# Binary Compatible Graphics Support in Android for Running iOS Apps

Jeremy Andrus
Department of Computer Science
Columbia University
jeremya@cs.columbia.edu

Naser AlDuaij
Department of Computer Science
Columbia University
alduaij@cs.columbia.edu

Jason Nieh
Department of Computer Science
Columbia University
nieh@cs.columbia.edu

## ABSTRACT

Mobile apps make extensive use of GPUs on smartphones and tablets to access Web content. To support pervasive Web content, we introduce three key OS techniques for binary graphics compatibility necessary to build a real-world system to run iOS and Android apps together on the same smartphone or tablet. First diplomat usage patterns manage resources to bridge proprietary iOS and Android graphics implementations. Second, thread impersonation allows a single thread-specific context to be shared amongst multiple threads using multiple iOS and Android personas. Third, dynamic library replication allows multiple, independent instances of the same library to be loaded in a single process to support iOS apps on Android while using multiple graphics API versions at the same time. We use these techniques to build a system prototype, and demonstrate that it runs widely-used iOS apps, including apps such as Safari that use the popular GPU-accelerated WebKit framework, using a Google Nexus tablet running Android.

## CCS CONCEPTS

• **Computing methodologies** → **Graphics systems and interfaces**; • **Computer systems organization** → **Architectures**; • **Software and its engineering** → **Operating systems**; *Runtime environments*; • **Information systems** → *Browsers*;

## KEYWORDS

Android, iOS, Operating System Compatibility, Mobile Computing, Computer Graphics, GPUs, OpenGL, Web Browsers

**ACM Reference format:**
Jeremy Andrus, Naser AlDuaij, and Jason Nieh. 2017. Binary Compatible Graphics Support in Android for Running iOS Apps. In *Proceedings of Middleware '17, Las Vegas, NV, USA, December 11–15, 2017,* 13 pages.
DOI: 10.1145/3135974.3135981

## 1 INTRODUCTION

Binary compatibility, the ability to run an application anywhere, has been a long sought goal. Hardware virtualization solutions such as VMware are now widely used in desktop and server environments, running multiple operating system (OS) instances to run applications (apps) built for different software ecosystems. However, mobile devices such as tablets and smartphones are changing the way computing platforms are designed, which has important implications for binary compatibility support. Unlike the clear separation of hardware and software concerns in the traditional PC world, mobile platforms are highly vertically integrated platforms. Hardware components are integrated together in compact devices using non-standard interfaces. Software is customized for the hardware, often using proprietary libraries to interface with specialized hardware. Apps are tightly integrated with particular libraries and frameworks, and often only available on particular hardware platforms. Mobile platforms integrate a plethora of devices, such as GPUs, that use non-standard interfaces, which are directly used by apps to optimize performance. The lack of hardware standards together with the resource constraints of mobile platforms have made existing virtualization approaches unusable on smartphones and tablets.

To address this problem, we developed *Cycada* [3, 4] (formerly known as *Cider*), an OS compatibility architecture that runs apps built for different mobile ecosystems, iOS or Android, together on the same device. *Cycada* leverages the wide availability of open source software and the use of standardized APIs for mobile app development to build binary compatibility into an existing mobile OS. *Cycada* mimics the ABI of a foreign OS, iOS, enabling the domestic OS, Android, to run unmodified foreign binaries. It introduces two new binary compatibility mechanisms, compile-time code adaptation, and diplomatic functions. Compile-time code adaptation allows existing unmodified foreign (iOS) source code to be reused within the domestic (Android) kernel, reducing implementation effort required to support domestic and foreign binary interfaces.

A diplomat, or diplomatic function, temporarily switches the persona of a calling thread to execute domestic code from within a foreign app. A thread's persona, or execution mode, selects the kernel ABI personality and thread local storage (TLS) information used during execution. Diplomatic functions allow foreign, iOS, apps to leverage domestic, Android, libraries to access proprietary hardware and software interfaces. Diplomats are more than simple API "thunks," "glue code," or "trampolines." Beyond adapting two different API surfaces, a diplomat manages transitions between APIs across different thread-level personas.

Previous work demonstrated the feasibility of the *Cycada* approach, but did not fully address the problem of binary compatibility support for graphics acceleration as required by commonly used frameworks such as WebKit [7], the HTML and JavaScript rendering engine. WebKit consists of over 5 million lines of code [11] and


*Middleware '17, Las Vegas, NV, USA*

978-1-4503-4720-4/17/12…$15.00
DOI: 10.1145/3135974.3135981

heavily uses the GPU to accelerate Web page layout and rendering. Support for WebKit is essential for mobile app performance and functionality, as Web content is pervasively embedded in mobile apps, not just in Web browsers. However, binary compatible graphics support is a key challenge on mobile platforms because of their vertically integrated stack of proprietary closed-source vendor libraries that communicate directly to the kernel or device drivers through opaque, undocumented calls to control black-box GPU hardware.

Providing a framework to run fully hardware-accelerated foreign applications natively not only allows for running iOS apps on Android but potentially allows for running many combinations of foreign apps on different OSes. This opens up the possibility for many OS designers to support cross-platform software or to even extend and compliment containers [32, 34, 40, 44].

We present a graphics-focused study of *Cycada*, and extend its binary compatibility support through three new OS compatibility techniques necessary to build a complete system able to run apps built for different mobile ecosystems, iOS and Android, that require complex GPU-accelerated frameworks such as WebKit, together on the same device.

First, we extend *Cycada*'s basic diplomat construction to perform library-wide prelude and postlude operations in the context of the foreign OS before and after domestic library usage. Using these prelude and postlude functions, we formalize four *diplomat usage patterns*, direct, indirect, data-dependent, and multi, which together provide the complex and rigorous management of resources necessary for bridging between proprietary iOS and Android implementations of complex real-world graphics APIs, such as OpenGL.

Second, we introduce *thread impersonation* which allows thread-specific context to be shared among multiple threads running under multiple personas. A thread *impersonating* another thread temporarily takes on the identity of another thread to perform an action that may be thread-dependent.

Third, we load multiple, independent instances of a single library within the same process through *dynamic library replication* (DLR). DLR enables a dynamic linker to create separate loaded instances of a dynamic library in a process with unique virtual addresses for each instance of every symbol in the library including global and initialization data.

We extend *Cycada* with these three techniques to provide binary compatible support for the OpenGL ES (GLES) standard available on both iOS and Android, and heavily used by frameworks such as WebKit. While GLES is standardized, it relies on other graphics infrastructure such as graphics resource management to manage the state information, commands, and resources needed to draw using GLES, and this may not be standardized. Furthermore, the GLES standard is intended to be extensible, and vendor libraries and GPU hardware are free to implement new or any subset of available extensions. These three techniques are used to allow *Cycada*'s graphics compatibility mechanisms to bridge differences in GLES implementations across the platforms.

We extend the *Cycada* prototype, and demonstrate its effectiveness in enabling widely-used iOS apps such as Safari that make extensive use of WebKit to run on Android with reasonable graphics performance. We also demonstrate through detailed microbenchmarks that *Cycada* provides robust binary compatible graphics device support across a broad range of graphics functions. Our detailed study, new OS compatibility techniques, experimental results, and our overall experiences building binary compatible graphics devices support are also useful for other approaches such as virtualization in the context of mobile platforms.

Our experiences with *Cycada* show that our compatibility mechanisms, diplomat usage patterns, thread impersonation, and DLR, are key to supporting graphics binary compatibility. Our study shows: **(1)** *Real-world graphics implementations vary greatly between platforms.* Although the GLES API is standardized, both iOS and Android take advantage of GLES extensions such that more than half of the extensions used in one platform are not available in the other. Each extensions adds API entry points or modifies the behavior of an existing API. GPU binary compatibility or virtualization approaches that perform simple API forwarding or basic API thunks will not work for these mobile platforms due to the large differences between the resulting extended APIs. Despite their GLES differences, it is possible to map iOS GLES to Android GLES. Most iOS GLES functions, including extension functions, can be supported by leveraging one or more Android GLES functions via diplomat usage patterns. **(2)** *iOS and Android have substantially different graphics resource management APIs.* Graphics rendering APIs such as GLES require display and memory management APIs to provide window and memory management. iOS uses an Apple-proprietary API, EAGL, while Android uses the standardized EGL API. These APIs are different enough that running iOS apps on Android requires reimplementing Apple's EAGL APIs, which also requires reverse engineering the EAGL library in the absence of access to non-public Apple specifications and source code. Fortunately, the EAGL API is small, and many functions can leverage aspects of Android's EGL through a combination of diplomat usage patterns and DLR. However, Android provides an incomplete key EGL extensions which complicates its use. **(3)** *iOS graphics libraries are designed for multi-threaded use that is not supported in Android's graphics libraries. Cycada* uses thread impersonation and DLR to allow Android threads to impersonate iOS graphics threads. Each thread uses diplomats to access multiple, isolated instances of the Android graphics libraries. The isolated graphics libraries and thread impersonation allow each Android thread to perform thread-specific actions and support multi-threaded iOS graphics functionality. **(4)** *iOS provides richer support than Android for multiple GLES API versions.* Both platforms support multiple GLES API versions which are useful for different purposes but incompatible. However, iOS allows multiple GLES versions to be used simultaneously by different threads in the same process while Android does not. This is widely used by multi-threaded iOS apps. For example, an iOS game may use GLES v1 APIs to render game graphics, but use a WebKit view to render an HTML "about" page which uses GLES v2 APIs. *Cycada* uses DLR to support iOS apps using multiple GLES API versions on Android.

