegate in Apple's Mac Developer Library Webkit Framework Reference

[59] Discussion at StackOverflow http://stackoverflow.com/questions/3163103/how-can-i-automatically-click-cancel-when-an-ok-cancel-alert-pops-up-in-the-uiwe

```
NSURLRequest *newRequest =
  [[ NSURLRequest alloc ] initWithURL: newURL ];

dispatch_sync(dispatch_get_main_queue(), ^{
  [currentUIWebViewController.webView loadRequest:newRequest];
});

// JavaScript counterpart in the web view:
window.onHashChange = function() {
  bridgeHead(window.location.hash);
}
```

Direct access bridges

- **webview.eval** JavaScript can be evaluated to runtime. Evaluation pauses the event loop and always takes precedence from the hybrid app's JavaScript.

```
NSString *evalString =
  [ NSString stringWithFormat:@"bridgeHead('%@');", messageJSON ];

dispatch_sync(dispatch_get_main_queue(), ^{
  [currentUIWebViewController.webView
        stringByEvaluatingJavaScriptFromString:evalString];
});
```

- **jscore.sync** The JavaScriptCore Framework allows developers to evaluate JavaScript within native code [60]. It is possible to connect custom JavaScript context to the web view as `UIWebView` uses the same framework internally. This allows direct manipulation of the web page's JavaScript runtime, although it is not officially supported. As with `webview.eval`, any evaluation pauses the event loop.

```
NSString *evalString =
  [ NSString stringWithFormat:@"bridgeHead('%@');",
    messageJSON ];

dispatch_sync(dispatch_get_main_queue(), ^{
  [currentUIWebViewController.jsContext evaluateScript:
    evalString];
});
```

5.7 Native Bridges for WKWebView

The faster `UIWebView` alternative `WKWebView` has fewer options for request based native bridges as some of the request types bypass the apps's native code completely.

[60] Apple iOS Developer JavaScriptCore Framework Reference in Apple iOS Developer

Request based

Request interception is not as broadly supported in `WKWebView` causing some bridges like the HTML script and image elements to pass through the native code. The following bridges that work in the `UIWebView` are not available: `html.script`, `html.img`, `html.svgImage`, `html.link`, `xhr.async` and `xhr.sync`.

Connection bridges

All previously listed connection bridges work (as expected since only request intercepting has had changes).

Direct access

Internally `WKWebView`'s fast JavaScript context runs in a separate process for security reasons. This also causes the previously introduced `jscore.sync` bridge not to work as the JavaScriptCore framework can not attach itself to a different process [61]. However there is a new and officially approved way to do native bridging:

- **webkit.usercontent** A class conforming to `WKScriptMessageHandler` protocol can receive messages from JavaScript running in a web page. Only string messages are supported (essentially JSON).[62].

  ```
  webkit.messageHandlers.nativeBridge.postMessage("nati...")
  ```

- **webkit.alert** Alert dialog with a message and a dismiss button
  webkit.confirm Dialog with a message to confirm or cancel
  webkit.prompt Dialog for user to input text

  These methods are the same as `UIWebView`'s counterparts, but in `WKWebView` these methods are *designed* to be extended and should not result in App Store rejection. The `WKUIDelegate` has methods for extending the dialogs.[63]

- **webkit.title** *Key-Value Observing* (KVO) can be used for observing `WKWebView` object's title property changes. The observer gets notified from every title change [64]. This method is unique to `WKWebView` as the property observing can not be used in the `UIWebView`.[65]

  ```
  document.title = "nativebridge://...";
  ```

[61] Apple's Radar bug tracker comments in http://www.openradar.me/17680867
[62] WKScriptMessageHandler in Apple iOS Developer WebKit Framework Reference
[63] WKUIDelegate in Apple iOS Developer WebKit Framework Reference
[64] Cocoa Core Competencies in Apple iOS Developer Library
[65] Is UIWebView KVO compliant? http://stackoverflow.com/questions/8145482/is-uiwebview-kvo-compliant

From Native to WKWebview

All bridges (`http.websockets`, `webview.eval`, `location.hash`) that work with `UIWebView` work with `WKWebView` except `jscore.sync` for the reasons given previously. Oddly, there is no counterpart to the `webkit.usercontent` bridge when accessing the web view from native.

5.8 Bridge Deprecation

As the iOS platform evolves some bridging mechanisms are no longer available. For example internal changes introduced in iOS8 caused two known bridges, *cookies* and *localStorage* to stop working. Cookies are text values stored and persistent in the browser (or in the web view) by an identifier key. Similar storage mechanism *localStorage* can be used to store larger values. JavaScript can be used to store and read these values. Until iOS 8 it was possible to observe changes in the *CookieStore* (the database where the cookies are stored internally) and in the internal SQLite database used for the localStorage. The maximum message size of the `xhrlocal` bridge also changed at the same time, coming down from 7*1024 to 6*1024.

Apple's App Store review guidelines and policies are also subject to change over time causing an app to be rejected in the review process if it uses a recently forbidden method. As long as Apple does not fully endorse native bridging (especially with the `UIWebView`), multiple bridging methods are needed to keep hybrid apps working.

6 Results

NativeBridgeBenchmark is a valuable tool for a hybrid app developer who can specify her own parameters and select use her own test devices. For this chapter a sample test suite run was performed with four iOS devices. First the testing setup (parameters, devices) is described. Then some of the results are picked for analysis. Then a suggestion for a method of selecting the best native bridges is given. Lastly the results are compared to related work.

6.1 Test Setup

The section "Frequency and Size of Native Bridge Messages" outlined that the number of messages sent varies by the app type, user and usage situation. Two different sizes of message payloads were identified: small, that is a descriptor of an action (usually as a JSON, for example, a filename of the sound to be played) and large, some binary data (for example, contents of a photo or audio recording).

For this work, a demonstrative test suite was run in the NativeBridgeBenchmark tool with the following configurations:

Name	**Payload size**	**Iterations**	**Interval**
small	6*1024	100	1000ms
large	2048*1024	100	1000ms

The interval is set to 1000ms, which represents intense app usage where the user makes a selection (for example taps the screen) every second, causing a native bridge message to be sent. It does not represent a case where the screen is tapped rapidly, as that is not a common app usage (for example, typing with keyboard in a hybrid app always uses the native keyboard control). It also does not represent a case where accelerometer updates are streamed every 25ms with the native bridge. However, the results of this run are also applicable to that situation: if the time to transmit a (small) message is less than 25ms, then that bridge is suitable for delivering those accelerometer update messages (the transfer time will be the same if the interval is set to 25ms).

The "small" payload was set to 6*1024 characters of payload to show just enough differences between bridges, as a near zero payload did not diversify enough. 6*1024 characters can hold all small descriptor messages (including, for example, contact names and details from the device's phonebook). This payload size was also the maximum size that `xhrlocal` bridges can handle, so it is the upper bound of small messages that can be tested with all bridges.

The "large" payload was calibrated to match the slowest device's capabilities of delivering a message within the 1000ms window with all bridges. So this payload is the lower bound of large messages. It is, for example, big

enough for small photos and audio recordings (that represent only a minority of the typical app usage previously described).

It was decided to not include any more dimensions (e.g. different intervals) to this sample test run, as the combinations would have been too large to meaningfully analyze in the scope of this work. Initially a "medium" configuration was also included, but the results did not significantly differ from the large payload.

Test was run on both older and newer iPads, iPhone and iPod Touch to cover all iOS device types. The iPod touch is almost identical to an iPhone. All devices ran iOS version 9.2.1 (latest as of 6th March 2016). The following table shows the specifications of the devices used:

Model	**Released**	**Processor**	**Resolution**
iPhone 6	Sep 19, 2014	ARMv8-A 1.4 GHz dual core 64-bit	1334 x 750
iPod Touch 6th gen	Sep 9, 2015	ARMv8-A 1.1 GHz dual-core 64-bit	1136 x 640
iPad Air 2	Oct 22, 2014	ARMv8-A 1.5 GHz tri-core 64-bit	2048 x 1536
iPad Mini 1st gen	Nov 2, 2012	ARMv7-A 1.0 GHz dual-core 32-bit	1024 x 768

Before the test runs the devices were restarted and had a full battery charge while being connected to the power. NativeBridgeBenchmark forces the screen to stay on (so it does not dim during testing) and auto locking was disabled (the application stays on the screen at all times). Rotation was locked to the portrait position. The screen was not touched after starting the test run as this would generate touch events that would potentially slow down the web view [22].

In a test run all devices performed tests for both payloads in one direction (e.g. from `UIWebView` to native). After a run the test application was restarted and another run started (e.g. from native to `UIWebView`. The biggest test run has 21 methods to test, so that test took a minimum of 70 minutes (21 tests * 2 payloads * 100s) to complete. The following table lists all the test runs:

Devices	**Originator**	**Target**	**Payload**
4	UIWebView	Native	small, large
4	Native	UIWebView	small, large
4	WKWebView	Native	small, large
4	Native	WKWebView	small, large

In the results, every test run had 1 to 4 clear outliers. Since one method test takes 100 seconds to execute (100 samples, every 1 second) it is likely that these outliers were due to internal iOS background processing and not related to the testing tool or method. The few outliers were filtered from the results with the following criteria:

Delete the sample if the sample value is 3 times greater than the standard deviation from the test's average (scoped with the payload and the same device).

The test sample size is therefore between 96 to 100 for each configuration. It was also decided to cut 10% of the samples (~9 samples) from the beginning and the end of the series to eliminate any *warm-up* or *cooldown* effects. With this filtering applied the sample sizes are ~77 for each test configuration in the results.

6.2 Discussion

The following discussion of some of the results is intended to guide an interested reader to both understand the full results in the appendix and to help in designing and running her own benchmarks. Since the tool was developed by one programmer, these sample results are cursory and subject to programming mistakes in the implementation. More important than individual test results is the testing method, the modifiable open source tool and the listing along with sample implementation of what can be used as a native bridge.

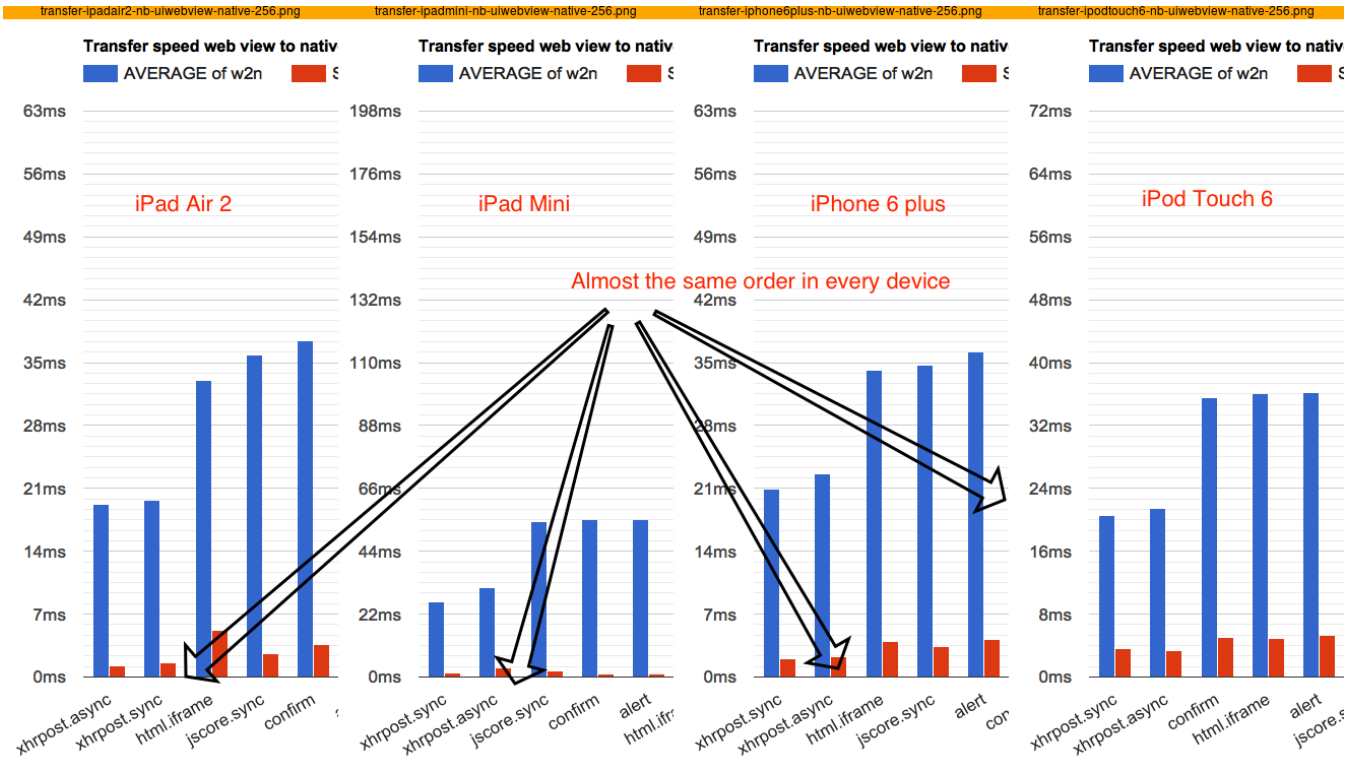


Figure 16: The best performing methods are the roughly the same in all four devices (iPad Air 2, iPad Mini, iPhone 6 plus, iPod Touch)

The number of different test combinations is quite large (2 web views * 2 directions * 2 payloads * 4 devices) to have a short overview. However, it

seems that the results are roughly uniform across different devices as can be seen in figure 16. For this reason we will select the iPad Air 2 and the large payload size for this brief discussion. It is worth mentioning that the large payload visualises the differences better, but the results do not apply for the other payload sizes.

For each web view and direction we will see what are the most performant pairs in both transport speed (the latency to deliver the message) and render pausing (the time that JavaScript is unable to run and update the user interface). For both numbers the lower, the better. The minimum possible value for the rendering pause is 16.67ms as the user interface is drawn 60 frames per second (1000ms/60 = 16.67).

In the figures blue color represents the average and red represents the standard deviation. The graphs are always sorted ascending by the average (the best is usually the lowest value).

Series Graphs

The following sample series graphs are not very useful for analysing the results because they are visually difficult to interpret due to the number of bridges implemented and their similar performance. They are, however, important for our later analysis. Figure 17 shows the individual samples in a series for both pause (top) and transfer (bottom) when testing from `UIWebView` to native. The horizontal axis for both graphs is seconds (sample interval was set to 1000ms). The labels for the horizontal axis were removed for clarity (the labels would be 1,2,...,100). In the upper pause graph the vertical axis is the maximum time the renderer was unable to render anything in the user interface as one message was sent. After each message the maximum was reset and then incremented if a larger value was found. If the message sending did not pause at all (for example with a fast bridging method, fast device and small payload) the line stays flat at around 17ms (annotated in the graph).

In the lower transfer graph, the time in the vertical axis is the total time spent to deliver each payload. At the bottom of both the series we can see a blue line (`xhrpost.async`, annotated) that is clearly the least pausing bridge in the upper pause graph, but it is hard to say if this bridge also transfers faster than the brown (`xhrpost.sync`, annotated). For this reason the rest of the figures presented are average aggregates for better comparability. For every upcoming aggregate graph there is a corresponding series graph available in the appendix.

From `UIWebView` to native

Figure 18 shows that the connection based `xhrpost.async` (the previous blue line in figure 17) transfers the payload fastest (first graph, annotated

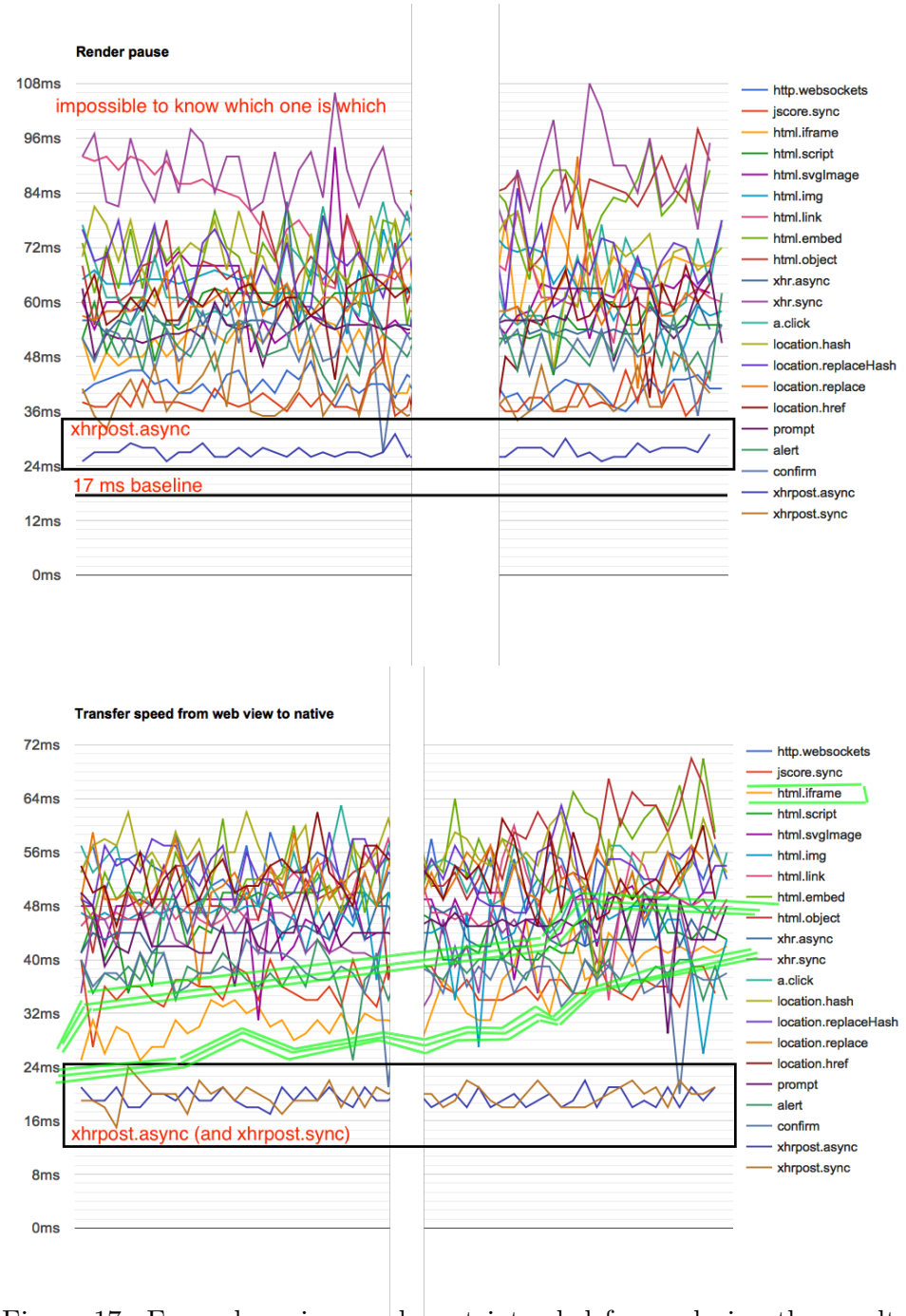


Figure 17: Example series graph, not intended for analysing the results. Shortened in the middle. From `UIWebView` to native (iPad Air 2, large payload)

with 1) while it also keeps pausing at a minimum (second graph, annotated with 1). The second fastest synchronous version `xhrpost.sync` (annonated with 2) is almost as fast but causes the renderer to pause significantly more as can be seen from the pause graph where it's the third. This is expected as the internal implementation pauses JavaScript execution until the message has been transmitted. In addition at the top parts of both the graphs are the direct `jscore.sync` (underlined) along with the `confirm,alert,prompt` (underlined).

While the `html.iframe` (annotated with 3) is even faster than the previously mentioned bridge methods, it causes significant pauses in rendering. This method also has a higher standard deviation (red bar) than the others, meaning that its' performance varies significantly. This variation can be seen in figure 17 where the yellow line that represents `html.iframe` (annotated with three green lines) is becoming slower over time, deviating from the consistently fast `xhrpost.async` line at the bottom. In general the standard deviation seems to be higher for these methods that are triggered by adding an element to the DOM (`html.*`). The browser rendering is complex and this specific variation can be caused by memory usage (although the nodes are just single elements) or just a combination of different rendering issues. This slowdown is not specific to just hybrid apps: the WebKit project bug tracker lists numerous longstanding memory and CPU issues with DOM nodes [66]. As the test added 1 node to the DOM every 1000ms, it could be that

Lastly the opposite is true for the connection based `http.websockets` where the pause bar is low, but the transfer speed is over two times larger (slower) than the fastest bridge.

With the large payload and the selected device we can see that only `xhrpost.async` and `xhrpost.sync` transfer faster than the previously identified latency noticeability threshold of 20ms. And all methods will cause render to pause for more than 19ms, causing a noticeable delay to the user interface (for example when dragging). It is worth mentioning that the large payload is extremely large for a native bridge message. In a real application these kind of messages are rarely seen.

Figure 19 that shows the same test with the smaller payload is given here for context. It shows that the six leftmost methods (annotated with red) can be used without any noticeable delay when the user is dragging, and that all these six methods can be used to transfer messages without noticeable latency (all are under 20ms in rendering pause). The rest of the methods (annotated with yellow) in the same graph will cause a noticeable latency (cause over 20ms of rendering pause).

The transfer speed of those same bridges is in figure 20. Interestingly,

[66]The WebKit project issue opened in 2009: Unbounded memory growth when adding and removing images, https://bugs.webkit.org/show_bug.cgi?id=31253

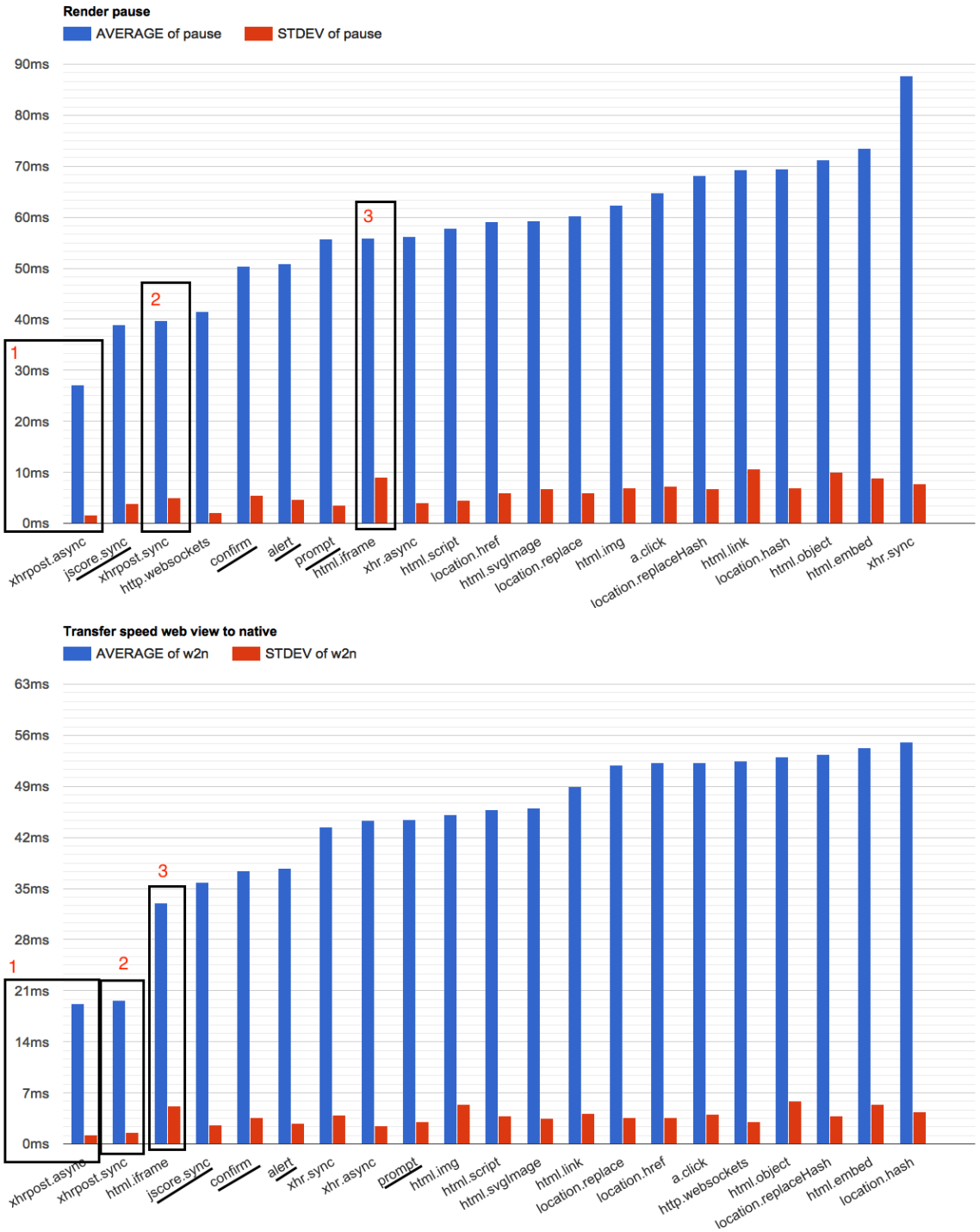


Figure 18: From `UIWebView` to native (iPad Air 2, large payload)

