POSIX Has Become Outdated

VAGGELIS ATLIDAKIS, JEREMY ANDRUS, ROXANA GEAMBASU, DIMITRIS MITROPOULOS, AND JASON NIEH

Vaggelis Atlidakis is a PhD student in the Department of Computer Science at Columbia University. His research interests revolve around broad areas of computer systems, including operating systems, security and privacy, and applications of machine learning to systems. vatlidak@cs.columbia.edu

Jeremy Andrus holds a PhD in operating systems and mobile computing from Columbia University and currently works as a Kernel Engineering Manager at Apple. jeremy_andrus@apple.com

Roxana Geambasu is an Assistant Professor of Computer Science at Columbia University. Her research interests are wide-ranging and include distributed systems, security and privacy, operating systems, databases, and the application of cryptography and machine learning to systems. roxana@cs.columbia.edu

The POSIX standard for APIs was developed over 25 years ago. We explored how applications in Android, OS X, and Ubuntu Linux use these interfaces today and found that weaknesses or deficiencies in POSIX have led to divergence in how modern applications use the POSIX APIs. In this article, we present our analysis of over a million applications and show how developers have created workarounds to shortcut POSIX and implement functionality missing from POSIX.

The Portable Operating System Interface (POSIX) is the IEEE standard operating system (OS) service interface for UNIX-based systems. It describes a set of fundamental abstractions needed for efficient construction of applications. Since its creation over 25 years ago, POSIX has evolved to some extent (e.g., the most recent update was published in 2013 [9]), but the changes have been small overall. Meanwhile, applications and the computing platforms they run on have changed dramatically: modern applications for today's smartphones, desktop PCs, and tablets interact with multiple layers of software frameworks and libraries implemented atop the OS.

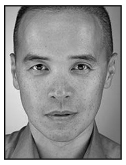
Although POSIX continues to serve as the single standardized interface between these software frameworks and the OS, little has been done to measure whether POSIX abstractions are effective in supporting modern application workloads, or whether new, non-standard abstractions are taking form, dethroning traditional ones.

We present the first study of POSIX usage in modern OSes, focusing on three of today's most widely used mobile and desktop OSes—Android, OS X, and Ubuntu—and popular consumer applications characteristic to these OSes. We built a utility called *libtrack*, which supports both dynamic and static analyses of POSIX use from applications. We used libtrack to shed light on a number of important questions regarding the use of POSIX abstractions in modern OSes, including which abstractions work well, which appear to be used in ways for which they were never intended, which are being replaced by new and non-standard abstractions, and whether the standard is missing any fundamental abstractions needed by modern workloads. Our findings can be summarized as follows:

First, *usage is driven by high-level frameworks, which impacts POSIX's portability goals.* The original goal of the POSIX standard was application source code portability. However, modern applications are no longer being written to standardized POSIX interfaces. Instead, they rely on platform-specific frameworks and libraries that leverage high-level abstractions for inter-process communication (IPC), thread pool management, relational databases, and graphics support.

Modern OSes gravitate towards a more layered programming model with "taller" interfaces: applications directly link to high-level frameworks, which invoke other frameworks and libraries that may eventually utilize POSIX. This new, layered programming model imposes challenges to application portability and has given rise to many different cross-platform SDKs that attempt to fill the gap left by a standard that has not evolved with the rest of the ecosystem.

Dimitris Mitropoulos is a Postdoctoral Researcher in the Computer Science Department at Columbia University in the City of New York. He holds a PhD ('14) in Cybersecurity with distinction from the Athens University of Economics and Business. His research interests include application security, systems security, and software engineering. dimitris.i.mitropoulos@gmail.com


Jason Nieh is a Professor of Computer Science and Co-Director of the Software Systems Laboratory at Columbia University. Professor Nieh has made research contributions in software systems across a broad range of areas, including operating systems, virtualization, thin-client computing, cloud computing, mobile computing, multimedia, Web technologies, and performance evaluation. nieh@cs.columbia.edu

Second, *extension APIs, namely* `ioctl`, *dominate modern POSIX usage patterns as OS developers increasingly use them to build support for abstractions missing from the POSIX standard.* Extension APIs have become the standard way for developers to circumvent POSIX limitations and facilitate hardware-supported functionality for graphics, sound, and IPC.

