A-Mash: Providing Single-App Illusion for Multi-App Use through User-centric UI Mashup

Sunjae Lee[1]*, Hoyoung Kim[1], Sijung Kim[1], Sangwook Lee[1], Hyosu Kim[2], Jean Young Song[3]
Steven Y. Ko[4], Sangeun Oh[5], Insik Shin[1,6]*
[1]KAIST, S. Korea [2]Chung-Ang University, S. Korea [3]DGIST, S. Korea
[4]Simon Fraser University, Canada [5]Ajou University, S. Korea, [6]Fluiz Corp., S. Korea
*{sunjae1294,insik.shin}@kaist.ac.kr

ABSTRACT

Mobile apps offer a variety of features that greatly enhance user experience. However, users still often find it difficult to use mobile apps in the way they want. For example, it is not easy to use multiple apps simultaneously on a small screen of a smartphone. In this paper, we present A-Mash, a mobile platform that aims to simplify the way of interacting with multiple apps concurrently to the level of using a single app only. A key feature of A-Mash is that users can *mash up* the UIs of different existing mobile apps on a single screen according to their preferences. To this end, A-Mash *1)* extracts UIs from unmodified existing apps (*dynamic UI extraction*) and *2)* embeds extracted UIs from different apps into a single wrapper app (*cross-process UI embedding*), while *3*) making all these processes hidden from the users (*transparent execution environment*). To the best of our knowledge, A-Mash is the first work to enable UIs of different unmodified legacy apps to seamlessly integrate and synchronize on a single screen, providing an illusion as if they were developed as a single app. A-Mash offers great potential for a number of useful usage scenarios. For instance, a user can mashup UIs of different IoT administration apps to create an *all-in-one* IoT device controller or one can mashup today's headlines from different news and magazine apps to craft one's own news headline collection. In addition, A-Mash can be extended to an AR space, in which users can map UI elements of different mobile apps to physical objects inside their AR scenes. Our evaluation of the A-Mash prototype implemented in Android OS demonstrates that A-Mash successfully supports the mashup of various existing mobile apps with little or no performance bottleneck. We also conducted in-depth user studies to assess the effectiveness of the A-Mash in real-world use cases.

CCS CONCEPTS

• **Human-centered computing → Graphical user interfaces**; **User interface management systems**; **User centered design**.

KEYWORDS

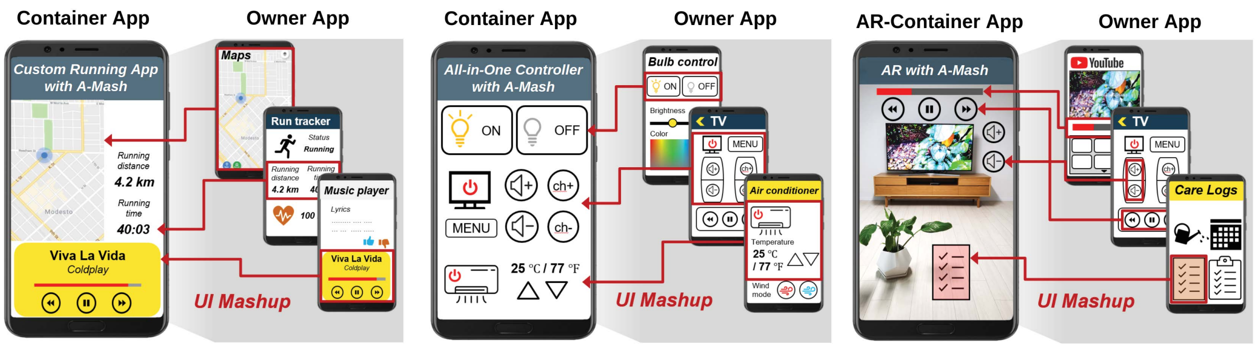
Multi-tasking; User Interface mashup; Multi app execution; Mobile platform; Multi-app single-screen

ACM Reference Format:
Sunjae Lee[1]*, Hoyoung Kim[1], Sijung Kim[1], Sangwook Lee[1], Hyosu Kim[2], Jean Young Song[3], Steven Y. Ko[4], Sangeun Oh[5], Insik Shin[1,6]*. 2022. A-Mash: Providing Single-App Illusion for Multi-App Use through User-centric UI Mashup. In *The 28th Annual International Conference On Mobile Computing And Networking (ACM MobiCom '22), October 17–21, 2022, Sydney, NSW, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3495243.3560522

1 INTRODUCTION

Mobile apps have made our lives significantly easier and convenient. Various apps provide a wide variety of features that greatly simplify and enrich the end-user experience in a variety of areas, including social media, entertainment, and shopping. Furthermore, as the mobile apps become increasingly sophisticated and diversified, smartphone users often want to use their smartphones across app boundaries [21, 37]. A typical example is where users perform multi-tasking. For instance, when running outside for an exercise, a user might want to constantly check her location using Google maps, keep track of her records through Run tracker app, and even listen to music through a music player app. However, since modern-day mobile platforms do not support multiple apps to use the smartphone screen simultaneously, she needs to constantly switch between apps to perform each task independently.

Recently, a few mobile platforms have introduced new techniques that allow multiple apps to run concurrently in a single screen. Split-screen mode [2, 15] allows two apps to run side by side by splitting the screen into two mini windows, Free-form mode [29] allows individual apps to be drawn on a separate movable windows. However, not to mention that it only supports maximum of two apps at a time, it is inconvenient to interact with the apps when they are drawn in such small windows. On the other hand, a couple of studies [8, 38] have considered a similar problem in the web environment and introduced techniques to mash up UI elements of different webpages to create a single multi-purpose web interface. Such an approach can be an effective solution in mobile environments as well since it can not only remove the need to switch between apps when performing tasks that involve multiple apps, but also display only the necessary UIs in the given small size of screen. However, there is a large gap (i.e., code availability) between the web and mobile environments to apply such techniques directly to mobile apps (see Section 10).



(a) Custom multi-tasking app construction.

(b) App functionality integration.

(c) AR/MR environments augmented with UIs of mobile apps.

Figure 1: Use cases of A-Mash.

Motivated by the limitations of current state-of-the-art techniques, we propose A-Mash[1], a mobile platform that provides users with a single-app illusion when using multiple apps simultaneously. That is, A-Mash enables users to use some selected features of multiple apps simultaneously in a single screen as if they were developed as a single app. To this end, A-Mash allows users to extract UI elements of unmodified existing mobile apps and mash them together into a single integrated space, called a *UI container*[2] app. A UI container app serves as a UI playground where users can create their own UI mashups by borrowing UIs from other existing apps (called *owner* apps) and freely arrange, combine, and tailor them according to their own needs and preferences. Users can then load their UI mashups at any time and interact with the UIs through the container app.

A-Mash offers great potential for numerous useful usage scenarios, depending on which set of UIs users decide to mashup (see Figure 1). For instance, as illustrated in Figure 1(a), upon the aforementioned multi-tasking scenario that involves three apps: *1)* Google maps, *2)* run tracker app, and *3)* music player app, a user can extract *i)* map UI from the Google maps, *ii)* timer UI and *iii)* distance UI from the run tracker app, and *iv)* music play bar UI from the music player app to craft a custom app interface where she can check her location, track both the running time and distance, and even control her music, all within a single screen. Likewise, as depicted in Figure 1(b), the user can build an all-in-one IoT device controller app by borrowing UI elements from different administrator app and placing them all together in the single container. One could also imagine an AR/MR scenario, where a user can augment their AR/MR scene with UI elements of other mobile apps. As shown in Figure 1(c), a user can place control buttons of a TV remote control app next to a TV to intuitively interact with the TV in her AR/MR environment. As such, A-Mash can be utilized to enrich one's AR/MR environment using existing mobile apps.

This paper presents the design and implementation of A-Mash that aims to achieve the following goals: i) *transparency*, ii) *high coverage*, iii) *performance* and iv) *user-centric design*. i) For transparency, A-Mash seeks to be fully compatible with existing mobile apps; it does not require the source code of existing apps or any modification to their code. ii) For high coverage, A-Mash allows users to mash up a wide range of app functionalities beyond what existing apps make available through open APIs; since UI is the primary medium for triggering the app's functionalities, A-Mash allows users to mash up all the app functionalities accessible through UIs. iii) For performance, A-Mash eliminates any form of inter-process communication while supporting UI mashup between apps and does not employ any emulation of owner apps' execution within a container app; A-Mash is able to execute the functionalities of UIs, including rendering and event handling, within the process boundary of the UI's original owner app without requiring any direct communication between owner apps and the container app. iv) For user-centric design, A-Mash allows users to freely arrange and tailor borrowed UIs to fit their personal needs, making it possible to craft multi-app experiences in a highly flexible manner.