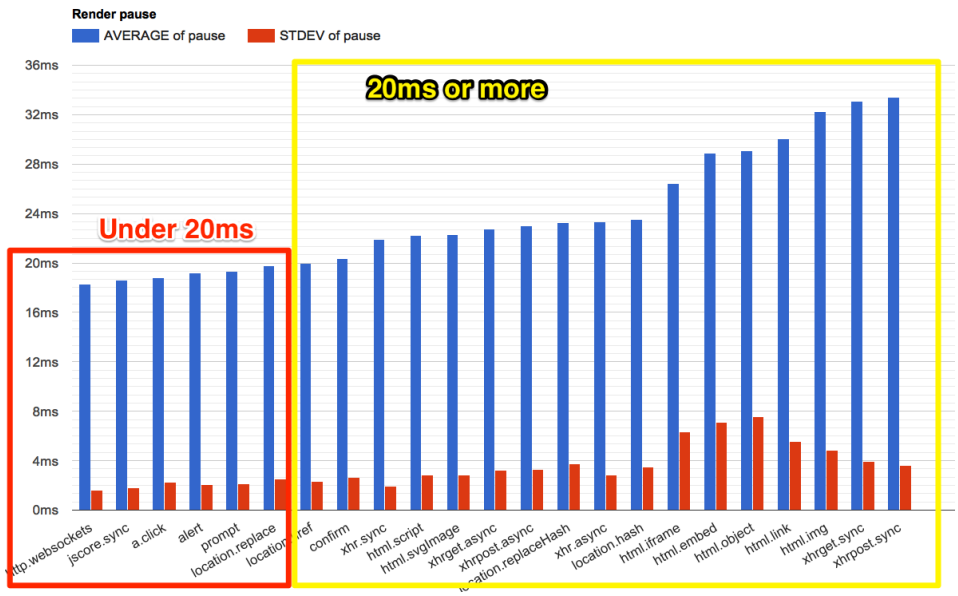


Figure 19: Showing bridges that don't cause noticeable delay and bridges that cause. From `UIWebView` to native (iPad Air 2, small payload)

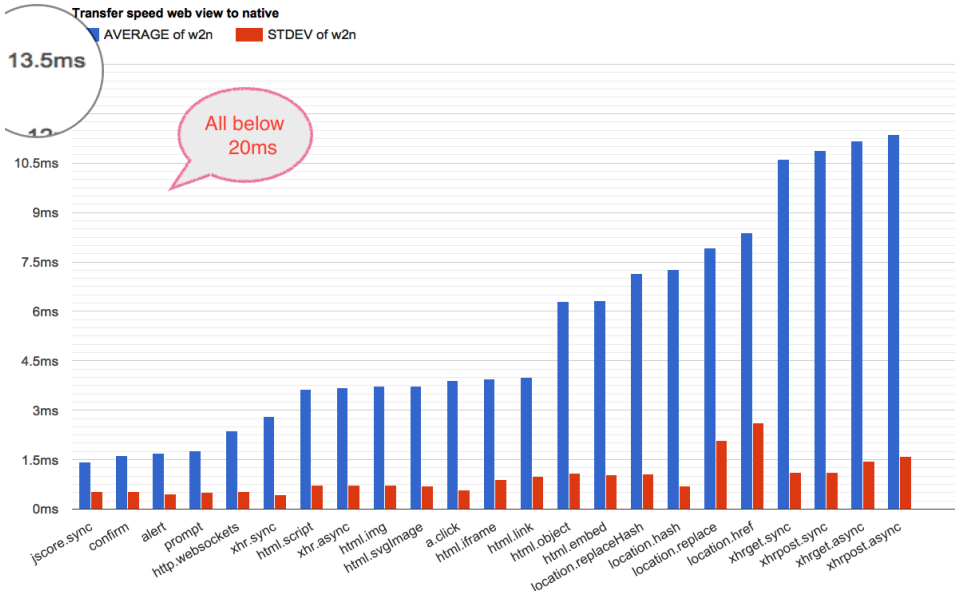


Figure 20: All bridges transfer fast enough (below 20ms) on this device for not being noticeable. From `UIWebView` to native (iPad Air 2, small payload)

this shows that all the bridges transfer in less than 20ms. As mentioned previously, measurement methods that only account for transfer speed, but do not consider render pause, are clearly inadequate for selecting native

bridges that make hybrid app indistinguishable from native.

In real hybrid applications the user interface can update the state change visually before the message has been completely transferred (or even sent!). Also it might not be common to send messages while dragging and even if messages were sent, then a couple of slower messages every now and then might not be noticeable by the user even if a slower bridge was used.

The rest of the discussion will continue to use the large payload to better show the differences between the different methods.

From `WKWebView` to native

When changing the web view to `WKWebView` some of the ordering has changed in the figure 21. But, similarly to the previous figure, the `xhrpost` is both the fastest and the render friendliest. The only real difference here is the slightly slower `webkit.usercontent` method that should have similar internal implementation to the `jscore.sync` but apparently there are some internal differences. The rest of the methods seem to follow the same behaviours as in the previous figure although the `WKWebView` is very different from `UIWebView`.

From native to `UIWebView`

As discussed previously, there are significantly fewer options when communicating from the native side. Here in the figure 22 the two direct bridges (`jscore.sync` and `webview.eval`) are faster than the slower `http.websockets`, but it is again more render friendly with very low standard deviation. When looking up the same data as the sample series in figure 23, we can see that the samples have indeed very low deviation (green line on the bottom).

From native to `WKWebView`

With `WKWebView` in figure 24 it is surprising that unlike previously the `http.websockets` pausing is not lower than `webview.eval` (similar to the `jscore.sync` making the latter a better method overall).

Small vs Large Payload

The figure 25 compares two render pause graphs with different payloads. In the bottom graph it reminds us again that `xhrpost.async` (annotated with 1) is clearly better with large payloads as discussed above. However, with a smaller payload size the chart (top) looks completely different: `xhrpost.async` has been pushed to the middle. This is most likely due to the overhead involved in opening the HTTP connection for the message. The previously not very performant method `a.click` (annotated with 2) is also in the top 3, when it is 15th with a larger payload.

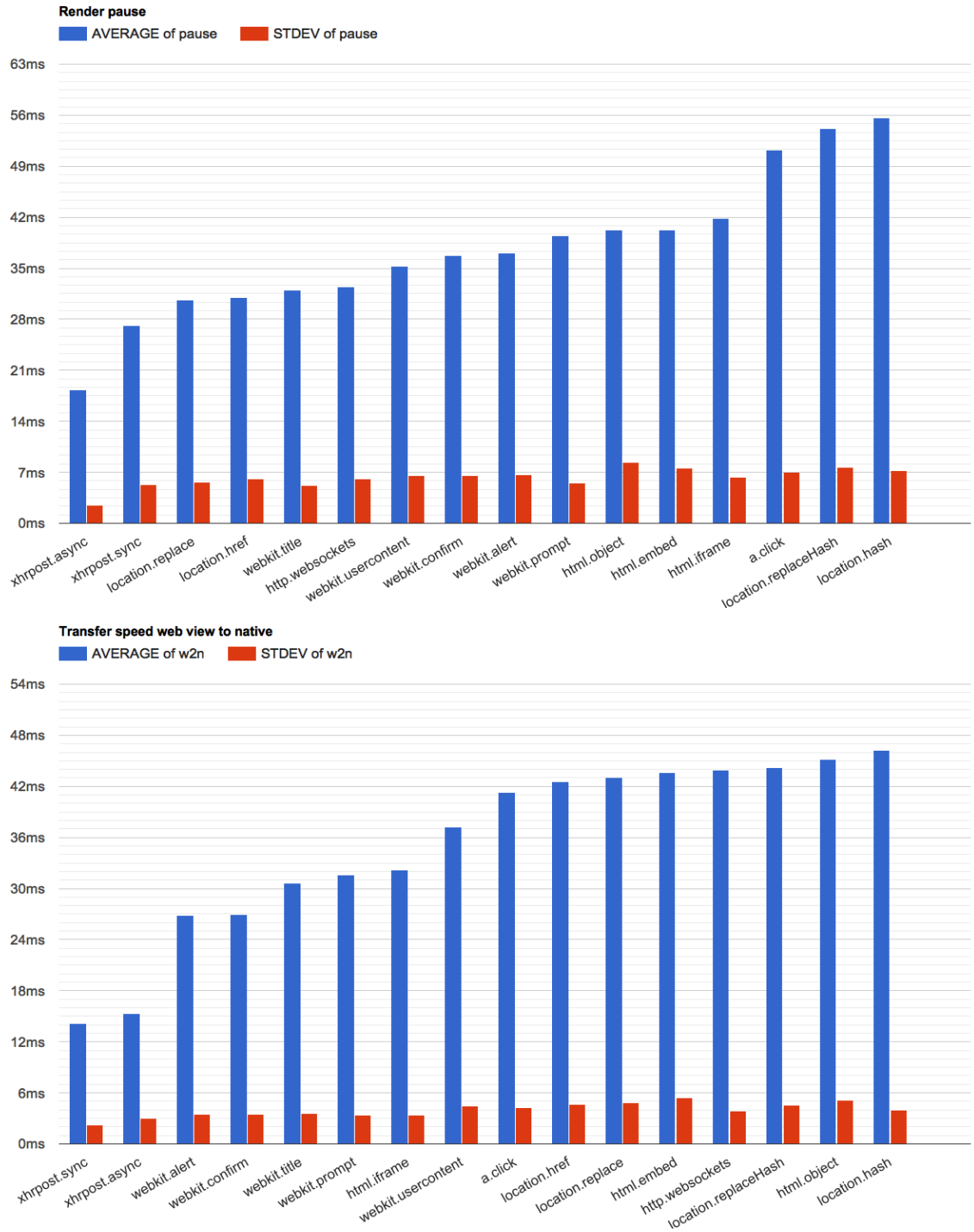


Figure 21: From `WKWebView` to native (iPad Air 2, large payload)

Some methods like `http.websockets` and especially `jscore.sync` (both underlined) are in the top positions with both payloads sizes, suggesting that those might be good candidates for overall good methods.

Figure 26 continues the same comparison with the transfer speeds. Here the `xhrpost` bridges perform completely differently: the fastest method, `xhrpost.async` (annotated) in the bottom chart becomes the slowest when the payload size is small (upper chart).

As noted previously the `jscore.sync` is in the top 4, but the `http.websockets` (both underlined) is significantly slower to transfer when the payload size is small.

Bridges `html.*` and `jscore.sync` vs `webkit.usercontent`

As mentioned before, the `html.*` method testing seemed a bit irrelevant, since all these methods trigger the page load in similar way. Figure 27 shows that these methods have differences of 10-20ms.

Device Differences

Initially we looked at the best performing bridges in figure 16 and since the performance of the bridges was similar, we selected iPad Air 2 as the device for our discussion here. Now, let's see how this fast device compares to the slowest device. Figure 28 shows comparison between iPad Air 2 (fast 64-bit device) and the slowest device of our test setup: the original iPad mini with 32-bit processor. The leftmost methods (annotated) in both graphs are more or less the same (like in figure 16, having only small differences in their order).

The device performance implications seems to be greater in the slower methods. There is significant standard deviation with the `prompt` bridge (annotated) in the upper graph. Figure 29 shows the corresponding sample series where the prompt transfer suffered significantly in the middle of the 100 samples. This same outlier behaviour can not be noticed from the other device tests series, indicating that a slow performing device is affecting the transfer with this method. The other methods, however, seem to have stable transfer performance.

6.3 Selecting the Best Bridges

After the previous discussion we still have not answered the most obvious question: which ones are the best native bridges? We have already understood that the answer is dependent on the usage properties that are application specific. Depending on its use, the same application can have plenty of native bridge calls or virtually none. The NativeBridgeBenchmark tool was built to enable the selection of the best performing i.e. most indistinguishable bridges for hybrid apps. The tool was released as an open source project so that

developers can run tests to find out what the best bridges are for their target hybrid app's requirements and selected devices. A single application can contain multiple bridges (even all of them, like the NativeBridgeBenchmark client app does) and each message can be transmitted with the most suitable bridge.

iOS is receiving updates, so some of the bridges may deprecate, especially with a major iOS release. Also, the majority of the native bridges are essentially "hacks" that are not intended for this kind of message passing. Therefore developers need to be able to re-run the tests from time-to-time and check if the choices they previously made are still valid. It might also be hard or impossible to consistently test native bridge performance in an isolated way in their own application.

There can also be alternative implementations of certain bridges. For example, `http.websockets` can be implemented with any web server library, but `CocoaHTTPServer` is used in the tool. The open source libraries that were "leading" in 2014 might not be updated or have later lost their momentum. Actually, while working on this tool, it was discovered that the "large" messages crashed the application because their payload size was too large for the `CocoaHTTPServer` to handle. This was fixed in a pull-request (patch) in November 2013 (NativeBridgeBenchmark uses this patched version), but the patch still hasn't been accepted to the project[67]. Currently it looks like the project is completely stale and none of the issues are being solved (but it is still widely used and works).

Another thing to consider are new device releases. Apple is regularly updating iOS devices with faster processing and more available memory. Some bridges that consume more memory (potentially `html.*` ones) that did not make sense in an old iPhone with 512 megabytes of RAM could make more sense in a device that has 4 gigabytes of RAM available.

The main learning in this work has been that render pausing is as important as transfer speed. Transfer speed has been recognized in the existing research, for example by Corral et al. [9]) and benchmarking projects, like Parparita's tool [31]), but render pausing had not yet been considered. The performance of some methods is heavily dependent on payload size, which impacts both transfer latency and render pausing. Although the payload size is usually small, there are cases when the message size will be larger. All of these factors impact indistinguishability for the app user. Low transfer latency is important for situations where the user expects immediate response (like audio feedback from tapping), while low render pausing matters for cases where the user interface is constantly manipulated (like in dragging the screen). An app developer should therefore select different methods for different use cases (and potentially even for different devices).

[67] CocoaHTTPServer Pull-request #94
https://github.com/robbiehanson/CocoaHTTPServer/pull/94

Considering that implementing all the native bridges in NativeBridgeBenchmark took weeks of full-time work for a senior developer who was already familiar with the subject, it is highly unlikely that application developers have as much time on their hands while making an app with deadlines and limited budgets. The test case presented in this chapter was run on 4 different devices that represent the major categories of iOS devices (iPad, iPhone, iPod Touch). There are some minor differences in performance among these devices. This makes it difficult to define an absolute ranking of the native bridges or to select a method that would be best overall. Instead of an absolute ranking, we will suggest the following criteria that combine the results from all devices for transfer latency and rendering pause separately.

To do this, we will pick the top performing bridges for both transfer and render pausing from the data. We take both their positions from the results from all of the four devices and use this as a score. For example, when sending messages from `UIWebview` to native the data looks like this:

Table 4: `UIWebView` to Native, small

Device	**Method**	**Pause avg**
iPad Air 2	http.websockets	19.1ms
iPad Air 2	jscore.sync	19.9ms
iPad Air 2	a.click	20.1ms
...	...	...
iPod Touch 6	http.websockets	19.8ms
iPod Touch 6	a.click	20.1ms
iPod Touch 6	jscore.sync	20.2ms
...	...	...
iPhone 6 Plus	a.click	20.0ms
iPhone 6 Plus	jscore.sync	20.2ms
iPhone 6 Plus	http.websockets	20.5ms
...	...	...
iPad Mini 1st	a.click	21.0ms
iPad Mini 1st	http.websockets	21.0ms
iPad Mini 1st	jscore.sync	22.7ms
...	...	...

The best performing bridges are almost the same for these four devices. Now, we take the top performing bridges for render pause and look up their positions in the data and use this as a score. Each method will get a score by adding together the method's ranking in each of the series of tests run for the four devices. For example, from the table above we would assign a score

of 7 for `a.click` (3+2+1+1) because it's the third least pausing bridge on iPad Air 2, then the second best on iPod Touch 6 and so on.

Table 5: `UIWebView` to Native, small, positions from all devices

Transfer	**Score**	**Pause**	**Score**
jscore.sync	4	http.websockets	7
confirm	12	a.click	7
prompt	13	jscore.sync	10
alert	13	prompt	25
http.websockets	21	xhr.sync	27
xhr.sync	21	confirm	28
a.click	34	alert	31

The method just described is used with our real data in Table where we selected the "top 4" bridges for each of the four devices. In this case, for the small payload from `UIWebView` to native, there were seven bridges in all. The lefthand side of the table, with the "Transfer" column and the next column "Score", shows the sum of all the rankings with the four devices used in the test. The `jscore.sync` method had a score of 4, meaning that it was the fastest on all four devices. The righthand side of the table 'Pause' shows the rankings of the methods in terms of render pausing. As previously noted, `a.click` and `http.websockets` have been render friendly and therefore their scores are lower (better) than jscore.sync's.

This table shows that `jscore.sync` and `http.websockets` are good choices for small payloads, while `a.click`, which performs well in terms of render pausing, is the slowest of these methods when considering transfer speed. If the payloads are small and sent every second and the target deployment devices are similar to the ones used in the test and development time is limited, a developer should be safe by choosing `jscore.sync` for all messages. If there is more time, then the developer could optimize the application by using `http.websockets` or `a.click` for some cases.

With a large payload the positions change, as can be seen in Table 6. The `xhrpost` methods, that did not qualify in the top 4 with a smaller payload, are now the best in all categories. The previously good `http.websockets` is not included in either top 4 listing.

Table 6: `UIWebView` to Native, large, positions from all devices

Transfer	**Score**	**Pause**	**Score**
xhrpost.sync	5	xhrpost.async	4
xhrpost.async	7	xhrpost.sync	10
html.iframe	16	jscore.sync	10
jscore.sync	17	confirm	22
confirm	18	html.iframe	41

Table 7: Native to `UIWebView`, small, positions from all devices

Transfer	**Score**	**Pause**	**Score**
jscore.sync	4	jscore.sync	8
webview.eval	9	webview.eval	8
http.websockets	11	http.websockets	8

Table 7 shows the results of tests in the other direction (Native to `UIWebView`) with a small payload. Here we see that all the methods are equal when it comes to pausing (all share the same score of 8), but the `jscore.sync` is the fastest. With a large payload in Table 8, `http.websockets` turns out to be the best in terms of transfer speed, but pauses more than `jscore.sync`.

Table 8: Native to `UIWebView`, large, positions from all devices

Transfer	**Score**	**Pause**	**Score**
http.websockets	4	jscore.sync	5
webview.eval	8	http.websockets	10
jscore.sync	12	webview.eval	10

This method of scoring bridges by their ranking is one way to answer the question "which ones are the best?". For a more definite answer, we would have to know a lot more about the target hybrid app. For this reason we will not create similar tables for `WKWebView` as these tables can be generated from the appendix or, preferably, by running the tool (and running it again when new iOS versions are released).

6.4 Comparison to Related Work

As mentioned previously, there is very little systematic comparison done previously. The LinkedIn engineering blog post from 2012 briefly shares their comparisons of `webview.eval`, `html.iframe` and `http.websockets`. Their app originally used `webview.eval`, but they found out that this caused

"locking" and "unpredictable behavior". They mention that their application was taking over 100ms to deliver a message and that some of the messages were lost. Over 100ms is credible: a device from that time (2012) can take over 100ms to deliver a large payload as can be seen from our test results. Also sending messages while interacting with the user interface caused the user interface to freeze. Again, from the test results we can see that all bridges will pause the renderer on that device for 40ms-230ms. In general they mention that switching over to `http.websockets` solved the issues, something we have also seen for example in table 8.

In the end the LinkedIn team settled with a hybrid model of using different bridges for sending messages for different directions. They conclude with "On average, WebSocket is faster, but practically negligibly so. However, it is far more consistent than either of the URL scheme [`html.iframe`] implementations which had widely varied timings. That, coupled with the asynchronous behavior, make WebSockets a win for many solutions." [15].

They published some comparison numbers of sending 100 messages, but did not mention the payload size or interval of the message sending. This makes it difficult to have a deeper comparison with our results.

Hybrid app development framework Trigger.io Forge's blog post "Why Trigger.io doesn't use PhoneGap – 5x faster native bridge" where they claimed the 5x performance difference with PhoneGap only applies to Android. On iOS they state that "PhoneGap uses a similar method, which explains why our performance is comparable in the benchmarking we did.". The test only tests small payload sending where they compare Trigger.io's bridge against PhoneGap's bridge with a very small interval. They did publish the test tool as an open source repository in GitHub, but running this requires their proprietary runtime, so it is only partially open sourced [68]. They also ignored the render pausing and since the claimed performance was mostly only on Android, it does not apply to our tests.

Parparita's *WebView Communication* test project was one of the most complete test suites before NativeBridgeBenchmark existed. NativeBridgeBenchmark includes all the methods from *WebView Communication* and also some additional methods. Since *WebView Communication* does not contain a HTTP server `http.websockets`, `xhrpost` and `xhrget` are missing. The non-obvious prompt, `alert` and `confirm` are missing for `UIWebView` and `webkit.confirm` for `WKWebView`. Additionally different HTML elements to trigger like `html.svgImg` and `html.embed` are not implemented.

The testing method is also different: all bridges are measured as round-trip time (a call needs to get a "pong" reply for the "ping"). The message payload and the sending interval are not configurable (always small payload and very fast interval). In the blog post Parparita gives some results as an example numbers from one device. Most of the results seem to be comparable,

[68] https://github.com/trigger-corp/Forge-vs-Cordova-Performance

even if the testing method differs completely. Since both the `UIWebView` and `WKWebView` are based on the open source WebKit project, Parparita has been able to inspect the internal implementation of the bridges and use that as a clue to what bridges to implement [31].

In the paper "Mobile multiplatform development: An experiment for performance analysis" Corral et al. compare performance (transfer time) between an app made with PhoneGap and a native application. In the comparison there are 8 cases that are timed in both apps. The cases include for example "Launch sound notification" and "Retrieve data from contact list" that bundle the native bridge and the task (SDK implementation to do the actual work) in the same test (time to response). Since they only measure against "PhoneGap" they are effectively only testing one bridge (possibly `html.iframe`) [9].

6.5 Future Work

The NativeBridgeBenchmark currently only implements the foundation of native bridge benchmarking. There are several new dimensions and features to add to the tool. Some of the bridging methods that allow synchronous messaging (where a reply is given immediately as the bridge is open). This could be useful for cases where a response from either side is expected to be available immediately, for example, the last known location or the current battery level. Synchronous bridges might also have some benefits for specific payload types or interval patterns. For example the `prompt` can be used as a synchronous bridge as it blocks the event loop until a message is returned:

```
var responseFromNative = prompt(messageToNative);
```

In the example above the native can respond with a message that gets assigned to `responseFromNative` after it has received, read and performed the actions described in `messageToNative`. Methods like `webview.eval` are naturally synchronous as they can evaluate any number of JavaScript lines in the web view and get the last evaluation result back as a string.

The NativeBridgeBenchmark measurement method (and the results) of transfer speed still apply even if the bridges were used synchronously. Most likely the total transfer time is shorter as the response can be sent right away and no separate call is needed. On the other hand, keeping the event loop occupied for longer time increases the rendering pause. It's likely that the synchronous methods benefit very specific payload sizes and/or interval patterns.

The rendering pause is detected with the `requestAnimationFrame` callback. This has a known side effect of never being called more than 60 times a second. To detect even pauses that take less than 16.667ms, a different callback (`setTimeout`) would be better.

Splitting the message sending in smaller pieces and measuring those

individually is worth looking into. By knowing the impact of 1) bridge initialization, 2) transferring the message and 3) parsing the message on the receiving side, will allow further optimizations. For example with the `xhrpost.async` method it looks like a lot of overhead is involved in opening a (non-interrupted) HTTP request for each message as this method is the fastest with the large payload, but much slower with the small payload.

Some bridges can support streaming processing of the message data. At least `http.websockets` web server receives message in chunks that can be processed before the full message is received. The data transfer speed could also be limited by the server so that the pausing impact would be minimal (but the transfer would slow down). Also using multiple WebSocket connections at the same time might have interesting performance implications.

Parparita's analysis of the WebKit source code are interesting. Following the changes in the project can predict the upcoming changes before a new version of iOS that bundles the web view engines from the WebKit project is released. Analyzing the source while creating the native bridge method is beneficial for the best possible implementation.

Additional metrics like CPU and memory usage and energy (battery) impact are some of the obvious features to be added in the tool. Also different optimizations such as sending multiple messages at once (when the sending rate is high) or delaying sending if the event loop is busy are worth investigating. Alternative implementations for different bridges, like a reusing the HTML elements (like `html.img`) and not creating a new one for each message is another obvious optimization. This re-use method was first included in the current tests, but was excluded because a further work is needed to ensure that all messages get sent when the sending interval is high. And since NativeBridgeBenchmark was designed to be client agnostic, writing clients for Android and other platforms is naturally in the possible future roadmap.