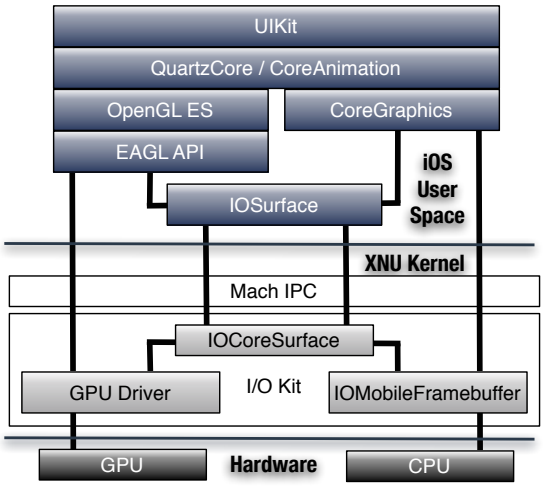


**Figure 1: Overview of iOS Graphics**

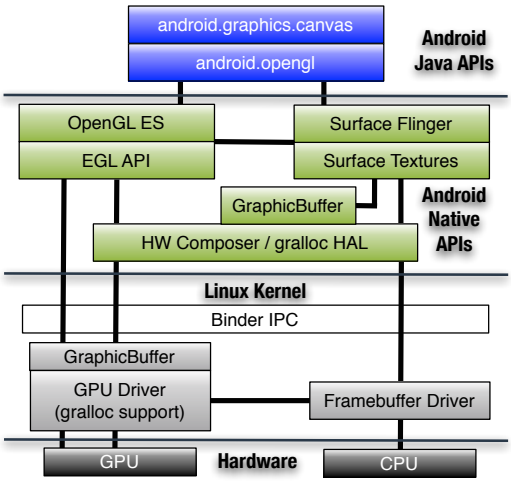


**Figure 2: Overview of Android Graphics**

## 2 IOS AND ANDROID GRAPHICS OVERVIEW

Modern graphics subsystems can be broken into three major components: rendering or drawing, display and window management, and memory management. On mobile platforms such as iOS and Android, the most widely-used subsystem for GPU-accelerated graphics is often loosely referred to as GLES. However, the GLES API [29, 31] is more properly thought of as a rendering, or drawing, API. GLES takes no responsibility for display and window management. To bridge between the rendered output of GLES and what is shown, GLES relies on the Embedded-System Graphics Library (EGL) API. A native window API such as EGL can be thought of as the canvas GLES draws on. GLES/EGL objects require memory to store graphics state. The memory management, done by the OS, usually involves a separate OS-specific API, allowing the resulting memory objects to be efficiently shared between apps or between different drawing APIs, such as OpenGL and non-OpenGL APIs.

To understand how GLES is supported in iOS and Android, we first review some basics. A GLES *object* is an opaque structure that refers to memory objects which store graphics data (e.g. renderbuffers, framebuffers, and textures). A GLES *context* is a state container for all GLES objects associated with a given instance of GLES. When a thread calls a GLES function, the function is called in a GLES context to manipulate a GLES object. A thread can create many GLES contexts. Because there are multiple versions of GLES which have different characteristics and are not compatible with each other, an *EGL context*, created with the EGL native window management API, defines the rendering API version used, and therefore the set of GLES functions that can be used within that EGL context.

Figure 1 provides an overview of the major graphics components in iOS. iOS apps use user space libraries such as UIKit and CoreAnimation to render user content, such as buttons, text, and images, using CoreGraphics and GLES system libraries. These system libraries communicate directly to the iOS kernel via opaque Mach IPC calls, and use I/O Kit drivers to allocate and share graphics memory, control hardware facilities such as frame rates and subsystem power, and perform complex rendering tasks such as those required for 3D graphics. For 3D rendering and drawing, iOS uses the standard GLES API. For display and window management, Apple devised a non-standard native window API, Embedded Apple GL (*EAGL*). The iOS GLES library renders content into a window or display managed by the EAGL library. For memory management, all graphics memory is allocated and manipulated through the IOSurface API which communicates via opaque Mach IPC messages to the IOCoreSurface I/O Kit driver. An IOSurface object is a memory abstraction that facilitates zero-copy transfers of large graphics buffers between apps and rendering APIs. Most GLES objects, such as textures, reference IOSurface memory objects for storing graphics data. For 2D graphics, the CoreGraphics or QuartzCore APIs are used to draw directly into IOSurfaces. IOSurfaces from both CoreGraphics and GLES are composited together using the IOMobileFramebuffer kernel driver, again accessed as an I/O Kit driver via opaque Mach IPC calls. These calls are IPC messages where both client and server ends of the communication hide or obfuscate the details of messages passed.

Figure 2 provides an overview of the major graphics components in Android. Android Java apps use the android.graphics.canvas and android.opengl APIs to render both 2D and 3D graphics. These Java APIs make extensive use of Java native calls to system libraries which communicate to the Android Linux kernel via opaque `ioctl`s and Binder IPC. Opaque `ioctl`s are `ioctl` system calls on a proprietary driver where both the command and the arguments are intentionally obfuscated or hidden creating an opaque interface into the kernel. For rendering and drawing, Android uses the standard GLES API. All 3D drawing is done via GLES, and as of Android 4.0, all 2D drawing is also accelerated by GLES [47]. For display and window management, Android uses the Khronos standardized EGL [30] API, unlike the proprietary EAGL used by iOS. The Android GLES library renders content into a window or display managed by the EGL library. For memory management, all graphics memory is managed by the GraphicBuffer API, and allocated through the HW Composer or gralloc APIs which use non-standard, often opaque, Linux kernel driver interfaces. Similar to iOS, Android GraphicBuffer objects facilitate cross-process and cross-API

zero-copy memory transfers. Unlike their IOSurface counterparts in iOS, GraphicBuffer objects are a much lower-level abstraction managed by the Surface Texture API. Surface Textures are used by both 2D and 3D drawing APIs, and are composited together by the Surface Flinger which uses the HW Composer API and Linux kernel framebuffer driver.

## 3 *CYCADA* GRAPHICS ARCHITECTURE

Graphics binary compatibility support in Android for iOS is a key challenge because the graphics subsystems in both OSes are driven by closed-source libraries that discard all abstractions and communicate directly with kernel drivers through opaque, undocumented Mach IPC calls and `ioctl`s, which in turn control complex, black-box pieces of hardware. Given the tight coupling of user space libraries to opaque, undocumented kernel APIs, rewriting any libraries or drivers, or emulating hardware would be at best an enormous and difficult reverse engineering effort attempting to keep up with product development cycles of large companies. Alternatively, any attempt to interpose on the kernel ABI would be useless without understanding the driver-specific `ioctl` commands or opaque Mach IPC messages.

At a high level, *Cycada* addresses this problem by leveraging the fact that the GLES standard is used across mobile platforms. Loosely speaking, instead of having iOS apps use their own iOS GLES libraries, *Cycada* has them use Android GLES libraries through diplomats [4]. Diplomats allow foreign apps to use domestic libraries to access proprietary software and hardware interfaces on the device. In *Cycada*, a thread has two personas, or execution modes, a foreign one for executing foreign code with a foreign kernel ABI (iOS) and a domestic one for executing domestic code with a domestic kernel ABI (Android). A diplomat function temporarily switches the persona of a calling thread to execute domestic code in a foreign app, or vice-versa. Using diplomats, *Cycada* replaces calls into foreign hardware-managing libraries (e.g. GLES) with calls into domestic libraries managing domestic GPU hardware. Each diplomat maps iOS functionality onto equivalent Android functionality.

We extend the basic *Cycada* diplomat construction to include a *prelude* and *postlude* operation in the context of the foreign persona. Before our extended diplomats switch the persona of the calling thread, they invoke a prelude function that executes in the foreign persona. After invoking the domestic function and switching the persona of the calling thread back to the foreign persona, our new diplomats invoke a postlude function in the context of the foreign persona. These prelude and postlude functions allow *Cycada* to support the necessary multiplexing of multiple loaded instances of a single Android graphics library (Section 8).

The complete process of calling a domestic function from foreign code through a diplomat is: **(1)** Upon first invocation, a diplomat loads the appropriate domestic library and locates the required entry point (function), storing a pointer to the function in a locally-scoped static variable for efficient reuse. **(2)** A prelude function executes foreign code using the foreign persona. This function is common to all diplomats and specified at compile time. **(3)** Arguments to the domestic function call are stored on the stack. **(4)** A new `set_persona` system call is invoked from the foreign persona to switch the calling thread's kernel ABI and TLS area pointer to their domestic persona values. **(5)** Arguments to the domestic function call are restored from the stack. **(6)** The domestic function call is directly invoked through the symbol stored in step 1. **(7)** Upon return from the domestic function, the return value is saved on the stack. **(8)** The `set_persona` syscall is invoked from the domestic persona to switch the kernel ABI and TLS area pointer back to their foreign persona values. **(9)** Any domestic TLS values, such as `errno`, are appropriately converted and updated in the foreign TLS area. **(10)** A postlude function executes foreign code using the foreign persona based on the foreign library being replaced. This function is common to all diplomats and specified at compile time. **(11)** The domestic function's return value is restored from the stack, and control is returned to the calling foreign function.