To achieve the above design goals, A-Mash addresses several technical challenges: 1) *Transparent execution environment.* The key aspect of A-Mash's UI mashup inherently requires the capability to execute apps such that only some selective UIs are visible to the user, while the others are invisible. To this end, A-Mash leverages logical display abstraction to create a new type of display, called *off-screen display*, that is invisible to the users but works as an app's execution environment. 2) *Dynamic UI extraction*: A user may want to extract some UIs that are deep inside the app's activity (i.e., page). To this end, A-Mash reaches to the proper execution point (i.e., activity) of target UIs by developing an advanced form of *UI-driven record & replay* technique [4, 19, 40] and transparently splits the target UIs from existing legacy apps by employing the *transparent UI tree partitioning* technique inspired by [20]. 3) *Cross-process UI embedding*: A-Mash seeks to provide the single-app illusion such that the UIs extracted from different owner apps work seamlessly as if they were part of the container app. To support such an illusion transparently and efficiently, A-Mash proposes *cross-process floating widgets* that allow the extracted UIs to be displayed on top of the container app. The unique widgets can operate based on the container app's life cycle without requiring any forms of inter-process communication.

We demonstrate the concept of A-Mash by implementing a working prototype on Android using Google Pixel 4XL smartphone. Our coverage evaluation using 20 off-the-shelf mobile apps shows that A-Mash is highly compatible with unmodified existing apps and can elicit many novel usage scenarios. Our in-lab performance

[1]Our demo video is available at http:/cps.kaist.ac.kr/amash.
[2]Throughout the rest of the paper, "UI container" app is abbreviated to "container" app

evaluation proves that A-Mash's design inflates little or no performance degradation compared to the traditional app usage model. To further explore the applicability and effectiveness of A-Mash, we conduct a case study of extending A-Mash to an AR environment and carry out user performance evaluation and in-depth usability study. Lastly, we discuss about how A-Mash can co-exist with existing designs of mobile applications in the overall mobile app ecosystems. In specific, we uncover several new security concerns, including spoofing attacks using A-Mash and issues of app governance, and suggest practical solutions to each concern (see Section 11).

2 SYSTEM OVERVIEW

2.1 Background

UI Architecture. A UI is a building block for constructing a visual interaction channel between the user and the app. Android framework provides various types of UIs (e.g., buttons, toggle switches, texts, images, and much more) and uses a tree-shaped data structure, called *UI tree*, to dynamically manage UIs during app execution. One display is assigned only one UI tree, and its contents are visually projected onto the display screen via graphics stacks.

Logical Display. Modern mobile platforms use the concept of *logical display* to extend display functionalities. (e.g., connecting to external displays or pairing with smart TVs). For such display extension, Android offers a display abstraction, called `LogicalDisplay`, to handle various forms of displays, including *PhysicalDisplay, WifiDisplay, and VirtualDisplay*. Each display uses this abstraction in their own ways to interact with underlying Android graphics stacks. One key functionality of the `LogicalDisplay` is providing apps with an isolated execution environment. Apps running on different logical displays can run concurrently in the foreground. Other operating systems also provide their own display abstraction to support various forms of displays (e.g., `UIScreen` in iOS.).

Foreground & Background Processes. When an Android application is being launched, it creates several corresponding Linux processes. Android framework dynamically manages those processes to maintain system resources efficiently, based on the Android activity lifecycle policy. When the app is visible to the users, it is treated as a foreground app, and runs on the foreground process with highest resource priority. When the app is not visible to the users (possibly another app launched on top of it), it is treated as a background app, and gets moved to the background process (i.e., cached process). When the app moves to the background, OS shuts down its graphics stack along with other foreground only tasks (e.g., input handling).

2.2 Workflow

To enable use cases in Figure 1, A-Mash adopts four phase end-user workflow. The first two phases are for creating a UI mashup using MashupRecorder app[3], and the next two phases are for loading and interacting with the UI mashup using the container app.

UI Selection. The first step towards mashing up UIs is specifying which UI to extract from which app. MashupRecorder provides a user-friendly interface for writing the specification by adopting the programming by demonstration.

When the user requests to create a new UI mashup, MashupRecorder shows a list of installed apps and asks the user to launch an app and navigate to the UI that the user wants to extract, just as they would in the standard app usage pattern. Then, the user can activate the UI selection mode by giving a multi-finger gesture on the screen. Upon the activation, a gray translucency layer appears over the app and the user can click any UIs on the screen to specify which UI to extract. Meanwhile, A-Mash continuously monitors UI events triggered during user interactions and records each UI event's type and the resource id of the associated UI that fired the event handlers. These recordings will be later used in the Mashup Loading phase to automate the app navigation (see Section 4). When the user is done with the UI selection, MashupRecorder terminates the app and navigates the user back to the list of installed apps. The user can then repeat this process to extract multiple UIs from different apps or hit "finish" to end the UI selection phase.

UI tailoring. Upon finishing the UI selection phase, all selected UIs appear on the screen and users can tailor the UIs to their needs. Users can resize and reposition each UI to craft a custom interface and save the layouts of the interface for a later use. Such layout information along with the information recorded during the UI selection phase are saved as an XML file. Each XML file represents a specification of a UI mashup and can be manually modified or shared between users.

Mashup Loading. After successfully creating a UI mashup, users can load them at any time using a container app. Upon the user's request, the container app reads the corresponding XML file to identify which UI to borrow from which app. Then, it launches the corresponding apps in the background and automatically extracts, layouts, and displays each selected UI. Note that the whole procedure is done in a transparent manner so that it does not feel intrusive to the users.

UI Interaction. After UIs have been loaded, users can freely interact with the them. Since each UI functions as it did in its original owner app, this allows users to access various functionalities of different apps simultaneously, all within the container app. Once users are done with the interaction, they can either terminate the UIs completely or temporarily hide the UIs for future fast reloading.

2.3 System Design

In order to enable the workflow described in Section 2.2, A-Mash adopts following three techniques (see Figure 2).

Transparent app execution environment. In order to execute the app's foreground tasks without displaying them to the users, we develop a new type of logical display, called *offscreen display*. Offscreen display allows apps to use the foreground process to perform all their foreground tasks (e.g., UI rendering, UI event handling, etc.) even though nothing is actually displayed on the screen. The novelty of our technique lies in giving an illusion to the apps that they are being properly rendered and executed in the foreground when in fact, they are invisible to the users. Apps do not need to be modified in any way nor require a dedicated run-time system.

[3]A-Mash provides users with a MashupRecorder app that allows users can use to create a specification file for a new UI Mashup. The specification file is read by the container app to load the UI mashup upon user's request.

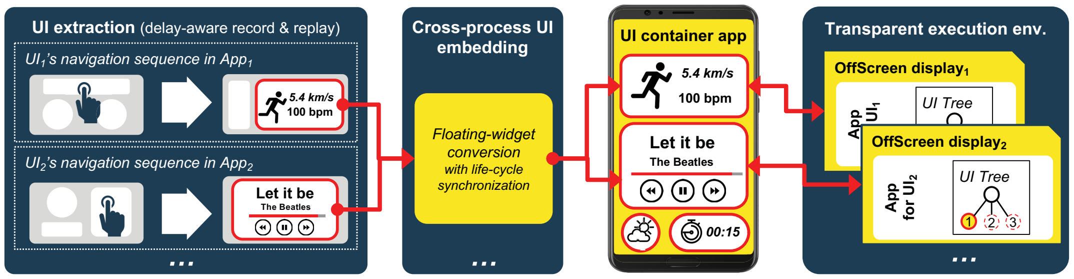

UI extraction (delay-aware record & replay)
UI₁'s navigation sequence in App₁
5.4 km/s
100 bpm
UI₂'s navigation sequence in App₂
Let it be
The Beatles
Cross-process UI embedding
Floating-widget conversion
with life-cycle synchronization
UI container app
5.4 km/s
100 bpm
Let it be
The Beatles
00:15
Transparent execution env.
OffScreen display₁
App UI₁
UI Tree
OffScreen display₂
App for UI₂
UI Tree
1
2
3

Figure 2: A-Mash System Architecture

Dynamic UI extraction. In locating and extracting UI of interest, A-Mash provides a fast and safe UI extraction mechanism. First, for locating the UI, A-Mash uses *delay-aware record&replay* technique. In replaying the sequence of UI events recorded during the UI selection phase (see Section 2.2), A-Mash continuously monitors the UI tree and checks if the UI associated with the event has been loaded. Then, only after the associated UI has been detected, A-Mash replays the UI event. Our technique works for both statically loaded UIs and dynamically loaded UIs and guarantees minimum delay between separate UI events. Next, for extracting the UI, inspired by a recent work [20] which introduces a technique to extract UI element transparently from an existing app, A-Mash adopts its *Single UI Tree Illusion* to safely extract the UI without compromising the app's behavior.

