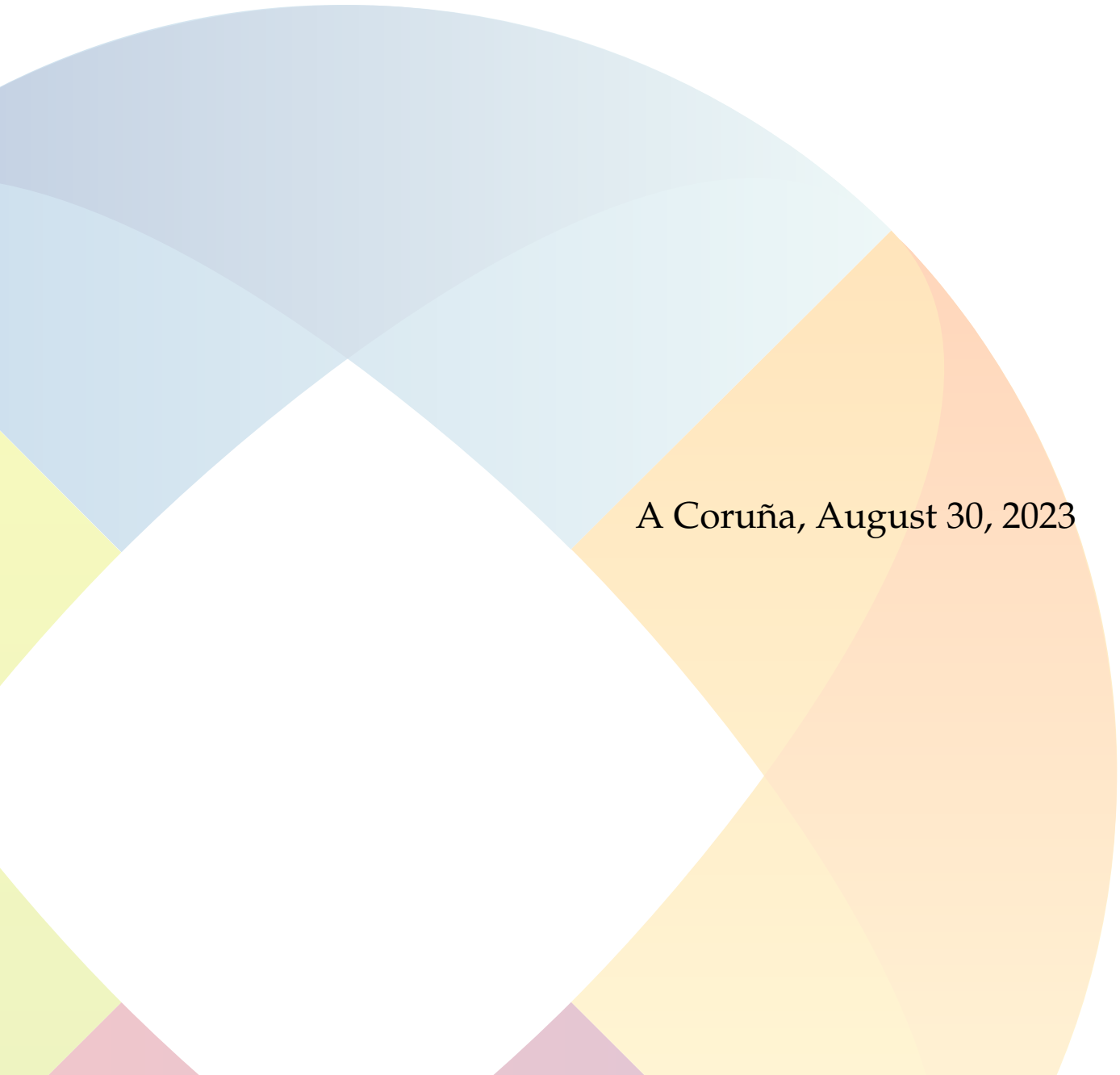

Field Survey: Cross-Platform Mobile App Frameworks in 2023

A Coruña, August 30, 2023

1 Executive summary

This document summarizes the findings of a field survey of mobile application development frameworks, performed by Igalia engineers in the first half of 2023.

Our primary goal is to enable cross-platform mobile app framework creators to make state-of-the-art design decisions, allowing apps built in their frameworks to compete with native apps as well as with other cross-platform apps. We hope however that the study can also provide insight to developers of native platform toolkits as well as to app developers.

After studying Capacitor, React Native, NativeScript, Flutter, and Ark, we identify the following patterns.

1. *The so-called "declarative" style of UI development has won, universally.* The last 5 years have seen a progressive switch away from the older "imperative" style, on Android and iOS native platforms as well as in cross-platform frameworks.
2. *Even leading frameworks are immature.* The field is not in an equilibrium state. As a corrolary, now is as good a time as any for a competing solution to enter the market.
3. *JavaScript on mobile is not like JavaScript on the web.* Cross-platform app development frameworks have to choose a language; most choose JavaScript. This presents opportunities for technology re-use from browsers and an established development community, but also challenges related to the language itself and the practices of that community. Achieving fast startup and smooth, jank-free interaction and animation pushes some frameworks to make different tradeoffs than those made on the web.
4. *Performance is a framework concern.* Frameworks create a division of labor between "what" and "how": app developers describe what should happen, and turning that into good user experiences is a framework responsibility. The general technique that frameworks use to ensure good performance is compilation. Some examples include: ahead-of-time compilation of JavaScript in React Native, or Dart in Flutter; ahead-of-time compilation of Angular interfaces in Ionic/Capacitor; ahead-of-time compilation of shaders in Flutter's new Impeller renderer.
5. *Flutter represents the state of the art.* By this we mean that the structure of Flutter appears to be approaching a local maximum in the design space, and that other frameworks are moving in its direction. Flutter renders directly to

the GPU, instead of using native or Web widgets; we expect that React Native will eventually do the same. Flutter also has a highly-tuned ahead-of-time compiler, albeit based on its own Dart language; we expect that JavaScript-based frameworks will evolve TypeScript and their compiler toolchains in this direction as well, over a longer time-frame.

The rest of this report goes deeper into all of these points by making a detailed study of each framework. We also identify two next steps for cross-platform framework authors:

1. Flutter achieves good ahead-of-time performance of its apps because it has been able to co-design the Dart language for its use cases. To some extent, JavaScript should be able to benefit from the same design evolution. It may be expected that in the mid-term, that TypeScript gains a "sound" typing system, allowing ahead-of-time compilers to produce more efficient code. This will take some time and effort though, as JavaScript is a large community with many stakeholders.
2. Another approach to solving some of the challenges posed by JavaScript is to switch languages entirely. In that regard, implementing Flutter on top of Rust is a very attractive proposition; simply cloning the design decisions of Flutter is likely to yield a quite competitive system.

Contents

1 Executive summary **2**

2 Case studies **6**
2.1 Capacitor . . . 7
2.1.1 Overview . . . 7
2.1.2 Evaluation . . . 8
2.1.3 Aside: Can WebView act like native widgets? . . . 11
2.1.4 Aside: Is a WebView as good as native widgets? . . . 11
2.1.5 Summary . . . 11
2.2 React Native . . . 12
2.2.1 Overview . . . 12
2.2.2 Evaluation . . . 15
2.2.3 Aside: As good as native? . . . 17
2.2.4 Aside: Parallels with the browser . . . 17
2.2.5 Summary . . . 17
2.3 NativeScript . . . 19
2.3.1 Overview . . . 19
2.3.2 Evaluation . . . 20
2.3.3 Aside: Are markets wise? . . . 23
2.3.4 Aside: On the expressive power of application framworks . . . 23
2.3.5 Summary . . . 24
2.4 Flutter . . . 25
2.4.1 Overview . . . 25
2.4.2 Evaluation . . . 28
2.4.3 Aside: An escape hatch to the platform . . . 30
2.4.4 Aside: For want of a canvas . . . 31
2.4.5 Summary . . . 31
2.5 Ark . . . 32
2.5.1 Overview . . . 32
2.5.2 Evaluation . . . 36
2.5.3 Aside: On the importance of storytelling . . . 38
2.5.4 Aside: O platform, my platform . . . 38
2.5.5 Summary . . . 39

3 Future developments **41**
3.1 Possible futures . . . 41
3.1.1 Rust . . . 42
3.1.2 The web of pixels . . . 44
3.1.3 WebAssembly . . . 46

4 Conclusion **49**
4.1 Observations . . . 49
4.1.1 The age of declarative UI . . . 49
4.1.2 Unstable equilibrium . . . 49
4.1.3 Language . . . 50
4.1.4 Predictable over peak . . . 51
4.1.5 Performance is a framework concern . . . 51
4.2 Predictions . . . 52
4.2.1 Flutter on JS . . . 52
4.2.2 Flutter on Rust . . . 53
4.2.3 Flutter on Wasm . . . 53

5 Further resources **54**
5.1 EOSS 2023: Cross-Platform Mobile UI . . . 54
5.2 Publishing history . . . 54

2 Case studies

If you were designing a new mobile operating system, what kind of app development platform would you provide? This study aims to provide answers by observing the state of the field in an attempt to determine competitors' position and velocity in the design space.

We hope to identify actionable insights by specifically focussing our attention on cross-platform app development frameworks, where developers have purposefully decided to depart from the facilities provided by the native platforms. The majority of these cross-platform frameworks use JavaScript as the application development language, so we spend most of our time there: Capacitor, React Native, and NativeScript.

We also take a look at extensions to JavaScript by the Ark framework used in OpenHarmony, as well as the use of the Dart language by the Flutter framework.

The following five sections take a deep dive into the respective cross-platform frameworks. We follow up these case studies with a discussion of near-term and mid-term future developments, and conclude with expanded insights and recommendations.

2.1 Capacitor

Our first case study looks at the Ionic Framework[1] UI toolkit, based on the Capacitor[2] web run-time.

2.1.1 Overview

Capacitor is like an alternate web browser development kit for phones. It uses the platform's native WebView: WKWebView[3] on iOS or WebView[4] on Android. The JavaScript APIs available within that WebView are extended with Capacitor plugins[5] which can access native APIs; these plugins allow JavaScript do the sort of things that can't be done directly in a web browser.

(Over time, most of the features provided by Capacitor end up in the web platform in some way. For example, there are web standards to allow JavaScript to detect or lock screen rotation. The Fugu project[6] is particularly avant-garde in pushing these capabilities to the web. But, the wheel of web standards grinds very finely, and for an app to have access to, say, the user's contact list now, it is pragmatic to use an extended-webview solution like Capacitor.)

We call Capacitor a "web browser *development kit*" because creating a Capacitor project effectively copies the shell of a specialized browser into an app's source tree, with separate iOS and Android versions. Building the app makes a web browser with extensions that runs the app's JS, CSS, image, and other assets. Running the app creates a WebView and invokes the app's JavaScript, which should then create the user interface using the DOM APIs that are standard in a WebView.

Capacitor is quite full-featured when it comes to native platform capabilities but is a bit bare-bones when it comes to UI. To provide a richer development environment, the Ionic Framework[7] adds a set of widgets and some application life-cycle abstractions on top of the Capacitor WebView. Ionic is not like a "normal" app framework like Vue[8] or Angular[9] because it takes advantage of Capacitor APIs, and because it tries to mimic the presentation and behavior of the native plaform (iOS or Android, mainly), so that apps built with it don't look out-of-place.

The Ionic Framework itself is built using Web Components[10], which end up creating styled DOM nodes with JavaScript behavior. These components can compose with

[1] https://ionicframework.com/
[2] https://capacitorjs.com/
[3] https://developer.apple.com/documentation/webkit/wkwebview
[4] https://developer.android.com/reference/android/webkit/WebView
[5] https://capacitorjs.com/docs/apis
[6] https://fugu-tracker.web.app/
[7] https://ionicframework.com/
[8] https://vuejs.org/
[9] https://angular.io/
[10] https://developer.mozilla.org/en-US/docs/Web/API/Web_components

other app frameworks such as Vue or Angular, allowing developers to share some of their app code between web and mobile apps—developers write the main app in Vue, and then implement the components one way on the web and another way on mobile, using something like Ionic Framework.

To recap, an Ionic app uses Ionic Framework libraries on top of a Capacitor run-time. An app could use other libraries on top of Capacitor instead of Ionic. In any case, we are most interested in Capacitor, so let's continue our focus there.

2.1.2 Evaluation

Performance has a direct effect on the experience that users have when interacting with an application, and here we would like to break down the performance in three ways: one, startup latency; two, jank; and three, peak throughput. As Capacitor is structurally much like a web browser, many of these concerns, pitfalls, and mitigation techniques are similar to those on the web.

Startup latency

Startup latency is how long the app makes the user wait after starting an app, before allowing interaction. The app may hide some of this work behind a splash screen, but if there is too much work to do at startup-time, the risk is the user might get bored or frustrated and switch away. The goal, therefore, it to minimize startup latency. Most people consider time-to-interactive of less than than a second to be sufficient; there's room to improve but for better or for worse, user expectations here are lower than they could be.

In the case of Capacitor, when a user launches an application, the app loads a minimal skeleton program that loads a WebView. On iOS and most Android systems, the rendering and JS parts of the WebView run in a separate process that has access to the app's bundled assets: JavaScript files, images, CSS, and so on. The WebView then executes the JavaScript `main` function to create the UI.