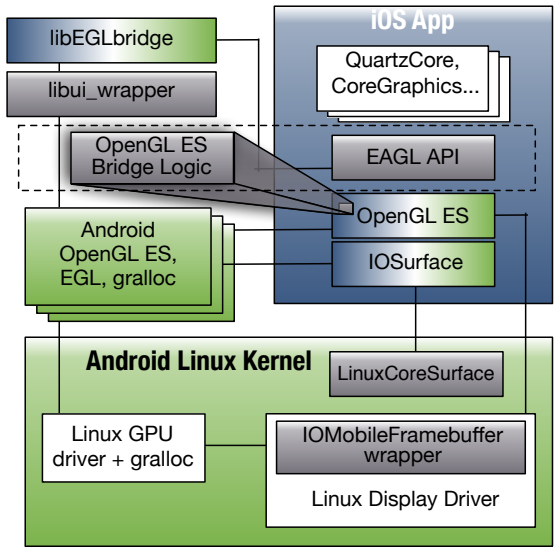


**Figure 3: Cycada iOS Graphics Compatibility**

Using diplomat usage patterns, thread impersonation, and dynamic library replication, we complete the *Cycada* graphics compatibility architecture to run unmodified iOS binaries on Android. Figure 3 depicts the components of this architecture. Components shown in grey represent new *Cycada* code, components in blue represent iOS code, and components in green represent Android code. Components containing both blue and green represent libraries containing diplomats. The dotted lines show components which have been enhanced and extended from the initial *Cycada* architecture [4]. Components can be loosely grouped based on graphics compatibility functionality they provide, and we discuss our new OS compatibility techniques in this logical order. Section 4 discusses iOS GLES support implemented by the diplomatic GLES library which includes GLES Bridge Logic to support *indirect* and *data-dependent* diplomats. Section 5 discusses iOS display and window management API support through a re-implemented iOS EAGL API that leverages *multi* diplomats composed in the diplomatic libEGLbridge library. Section 6 discusses iOS graphics memory management support which is implemented using a diplomatic IOSurface library and LinuxCoreSurface, a reimplementation of the iOS kernel framework, IOCoreSurface. Section 7 discusses *Cycada*'s multi-threaded iOS GLES support using thread impersonation.

| OpenGL ES | iOS | Android | Khronos |
|---|---|---|---|
| 1.0 Standard Functions | 145 | 145 | 145 |
| 2.0 Standard Functions | 142 | 142 | 142 |
| Extension Functions | 94 | 42 | 285 |
| Common Extension Functions | 27 | 27 | - |
| Extensions | 50 | 60 | 174 |
| Extensions not in Android | 33 | 0 | - |
| Extensions not in iOS | 0 | 43 | - |

**Table 1: OpenGL ES Implementation Breakdown**

## 4 GLES

GLES is the rendering, or drawing, API used by both iOS and Android, and its specification has been standardized by the Khronos Group, so at first glance, it seems straightforward to simply replace iOS GLES standard C-function symbols with diplomats that call into the Android GLES library to run iOS apps. However, the GLES standard is intended to be extensible [28], and vendor library implementations are free to implement available or even new extensions. The set of available extensions depends on both the GPU hardware and the vendor library used. Because Apple provides both the vendor library and GPU hardware for iOS platforms, iOS GLES implements a similar set of extensions across all iOS platforms of a given generation. However, Android runs on a multitude of platforms provided by many different manufacturers, so the GLES extensions implemented can vary and depend on the particular vendor library and GPU hardware.

Table 1 gives a summary of standard and extension GLES functions implemented in iOS and Android, as well as the total numbers reported by Khronos. The Android function list comes from a Nexus 7 tablet with an NVIDIA Tegra 3 GPU. We focus on GLES v1 and v2 standard functions, as GLES v3 is only supported by a minority of both iOS [6] and Android [48] devices. Additionally, the table only considers GLES functions added by extensions, not functionality added to existing functions. Table 1 shows that iOS and Android implement the complete set of GLES standard functions, but differ in extensions and extension functions they implement. Therefore, there is no possible one-to-one mapping from iOS to Android GLES functions.

### 4.1 Diplomat Usage Patterns

To support the complete set of iOS GLES functions, including extensions, on Android, we taxonomize diplomat usage based on common patterns uncovered through our study of iOS and Android graphics. Similar to the Gang of Four's design patterns [22], our diplomat usage patterns allow *Cycada* to quickly identify recurring solutions to binary compatibility problems. We formalize four diplomat usage patterns, *direct*, *indirect*, *data-dependent*, and *multi*, that together provide the management of resources necessary to bridge the intricacies of two mis-matched APIs.

First, standard GLES functions not augmented by extensions can be implemented using our extended *Cycada* diplomats. We refer to these functions as *direct* diplomats, which use the procedure from Section 3 to directly invoke a corresponding Android function. For iOS functions where direct invocation of an Android function is not possible, we introduce indirect, data-dependent, and multi diplomats.

| Type of Support | Functions |
|---|---|
| Direct Diplomats | 312 |
| Indirect Diplomats | 15 |
| Data-dependent Diplomats | 5 |
| Multi-Diplomats | 2 |
| Unimplemented (never called) | 10 |
| Total | 344 |

**Table 2: *Cycada* iOS OpenGL ES Support Breakdown**

An indirect diplomat uses a small amount of wrapper code around a standard diplomat. The wrapper runs in the foreign, iOS, context and can re-direct APIs to similar Android APIs with different names, or can manipulate input data to match an existing Android implementation. For example, APPLE_fence [23] is an extension implemented in iOS but not in Android. *Cycada* supports this extension using an indirect diplomat that maps APPLE_fence APIs to a similar extension, NV_fence [27], present on the NVIDIA Nexus 7 tablet. The custom iOS code performs minor input re-arranging within each APPLE_fence API before calling into a corresponding Android GLES NV_fence API.

Data-dependent diplomats augment standard diplomats by performing input-dependent logic or implementation before optionally calling the Android function. For instance, if an iOS extension adds the ability to render a new pixel format which is unsupported in Android, GLES functions that allocate or manipulate textures would need data-dependent diplomats that can understand the iOS texture format and manipulate it into a form understood by Android functions. For example, the standard GLES `glGetString` function in iOS has been modified by Apple to accept a non-standard parameter name, unknown in Android. That parameter name is intended to return Apple–proprietary extensions available on the platform. *Cycada* uses a data-dependent `glGetString` diplomat that interprets the input parameter and either calls the Android function, or returns a custom string indicating that no Apple-proprietary extensions are available. Some data-dependent diplomats may not invoke an Android function at all due to a lack of corresponding Android functionality. For example, the APPLE_row_bytes [26] extension handles two extra parameters to the `glPixelStorei` function and maintains state associated with the current GLES context which controls how three GLES functions read in or write out pixel data. These three GLES functions are implemented using data-dependent diplomats such that when the APPLE_row_bytes extension is being used, *Cycada* reads in and writes out the packed data manually.

Finally, multi diplomats are necessary when iOS functions or extensions do not map cleanly to a single Android function, and the behavior is too complex for custom logic. These diplomats leverage several different Android library functions through two or more coalesced diplomats. *Cycada* uses multi diplomats to implement window and memory management functionality discussed in Sections 5 and 6.

Table 2 indicates how effective our diplomat usage patterns are in supporting iOS GLES on Android. The majority of GLES functions are supported via direct diplomats, 20 GLES functions are supported via indirect or data-dependent diplomats, and two GLES functions require multi diplomats. While indirect diplomats are simple, data-dependent diplomats can, in some cases, require

a hundred lines of additional code, and multi diplomats involve the most implementation complexity. No GLES functions need to be completely reimplemented. Ten iOS GLES functions are not implemented in our prototype because they are never called. Note that the total number of functions in Table 2 does not match Table 1 because the numbers in Table 1 are not mutually exclusive since some GLES v1 and v2 standard functions are the same. This table shows that our diplomat usage patterns successfully bridge the gap between iOS and Android GLES APIs.

## 5 EAGL

Graphics resource management, including display and window management, is done in iOS using Apple's own EAGL Objective-C API, but in Android using Khronos standardized EGL API. There is no direct mapping from EAGL to EGL, requiring *Cycada* to implement substantial logic to support EAGL. However, the Android EGL library performs conceptually similar functions as EAGL, and the EAGL API is small. Thus, it is possible to construct an EAGL implementation from a combination of Android EGL and GLES libraries using multi diplomats with a modest amount of custom logic. For efficiency, we coalesced our multi diplomats into an Android library called *libEGLbridge* (Figure 3). This allows us to pay the overhead of one diplomat which calls into a custom Android API that uses standard Android functions and libraries to perform the required function.

The EAGL API consists of only 17 Objective-C methods. Six methods were supported using multi diplomats, 10 required implementation from scratch, and 1 was not implemented as it was never called. The 10 EAGL functions implemented from scratch required less than 30 lines of code in total. In contrast, the methods supported by the more complicated multi diplomats required approximately 5000 lines of code to implement the 6 methods.

To provide clearer insight into how multi diplomats were used to implement EAGL methods, we now describe an example under *Cycada*. Care must be taken when rendering using GLES APIs. The memory backing the render target is accessible to the window management API and could potentially be displayed on the screen at any time. To prevent corrupt, or half-rendered output, window management APIs usually provide some method of double (or more) buffering the output. Double buffering allows the rendering APIs to draw into a memory buffer while the window management APIs send a different memory buffer out to the screen.