Third, *new abstractions are arising, driven by the same POSIX limitations across the three OSes, but the new abstractions are not converging.* To deal with abstractions missing from the aging POSIX standard, modern OSes are implementing new abstractions to support higher-level application functionality. Although these interfaces and abstractions are driven by similar POSIX limitations and are conceptually similar across OSes, they are not converging on any new standard, increasing the fragmentation of POSIX across UNIX-based OSes.

We believe our findings have broad implications related to the future of POSIX-compliant OS portability, which the systems research community and standards bodies will likely need to address in the future. To support further studies across a richer set of UNIX-based OSes and workloads, we make *libtrack*'s source code, along with the application workloads and traces, publicly available at: https://columbia.github.io/libtrack/. The full version of our work was published in EuroSys 2016 [3].

Methodology

Our study involves two types of experiments with real, client-side applications on the three OSes: *dynamic experiments* and *static analysis*. In support of our study, we developed *libtrack*, a tool that traces the use of a given native C library from modern applications. *libtrack* implements two modules: a dynamic module and a static module.

libtrack

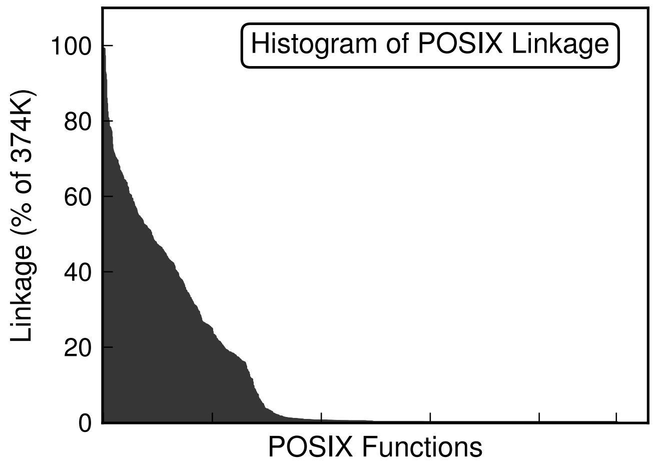
Dynamic Module: *libtrack*'s dynamic module traces all invocations of native POSIX functions for every thread in the system. For each POSIX function implemented in the C standard library of the OS, *libtrack* interposes a special "wrapper" function with the same name, and once a native POSIX function is called, *libtrack* logs the time of the invocation and a backtrace identifying the path by which the application invoked the POSIX function. It also measures the time spent executing the POSIX function. *libtrack* then analyzes these traces to construct a call graph and derive statistics and measurements of POSIX API usage.

Static Module: *libtrack* also contains a static module, which is a simple utility to help identify application linkage to POSIX functions of C standard libraries. Given a repository of Android APKs or of Ubuntu packages, *libtrack*'s static module first searches each APK or package for native libraries. Then it decompiles native libraries and scans the dynamic symbol tables for relocations to POSIX symbols. Dynamic links to POSIX APIs are indexed per application (or per package) and are finally merged to produce aggregate statistics of POSIX linkage.

Workloads

Using *libtrack*, we performed both dynamic and static experiments. We used different workloads for each experiment type centered around consumer-oriented applications; these do not reflect POSIX's standing in other types of workloads, such as server-side or high-performance computing workloads.

Dynamic Experiments: We performed dynamic experiments by interacting with popular Android, OS X, and Ubuntu applications (apps). We selected these apps from the official marketplaces for each OS: Google Play (45 apps), Apple AppStore (10 apps), and Ubuntu Software Center (45 apps). We chose popular apps based on the number of installs, selecting apps across nine categories, and interacted manually with these applications by performing typical operations, such as refreshing an inbox or sending an email with an email application, on commodity devices (laptops and tablets).