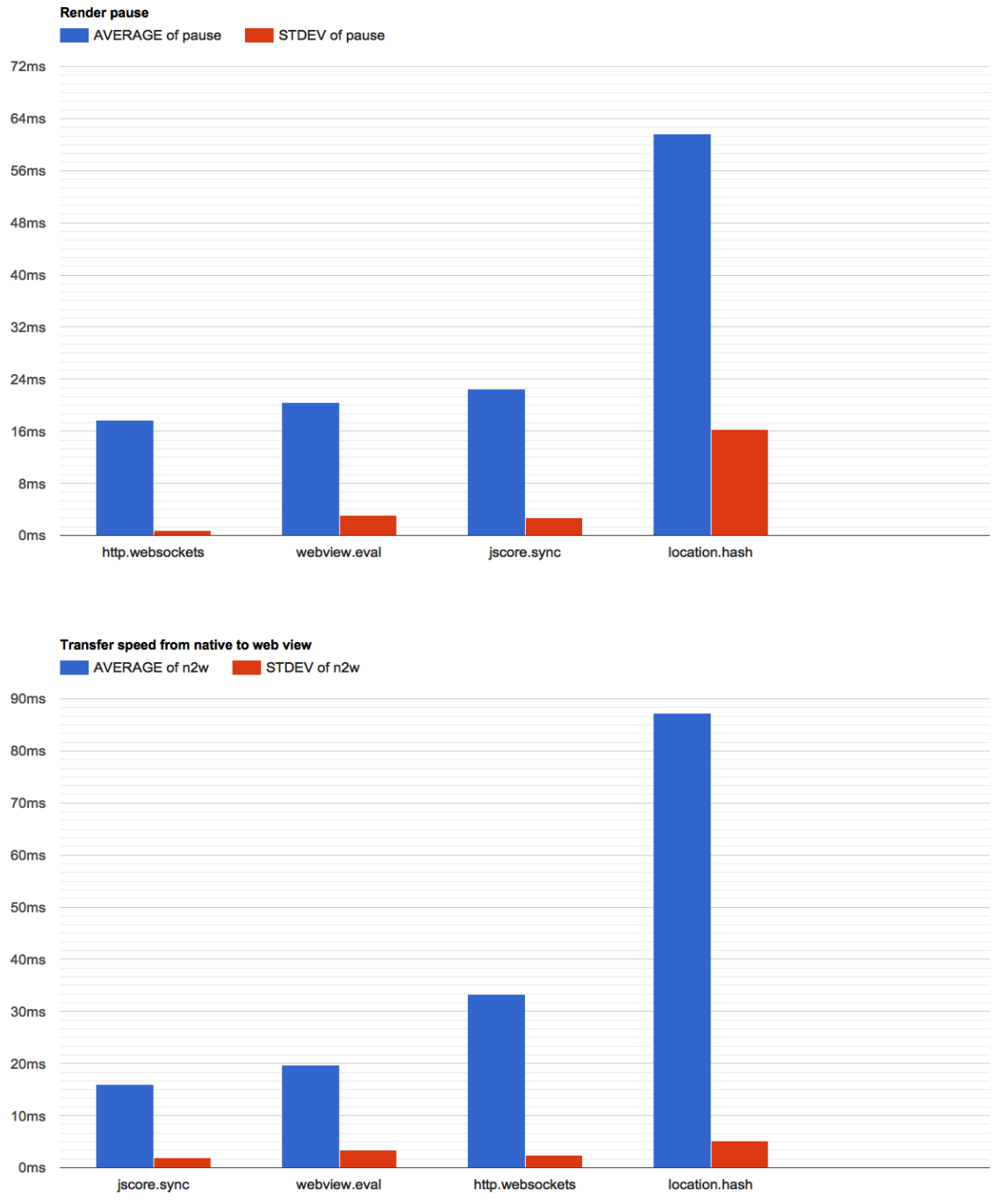


Figure 22: From native to `UIWebView` (iPad Air 2, large payload)

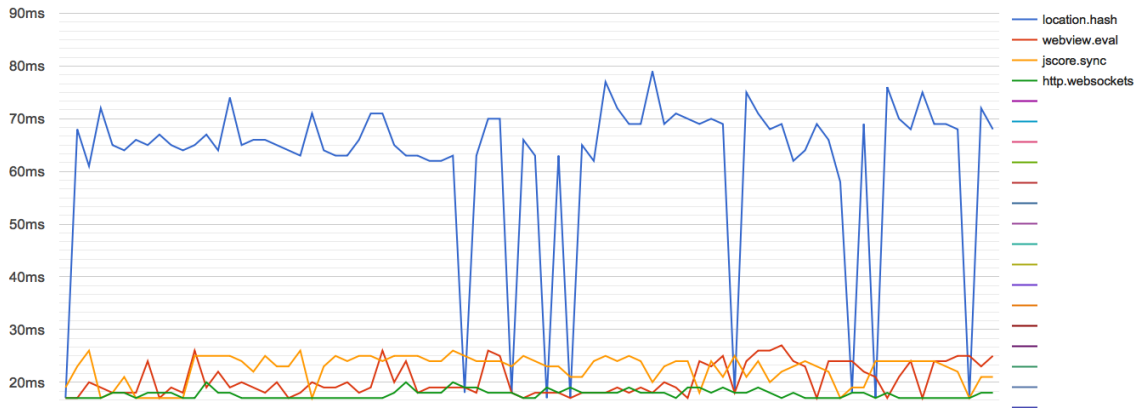


Figure 23: Consistent pause behaviour of `http.websockets` (green) in native to `UIWebView` (iPad Air 2, large payload)

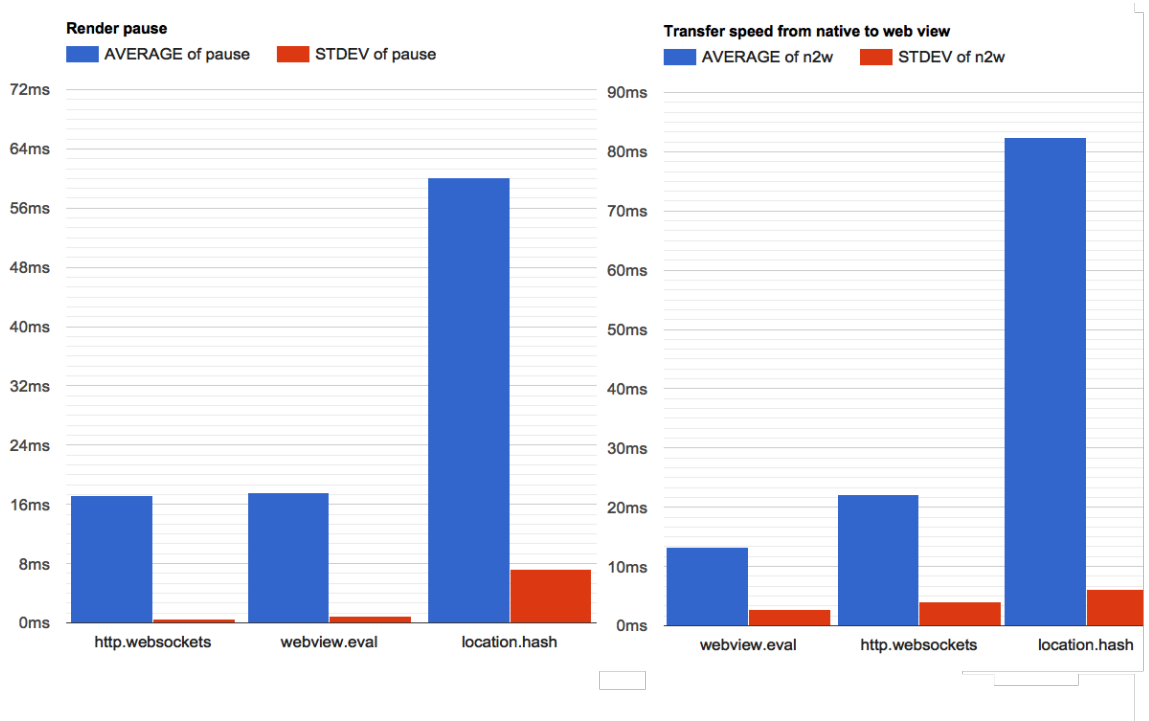


Figure 24: From native to `WKWebView` (iPad Air 2, large payload)

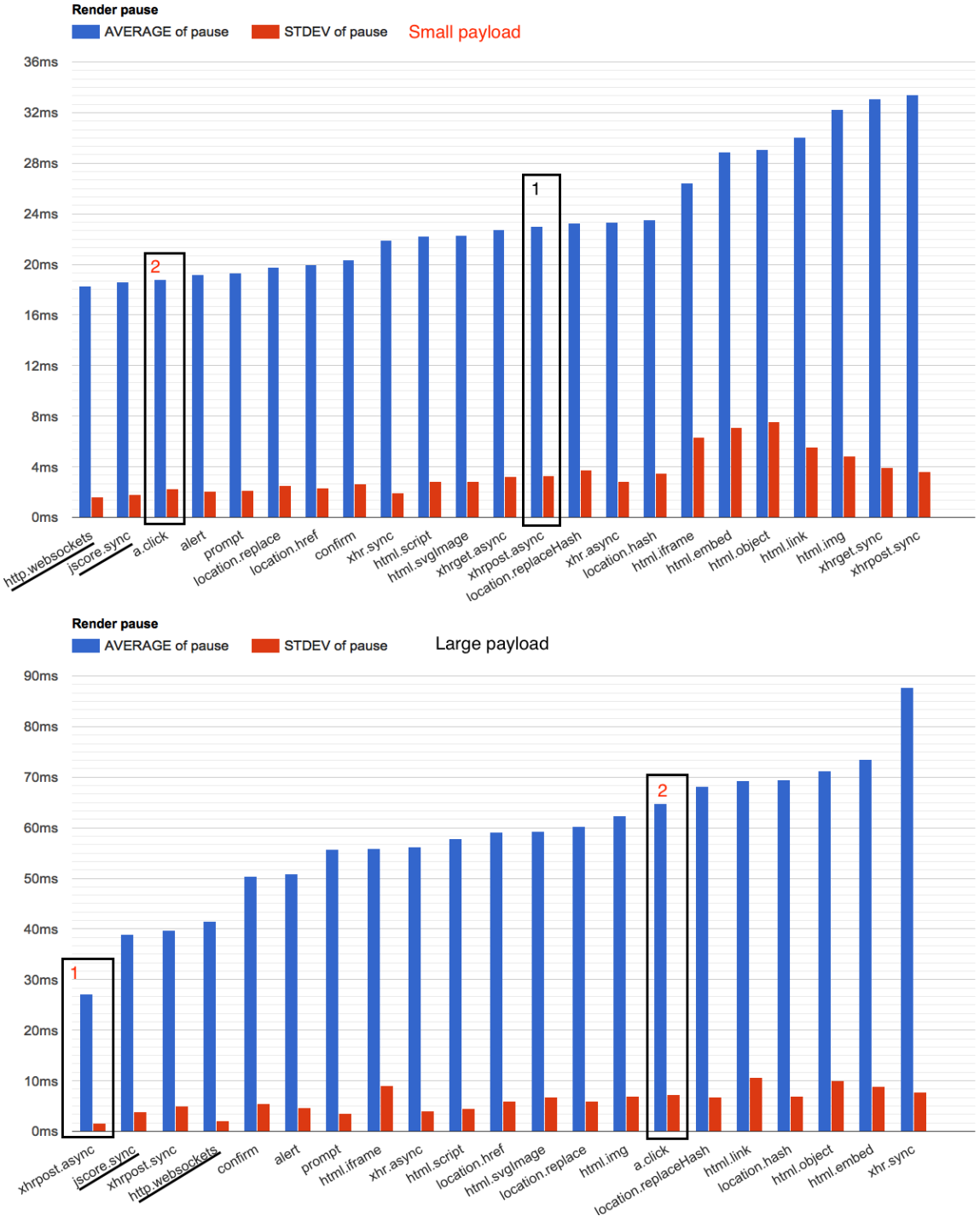


Figure 25: Small (top) vs large (bottom) pauses (iPad Air 2, `UIWebView` to native)

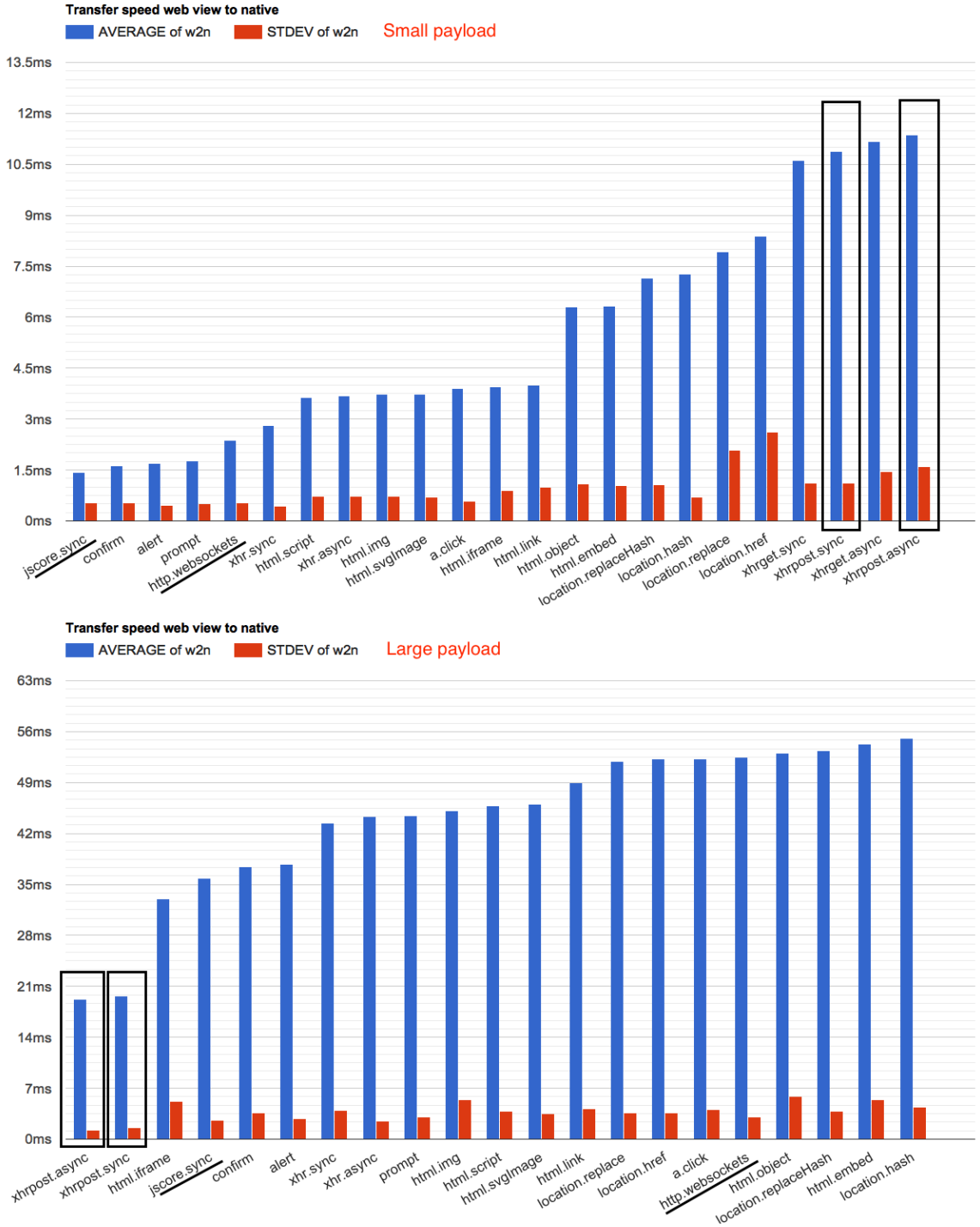


Figure 26: Small (top) vs large (bottom) transfer (iPad Air 2, `UIWebView` to native)

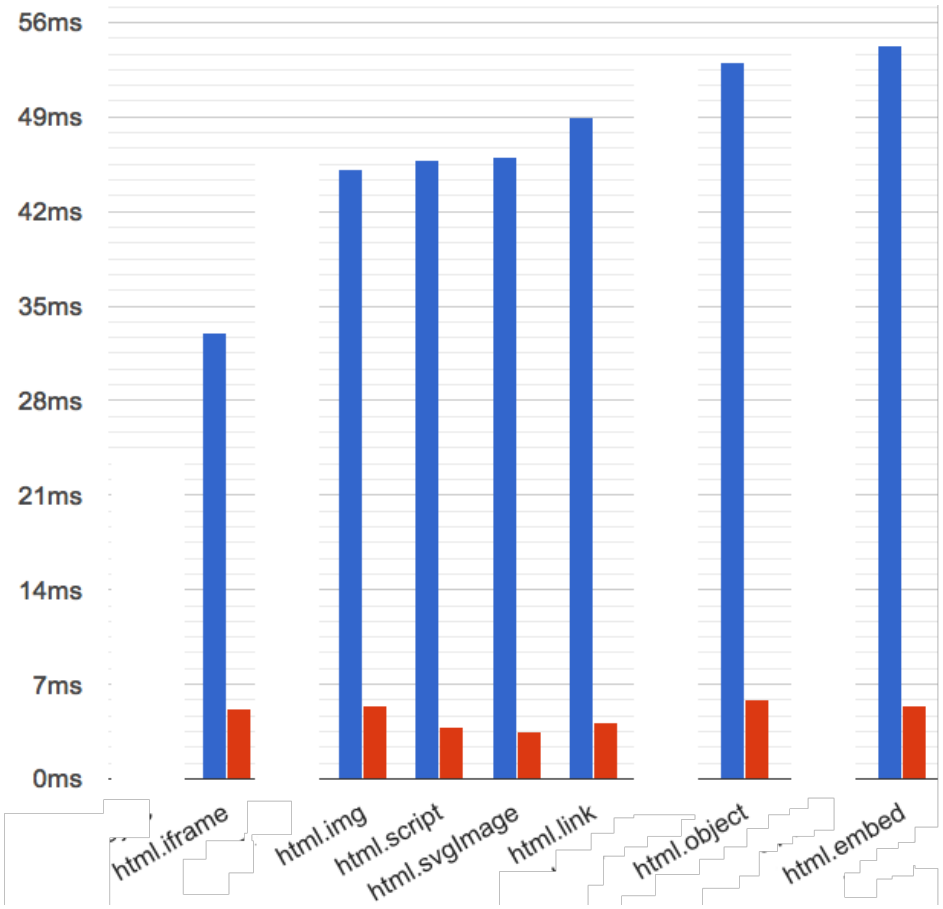


Figure 27: Comparison of `html.*` methods' transfer speed showing differences of 10-20ms.

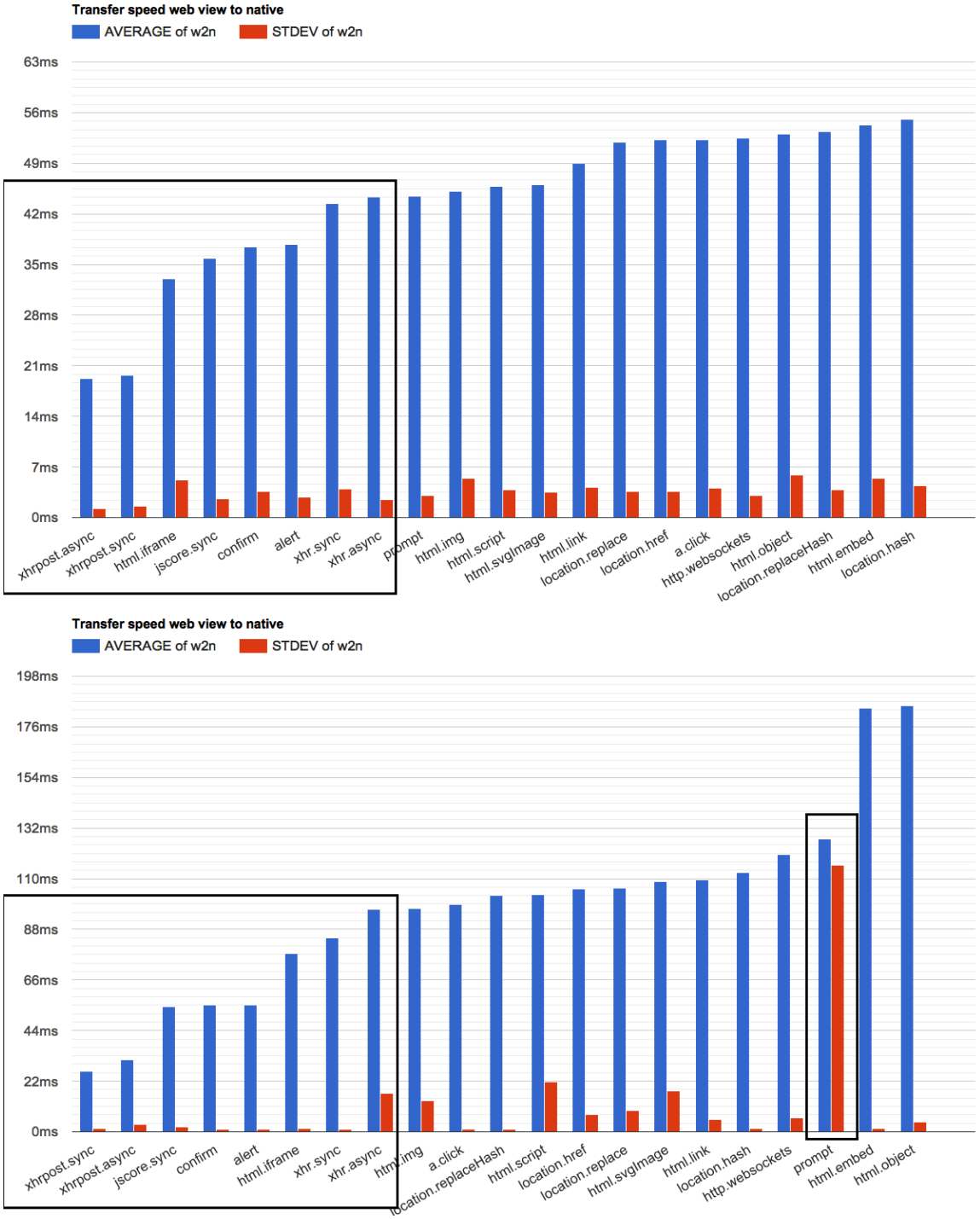


Figure 28: iPad Mini (top) vs iPad Air 2 (bottom) transfer (Large, `UIWebView` to native)

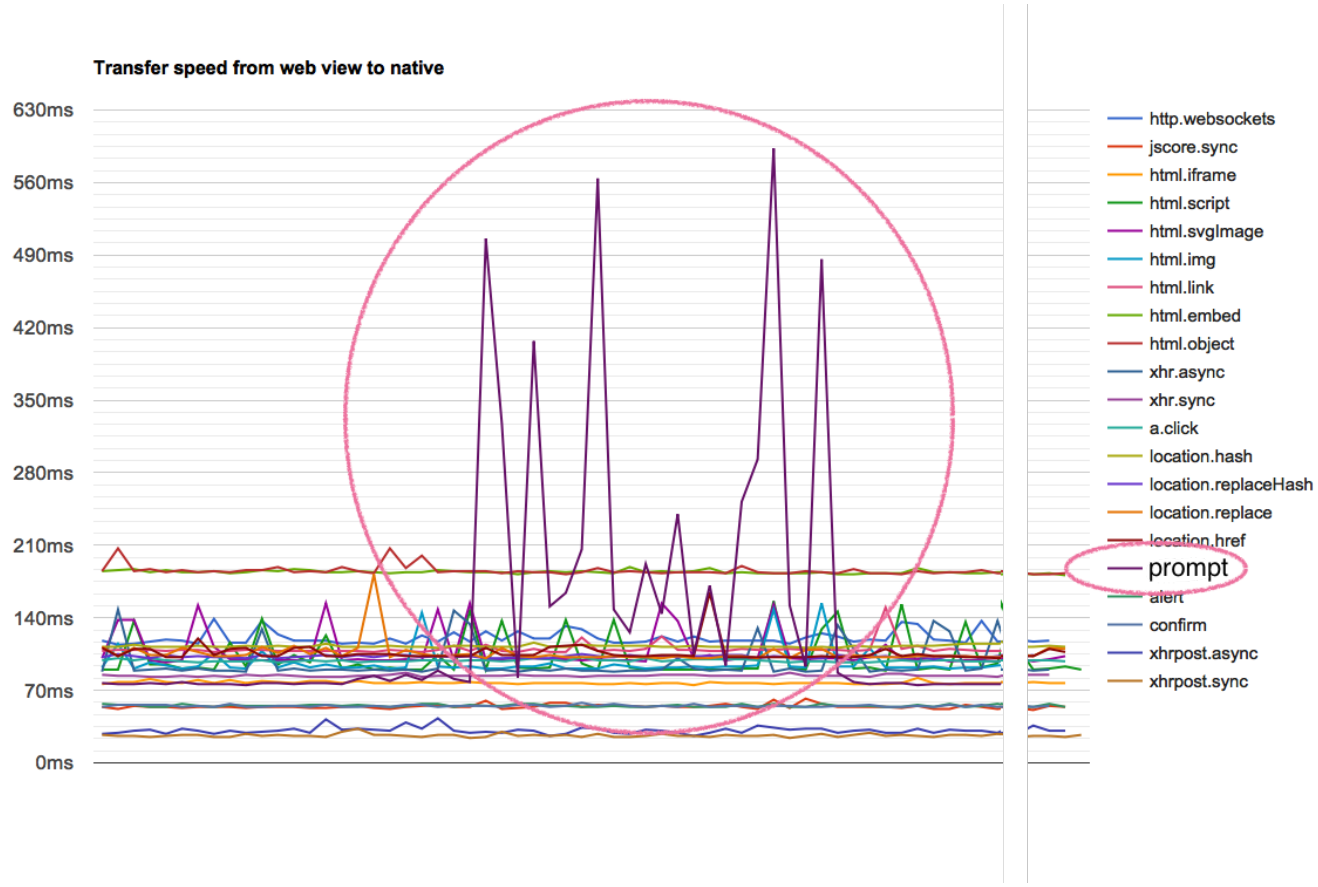


Figure 29: Variance in the `prompt` with old iPad mini (Large, `UIWebView` to native)

7 Conclusions

When starting a new mobile app project, the developers need to select the technologies they will use to build the app. An HTML5 hybrid app is a popular alternative for development speed, re-usability and skill-set reasons. If HTML5 hybrid is selected, its indistinguishability from a native app is one of the most important factors: how is the app going to compare to competition (possibly native) and is it performant enough for the users (does the application *feel* good when compared to other apps). In order to be indistinguishable, an app needs to have all the native device features that its native counterpart would have. To bring these features into the limited web view, a native bridge is needed. There are two different bridges to build in an app. The more common one is the bridge from the web view to the native code when a device capability needs to be accessed (e.g. when a user touches a button and a sound needs to be played). The other bridge, from the native code to the web view is needed when the web view is expecting some information from the device (such as a new GPS location update). On iOS there are limited possibilities for native bridging mechanisms (most likely because Apple wants the apps to have iOS specific features and a good user experience). Apple does allow these apps to be published in the App Store, as long as they follow Apple's design guidelines and do not look like mobile web pages.