GLES uses an object called a *framebuffer* to represent the abstracted memory and render target provided by the window management API. The first, or default, framebuffer always represents the display screen or window area. The standardized EGL window management API uses the function `eglSwapBuffers` to swap the rendering target of the default framebuffer between a "front" buffer (the GLES render target) and a "back" buffer (the memory sent out to the screen or window). In contrast, EAGL API only allows rendering to an off-screen (non-default) framebuffer. When frame rendering is complete, the programmer must call the `presentRenderbuffer` function that copies the off-screen framebuffer into the display screen or window area.

Since EAGL does not use the default framebuffer, the standard Android `eglSwapBuffers` will not work to transfer rendered frame data to the screen or window memory; the data was never put into the default framebuffer's memory. To display the contents of an off-screen framebuffer into which an iOS app has rendered content, *Cycada* implements the EAGL `presentRenderbuffer` function using a multi diplomat. This diplomat uses simple GLES vertex and fragment shader programs, via Android GLES APIs, to render the off-screen framebuffer contents into the default framebuffer. From the default framebuffer, *Cycada* can use `eglSwapBuffers` to display the content. The current *Cycada* prototype uses this inefficient implementation, but it could be improved through more complicated management of underlying graphics memory or the Android EGL/GLES libraries.

## 6 MEMORY MANAGEMENT

The massive size, low latency requirements, and cross–process composition of graphics objects requires an efficient, zero–copy mechanism that allows graphics memory to be shared between libraries and apps. iOS uses *IOSurface* objects for efficient graphics memory management. Kernel-level IOSurface support (IOCoreSurface in Figure 1), provides the zero-copy support which allows IOSurface objects to be efficiently passed between libraries and apps. IOCoreSurface kernel APIs and functionality were reverse engineered. The resulting module is shown as *LinuxCoreSurface* within the Android Linux kernel in Figure 3.

Android manages efficient graphics memory transfers using *GraphicBuffer* objects. While these objects perform the same high-level functions as iOS IOSurface objects, the IOSurface API is more complicated and offers a richer interface for manipulating, sharing, and remapping graphics memory. *Cycada* must therefore provide a mapping between IOSurfaces and GraphicBuffers for GLES to function correctly. In the following subsections, we discuss two key aspects of IOSurfaces and how they are supported in *Cycada*.

### 6.1 IOSurface Life Cycle Management

IOSurfaces are created using `IOSurfaceCreate`, which allocates the necessary memory buffer, and connects the allocated region to the supporting kernel infrastructure. To provide the necessary kernel support for advanced IOSurface memory operations, *Cycada* interposes on `IOSurfaceCreate` using an indirect diplomat to create an Android GraphicBuffer object as the underlying backing graphics memory for an IOSurface. Similarly, as the created IOSurface is associated with GLES textures, or other library objects, *Cycada* uses indirect diplomats to interpose Android GraphicBuffer management. For example, *Cycada* interposes on the `glDeleteTextures` API and removes any corresponding connection to the underlying Android GraphicBuffer.

### 6.2 Cross-API Object Sharing

An IOSurface, much like its GraphicBuffer counterpart, can be used by 3D as well as other 2D rendering APIs. These 2D APIs, such as CoreGraphics, use the CPU to draw directly into IOSurfaces as opposed to sending commands to the GPU to render content into the memory. To allow 2D and 3D APIs to share IOSurfaces, iOS provides the `IOSurfaceLock` and `IOSurfaceUnlock` functions to lock and unlock an IOSurface for CPU-only access, during which time the GPU may not access it. The Android GraphicBuffer object

can be locked for CPU-only access unless it has been associated with a GLES texture (via an EGLImage). As discussed in Section 5, iOS associates IOSurfaces, and thus GraphicBuffers, with GLES textures for 3D drawing. Thus the exact scenario in which we need to lock a GraphicBuffer to support IOSurfaces is unsupported by the Android API.

To circumvent this Android limitation, *Cycada* interposes on the `IOSurfaceLock` and `IOSurfaceUnlock` functions with multi diplomats. When an IOSurface is locked, *Cycada* disassociates the Android GraphicBuffer from the connected GLES texture allowing it to be locked for CPU-only access. However, this process is non-trivial, and unsupported by current Android GLES and EGL APIs. A GLES texture is required to be associated with some memory object, so while the IOSurface is locked for CPU access the *Cycada* multi diplomat rebinds the GLES texture to a single-pixel buffer allocated by `glTexImage2D`. The multi diplomat can then destroy the EGLImage object associated with the texture which implicitly disassociates the Android GraphicBuffer. At this point, the GraphicBuffer can be locked for CPU access. By assuming correct IOSurface locking behavior of the iOS app, we know that no Open GL function calls will occur that will try to render using the texture.

*Cycada* also interposes on `IOSurfaceUnlock` with another multi diplomat. We create a new EGLImage object and rebind it, and the GraphicBuffer, back to the GLES texture. Since GLES did not have access to the IOSurface (or GraphicBuffer) while it was locked, the disassociation and re-association process is transparent to iOS's GLES.

# 7 MULTI-THREADED GLES

GLES and EGL are used in heavily multi-threaded environments, however there are some restrictions based on the specifications. First, GLES functions are not thread safe, so apps are expected to synchronize access to GLES state outside of the GLES API. Second, an EGLContext object defines the set of GLES functions available, and it is possible for a standards-compliant EGL implementation to allow only a single context to be created per thread group [30]. Thus, the accepted standard for multi-threaded apps using GLES/EGL is to use a single thread dedicated for rendering.

iOS and Android are both heavily multi-threaded environments, but differ in the level of GLES threading support. iOS allows any thread to use a GLES context; one thread can create a GLES context and another can use it. Apple's Grand Central Dispatch (GCD) is used heavily and relies on this feature to asynchronously dispatch GLES jobs such as texture loading or off-screen rendering. Each thread in the system has its own context, and implicitly takes on the GLES and EAGL context of the thread that submitted the asynchronous job. Similarly, the iOS WebKit library spawns a rendering thread that allocates and initializes its own GLES context which is used by other threads related to WebKit. This level of multi-threaded support does not exist in Android GLES or EGL libraries, which only allow a GLES context to be used by a thread if it or its thread group leader created the context. In other words, a GLES context created by Android thread 1 could not be used by Android thread 2 unless thread 1 also happened to be the "main" thread.

## 7.1 Thread Impersonation

To support multi-threaded iOS GLES apps on Android, *Cycada* introduces thread impersonation. Thread impersonation allows thread-specific context to be shared among multiple threads enabling one thread to temporarily assume the persona of another thread. While more limited forms of this impersonation have been used in security contexts, *Cycada* thread impersonation presents a generalized mechanism to impersonate a thread across *all* personas in which the thread may execute. In these personas, a subset of thread-specific data may be used or shared by the impersonating thread.

In *Cycada*, iOS threads attempting to perform GLES operations will impersonate the Android thread that created an Android GLES context. We refer to the Android thread which created the GLES context as the target thread, and the iOS thread that invoked a GLES function the running thread.

Diplomats invoked by the running thread will leverage the prelude and postlude functions to migrate thread local data between the target and running threads. Android and iOS TLS state must be migrated because the target thread created the GLES context through a diplomat. Thus the target thread has both platforms' graphics state in their TLS.

However, not all data in TLS needs to be migrated. TLS is an array of void pointers unique to each persona of thread. Each array entry is a slot. Some TLS slots are reserved for system use for things such as a thread-local `errno` value, but apps can reserve other slots using the `pthread_key_create` function, which returns a globally-unique TLS slot ID. A given thread passes the returned slot ID into the `pthread_getspecific` or `pthread_setspecific` functions to get or set a thread-local, or thread-private, value. *Cycada* thread impersonation allows selective migration of TLS data by modifying Android's libc to send out a notification whenever a new TLS key is reserved through `pthread_key_create` and destroyed through `pthread_key_delete`; this is a trivial 12 line patch. By registering for a hook that is invoked on every `pthread_key_create` and `pthread_key_delete` call, we can selectively monitor TLS slot allocation.

Because *Cycada* migrates graphics contexts, we monitor graphics-specific TLS slot allocations by gating the Android `pthread_key_create` and `pthread_key_delete` hooks in the prelude and postlude of each graphics diplomat. This migrates only the graphics-relevant TLS data between the target and running thread's Android personas. We also migrate well-known iOS TLS slots used by Apple graphics libraries. Since vendor graphics libraries, along with their TLS slots, are opaque, we can assume that the TLS slots they reserve are not used by any other subsystems.

Formally, *Cycada* thread impersonation for graphics is done as follows: **(1)***Cycada* identifies graphics-related TLS state using `pthread_key_create` and `pthread_key_delete` Android libc hooks. **(2)**Whenever a GLES context is created or modified, *Cycada* ties it to the graphics-related TLS of the thread that created the GLES context. **(3)**Whenever a thread calls a GLES function using a GLES context it did not create, *Cycada* saves the running thread's graphics-related TLS state, in both its iOS and Android personas, and replaces it with TLS data associated with the GLES context. **(4)**Updates are made to the TLS values as needed as the thread executes graphics functions,

and these updates are reflected back into the TLS associated with the GLES context. **(5)**On GLES function return, *Cycada* restores the running thread's original graphics-related TLS state.

