FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Výkonnostní porovnání jazyků Kotlin, C/C++ a Dart na Android OS
Student:	Bc. Michal Lepíček
Vedoucí:	Ing. Michal Havryluk
Studijní program:	Informatika
Studijní obor:	Systémové programování
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

1) Seznamte se s jazykem Kotlin JVM a Dart včetně Flutter SDK používaným pro multiplatformní vývoj pro Android a iOS.
2) Proveďte analýzu Kotlin JVM, C/C++ a Dart s ohledem na způsob kompilace a alokaci zdrojů systému Android.
3) Proveďte rešerši současných výkonnostních metrik pro Android.
4) Navrhněte podobu úloh pro zpracování velkého objemu dat (čtení ze souboru, analýza/transformace a zápis do úložiště) a vykreslování komplexních UI.
5) Implementujte řešení úloh z předchozího bodu pomocí Dart (Flutter SDK), C/C++ (Android NDK) a Kotlin JVM (Android SDK).
6) Porovnejte a vyhodnoťte rychlost zpracování dat, vykreslování UI a míru využití hardwarových prostředků.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 15. února 2019

ČESKÉ VYSOKÉ
UČENÍ TECHNICKÉ
V PRAZE

F8

Fakulta informačních technologií
Katedra teoretické informatiky

Diplomová práce

Výkonnostní porovnání jazyků Kotlin, C/C++ a Dart na Android OS

Bc. Michal Lepíček
Systémové programování

Květen 2019
Vedoucí práce: Ing. Michal Havryluk

Poděkování / Prohlášení

Rád bych poděkoval své rodině za neustálou podporu, své babičce za jazykovou korekturu a své přítelkyni za trpělivost v tomto hektickém období. Mé díky také patří Michalu Havrylukovi za vedení této práce a v neposlední řadě přátelům, kteří mi zpříjemnili průchod celým studiem.

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 13. 13. 2013

.....

Abstrakt / Abstract

Tato diplomová práce se zabývá měřením výkonnosti aplikací pro mobilní operační systém Android. Nezbytnou součástí je návrh a implementace pomocí jazyků Kotlin (Android SDK), C/C++ (Android NDK) a Dart (Flutter SDK). V práci se zabývám různými faktory při vývoji a hlavně běhu aplikací na systému Android. Tyto faktory jsem analyzoval a podle nich navrhl dílčí úlohy, které jsou výkonnostně změřeny a porovnány mezi zmíněnými jazyky.

Klíčová slova: Android, Flutter, Dart, Kotlin, C/C++, výkonnost

This thesis is focused on measurement of performance of applications for Android, the mobile operating system. An essential part is design and implementation of applications via programming languages Kotlin (Android SDK), C/C++ (Android NDK) and Dart (Flutter SDK). In this work I am dealing with various factors of development process and especially runtime of Android applications. These factors I analyzed and accordingly designed partial tasks whose measured performance is compared among said languages.

Keywords: Android, Flutter, Dart, Kotlin, C/C++, performance

Obsah

Tabulky / Obrázky

Kapitola 1
Úvod

Téměř všichni máme chytré mobilní telefony a používáme nezměrné množství aplikací napsaných právě pro ně. Tyto aplikace mohou být vytvořeny mnoha způsoby, které bych rozdělil na 3 kategorie. Nativní přístup neboli psaní Android aplikací v jazyku Java/Kotlin a iOS aplikací v jazyku Objective-C/Swift. Dalším přístupem je použití jazyka kompilovaného do nativního kódu, C/C++ například, a vytvoření celé aplikace od nuly. Tento přístup s sebou přináší vysokou výkonnost, ale zároveň nutnost implementovat i části, které již někdo vytvořil a jsou k dispozici skrze mobilní systém či framework. Třetí kategorií je multiplatformní vývoj aplikací. V tomto případě se píše větší část aplikace pouze jednou pro oba systémy Android i iOS, ale často taková aplikace není koncovými uživateli přijímána tak dobře, jako kdyby byla aplikace napsaná nativně. Jde zřejmě o kompromisy v UI, protože většinou se musí vybrat standard z jedné platformy a ten zobrazovat i na platformě druhé. Často jde i o samotnou rychlost aplikace, protože nevyužívá nativní komponenty systému, nýbrž nadstavbové, které teprve poté interagují s nativními komponentami.

V roce 2015 byl poprvé odhalen Flutter – framework, který se řadí do třetí kategorie, tedy multiplatformní. Flutter se od již zavedených multiplatformních řešení, jako je například React Native, odlišuje tím, že kompiluje zdrojový kód do nativního. To zajišťuje vyšší výkonnost než většina multiplatformních řešení, které využívají Javascript a tzv. „bridge“ pro komunikaci s nativními komponentami.

Na konci roku 2018 byla představena první stabilní verze tohoto frameworku. Slibuje rychlý vývoj, UI dle platformy, na které aplikace běží, vykreslování v 60/120 FPS a další. Všechny tyto přísliby, a kompilace do nativního kódu, mě přesvědčily o tom, změřit výkonnost aplikací napsaných v tomto frameworku a zhodnotit zda je možné Flutter považovat za nástupce standardního řešení využívaného pro multiplatformní vývoj na mobilních operačních systémech.