This lack of official methods has lead to a search for various "hacks" that can be used to fill this gap. A native bridge is any method that can be used to pass a message between these two contexts. Most of these methods utilize some function that was not originally intended for this kind of message passing (like displaying an image in the web page). Due to this reason, the internal implementations of these components might not be as suitable for message passing as they are for their original intentions.

At the beginning of this work we set three main research questions. The first one was "How can the performance of a native bridge be systematically measured?". As part of this thesis a new open sourced tool, NativeBridgeBenchmark, was developed to give an executable answer to this question. The tool lays a foundation for the developer community to test and share findings of different native bridges' performance in a common format. With the tool, developers can run the same tests on multiple devices after a new iOS version or device is released. The performance numbers that the tool provides can show if performance has degraded or improved.

The second and third questions were "What are the properties that affect the performance characteristics of a native bridge method?" and "What kind of performance degradation can be noticed by the user?". Existing work on this subject focuses solely on the transfer speed of the message. While exploring this subject, it was discovered that the potential performance penalty comes from the single threaded JavaScript engine running inside the

web view that hosts the hybrid app's logic and the single threaded native UI thread where the web view is running. When these threads get blocked during a message transmission, the UI slows or locks down making the app look "choppy" or unresponsive. It was found that when sending a message with a native bridge the user interface slows down (pauses), but that the pausing is not related to the transfer speed of the native bridge. For example, some bridge was slower to transfer a message, but the user interface stayed more responsive while using it.

In order to stay responsive the user interface needs to be updated every 16.667ms (60 times a second). The existing research shows that, depending on the user's interaction, an added delay of 3ms to 20ms is noticeable both visually and auditorily. Transfer speed and rendering pause are the two independent properties that affect a native bridge's performance. Rendering pause causes the user interface to "lag", for example when the user is dragging an object on the screen. Pausing in this type of interaction for less than 11ms is noticeable by various research. The other property, transfer speed, adds latency to the actions performed. For example, when the user taps the screen and expects a response like a sound played (a capability that is not available in the web view), it is noticeable in around 20ms. The size of the message that is transmitted has the most impact on both of these properties. The interval of how often messages are sent has some additional effects, but it was intentionally left out of the scope of this work.

In the tool the testing is separated in "to web view" and "to native" directions so that the individual performance of each direction can be measured. With both directions the transfer delay and rendering pause are measured. An adjustable message payload size and sending interval allow for testing different kinds of messages and sending patterns. On iOS there are two different web view engines suitable for chromeless embedding. The tool supports both web views and is even extendable for additional web view engines. New native bridges can be added to the tool when discovered or as technology advances. The tool also allows users to implement a bridge in multiple different ways and to compare the implementations.

There were also additional research questions set for this work. The first one was "Which techniques can be used as a native bridge?". The main categories are request, connection, direct access and observe bridges. In request based bridges a network request is attempted from the web view, but terminated (and thus bridged) by the native side. In connection based bridges a real network connection is made (instead of interrupting) to the native side that has an HTTP server running. In direct access bridges the two sides are directly connected, for example, a method is exposed to the web view or the native executes JavaScript directly in the web view's JavaScript runtime. Observe bridges indirectly notice changes made in the other side. For example, when the web view changes the page's title, native will get notified and can then read the message from the title.

This work identified multiple different bridges from existing work and also discovered a few new ones. All the bridges discovered are implemented in the tool as concrete usage examples. The total number of identified bridges is the largest currently known set of bridges: 23 bridges for `UIWebView` to communicate with native and 4 bridges for native to communicate back. For `WKWebView` 16 bridges were identified for calling native and 3 bridges for the other way around.

The second additional question was "What is the typical usage pattern of the native bridge in a typical hybrid application?". Depending on the application, the usage of the native bridge and the size of the messages vary greatly. Some applications can be used for extended periods of time without any messages and some are heavily dependent on native bridge messages. The answer to this question lies within the second main research question about the properties affecting the performance characteristics of a native bridge method. All of the properties (message size, sending frequency and direction) come from application's features and its intended use.

Finally, the last and most obvious additional question was "What are the best performing native bridges?". The application usage pattern and the device used were identified as the required parameters for answering this question. This work includes a sample test result from the NativeBridgeBenchmark that is explored in detail to set out an example of how a developer would answer the question for her own application. The walkthrough of this, including sample results, shows that while it is possible to find some trends (`jscore.sync` is performant, `http.websockets` is slow, but renderer friendly) the payload size can change the results completely (`xhrpost` is the fastest or the slowest). Some methods can be assumed to be good choices under some conditions, while others should be used only if the payload size is known. If the app message payload sizes differ, then multiple different native bridges should be used simultaneously for the best performance. The majority of the bridges have significant performance differences and those are noticeable by the users.

As each major or minor release of iOS can change what can be used as a native bridge and how it performs, it is important to have a test suite that can be updated and that is decoupled from the actual apps to provide "a clean room" for testing. In addition to the tool itself, the most important contributions of this work are the testing method it implements, which takes into consideration the significance of measuring render pause in addition to transfer speed, and the thorough list of possible native bridges that was compiled (along with their example implementations).

Acknowledgements

I don't think that I would have started working on this project in 2014 without the love of my life, Krista Kauppinen. Not only did she get me started but she also helped me to push this across the finish line. The next person I'd like to thank is Matti Luukkainen, with whom I share many memories as my mentor, colleague, subordinate and finally as the supervisor of this work. Naturally Antti-Pekka Tuovinen's comments were the most helpful and I'm deeply grateful for those and all the time he spent with this work. I'd also like to thank Jaakko Kurhila for the endless discussions and trust he given me over the years.

I'm the last one to finish my studies (2003-2016) from the original *g-piiri* whose all members have already graduated a long time before me. Thank you guys for the support back in the days: Pirkka Esko, Tia Honkanen, Eero Pailinna, Petrus Repo and Arto Vihavainen (in alphabetical order). At that time Pekka Abrahamsson enabled a lot of things to happen, although I couldn't manage to motivate myself enough to finish the project. Lastly, I'd like to thank my parents Kauko and Ritva for giving me space to find my own ways in life.

References

[1] Adelstein, Bernard D., Begault, Durand R., Anderson, Mark R., and Wenzel, Elizabeth M.: *Sensitivity to Haptic-audio Asynchrony.* In *Proceedings of the 5th International Conference on Multimodal Interfaces*, ICMI '03. ACM, 2003.

[2] Ali, T., Saquib, M., Sengupta, C., and Lee, Yuan Kang: *Modeling of User-Perceived Web-Browsing Performance over a WLAN/3G Inter-Working Environment.* In *Communications, 2009. ICC '09. IEEE International Conference on*, 2009.

[3] Anderson, Glen, Doherty, Rina, and Ganapathy, Subhashini: *User Perception of Touch Screen Latency.* In *Design, User Experience, and Usability. Theory, Methods, Tools and Practice*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011.

[4] Apple Inc: *iOS Developer Library.* Apple Developer - iOS Developer Library, 2015. `https://developer.apple.com/library/ios/navigation/`.

[5] Baker, Matt: *Introducing the Rendering Frames Timeline.* WebKit Project Blog `https://webkit.org/blog/3996/introducing-the-rendering-frames-timeline/`, September 8, 2015.

[6] Björkskog, Christoffer, Jacucci, Giulio, Lorentin, Bruno, and Gamberini, Luciano: *Mobile implementation of a web 3D carousel with touch input.* International Conference on Human-Computer Interaction with Mobile Devices and Services - MobileHCI '09, 2009.

[7] Canalys Inc: *Top iOS and Android apps largely absent on Windows Phone and BlackBerry 10.* Press release, May 23, 2013.

[8] Charland, Andre and LeRoux, Brian: *Mobile Application Development: Web vs. Native.* ACM Queue, April 2011.

[9] Corral, Luis, Sillitti, Alberto, and Succi, Giancarlo: *Mobile multiplatform development: An experiment for performance analysis.* In *The 9th International Conference on Mobile Web Information Systems (MobiWIS).* Elsevier, 2012.

[10] David, Turner: *Introducing Android 1.5 NDK, Release 1.* Android Developers Blog, June 25, 2009. `http://android-developers.blogspot.fi/2009/06/introducing-android-15-ndk-release-1.html`.

[11] Deber, Jonathan, Jota, Ricardo, Forlines, Clifton, and Wigdor, Daniel: *How much faster is fast enough?: User perception of latency & latency improvements in direct and indirect touch.* In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15. ACM, 2015.

[12] Delwadia, Vipul, Marshall, Stuart, and Welch, Ian: *The Effect of User Interface Delay in Thin Client Mobile Games.* In *Proceedings of the Eleventh Australasian Conference on User Interface - Volume 106*, AUIC '10, 2010.

[13] Dunn, Connor: *Why Trigger.io doesn't use PhoneGap - 5x faster native bridge.* Trigger.io Cross Platform App Dev Blog `http://trigger.io/cross-platform-application-development-blog/2012/02/24/why-trigger-io-doesnt-use-phonegap-5x-faster-native-bridge/`, 2012.

[14] Dunn, Jonathan: *Under the hood: Rebuilding Facebook for iOS.* Notes by Facebook Engineering, August 23, 2012. `https://www.facebook.com/notes/facebook-engineering/under-the-hood-rebuilding-facebook-for-ios/10151036091753920`.

[15] Gandhi, Ashit: *LinkedIn for iPad: The Native/Web Messaging Bridge and WebSockets.* LinkedIn Engineering Blog `http://engineering.linkedin.com/mobile/linkedin-ipad-nativeweb-messaging-bridge-and-websockets`, 2012.

[16] Google Inc: *Android API Reference.* Android Developer Portal, 2015. `http://developer.android.com/reference/packages.html`.

[17] IBM Inc: *IBM Worklight Foundation V6.2.0 documentation.* IBM Knowledge Center, 2015.

[18] J, Agnew and J, Thornton: *Just noticeable and objectionable group delays in digital hearing aids.* Journal of the American Academy of Audiology, 2000.

[19] Jota, Ricardo, Ng, Albert, Dietz, Paul, and Wigdor, Daniel: *How Fast is Fast Enough?: A Study of the Effects of Latency in Direct-touch Pointing Tasks.* In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13. ACM, 2013.

[20] Khan, Faiz, Foley-Bourgon, Vincent, Kathrotia, Sujay, Lavoie, Erick, and Hendren, Laurie: *Using JavaScript and WebCL for Numerical Computations: A Comparative Study of Native and Web Technologies.* In

Proceedings of the 10th ACM Symposium on Dynamic Languages, DLS '14. ACM, 2014.

[21] Khoo, Yit Phang, Hicks, Michael, Foster, Jeffrey S., and Sazawal, Vibha: *Directing javascript with arrows.* In *Proceedings of the 5th Symposium on Dynamic Languages*, DLS '09. ACM, 2009.

[22] Lee, Youna, Rho, Seungmin, Hwang, Jae In, Ko, Heedong, and Kim, Junho: *osggap: Scene graph library for mobile based on hybrid web app framework (demo).* In *SIGGRAPH Asia 2013 Symposium on Mobile Graphics and Interactive Applications*. ACM, 2013.

[23] Liu, Yepang, Xu, Chang, and Cheung, Shing Chi: *Characterizing and Detecting Performance Bugs for Smartphone Applications.* In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014. ACM, 2014.

[24] Mallik, Arindam, Cosgrove, Jack, Dick, Robert P., Memik, Gokhan, and Dinda, Peter: *PICSEL: Measuring User-perceived Performance to Control Dynamic Frequency Scaling.* In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII. ACM, 2008.

[25] Martinsen, Jan Kasper, Grahn, Håkan, and Isberg, Anders: *The Effects of Parameter Tuning in Software Thread-Level Speculation in JavaScript Engines.* ACM Trans. Archit. Code Optim., 2015.

[26] Miller, Robert B.: *Response Time in Man-computer Conversational Transactions.* In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I). ACM, 1968.

[27] Nadkarni, Adwait, Tendulkar, Vasant, and Enck, William: *NativeWrap: Ad Hoc Smartphone Application Creation for End Users.* In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, WiSec '14. ACM, 2014.

[28] Ng, Albert, Annett, Michelle, Dietz, Paul, Gupta, Anoop, and Bischof, Walter F.: *In the blink of an eye: Investigating latency perception during stylus interaction.* In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14. ACM, 2014.

[29] Ng, Albert, Lepinski, Julian, Wigdor, Daniel, Sanders, Steven, and Dietz, Paul: *Designing for Low-latency Direct-touch Input.* In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12. ACM, 2012.

[30] Oh, Hyeong Seok, Kim, Beom Jun, Choi, Hyung Kyu, and Moon, Soo Mook: *Evaluation of Android Dalvik Virtual Machine.* In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12. ACM, 2012.

[31] Parparita, Mihai: *A faster UIWebView communication.* Personal blog `http://blog.persistent.info/2013/10/a-faster-uiwebview-communication.html`, 2013.

[32] Potter, Mary C, Wyble, Brad, Hagmann, Carl Erick, and McCourt, Emily S: *Detecting meaning in RSVP at 13 ms per picture.* Journal of Attention, perception & psychophysics, 2014.

[33] Puder, Arno, Tillmann, Nikolai, and Moskal, Michał: *Exposing Native Device APIs to Web Apps.* In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, MOBILESoft 2014. ACM, 2014.

[34] Quinn, Philip, Malacria, Sylvain, and Cockburn, Andy: *Touch Scrolling Transfer Functions.* In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13. ACM, 2013.

[35] Regan, Matthew J. P., Miller, Gavin S. P., Rubin, Steven M., and Kogelnik, Chris: *A Real-time Low-latency Hardware Light-field Renderer.* In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99. ACM Press/Addison-Wesley Publishing Co., 1999.

[36] Steve, Jobs: *Interview by Walt Mossberg and Kara Swisher.* D8'2010: AllthingsD.com D8 Conference 2010, June 1, 2010.

[37] Verkasalo, Hannu: *Empirical observations on the emergence of mobile multimedia services and applications in the u.s. and europe.* In *Proceedings of the 5th International Conference on Mobile and Ubiquitous Multimedia*, MUM '06. ACM, 2006.

[38] VisionMobile Inc: *Developer Economics Q3 2014: State of the Developer Nation.* Industry report, July, 2014.

[39] Xanthopoulos, Spyros and Xinogalos, Stelios: *A Comparative Analysis of Cross-platform Development Approaches for Mobile Applications.* In *Proceedings of the 6th Balkan Conference in Informatics*, BCI '13. ACM, 2013.

[40] Xu, Jingxi and Wah, Benjamin W.: *Exploiting just-noticeable difference of delays for improving quality of experience in video conferencing.* In *Proceedings of the 4th ACM Multimedia Systems Conference*, MMSys '13. ACM, 2013.

[41] Yang, Shengqian, Yan, Dacong, and Rountev, A.: *Testing for poor responsiveness in android applications.* In *Engineering of Mobile-Enabled Systems (MOBS), 2013 1st International Workshop on the*, May 2013.

A Test Results

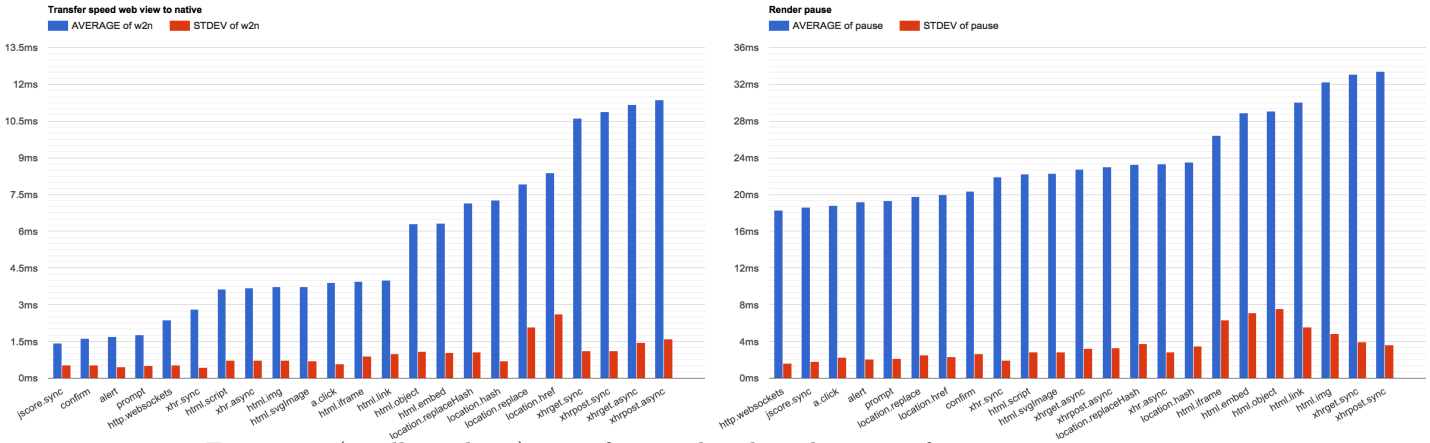


Figure 30: (small, ipadair2) Transfer speed and render pause from `UIWebView` to native

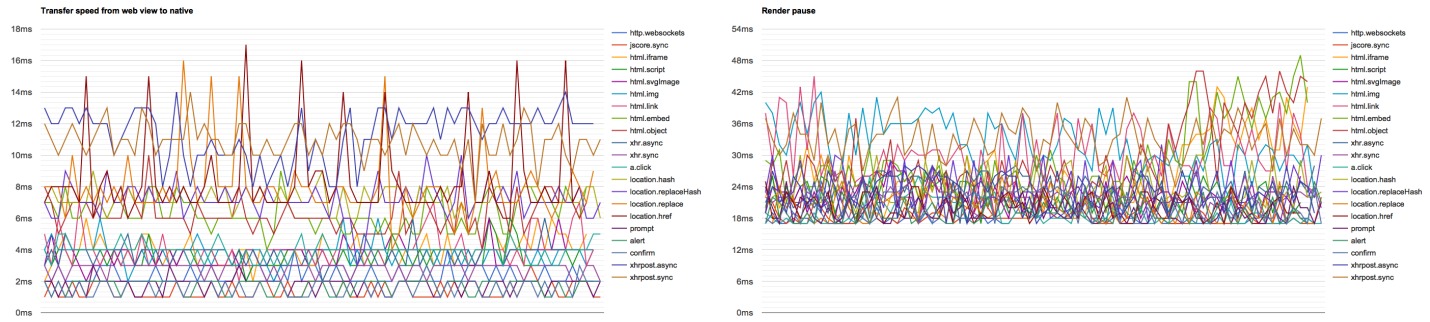


Figure 31: (small, ipadair2) Sample series from `UIWebView` to native

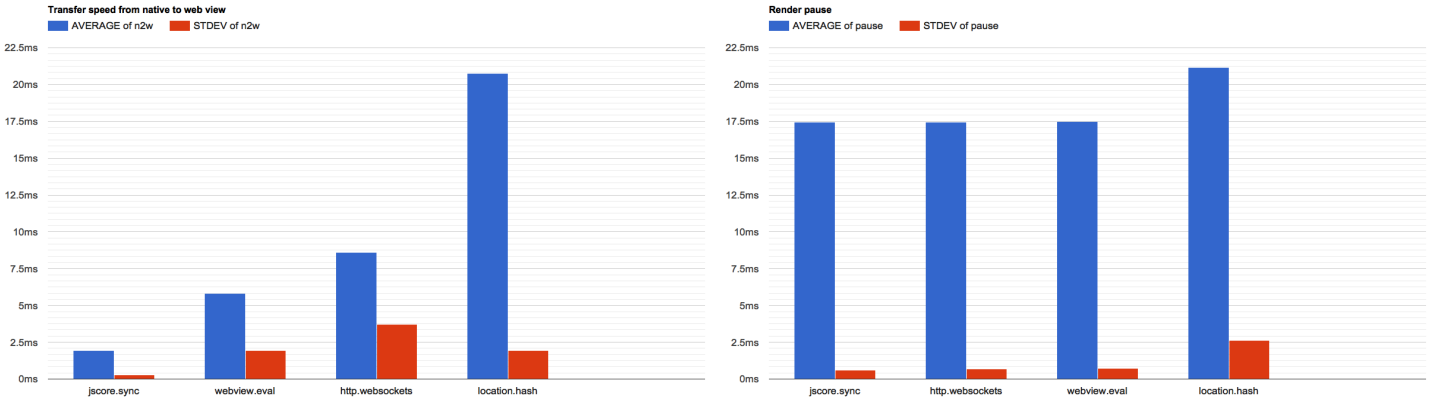


Figure 32: (small, ipadair2) Transfer speed and render pause from native to `UIWebView`

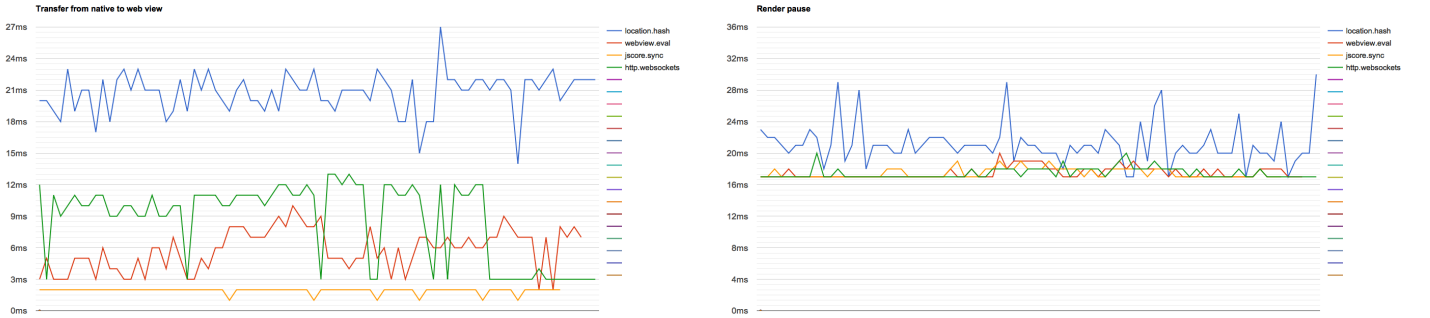


Figure 33: (small, ipadair2) Sample series from native to `UIWebView`

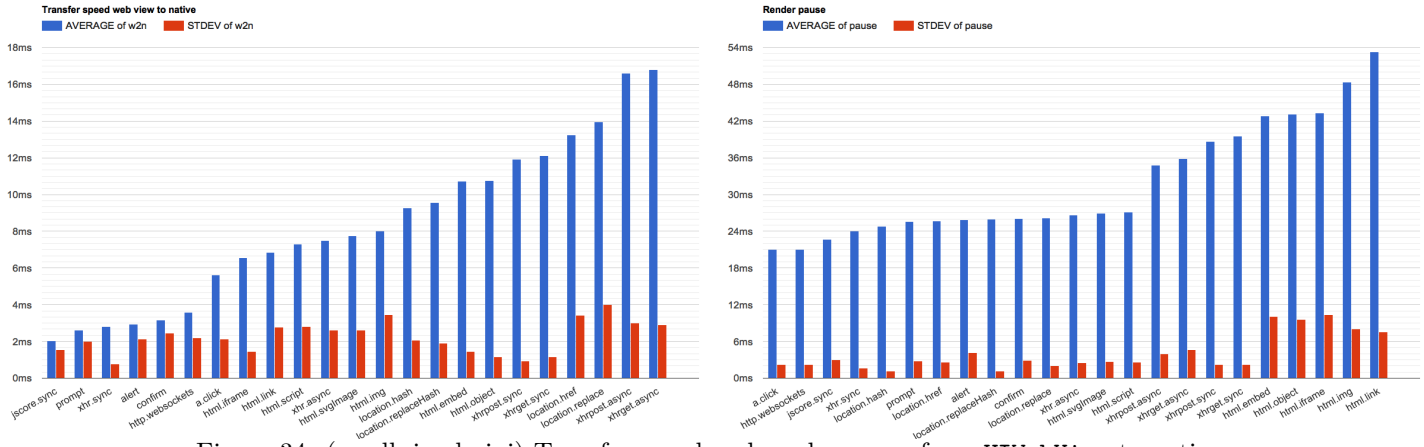


Figure 34: (small, ipadmini) Transfer speed and render pause from `UIWebView` to native