Capacitor application startup time will be dominated by parsing and compiling the JavaScript source files[11], as well as any initial work needed to boot the app framework (which could require running a significant amount of JS), and will depend on how fast the user's device is. The most effective performance mitigation here is to reduce the amount of JavaScript loaded, but this can be difficult for complex apps. The main techniques to reduce app size are toolchain-based. Tree-shaking[12], bundling[13], and minification[14] reduce the number of JS bytes that the engine needs to parse without

[11] https://infrequently.org/2021/03/the-performance-inequality-gap/
[12] https://rollupjs.org/faqs/#what-is-tree-shaking
[13] https://esbuild.github.io/api/#bundle
[14] https://esbuild.github.io/api/#minify

changing application logic. Code splitting[15] can defer some work until it is needed, but this might just result in jank later on.

Common Ionic application sizes would seem to be between 500 kB and 5 MB of JavaScript. In 2021, Alex Russell suggested that the soon-to-be standard Android performance baseline should be the Moto E7 Plus[16] which, if its performance relative to the mid-range Moto G4 phone[17] that he measured in 2019 translates to JavaScript engine speed, should be able to parse and compile uncompressed JavaScript at a rate of about 1 MB/s. That's not very much, if we want to get startup latency under a second, and a JS payload size of 5 MB would lead to 5-second startup delays for many users. Startup time would appear to be the biggest challenge for a Capacitor-based app.

(Calculation detail: The Moto G4 JavaScript parse-and-compile throughput is measured at 170 kB/s, for compressed JS. Assuming a compression ratio of 4 and that the Moto E7 Plus is 50% faster than the Moto G4, that gets us to 1 MB/s for uncompressed JS, for what was projected to be a performance baseline in 2023.)

Jank

Jank is when an application's animation and interaction aren't smooth, because the application somehow missed rendering one or more frames. Technically startup time is a form of jank, but it is useful to treat startup separately.

Generally speaking, the question of whether a Capacitor application will show jank depends on who is doing the rendering: if application JavaScript is driving an animation, then this could cause jank, in the same way as it would on the web. The mitigation is to lean on the web platform so that the WebView is the one in charge of ensuring smooth interaction, for example by using CSS animations or the native scrolling capability.

There may always be an impetus to do some animation in JavaScript, though, for example if the design guidelines require a specific behavior that can't be created using raw CSS.

Peak perf

Peak throughput is how much work an app can do per unit time. For an application written in JavaScript, this is a question of how well the JavaScript engine compiles the user's code.

Here we need to make an important aside on the strategies that a JavaScript engine uses to run a user's code. A standard technique that JS engines use to make JavaScript run fast is just-in-time (JIT) compilation, in which the engine emits specialized machine

[15] https://developer.mozilla.org/en-US/docs/Glossary/Code_splitting
[16] https://infrequently.org/2021/03/the-performance-inequality-gap/
[17] https://browser.geekbench.com/v4/cpu/compare/15987586?baseline=15667410

code at run-time and then runs that code. However, on iOS the platform prohibits JIT code generation for most applications. The usual justification for this restriction is that the low-level capability granted to a program that allows it to JIT-compile increases the likelihood of security exploits[18]. The only current exception to the no-JIT policy on iOS is for WebView (including the one in the system web browser), which iOS maintainers consider to be sufficiently sandboxed, so that any security exploit that uses JIT code generation won't be able to access other capabilities possessed by the application process.

The practical impact is that there is no JIT on iOS outside web browsers or web views; if a cross-platform application development toolkit wants peak JavaScript performance on iOS, it has to run that JS in a WebView. Capacitor does this, so it has fast JS on iOS. Note that these restrictions are not in place on Android; an app can have fast JS on Android without using a WebView, by using the system's JavaScript libraries.

Of course, actually getting peak throughput out of JavaScript is an art but the well-known techniques for JavaScript-on-the-web apply here; we won't cover them in this study.

The bridge

All of these performance observations are common to all web browsers, but with Capacitor there is the additional wrinkle that a Capacitor application can access native capabilities, for example access to a user's contacts. When application JavaScript goes to access the contacts API, Capacitor will send a message to the native side of the application over a *bridge*. The message will be serialized to JSON. Capacitor will include an implementation for the native side of the bridge into the app's source code, written in Swift for iOS or Java for Android. The native end of the bridge parses the JSON message, performs the requested action, and sends a message back with the result. Some messages may need to be proxied to the main application thread, because some native APIs can only be processed there.

The bridge does have an overhead. Apps that have high-bandwidth access to native capabilities will also have JSON encoding overhead, as well general asynchronous coordination overhead. It may even be possible that encoding or decoding a large JSON message causes the WebView to miss a frame, for example when accessing a large file in local storage.

The bridge is a necessary component to the design, though; an iOS WebView can't have direct in-process access to native capabilities. For Android, the WebView APIs do not appear to allow this either, though it is theoretically possible for an app to ship its own WebView that could access native APIs directly. In any case, the Capacitor multi-process solution does allow for some parallelism, and the enforced asynchronous nature of the APIs should lead to less modal application programming.

[18] https://wingolog.org/archives/2011/06/21/security-implications-of-jit-compilation

2.1.3 Aside: Can WebView act like native widgets?

Besides performance, one way that users experience application behavior is by way of expectations: users expect (but don't require) a degree of uniformity between different apps on the same platform, for example the same scrolling behavior or the same kinds of widgets. This aspect isn't the most important one to my investigation, because I'm more concerned with a new operating system that might come with its own design language with its own standard JavaScript-based component library, but it's one to note; an Ionic app is starting at a disadvantage relative to a platform-native app. Sometimes ensuring a platform-native UI is just a question of CSS styling and using the right library (like Ionic Framework), but sometimes it might take significant engineering or possibly even new additions to the web platform.

2.1.4 Aside: Is a WebView as good as native widgets?

As a final observation on user experience, it's worth asking the question of whether it is even possible to make a user interface using the web platform that is "as good" as native widgets. After all, the API and widget set (DOM) available to a WebView is very different from that available to a native application; is a WebView able to use CPU and GPU resources efficiently to create a smooth interface? Are web standards, CSS, and the DOM API somehow a constraint that prevents efficient user interface construction? Here we can only note the question, but without providing an answer. Past attempts to ship a web-based app story (webOS[19], Firefox OS[20]) were not successful, but the web platform has evolved since then and perhaps it is worth another try.

2.1.5 Summary

Capacitor, with or without the Ionic Framework on top, is the always-bet-on-the-web solution to the cross-platform mobile application development problem. It uses stock system WebViews and JavaScript, augmented with native capabilities as needed. Consistency with the UX of the specific platform (iOS or Android) may require significant investment, though.

Performance-wise, our evaluation is that Capacitor apps are well-positioned for peak JS performance due to JIT code generation and low jank due to offloading animations to the web platform, but may have problems with startup latency, as Capacitor needs to parse and compile a possibly significant amount of JavaScript each time it is run.

[19] https://en.wikipedia.org/wiki/WebOS
[20] https://en.wikipedia.org/wiki/Firefox_OS

2.2 React Native

Moving on to the next framework under consideration, we now dive into React Native[21]. This framework originated as a port of the popular React[22] framework as used on the web. Unlike Capacitor and React-on-the-web, which render to DOM nodes in a WebView, React Native instead renders to trees of platform-native UI widgets.

2.2.1 Overview

Background: React

At its most basic, React and React Native expose a *functional reactive* programming model. The model is *functional* in the sense that the user interface elements render as a *function* of the global application state.

React's rendering process starts with a root *element tree*, describing the root node of the user interface. An *element* is a JavaScript object with a `type` property. To render an element tree, if the value of the `type` property is a string, then the element is *terminal* and doesn't need further lowering, though React will visit any node in the `children` property of the element to render them as needed.

Otherwise if the `type` property of an element is a function, then the element node is *functional*. In that case React invokes the node's render function (the `type` property), passing the JavaScript element object as the argument. React will then recursively re-render the element tree produced as a result of rendering the component until all nodes are terminal. (Functional element nodes can instead have a class as their `type` property, but the concerns are pretty much the same.)

(In the language of React Native[23], a terminal node is a *React Host Component*, and a functional node is a *React Composite Component*, and both are *React Elements*. There are many imprecisely-used terms in React and we will continue this tradition by using the terms mentioned above.)

The rendering phase of a React application is thus a function from an element tree to a terminal element tree. Nodes of element trees can be either functional or terminal. Terminal element trees are composed only of terminal elements. Rendering lowers all functional nodes to terminal nodes. This description applies both to React (targetting the web) and React Native (which we are reviewing here).

It's probably useful to go deeper into what React does with a terminal element tree, before building to the more complex pipeline used in React Native, so here we go. The basic idea is that React-on-the-web does impedance matching between the functional description of what the UI should have, as described by a terminal element tree, and

[21] https://reactnative.dev/
[22] https://react.dev/
[23] https://reactnative.dev/architecture/glossary

the stateful tree of DOM nodes that a web browser uses to actually paint and display the UI. When rendering yields a new terminal element tree, React will compute the difference between the new and old trees. From that difference React then computes the set of imperative actions needed to mutate the DOM tree to correspond to what the new terminal element tree describes, and finally applies those changes.

In this way, small changes to the leaves of a React element tree should correspond to small changes to the DOM tree. Additionally, since rendering is a pure function of the global application state, we can avoid rendering at all when the application state hasn't changed. We'll dive into performance more deeply later on.

React Native doesn't use a WebView

React Native is similar to React-on-the-web in intent but different in structure. Instead of using a WebView on native platforms, as Ionic / Capacitor does, React Native renders the terminal element tree to platform-native UI widgets.

When a React Native functional element renders to a terminal element, it will create not just a JS object for the terminal node as React-on-the-web does, but also a corresponding C++ *shadow object*[24]. The fully lowered tree of terminal elements will thus have a corresponding tree of C++ shadow objects. React Native will then calculate the layout for each node in the shadow tree, and then *commit* the shadow tree: as on the web, React Native computes the set of imperative actions needed to change the current UI so that it corresponds to what the shadow tree describes. These changes are then applied on the main thread of the application.

The twisty path that leads one to implement JavaScript

The description above of React Native's rendering pipeline applies to the so-called "new architecture"[25], which has been in the works for some years and is only now (July 2023) starting to be deployed. The key development that has allowed React Native to move over to this architecture is tighter integration and control over its JavaScript implementation. Instead of using the platform's JavaScript engine (JavaScriptCore on iOS or V8 on Android), Facebook went and made their own whole new JavaScript implementation, Hermes[26]. Let's step back a bit to see if we can imagine why anyone in their right mind would make a new JS implementation.

In the Capacitor case study, we mention that the only way to get peak JS performance on iOS is to use the platform's WkWebView, which enables JIT compilation of JavaScript code. React Native doesn't want a WebView, though. An app could create an invisible WebView and run its JavaScript in it, but the real issue is that the interface to the JavaScript engine is so narrow as to be insufficiently expressive. The app can't cheaply,

[24] https://reactnative.dev/architecture/render-pipeline
[25] https://reactnative.dev/architecture/overview
[26] https://hermesengine.dev/

synchronously create a shadow tree of layout objects, for example, because every interaction with JavaScript has to cross a process boundary.

So, it may be that JIT is just not worth paying for, if it means having to keep JavaScript at arm's distance from other parts of the application. Without the need to use a platform's WebView, another possibility opens to us: instead of using the platform's JavaScript engine, the app could ship its own. It would be nice to use the same engine on iOS and Android, though. When React Native was first made, V8 wasn't able to operate in a mode that didn't JIT, so React Native went with JavaScriptCore on both platforms.

Having an app bundle its own JavaScript engine has the nice effect that it can easily be augmented with native extensions, for example to talk to the Swift or Java app that actually runs the main UI. That's what we describe above with the creation of the shadow tree, but that's not quite what the original React Native did; we can only speculate but we suspect that there was a fear that JavaScript rendering work (or garbage collection!) could be heavy enough to cause the main UI to drop frames. Phones were less powerful in 2016, and JavaScript engines were less good. For this reason (we think), the original React Native instead ran JavaScript in a separate thread. When a render would complete, the resulting terminal element tree would be serialized as JSON and shipped over to the "native" side of the application, which would actually apply the changes.

This arrangement did work, but it ran into problems whenever the system needed synchronous communication between native and JavaScript subsystems. As we understand it, this was notably the case when React layout would need the dimensions of a native UI widget; to avoid a stall, React would assume something about the dimensions of the native UI, and then asynchronously re-layout once the actual dimensions were known. This was particularly gnarly with regards to text measurements, which depend on low-level platform-specific rendering details.

To recap: React Native had to interpret its JS on iOS and was using a "foreign" JS engine on Android, so they weren't gaining anything by using a platform JS interpreter. They would sometimes have some annoying layout jank when measuring native components. And what's more, React Native apps would still experience the same problem as Ionic / Capacitor apps, in that application startup time was dominated by parsing and compiling the JavaScript source files.

The solution to this problem was partly to switch to the so-called "new architecture", which doesn't serialize and parse so much data in the course of rendering. But the other side of it was to find a way to move parsing and compiling JavaScript to the build phase, instead of having to parse and compile JS every time the app was run. On V8, one would do this by generating a snapshot[27]. On JavaScriptCore, which React Native used, there was no such facility. Faced with this problem and armed with Facebook's bank account, the React Native developers decided that the best solution would be to make a new JavaScript implementation optimized for ahead-of-time compilation.

[27] https://v8.dev/blog/custom-startup-snapshots

The result is Hermes[28]. There is a JavaScript parser, originally built to match the behavior of Esprima[29]; an SSA-based intermediate representation[30]; a set of basic optimizations[31]; a custom bytecode format[32]; an interpreter to run that bytecode[33]; a GC to manage JS objects[34]; and so on. Of course, given the presence of `eval`, Hermes needs to include the parser and compiler as part of the virtual machine, but the hope is that most user code will be parsed and compiled ahead-of-time.