Cross-process UI embedding. To enable a UI embedding across app process (i.e., from owner apps to container app), we cleverly adopt the concept of floating widget. Specifically, we develop a technique to convert legacy UIs into floating widgets– a type of UI that can float on top of other apps. This enables any type of UI to be rendered and operated on top of the container app. Then, to give the single-app illusion to the users, we synchronize the life-cycle of floating widgets with that of the container app. For instance, all floating widgets automatically disappear when the container app goes to the background, and comes alive when the container app resumes to the foreground.

3 TRANSPARENT EXECUTION ENVIRONMENT

A-Mash enables a mash-up of different apps by selectively displaying parts of their UIs in a container app. One of the key techniques that make this possible is our *off-screen display* that appears as a regular display to existing apps but does not draw anything on an actual screen. We describe this technique in this section.

3.1 Limitations of the State-of-the-Art

Previous systems have developed techniques to execute an app *in the background* without showing any visual output on a screen. For example, UIWear [35] and ULPM [36] directly modify Android to enable such functionality, while X-Droid [19] accomplishes the same goal by introducing an application-layer emulator that can execute an app without using a display. Although details differ, these systems use a similar high-level approach where they completely cut off an app from the underlying graphics stack.

However, such an approach would not work for A-Mash since the goal is to display an app's UI, albeit partially. Therefore, A-Mash takes a different approach that does not cut off an app from the graphics stack completely but provides an invisible, logical display called an *off-screen display* to the app. Combined with our UI tree partitioning technique described in Section 4.2, our off-screen display allows us to selectively display parts of an app's UI.

3.2 Off-Screen Display

Our off-screen display is an invisible display given to each app used as part of a mash-up. Since it appears as a regular display, there is no difference from a physical display from an app's point of view. Furthermore, since modern mobile platforms (e.g., Android, iOS) allow multiple apps to run concurrently in the foreground if they each have their own display [2, 14], we do not need separate scheduling techniques to keep all apps active. In order to create such an off-screen display for an app, we first create an empty window with a size equal to that of the physical display. The reason for this equal size is to ensure that responsive apps (i.e., apps that behave and render differently based on the resolution of the display) to behave exactly as they would do in the physical display. Then, to make it invisible, we set the window's transparency to the maximum value and make it not touchable. This ensures that users can neither see the window nor unintentionally inject inputs. Finally, we register the window as a new logical display and attach the app. Since this display is drawn on an invisible window, the app gets completely hidden from the screen.

One caveat is that since we allow multiple apps to be active in the foreground simultaneously, they all use system resources such as battery. However, from our usability study in Section 9.3, we observe that most of the usage scenarios involve two or three apps, which is an acceptable number of apps. We note that existing multi-app techniques (e.g., multi-window, PiP (Picture-in-Picture), etc.) keep a similar number of apps active in the foreground as well.

Furthermore, since A-Mash can simplify the steps for tasks that involve multiple apps, it can reduce the overall resource usage effectively. In fact, from our user performance evaluation in Section 9.2, we have observed that users can reduce their task execution time up to 55% when performing tasks that involve two apps. Based on our simulation of such user behavior, this could lead to up to 52% less battery consumption. Our evaluation in Section 8 provides more details.

4 DYNAMIC UI EXTRACTION

Though our off-screen display allows us to execute an app in an invisible fashion, our goal is to partially display an app's UI as part of a mash-up. Thus, we need a way to locate and extract the UI elements that we do want to display from an app. A-Mash develops two techniques to enable it: delay-aware record & replay and transparent UI Tree partitioning.

4.1 Delay-Aware Record & Replay

Before extracting the UI from an app, we must first ensure that the UI elements we want to extract have been properly instantiated. Since UI objects are instantiated only within the context of its hosting app window (called an *Activity* on Android), we must navigate the app to the proper Activity prior to extracting the UI elements. To automate such a process, A-Mash adopts record & replay, a technique used widely across various fields to reproduce an execution of a program. A-Mash's record & replay has the following characteristics: *i)* user-programmable, *ii)* fast, and *iii)* generic.

MashupRecorder app. To record how an app is navigated to a target Activity, MashupRecorder app continuously monitors UI events triggered with user interactions during the UI selection phase of our workflow. Upon each UI event, MashupRecorder logs the UI event's type (e.g., touch, key) and the resource ID of the corresponding UI element in an XML file.

Delay-aware replay. When an owner app gets launched on an off-screen display, A-Mash automatically navigates the app to the target UIs by replaying the sequence of UI events written in the XML file. In the process, we need to find an optimal time gap between any two consecutive UI events to achieve a minimal delay and provide high accuracy. If the time gap is too short, the UI will not be ready to trigger the associated event handler, and if the time gap is too long, it will cause an unnecessary delay. To address this problem, A-Mash continuously monitors the UI tree and checks if the associated UI has been loaded. Then, upon discovering the UI, A-Mash replays the UI event. This allows A-Mash to replay UI events with a minimal time gap and still provide the correctness of the replay.

Generalizing search queries. Often times, users may want to extract UIs that depend on search results. For example, a user may extract a navigation UI that appears only after searching for a specific place. In such cases, it is possible for A-Mash to automatically extract the identical UIs by replaying the search queries recorded in advance. However, some users may also want to generalize these search queries so that the extracted UIs can show different contents (e.g., information about different places) according to their contextual situation. To satisfy these two conflicting needs, A-Mash's container app gives users a choice by providing a text input UI where users can type different queries for using UIs with different contents. Users can easily enable or disable the text input UI according to their needs.

4.2 Transparent UI Tree Partitioning

Using our record & replay, we can automatically navigate any app to its target Activity. The next step is to extract the target UI elements from the app and display them in the container app. However, since an app's UI tree is tightly coupled with the app's internal state and control flow, arbitrarily partitioning a UI tree can cause unexpected side effects. Therefore, to safely extract UIs from its UI tree, we adopt the Transparent UI Tree partitioning technique from a work, FLUID-XP [20]. Its key idea is to reconstruct the UI tree to have two types of edges: *logical edges* and*rendering edges*. The former is used by the app logic to execute any UI-related functionalities (e.g., updating UI states, UI event handling), and the latter is used by the rendering process to draw the UI tree. Then, to split the target UIs from the UI tree, we remove only the rendering edges in between the UIs and the UI tree. All logical edges are maintained as it was. This allows app logic to function as if the UI tree has never been partitioned, whereas from the rendering process' perspective, target UIs are properly extracted from its UI tree. The dangling rendering edge of the target UI can later be attached to any other UI tree to be rendered separately from its original UI tree.

5 CROSS-PROCESS UI EMBEDDING

Once UIs have been extracted, they are ready to be embedded into the container app to create a mash-up that combines their respective features. A-Mash aims to provide a single-app illusion—an illusion that gives users a look and feel that they are interacting with a single unified app. This section explores various approaches available for embedding UIs and introduces how A-Mash efficiently implements the single-app illusion.

5.1 Possible Approaches

A body of works closely related to UI mash-up is cross-device UI distribution. There are largely two approaches: pixel-level distribution [20] and object-level distribution [25, 35, 39]. The former approach can be adopted in a way that extracted UIs are rendered by its owner app and transferred to the container app in the form of pixels so that UIs can be re-rendered inside the container app. However, this approach requires time-consuming inter-process communication between two processes (for pixel streaming). The latter approach can be used to serialize the UI object along with its graphics resources to reconstruct and render the UIs locally inside the container app. Although this provides responsive UI interaction, their design inherently suffers from the lack of UI coverage since they only support UIs that are under the control of Android's Java-based UI stack. UIs directly rendered by native graphics libraries, such as Skia [30], are not supported in this case.

5.2 Cross-Process Floating Widget

To overcome the limitations of previous approaches, A-Mash leverages a special type of UI containers called *floating widgets* that can overlay the UI of an app above all other UIs at all times even when the app goes in the background. Well-known examples of floating widgets include PiP (Picture-in-Picture) and system alerts. Floating widgets are responsive because their processing, including rendering and event handling, takes place locally. In addition, floating widgets support all UI types as they are built-in, platform-provided containers. Thus, we convert extracted UIs into floating widgets and display them inside a container app. We explain this process further as follows.

Converting extracted UIs into floating widgets. As the first step for displaying extracted UIs, we convert them into floating widgets. Conventionally, converting a UI from an app into a floating widget would require intensive modification to the app's source code. Specifically, there are three things that would need to happen. First, the UI needs to be removed from the app's original UI tree and attached to the UI tree of the floating widget. Second, the app would require a new background service that keeps the UI alive while the app is in the background. Third, since the UI is no longer accessible from the app's UI tree, all of the UI's relevant functionalities (e.g., event handling, UI updates) should be modified and migrated to that background service.