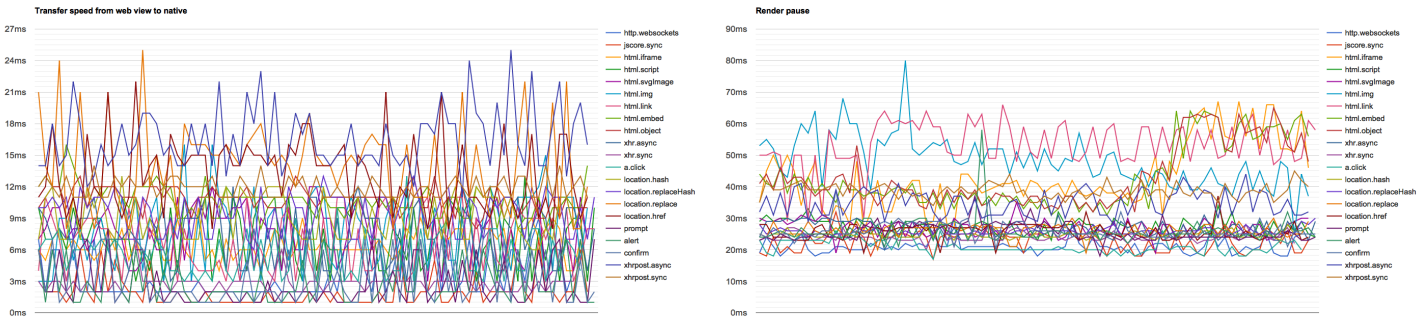


Figure 35: (small, ipadmini) Sample series from `UIWebView` to native

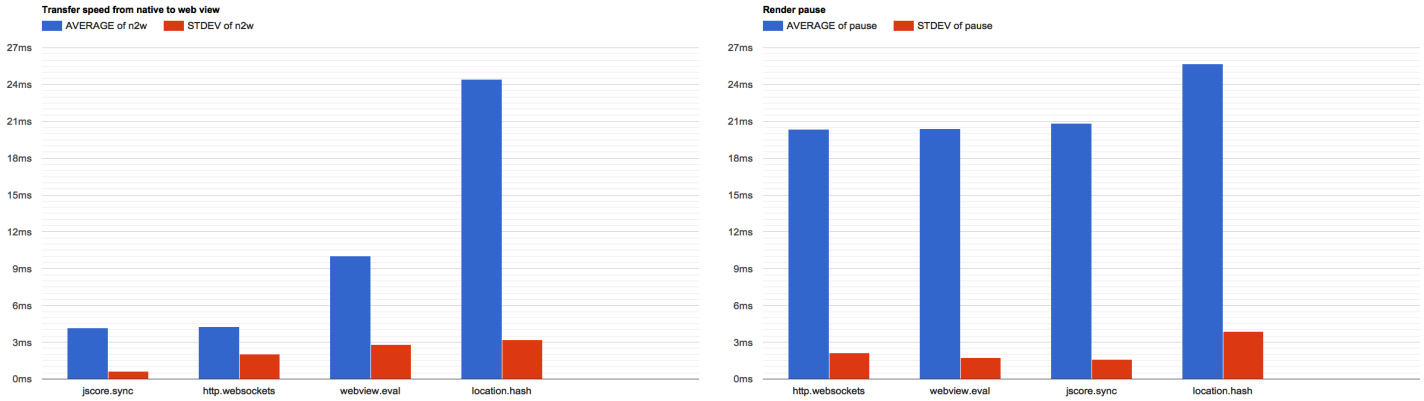


Figure 36: (small, ipadmini) Transfer speed and render pause from native to `UIWebView`

Figure 37: (small, ipadmini) Sample series from native to `UIWebView`

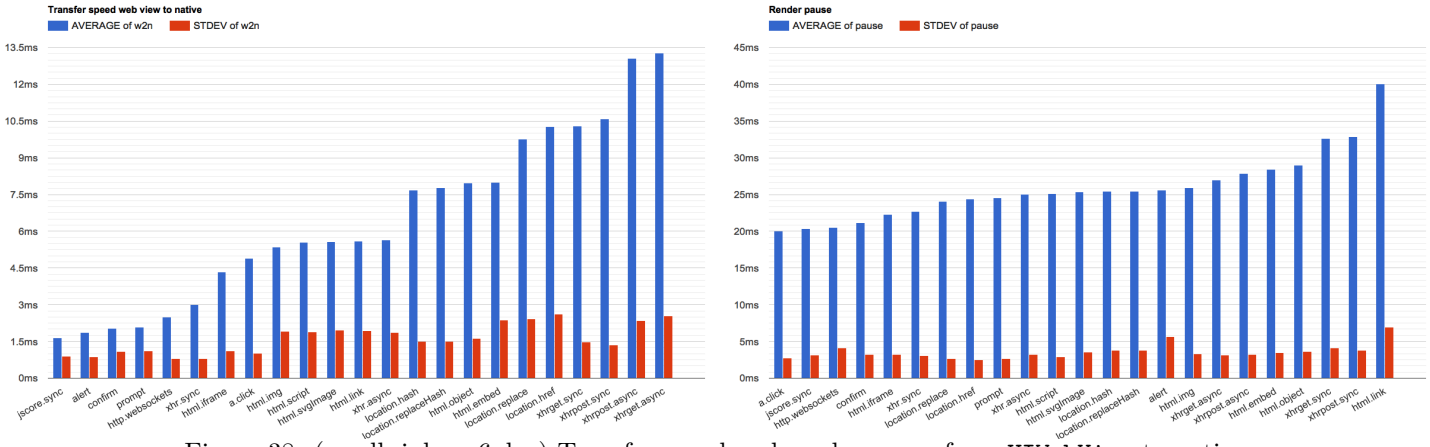


Figure 38: (small, iphone6plus) Transfer speed and render pause from `UIWebView` to native

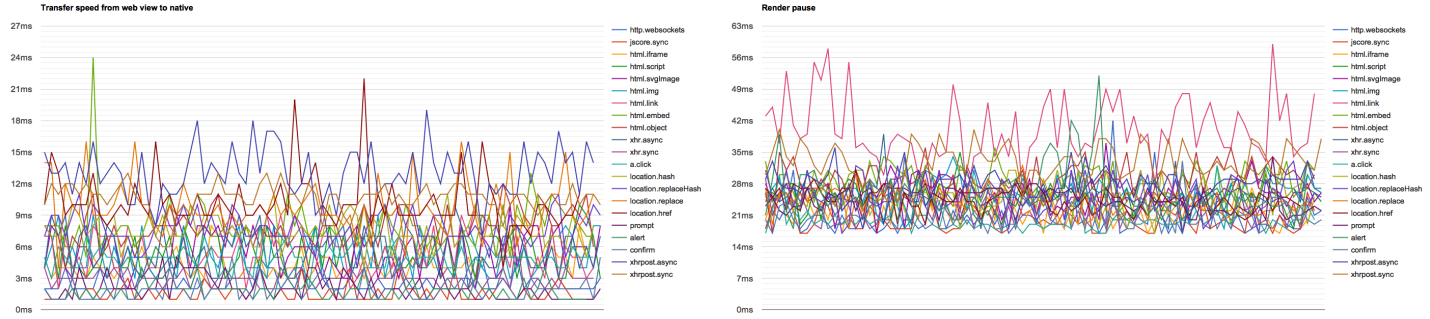


Figure 39: (small, iphone6plus) Sample series from `UIWebView` to native

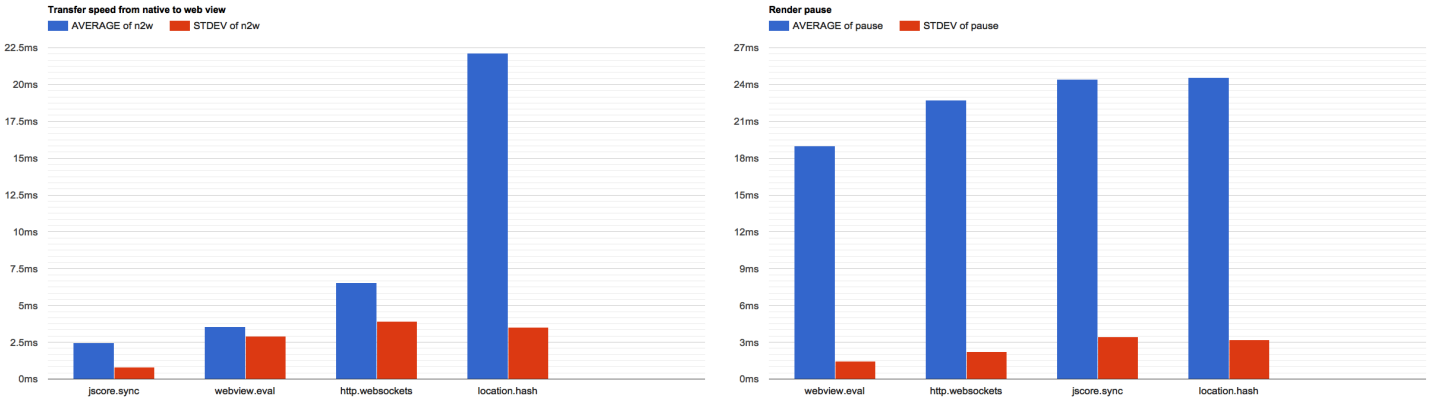


Figure 40: (small, iphone6plus) Transfer speed and render pause from native to `UIWebView`

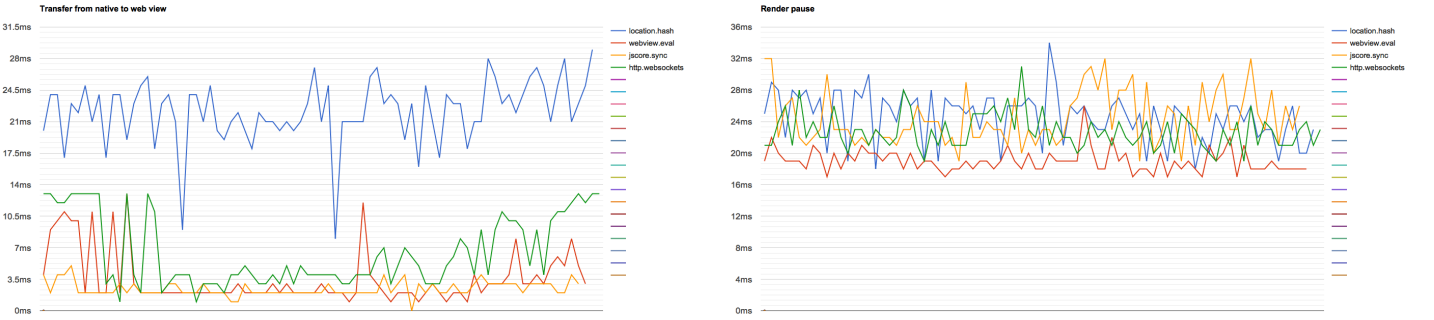


Figure 41: (small, iphone6plus) Sample series from native to `UIWebView`

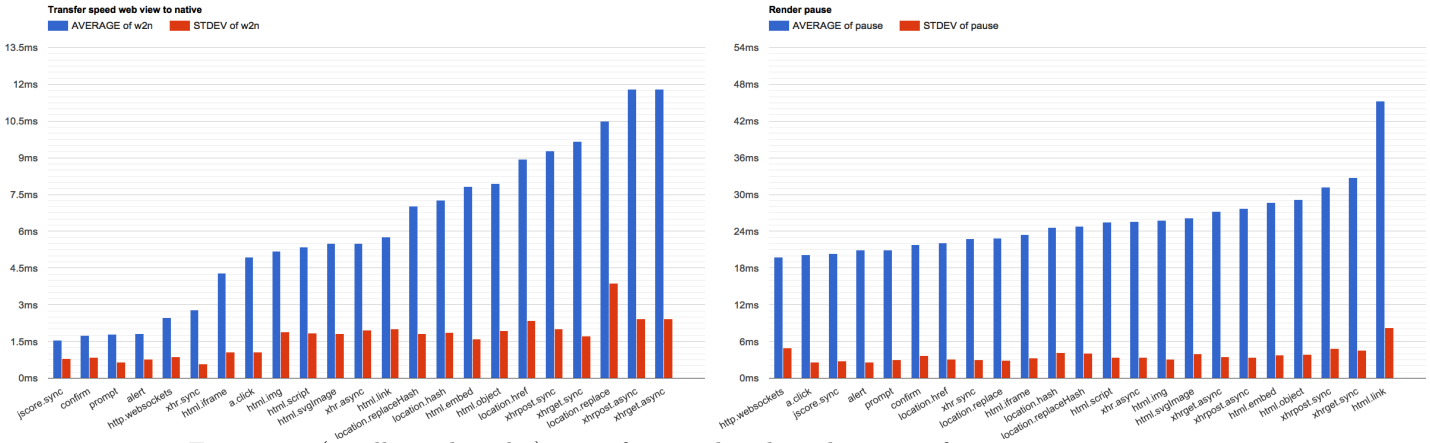


Figure 42: (small, ipodtouch6) Transfer speed and render pause from `UIWebView` to native

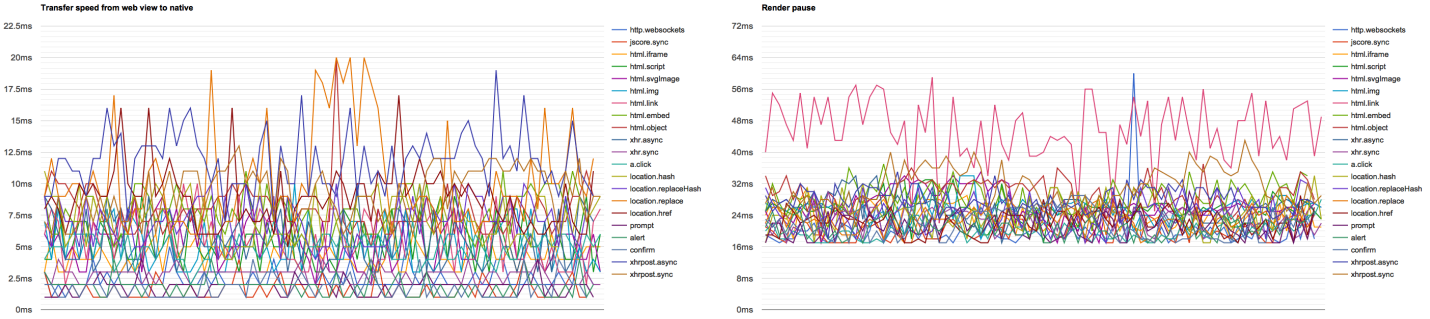


Figure 43: (small, ipodtouch6) Sample series from `UIWebView` to native

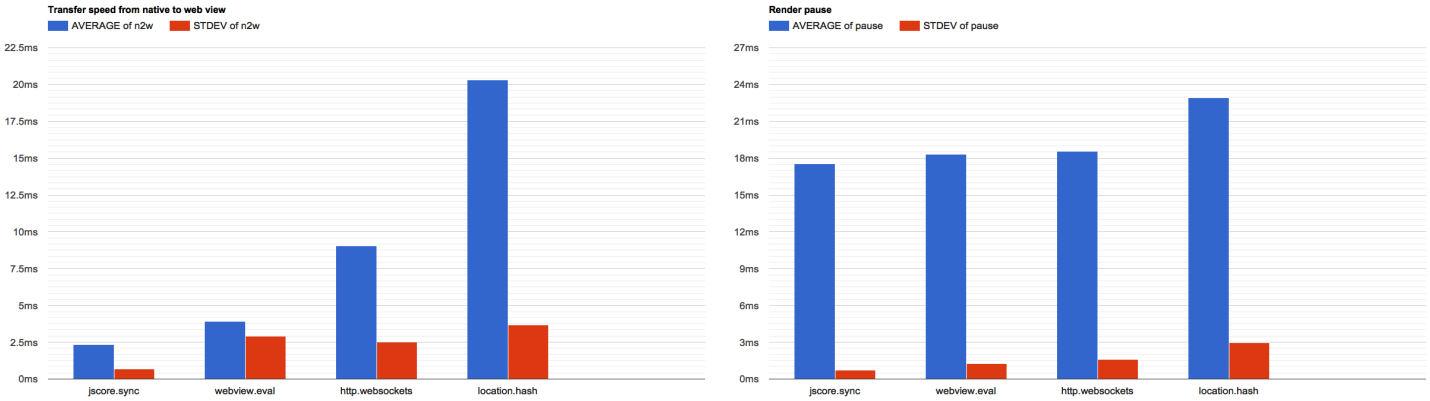


Figure 44: (small, ipodtouch6) Transfer speed and render pause from native to `UIWebView`

Figure 45: (small, ipodtouch6) Sample series from native to `UIWebView`

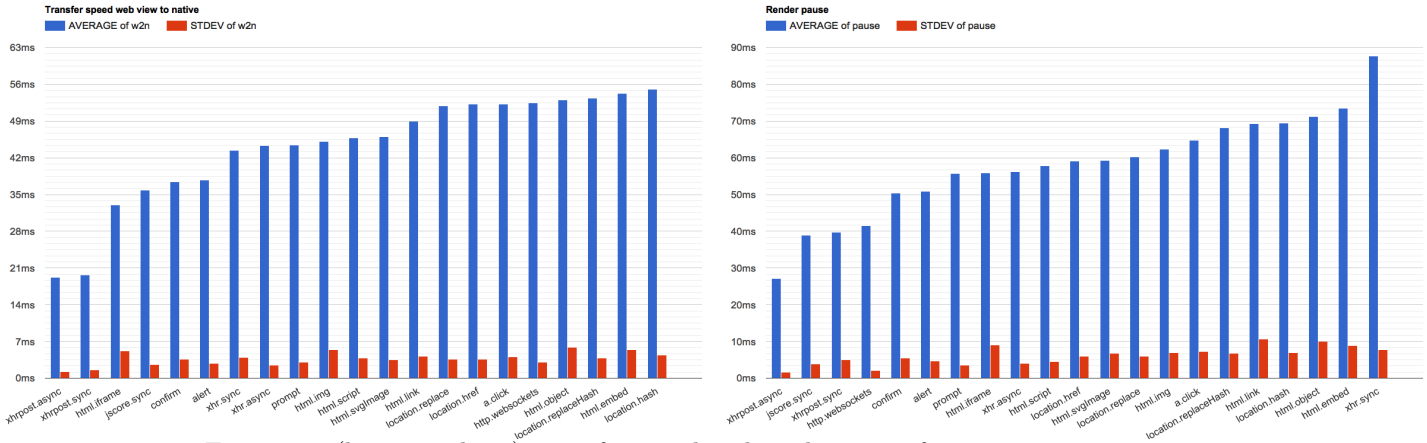


Figure 46: (large, ipadair2) Transfer speed and render pause from `UIWebView` to native

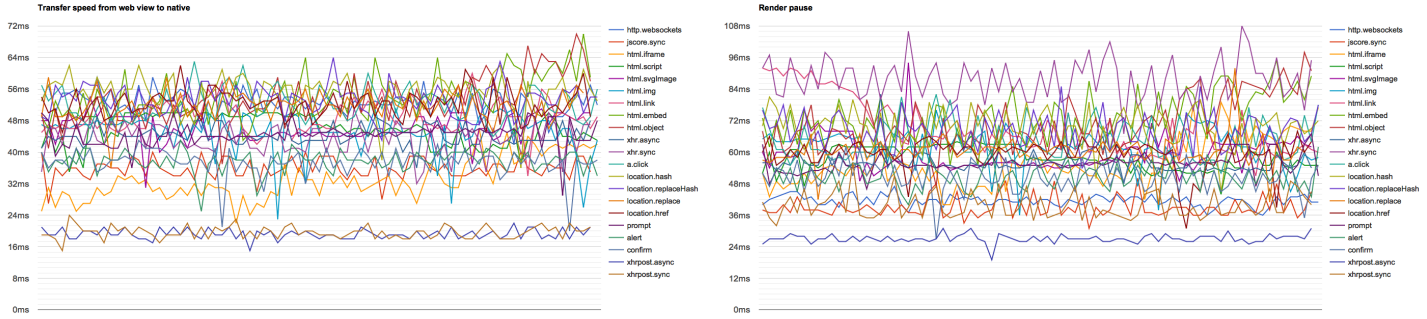


Figure 47: (large, ipadair2) Sample series from `UIWebView` to native

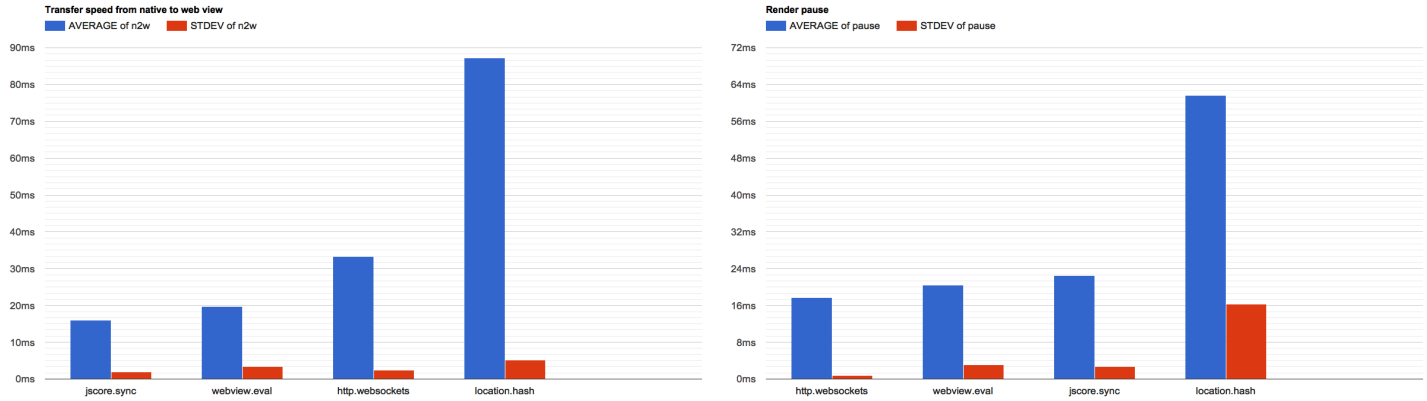


Figure 48: (large, ipadair2) Transfer speed and render pause from native to `UIWebView`