If this were it, one might think that Hermes would be a dead end: V8 is complete; Hermes is not. For example, Hermes doesn't have `with`, async function implementation has been lagging, and so on. Why Hermes when you can V8 (with snapshots), now that V8 doesn't require JIT code generation[35]?

However, given that V8's main target isn't as an embedded library in a mobile app, perhaps the binary size question is the one differentiating factor (in theory) for Hermes. By focussing on lowering distribution size, perhaps Hermes will be a compelling JS engine in its own right. In any case, Facebook can afford to keep Hermes running for a while, regardless of whether it has a competitive advantage or not.

To repeat, this case study is an exegesis rather than a criticism: we assume that React Native is built the way it is for a reason, and we attempt to identify those reasons. In the case of Hermes, we note that as an abstract principle, if you can afford it, it's good to have code you control. For example one benefit that we see React Native getting from Hermes is control over the threading model[36]; React Native can mostly execute JS in its own thread, but interrupt that thread and switch to synchronous main-thread execution in response to high-priority events coming from the user. One might be able to do that with V8 at some point but the mobile-apps-with-JS domain is still in flux, so it's nice to have a sandbox that React Native developers can use to explore the system design space.

2.2.2 Evaluation

With that long overview out of the way, let's take a look to what kinds of performance we can expect out of a React Native system.

[28] https://hermesengine.dev/
[29] https://esprima.org/
[30] https://github.com/facebook/hermes/blob/main/include/hermes/IR/IR.h
[31] https://github.com/facebook/hermes/blob/main/include/hermes/Optimizer/PassManager/Passes.def
[32] https://github.com/facebook/hermes/blob/main/include/hermes/BCGen/HBC/BytecodeList.def
[33] https://github.com/facebook/hermes/blob/main/lib/VM/Interpreter.cpp
[34] https://hermesengine.dev/docs/hades/
[35] https://v8.dev/blog/jitless
[36] https://reactnative.dev/architecture/threading-model

Startup latency

Because React Native apps have their JavaScript code pre-compiled to Hermes bytecode, we can expect that the latency imposed by JavaScript during application startup is lower than is the case with Ionic / Capacitor, which needs to parse and compile the JavaScript at run-time.

However, it must be said that as a framework, React tends to result in large application sizes[37] and incurs significant work at startup time[38]. One of React's strengths is that it allows development teams inside an organization to compose well: because rendering is a pure function, it's easy to break down the task of making an app into subtasks to be handled by separate groups of people. Could this strength lead to a kind of weakness, in that there is less of a need for overall coordination on the project management level, such that in the end nobody feels responsible for overall application performance? We can only speculate. The concrete differences between React Native and React (the C++ shadow object tree, the multithreading design, precompilation) could mean that React Native is closer to an optimum in the design space than React.

Jank

In theory, React Native is well-positioned to avoid jank: JavaScript execution is mostly off the main UI thread. The threading model[39] changes to allow JavaScript rendering to be pre-empted onto the main thread do raise some questions, though: what if that work takes too much time, or what if there is a GC pause during that pre-emption? We would not be surprised to see an article in the next year or two from the Hermes team about efforts to avoid GC during high-priority event processing.

Another jank question relates to interactivity and latency. Say the user is dragging a UI element around the screen, and the UI needs to re-layout itself. If rendering is slow, then we might expect to see a lag between UI updates and the dragging motion; the app technically isn't dropping frames, but the render can't complete in the 16 milliseconds needed for a 60 frames-per-second update frequency.

Peak perf

But why might rendering be slow? On the one side, there is the fact that Hermes is not a high-performance JavaScript implementation. It uses a simple bytecode interpreter, and will never be able to meet the performance of V8 with JIT compilation.

However the other side of this is the design of the application framework. In the limit,

[37] https://timkadlec.com/remembers/2020-04-21-the-cost-of-javascript-frameworks/#javascript-bytes

[38] https://timkadlec.com/remembers/2020-04-21-the-cost-of-javascript-frameworks/#javascript-main-thread-time

[39] https://reactnative.dev/architecture/threading-model

React suffers from the O(n) problem: any change to the application state requires the whole element tree to be recomputed. Rendering and layout work is proportional to the size of the UI, which may have thousands of nodes.

Of course, React tries to minimize this work, by detecting subtrees whose layout does not change, by avoiding re-renders when state doesn't change, by minimizing the set of mutations to the native widget tree. But the native widgets aren't the problem: the programming model is, or it can be anyway.

2.2.3 Aside: As good as native?

Again in theory, React Native can used to write apps that are as good as if they were written directly against platform-native APIs in Kotlin or Swift, because it uses the same platform UI toolkits as native applications. React Native can also do this at the same time as being cross-platform, targetting iOS and Android with the same code. In practice, besides the challenge of designing suitable cross-platform abstractions, React Native has to grapple with potential performance and memory use overheads of JavaScript, but the result has the potential to be quite satisfactory.

2.2.4 Aside: Parallels with the browser

During the course of our investigation, we were struck when reading about React Native's rendering pipeline by how much it resembled what a browser itself will do[40] as part of the layout, paint, and render pipeline: translate a tree of objects to a tree of immutable layout objects, clip those to the viewport, paint the ones that are dirty, and composite the resulting textures to the screen.

It is striking how many many levels we have here: the element tree, the recursively expanded terminal element tree, the shadow object tree, the platform-native widget tree, surely a corresponding platform-native layout tree, and then the GPU backing buffers that are eventually composited together for the user to see. Could we do better? We could certainly imagine any of these mobile application development frameworks switching to their own Metal/Vulkan-based rendering architecture at some point, to flatten out these layers.

2.2.5 Summary

By all accounts, React Native is a real delight to program for; it makes developers happy. The challenge is to make it perform well for users. With its new rendering architecture based on Hermes, React Native may well be on the path to addressing

[40] https://developer.chrome.com/articles/renderingng-data-structures/#the-immutable-fragment-tree

many of these problems. Bytecode pre-compilation should go a long way towards solving startup latency, provided that React's expands-to-fit-all-available-space tendency is kept in check.

If one were designing a new mobile operating system from the ground up, though, we suspect that the end result would be somewhat different than React Native. At the very least, one would include Hermes and the base run-time as part of your standard library, so that every app doesn't have to incur the space costs of shipping the run-time. Also, in the same way that Android can ahead-of-time and just-in-time compile its bytecode[41], we expect that a mobile operating system based on React Native would extend its compiler with on-device post-install compilation and possibly JIT compilation as well. And at that point, why not switch back to V8?

[41] https://source.android.com/docs/core/runtime/jit-compiler

2.3 NativeScript

2.3.1 Overview

Relative to Capacitor and React Native, NativeScript[42] occupies a point in the design space which is further on the road towards the platform, unabashedly embracing the specificities of the API available on iOS and Android, exposing these interfaces directly to the application programmer.

In practice what this looks like is that a NativeScript app is a native app which simply happens to call JavaScript on the main UI thread. That JavaScript has access to *all* native APIs, directly, without the mediation of serialization or message-passing over a bridge or message queue.

Hearing this, one might reasonably doubt that NativeScript exposes *all* native APIs; after all, new versions of iOS and Android come out quite frequently, and surely it would take some effort on the part of NativeScript developers to expose the new APIs to JavaScript. But no, it really includes all of the various native APIs: the NativeScript developers wrote a build-time inspector that uses the platform's native reflection capabilities to grovel through all available APIs and to automatically generate JavaScript bindings, with associated TypeScript type definitions so that the developer knows what is available.

Some of these generated files are checked into source, so one can get an idea of the range of interfaces that are accessible to programmers; for example, see the iOS type definitions for x86-64[43]. There are bindings for everything.

Given access to all the native APIs, how should one go about making an app? One approach would be to write the same kind of program that one would write in Swift or Kotlin, but using JavaScript instead. However, this would require more than just the *ability* to access native capabilities when needed: it needs a thorough knowledge of the platform interfaces, plus NativeScript itself on top. Most people don't have this knowledge, and those that do are probably programming directly in Swift or Kotlin already.

On one level, NativeScript's approach is to take refuge in that most ecumenical of adjectives, "unopinionated". Whereas Ionic / Capacitor encourages use of web platform interfaces, and React Native only really supports React as a programming paradigm, NativeScript provides a low-level platform onto which one can layer a number of different high-level frameworks.

Trying to actually use a high-level JavaScript framework presents some challenges, as most of these frameworks are oriented to targetting the web: they take descriptions of user interfaces and translate them to the DOM. When targetting NativeScript, one

[42] https://nativescript.org/

[43] https://github.com/NativeScript/NativeScript/tree/main/packages/types-ios/src/lib/ios/objc-x86_64

could make it so that they target native UI widgets instead. However given the baked-in assumptions of how widgets should be laid out (notably via CSS), there is some impedance-matching to do between DOM-like APIs and native toolkits.

NativeScript's answer to this problem is a middle layer: a cross-platform UI library[44] that provides DOM-like abstractions and CSS layout in a way that bridges the gap between web-like and native. NativeScript apps can even define parts of their UI using a NativeScript-specific XML vocabulary[45], which NativeScript compiles to native UI widget calls at run-time[46]. Of course, there is no CSS engine in UIKit or Android's UI toolkit, so NativeScript includes its own, implemented in JavaScript of course[47].

One could program directly to this middle layer, but we suspect that its real purpose is in enabling Angular, Vue, Svelte, or the like. The pitch would be that NativeScript lets app developers use pleasant high-level abstractions, but while remaining close to the native APIs; one can always drop down for more power and expressiveness if needed.

Diving back down to the low level, as we mentioned all of the interactions between JavaScript and the native platform APIs happen on the main application UI thread. NativeScript does also allow programmers to create background threads, using an implementation of the Web Worker API[48]. One could even in theory run a React-based UI in a worker thread and proxy native UI updates to the main thread; as an unopinionated platform, NativeScript can support many different frameworks and paradigms.

Finally, there is the question of how NativeScript runs the JavaScript in an application. Recall that Ionic / Capacitor uses the native JS engine, by virtue of using the native WebView, and that React Native used to use JavaScriptCore on both platforms but now uses its own Hermes implementation. NativeScript is another point in the design space, using V8 on both platforms. (They used to use JavaScriptCore on iOS but switched to V8[49] once V8 was able to run on iOS in "jitless" mode.) Besides the reduced maintenance burden of using a single implementation on all platforms, this also has the advantage of being able to use V8 snapshots[50] to move JavaScript parse-and-compile work to build-time, even on iOS.

2.3.2 Evaluation

NativeScript is fundamentally simple: it is V8 running in an application's main UI thread, with access to all platform native APIs. So how do we expect it to perform?

[44] https://github.com/NativeScript/NativeScript/tree/main/packages/core/ui
[45] https://github.com/NativeScript/NativeScript/blob/main/apps/automated/src/xml-declaration/tns.xsd
[46] https://github.com/NativeScript/NativeScript/tree/main/packages/core/ui/builder
[47] https://github.com/NativeScript/NativeScript/blob/main/packages/core/ui/styling/css-selector.ts
[48] https://docs.nativescript.org/advanced-concepts.html#multithreading-model
[49] https://blog.nativescript.org/the-new-ios-runtime-powered-by-v8/
[50] https://v8.dev/blog/custom-startup-snapshots

Startup latency

In theory, applications with a NativeScript-like architecture should have no problem with startup time, because they can pre-compile all of their JavaScript into V8 snapshots. Snapshots are cheap to load up because they are already in a format that V8 is ready to consume.

In practice, it would seem that V8 snapshots do not perform as expected for NativeScript[51]. There are a number of aspects about this situation that we don't fully understand, which relate to the state of the tooling around V8 rather than to the fundamental approach of ahead-of-time compilation. V8 is purpose-built for Chrome, and it could be that not enough maintenance resources have been devoted to this snapshot facility.

In the meantime, NativeScript instead uses V8's code cache feature[52], which caches the result of parsing and compiling JavaScript files on the device. In this way the first time an app is installed or updated, it might start up slowly, but subsequent runs are faster. If one were designing a new operating system, one would probably want to move this work to app install-time.

As we mentioned above, NativeScript apps have access to all native APIs. That is a lot of APIs, and only some of those interfaces will actually be used by any given app. In an ideal world, we would expect the build process to only include JavaScript code for those APIs that are needed by the application. However in the presence of `eval` and dynamic property lookup, pruning the native API surface to the precise minimum is a hard problem for a bundler to perform on its own. The solution for the time being is to manually allow and deny subsets of the platform native API[53]. It's not an automatic process though, so it can be error-prone.

Besides the work that the JavaScript engine has to do to load an application's code, the other startup overhead involves whatever work that JavaScript might need to perform before the first frame is shown. In the case of NativeScript, more work is done before the initial layout than one would think: the main UI XML file is parsed by an XML parser written in JavaScript, any needed CSS files are parsed and loaded (again by JavaScript), and the tree of XML elements is translated to a tree of UI elements[54]. The layout of the items in the view tree is then computed (in JavaScript, but calling into native code to measure text and so on), and then the app is ready.