In *Cycada*, a thread has both an iOS and an Android persona, each with its own TLS. *Cycada* must ensure that graphics-related use of TLS in the iOS persona matches what is expected by iOS apps. Generally, this is done by relying on iOS libraries to manipulate the TLS in the iOS persona without any additional work by *Cycada*. However, when a thread submits an asynchronous job to GCD, *Cycada* must associate the iOS TLS data of the submitting thread with the EAGL context so that when the GCD job is run on a different thread, that thread's iOS TLS can be properly updated.

With diplomats, *Cycada* must ensure thread migration in Android is done to match the necessary iOS GLES and EAGL contexts. For example, if thread A passes its context to thread B before calling a diplomat, thread B must impersonate thread A in both iOS *and* Android. However, iOS and Android use separate TLS areas for execution, and only the kernel has knowledge of both TLS areas. Thus to effect thread migration, *Cycada* introduces two system calls. The `locate_tls` syscall can extract TLS values from any given persona in which a thread has executed. Similarly, the `propagate_tls` syscall pushes TLS values into any given persona. Using these two syscalls, *Cycada* ensures proper GLES functionality across multi-persona thread migration.

The GLES specification requires external thread synchronization. By assuming that an iOS app/framework correctly synchronizes calls to GLES functions, *Cycada* can guarantee that calls to the Android GLES library, through diplomats, will be properly synchronized.

## 8 EAGL MULTI-CONTEXT SUPPORT

The iOS EAGL library can instantiate multiple *EAGLContext* objects, each with their own GLES connection. Each GLES connection can use a different API version. For example, consider an iOS game with an initial menu interface, and a UIWebView to render an HTML "about" page. The UIWebView class uses the WebKit library to render HTML which implicitly creates its own EAGLContext object connected to GLES API v2. If the game uses GLES, it will create its own EAGLContext object with its own connection to the GLES API. The iOS app is free to use either GLES v2 *or* v1. If the game uses GLES v1, the process has now instantiated two unique EAGLContext objects each using a different version of the GLES API. There is no EGL or Android mechanism to support this paradigm. Only a single EGL connection to a single GLES API version can be made per-process.

### 8.1 Dynamic Library Replication

To support multiple EAGLContexts in a single process, we introduce *dynamic library replication* (DLR). At a high level, our solution reloads and re-initializes, or re-instances, all the Android graphics libraries. Each new library instance, or *replica*, is loaded and linked as if no other libraries have been loaded.[1] This causes each replica to occupy its own virtual memory space, and invoke its own pseudo-private copies of all library functions and their dependencies. A replica is a library namespace which includes all dependent libraries. For example, the NVIDIA graphics support library, *libGLESv2_tegra.so* requires the *libnvrm.so* library which requires the *libnvos.so* library. Each replica of the *libGLESv2_tegra.so* library would occupy its own virtual address space *and* it would also link against privately loaded copies of all required libraries such as *libnvrm.so* and *libnvos.so*.

DLR is needed because of Android's EGL implementation, which can be broken into two pieces: an open source library exporting all the standardized EGL functions, and a vendor-provided, device-specific EGL implementation. Android apps link against the open source library, and when an app initializes the EGL interface using the `eglInitialize` function, the open source library loads the vendor-specific EGL and GLES libraries. The restriction of a single EGL-to-GLES connection per process is seemingly arbitrary, but enforced by both vendor and open source libraries.

Maintaining per-thread EGL connection information only partially solves the problem. While individual threads in a process can separately initialize and maintain EGL-to-GLES connections through the open source library, the vendor provided EGL and GLES libraries are proprietary and closed source and assume a single, process-wide EGL connection.

To bypass arbitrary vendor restrictions on singleton EGL connections, *Cycada* uses DLR - a dynamic linker mechanism that loads multiple, independent instances (replicas) of a dynamic library. Normally, on a call to `dlopen` the linker will not re-initialize or reload a library if it's already loaded. The linker will return a handle to the previously loaded instance. The DLR-enabled linker introduces a new function, `dlforce`, which opens a library (the replica), and all its dependencies, as if they were never loaded before. The replica and its dependencies will have unique virtual addresses, and all of their library constructors will be called. The linker keeps track of each replica, and the same `dlforce` function can be used to modify the behavior of other linker functions such as `dlsym` and `dlopen` to search only those libraries loaded from the given `dlforce` handle. This allows library code within a replica, or its dependencies, to use the dynamic loader normally, creating isolated trees of libraries.

*8.1.1 EGL Extension:* `multi_context`. *Cycada* uses DLR in the Android EGL open source library through a custom EGL extension named *EGL_multi_context* and a supporting library, *libui_wrapper* (Figure 3). This extension API, shown in Figure 4, adds four new EGL functions for creating and manipulating EGLContext objects that maintain isolated, unique GLES connections within the same process. The libui_wrapper library links against the vendor GLES and EGL libraries and encapsulates other Android system libraries which implicitly link against GLES or EGL. The `eglReInitializeMC` function creates a replica of the vendor EGL and GLES libraries. The `eglSwitchMC` function allows a thread to select which replica, and thus which GLES connection, it will use by setting the thread's EGLContext object to the one contained within the replica.

Creating EGL and GLES replicas, through a modified Android open source EGL library, results in unique GLES connection management challenges related to TLS. The unmodified Android EGL library allows one EGL-to-GLES connection (*EGLConnection* object) per-process, and it stores this information in a library-static global variable. Creating replicas of the vendor EGL and GLES libraries

[1] We do *not* reload libc; all lib. instances use a single, shared libc instance.

```
EGLBoolean eglReInitializeMC(EGLNativeDisplayType display,
                             EGLDisplay *dpy,
                             EGLint *major, EGLint *minor);
EGLBoolean eglSwitchMC(EGLContext new_ctx, EGLContext old_ctx);
EGLBoolean eglGetTLSMC(void **tls_vals, int nvals);
EGLBoolean eglSetTLSMC(void **tls_vals, int nvals);
```

**Figure 4: Custom EGL Extension: *EGL_multi_context***

allows multiple threads to use different EGLConnections concurrently. A single, global EGLConnection variable no longer suffices, so *EGL_multi_context* stores this per-thread EGLConnection object in the TLS. A common paradigm in GLES programming is to create a context on one thread (generally the main thread), and pass the context information to another thread that performs rendering or texture loading functions. Because the *EGL_multi_context* extension has moved the previously global GLES connection information into a thread-local variable, we require the ability to copy, or *migrate*, TLS values between threads. This is accomplished using thread impersonation with `eglGetTLSMC`/`eglSetTLSMC` extension functions.

## 8.2 Unintended Consequences

Moving data from a global location into thread-specific variables, and creating multiple copies of the same library can have unintended consequences. *Cycada*'s EAGL implementation relies on a custom library, *libEGLbridge* (Section 5), that provides targeted Android functionality through a set of diplomats. This library uses Android GraphicBuffer objects which use APIs from vendor-proprietary libraries that link against the libraries used by the vendor-proprietary EGL/GLES libraries. The mechanism by which GraphicBuffer memory is shared between APIs requires that those APIs use the *same* GLES connection as the GLES rendering functions. In other words, a GraphicBuffer allocated by libEGLbridge using the first instantiation of the EGL and GLES libraries cannot be used by GLES functions in replicas created by multiple iOS EAGLContexts.

To avoid the library dependencies morass, *Cycada* separates the libEGLbridge functionality into two pieces. The first piece contains all the diplomats used by the iOS code, and avoids linking against libraries. The second piece, "libui_wrapper," contains all of the logic that links against Android graphics libraries. When a new EAGLContext object is created, a diplomat in libEGLbridge creates a replica of the libui_wrapper library and the EGL/GLES libraries by using the prelude functionality of diplomats. That way, the libui_-wrapper functionality uses the same replica of GLES as the gralloc functions allocating a GraphicBuffer.

# 9 EVALUATION

We have extended the *Cycada* prototype [4], to more completely bridge the differences between iOS and Android graphics subsystems. Our extended prototype runs widely used iOS apps on Android, including those that make extensive use of WebKit, such as the Apple-only apps Safari and iBooks, and third-party apps Yelp, Holy Bible, and Wikipedia. Support for other devices such as GPS and networking is also completely functional, so apps such as Yelp can be used to find local restaurants. We present some experimental results that both demonstrate the feasibility of our approach and measure its performance using both app-level benchmarks and targeted micro-benchmarks. We used *Cycada* on a Nexus 7 tablet with Android 4.2.2, and compared its performance with an iPad mini running iOS 6.1.2.

**iOS WebKit Functionality:** We first ran the iOS Safari Web browser on *Cycada* to demonstrate its functionality. Safari is an excellent test app because it uses a wide range of complex graphics functionality via WebKit. While advanced graphics APIs, such as GLES, are often thought of exclusively in the context of games or third-party apps, GPUs and their associated APIs are also used to accelerate many different aspects of computation. For example, WebKit uses CoreImage, QuartzCore, CoreGraphics, and IOSurface libraries in iOS which together use GLES to accelerate image and graphics processing. Additionally, the deep vertical integration of iOS allows library designers, as opposed to third-party developers, to bypass standard GLES extension query mechanisms, and make simplifying assumptions about available GLES extensions. Our prototype supports these implicit assumptions by mapping missing extension functionality onto existing Android GLES functions. Thus, running an app such as Safari in *Cycada* requires a near-complete GLES bridge implementation.