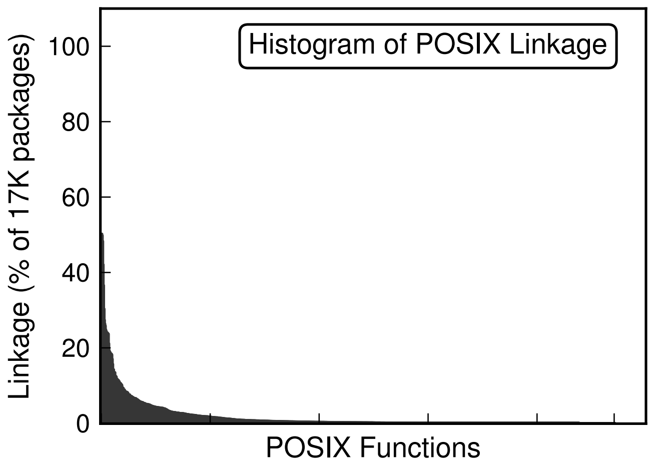


Figure 1: POSIX abstractions linkage

Static Experiments: We performed static experiments of POSIX usage at large scale by downloading over a million consumer applications and checking these applications, and associated libraries, for linkage to POSIX functions of C standard libraries. For Android, we downloaded 1.1 million free Android apps from a Dec. 4, 2014 snapshot of Google Play [10]. For Ubuntu, we downloaded 71,199 packages available for Ubuntu 12.04 on Dec. 4, 2014, using the Aptitude package manager.

Results

We organized the results from our study in a sequence of questions regarding the use of POSIX in modern OS. We began by asking which POSIX functions and abstraction families were being used and which were not being used by modern workloads.

Which Abstractions Are Used and Which Are Not Used by Modern Workloads?

To answer this question, we first used our static analysis on Android and Ubuntu in order to examine which of the implemented abstractions were actually linked by applications or their libraries, giving us a more accurate view into what abstractions are not used by modern workloads. Afterwards, we used results from our dynamic experiments to examine what abstractions are effectively invoked by modern workloads, telling us what the popular POSIX abstractions are.

Linked Abstractions: Figure 1 shows the number of Android apps and Ubuntu packages that dynamically link to POSIX functions of the respective C standard libraries. The results come from our large-scale static analyses of 1.1 million Android apps and 71,989 Ubuntu packages.

Overall, in Android, of the 821 POSIX functions implemented, 114 of them are never dynamically linked from any native library, and approximately half (364 functions) are dynamically linked from 1% or fewer of the apps. Furthermore, our static analysis of Ubuntu packages shows that desktop Linux has a similar, albeit less definitive, trend with Android: phasing out traditional IPC and FS POSIX abstractions.

Dynamically Invoked Abstractions: Although linkage is a definite way to identify unused functions, it is only a speculative way to infer usage. Therefore, we next examine the actual usage of POSIX functions during our dynamic experiments with 45 Android apps. POSIX functions are categorized in abstraction subsystems; the most popular in term of invocations, as well as the most expensive in terms of CPU time, are shown in Figure 2.

We observe that memory is the most heavily invoked subsystem. Typical representatives of this subsystem include user-space utilities for memory handling and system call-backed functions for (de)allocation of memory regions and protection properties setting. As shown in Figure 2, the memory subsystem is also the most expensive subsystem in terms of CPU consumption. The popularity and the cost of memory calls are driven by the proliferation of high-level frameworks that are heavily dependent on memory-related OS functionality. The second most heavily invoked POSIX subsystem is threading. Its popularity is mainly due to Thread Local Storage (TLS) operations, which are very usual in Android to help map between high-level framework threads and low-level native `pthreads`. These operations are relatively lightweight and therefore, contrary to memory, threading accounts for relatively little CPU time. Most of the remaining subsystems include popular or expensive functions, but their CPU time cost is usually proportional to the volume of invocations.

Crucially, there is one surprising exception: `ioctl`. This function alone accounts for more than 16% of the total CPU time, despite its relatively low volume of invocations (0.6%). The striking popularity of `ioctl`—an extension API that lets applications bypass well-defined POSIX abstractions and directly interact with the

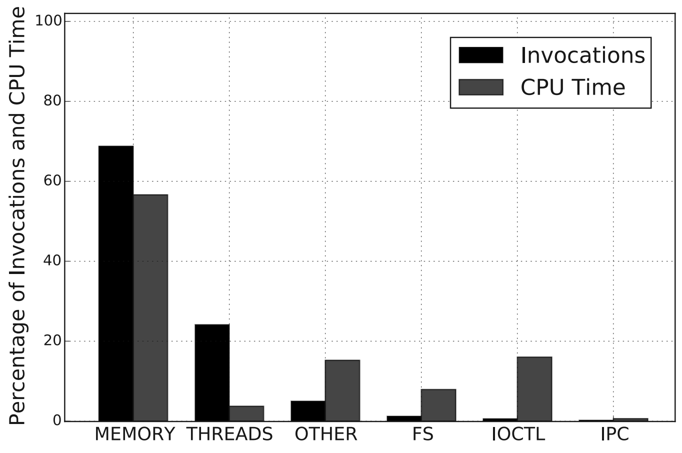


Figure 2: POSIX invocations and CPU consumption for 45 Android apps

kernel—proclaims that POSIX abstractions experience significant limitations in supporting modern application workloads. Therefore, developers increasingly resort to `ioctl` and implement custom support missing from POSIX.