At this point, we should admit that in these studies we approach the question of app framework design from the perspective of compiler engineers rather than UI specialists. As in compilers, performance measurement and monitoring are key to UI development, but we suspect that also as in compilers there is a role for gut instinct. Incremental improvements are best driven by metrics, but qualitative leaps are often the result of

[51] https://github.com/NativeScript/NativeScript/issues/8926
[52] https://v8.dev/blog/code-caching
[53] https://docs.nativescript.org/advanced-concepts.html#metadata-filtering
[54] https://docs.nativescript.org/advanced-concepts.html#the-layout-process

somewhat ineffable hunches or even guesswork. In that spirit we can only surmise that React Native has an advantage over NativeScript in time-to-first-frame, because its layout is performed in C++ and because its element tree is computed directly from JavaScript instead of having JavaScript interpret XML and CSS files. In any case, we look forward to the forthcoming part 2 of the NativeScript and React Native performance investigations[55] that were started in November 2022.

If startup latency proves to actually be a problem in NativeScript apps using NativeScript's UI framework, we suggest leaning into something in the shape of Angular's ahead-of-time compilation mode[56], but for the middle NativeScript UI layer.

Jank

On the face of it, NativeScript is the most jank-prone of the three frameworks we have examined, because it runs JavaScript on the main application UI thread, interleaved with UI event handling and painting and all of that. If an app's JavaScript takes too long to run, the app might miss frames or fail to promptly handle an event.

On the other hand, relative to React Native, the user's code is much closer to the application's behavior. There's no asynchrony between the application's logic and its main loop: in NativeScript it is easy to identify the code causing jank and eventually fix it.

The other classic JavaScript-on-the-main-thread worry relates to garbage collection pauses. V8's garbage collector does try to minimize the stop-the-world phase by tracing the heap concurrently and leveraging parallelism during pauses[57]. Also, the user interface of a mobile app runs in an event loop, and typically spends most of its time idle; V8 exposes some API that can take advantage of this idle time to perform housekeeping tasks instead of needing to do them when handling high-priority events.

That said, having looked into the code of both the iOS and Android run-times, NativeScript does not currently take advantage of this facility. After some investigation we have determined that V8 itself seems to be in flux, as the IdleNotificationDeadline API[58] is on its way out; is the thought that concurrent tracing is largely sufficient? We expect that if conservative stack scanning[59] lands, we will see a re-introduction of this kind of API, as it does make sense to synchronize with the event loop when scanning the main thread stack.

[55] https://blog.nativescript.org/perf-metrics-universal-javascript-part1/
[56] https://angular.io/guide/aot-compiler
[57] https://v8.dev/blog/trash-talk
[58] https://chromium.googlesource.com/v8/v8/+/refs/heads/main/include/v8-isolate.h#1281
[59] https://groups.google.com/g/v8-reviews/c/5ptLig88soA?pli=1

Peak performance

As we have seen in our previous evaluations, question of peak performance boils down to "is the JavaScript engine state-of-the-art, and can it perform just-in-time compilation". In the case of NativeScript, the answers are yes and maybe, respectively: V8 is state-of-the-art, and it can JIT on Android, but not on iOS.

Perhaps the mitigation here is that the hardware that iOS runs on tends to be significantly more powerful than median Android devices; if we have to pick a subset of users to penalize with an interpreter-only run-time, people with iPhones are the obvious choice, because they can afford it.

2.3.3 Aside: Are markets wise?

Recall that our perspective in this series is that of the designer of a new JavaScript-based mobile development platform. We are trying to answer the question of what would it look like if a new platform offered a NativeScript-like experience. In this regard, only the structure of NativeScript is of interest, and notably its "market success" is not relevant, except perhaps in some Hayekian conception of the world in which markets are necessarily smarter than any one of us.

It must be said, though, that React Native is the 800-pound gorilla of JavaScript mobile application development. The 2022 State of JS survey[60] shows that among survey respondents, more people are aware of React Native than any other mobile framework, and people are generally more positive about React Native than other frameworks. Does NativeScript's mitigated market share indicate something about its architecture, or does it speak speak more to the size of Facebook's budget, both on the developer experience side and on marketing?

2.3.4 Aside: On the expressive power of application framworks

The answer to the market wisdom question might be found in a 35-year-old computer science paper, "On the expressive power of programming languages"[61][62].

In this paper, Matthias Felleisen considers the notion of what it means for one programming language to be more expressive than another. For example, is a language with just `for` less expressive than a language with both `for` and `while`? Intuitively we would say no, these are similar things; one can make a simple local transformation of `while (x) {...}` to `for (;x;) {...}`, yielding exactly the same program semantics. On the other hand a language with just `for` is less expressive than one which also has `goto`; there is no simple local rewrite that can turn `goto` into `for`.

[60] https://2022.stateofjs.com/en-US/libraries/mobile-desktop/
[61] https://dl.acm.org/doi/10.1016/0167-6423%2891%2990036-W
[62] https://www.sciencedirect.com/science/article/pii/016764239190036W/pdf

In the same way, we can consider the question of what it would mean for one library to be more expressive than another. After all, the API of a library exposes a language in which its user can write programs; we should be able to reason about these languages. So between React Native and NativeScript, which one is more expressive?

By Felleisen's definitions, NativeScript is clearly the more expressive language: there is no simple local transformation that can turn imperative operations on native UI widgets into equivalent functional-reactive programs. Yes, with enough glue code React Native can reach directly to native APIs in a similar way as NativeScript, but everything that touches the native UI tree is expressly under React Native's control: there is no sanctioned escape hatch.

One might think that "more expressive" is always better, but Felleisen's take is more nuanced than that. Yes, he says, more expressive languages do allow programmers to make more concise programs, because they allow programmers to define abstractions that encapsulate patterns, and this is a good thing. However he also concludes that "an increase in expressive power is related to a decrease of the set of 'natural' (mathematically appealing) operational equivalences." Less expressive programming languages are easier to reason about, in general, and indeed that is one of the recognized strengths of React's programming model: it is easy to compose components and have confidence that the result will work.

2.3.5 Summary

A NativeScript-like architecture offers the possibility of performance: the developer has all the capabilities needed for writing pleasant-to-use applications that blend in with the platform-native experience. It is up to the developers to choose how to use the power at their disposal. In the wild, we expect that the low-level layer of NativeScript's API is used mainly by expert developers, who know how to assemble well-functioning machines from the parts on offer.

As a primary programming interface for a new JavaScript-based mobile platform, though, just providing a low-level API would seem to be not enough. NativeScript rightly promotes the use of more well-known high-level frameworks on top: Angular, Vue, Svelte, and such. Less experienced developers should use an opinionated high-level UI framework; these developers don't have good opinions yet and the API should lead them in the right direction.

2.4 Flutter

Flutter[63] is a departure from the previous frameworks in that it is based not on JavaScript but on the Dart[64] language.

2.4.1 Overview

We take three approaches to understanding Flutter, coming at it first from the past, then from the top, and finally from the bottom.

The present, from the past

With the other frameworks we looked at, we didn't have to say much about their use of JavaScript. JavaScript is an obvious choice, in 2023 at least: it is ubiquitous, has high quality implementations, and as a language it is quite OK and progressively getting better. Up to now, "always bet on JS" has had an uninterrupted winning streak.

But winning is not the same as unanimity, and Flutter and Dart represent an interesting pole of contestation. To understand how we got here, we have to go back in time. Ten years ago, JavaScript just wasn't a great language: there were no modules, no async functions, no destructuring, no classes, no extensible iteration, no optional arguments to functions. In addition it was hobbled with a significant degree of what can only be called accidental sloppiness: `with` which can dynamically alter a lexical scope, direct `eval` that can define new local variables, `Function.caller`, and so on. Finally, larger teams were starting to feel the need for more rigorous language tooling that could use types to prohibit some classes of invalid programs.

All of these problems in JavaScript have been addressed over the last decade, mostly successfully. But in 2010 or so if you were a virtual machine engineer, you might look at JavaScript and think that in some other world, things could be a lot better. That's effectively what happened: the team that originally built V8 broke off and started to work on what became Dart.

Initially, Dart was targetted for inclusion in the Chrome web browser as an alternate "native" browser language. This didn't work, for various reasons, but since then Dart grew the Flutter UI toolkit, which has breathed new life into the language. And this is a review of Flutter, not a review of Dart, not really anyway; we consider Dart to be spiritually another JavaScript, different but in the same family. Dart's implementation has many interesting aspects as well that we'll get into later on, but all of these differences are incidental: they could just as well be implemented on top of JavaScript, TypeScript, or another source language in that family. Even if Flutter isn't strictly part of the JavaScript-based mobile application development frameworks that we are

[63] https://flutter.dev/
[64] https://dart.dev/

comparing, it is valuable to the extent that it shows what is possible, and in that regard there is much to say.

Flutter, from the top

At its most basic, Flutter is a UI toolkit for Dart. In many ways it is like React. Like React, its interface follows the functional-reactive paradigm: programmers describe the "what", and Flutter takes care of the "how". Also, like the phenomenon in which new developers can learn React without really knowing JavaScript, Flutter is the killer app for Dart: Flutter developers mostly learn Dart at the same time that they pick up Flutter.

In some other ways, Flutter is the logical progression of React, going in the same direction but farther along. Whereas React-on-the-web takes the user's declarative specifications of what the UI should look like and lowers them into DOM trees, and React Native lowers them to platform-native UI widgets, Flutter has its own built-in layout, rasterization, and compositing engine: Flutter draws all the pixels.

This has the predictable challenge that Flutter has to make significant investments so that its applications don't feel out-of-place on their platform, but on the other hand it opens up a huge space for experimentation and potential optimization: Flutter has the potential to beat native at its own game. Recall that with React Native, the result of the render-commit-mount process[65] is a tree of native widgets. The native platform will surely then perform a kind of layout on those widgets, divide them into layers that correspond to GPU textures, paint those layers, then composite them to the screen—basically, what a web engine will do[66].

What if we could instead skip the native tree and go directly to the lower GPU layer? That is the promise of Flutter. Flutter has the potential to create much more smooth and complex animations than the other application development frameworks we have mentioned, with lower overhead and energy consumption.

In practice… that's always the question, isn't it? Our understanding is that Flutter mostly lives up to its promise, but with one significant qualification which we'll get to in a minute. But before that, let's traverse Flutter from the other direction, coming up from Dart.

Dart, from the bottom

To explain some aspects of Dart we should understand how it is that its developers decided to make a new language, and what decisions guided them in this way.

Let's say you are the team that originally developed V8, and you decide to create a new language. You write a new virtual machine that looks like V8, taking Dart source

[65] https://reactnative.dev/architecture/render-pipeline
[66] https://developer.chrome.com/articles/layoutng/

code as input and applying advanced adaptive compilation techniques to get good performance. You can even be faster than JS because your language is just a bit more rigid than JavaScript is: you have traded off expressivity for performance. (Recall from our NativeScript case study that expressivity isn't a value judgment: there can be reasons to pay for more "mathematically appealing operational equivalences", in Felleisen's terminology, in exchange for applying more constraints on a language.)

But, you fail to ship the VM in a browser[67]; what do you do? The project could die; that would be annoying, but you work for Google, so it happens all the time. However, a few interesting things happen around the same time that will cause you to pivot. One is a concurrent experiment by Chrome developers to pare the web platform down to its foundations and rebuild it. This effort will eventually become Flutter; while it was originally based on JS[68], eventually they will choose to switch to Dart[69].

The second thing that happens is that recently-invented smart phones become ubiquitous. Most people have one, and the two platforms are iOS and Android. Flutter wants to target them. You are looking for your niche, and you see that mobile application development might be it. As the Flutter people continue to experiment, you start to think about what it would mean to target mobile devices with Dart.

The initial Dart VM was made to JIT[70], but as we know, Apple doesn't let people do this on iOS. So instead you look to write a quick-and-dirty ahead-of-time compiler, based on your JIT compiler that takes your program as input, parses and baseline-compiles it, and generates an image that can be loaded at runtime. It ships on iOS. Funnily enough, it ships on Android too, because AOT compilation allows you to avoid some startup costs; forced to choose between peak performance via JIT and fast startup via AOT, you choose fast startup.

It's a success, you hit your product objectives, and you start to look further to a proper native-code ahead-of-time compiler that can stand alone without the full Dart run-time. After all, if you have to compile at build-time, you might as well take the time to do some proper optimizations[71]. You actually change the language to have a sound typing system[72] so that the compiler can make program transformations that are valid as long as it can rely on the program's types.

Incidentally, it would seem that historically the shift to a sound type system actually started before Flutter and thus before AOT, because of a Dart-to-JavaScript compiler that you inherited from the time in which you thought the web would be the main target. The Dart-to-JS compiler used to be a whole-program compiler; this enabled it to do flow-sensitive type inference, resulting in faster and smaller emitted JavaScript. But whole-program compilation doesn't scale well in terms of compilation time, so Dart-to-

[67] https://lists.webkit.org/pipermail/webkit-dev/2011-December/018775.html
[68] https://github.com/flutter/flutter/commit/00882d626a478a3ce391b736234a768b762c853a
[69] https://bugs.chromium.org/p/chromium/issues/detail?id=454613
[70] https://mrale.ph/talks/vmil2020/#/3
[71] https://github.com/dart-lang/sdk/issues/30480
[72] https://medium.com/dartlang/dart-and-the-performance-benefits-of-sound-types-6ceedd5b6cdc

JS switched to separate per-module compilation. But then you lose lots of types! The way to recover the fast-and-small-emitted-JS property was through a stronger, sound type system for Dart.

At this point, you still have your virtual machine, plus your ahead-of-time compiler, plus your Dart-to-JS compiler. It is not a bad situation to be in, in 2023: you can offer a good development experience via the just-in-time compiled virtual machine. Apparently you can even use the JIT on iOS in developer mode, because attaching `ptrace` to a binary allows for native code generation. Then when you go to deploy, you make a native binary that includes everything.