However, our extracted UIs do not need any such modification due to the following two reasons. First, our extracted UIs belong to their owner apps that never go in the background. This is because each of the owner apps has an off-screen display and Android never puts an app that uses a display in the background. Thus, there is no additional background service necessary. Second, our extracted UIs are removed from their original UI trees, but only from the rendering point of view since only their rendering edges are removed, not their logical edges (as mentioned in Section 4.2). This has two implications. First, from the app logic's point of view, extracted UIs are still accessible via logical edges. Therefore, there is no need to modify or migrate any of the relevant functionalities for extracted UIs to function properly. The second implication is that we can create new rendering edges to attach an extracted UI to a floating widget's UI tree. This means that when the UI tree of the floating widget is rendered and displayed, the extracted UI will also be rendered and displayed. Combining everything together, we convert an extracted UI into a floating widget by first creating an empty floating widget and then attaching the extracted UI to the floating widget's UI tree with new rendering edges.

Embedding floating widgets into a container app. In order to give a single-app illusion to the users, floating widgets' life-cycle should be seamlessly synchronized with that of their container app. For instance, when the container app goes to the background, all floating widgets should automatically disappear from the screen. However, since a floating widget operates only within the boundary of its owner app, the container app cannot enforce such actions.

This requires us to implement a capability to control floating widgets from a container app, which we accomplish by designing a new UI type called *UI display*. As the name suggests, it works as both a UI element and a (logical) display. Since it works as a UI element, a container app can embed it and control it just as any other UI elements. In addition, since it works as a display, it is globally accessible via a system service of Android called the *Display Manager*. Furthermore, a property of a floating widget is that it can be used as a *surface* that is essentially an output data buffer for a display. Using this property, we create a UI display for each floating widget that uses the floating widget as the surface so that the UI display can draw the UIs inside the floating widget. The end result is that a container app can access floating widgets that contain extracted UIs through the *Display Manager* and then use them as UI elements.

To synchronize the life-cycle of UI displays and a container app, we implement synchronization logic in the container app's activity life-cycle callbacks. Upon onStop(), all UI displays go to a hidden state; upon onResume(), all UI displays become visible; and upon onDestroy(), all owner apps get terminated and associated resources, including UI displays and off-screen displays get freed.

6 IMPLEMENTATION

MashupRecorder and Mashup XML file. A-Mash saves the specification of each UI mashup in an XML format. Due to space limit, we cannot discuss in detail about the XML vocabularies; but in summary, it includes *i)* which app to launch, *ii)* how to navigate each app to the target UI, *iii)* which UIs to extract, and iv) the position and size of each UI on the container app.

To allow MashupRecorder app to automatically generate the XML file, we modified the Android's `View` class to continuously monitor the UI events (e.g., `onClick()`, `onKeyEvent()`, `onLongPress()`) triggered during the user interaction. It records each UI event's type and the resource ID of the associated UI that triggered the event handlers. Note that the apps launched through any other means are not affected by the above modification. Then, when the user finishes the recording and UI re-tailoring processes, MashupRecorder saves the recorded UI events and each UI's final position and size as an XML format inside the device's public storage so that container apps can have access to these files. In case a UI event cannot be expressed with our input record syntax (e.g., UI does not have resource id), A-Mash falls back to the traditional x-y coordinate based input record & replay.

Backward-compatibility. From our user study, we have observed that users sometimes wish to bring an owner app to the foreground and use it in full screen mode. Therefore, to provide backward-compatibility with the traditional app execution model, we modified Android's `Task Manager` to allow apps in the offscreen displays to be migrated to the physical display's task stack. While interacting with the UI mashup, users can bring the UI's owner app to the physical display by long pressing the UI.

UI transition. To allow users to extract UIs from different activities of a single app, A-Mash provides transition between UIs. Similar to the conventional app activity transition, A-Mash enables UIs from the same app to replace each other when the app changes its foreground activity. This is possible because we perform UI extraction at the activity level. Whenever a new activity starts, A-Mash checks the XML file to see if there are any UI events to replay or UIs to extract from the activity. Likewise, when the app changes its screen orientation and reloads all of its UIs according to the new screen layouts, A-Mash repeats the UI extraction on the newly loaded UIs.

7 CASE STUDY: AR EXTENSION

To explore the range of A-Mash's applicability, we have extended its coverage to AR environment. With the same system design and implementation as before, we created a new container app called *AR-container app*. AR-container app is a camera-based AR application that uses TensorFlow's MobileNet SSD Model for object recognition. It allows users to map an UI mashup to a physical object. When the user points the camera to the object and clicks on it, the AR-container app loads the associated UI mashup onto the AR scene. Users can fill their AR environment with their own rules to create a personalized AR space. For instance, as shown in Figure 1(c), a user can attach TV controllers to the TV, and at the same time, attach some personal memos to the flowerpot to remind themselves of something when watering the flower. In doing so, users do not need to understand their underlying source code to import legacy mobile apps into the AR environment.

Interaction Type	App Name	Corresponding UIs	Use-Case Scenarios	# of Downloads	LoC
Multimedia	YouTube	Video Progress Bar	Controlling the most recent video play in user's subscription list.	10B+	11
	Spotify	New Songs List and Music Player	Controlling the current-played music and new songs of user's favorite music artists.	1B+	11
	Melon	Music Player	Controlling the current-played music.	50M+	9
News	CNN	Headline News Title	Viewing the most recent headline news title.	50M+	11
	Wikipedia	Today's Featured Article	Viewing title and content of today's featured article.	50M+	5
	Yahoo Finance	Stock Chart	Viewing the stock chart.	10M+	11
Physical Interaction	Phone by Google	Number Digits and Call Button	Viewing the emergency call number (e.g., 911) with instant call button.	1B+	11
	Smart Remote for LG TVs	Remote Control Buttons	Controlling the smart TV with the buttons from the remote control app.	1M+	14
	TripAdvisor	List of Nearby Attractions	Viewing the list of nearby tourist attractions based on GPS location.	100M+	8
	Starbucks	List of Nearby Cafes	Viewing the list of nearby cafes based on GPS location.	5M+	9
	Adidas Run Tracker	Distance and Time for Running	Starting the running exercise and viewing its distance and time measurements.	50M+	11
	Ediya Members	List of Nearby Cafes	Viewing the list of nearby cafes based on GPS location.	1M+	11
Information Search	Coupang	Recommend Item for Search Keyword	Viewing the information of recommend item based on search keyword.	10M+	22
	lotte.com	Recommend Item for Search Keyword	Viewing the information of recommend item based on search keyword.	5M+	22
	Vivino	Search Result (Wine Label Image and Score)	Viewing the information of wine based on search keyword.	10M+	14
	Wine Scanner & Expert Reviews	Search Result (Price of Wine)	Viewing the information of wine based on search keyword.	10K+	15
	SoundHound	Search Result (Artist, Title and Album Image)	Searching for nearby music via microphone and viewing the result of the search.	100M+	14
Productivity	Perfect Piano	Piano Keyboards	Playing the piano with the piano keyboards from the app.	100M+	8
	ColorNote	Note Content	Viewing the content of notes.	100M+	8
	Paint by Ng-Labs	Drawing Canvas	Drawing a paint on the canvas from the app.	1M+	5

Table 1: A list of applications, their respective UIs, and possible use-case scenarios for coverage test.

There are several implementation details that enable the AR-container app. Most importantly, we need to carefully synchronize the UIs with the AR-container app. To synchronize the UIs with their associated physical objects, we re-position and resize the UIs based on the locations of the objects. This gives an impression that the UIs are physically *glued* to the objects. In addition, when the object gets out of the AR scene, we hide its associated UIs from the screen, and when the object re-enters the scene, we bring the UIs back. Note that this is very similar to what we do when synchronizing UI mashups with the life-cycle of the conventional container app.

8 EVALUATION

We have implemented a A-Mash prototype to demonstrate and evaluate its full functionality to mash up UIs of unmodified apps. The A-Mash platform prototype is implemented using Android Open Source Project (AOSP v.10). Across the evaluation, we use Google Pixel 4 XL and two in-house UI container apps to evaluate the prototype –a conventional container app for the performance evaluation and an AR-container app for the user study.

8.1 Coverage

To evaluate how well A-Mash supports unmodified apps, we downloaded and explored 20 off-the-shelf mobile apps from the Google Play store. As shown in Table 1, we successfully crafted various use-case scenarios through UI mashup. A countless number of possible combinations can be made using various UIs. For instance, 1) ColorNote's noteUI and 2) Wikipedia's Today's featured Article UI can be mashed up to read the article while taking notes; 1) Adidas Run Tracker's distance and time UI, 2) Spotify's Song list app, and 3) TripAdvisor's list of nearby Attractions UI can be mashed up to create a custom running app that shows a list of nearby attraction to run for, and lets you control your music app at the same time.

The LoC column shows lines of code generated in the associated XML file when extracting the UI. Although it is generated automatically by the MashupRecorder app, anyone who can write XML code, including app developers, UI designers, or tech-savvy users, can easily create or modify the XML file themselves.