To gain a view on the type of functionality implemented using `ioctl` across the three OSes, we inspect stack traces leading to `ioctl` invocations and identify which libraries triggered them. Table 1 shows the top libraries across the three OSes. In Android, graphics libraries lead to the lion's share of `ioctl` invocations, followed by *Binder*. In OS X, the majority of `ioctl` invocations come from network libraries. In Ubuntu, graphics libraries trigger approximately half of `ioctl` invocations, and the remaining part is mainly due to libraries providing network functionality. We next ask *what* are the abstractions missing from POSIX, if any, and *why* do framework libraries resort to unstructured, extension APIs to implement their functionality?

Does POSIX Lack Abstractions Needed by Modern Workloads?

Taking hints from Table 1, we investigate graphics functionality in modern OSes, which significantly rely on `ioctl` signaling that such abstractions are missing in POSIX. In the following section, we additionally discuss new IPC abstractions, which are replacing older abstractions and are implemented atop `ioctl`.

Graphics: POSIX explicitly omits graphics as a point of standardization. As a result, there is no standard OS interface to graphics cards, and different platforms have chosen different ways to optimize this critical resource. The explicit omission of a graphics standard could be due to the emergence of OpenGL, a popular cross-platform graphics API used in drawing and rendering. While OpenGL works well for applications, OS and graphics library designers have no well-defined interfaces to accomplish increasingly complicated operations.

The lack of a standard kernel interface to GPUs has led to limited extensibility. To alleviate this fragmentation, modern GPU drivers and libraries are trending towards a general purpose computational resource, and POSIX is not in a position to standardize the use of this new, powerful paradigm. These problems have been studied in detail the past four years. For example, the PTask API [8] defines a dataflow-based programming model that allows the application programmer to utilize GPU resources in a more intuitive way, matching API semantics with OS capabilities. This new OS abstraction results in massive performance gains.

The lack of a standard graphics OS abstraction also causes development and runtime problems. With no alternative, driver developers are forced to build their own structure around the only available system call: `ioctl`. Using opaque input and output tokens, the `ioctl` call can be general purpose, but it was never intended for the complicated, high-bandwidth interface GPUs require. Graphics library writers must either use `ioctl` as a funnel into which all GPU command and control is sent via opaque data blobs, or they must design a vendor-specific demux interface akin to the `syscall` system call.

Interestingly, in OS X, graphics functionality does not account for any significant portion of `ioctl` invocations. The reason is that the driver model in OS X, `IOKit`, is more structured than in Android or Ubuntu, and it uses a well-defined Mach IPC interface that can effectively marshal parameters and graphics data across the user-kernel boundary. This creates a reusable, vendor-agnostic API. However, the current interface is designed around the same black-box hardware that runs on other platforms. Therefore, a standardized OS interface to graphics processors would likely have the same benefits on OS X as it would on Ubuntu or Android.

OS	First library	Second library	Third library	Invocations
|Android	Graphics (74%) (e.g., libnvrm)	Binder (24%) (e.g., libbinder)	Other (1%)	1.3M
|OS X	Network (99%) (e.g., net.dylib)	Loader (0.6%) (e.g., dyld)	—	682
|Ubuntu	Graphics (52%) (e.g., libgtk)	Network (47%) (e.g., libQtNet)	Other (1%)	0.4M

Table 1: Top libraries that invoke ioctl

Tx/Rx—Android	Pipes avg (μs)	UNIX avg (μs)	Binder avg (μs) |
32 bytes	40	54	115
128 bytes	44	56	114
1 page	62	73	93
10 pages	291	276	93
100 pages	2402	1898	94

Tx/Rx—OS X	Pipes avg (μs)	UNIX avg (μs)	Mach | avg (μs)
32 bytes	6	11	19
128 bytes	7	51	19
1 page	8	54	12
10 pages	18	175	15
100 pages	378	1461	12

Tx/Rx—Ubuntu	Pipes avg (μs)	UNIX | avg (μs)
32 bytes	18	18
128 bytes	20	10
1 page	21	20
10 pages	58	27
100 pages	923	186

Table 2: Latency comparison of different IPC mechanisms

What POSIX Abstractions Are Being Replaced?