For the web, you also have your nice story, even nicer than with JavaScript in some ways: because the type checker and ahead-of-time compiler are integrated in Dart, you don't have to worry about WebPack or Vite or minifiers or uglifiers or TypeScript or JSX or Babel or any of the other things that JavaScript people are used to. Granted, the tradeoff is that innovation is mostly centralized with the Dart maintainers, but currently Google seems to be investing enough so that's OK.

Stepping back, this story is not unique to Dart; many of its scenes also played out in the world of JavaScript over the last 5 or 10 years as well. Hermes[73] does ahead-of-time compilation (as does QuickJS[74], for that matter), albeit only to bytecode, and V8's snapshot facility is a form of native AOT compilation. But the tooling in the JavaScript world is more diffuse than with Dart. With the perspective of developing a new JavaScript-based mobile operating system in mind, the advantages that Dart (and thus Flutter) has won over the years are also on the table for JavaScript to win. Perhaps even TypeScript could eventually migrate to have a sound type system, over time; it would take a significant investment but the JS ecosystem does evolve, if slowly.

2.4.2 Evaluation

So how do we expect Flutter applications to perform? If we were writing a new mobile OS based on JavaScript, what would it mean in terms of performance to adopt a Flutter-like architecture?

Startup latency

Flutter applications are well-positioned to start fast, with ahead-of-time compilation. However they have had problems realizing this potential[75], with many users seeing a big stutter when they launch a Flutter app.

To explain this situation, consider the structure of a typical low-end Android mobile device: there are a small number of not-terribly-powerful CPU cores, but attached to

[73] https://hermesengine.dev/
[74] https://bellard.org/quickjs/
[75] https://github.com/flutter/flutter/projects/188

the same memory there is also a decent GPU with many cores. For example, the SoC in the low-end Moto E7 Plus[76] has 8 CPU cores and 128 GPU cores (texture shader units). One could paint widget pixels into memory from either the CPU or the GPU, but it's better to do it in the GPU because it has so many more cores: in the time it takes to compute the color of a single pixel on the CPU, on the GPU you could do even 128 times as many, given that the comparison is often between multi-threaded rasterization on the GPU versus single-threaded rasterization on the CPU.

Flutter has always tried to paint on the GPU. Historically it has done so via a GPU backend to the Skia graphics library, notably used by Chrome among other projects. But, Skia's API is a drawing API, not a GPU API; Skia is the one responsible for configuring the GPU to draw what we want. And here's the problem: configuring the GPU takes time. Skia generates shader code at run-time for rasterizing the specific widgets used by the Flutter programmer. That shader code then needs to be compiled to the language the GPU driver wants, which looks more like Vulkan[77] or Metal[78]. The process of compilation and linking takes time, potentially seconds, even.

The solution to "too much startup shader compilation" is much like the solution to "too much startup JavaScript compilation": move this phase to build time. The new Impeller[79] rendering library does just that. However to do that, it had to change the way that Flutter renders: instead of having Skia generate specialized shaders at run-time, Impeller instead lowers the shapes that it draws to a fixed set of primitives, and then renders those primitives using a smaller, fixed set of shaders[80]. These primitive shaders are pre-compiled at build time and included in the binary. By switching to this new renderer, Flutter should be able to avoid startup jank.

Jank

Of all the application development frameworks we have considered, we consider that Flutter is the best positioned to avoid jank. It has the React-like asynchronous functional layout model, but "closer to the metal"; by skipping the tree of native UI widgets, it can potentially spend less time for each frame render.

For Flutter on iOS, the shell of the application is actually written in Objective C++. On Android it's the same, except that it's Java. Starting up the app runs the shell, which then creates a FlutterView widget and spawns a new thread to actually run Flutter (and the user's Dart code). Mostly, Flutter runs on its own, rendering frames to the GPU resources backing the FlutterView directly.

If a Flutter app needs to communicate with the platform, it passes messages across

[76] https://www.cpu-monkey.com/en/cpu-qualcomm_snapdragon_460
[77] https://www.vulkan.org/
[78] https://developer.apple.com/metal/
[79] https://docs.flutter.dev/perf/impeller
[80] https://github.com/flutter/engine/tree/main/impeller/entity/shaders

an asynchronous channel back to the main thread[81]. Although these messages are asynchronous, this is probably the largest potential source of jank in a Flutter app, outside the initial frame paint: any graphical update which depends on the answer to an asynchronous call may lag.

Peak performance

Dart's type system and ahead-of-time compiler optimize for predictable good performance rather than the more variable but potentially higher peak performance that could be provided by just-in-time compilation.

This story should probably serve as a lesson to any future platform. The people that developed the original Dart virtual machine had a built-in bias towards just-in-time compilation, because it allows the VM to generate code that is specialized not just to the program but also to the problem at hand. A given system with ahead-of-time compilation can always be made to perform better via the addition of a just-in-time compiler, so the initial focus was on JIT compilation. On iOS of course this was not possible, but on Android and other platforms where this was available it was the default deployment model.

However, even Android switched to ahead-of-time compilation instead of the JIT model in order to reduce startup latency: doing any machine code generation at all at program startup was more work than was needed to get to the first frame. One could add JIT back again on top of AOT but it does not appear to be a high priority.

We expect that Capacitor could beat Dart in some raw throughput benchmarks, given that Capacitor's JavaScript implementation can take advantage of the platform's native JIT capability. Does it matter, though, as long as you are hitting your frame budget? This is an open question.

2.4.3 Aside: An escape hatch to the platform

What happens if you want to embed a web view into a Flutter app?

The answer is unsatisfactory: for better or for worse, at this point it is too expensive even for Google to make a new web engine. Therefore Flutter will have to embed the native WebView. However Flutter runs on its own threads; the native WebView has its own process and threads but its interface to the app is tied to the main UI thread.

Therefore either Flutter needs to make the native WebView (or indeed any other native widget) render itself to (a region of) Flutter's GPU backing buffer, or otherwise copy the native widget's pixels into their own texture and then composite them in Flutter-land. It's not an ideal arrangement. The Android[82] and iOS[83] platform view documentation

[81] https://docs.flutter.dev/development/platform-integration/platform-channels
[82] https://docs.flutter.dev/development/platform-integration/android/platform-views
[83] https://docs.flutter.dev/development/platform-integration/ios/platform-views

discuss some of the tradeoffs and mitigations.

2.4.4 Aside: For want of a canvas

There is a very funny situation in the React Native world in which, if the application programmer wants to draw to a canvas, they have to embed a whole WebView into the React Native app[84] and then proxy the canvas calls into the WebView[85]. Flutter is happily able to avoid this problem, because it includes its own drawing library with a canvas-like API. Of course, Flutter also has the luxury of defining its own set of standard libraries instead of necessarily inheriting them from the web, so when and if they want to provide equivalent but differently-shaped interfaces, they can do so.

Flutter manages to be more expressive than React Native in this case, without losing much in the way of understandability. Few people will have to reach to the canvas layer, but it is nice to know it is there.

2.4.5 Summary

Dart and Flutter are terribly attractive from an engineering perspective. They offer a delightful API and a high-performance, flexible runtime with a built-in toolchain. Could this experience be brought to a new mobile operating system as its primary programming interface, based on JavaScript? React Native is giving it a try, but we think there may be room to take things further to own the application from the program all the way down to the pixels.

[84] https://github.com/react-native-webview/react-native-webview/blob/master/docs/Guide.md#communicating-between-js-and-native

[85] https://github.com/iddan/react-native-canvas#readme

2.5 Ark

2.5.1 Overview

Ark is a new JavaScript-based mobile development platform.

To a first approximation, Ark—or rather, what we are calling Ark; we don't actually know the name for the whole architecture—is a loosely Flutter-like UI library implemented on top of a dialect of JavaScript, with build-time compilation to bytecode (like Hermes) but also with support for just-in-time and ahead-of-time compilation of bytecode to native code. It is made by Huawei.

How did we get here?

Huawei had been working for a while on a compiler and language run-time called Ark Compiler[86], with the goal of getting better performance out of Android applications, back when they were shipping Android. As we understand it, this compiler took the Java / Dalvik / Android Run Time bytecodes as its input, and outputted native binaries along with a new run-time implementation.

For political reasons, Huawei was forced to pivot away from Android for their smartphone operating system. To differentiate their OS from Android, they chose to take their existing Ark compiler toolchain and retarget it as a JavaScript compiler and run-time, and to build a UI framework on top of that.

In the end, Huawei actually built two different UI frameworks: something web-like and something like Flutter.

The web-like programming interface specifies UIs using an XML dialect, HML[87], and styles the resulting node tree with CSS. Apps augment these nodes with JavaScript behavior; the main app is a set of DOM-like event handlers[88]. There is an API to dynamically create DOM nodes[89], but unlike the other systems we have examined, the HarmonyOS documentation doesn't really sell the reader on using a high-level framework like Angular.

The web-like programming interface is not a terribly compelling framework, from a technical perspective; the programming model is more like what was available back in the DHTML days[90]. We wouldn't expect people to be able to make rich applications that delight users, given these primitives, though CSS animation and the HML loop

[86] https://www.huaweicentral.com/ark-compiler-huaweis-self-developed-android-application-compiler-explained/
[87] https://developer.harmonyos.com/en/docs/documentation/doc-guides-V3/js-framework-syntax-hml-0000001477981005-V3
[88] https://developer.harmonyos.com/en/docs/documentation/doc-guides-V3/js-framework-syntax-js-0000001428061552-V3
[89] https://developer.harmonyos.com/en/docs/documentation/doc-references-V3/js-components-create-elements-0000001478181509-V3?catalogVersion=V3
[90] https://en.wikipedia.org/wiki/Dynamic_HTML

and conditional rendering[91] from the template system might be just expressive enough for simple applications.

The more interesting side is the so-called "declarative" UI programming model which exposes a Flutter/React-like interface. The programmer describes the "what" of the UI by providing a tree of UI nodes in its `build` function, and the framework takes care of calling `build` when necessary and of rendering that tree to the screen.

Here we need to show some example code, because it is different from standard JavaScript, seemingly taking its inspiration from SwiftUI[92]. A small example from the fine manual[93]:

```
@Entry
@Component
struct MyComponent {
  build() {
    Stack() {
        Image($rawfile('Tomato.png'))
        Text('Tomato')
            .fontSize(26)
            .fontWeight(500)
    }
  }
}
```

The `@Entry` decorator (*) marks this `struct` (**) as being the main entry point for the app. `@Component` marks it as being a component, like a React *functional component*. Components conform to an interface (***) which defines them as having a `build` method which takes no arguments and returns no values: it creates the tree in a somewhat imperative way.

But as we can see the flavor is somewhat declarative, so how does that work? Also, `build() { ... }` looks syntactically a lot like `Stack() { ... }`; what's the deal, are they the same?

Before going on to answer this, note the asterisks above: these are concepts that aren't in JavaScript. Indeed, programs written for HarmonyOS's declarative framework aren't JavaScript; they are in a dialect of TypeScript that Huawei calls ArkTS. In this case, an `interface` is a TypeScript concept[94]. Decorators would appear to correspond to an experimental TypeScript feature[95], looking at the source code.

[91] https://developer.harmonyos.com/en/docs/documentation/doc-references-V3/js-components-container-list-0000001427584924-V3#EN-US_TOPIC_0000001544695117__example

[92] https://developer.apple.com/xcode/swiftui/

[93] https://developer.harmonyos.com/en/docs/documentation/doc-guides-V3/ui-ts-creating-simple-page-0000001493575800-V3

[94] https://www.typescriptlang.org/docs/handbook/interfaces.html

[95] https://www.typescriptlang.org/docs/handbook/decorators.html

But `struct` is an ArkTS-specific extension[96], and Huawei has actually extended the TypeScript compiler to specifically recognize the `@Component` decorator, such that when one "calls" a struct, for example as above in `Stack() { ... }`, TypeScript will parse that as a new expression type `EtsComponentExpression`[97], which may optionally be followed by a block. When `Stack()` is invoked, its children (instances of `Image` and `Text`, in this case) will be populated via running the block.

Now, though TypeScript isn't everyone's cup of tea, it is quite normalized in the JavaScript community and not a hard sell. Language extensions like the handling of `@Component` pose a more challenging problem. Still, Facebook managed to sell people on JSX[98], so perhaps Huawei can do the same for their dialect. More on that later.

Under the hood, it would seem that we have a similar architecture to Flutter: invoking the components[99] creates a corresponding tree of *elements*[100] (as with React Native's shadow tree), which then are lowered to *render nodes*[101], which draw themselves onto layers using Skia, in a multi-threaded rendering pipeline[102]. Underneath, the UI code actually re-uses some parts of Flutter, though from what I can tell HarmonyOS developers are replacing those over time.

Restrictions and extensions

So we see that the source language for the declarative UI framework is TypeScript, but with some extensions. It also has its restrictions, and to explain these, we have to talk about implementation.

Of the JavaScript mobile application development frameworks we discussed, Capacitor and NativeScript used "normal" JS engines from web browsers, while React Native built their own Hermes implementation. Hermes is also restricted, in a way, but mostly inasmuch as it lags the browser JS implementations; it relies on source-to-source transpilers to get access to new language features. ArkTS—that's the name of HarmonyOS's "extended TypeScript" implementation—has more fundamental restrictions.