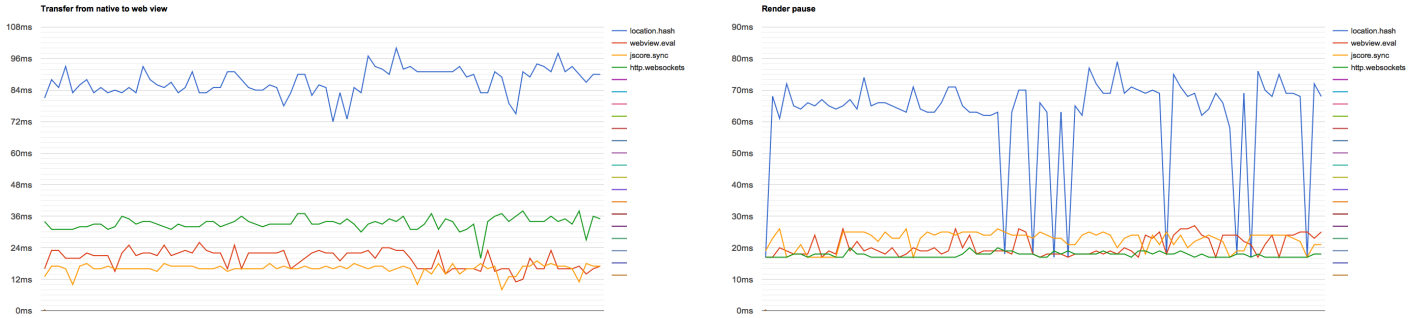


Figure 49: (large, ipadair2) Sample series from native to `UIWebView`

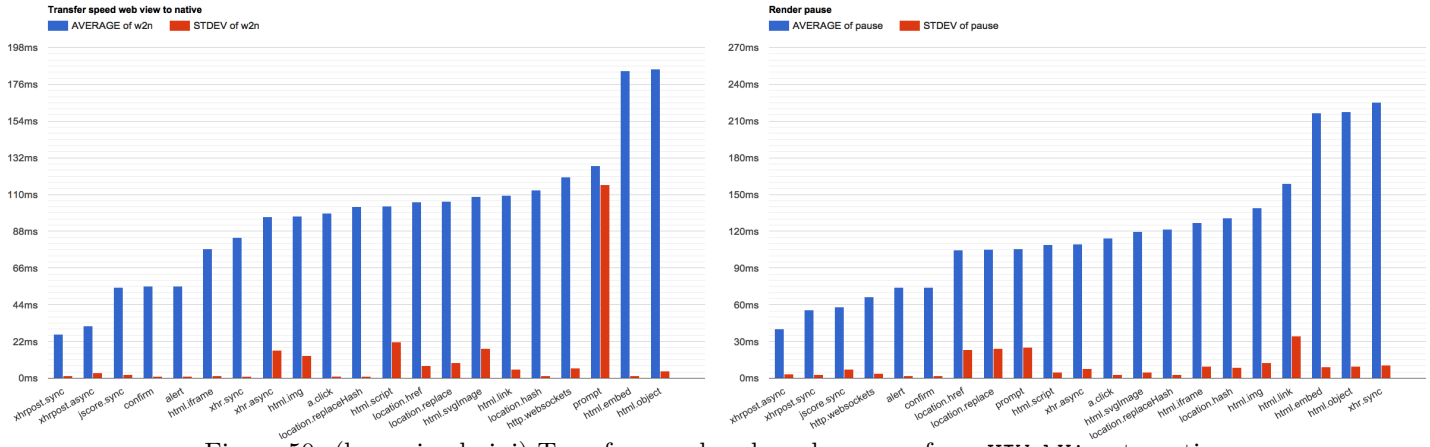


Figure 50: (large, ipadmini) Transfer speed and render pause from `UIWebView` to native

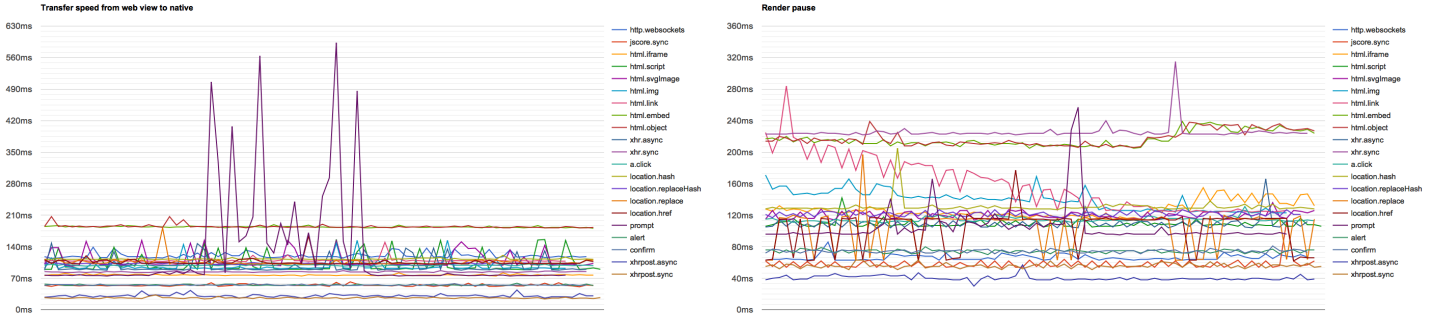


Figure 51: (large, ipadmini) Sample series from `UIWebView` to native

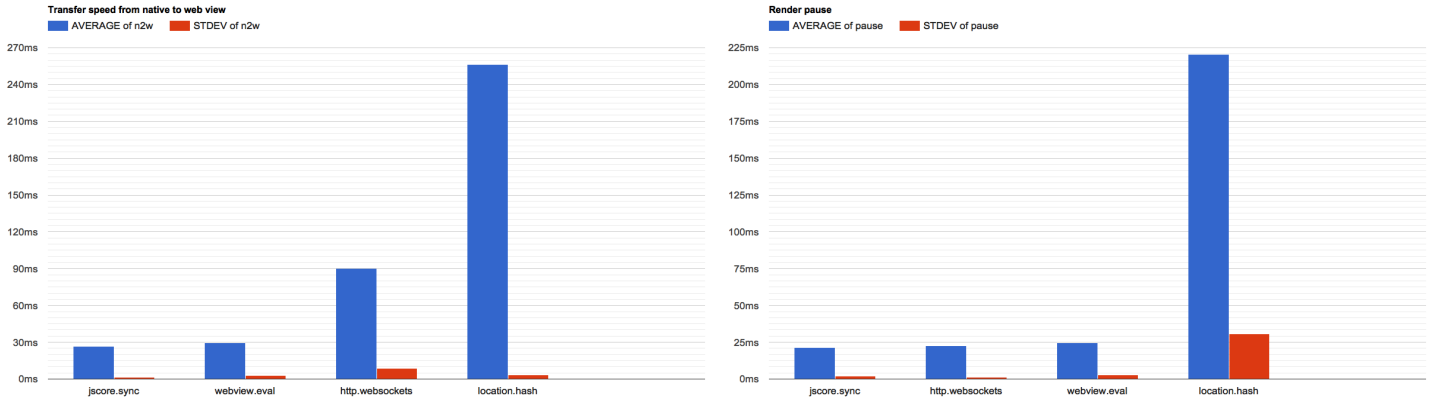


Figure 52: (large, ipadmini) Transfer speed and render pause from native to `UIWebView`

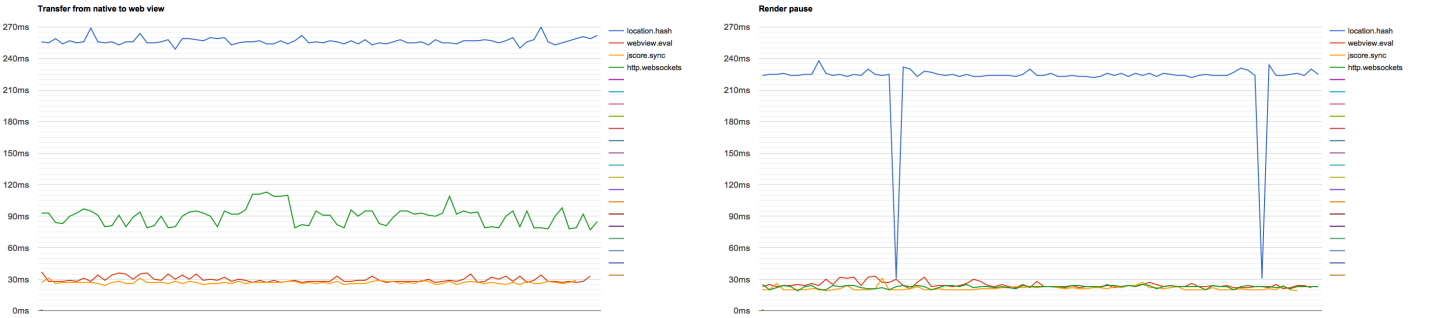


Figure 53: (large, ipadmini) Sample series from native to `UIWebView`

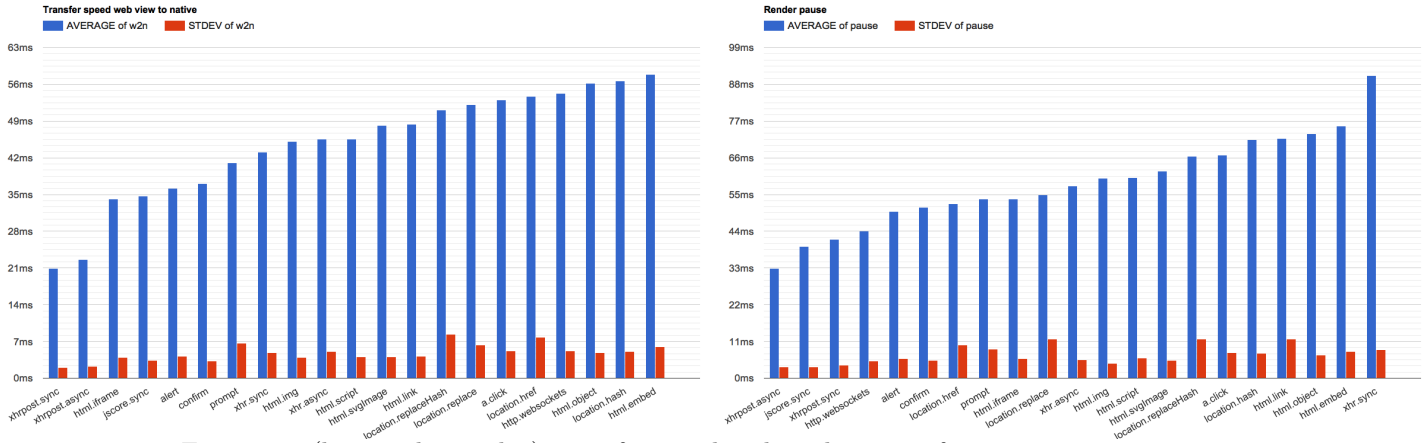


Figure 54: (large, iphone6plus) Transfer speed and render pause from `UIWebView` to native

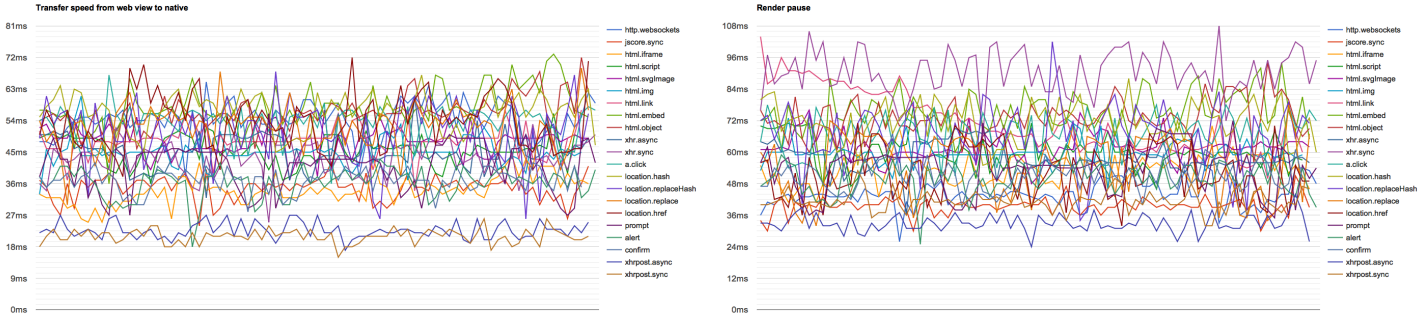


Figure 55: (large, iphone6plus) Sample series from `UIWebView` to native

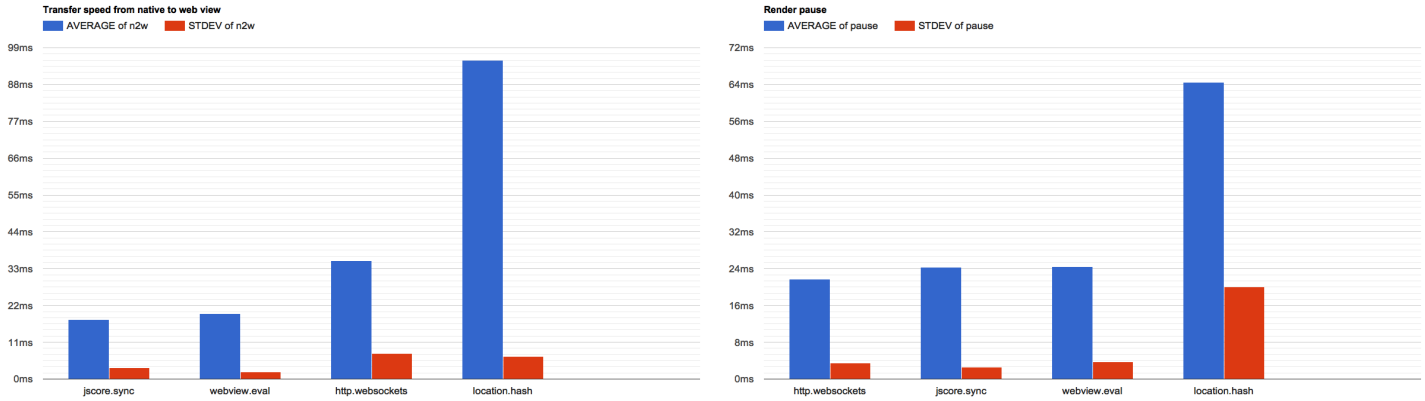


Figure 56: (large, iphone6plus) Transfer speed and render pause from native to `UIWebView`

Figure 57: (large, iphone6plus) Sample series from native to `UIWebView`

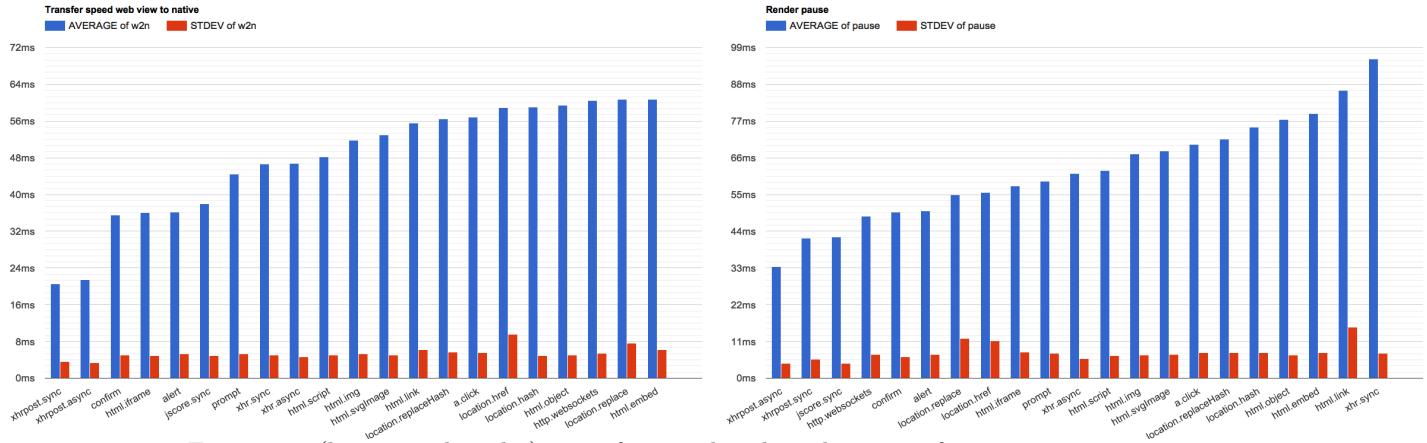


Figure 58: (large, ipodtouch6) Transfer speed and render pause from `UIWebView` to native

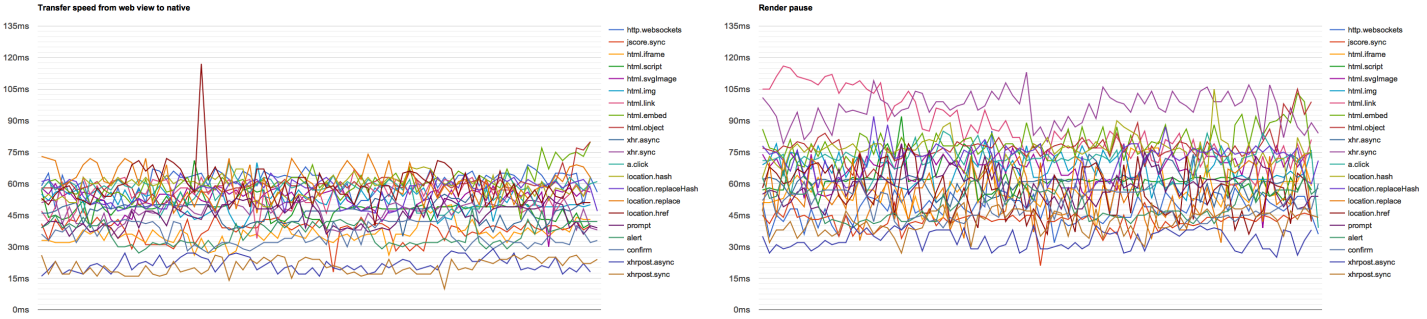


Figure 59: (large, ipodtouch6) Sample series from `UIWebView` to native

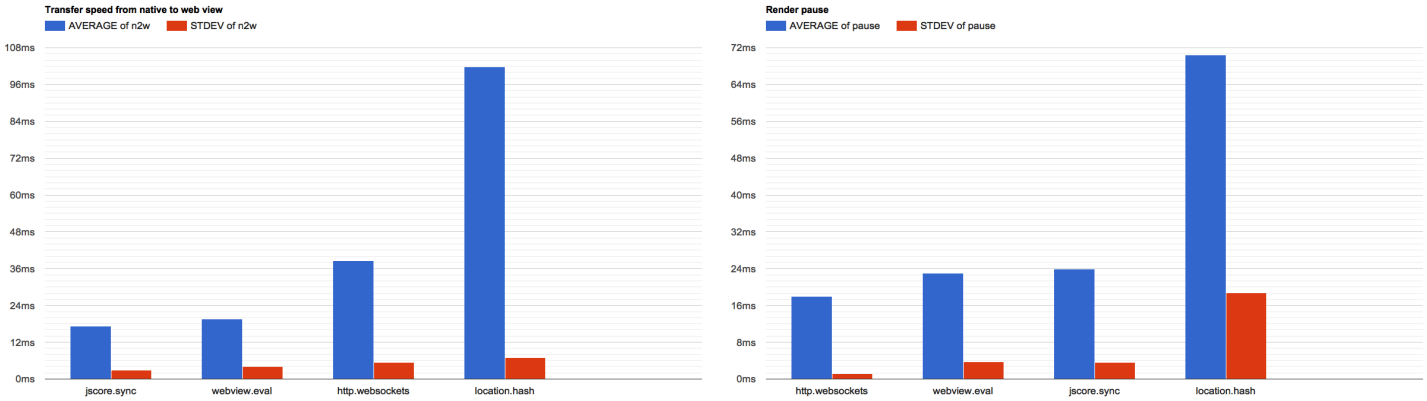


Figure 60: (large, ipodtouch6) Transfer speed and render pause from native to `UIWebView`

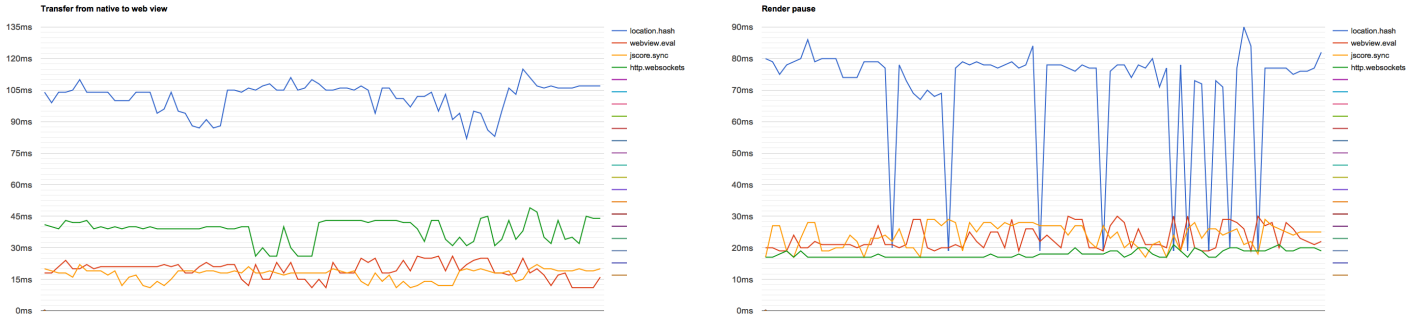


Figure 61: (large, ipodtouch6) Sample series from native to `UIWebView`

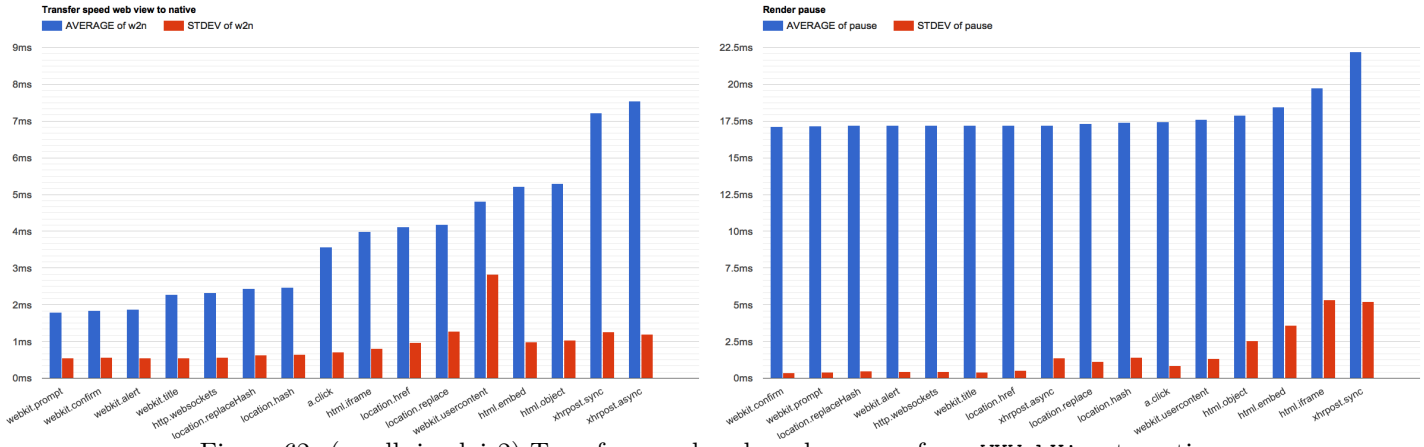


Figure 62: (small, ipadair2) Transfer speed and render pause from `WKWebView` to native

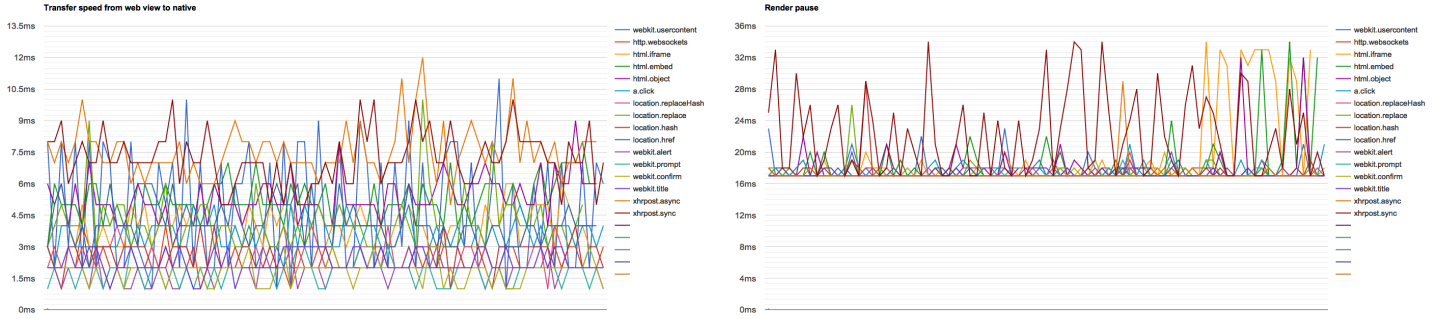


Figure 63: (small, ipadair2) Sample series from `WKWebView` to native

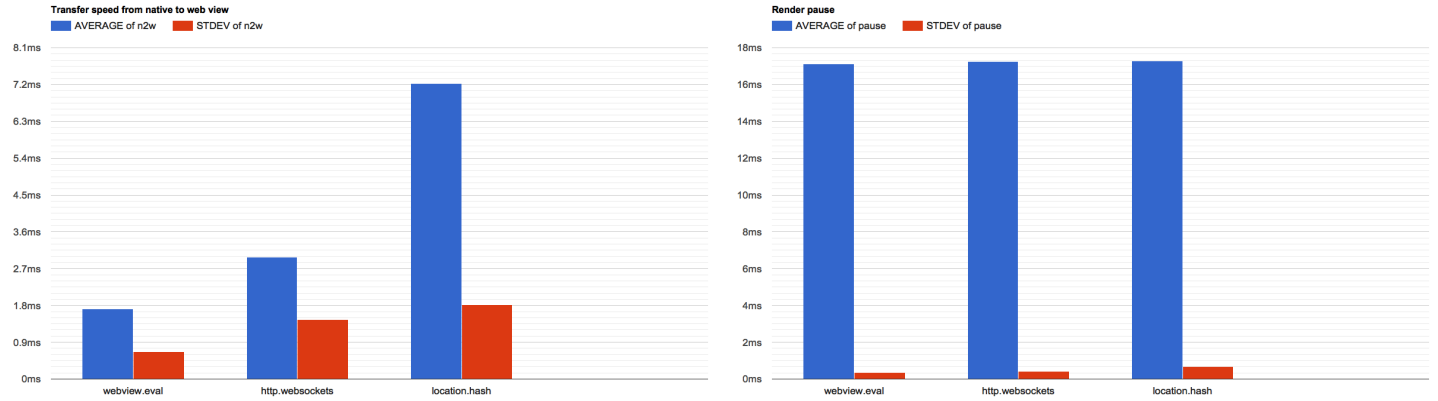


Figure 64: (small, ipadair2) Transfer speed and render pause from native to `WKWebView`