Continuing with hints from Table 1, we discuss abstractions that exist in POSIX but appear to be replaced by new abstractions.

Inter-Process Communication: A central IPC abstraction in POSIX is the message queue API (`mq_*`). On all platforms we studied, applications used some alternate form of IPC. In fact, Android is missing the `mq_*` APIs altogether. IPC on all of these platforms has divergently evolved beyond POSIX.

- **IPC in Android:** Binder is the standard method of IPC in Android. It overcomes POSIX IPC limitations and serves as the backbone of Android inter- and intra-process communication. Using a custom kernel module, Binder IPC supports file descriptors passing between processes, implements object reference counting, and uses a scalable multithreaded model that allows a process to consume many simultaneous requests. In addition, Binder leverages its access to processes' address space and provides fast, single-copy transactions. Binder exposes IPC abstractions to higher layers of software, and Android apps can focus on logical program flow and interact with Binder through standard Java objects and methods, without the need to manage low-level IPC details. Because no existing API supported all the necessary functionality, Binder was implemented using `ioctl` as the singular kernel interface. Binder IPC is used in every Android application, and accounts for nearly 25% of the total measured POSIX CPU time that funnels through the `ioctl` call.
- **IPC in Linux:** The D-Bus protocol [5] provides apps with high-level IPC abstractions in GNU/Linux. It describes an IPC messaging bus system that implements (1) a system daemon monitoring system-wide events and (2) a per-user login session daemon for communication between applications within the same session. The applications we inspect use mostly the *libdbus* implementation of D-Bus (38 out of 45 apps). An evolution of D-Bus, called the Linux Kernel D-Bus, or *kdbus*, is also gaining increasing popularity in GNU/Linux OSes. It is an in-kernel implementation of D-Bus that uses Linux kernel features to overcome inherent limitations of user-space D-Bus implementations. Specifically, it supports zero-copy message passing between processes, and it is available at boot allowing Linux security to directly leverage it.
- **IPC in OS X:** IPC in OS X diverged from POSIX since its inception. Apple's XNU kernel uses an optimized descendant of CMU's Mach IPC [2, 7] as the backbone for inter-process communication. Mach comprises a flexible API that supports high-level concepts such as: object-based APIs abstracting communication channels as *ports*, real-time communication, shared memory regions, RPC, synchronization, and secure resource management. Although flexible and extensible, the complexity of Mach has led Apple to develop a simpler higher-level API called *XPC*. Most apps use XPC APIs that integrate with other high-level APIs, such as Grand Central Dispatch.

To highlight key differences in POSIX-style IPC and newer IPC mechanisms, we adapt a simple Android Binder benchmark from the Android code base to measure both pipes and UNIX domain sockets as well as Binder transactions. We also use the MPMMTest application from Apple's open source XNU [1]. We measure the latency of a round-trip message using several different message sizes, ranging from 32 bytes to 100 (4096 byte) pages. We run our benchmarks using an ASUS Nexus-7 tablet with Android 4.3 Jelly Bean, a MacBook Air laptop (4-core Intel CPU @2.0 GHz, 4 GB RAM) with OS X Yosemite, and a Dell XPS laptop (4-core Intel CPU @1.7 GHz, 8 GB RAM) with Ubuntu 12.04 Precise Pangolin.

The results, averaged over 1000 iterations, are summarized in Table 2. Both Binder and Mach IPC leverage fast, single-copy and zero-copy mechanisms, respectively. Large messages in both Binder and Mach IPC are sent in near-constant time. In contrast, traditional POSIX mechanisms on all platforms suffer from large variation and scale linearly with message size.

Asynchronous I/O: Our experiments with Android, OS X, and Ubuntu apps reveal another evolutionary trend: the replacement of POSIX APIs for asynchronous I/O with new abstractions built atop POSIX multithreading abstractions. The nature and purpose of threads has been a debate in OS research for a long time [4, 6], and POSIX makes no attempt to prioritize a threading model over an event-based model.