The Ark compiler was originally built as an alternate compiler and virtual machine for Android apps. These targets don't really have the ability to load new Java or Kotlin source code at run-time. In Java there are class loaders, but those load bytecode;

[96] https://gitee.com/openharmony/third_party_typescript#changes
[97] https://gitee.com/openharmony/third_party_typescript/blob/master/src/compiler/types.ts#L2065
[98] https://legacy.reactjs.org/docs/introducing-jsx.html
[99] https://gitee.com/openharmony/arkui_ace_engine/blob/master/frameworks/core/pipeline/base/component.h
[100] https://gitee.com/openharmony/arkui_ace_engine/blob/master/frameworks/core/pipeline/base/element.h
[101] https://gitee.com/openharmony/arkui_ace_engine/blob/master/frameworks/core/pipeline/base/render_node.h
[102] https://gitee.com/openharmony/arkui_ace_engine/blob/master/frameworks/core/pipeline/pipeline_context.h

Android devices don't have to deal with the Java source language. If we use a similar architecture for JavaScript, though, what do we do about `eval`?

ArkTS's answer is: don't. As in, `eval` is not supported on HarmonyOS. In this way the implementation of ArkTS can be divided into two parts: a frontend that produces bytecode, and a runtime that runs the bytecode; Ark never has to deal with the source language on the device where the runtime is running. Like Hermes, the developer produces bytecode when building the application and ships it to the device for the runtime to handle.

Incidentally, before we move on to discuss the runtime, there are actually two front-ends that generate ArkTS bytecode: one written in C++ that seems to only handle standard TypeScript and JavaScript[103], and one written in TypeScript that also handles "extended TypeScript"[104]. The former has a test262 runner with about 10k skipped tests[105], and the latter doesn't appear to have a test262 runner.

The ArkTS runtime[106] is itself built on a non-language-specific common Ark runtime[107], and the set of supported instructions is the union of the core ISA[108] and the JavaScript-specific instructions[109]. Bytecode can be interpreted[110], JIT-compiled, or AOT-compiled.

On the side of design documentation, it's somewhat sparse. There are some core design docs[111]; readers may be interested in the rationale to use a bytecode interface[112] for Ark as a whole, or the optimization overview[113].

Indeed ArkTS as a whole has a surfeit of optimizations, to the extent that one wonders which are actually needed. There are source-to-source optimizations on bytecode[114], which we expect are useful when generating ArkTS bytecode from JavaScript, where the JavaScript front-end probably doesn't have a full compiler implementation. There is a completely separate optimizer[115] in the eTS part of the run-time, based on what would appear to be a novel "circuit-based" IR[116] that bears some similarity to sea-of-

[103] https://gitee.com/openharmony/arkcompiler_ets_frontend/tree/master/es2panda
[104] https://gitee.com/openharmony/arkcompiler_ets_frontend/tree/master/ts2panda
[105] https://gitee.com/openharmony/arkcompiler_ets_frontend/blob/master/es2panda/test/test262skiplist.txt
[106] https://gitee.com/openharmony/arkcompiler_ets_runtime
[107] https://gitee.com/openharmony/arkcompiler_runtime_core
[108] https://gitee.com/openharmony/arkcompiler_runtime_core/blob/master/isa/isa.yaml
[109] https://gitee.com/openharmony/arkcompiler_ets_runtime/blob/master/ecmascript/ecma_isa.yaml
[110] https://gitee.com/openharmony/arkcompiler_ets_runtime/blob/master/ecmascript/interpreter/interpreter-inl.h
[111] https://gitee.com/openharmony/arkcompiler_runtime_core/tree/master/compiler/docs
[112] https://gitee.com/openharmony/arkcompiler_runtime_core/blob/master/docs/rationale-for-bytecode.md
[113] https://gitee.com/openharmony/arkcompiler_runtime_core/blob/master/docs/ir_format.md
[114] https://gitee.com/openharmony/arkcompiler_runtime_core/tree/master/bytecode_optimizer
[115] https://gitee.com/openharmony/arkcompiler_ets_runtime/blob/master/ecmascript/compiler/pass.h
[116] https://gitee.com/openharmony/arkcompiler_ets_runtime/blob/master/ecmascript/compiler/

nodes. Finally the whole thing appears to bottom out in LLVM[117], which of course has its own optimizer. We can only assume that this situation is somewhat transitory. Also, ArkTS does appear to generate its own native code sometimes, notably for inline cache stubs.

Of course, when it comes to JavaScript, besides the fundamental language semantics, there is also a large and growing standard library. In the case of ArkTS, this standard library is part of the run-time[118], like the interpreter, compilers, and the garbage collector (generational concurrent mark-sweep with optional compaction[119]).

2.5.2 Evaluation

We are struck by how *expensive* a proposition it is to change everything about an application development framework. What is it that we can imagine that Huawei bought with the thousand or more person-years of investment that it might take to implement Ark? Towards what other local maximum might we be heading?

Startup latency

We didn't mention it before, but it would seem that one of the goals of HarmonyOS is in the name: Huawei wants to harmonize development across the different range of deployment targets. To the extent possible, it would be nice to be able to write the same kinds of programs for IoT devices as for feature-rich smartphones and tablets and the like. In that regard one can see through all the source code how there is a culture of doing work ahead-of-time and preventing work at run-time; for example see the design doc for the interpreter[120], or for the file format[121], or indeed the lack of JavaScript `eval`.

Of course, this wide range of targets also means that the HarmonyOS platform bears the burden of a high degree of abstraction; not only can one change the kernel, but also the JavaScript engine (using JerryScript[122] on "lite" targets).

We mention this background because sometimes in news articles and indeed official communication from recent years there would seem to be some confusion that HarmonyOS is just for IoT, or aimed to be super-small, or something. In this evaluation we focus on the feature-rich side of things, and there our understanding is that the developer will generate bytecode ahead-of-time. When an app is installed on-device,

circuit_ir_specification.md

[117] https://gitee.com/openharmony/arkcompiler_ets_runtime/blob/master/ecmascript/compiler/llvm_ir_builder.h

[118] https://gitee.com/openharmony/arkcompiler_ets_runtime/tree/master/ecmascript

[119] https://gitee.com/openharmony/arkcompiler_ets_runtime/tree/master/ecmascript/mem

[120] https://gitee.com/openharmony/arkcompiler_runtime_core/blob/master/docs/design-of-interpreter.md

[121] https://gitee.com/openharmony/arkcompiler_runtime_core/blob/master/docs/file_format.md

[122] https://jerryscript.net/

the AOT compiler will turn it into a single ELF image. This should generally lead to fast start-up.

However it would seem that the rendering library[123] that paints UI nodes into layers and then composits those layers uses Skia in the way that Flutter did pre-Impeller, which to be fair is a quite recent change to Flutter. We expect therefore that Ark (ArkTS + ArkUI) applications also experience shader compilation jank at startup, and that they may be well-served by tesellating their shapes into primitives like Impeller does so that they can precompile a fixed, smaller set of shaders.

Jank

ArkUI's fundamental architectural similarity to Flutter leads us to think that jank will not be a big issue. There is a render thread that is separate from the ArkTS thread, so like with Flutter, async communication with main-thread interfaces is the main jank worry. And on the ArkTS side, ArkTS even has a number of extensions to be able to share objects between threads without copying, should that be needed. We have not been able to determine how well-developed and well-supported these extensions are, though.

It is interesting that unlike iOS or Android, from what we can gather from the open source OpenHarmony project, HarmonyOS is *only* exposing these "web-like" and "declarative" UI frameworks for app development. This makes it so that the same organization is responsible for the software from top to bottom, which can lead to interesting cross-cutting optimizations: functional reactive programming isn't just a developer-experience narrative, but it can directly affect the shape of the rendering pipeline. If there is jank, someone in the building is responsible for it and should be able to fix it, whether it is in the GPU driver, the kernel, the ArkTS compiler, or the application itself.

Peak performance

As part of this study, we have not been able to evaluate ArkTS for peak performance. That said, although there is a JIT compiler, it does not seem that it is as tuned for adaptive optimization as V8 is.

At the same time, we find it interesting that HarmonyOS has chosen to modify JavaScript. While it is doing that, could they switch to a sound type system, to allow the kinds of AOT optimizations that Dart can do? It would be an interesting experiment.

As it is, though, we expect that ArkTS is well-positioned for predictably good performance with AOT compilation, although this evaluation needs verification.

[123] https://gitee.com/openharmony/arkui_ace_engine/tree/master/frameworks/core/pipeline/base

2.5.3 Aside: On the importance of storytelling

In these studies we aim to be charitable towards the frameworks that we review, to give credit to what they are trying to do, even while noting where they aren't currently there yet. That's part of why we need a plausible narrative for how the frameworks got where they are, because that lets us have an idea of where they are going.

In that sense we consider Ark to be at an interesting inflection point. An initial contact with ArkUI and HarmonyOS via the official documentation is disheartening—there are too many architectural box diagrams, too many generic descriptions of components, too many promises with buzzwords. One gets the impression that the project was trying to justify itself to a kind of clueless management chain. Is there actually anything there?

But in light of Ark's adoption of a modern rendering architecture and a bold new implementation of JavaScript, and its willingness to experiment with the language, we consider that there is an interesting story to be told, but this time not to management but to app developers.

Of course one wouldn't want to market to app developers when the system is still a mess because you haven't finished rebuilding an MVP yet. Retaking our charitable approach, then, we interpret the architectural box diagrams as a clever blind to avoid exciting outside interest while the app development kit wasn't ready yet. As and when the system starts working well, presumably over the next year or so, we expect HarmonyOS to invest much more heavily in marketing and developer advocacy; the story is interesting, but it needs to be told to be heard.

2.5.4 Aside: O platform, my platform

All of the previous app development frameworks that we looked at were cross-platform; Ark is not. It could be, of course: it does appear to be thoroughly open source. But HarmonyOS devices are the main target. What implications does this have?

A similar question arises in perhaps a more concrete way if we start with the mature Flutter framework: what would it mean to make a Flutter phone?

The first thought that comes to mind is that having a Flutter OS would allow for the potential for more cross-cutting optimizations that cross abstraction layers. But then, what does Flutter really need? It has the GPU drivers, and we aren't going to re-implement those. It has the bridge to the platform-native SDK, which is not such a large and important part of the app. Flutter gets input from the platform, but that's also not so specific. So maybe optimization is not the answer.

On the other hand, a Flutter OS would not have to solve the make-it-look-native problem; because there would be no other "native" toolkit, Flutter apps won't look out of place. That's nice. It's not something that would make the *platform* compelling, though.

HarmonyOS does have this embryonic concept of app mobility, where users can put an

app from their phone on their fridge, or something. Clearly we are not doing it justice here, but let's assume it's a compelling use case. In that situation it would be nice for all devices to present similar abstractions, so users could somehow install the same app on two different kinds of devices, and they could communicate to transfer data.

One reasonable way to "move" an app is to have it stay running on the user's phone and the phone just communicates pixels with the user's fridge (or whatever); this is the low-level solution. It seems however that HarmonyOS appears to be going for the higher-level solution where the app actually runs logic on the device. In that case it would make sense to ship UI assets and JavaScript / extended TypeScript bytecode to the device, which would run the app with an interpreter (for low-powered devices) or use JIT/AOT compilation. The Ark runtime itself would live on all devices, specialized to their capabilities.

In a way this is the Apple watchOS solution; developers publish their apps as LLVM bitcode, and Apple compiles them for the specific devices. A FlutterOS with a Flutter run-time on all devices could do something similar. As with watchOS, app developers wouldn't have to ship the framework itself in the app bundle; it would be on the device already.

Finally, publishing apps in some kind of intermediate representation also has security benefits: the operating system can ensure some invariants via the toolchain. Of course, one would have to ensure that the Flutter API is sufficiently expressive for high-performance applications, while also not having arbitrary machine code execution vulnerabilities; there is a question of language and framework design as well as toolchain and runtime quality of implementation. HarmonyOS could be headed in this direction.

2.5.5 Summary

Ark is a fascinating effort that holds much promise. It's also still in motion; where will it be when it anneals to its own local optimum? It would appear that the system is approaching usability, but we expect a degree of churn in the near-term as Ark designers decide which language abstractions work for them and how to, well, *harmonize* them with the rest of JavaScript.

We see the biggest open question as being whether developers will love Ark in the way they love, say, React. In a market where Huawei is still a dominant vendor, we think the material conditions are there for a good developer experience: people tend to like Flutter and React, and Ark is similar. Huawei "just" needs to explain their framework well (and where it's hard to explain, to go back and change it so that it is explainable).

But in a more heterogeneous market, to succeed Ark would need to make a cross-platform runtime like the one Flutter has and engage in some serious marketing efforts, so that developers don't have to limit themselves to targetting the currently-marginal HarmonyOS. Selling extensions to JavaScript will be much more difficult in a

context where the competition is already established, but perhaps Ark will be able to productively engage with TypeScript maintainers to move the language so it captures some of the benefits of Dart that facilitate ahead-of-time compilation.

3 Future developments

3.1 Possible futures

To recap, we looked at:

- Ionic/Capacitor, which makes mobile app development more like web app development;
- React Native, a flavor of React that renders to platform-native UI components rather than the Web, with ahead-of-time compilation of JavaScript;
- NativeScript, which exposes all platform capabilities directly to JavaScript and lets users layer their preferred framework on top;
- Flutter, which bypasses the platform's native UI components to render directly using the GPU, and uses Dart instead of JavaScript/TypeScript; and
- Ark, which is Flutter-like in its rendering, but programmed via a dialect of TypeScript, with its own multi-tier compilation and distribution pipeline.

Taking a step back, with the exception of Ark which has a special relationship to HarmonyOS and Huawei, these frameworks are all layers on top of what is provided by Android or iOS. Why would you do that? Presumably there are benefits to these interstitial layers; what are they?