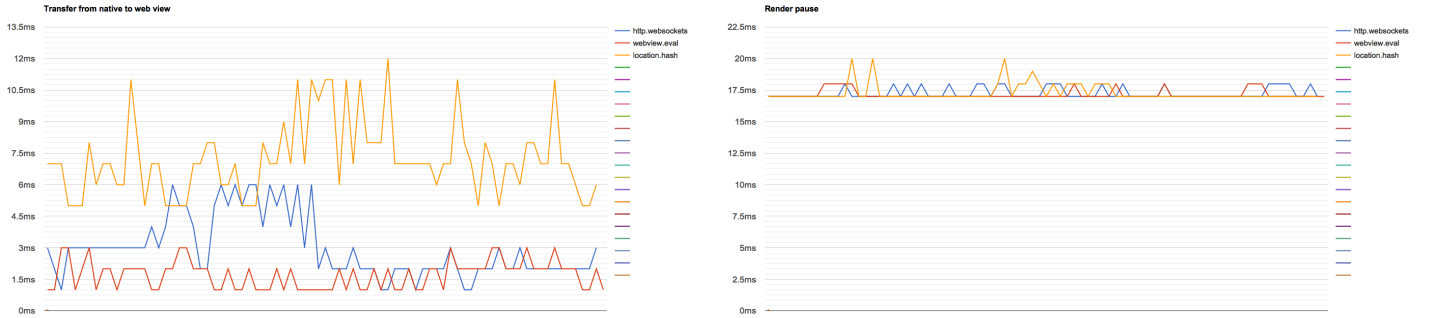


Figure 65: (small, ipadair2) Sample series from native to `WKWebView`

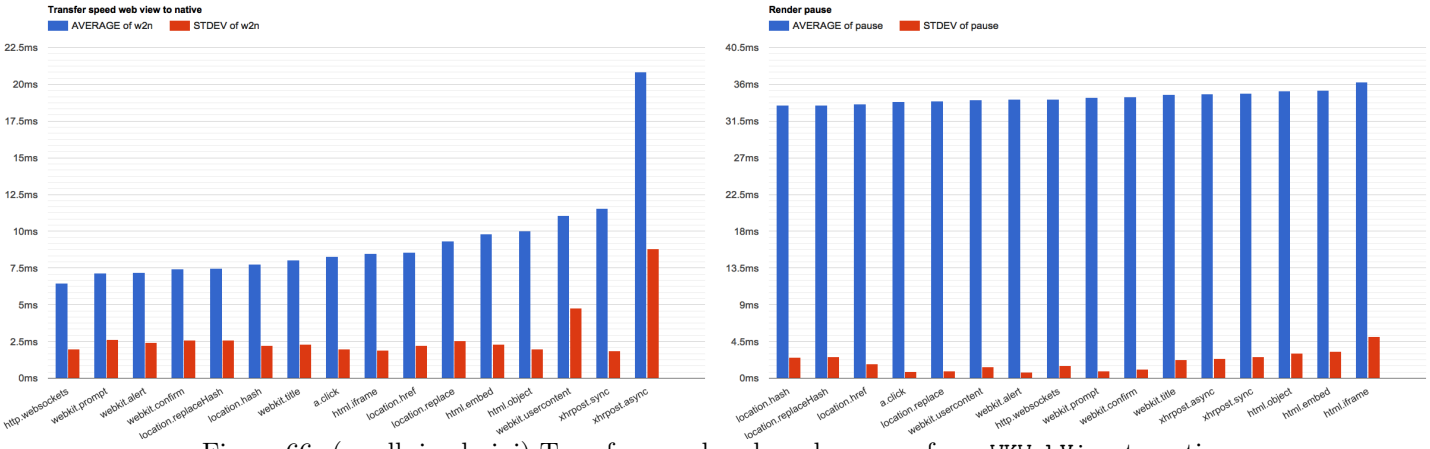


Figure 66: (small, ipadmini) Transfer speed and render pause from `WKWebView` to native

Figure 67: (small, ipadmini) Sample series from `WKWebView` to native

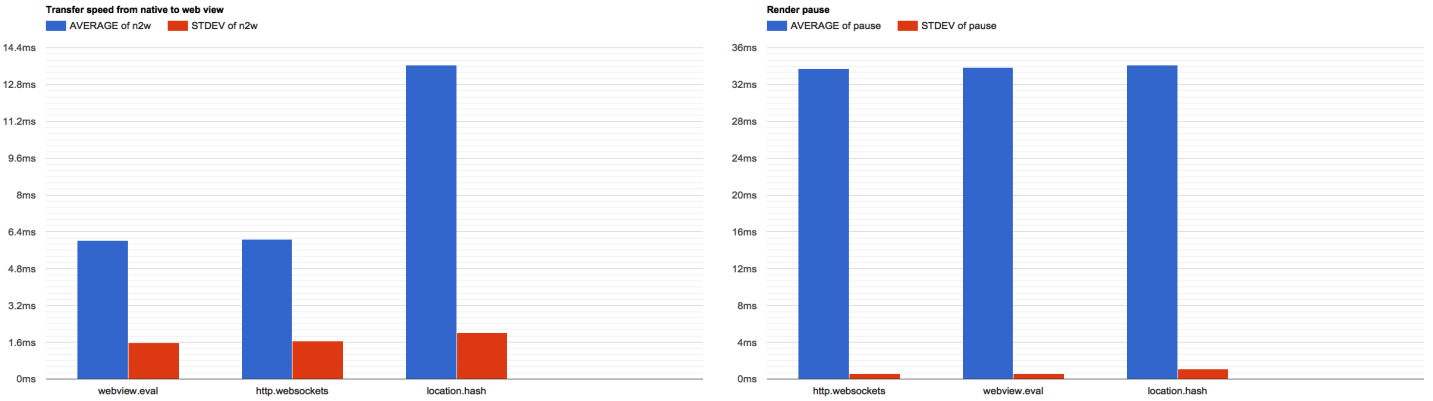


Figure 68: (small, ipadmini) Transfer speed and render pause from native to `WKWebView`

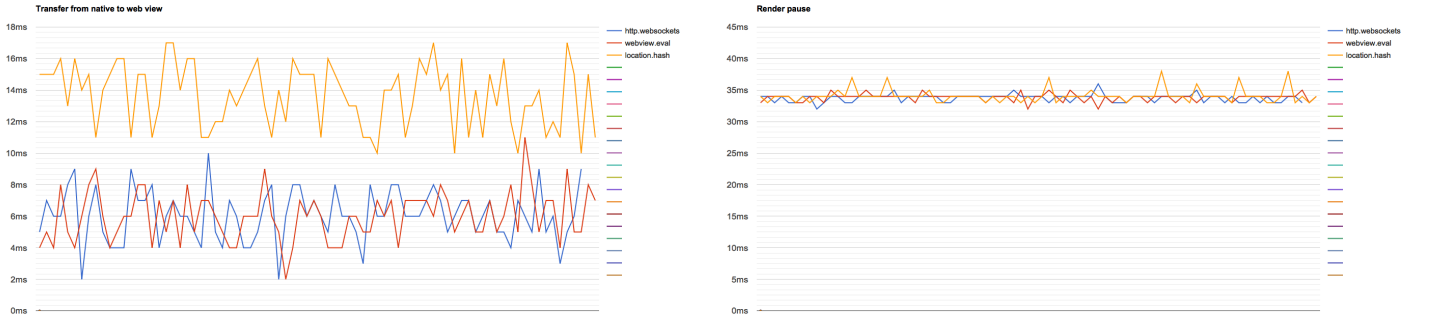


Figure 69: (small, ipadmini) Sample series from native to `WKWebView`

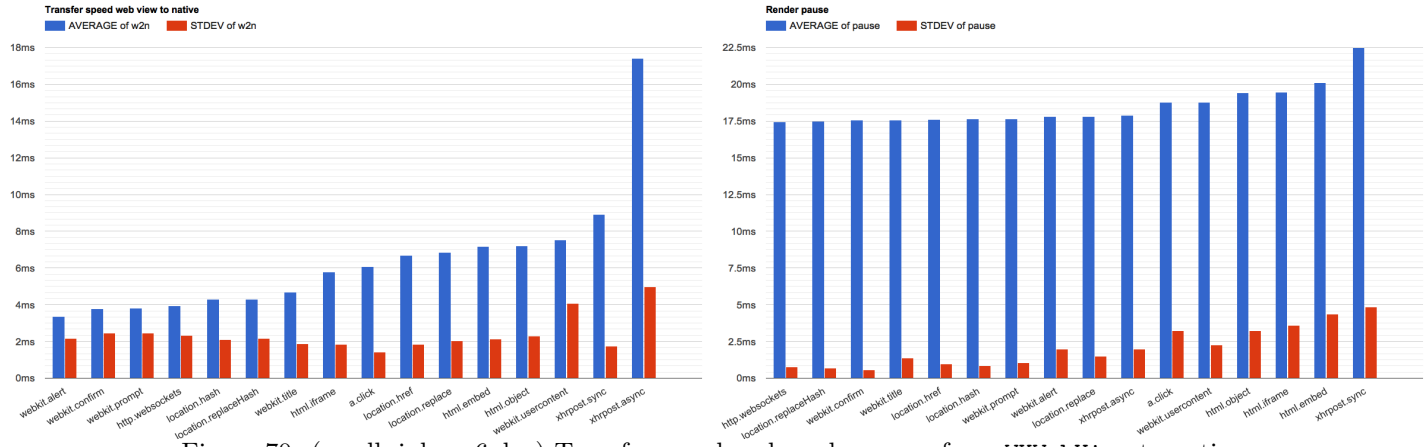


Figure 70: (small, iphone6plus) Transfer speed and render pause from `WKWebView` to native

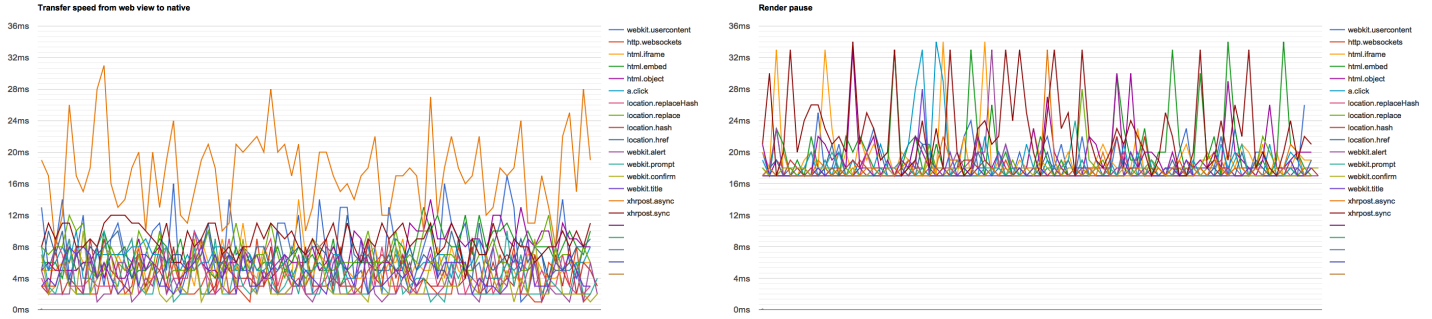


Figure 71: (small, iphone6plus) Sample series from `WKWebView` to native

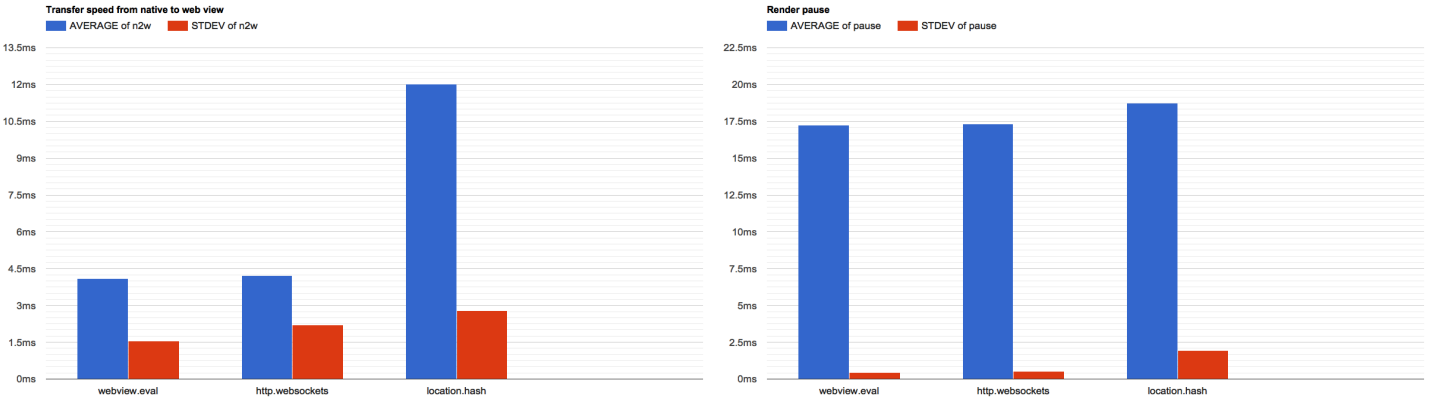


Figure 72: (small, iphone6plus) Transfer speed and render pause from native to `WKWebView`

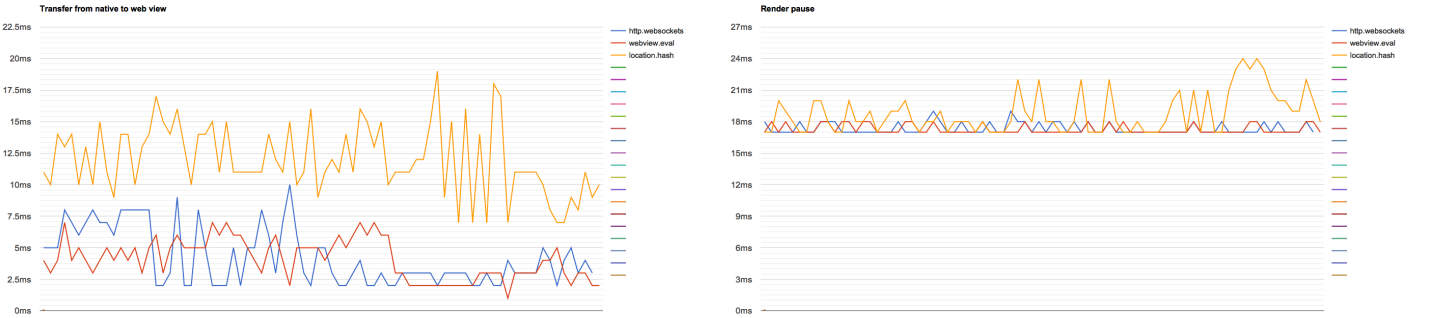


Figure 73: (small, iphone6plus) Sample series from native to `WKWebView`

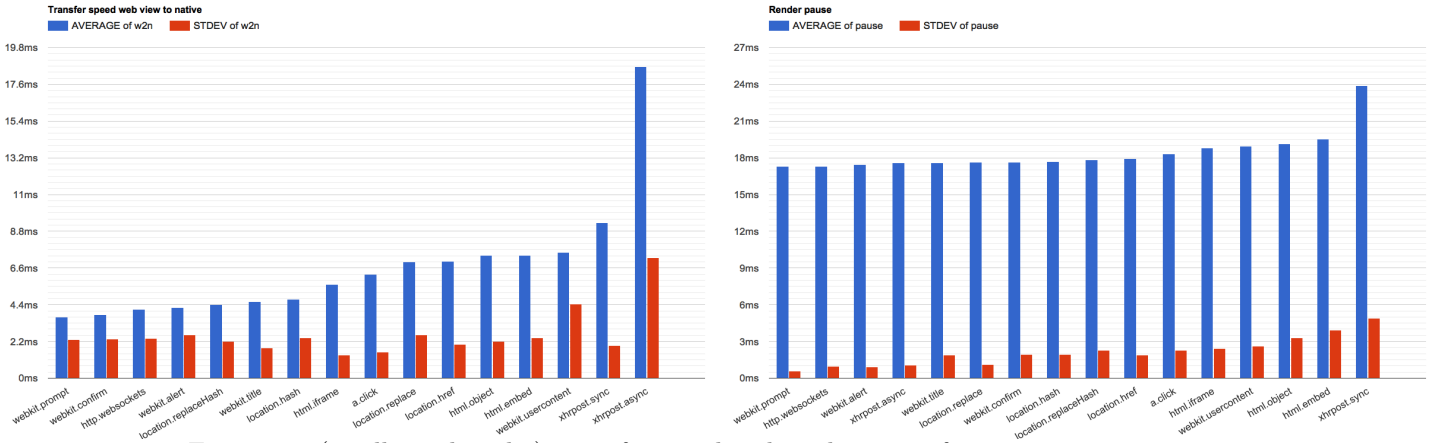


Figure 74: (small, ipodtouch6) Transfer speed and render pause from `WKWebView` to native

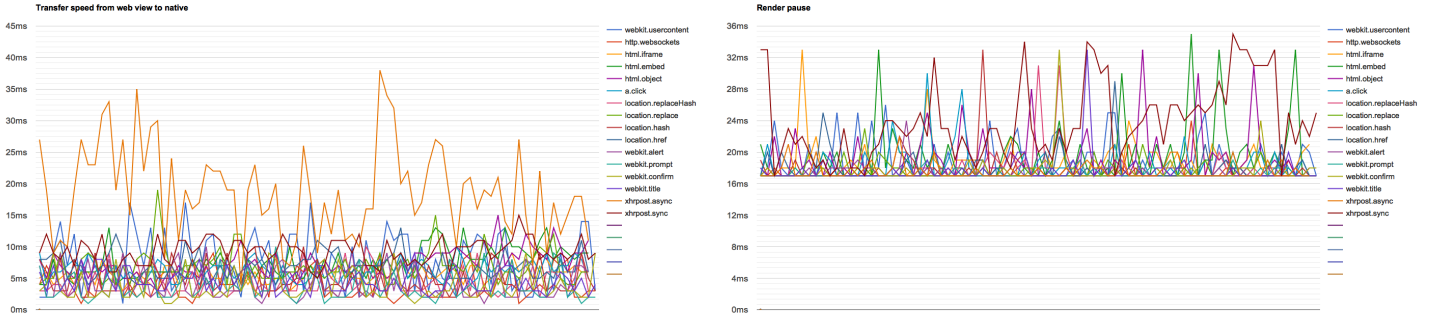


Figure 75: (small, ipodtouch6) Sample series from `WKWebView` to native

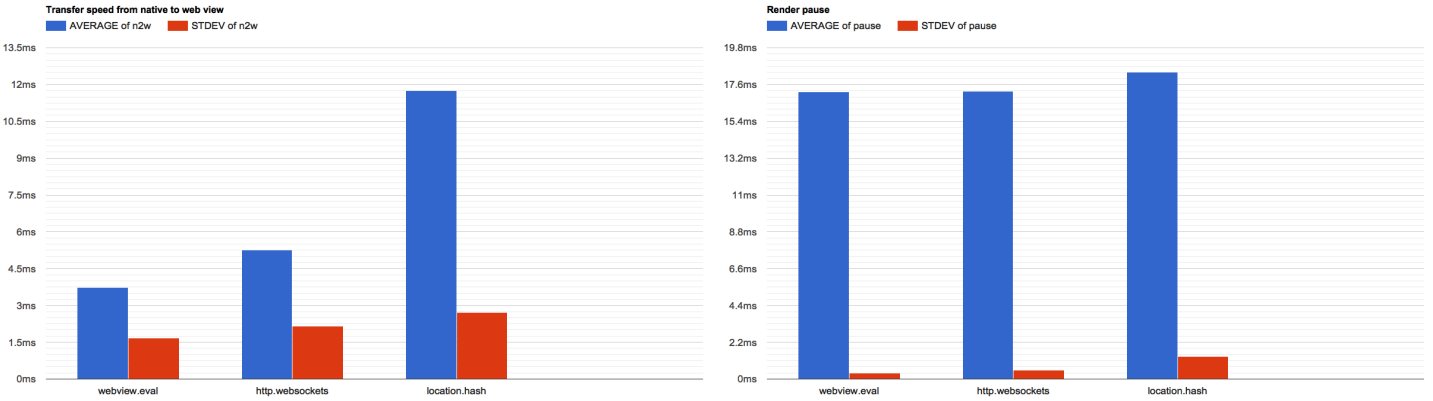


Figure 76: (small, ipodtouch6) Transfer speed and render pause from native to `WKWebView`

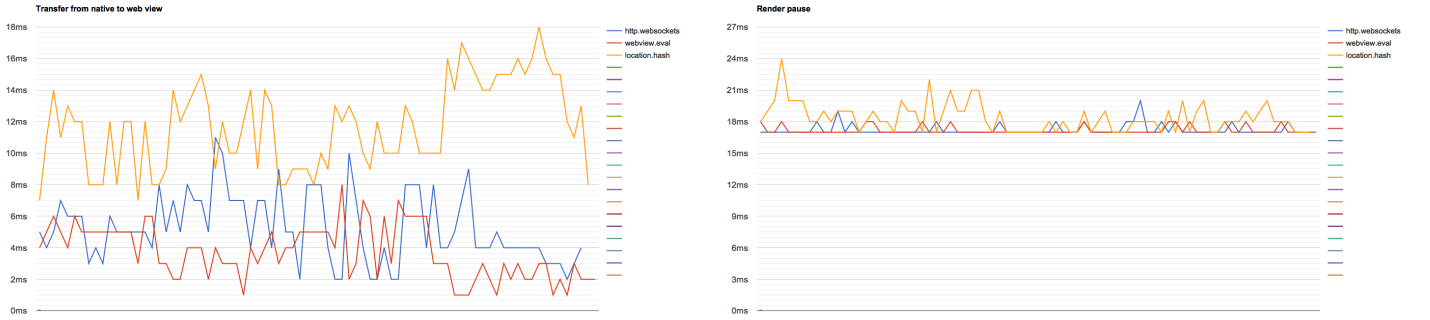


Figure 77: (small, ipodtouch6) Sample series from native to `WKWebView`

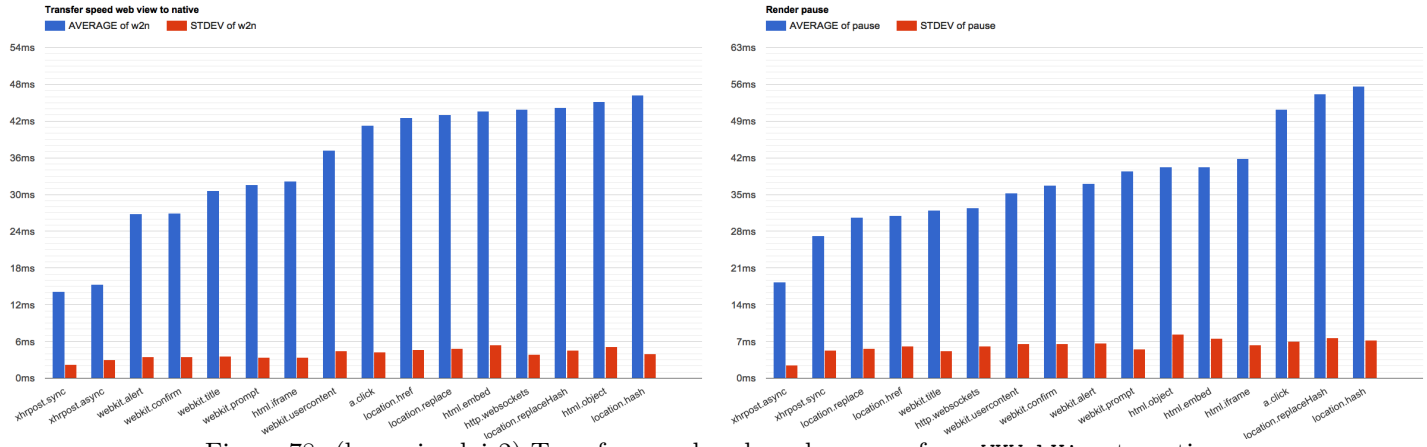


Figure 78: (large, ipadair2) Transfer speed and render pause from `WKWebView` to native