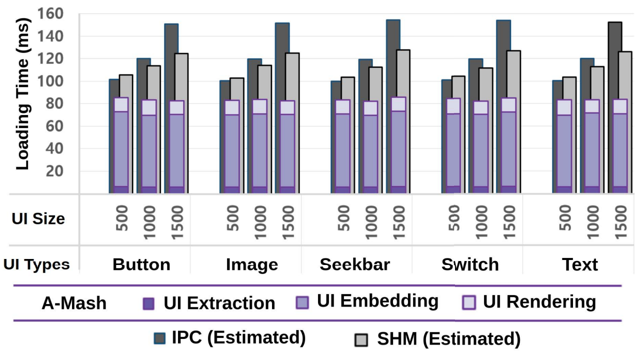


Figure 3: UI loading delay

8.2 Performance

We evaluate the performance of A-Mash for its interactive UI mashup experience. We repeat each experiment a hundred times and use a conventional container app as a testbed for A-Mash. We measure the delay by triggering a system-level log event at each stage of measurements.

Since, to our knowledge, there are no counterparts of A-Mash, we simulate the behavior of other possible approaches introduced in Section 5 for comparison. For the simulation, we created an Android background service app that either uses binder IPC or shared memory (SHM) to transmit UI's raw pixel values to its corresponding container app. Note that the simulated results are very conservative estimation (disadvantageous to A-Mash) since it does not include delays for fetching the UI's pixel data from the graphics buffer.

8.2.1 UI Loading Delay. Figure 3 plots UI loading delay, i.e., the time it takes for the container app to load UI from another app. We break it down into 1) *UI extraction*, 2) *UI embedding* (i.e., embedding UIs to the container app), and 3) initial *UI rendering* times[4]. The delays were measured for five different UI types under three different sizes. The "UI size" row in the graph represents the width and height of the UI; 500 indicates that the UI has 500x500 resolution.

[4]Note that the time for launching and navigating the app is excluded because these delays are very dependent on the owner app's behavior. These delays will be covered in more detail in the user performance evaluation (see Section 9.2).

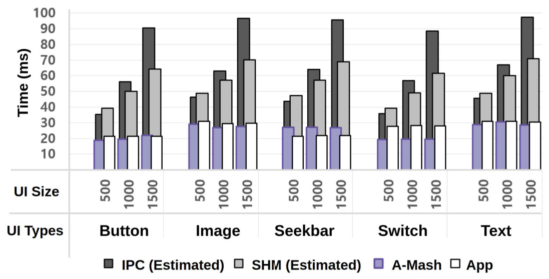


Figure 4: UI interaction delay

The graph shows that A-Mash yields lower UI loading delays compared to the IPC and SHM approaches in all cases. Furthermore, A-Mash causes constant delays regardless of the type and size of UI, while IPC and SHM increases delays depending on the size of UI. Such a performance gap mainly comes from our cross-process UI embedding technique. A-Mash renders UIs locally within their owner apps' processes and displays them directly on the screen. This way, A-Mash imposes no additional overhead other than extracting UIs and converting them to a UI display. On the other hand, the IPC and SHM approaches perform almost the same steps as A-Mash does (i.e., extracting UI, rendering them to a separate virtual displays for pixel extraction) plus, they take additional steps to 1) transmit the pixel data over to the container app and 2) re-render them on the container app, which incurs substantial overheads. The overhead of transmitting the pixel data can be mitigated if they use pixel encoders. However, considering that the encoding process itself takes 40~60ms, it is not an appropriate solution.

8.2.2 UI Interaction Delay. Figure 4 plots the UI interaction delay, i.e., the time taken to handle users' input and reflect the result of input handling to the screen. Here, we compare the delays of A-Mash with the traditional app execution model.

The figure shows interesting results. In many cases, A-Mash shows even lower delays than the traditional app execution model. We attribute these results to the rendering procedure of Android. Specifically, since extracted UIs are rendered separately from all other UIs in its own isolated display (i.e., UI display), it does not go through the composition phase of the rendering process. In other words, while the traditional app execution model needs to composite the pixels of the target UIs with all other UIs in the same display, A-Mash requires no such process since each extracted UIs are rendered independently on its own display.

On the contrary, IPC and SHM based approaches show very high delays. This is because once UIs have been loaded, UIs under A-Mash work the same way as they did in its owner app, but for IPC and SHM, they must go through the pixel transmission and duplicate rendering again every time the user gives an input.

8.3 Resource Consumption

This section evaluates the resource consumption of A-Mash in terms of memory and battery under various conditions.

8.3.1 Memory Consumption. One of the key features of A-Mash is the use of logical displays, and we measured how much memory A-Mash consumes for each additional UI (logical) display instance. We used Android Debug Bridge's (adb) `dumpsys meminfo` command to periodically log the memory usage of each app. During experiments,

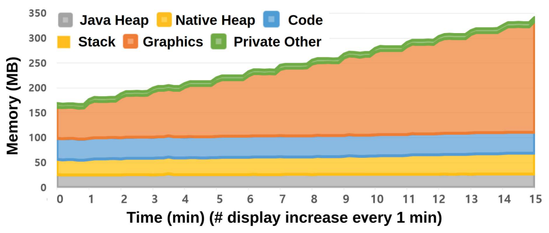


Figure 5: Memory consumption per display

we increased the number of UI displays every 1 minute using our container app for 15 minutes, while each UI display has a button UI of 1500 x 1500 resolution.

Figure 5 shows the memory usage of Android system and the container app over time. Note that it excluded the memory consumption of all other processes since they are irrelevant to our evaluation. The graph clearly demonstrates that as the number of UI display increases, the graphics' memory consumption increases as well. On average, each new UI display consumes 9MB of extra memory. This exactly matches the amount of memory required to support a frame of 1500 x 1500 display in ARGB8888 format (4 bytes per pixel). Likewise, each off-screen display with 1440 x 3040 resolution (Pixel 4 XL) requires 17.5 MB memory, which translates to 0.28% of the memory size (6GB) of Pixel 4 XL.

8.3.2 Battery Consumption. To estimate the effect of A-Mash on the battery consumption more clearly, we used Android's battery usage profiler tool (Batterystats [17]) to measure the battery consumption of individual apps.

Battery consumption of UI display– First, to measure the battery consumption of each new UI (logical) display, under the same experimental environment as in the memory consumption evaluation, we measured the battery consumption of both the container app and the Android system. We conducted experiments with UI displays under two conditions: active and inactive. Active displays render new frame every 1 second and inactive displays do not render after initial renderings, both are in visible state. The result showed that the battery consumption increases linearly (consumes additional 0.01 mAh for each new display) as the number of active display increases from 0 to 15. On the other hand, the battery consumption of inactive displays remains constant (0.1 mAh per minute) regardless of the number of displays. This implies that simply creating an inactive UI display does not significantly affect battery consumption. Rather, how actively each UI display is being used determines the actual battery consumption.

Battery consumption of multiple foreground apps– Figure 6 shows the battery consumption as the number of apps in the foreground process increases from one to eleven (one using the physical display and ten using offscreen displays). In this experiment, we launched a new app every 10 minute. Note that we measure the battery consumption of the whole device, including the screen. As shown in the graph, there is no clear correlation between the number of apps and the battery consumption. Based on our observation during the experiment, the battery consumption is rather determined by how actively each app is working. For instance, YouTube app alone consumes 20 mAh every 10 min since it automatically plays a video from the recent watched list. All other apps showed no significant battery consumption except at its start up. We have also

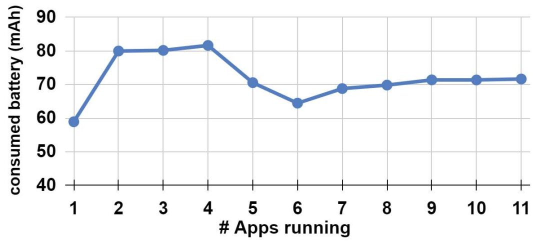


Figure 6: Battery consumption of foreground apps

measured how much battery users would actually consume in the real-world situations by simulating the user performance results from Section 9.2. In the interest of brevity, we omit the figures; but in summary, we found that for task scenarios that involves multiple tasks (Scenario 3 & 4), users reduce their task completion time by 52% and 55%, and as a result, save battery consumption 39% and 52%, respectively.

9 USABILITY STUDY

To understand the effectiveness and usefulness of A-Mash from the user's point of view, we conducted three separate in-lab user studies with five use-scenarios described in Section 7. The first study aims to evaluate how effectively A-Mash simplifies the use of multiple apps simultaneously by comparing the time spent on tasks with or without A-Mash. The second study focuses on assessing the usefulness and satisfaction of using A-Mash across the scenarios compared to using existing apps. The third study compares the subjective usability of A-Mash and Android's Split-Screen mode [15] which is the state-of-the-art multi-app interaction technique.