Probably the most basic answer is that an app framework layer offers the promise of abstracting over the different platforms. This way, a company can just have one mobile application development team instead of one per platform. In practice one still needs to test on iOS and Android at least, but this is cheaper than having fully separate Android and iOS teams.

Given that we are abstracting over platforms, it is natural also to abandon platform-specific languages like Swift or Kotlin. This is the moment in the strategic planning process that unleashes chaos: there is a fundamental element of randomness and risk when choosing a programming language and its community. Languages exist on a hype and adoption cycle; ideally we should catch one on its way up, and we also want it to remain popular over the life of our platform (10 years or so). This is not an easy thing to do and it's quite possible to bet on the wrong horse. However the communities around popular languages also bring their own risks, in that they have fashions that change over time, and we might have to adapt our platform to the language as fashions come and go, whether or not these fashions actually make better apps.

Choosing JavaScript as the app language places more emphasis on the benefits of

popularity, and is in turn a promise to adapt to ongoing fads. Choosing a more niche language like Dart places more emphasis on predictability of where the language will go, and ability to shape the language's future; Flutter is a big fish in a small pond.

There are other language choices, though; if we are building our own thing, we can choose any direction we like. What if we used Rust? What if we doubled down on WebAssembly, somehow? In some ways we'll never know unless we go down one of these paths; one has to pick a direction and stick to it for long enough to ship something, and endless tergiversations on such basic questions as language are not helpful. But in the early phases of platform design, all is open, and it would be prudent to spend some time thinking about what it might look like in one of these alternate worlds. In that spirit, let us explore these futures to see how they might be.

3.1.1 Rust

The arc of history bends away from C and C++ and towards Rust. Given that a mobile development platform must have some low-level code, there are arguments in favor of writing it in Rust already instead of choosing to migrate in the future.

One advantage of Rust is that programs written in it generally have fewer memory-safety bugs than their C and C++ counterparts, which is important in the context of smart phones that handle untrusted third-party data and programs, i.e., web sites.

Also, Rust makes it easy to write parallel programs. For the same implementation effort, we can expect Rust programs to make more efficient use of the hardware than C++ programs.

And relative to JavaScript et al, Rust also has the advantage of predictable performance: it has quite a good ahead-of-time compiler, and does not do any compilation work at run-time.

These observations are just conversation-starters, though, and when it comes to imagining what a real mobile device would look like with a Rust application development framework, things get more complicated. Firstly, there is the approach to UI: how do we get pixels on the screen and events from the user? The three general solutions are to use a web browser engine, to use platform-native widgets, or to build everything in Rust using low-level graphics primitives.

The first approach is taken by the Tauri[1] framework: an app is broken into two pieces, a Rust server and an HTML/JS/CSS front-end. Running a Tauri app creates a WebView in which to run the front-end, and establishes a bridge between the web client and the Rust server. In many ways the resulting system ends up looking a lot like Ionic/Capacitor, and many of the UI questions are left open to the app developer: what UI framework to use, all of the JavaScript programming, and so on.

Instead of using a platform's WebView library, a Rust app could instead ship a

[1] https://github.com/tauri-apps/tauri/blob/dev/ARCHITECTURE.md

WebView. This would of course make the application binary size larger, but tighter coupling between the app and the WebView may allow us to run the UI logic from Rust itself instead of having a large JS component. Notably this would be an interesting opportunity to adopt the Servo[2] web engine, which is itself written in Rust. Servo is a project that in many ways exists *in potentia*; with more investment it could become a viable alternative to Gecko, Blink, or WebKit, and whoever does the investment would then be in a position of influence in the web platform.

If we look towards the platform-native side, though there are quite a number of Rust libraries that provide wrappers to native widgets[3], practically all of these primarily target the desktop. Only cacao[4] supports iOS widgets, and there is no equivalent binding for Android, so any NativeScript-like solution in Rust would require a significant amount of work.

In contrast, the ecosystem of Rust UI libraries that are implemented on top of OpenGL and other low-level graphics facilities is much more active and interesting. Probably the best recent overview of this landscape is Raph Levien's 2022 post on Xilem[5]; see the "quick tour of existing architectures" subsection. In summary, everything is still in motion and there is no established consensus as to how to approach the problem of UI development, but there are many interesting experiments in progress. With my engineer hat on, exploring these directions looks like fun. As Levien notes, some degree of exploration seems necessary as well: we will only know if a given approach is a good idea if we spend some time with it.

However if instead we consider the situation from the perspective of someone building a mobile application development framework, Rust seems more of a mid/long-term strategy than a concrete short-term option. Sure, build low-level libraries in Rust, to the extent possible, but there is no compelling-in-and-of-itself story yet that we can sell to potential UI developers, because everything is still so undecided.

Finally, let us consider the question of scripting: sometimes one needs to add logic to a program at run-time. It could be because actually most of an app is dynamic and comes from the network; in that case the app is like a little virtual machine. If our app development framework is written in JavaScript, like Ionic/Capacitor, then we have a natural solution: just serve JavaScript. But if the app is written in Rust, what do we do? Waiting until the app store pushes a new version of the app to the user is not an option.

There would appear to be three common solutions to this problem. One is to use JavaScript – that's what Servo does, for example. As a web engine, Servo doesn't have much of a choice, but the point stands. Currently Servo embeds a copy of SpiderMonkey, the JS engine from Firefox, and it does make sense for Servo to take advantage of an industrial, complete JS engine. Of course, SpiderMonkey is written in C++; if there were a JS engine written in Rust, probably Rust programmers would

[2] https://servo.org/
[3] https://www.areweguiyet.com/
[4] https://github.com/ryanmcgrath/cacao
[5] https://raphlinus.github.io/rust/gui/2022/05/07/ui-architecture.html

prefer it. Also it would be fun to write, or rather, fun to *start* writing; reaching the level of ECMA-262 conformance of SpiderMonkey is at least a hundred-million-dollar project. Starting Boa[6] was cheap, and we may wish them the many millions of dollars needed to see it through to completion.

There is no law obliging us to script our app via JavaScript, of course; there are many languages out there that have "extending a low-level core" as one of their core use cases. This approach has had mitigated success over the years—who embeds Python into an iPhone app?—which should probably rule out this strategy as a core part of an application development framework. Still, we should mention one Rust-specific option, Rhai[7]; the pitch is that by being Rust-specific, Rhai is more expressive than any other dynamic language when it comes to interoperation with Rust. Still, it is not a solution to bet on: Rhai internalizes so many Rust concepts (notably around borrowing and lifetimes) that it seems that one has to know Rust to write effective Rhai, and knowing both is quite rare. Anyone who writes Rhai would probably rather be writing Rust, and that's not a good equilibrium.

The third option for scripting Rust is WebAssembly. We'll get to that in a minute.

3.1.2 The web of pixels

Let's return to Flutter for a moment. Like the more active Rust GUI development projects, Flutter is an all-in-one rendering framework based on low-level primitives; all it needs is Vulkan or Metal or (soon) WebGPU, and it handles the rest, layering on opinionated patterns for how to build user interfaces. It didn't arrive to this state in a day, though. To hear Eric Seidel tell the story[8], Flutter began as a kind of "reset" for the Web, a conscious attempt to determine from the pieces that compose the Web rendering stack, which ones enable smooth user interfaces and which ones get in the way. After taking away all of the parts they didn't need, Flutter wasn't left with much: just GPU texture layers, a low-level drawing toolkit, and the necessary bindings to input events. Of course what the application programmer sees is much more high-level, but underneath, these are the platform primitives that Flutter uses.

So, imagine you work at Google. You used to work on the web—maybe on WebKit and then Chrome like Eric, maybe on web standards—but you broke with this past to see what Flutter might become. Flutter works: great job everybody! The set of graphical and input primitives that you use is minimal enough that it is abstract by nature; it doesn't much matter whether you target iOS or Android, because the primitives will be there. But the web is still the web, and it is annoying, aesthetically speaking. Could we Flutter-ize the web? What would that mean?

That's exactly what former HTML specification editor and now Flutter team member

[6] https://boajs.dev/
[7] https://rhai.rs
[8] https://www.youtube.com/watch?v=h7HOt3Jb1Ts

Ian Hixie proposed last January in a brief manifesto, "Towards a modern Web stack"[9]. The basic idea is that the web and thus the browser is, well, a bit much. Hixie proposed to start over, rebuilding the web on top of WebAssembly[10] (for code), WebGPU[11] (for graphics), WebHID[12] (for input), and ARIA[13] (for accessibility). Technically it's a very interesting proposition! After all, people that build complex web apps end up having to fight with the platform to get the results they want; if we can reorient them to focus on these primitives, perhaps web apps can compete better with native apps.

However if we game out what is being proposed, the end result is less clear. The existing web is largely HTML, with JavaScript and CSS as add-ons: a web of structured text. Hixie's flutterized web proposal, on the other hand, is a web of pixels. This has a number of implications. One is that each app has to ship its own text renderer and internationalization tables, which is a bit silly to say the least. And whereas we take it for granted that we can mouse over a web page and select its text, with a web of pixels it is much less obvious how that would happen. Hixie's proposal is that apps expose structure via ARIA[14], but there seems to be no association between pixels and ARIA properties: the pixels themselves really have no built-in structure to speak of.

And of course unlike in the web of structured text, in a web of pixels it would be up each app to actually describe its structure via ARIA: it's not a built-in part of the system. But if we combine this with the rendering story ("here's WebGPU, now draw the rest of the owl"[15]), Hixie's proposal leaves a void for frameworks to fill between what the app developer wants to write (e.g. Flutter/Dart) and the platform (WebGPU/ARIA/etc).

That said, the authors of this report are old enough to remember when X11 apps on Unix desktops[16] changed from having fonts rendered on the server (i.e. by the operating system) to having them rendered on the client (i.e. the app), which was associated with a similar kind of anxiety. There were similar factors at play: slow-moving standards (X11) and not knowing at build-time what the platform would actually provide (which X server would be in use, etc). But instead of using the server, one could just ship pixels, and that's how GNOME[17] got good text rendering, with Pango[18] and FreeType[19] and fontconfig[20], and eventually HarfBuzz[21], the text shaper used in Chromium and Flutter and many other places. Client-side fonts not only enabled more complex text shaping but also eliminated some round-trips for text measurement during UI layout, which is

[9] https://docs.google.com/document/d/1peUSMsvFGvqD5yKh3GprskLC3KVdAlLGOsK6gFoEOD0/edit
[10] https://webassembly.org
[11] https://developer.chrome.com/blog/webgpu-release/
[12] https://developer.mozilla.org/en-US/docs/Web/API/WebHID_API
[13] https://w3c.github.io/aria/
[14] https://w3c.github.io/aria/
[15] https://i.kym-cdn.com/photos/images/newsfeed/000/572/078/d6d.jpg
[16] https://www.x.org/wiki/guide/fonts/
[17] https://www.gnome.org/
[18] https://en.wikipedia.org/wiki/Pango
[19] https://freetype.org/
[20] https://www.freedesktop.org/wiki/Software/fontconfig/
[21] https://github.com/harfbuzz/harfbuzz

a bit of a theme in this article series. So could it be that pixels instead of text does not represent an apocalypse for the web? We cannot say either way with confidence.

Incidentally we should not move on from this point without pointing out another narrative thread, which is that of continued human effort over time. Raph Levien[22], referenced above as a Rust UI toolkit developer, actually spent quite some time doing graphics for GNOME in the early 2000s; we remember working with his `libart_lgpl`. Behdad Esfahbod[23], author of HarfBuzz, built many parts of the free software text rendering stack before moving on to Chrome and many other things. Working on this low level as Levien and Esfahbod do, constantly translating text to textures, it would seem that the accessibility and interaction benefits of using a platform-provided text library start to fade: you are the boss of text around here and you can implement the needed functionality yourself. From this perspective, pixels don't represent risk at all. In the old days of GNOME 2, client-side font rendering didn't lead to bad UI or poor accessibility. To be fair, there were other factors pushing to keep work in a commons, as the actual text rendering libraries still tended to be shipped with the operating system as shared libraries. Would similar factors prevail in a statically-linked web of pixels?

In a way it's a moot question for us, because in this series we are focussing on native app development. So, if we ship a platform, should our app development framework look like the web-of-pixels proposal, or something else? Our survey shows that as a platform, we need more. We need a common development story for how to build user-facing apps: something that looks more like Flutter and less like the primitives that Flutter uses. Though we surely will include a web-of-pixels-like low-level layer, because we need it ourselves, probably we should also ship shared text rendering libraries, to reduce the install size for each individual app.

And of course, having text as part of the system has the side benefit of making it easier to get users to install OS-level security patches: it is well-known in the industry that users will make time for system updates if they get a new goose emoji in exchange.

3.1.3 WebAssembly

WebAssembly is clearly a technology that will be a part of the future of computing. What would it mean for a mobile app development platform to embrace WebAssembly?

Before answering that question, a brief summary of what WebAssembly is. WebAssembly 1.0 is portable bytecode format that is a good compilation target for C, C++, and Rust. These languages have good compiler toolchains that can produce WebAssembly. The nice thing is that a WebAssembly component is completely isolated from its host: it can't harm the host (approximately speaking). All points of interoperation with the host are via copying data into memory owned by the WebAssembly guest; the compiler

[22] https://levien.com/
[23] https://behdad.org/

toolchains abstract over these copies, allowing a Rust-compiled-to-native host to call into a Rust-compiled-to-WebAssembly module using idiomatic Rust code.

So, WebAssembly 1.0 can be used as a way to script a Rust application. The guest script can be interpreted[24], compiled just in time, or compiled ahead of time for peak throughput.