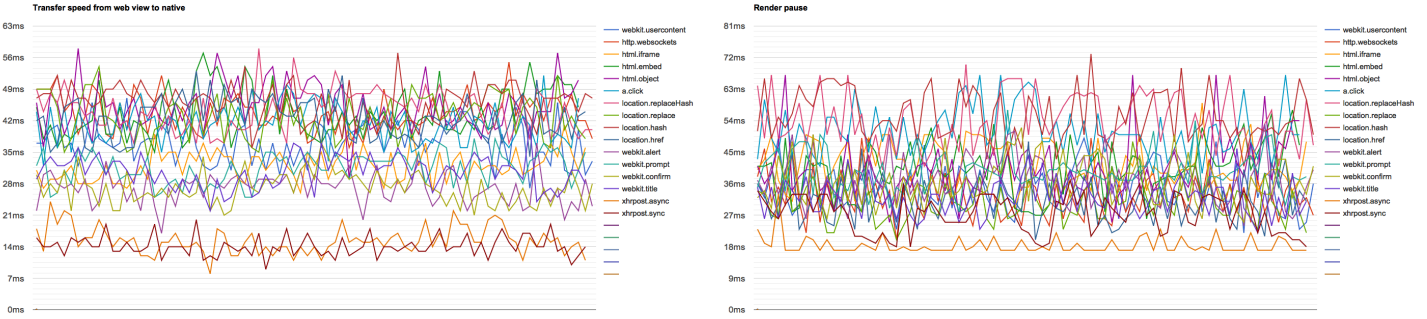


Figure 79: (large, ipadair2) Sample series from `WKWebView` to native

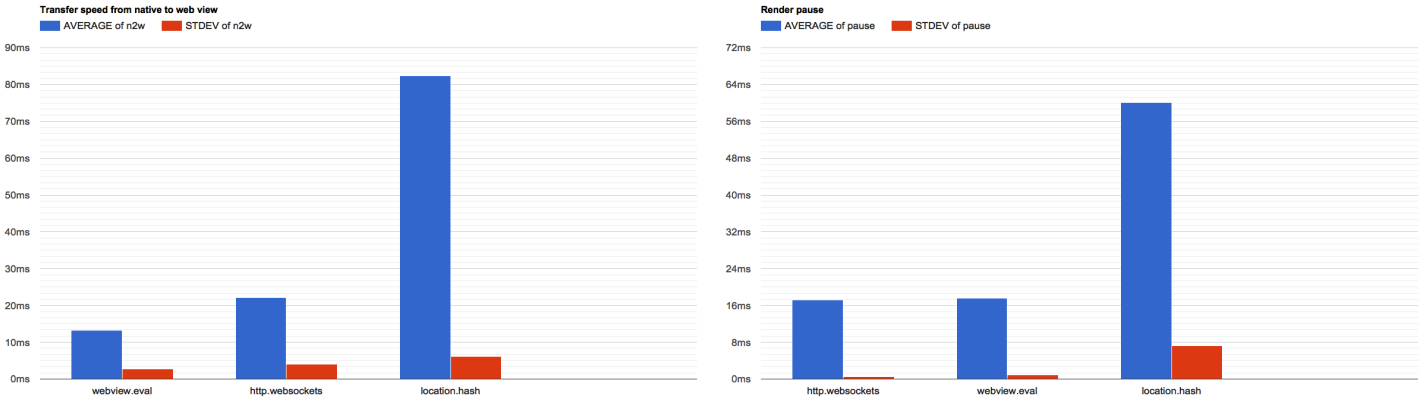


Figure 80: (large, ipadair2) Transfer speed and render pause from native to `WKWebView`

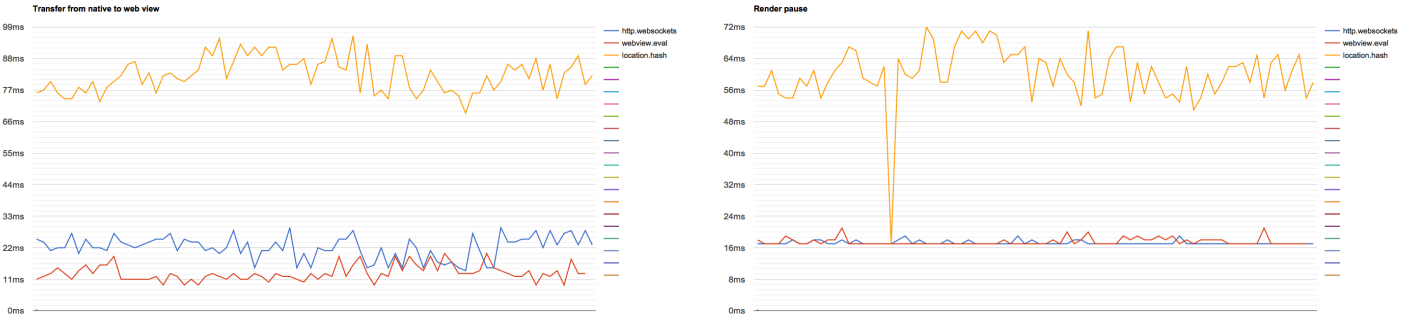


Figure 81: (large, ipadair2) Sample series from native to `WKWebView`

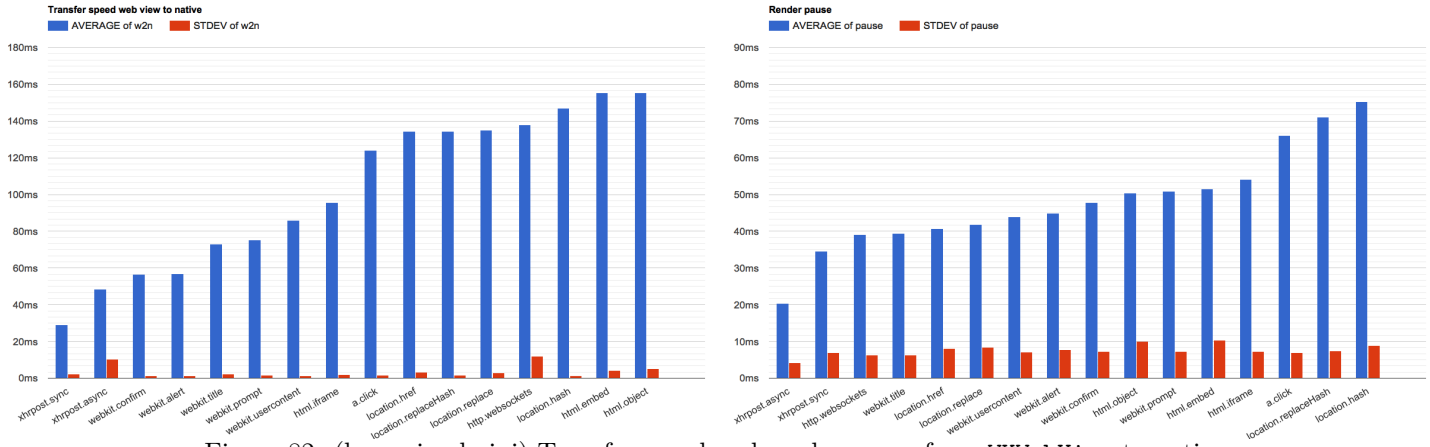


Figure 82: (large, ipadmini) Transfer speed and render pause from `WKWebView` to native

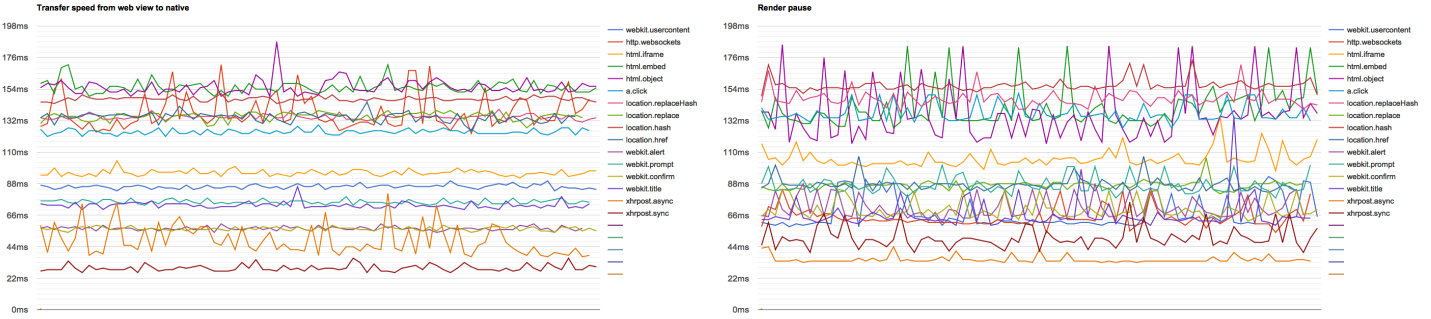


Figure 83: (large, ipadmini) Sample series from `WKWebView` to native

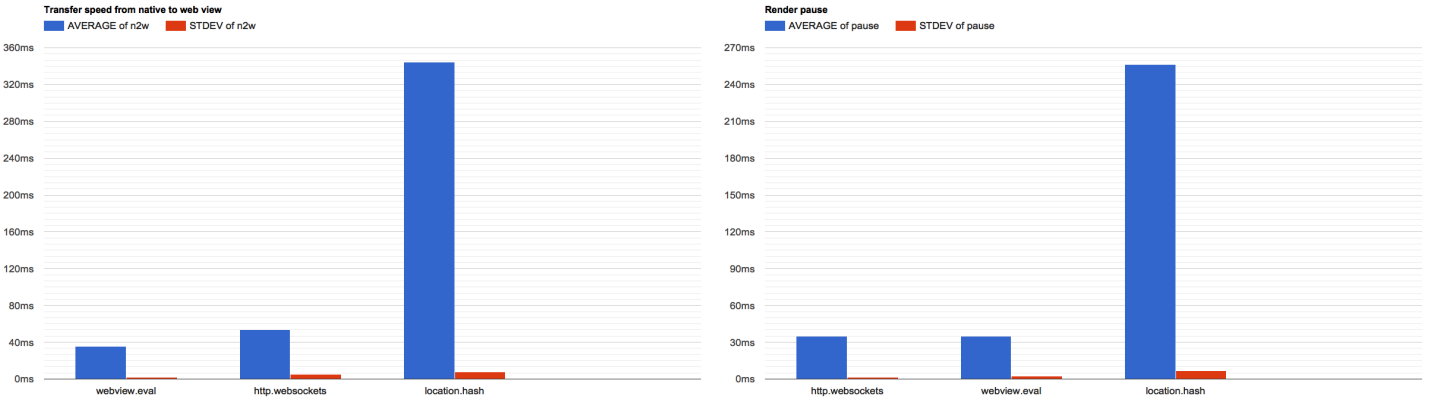


Figure 84: (large, ipadmini) Transfer speed and render pause from native to `WKWebView`

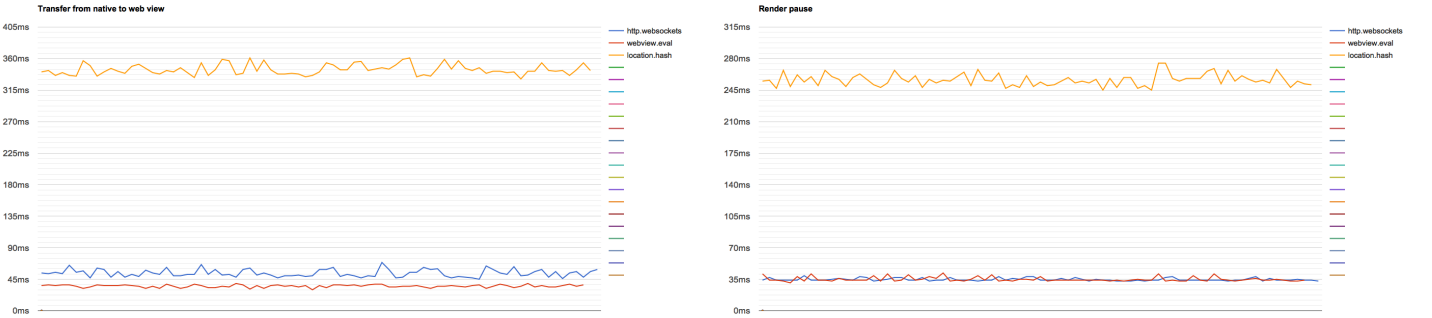


Figure 85: (large, ipadmini) Sample series from native to `WKWebView`

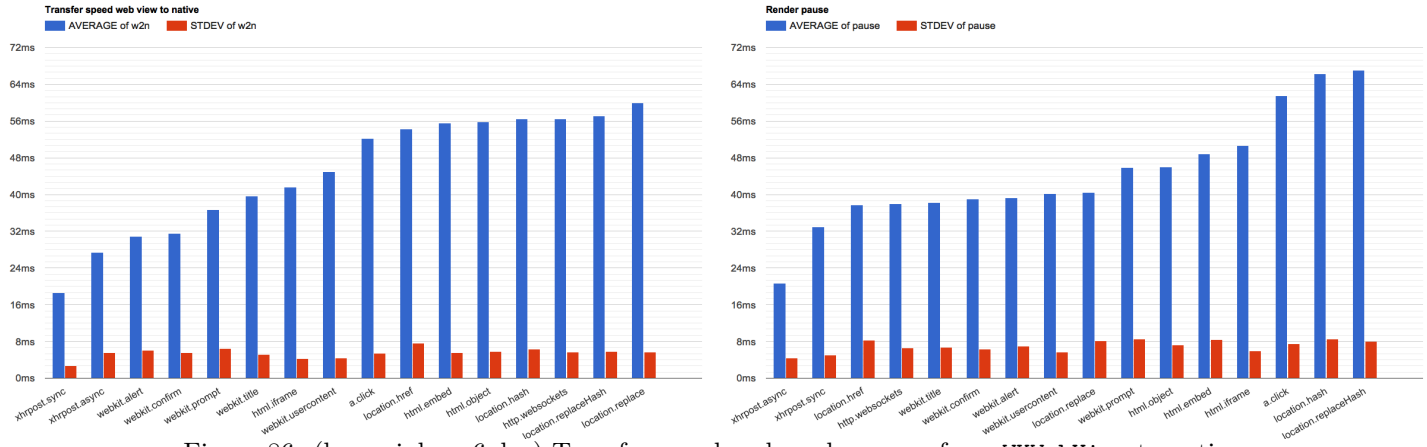


Figure 86: (large, iphone6plus) Transfer speed and render pause from `WKWebView` to native

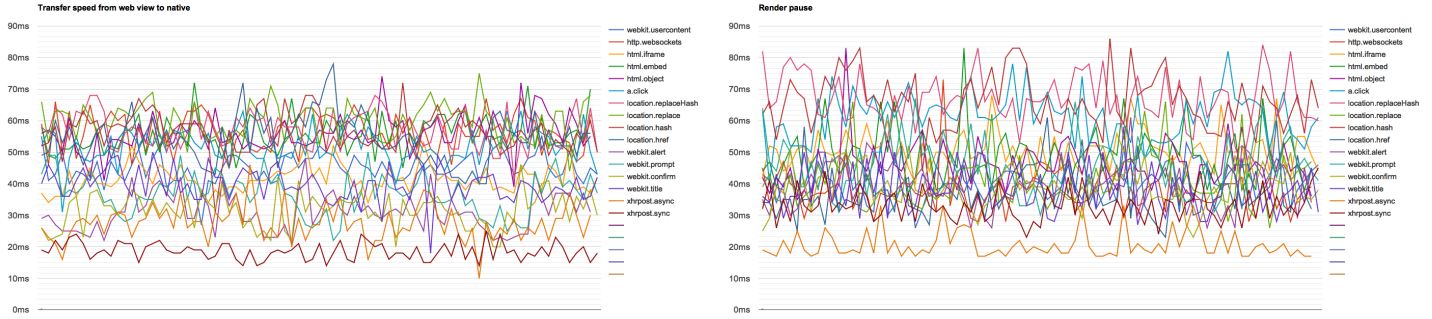


Figure 87: (large, iphone6plus) Sample series from `WKWebView` to native

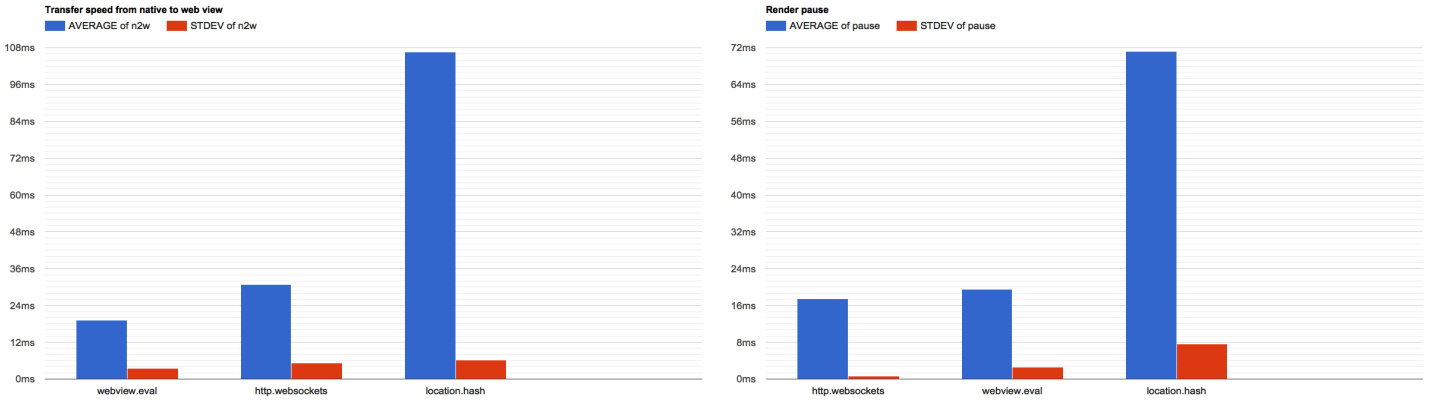


Figure 88: (large, iphone6plus) Transfer speed and render pause from native to `WKWebView`

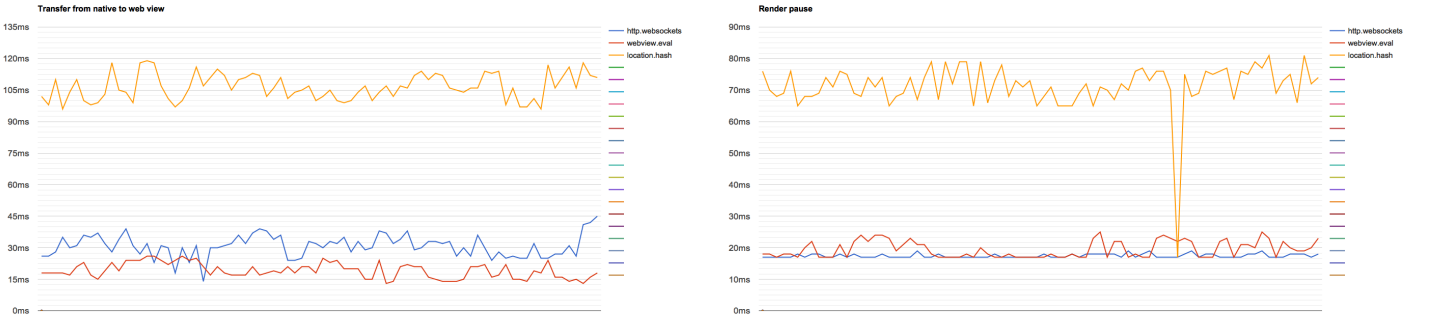


Figure 89: (large, iphone6plus) Sample series from native to `WKWebView`

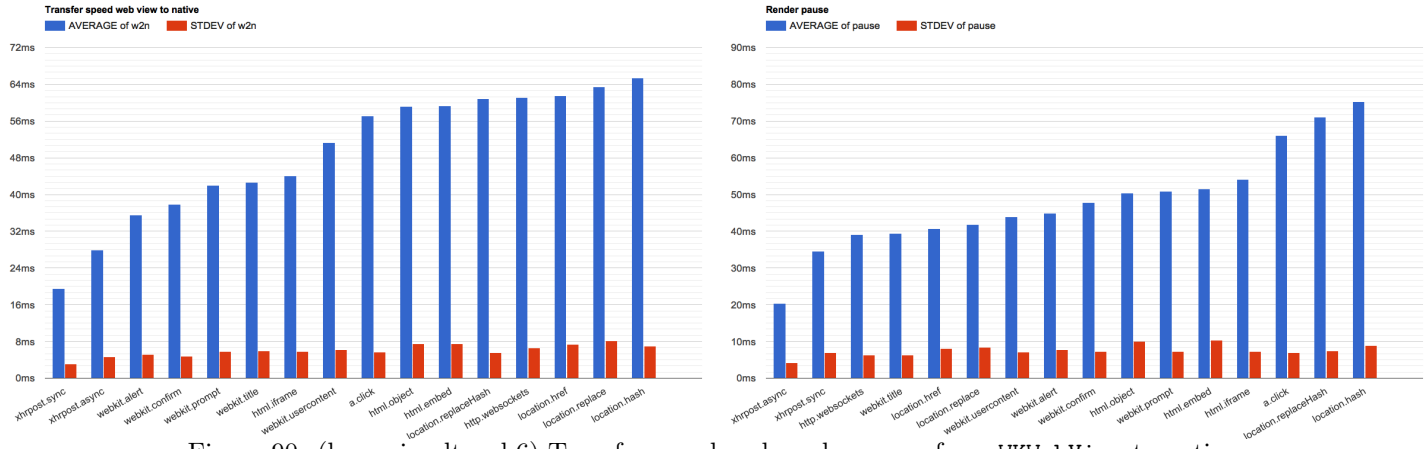


Figure 90: (large, ipodtouch6) Transfer speed and render pause from `WKWebView` to native

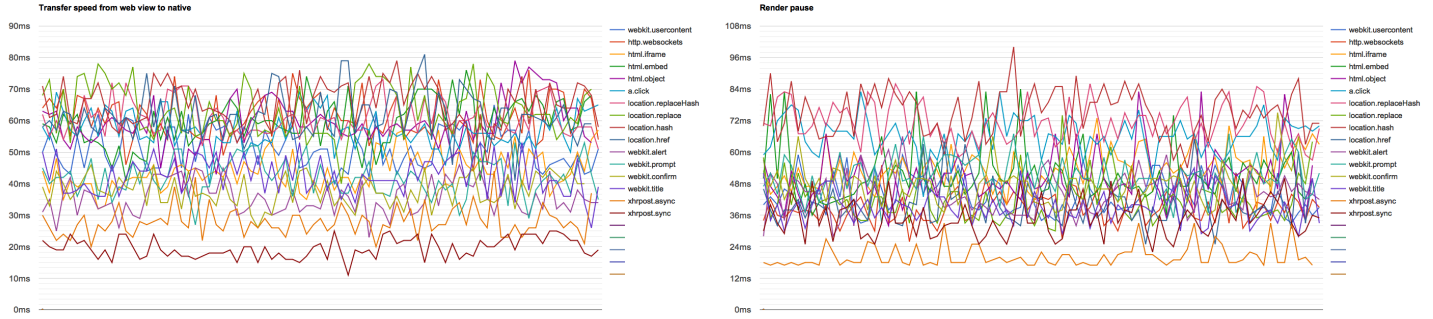


Figure 91: (large, ipodtouch6) Sample series from `WKWebView` to native

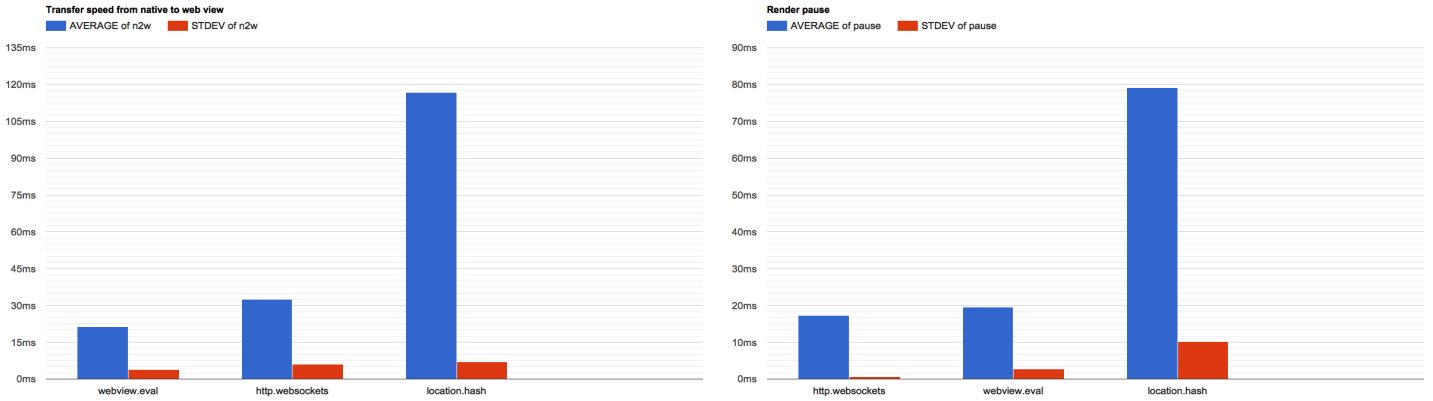


Figure 92: (large, ipodtouch6) Transfer speed and render pause from native to `WKWebView`

Figure 93: (large, ipodtouch6) Sample series from native to `WKWebView`