Participants were recruited through advertisement on an online school community and an online local marketplace, and were paid approximately 9 USD. The study procedures were in accordance with out institution's IRB policies.

9.1 Study Scenarios

We carefully chose five distinctive task scenarios to evaluate various interaction with A-Mash.

Scenario 1: TV Control. Participants were asked to cast a YouTube video on a smart TV and adjust the volume with a TV remote app. When using A-Mash, volume control UI automatically popped up as the user clicked on a TV through the AR-container app.

Scenario 2: Music Playing. Participants were asked to play all the songs on the top chart in a music streaming app. When using A-Mash, the player UI automatically popped up and played the songs as the user clicked on a cup through the AR-container app.

Scenario 3: Banana Shopping. Participants were asked to compare banana prices from two different shopping apps. When using A-Mash, the prices of bananas in different shopping apps popped up as the user clicked on a banana through the AR-container app.

Scenario 4: Wine Searching. Participants were asked to find wine ratings from a wine rating app and its prices from a shopping app. When using A-Mash, the rates and prices automatically popped up as the user clicked on a wine bottle through the AR-container app.

Scenario 5: Coffee Ordering. Participants were asked to find the closest coffee shop from two different coffee ordering apps to place an order. When using A-Mash, it showed the list of nearby coffee shops from both apps when the user clicked on a cup through the AR-container app.

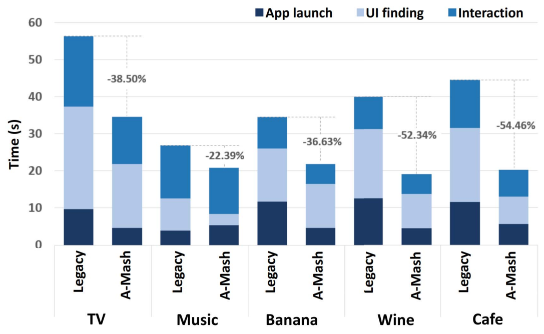


Figure 7: Task completion time

9.2 Study 1: Total Task Time Reduction

Participants and apparatus. We recruited twenty-three participants through online school communities (15 males, 8 females, mean age=25.4, stdev=6.2, max=53, min=19). Each participant were given two Pixel 4XL Android smartphone, one for using AR-extended A-Mash and another for using legacy apps.

Design. The experiment was conducted using within-subject user study. The independent variable was the system (i.e., A-Mash or legacy apps) that participants used for performing the given tasks. The dependent variable was the task completion time for each task.

Procedure. Each session took 30-60 minutes long, and at the start of each session, participants were given a demo video to understand how to use A-Mash and an existing legacy app for the given task. Then, to prevent the learning effects when using A-Mash, we instructed them to perform the task first using A-Mash, and then do the same task using legacy apps. This step was repeatedly performed for five different scenarios in Section 9.1. We screen recorded the devices (Pixel 4XL) and calculated the task completion time afterwards.

Results and findings. Figure 7 compares the average total task time the participants spent on each task when using A-Mash and legacy apps. The figure shows that A-Mash significantly reduced the average total task time for all five scenarios. Paired t-test demonstrates a significant difference between the two different conditions (Music Playing: $p<0.01$, Other scenarios: $p<0.0001$).

To better understand the results, we divided the task completion time into three different stages: 1) *App launch*: the time to find the right app to perform a given task; when using A-Mash, it is the time taken to find its trigger object on the AR-container app. 2) *UI finding*: the time to find a UI for the given task. 3) *Interaction*: the time to conduct the given task using the UI. A-Mash reduced time in all three stages except the app launch stage in Scenario 2: Music Playing. We believe that this happened mainly due to two factors. First, most of the participants used the music streaming app every day, so they could quickly find it from the list of apps. Second, many participants thought it was counter-intuitive to use cups to run music streaming apps and rather expected to use speakers to do it.

Another interesting observation is that A-Mash has not only reduced the average task completion time, but also lowered the deviation between participants (standard deviation reduced by 53% on

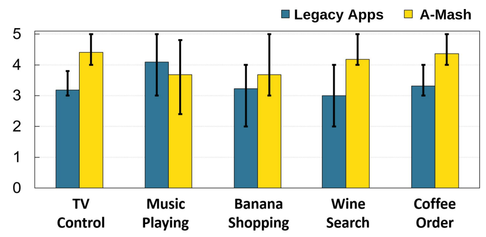


Figure 8: Satisfaction scores for individual scenarios

average). We believe that this indicates that A-Mash could be more effective for those who took a relatively longer time to complete tasks, possibly due to lower digital literacy or unfamiliarity with the given apps, when completing the task with legacy apps. We infer from this result that A-Mash can help people with low digital literacy to more easily and conveniently conduct a task using digital devices.

9.3 Subjective Usability of A-Mash

Participants and apparatus. We recruited twenty-two new participants from online school communities and local online marketplace (12 females, 10 males, mean age=30.5, stdev=9.78, max=58, min=20). Each participant were given two Pixel 4XL Android smartphone, one for using AR-extended A-Mash and another for using legacy apps.

Design. The experiment was conducted using within-subject user study. The independent variable was the system (i.e., A-Mash or legacy apps) that participants used for performing the given tasks, and the dependent variable was the satisfaction score of the system that they evaluated for each task scenario.

Study procedure. The study procedure was similar with the first user study except that the participants were asked to do the UI mapping themselves. Participants were asked to select a mobile app UI and map it on a nearby object, which helped them to understand the concept of customizing the mapping between the object and a mobile app function. After performing each scenario using A-Mash and an existing legacy app, participants were asked to rate the satisfaction of each method. They were also asked to freely describe how they felt about using A-Mash.

Results and findings. Overall, participants thought using A-Mash was satisfying. Figure 8 compares the satisfaction of using a legacy app and A-Mash for each scenario. The error bar indicates the 20-80% distribution of participants' rating. This is mainly because most participants found that A-Mash is intuitive and easy to use and enables to accomplish tasks quickly and efficiently in all scenarios except one (Scenario 2: Music Playing). Below, we share some of the quotes from the participants to help understand the experience of using A-Mash.

Scenario 1: TV Control. "It is easy to use since I don't have to navigate the app to find the right UI (to execute a desired task) (P12)". Participants appreciated that it was effortless to learn how to use A-Mash and how UIs of two different applications were brought into one frame for convenience (YouTube app and TV remote app).

Scenario 2: Music Playing. "It will be more convenient if I can map an app to a most-frequently used object (P4)". Many participants felt that mapping a music player to a cup is rather unexpected and uncomfortable to use; there were similar user comments from the first user study as well. On the other hand, a participant mentioned that *"I can create my own apps by putting my own functions and emotions together into the objects I want (P7)".* For those participants who understood that they can map the music player to any other objects, they appreciated that they could create a customized, personal interaction channel through the surrounding objects with A-Mash.

Scenario 3: Banana Shopping. "This will make the process of repeat purchase very convenient (P8)". Similarly, *"It was useful to be able to check the prices with the camera without having to search different apps (P11)".*

Scenario 4: Wine Searching. "It shows me only the information I want from two different apps at once (P9)". Similar to Scenario 3, *"It's convenient and useful because I can access information I want without having to search through different apps (P10)".*

Scenario 5: Coffee Ordering. "The way we use smartphones is not autonomous enough because we need to find and navigate an app to access specific functionality. A-Mash is better in terms of efficiency because we can use any app's functionality instantly outside its app boundary. (P14)".

We summarize two main take-home messages from this user study: 1) Every user has a different preference in setting and using a functionality even within a single app, and the ability to customize UIs can be a huge advantage and satisfying for each user. 2) A-Mash could be especially powerful when conducting repeated tasks because of the shortcut effect it creates. Users can be as creative as possible in utilizing A-Mash to reduce time and simplify tasks they do using apps on a daily basis.

9.4 Comparing with the State-of-the-Art

Participants and apparatus. We recruited twenty-two new participants from online school communities (12 males, 10 females, mean age=23.9, stdev=4.2, max=34, min=19). Each participant was given two Pixel 4XL Android smartphone, one for using A-Mash and another for the Android's split-screen mode [15]. This time, for a fair comparison, participants were given the UI container app instead of the AR-container app for the A-Mash. Inside the UI container app, we provided shortcut buttons that bring up the pre-defined set of UIs necessary for each task scenarios. For the split-screen mode, we provided app pair shortcuts that automatically launch two pre-defined apps side-by-side in the split-screen mode.

Design. The experiment was conducted using within-subject user study. The independent variable was the system (i.e., A-Mash or the split-screen mode) that participants used for performing the given tasks, and the dependent variable was the subjective usability score of the system that they evaluated for each task scenario.

Study procedure. We once again used task scenarios in Section 9.1, but this time, we excluded the *scenario 2: Music Playing* because it does not involve multi-app execution.