We performed two sets of experiments. First, we used Safari to browse the main page of the top 30 websites in the US [1] and compared the visual results of Safari running on *Cycada* on a Nexus 7 tablet with Safari running on an iPad mini. All top 30 websites rendered their content correctly and appeared visually similar to the respective content on the iPad mini. Second, we used Safari to run the Acid3 test [51], a Web test page from the Web Standards Project that checks browser compliance with Web standards, including Document Object Model (DOM) and JavaScript. Safari on *Cycada* passes the test, showing smooth animation, a score of 100/100, and having the final page look exactly, pixel for pixel, like the reference rendering.

**Performance:** We compared four different Android and iOS system configurations to measure the performance of *Cycada*: iOS app running on *Cycada* (*Cycada* iOS), Android app running on *Cycada* (*Cycada* Android), iOS app running on iOS (iOS), Android app running on Android. We normalize our results to the Android app running on Android. A Nexus 7 tablet with Android 4.2.2 was used in all cases except for iOS app on iOS which was run on an iPad mini. Both tablets were released around the same time frame and have a similar form factor, providing a useful point of comparison.

We first ran a simple set of micro benchmarks using the `lmbench` test suite to measure the raw overhead of using diplomats. A diplomat involves two system calls: one to switch the thread from the iOS persona to Android persona, and one to switch back. We first

| Null Syscall | |
|---|---|
| **System** | **Time** |
| Stock Android | 225 ns |
| *Cycada* Android | 244 ns |
| *Cycada* iOS | 305 ns |
| iPad mini iOS | 575 ns |

| Diplomatic Calls | |
|---|---|
| **Function Call** | **Time** |
| Standard Function | 9 ns |
| Diplomat | 816 ns |
| Diplomat +Pre/Post | 828 ns |
| Diplomat + GL Pre/Post | 933 ns |

**Table 3: Kernel-level / ABI Micro-Benchmarks**

ran the null system call `lmbench` micro-benchmark which invokes system calls that perform no work within the kernel. Using *Cycada*, we then ran a custom micro-benchmark using the `lmbench` infrastructure that measures the time to invoke a standard iOS function, a diplomat with no prelude or postlude, a diplomat with an empty prelude and postlude, and a diplomat using the *Cycada* GLES prelude and postlude functions.

Table 3 shows the results of our kernel and ABI micro-benchmarks. For all tests run on the Nexus 7, the CPU was pinned to 1.3HGz. Since there is no interface to pin the CPU frequency in iOS, we used lmbench to "warm up" the caches, and ramp the CPU frequency to its maximum of 1GHz. The null syscall results on all configurations show that *Cycada* adds about 8% overhead to an Android kernel trap and 35% to an iOS trap due to an unoptimized kernel entry path. The iPad mini had a higher cost to trap into the kernel due primarily to the protection logic guarding against return-to-user attacks [8]. The diplomatic call results show that standard function calls are much faster than system calls and diplomats. The additional prelude and postlude mechanisms introduced by *Cycada* add little overhead to diplomats without them, and the fully functional GLES prelude and postlude cost roughly 100ns. A GLES diplomatic call costs almost the same as three system calls.

We then used two app benchmarks that could be run on both Android and iOS: a Web browser running SunSpider [5], and the PassMark [41, 42] app. SunSpider is a widely-used JavaScript benchmark that stresses many aspects of the browser's JavaScript engine including bit operations, cryptography, raytracing, JSON input, and pure math. We ran SunSpider using Safari on *Cycada* and iOS, and Chrome, the default browser, on *Cycada* and Android. PassMark is a freely available, cross-platform benchmark suite, and we used its 2D and 3D tests to measure graphics performance. We ran the iOS PassMark app on *Cycada* and iOS, and the Android PassMark app on *Cycada* and Android.

Figure 5 shows the SunSpider latency measurements normalized to the performance of the stock Android browser on Android. Lower numbers are better. The Android browser on *Cycada* and Safari on iOS perform similar to the stock Android browser on Android. However, Safari on *Cycada* is more than four times slower overall, and over ten times slower for "access" and "bitops" tests. This slowdown mostly results from a lack of Just-In-Time (JIT) compilation of JavaScript on *Cycada* due to a Mach VM memory bug in *Cycada* that prevents JIT from working properly. For comparison, the purple bar in Figure 5 shows results for disabling JIT for JavaScript on iOS, normalized to iOS performance. Disabling JIT results in a 4.2x slowdown on iOS relative to standard iOS. This is roughly equal to the 4.4x slowdown on *Cycada* with the additional overhead resulting from our unoptimized system call path (Table 3). The high overhead of running SunSpider without JIT, especially in the "regex" test, generally falls in line with measurements done by the WebKit team on the first introduction of ARM JIT [52], and subsequent speedups introduced by the data flow graph (DFG) JIT [20].[2]

Figure 7 shows a breakdown of the Android GLES functions called while running SunSpider in Safari on *Cycada*. Note that the SunSpider test itself does not invoke graphics functions, rather the WebKit framework uses GLES to render the resulting dynamic HTML output. We show the percentage of time consumed by each GLES function relative to the total time consumed by all GLES functions, with the functions ordered in descending order based on how much total time they consume in running the benchmark. We show the top 14 functions, which consume over 90% of the total time. Function names starting with *gl* correspond to direct, indirect, or data-dependent diplomats to Android GLES functions, names starting with *egl* correspond to multi diplomats to Android EGL functions, and names starting with *aegl* are custom multi diplomats, located in libEGLbridge, supporting the *Cycada* EAGL implementation (Section 5 and Figure 3). Approximately 40% of the graphics-related time is spent in EAGL implementation related functions such as `aegl_bridge_draw_fbo_tex`. Clearly there is room for improvement in our unoptimized prototype.

Figure 9 shows average execution time per function call for the same GLES functions in Figure 7. Of the top 14 functions, only 1 costs less than 10$\mu$s, and the most time consuming functions take, on average, over 300$\mu$s. Since Table 3 shows the cost of a diplomatic call is less than 1$\mu$s, the overhead due to diplomats is small for most GLES functions, and certainly small for GLES functions that account for most of the GLES execution time in running SunSpider in Safari. The dominant cost is in the graphics logic required to bridge between iOS and Android, not in the diplomat mechanism.

Figure 6 shows the PassMark 2D and 3D graphics measurements normalized to the performance of the stock Android app on Android. Higher numbers are better. This measurement generally matches the PassMark graphics measurements first reported in [4]. However, through the enhanced graphics support we described, and some preliminary optimizations of the prototype, *Cycada* now outperforms Android in the GPU-intensive complex 3D test by more than 20% running on the same Nexus 7 tablet. Some of this performance increase could also be attributed to slight variations in how the 3D scenes were rendered on each platform, or differences in the exact GLES calls made on either platform due to differences in supported GLES extensions.

Similar to results found in [4], all the PassMark graphics measurements show that *Cycada* iOS performance relative to Android is highly correlated to iOS performance relative to Android, even though *Cycada* is running on the Nexus 7 while iOS is running on the iPad mini. For the 2D tests in which stock iOS does significantly worse than stock Android, *Cycada* iOS also does significantly worse than *Cycada* Android. In the complex vectors and 3D tests in which stock iOS does noticeably better than stock Android, *Cycada* iOS also does noticeably better. The reason for this is that both *Cycada* and iOS use the same frameworks and libraries, which in some cases have better performance than Android and in some cases are worse. Comparing *Cycada* and iOS, we see that *Cycada* performs

[2]The optimizations in the WebKit FTL JIT article [20] were not implemented in the WebKit we use, however the article presents DFG JIT performance vs. baseline JIT which indicates more than 3x speedup vs. baseline and 30x vs. non-JIT on at least one representative benchmark.