Of course, people that would want to script an application probably want a more dynamic language than Rust. In a way, WebAssembly is in a similar situation as WebGPU in the web-of-pixels proposal: it is a low-level tool that needs higher-level toolchains and patterns to bridge the gap between developers and primitives.

Indeed, the web-of-pixels proposal specifies WebAssembly as the compute primitive. The idea is to ship an application as a WebAssembly module, and give that module WebGPU, WebHID, and ARIA capabilities via imports[25]. Such a WebAssembly module doesn't script an existing application: it *is* the app. So another way for an app development platform to use WebAssembly would be like how the web-of-pixels proposes to do it: as an interchange format and as a low-level abstraction. As in the scripting case, we can interpret or compile the module. Perhaps an infrequently-run app would just be interpreted, to save on disk space, whereas a more heavily-used app would be optimized ahead of time, or something.

We should mention another interesting benefit of WebAssembly as a distribution format, which is that it abstracts over the specific chipset on the user's device; it's the device itself that is responsible for efficiently executing the program, possibly via compilation to specialized machine code. We understand for example that RISC-V people are quite happy about this property because it lowers the barrier to entry for them relative to an ARM monoculture.

WebAssembly does have some limitations, though. One is that if the throughput of data transfer between guest and host is high, performance can be bad due to copying overhead. The nascent memory-control proposal[26] aims to provide an `mmap` capability, but it is still early days. The need to copy would be a limitation for using WebGPU primitives.

More generally, as an abstraction, WebAssembly may not be able to express programs in the most efficient way for a given host platform. For example, its SIMD operations work on 128-bit vectors, whereas host platforms may have much wider vectors. Any current limitation will recede with time, as WebAssembly gains new features, but every year brings new hardware capabilities (tensor operation accelerator, anyone?), so there will be some impedance-matching to do for the foreseeable future.

The more fundamental limitation of the 1.0 version of WebAssembly is that it's only a good compilation target for some languages. This is because some of the fundamental

[24] https://dl.acm.org/doi/abs/10.1145/3563311
[25] https://webassembly.github.io/spec/core/syntax/modules.html#syntax-import
[26] https://github.com/WebAssembly/memory-control/blob/main/proposals/memory-control/Overview.md

parts of WebAssembly that enable isolation between host and guest (structured control flow, opaque stack, no instruction pointer) make it difficult to efficiently implement languages that need garbage collection, such as Java or Go. The coming WebAssembly 2.0 starts to address this need by including low-level managed arrays and records, allowing for reasonable ahead-of-time compilation of languages like Java. Getting a dynamic language like JavaScript to compile to efficient WebAssembly can still be a challenge, though, because many of the just-in-time techniques needed to efficiently implement these languages will still be missing in WebAssembly 2.0.

Before moving on to WebAssembly as part of an app development framework, one other note: currently WebAssembly modules do not compose very well with each other and with the host, requiring extensive toolchain support to enable e.g. the use of any data type that's not a scalar integer or floating-point value. The component model working group[27] is trying to establish some abstractions and associated tooling, but (again!) it is still early days. Anyone wading into this space needs to be prepared to get their hands dirty.

To return to the question at hand, an app development framework can use WebAssembly for scripting, though the problem of how to compose a host application with a guest script requires good tooling. Or, an app development framework that exposes a web-of-pixels primitive layer can support running WebAssembly apps directly, though again, the set of imports remains to be defined. Either of these two patterns can stick with WebAssembly 1.0 or also allow for garbage collection in WebAssembly 2.0, aiming to capture mindshare among a broader community of potential developers, potentially in a wide range of languages.

As a final observation: WebAssembly is ecumenical, in the sense that it favors no specific church of how to write programs. As a platform, though, one might prefer a state religion, to avoid wasting internal and external efforts on redundant or ill-advised development. After all, if it's your platform, presumably you know best.

[27] https://github.com/WebAssembly/component-model

4 Conclusion

4.1 Observations

Having taken a look at the existing cross-platform app development framworks, we are struck with some commonalities.

4.1.1 The age of declarative UI

Firstly, we note that *the declarative UI paradigm has won*. Note, this victory is not limited to Flutter, Ark, and React Native, which explicitly expose a declarative API; over the last 2–5 years it has also become the case for native platforms. Contemporary iOS applications are written in Swift using the declarative SwiftUI framework, and Android apps are written in Kotlin using Jetpack Compose. Even the frameworks that aren't inherently declarative at a low level—Capacitor and NativeScript—even these frameworks expose some middleware allowing users to program apps using popular declarative frameworks.

Therefore, unless a framework author has a strong conviction that a new UI paradigm is coming, any new framework should be declarative by default. It may be useful to be able to access imperative primitives, as in NativeScript and Capacitor, but ideally the framework author should make it unnecessary; bypassing declarative layout indicates that the framework is not doing a good job.

4.1.2 Unstable equilibrium

Despite the commonalities between frameworks, both native and cross-platform, we also can't help but note that the "market" is in an unstable state. Consider:

1. Jetpack Compose uses a compiler plugin to implement its `@Composable` decorator. This is the sign of a stopgap, intermediate solution; if Android really places a bet on Compose, we would expect to see a core language extension as SwiftUI did with its ResultBuilder[1].
2. Ark also uses a modified TypeScript compiler to implement its `@Component` and `@Entry` decorators. Again, if this were a more mature solution, we would expect some kind of more official extension to TypeScript and/or JavaScript, with

[1] https://github.com/apple/swift-evolution/blob/main/proposals/0289-result-builders.md

accompanying documentation, formal semantics, and so on. As it is, the code is the best documentation, indicating that both the code and the semantics are in flux.

3. Both Flutter and React Native are reworking their rendering pipelines, via the Impeller[2] and Fabric[3] projects, respectively. At least these are not language modifications like in the Ark and Android cases, but they do indicate churn.
4. For JavaScript app development, it is impossible to bet on a winning JavaScript framework over the time-frame needed for an app development platform. The most that one can hope for is that in 5 years, the framework you choose will be a survivor, and that new frameworks can build on top of the primitives you provide.

All of these observations lead us to conclude that *there is no current market winner*. As an app developer, this is a problem; as an app framework author, it's an opportunity: the game is still on. There is room for new frameworks, and they stand a good chance at success, or at least at doing as well as the competition.

4.1.3 Language

We also observe that *the choice of programming language strongly shapes the future of an app development framework*.

Languages unite, and they divide. Consider SwiftUI; its programming community is almost completely partitioned off from the non-Apple programming universe. Contrast to NativeScript, which can access all Apple capabilities, but whose use of JavaScript makes the framework accessible to a much broader set of programmers and libraries. Or take the case of Flutter, whose use of Dart ties its users not to a hardware platform but the software framework; the non-Flutter Dart programming world is negligible.

Choosing a language is perilous, and in some ways it would be nice to avoid making choices that reduce the addressable set of potential developers. But this isn't possible: frameworks can't avoid choosing a programming language. Declarative UI means UI-as-a-function, and functions are a programming language concern.

Most of the frameworks we have studied chose JavaScript. As perhaps the most popular and accessible programming language, there is a certain logic to this choice; one not only targets a large set of potential app developers, but frameworks can also take advantage of the wide array of third-party, freely-usable libraries available on NPM and other shared code repositories.

However, we note two risks associated with JavaScript. One is that relative to the web, any JavaScript-based framework will be only a marginal player in the JS language ecosystem. This poses a challenge for co-design between the language, the framework,

[2] https://docs.flutter.dev/perf/impeller
[3] https://reactnative.dev/architecture/fabric-renderer

and apps; the new framework is a small boat, sailing in the wake of a fleet of web supertankers, and adapting JavaScript to the needs of, for example, ahead-of-time compilation will not be easy.

Also, those JavaScript supertankers are following their own stars, and they do change direction; our framework might be effectively forced to react to changes of dominant paradigm in the broader JavaScript world. By choosing a smaller language (Dart), Flutter doesn't have this risk.

4.1.4 Predictable over peak

A common theme of how the different frameworks have evolved in recent years is that when faced with a choice, frameworks choose *predictability over peak performance*.

For example, Flutter's old rendering architecture compiled custom shaders for the widgets used by an app. This could reduce the amount of information that needs to pass from CPU to GPU on each frame, but the need to compile those shaders led to performance nonuniformities at program startup. The new rendering architecture chooses to send more triangles, but paints them using a reduced, fixed set of shaders.

As another example, both React Native and ArkUI chose to mitigate the performance nonuniformities of JavaScript JIT compilation by taking the extraordinary step of bundling their own JavaScript implementation that incorporates ahead-of-time compilation. The fact that these companies have chosen such an expensive proposition is a strong bet on predictability.

It would seem that the advances in peak throughput yielded by JIT compilation in the 2010s are now being reconsidered. Not only are there those JavaScript ahead-of-time compilers, but also Dart and Swift are explicitly designed so as to facilitate good ahead-of-time code generation.

4.1.5 Performance is a framework concern

In any case, it would seem that all frameworks agree that *performance is a framework concern*. App developers just tell the framework what the UI should look like; the frameworks are then responsible for translating those declarative descriptions to good user experiences.

Translation between high and low levels is a form of compilation, and indeed compilation is the general technique available to frameworks. Beyond the jank-reducing benefits of ahead-of-time compilation as mentioned above, compilers pervade all parts of frameworks: from the treatment of the `@Composable` annotation in Jetpack Compose, to the construction of DOM trees from Angular documents in Capacitor, to the type-directed compilation of sound, null-safe Dart, to the tree-shaking and dead code elimination needed to ship JavaScript apps, compilers are key.

For minimal overhead and maximal fitness for purpose, a framework needs to tightly

integrate with its compiler, or indeed compilers, as there are generally a number of them. The corrolary is that in most cases, and where there is budget, a framework does better when it controls its compiler: when it can make extensions to the compiler or even to the language being compiled. We see this in React Native with Hermes, with Flutter and Dart, even with ArkUI with the Ark compiler and run-time.

4.2 Predictions

Given the state of the art, what is an app developer to do? As a consumer of upstream technology, apps developers want good performance, a reasonably complete set of documented libraries, and a simple and integrated toolchain.

Flutter is not a bad choice; it is the state of the art. But if you want to use native text rendering, React Native might not be a bad option, though its opinionated design might not be appropriate for all apps; we would be tempted instead to layer on top of NativeScript, and hope that we can manage the startup time problem.

But if we move from app developers to framework developers, many interesting options open up before us. Considering the state of the art, we expect to see a number of concrete developments in the near future.

4.2.1 Flutter on JS

Flutter's success is due to many factors; among them is the co-design with the Dart language. However, using an effectively proprietary language does have drawbacks, and we expect that a Flutter-like framework based on JavaScript could have more success than Flutter itself.

We expect three developments in the mid-term future. One is a move towards sound typing in TypeScript, together with new ahead-of-time toolchains. Already Ark and React Native are well-positioned to experiment in this direction, and it would seem that React Native is in a gradual process of moving away from their Flow type system towards TypeScript. It will take a broad and deep skill set to make this change, from language design to standards body consensus-building to low-level compiler hacking, but the material conditions are in place.

Another development that we expect is the incorporation of some kind of ResultBuilder-like[4] extension to TypeScript. JavaScript object literals are a good minimum-viable-product solution, but there are benefits to be had with a built-in declarative facility, notably around state dependency management as implemented by Jetpack Compose. Ark's "extended TypeScript" dialect is already a move in this direction, and we expect some kind of standardization effort if Ark is to succeed beyond Huawei's own ecosystem.

[4] https://github.com/apple/swift-evolution/blob/main/proposals/0289-result-builders.md

Finally, we expect React Native to gain a GPU backend, perhaps WebGPU. Rendering to native widgets is needless overhead; it would be more efficient to bypass stateful widget trees.

4.2.2 Flutter on Rust

We also expect that an "outsider" development shop will produce a Flutter on Rust. The marketing pitch of Flutter on Rust is very attractive, and we expect that some organization will produce a close copy of Flutter that uses Rust both for framework development and app development. Rust UI experiments will continue, but a goal-focused Flutter clone will be most successful in the medium term.

Another Rust UI development that we expect is that those Rust UI projects that render to web views will start to use Servo[5] as the browser engine, first as a full-blown web view, and then as a library that doesn't include JavaScript.

4.2.3 Flutter on Wasm

Finally, the attraction of WebAssembly is too strong to avoid. There will be many failures and false starts in this area, but we also expect that in time, there will be a NativeScript-like app development framework that automatically exposes native capabilities to WebAssembly instead of to JavaScript. There will be much experimentation and fragmentation on top, but it is also a new market segment for which we expect some opportunities as well. There is much to be developed in this area, though, especially as regards life-cycles for host-provided objects. Some WebAssembly runtimes will expose these objects as GC-managed data types, and some will instead require toolchains to enforce a reference-counting discipline for handles to external resources. The component model working group[6] will play a role here but we do not expect production-ready systems for some time.

[5] https://servo.org
[6] https://github.com/WebAssembly/component-model

5 Further resources

5.1 EOSS 2023: Cross-Platform Mobile UI

In June 2023, the author of this report presented an early version of this work at the Embedded Open Source Summit, a Linux Foundation event in Prague. The video of that talk is now online[1]; as of August 2023, it is the third-most-viewed recording from that event. PDF slides are also available.[2]

5.2 Publishing history

This document was originally published on July 18, 2023.

It was updated in August 2023 to add this "Further resources" chapter, with mention of the talk at EOSS.

[1] https://www.youtube.com/watch?v=r6ctxZphtkI

[2] https://static.sched.com/hosted_files/eoss2023/d9/2023-eoss-cross-platform-ui-slides.pdf