At the start of each session, participants were given a quick tutorial on how to use A-Mash and the split-screen mode. Then, participants went through a short practice session to get familiar with the apps involved in the task scenarios. To reduce learning effects, we divided the participants into two groups: The first group was asked to perform each task using A-Mash first, and then do the same task with the split-screen mode, while the other group was asked to do it in the opposite order. After performing each task scenario, participants were asked to score how easy the task was on a 7-point Likert scale. After all scenarios were done, participants evaluated both systems using the System Usability Scale [5], and were asked about the overall preference towards two systems.

Results and findings. Overall, participants felt A-Mash has better usability than the split-screen mode. On average, A-Mash scored 71.1 (std=16.7) out of 100 in the System Usability Score, while the split-screen mode scored 62.6 (std=16.6). In addition, 16 out of 22 participants responded that they prefer A-Mash over the split-screen mode, out of which 12 participants selected either *"very prefer"* or *"prefer"* (i.e., 6 and 7 score on a 7-point Likert scale).

We also compared the average score of how easy the task was when using A-Mash and the split-screen mode. On average, A-Mash scored higher than the split-screen in all scenarios. In particular, A-Mash has significantly higher score in *Wine Searching* and *Coffee ordering* scenarios, with Wilcoxon signed-rank test results of (W = 11, p = 0.000) and (W = 13, p = 0.001), respectively. The most dominant reason behind such result was the small window size of the split-screen. 17 out of 22 participants commented at least once that the split-screen feels uncomfortable because the UIs and the app's visible area becomes too small to interact with. On the other hand, all participants responded at least once that A-Mash is useful because it only the selective UIs were displayed in the screen, which made each UI be more visible and easily interactable. Some comments are as follows: *"It (A-Mash) removes all the unnecessary UIs and collects only what I needed. It feels very neat (P8)", "In the case of split screens, it was inconvenient because the apps were not tailored to the size of the split-screen. Buttons were reduced to small size, and I accidentally exited the split-screen several times trying to adjust the window size. In the case of A-Mash, all UIs were easy to see on a large screen, so they were easy to use. (P12)."*

One interesting observation from the users responses is that out of 66 responses that said using A-Mash was very easy (i.e., 6 and 7 score on a 7-point Likert scale), 50 said it was *because A-Mash reduces the number of steps to conduct a task and shows only the necessary UIs.* Another interesting observation is that out of 16 cases where participants felt using the split-screen mode is easier than using A-Mash, 13 said it was because they were unsatisfied with the pre-defined set of UIs that we provided. These observations in-line with the two take-home messages from the second user study: 1) Users have different preferences in settings, thus ability to customize the set of UIs is a huge advantage, and 2) A-Mash could be especially powerful for tasks repeated on daily basis, thanks to its ability to reduce the steps and simplify the tasks.

10 RELATED WORK

UI mashup in other fields. UI Mashup is an area of research actively being explored in the web environment. Fusion [38] enables web developers to extract UI elements from existing webpages and turn them into a javascript gluecode. The developers can embed these gluecodes into their webpages to create a UI mashup out of existing webpages. C3W [8] is a web framework that allows end-users to extract input elements from existing webpages and mash them together into a spreadsheet-like container webpage. Although they provide a useful UI mashup experience, their solutions are not applicable to mobile environments. Unlike web environments where UI's source code is easily accessible through the HTML DOM interface and multiple webpages can run concurrently in the foreground, in mobile environments, app's entire source code is hidden and the number of app that can run in the foreground is restricted to one. Therefore, the technical challenges that A-Mash addresses are completely different from that of previous works. To the best of our knowledge, A-Mash is the first work to bring the concept of UI mashup into the mobile environment.

Beyond the web environment, few works have attempted to provide UI mashups using desktop GUI apps [6, 31, 32]. These works operate at the pixel level to copy and paste specific area of the app into another window. However, as confirmed in Section 8.2, pixel-level approach inflates too much delay in the mobile environment.

Other forms of functionality mashup. There are other forms of functionality mashups besides using UI. The most widely used approach is data-centric mashups [7, 23, 34], where uses can select webpage elements to export its data and manipulate them to their needs. A similar approach, scrAPIr [1] seeks to make web Data API accessible by the end-users so that end-users can access their data without programming. In the mobile environment, Xdroid [19] provides mobile app developers with a mashup library that can emulate the functionalities of other existing apps. Developers can use Xdroid library to specify which functionality to emulate and embed into their apps. Although they provide mashup experience through their own means, their usage scenarios are either confined only to data mashup or targets developers, not the end-users.

Existing techniques for multi-app interaction. There are few existing techniques that allow users to interact with multiple apps concurrently on a single screen. Split-screen mode [2, 15] allows two apps to run side by side on a single screen and free-form mode allows apps to run in separate movable, resizable windows. Although these techniques could be useful for devices with a larger screen, it makes the app look too small to properly interact with. Some other options include PiP mode [16] where an app can display its video in a pinned small window overlaying other apps, and launcher widget [11] that allows developers to create a miniature app that can be embedded into launcher apps. These techniques can make use of apps' UIs in a more flexible ways, but it is confined to specific types of UIs; PiP mode only supports video UI, and launcher widget requires developers to write a separate app for each widget.

App shortcuts A few works provide shortcuts to app's specific activity or tasks. uLINK [4], SUGILITE [22], and RandR [27] adopt record-and-replay techniques to allow users to create a custom automation script. Apple's shortcut [3] and Samsung's routines [28] allow users to create an automated script out of special apps that exports its functionalities. They can serve as a good task automation tool, but they cannot provide interactive multi-app experience as A-Mash does.

11 DISCUSSION

Extensibility for UI container apps. Although the current version of A-Mash provides only the UI container app with a blank screen for UI mashup, we have already demonstrated in Section 9 that it can be successfully extended to an AR-based container app.

Based on lessons from this case study, we can think of two extension ideas to unlock the potential of UI container apps. One is to extend existing commercial apps themselves by providing UI mashup APIs. They allow third-party app developers to easily embed UI elements from different legacy apps into their own apps. As a result, this feature will induce multi-app scenarios that borrow functionalities from different apps in the unit of UI elements, which are not supported by traditional mobile systems yet. The other is to develop container apps specialized for AR/VR platforms such as Google ARCore [10] and Microsoft HoloLens [24]. If the container apps can actively utilize various features (e.g., motion tracking) provided from AR/VR platforms rather than simply using a camera as in the current A-Mash, we can elicit interesting use cases suitable for AR/VR environments. To this end, it is necessary to support advanced technologies for inter-platform UI mashup beyond the level of inter-app UI mashup supported by A-Mash. We leave these two extensions as our future works.

Unsupported commercial apps. A-Mash cannot support commercial apps that employ customized UI engines such as unique apps developed with cross-platform tools (e.g., Flutter [13], React Native [26]) or 3D games using third-party game libraries (e.g., Unity [33]). This is because UI elements of such apps are managed through different data structures placed inside the app-level UI engines instead of accessible UI trees provided by mobile platforms. This makes it impossible for the current prototype of A-Mash to perform UI-driven record-and-replay and extract some UIs from apps for UI mashup. However, because the app-level UI engines also internally utilize UI tree structures similar to Android, we can comprehensively apply A-Mash 's design if they are provided as open-source.

Invalidation of UI mashup specifications. When some legacy apps are updated, relevant specifications (i.e., XML files for UI mashup) may be invalidated because they may become inconsistent with the updated apps. For example, this error may occur if an updated app's activity transition patterns are entirely changed or its UI elements for mashup are removed. To handle such problems, A-Mash needs to verify whether all target UIs of legacy apps can still be extracted whenever they are updated. If some UI elements cannot be found for some reason, A-Mash can delete only the parts corresponding to the UIs from specifications and inform users about this situation.

Potential security implications. As A-Mash presents a new app interaction mechanism, new security threats could arise. For instance, it can be exploited for a spoofing attack. With A-Mash, a malicious app can easily mimic not only the visual appearance, but also the functional behavior of other benign apps just by borrowing their UIs. This would make spoofing attacks even harder to detect. Furthermore, our offscreen-display can be exploited to secretly perform malicious activities or inject synthetic input into the benign apps. To prevent such security threats, features of A-Mash (e.g., Offscreen-Display, UI extraction, cross-process UI embedding) should be hidden from the user-level applications. All its functionalities should be performed only with the system level permission. In other words, the APIs to trigger these functionalities should be opened only to apps with system permissions, so that third-party malicious apps cannot exploit them. In practice, it would be ideal for platform or device vendors to develop the container app themselves, and pre-build it inside the platforms, just as they would do with other system apps (e.g., Google PlayStore, Gallery, Camera).