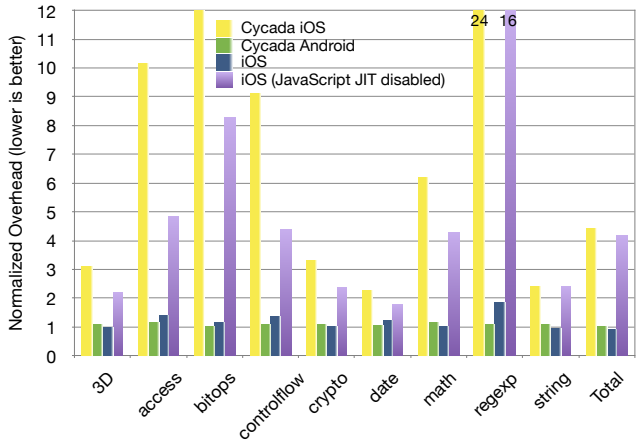


**Figure 5: SunSpider Benchmarks**

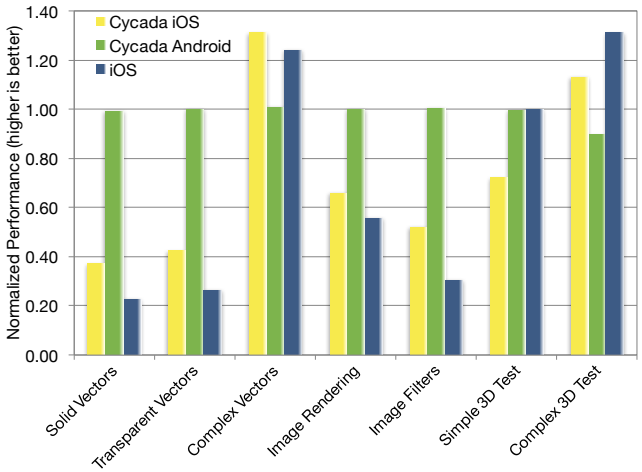


**Figure 6: PassMark Graphics Benchmarks**

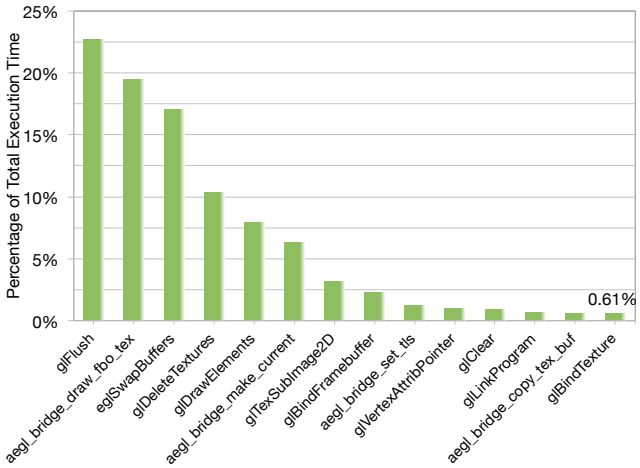


**Figure 7: SunSpider Total Time % per Function**

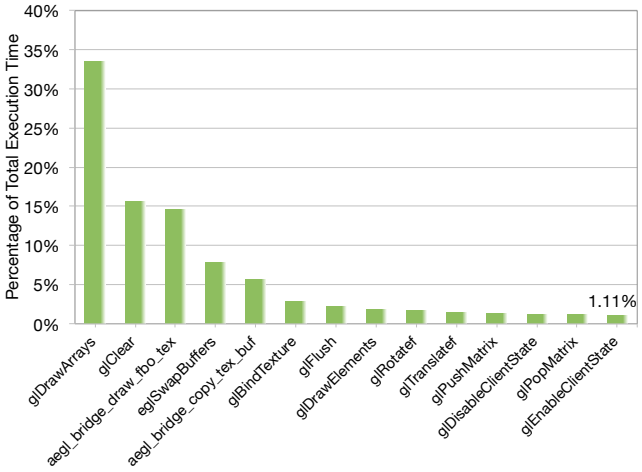


**Figure 8: PassMark Total Time % per Function**

better than iOS on the 2D tests and worse on the 3D tests. Because the 2D tests make heavier use of the CPU, *Cycada* outperforms iOS because it uses the faster Nexus 7 CPU versus the slower iPad mini CPU. Because the 3D tests are highly GPU intensive, they provide a better indicator of the graphics overhead of *Cycada* compared to iOS. The simple 3D test has higher overhead as it is designed to maximize frame-rate and thus stresses our unoptimized EAGL implementation which is responsible for moving rendered scenes onto the display. The complex 3D tests involve more processing intensive GPU functions to render complex scenery, so since the GLES calls used are more expensive, the overhead of *Cycada* is less.

Figures 8 and 10 show the percentage of total GLES execution time and average execution time per GLES function, ordered from the function with the largest to the smallest total execution time. We show the top 14 functions, which consume over 90% of the total time. The two most heavily used GLES functions are `glDrawArrays`, which draws an array of vertices, and `glClear`, which clears the framebuffer. Both are standard GLES functions, heavily used by the simple and complex 3D tests, called via direct diplomats. Based on their average execution time shown in Figure 10 relative to the cost of diplomats shown in Table 3, diplomat overhead is small. The primary overhead of *Cycada* is due to functions such as `aegl_bridge_draw_fbo_tex` and `aegl_bridge_copy_tex_buf`, which consume roughly 20% of the GLES execution time. These functions correspond to a highly optimized hardware supported path in iOS on the iPad mini.

## 10 RELATED WORK

Many different approaches have been proposed to run apps from multiple OSes on the same hardware [2, 9, 12, 15, 17–19, 21, 24, 25, 34, 35, 38, 39, 43]. However, as discussed [4], graphics support remains a challenge. Various approaches rely on the assumed ubiquity of X Windows, and assume that apps simply conform to the X Windows standard. This is insufficient to support GPU-intensive apps that rely on a wide range of graphics support, and is not applicable to the vertically integrated, proprietary graphics stacks common on mobile platforms in lieu of X. Wine, a Windows API reimplementation project for Linux, provides incomplete support

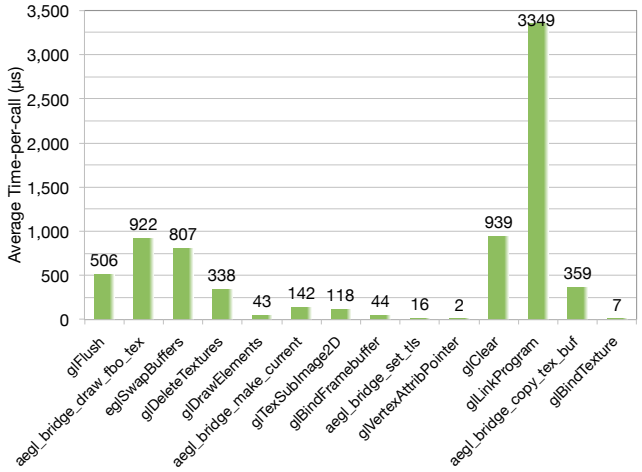


**Figure 9: SunSpider Average Time per Function**

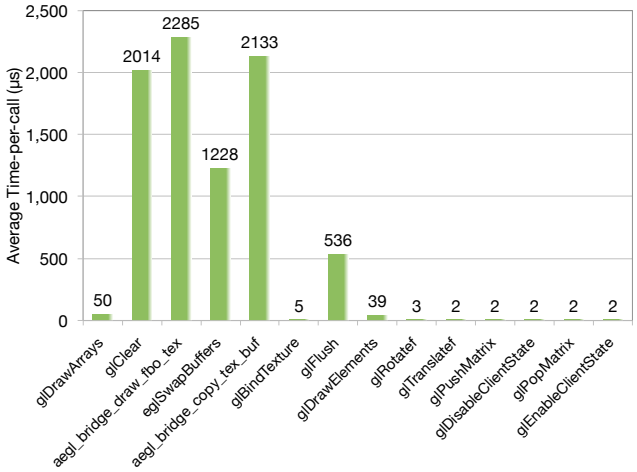


**Figure 10: PassMark Average Time per Function**

for Microsoft's advanced graphics API, DirectX, on top of Linux using OpenGL functions. This is in contrast to *Cycada* which leverages existing iOS frameworks and libraries instead of requiring a massive reimplementation effort.

Desktop virtualization solutions have developed several solutions that allow guest VMs to use GPU acceleration through mediated access to host hardware resources and libraries. These solutions can be grouped into four basic categories: API remoting or forwarding, device emulation, split-mode or mediated pass-through drivers, and direct pass-through. Desktop or server API remoting [45] ("indirect rendering") solutions, such as VirtualGL [49], rely on desktop windowing protocols such as GLX [53] to stream OpenGL commands to a remote server for execution. Mobile OSes such as iOS discard abstractions such as GLX in favor of custom APIs targeted for mobile usage patterns and low-latency direct hardware communication. Similarly, API forwarding solutions rely on being able to forward API calls to another platform because they share the same API. However, mobile OSes such as iOS use some standards, but not others, and the ones they use, they may extend, such that there is no longer a complete, shared API to forward from iOS to Android, making API forwarding or remoting problematic.

Device emulation is only used for simple, 2D hardware [10] due to the massive complexity of GPU hardware. The approach is more problematic on resource constrained mobile platforms such as iOS with proprietary interfaces that would be difficult to reverse engineer and emulate.

Split-mode, or mediated pass-through, solutions such as VMGL [33], VMware's vGPU [16, 46], and XenGT [50], take a hybrid approach of forwarding some aspects of the guest API, direct mapping some host GPU resources, and emulating other aspects of a graphics driver. The difficulty of emulation depends in part on the API supported. For example, VMware's approach focuses on Direct3D, which requires apps to perform their own graphics resource management and lacks the additional challenges of GLES support which requires EGL, or for iOS, a proprietary EAGL.

Direct pass-through solutions, such as NVIDIA GRID [37], leverage traditional hardware virtualization and require specialized hardware available in desktop-class GPUs [36]. They rely on hardware support not available in mobile platforms, and do not address the problems with providing graphics device support for mapping vertically integrated graphics stacks (e.g. iOS) to other platforms.

Although none of these approaches by themselves can support graphics device support across mobile platforms such as iOS and Android, many facets of *Cycada* may be adapted and applied in the context of these other approaches. For example, to enable mobile graphics virtualization, API forwarding could leverage some of the mechanisms introduced by *Cycada* to provide similar graphics support for VMs. The primary difference in applying these techniques would be some of the performance costs. For example, a hosted mobile graphics virtualization solution based on KVM/ARM [13, 14] would require a world-switch that could cost thousands of cycles, in contrast to using diplomats to switch between thread personas from iOS to Android at a lower cost of a couple of system calls.