Our study reveals that while POSIX locking primitives are still extremely popular, new trends in application abstractions are blurring the line between event and thread by combining high-level language semantics with pools of threads fed by event-based loops. This new programming paradigm is enforced by the nature of GUI applications that require low-latency. While an event-based model may seem the obvious solution, event processing in the input or GUI context leads to suboptimal user experience. Therefore, dispatching event-driven work to a queue backed by a pool of pre-initialized threads has become a de facto programming model in Android, OS X, and Ubuntu. Although this paradigm appears in all the OSes we studied, the implementations are extremely divergent.

Android defines several Java classes that abstract the concepts of creating, destroying, and managing threads (`ThreadPool`), looping on events (`Looper`), and asynchronous work dispatching (`ThreadPoolExecutor`). Ubuntu applications can choose from a variety of libraries providing similar functionality, but through vastly different interfaces. For example, the `libglib`, `libQtCore`, and `libnspr` all provide high-level thread abstractions based on the GNOME desktop environment. In OS X the APIs are, yet again, different. The vast majority of event-driven programming in OS X is done through Mach IPC. Apple has written high-level APIs around event handling, thread pool management, and asynchronous task dispatch. Most of these APIs are enabled through Grand Central Dispatch (GCD). GCD manages a pool of threads and even defines POSIX alternatives to semaphores, memory barriers, and asynchronous I/O. The GCD functionality is exported to applications from classes such as `NSOperation`.

In summary, driven by the strong need for asynchrony and the event-based nature of modern GUI applications, different OSes have created similar but not converging and non-standard adherent-threading support mechanisms.

Conclusion

Perfect application portability across UNIX-based OSes is clearly beyond the realm of possibility. However, maintaining a subset of harmonized abstractions is still a viable alternative for preserving some uniformity within the UNIX-based OSes. Our study shows that new abstractions, beyond POSIX, are taking form in three modern UNIX-based OSes (Android, OS X, and Ubuntu), and that changes are not converging to any new unified set of abstractions. We believe that a new revision of the POSIX standard is due, and we urge the research community to investigate what that standard should be. Our study provides a few examples of abstractions—such as graphics, IPC, and threading—as starting points for re-standardization, and we recommend that changes should be informed by popular frameworks that have a principal role in modern OSes.

Availability

The extended version of our work was published in EuroSys 2016 [3]. To support further studies across a richer set of UNIX-based OSes and workloads, we open source our code along with the application workloads and traces available: https://columbia.github.io/libtrack/.

Acknowledgments

This work was supported in part by DARPA Contract FA8650-11-C-7190, NSF grants CNS-1351089, CNS-1162447, CNS-1422909, and CCF-1162021, a Google Research Award, and a Microsoft Research Faculty Fellowship.

References

[1] Apple, Inc., OS X 10.11.2, Source: http://opensource.apple.com/tarballs/xnu/xnu-3248.20.55.tar.gz, accessed 03/20/2016.

[2] Apple, Inc., Mach Overview: https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/KernelProgramming/Mach/Mach.html, Aug. 2013, accessed 03/22/2015.

[3] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh, "A Measurement Study of POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing," in *Proceedings of the 11th European Conference on Computer Systems (EuroSys 2016),* April 2016.

[4] R. V. Behren, J. Condit, and E. Brewer, "Why Events Are a Bad Idea (for High-Concurrency Servers)," in *Proceedings of the 9th Conference on Hot Topics in Operating Systems,* vol. 9, May 2003, pp. 19–24: https://www.usenix.org/legacy/events/hotos03/tech/full_papers/vonbehren/vonbehren.pdf.

[5] freedesktop.org., D-Bus: http://www.freedesktop.org/wiki/Software/dbus/, accessed 10/18/2015.

[6] J. K. Ousterhout, "Why Threads Are a Bad Idea (for Most Purposes)," presentation given at the 1996 USENIX Annual Technical Conference, Jan. 1996.

[7] R. Rashid, R. Baron, A. Forin, R. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi, "Mach: A Foundation for Open Systems," in *Proceedings of the 2nd Workshop on Workstation Operating Systems*, IEEE, Sept. 1989, pp. 109–112.

[8] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating System Abstractions to Manage GPUs as Compute Devices," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, Oct. 2011, pp. 233–248.

[9] The Open Group and IEEE, last POSIX revision: http://pubs.opengroup.org/onlinepubs/9699919799, accessed 03/19/2015.

[10] N. Viennot, E. Garcia, and J. Nieh, "A Measurement Study of Google Play," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2014),* June 2014, pp. 221–233.