Intrusions on app governance and app's business model. The concept of extracting and mashing up only "parts" of the app can potentially corrupt the intended display of the apps and violate the expectation of the app developers. For instance, some relationships between UI elements that convey necessary information to the user, including disclaimers, caution symbols, and other necessary disclosures in using the apps, could be hidden from the users. In addition, A-Mash can be a direct circumvention to the in-app advertisement policies since it enables users to use the app's features without watching the in-app advertisement. To prevent such intrusion, we can provide app developers with the means to specify their intentions on the UI elements. More specifically, we can add a few UI attributes/developer APIs that app developers can use to specify which UIs are not allowed to be extracted, and which UIs must be included in the mashup by default when using their apps for the UI mashup.

Stability and compatibility of underlying OS abstractions. Although the implementation of the A-Mash is specialized for Android, its key design components are not specific to the Android platform. The underlying OS abstractions, `Logical display, UI tree`, and `Activity`, are abstractions commonly used in many mobile platforms under different names (e.g., `RootViewController`, `Element tree`, and `ViewController` respectively in iOS). Therefore, with proper implementation, our design can be applied to many other mobile platforms as well. Moreover, since these abstractions have been the foundations of the mobile operating systems since their early stage [9, 12, 18], it is unlikely for them to be deprecated in the near future.

12 CONCLUSION

We designed and implemented A-Mash, a mobile platform that provides an innovative way to interact with multiple apps simultaneously on a single screen. A-Mash enables end-users to extract UIs from multiple apps and mash them up into single screen to create their personalized mobile interface that encompasses multiple app's functionalities. A-Mash selectively extracts UI elements from existing mobile apps, seamlessly embed extracted UIs into our container app, and provide apps with a transparent execution environment so that they can work behind the scene to give a flawless single-app illusion. Our prototype implementation has proven that A-Mash is high transparency, coverage, performance, and usability. We expect A-Mash to foster development of creative and useful multi-app usage scenarios to provide richer and more interactive mobile user experience.

ACKNOWLEDGEMENTS

We thank our anonymous reviewers and shepherd for their insightful and constructive comments that helped us improve this paper. This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science and ICT (NRF-2021R1F1A1063785, NRF-2022R1C1C1003123, NRF-2022R1C1C1012664, NRF-2020R1A2C2005479, and ERC (NRF-2018R1A5A1059921)).

REFERENCES

[1] Tarfah Alrashed, Jumana Almahmoud, Amy X Zhang, and David R Karger. 2020. ScrAPIr: Making Web Data APIs Accessible to End Users. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI)*.

[2] Apple. 2022. Multitasking and Multiple Windows. https://developer.apple.com/design/human-interface-guidelines/ios/system-capabilities/multitasking/.

[3] Apple. 2022. Shortcuts User Guide. https://support.apple.com/guide/shortcuts/welcome/ios.

[4] Tanzirul Azim, Oriana Riva, and Suman Nath. 2016. ULink: Enabling User-Defined Deep Linking to App Content. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[5] John Brooke. 1995. SUS: A quick and dirty usability scale. *Usability Eval. Ind.* 189 (11 1995).

[6] Morgan Dixon and James Fogarty. 2010. Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*.

[7] Rob Ennals, Eric Brewer, Minos Garofalakis, Michael Shadle, and Prashant Gandhi. 2007. Intel Mash Maker: Join the Web. *ACM SIGMOD Record* (2007).

[8] Jun Fujima, Aran Lunzer, Kasper Hornbæk, and Yuzuru Tanaka. 2004. C3W: Clipping, Connecting and Cloning for the Web. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters (WWW Alt.)*.

[9] Google. 2022. Activity. https://developer.android.com/reference/android/app/Activity.

[10] Google. 2022. ARCore: Build new augmented reality experiences that seamlessly blend the digital and physical worlds. https://developers.google.com/ar.

[11] Google. 2022. Create a simple widget. https://developer.android.com/guide/topics/appwidgets.

[12] Google. 2022. Display. https://developer.android.com/reference/android/view/Display.

[13] Google. 2022. Flutter: Build apps for any screen. https://flutter.dev/.

[14] Google. 2022. Multi-Resume. https://source.android.com/devices/tech/display/multi_display/multi-resume.

[15] Google. 2022. Multi-window support. https://developer.android.com/guide/topics/large-screens/multi-window-support.

[16] Google. 2022. Picture-in-picture (PiP) support. https://developer.android.com/guide/topics/ui/picture-in-picture.

[17] Google. 2022. Profile battery usage with Batterystats and Battery Historian. https://developer.android.com/topic/performance/power/setup-battery-historian.

[18] Google. 2022. View. https://developer.android.com/reference/android/view/View.

[19] Donghwi Kim, Sooyoung Park, Jihoon Ko, Steven Y Ko, and Sung-Ju Lee. 2019. X-Droid: A Quick and Easy Android Prototyping Framework with a Single-App Illusion. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST)*.

[20] Sunjae Lee, Hayeon Lee, Hoyoung Kim, Sangmin Lee, Jeong Woon Choi, Yuseung Lee, Seono Lee, Ahyeon Kim, Jean Young Song, Sangeun Oh, et al. 2021. FLUID-XP: Flexible User Interface Distribution for Cross-Platform Experience. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (MobiCom)*.

[21] Tong Li, Mingyang Zhang, Hancheng Cao, Yong Li, Sasu Tarkoma, and Pan Hui. 2020. "What Apps Did You Use?": Understanding the Long-Term Evolution of Mobile App Usage. In *Proceedings of The Web Conference 2020* (Taipei, Taiwan) *(WWW '20)*. Association for Computing Machinery, New York, NY, USA, 66–76. https://doi.org/10.1145/3366423.3380095

[22] Toby Jia-Jun Li, Amos Azaria, and Brad A Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI)*.

[23] Lin, James and Wong, Jeffrey and Nichols, Jeffrey and Cypher, Allen and Lau, Tessa A. 2009. End-User Programming of Mashups with Vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces (IUI)*.

[24] Microsoft. 2022. HoloLens: Mixed Reality Technology for Business. https://www.microsoft.com/hololens.

[25] Sangeun Oh, Ahyeon Kim, Sunjae Lee, Kilho Lee, Dae R Jeong, Steven Y Ko, and Insik Shin. 2019. FLUID: Flexible User Interface Distribution for Ubiquitous Multi-Device Interaction. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom)*.

[26] Meta Platforms. 2022. React Native: Learn once, write anywhere. https://reactnative.dev/.

[27] Onur Sahin, Assel Aliyeva, Hariharan Mathavan, Ayse Coskun, and Manuel Egele. 2019. RANDR: Record and Replay for Android Applications via Targeted Runtime Instrumentation. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[28] Samsung. 2022. Set up and use Bixby Routines on your Galaxy phone. https://www.samsung.com/us/support/answer/ANS00083201/.

[29] Samsung. 2022. Using the Smart Pop-Up View my Galaxy Phone. https://www.samsung.com/au/support/mobile-devices/using-the-smart-pop-up-view/.

[30] Skia. 2022. Welcome to Skia: The 2D Graphics Library. https://skia.org/.

[31] Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Phillips, and Nicolas Roussel. 2006. User Interface Façades: Towards Fully Adaptable User Interfaces. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST)*.

[32] Desney S Tan, Brian Meyers, and Mary Czerwinski. 2004. WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. In *Proceedings of the Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA)*.

[33] Unity. 2022. Real-Time Development Platform. https://unity.com/.

[34] Jeffrey Wong and Jason I. Hong. 2007. Making Mashups with Marmite: Towards End-User Programming for the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*.

[35] Jian Xu, Qingqing Cao, Aditya Prakash, Aruna Balasubramanian, and Donald E Porter. 2017. UIWear: Easily Adapting User Interfaces for Wearable Devices. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking (MobiCom)*.

[36] Jian Xu, Suwen Zhu, Aruna Balasubramanian, Xiaojun Bi, and Roy Shilkrot. 2018. Ultra-Low-Power Mode for Screenless Mobile Interaction. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST)*.

[37] Qiang Xu, Jeffrey Erman, Alexandre Gerber, Zhuoqing Mao, Jeffrey Pang, and Shobha Venkataraman. 2011. Identifying Diverse Usage Behaviors of Smartphone Apps. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC)*.

[38] Xiong Zhang and Philip J Guo. 2018. Fusion: Opportunistic Web Prototyping with UI Mashups. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST)*.

[39] Jiahuan Zheng, Xin Peng, Jiacheng Yang, Huaqian Cai, Gang Huang, Ying Zhang, and Wenyun Zhao. 2017. CollaDroid: Automatic Augmentation of Android Application with Lightweight Interactive Collaboration. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW)*.

[40] Jiahuan Zheng, Liwei Shen, Xin Peng, Hongchi Zeng, and Wenyun Zhao. 2020. MashReDroid: enabling end-user creation of Android mashups based on record and replay. *Science China Information Sciences* 63, 10 (2020).