## 11 CONCLUSIONS

We have presented a graphics-focused study of Cycada, and extended its binary compatible graphics support for running iOS apps on Android through three new OS compatibility mechanisms: **(1)** extended diplomat construction and new diplomat usage patterns, direct, indirect, data-dependent, and multi, **(2)** thread impersonation which allows one thread to temporarily assume the persona of another thread, and **(3)** dynamic library replication which allows the linker to create separate loaded instances of a dynamic library. These mechanisms proved essential to support our prototype in running widely used iOS apps, many of which utilize WebKit and other frameworks that rely heavily on the GPU. We discussed our experiences with these mechanisms in the context of GPU device support, and demonstrated their feasibility by using iOS's Safari on Cycada to visit popular websites and run browser benchmarks.

## 12 ACKNOWLEDGMENTS

This work was supported in part by NSF grants CNS-1717801, CNS-1563555, CNS-1422909, and CCF-1162021.

## REFERENCES


[1] Alexa Internet, Inc. Alexa - Top Sites in United States. http://www.alexa.com/topsites/countries/US, Apr. 2014.

[2] Amstadt, B., and Johnson, M. K. Wine. *Linux Journal* (Aug. 1994).

[3] Andrus, J. *Multi-Persona Mobile Computing.* PhD thesis, Columbia University, Feb. 2015.

[4] Andrus, J., Van't Hof, A., AlDuaij, N., Dall, C., Viennot, N., and Nieh, J. Cider: Native Execution of iOS Apps on Android. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2014), ASPLOS 2014, pp. 367–382.

[5] Apple, Inc. SunSpider 1.0.2 JavaScript Benchmark. https://www.webkit.org/perf/sunspider/sunspider.html, 2013.

[6] Apple, Inc. iOS Device Compatibility Reference: OpenGL ES Graphics. https://developer.apple.com/library/ios/documentation/DeviceInformation/Reference/iOSDeviceCompatibility/OpenGLESPlatforms/OpenGLESPlatforms.html#//apple_ref/doc/uid/TP40013599-CH106-SW1, Feb. 2014.

[7] Apple, Inc. The WebKit Open Source Project. http://www.webkit.org/, Apr. 2014.

[8] Apple Kernel Engineer. Personal Communication, Mar. 2014.

[9] Baumann, A., Lee, D., Fonesca, P., Glendenning, L., Lorch, J. R., Bond, B., Olinsky, R., and Hunt, G. C. Composing OS Extensions Safely and Efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Apr. 2013), EuroSys 2013, pp. 239–252.

[10] Bellard, F. QEMU, A Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference* (Apr. 2005), USENIX ATC 2005, pp. 41–46.

[11] Black Duck Software, Inc. WebKit Open Source Project on Ohloh. http://www.ohloh.net/p/WebKit, Apr. 2014.

[12] Chernoff, A., Herdeg, M., Hookway, R., Reeve, C., Rubin, N., Tye, T., Yadavalli, S. B., and Yates, J. FX!32: A Profile-Directed Binary Translator. *IEEE Micro 18*, 2 (Mar. 1998), 56–64.

[13] Dall, C., Li, S.-W., Lim, J. T., Nieh, J., and Koloventzos, G. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture* (June 2016), ISCA 2016, pp. 304–316.

[14] Dall, C., and Nieh, J. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2014), ASPLOS 2014, pp. 333–348.

[15] Dolezel, L. The Darling Project. http://darling.dolezel.info/en/Darling, Aug. 2012.

[16] Dowty, M., and Sugerman, J. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review 43* (July 2009), 73–82.

[17] Dreyfus, E. Linux Compatibility on BSD for the PPC Platform. http://onlamp.com/lpt/a/833, May 2001.

[18] Dreyfus, E. IRIX Binary Compatibility, Parts 1–6. http://onlamp.com/lpt/a/2623, Aug. 2002.

[19] Dreyfus, E. Mac OS X Binary Compatibility on NetBSD: Challenges and Implementation. In *Proceedings of the 2004 EuroBSDCon* (Oct. 2004).

[20] Filip Pizlo. Surfin' Safari - Blog Archive - Introducing the WebKit FTL JIT. https://www.webkit.org/blog/3362/introducing-the-webkit-ftl-jit/, May 2014.

[21] FreeBSD Documentation Project. Linux Binary Compatibility. In *FreeBSD Handbook*, B. N. Handy, R. Murphey, and J. Mock, Eds. 2000, ch. 11.

[22] Gamme, E., Johnson, R., Helm, R., and John, V. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, Oct. 1994.

[23] Geoff Stahl. GL_APPLE_fence. https://www.opengl.org/registry/specs/APPLE/fence.txt, Aug. 2002.

[24] Hohensee, P., Myszewski, M., and Reese, D. Wabi CPU Emulation. In *Proceedings of the 8th Symposium on High Performance Chips* (Aug. 1996), Hot Chips 1996, pp. 47–65.

[25] Hunt, G. C., and Brubacher, D. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium* (July 1999), WINSYM 1999.

[26] John Rosasco and Andrew Barnes. GL_APPLE_row_bytes. http://www.opengl.org/registry/specs/APPLE/row_bytes.txt, Oct. 2006.

[27] John Spitzer and Mark Kilgard and Acorn Pooley. GL_NV_fence. https://www.khronos.org/registry/gles/extensions/NV/fence.txt, Dec. 2008.

[28] Khronos Group. OpenGL Extensions – OpenGL.org. http://www.opengl.org/wiki/OpenGL_Extensions.

[29] Khronos Group. OpenGL ES Common Profile Specification Version 2.0.25 (Full Specification). http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf, Nov. 2010.

[30] Khronos Group. Khronos Native Platform Graphics Interface (EGL Version 1.4). http://www.khronos.org/registry/egl/specs/eglspec.1.4.20130211.pdf, Feb. 2013.

[31] Khronos Group. OpenGL ES – The Standard for Embedded Accelerated 3D Graphics. http://www.khronos.org/opengles/, Jan. 2013.

[32] Laadan, O., and Nieh, J. Operating System Virtualization: Practice and Experience. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference* (May 2010), SYSTOR 2010, pp. 17:1–17:12.

[33] Lagar-Cavilla, H. A., Tolia, N., Satyanarayanan, M., and de Lara, E. VMM-independent Graphics Acceleration. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (June 2007), VEE 2007, pp. 33–43.

[34] Linux Containers. Linux Containers - LXC - Introduction, Sept. 2017. https://linuxcontainers.org/lxc/introduction/.

[35] Nieuwejaar, N., Schrock, E., Kucharski, W., Blaine, R., Pilatowicz, E., and Leventhal, A. Method for Defining Non-Native Operating Environments. US 7689566, Filed Dec. 12, 2006, Issued Mar. 30, 2010. http://www.patentlens.net/patentlens/patent/US_7689566/.

[36] NVIDIA Corporation. High Performance Computing (HPC) and Supercomputing | NVIDIA Tesla | NVIDIA. http://www.nvidia.com/object/tesla-supercomputing-solutions.html, Apr. 2014.

[37] NVIDIA Corporation. Shared Virtual GPU (vGPU) Technology | NVIDIA. http://www.nvidia.com/object/virtual-gpus.html, Apr. 2014.

[38] OpenVZ. http://openvz.org/Main_Page.

[39] Oracle Corporation. Consolidating Applications with Oracle Solaris Containers. http://www.oracle.com/technetwork/server-storage/solaris/documentation/consolidating-apps-163572.pdf, July 2011.

[40] Osman, S., Subhraveti, D., Su, G., and Nieh, J. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Dec. 2002), OSDI 2002, pp. 361–376.

[41] PassMark Software, Inc. PerformanceTest Mobile on the App Store. https://itunes.apple.com/us/app/performancetest-mobile/id494438360?ls=1&mt=8, June 2012.

[42] PassMark Software, Inc. PassMark PerformanceTest – Android Apps on Google Play. https://play.google.com/store/apps/details?id=com.passmark.pt_mobile, Jan. 2013.

[43] Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R., and Hunt, G. C. Rethinking the Library OS from the Top Down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2011), ASPLOS 2011, pp. 291–304.

[44] Potter, S., and Nieh, J. Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference* (June 2010), USENIX ATC 2010, pp. 103–116.

[45] Stegmaier, S., Magallón, M., and Ertl, T. A Generic Solution for Hardware-Accelerated Remote Visualization. In *Proceedings of the Symposium on Data Visualisation* (2002), VISSYM 2002, pp. 87–94.

[46] Sugerman, J., Venkitachalam, G., and Lim, B.-H. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference* (June 2001), USENIX ATC 2001, pp. 1–14.

[47] The Android OpenSource Project. Graphics | Android Developers. https://source.android.com/devices/graphics.html, Jan. 2013.

[48] The Android OpenSource Project. Dashboards | Android Developers. http://developer.android.com/about/dashboards/index.html, Apr. 2014.

[49] The VirtualGL Project. VirtualGL | Main / The VirtualGL Project. http://www.virtualgl.org/, May 2014.

[50] Tian, K. Graphics Virtualization (XenGT) | 01.org. https://01.org/xen/blogs/srclarkx/2013/graphics-virtualization-xengt, Apr. 2014.

[51] Web Standards Project. The Acid3 Test. http://www.acidtests.org/, Mar. 2008.

[52] WebKit Community. Bug 24986 – [multi-patch] ARM JIT port. https://bugs.webkit.org/show_bug.cgi?id=24986, June 2009.

[53] X.Org Foundation. GLX. http://dri.freedesktop.org/wiki/GLX/, Apr. 2013.