1.1 Cíl práce

Cílem práce je navrhnout, zrealizovat a zhodnotit výkonnost aplikací ve třech kategoriích:

1. Zpracování dat (čtení ze souboru, analýza/transformace, zápis do úložiště)
2. Vykreslování UI
3. Využití hardwarových prostředků

Cíleným mobilním operačním systémem pro měření výkonnosti je Android s aplikací psanou:

1. V jazyce Kotlin (Android SDK)
2. V jazyce C/C++ (Android NDK)
3. V jazyce Dart (Flutter SDK)

1.2 Proč Flutter a Dart

Na trhu již existuje několik možností jak vyvíjet aplikace pro systém Android a případně multiplatformně pro iOS. Proč tedy pro porovnání zvolit pouze Flutter s jazykem Dart a ne jiná, již několik let zaběhnutá řešení? Pro odpověď na tuto otázku není zapotřebí žádná analýza, ale stačí pouhopouhé obecné znalosti ohledně konkurenčních řešení pro vývoj aplikací pro Android, zejména funkčnost těchto řešení.

Prvně bych rozdělil vývojové možnosti do tří kategorií. Možnostmi budiž: nativní jazyk, hybridní aplikace a kompilované aplikace.

1.2.1 Nativní jazyk

Pro systém Android se jedná o Googlem vyvíjený Android SDK, který podporuje jazyk Java a od konce roku 2017 jazyk Kotlin. Tyto jazyky jsou pro platformu Android „nativní“. Pro platformu iOS by se jednalo o jazyky Objective C a Swift.

Java i Kotlin jsou (v případě systému Android) kompilovány do JVM bajtkódu.

Hlavní výhodou této kategorie je primární zaměřenost vývojářů systému čili dostupnost veškerých nejnovějších funkcí a vlastností systému. Z pohledu multiplatformních vývojářů lze též vyvodit jasnou nevýhodu v nutnosti umět a hlavně napsat aplikaci ve dvou jazycích – tedy stejnou funkcionalitu implementovat dvakrát.

1.2.2 Hybridní aplikace

Toto je pravděpodobně nejjednodušší kategorie. Jedním z představitelů této kategorie je Ionic. Využívají se vývojářům známé jazyky HTML, CSS a Javascript, které se používají pro vytvoření webových stránek. Myšlenka je přímočará – aplikace disponuje jediným prvkem WebView, které načte webovou stránku navrženou tak, aby vypadala, že jde o plnohodnotnou aplikaci.

Výhodou je napsání pouze jediné webové stránky, která nebude fungovat pouze na klasickém desktopovém prohlížeči, ale též na mobilní platformě v podobě aplikace. Nevýhodami jsou zjevná pomalost zobrazení rozhraní aplikace či komunikace Javascriptu s nativními komponentami systému a limity WebView na dané mobilní platformě.

1.2.3 Kompilované aplikace

Tuto kategorii nazývám „kompilované aplikace“, protože se jedná o přístup, kdy se napíše kód v jistém jazyku, který bude (alespoň částečně) zkompilován do knihovny. Nativní kód poté využije tuto knihovnu, což má za následek vyšší rychlost než hybridní aplikace, případně i než nativní jazyky. Pojem rychlejší je pro každého subjektivní, avšak zde jde o označení/popis situace neexistujícího prostředníka mezi využívaným jazykem a nativními komponentami systému - Javascript pro hybridní aplikace a případný virtuální stroj pro nativní jazyk.

Do této kategorie patří mj.: React Native, Native Script a také **Flutter**. Rozdíly mezi těmito zmíněnými jsou: Flutter využívá jazyk Dart, kdežto zbylé dva využívají Javascript. Důležitějším rozdílem je, že Dart se kompiluje celý do ARM C/C++ knihovny, kdežto zbylé dva kompilují pouze UI elementy do jejich nativních ekvivalentů a zbytek běží stále jako Javascript, který musí komunikovat s nativními komponentami systému skrze prostředníka.

Patří sem i využití C/C++, ale podpora tohoto jazyka nezahrnuje jednoduché navržení celé aplikace, což bude vysvětleno v této práci.

To, že je Dart kompilován celý bez kompromisu je hlavním důvodem, proč výkonnostně měřit pouze tento jazyk.

Kapitola 2
Analýza

Cílem této práce je zjistit, který framework a popřípadě jazyk je vhodný pro vývoj na mobilní systém Android. Každý vývojář má své osobní preference a proto nebývají porovnání vždy přínosná. Lze se však zaměřit na několik vlastností, které objektivně přispívají k rozhodnutí, který framework či jazyk použít za účelem splnění cílů vyvíjené aplikace. Prvně je tedy třeba analyzovat možnosti, na které se tato práce soustředí, aby bylo jasné v čem se liší, na co se zaměřují a v čem je tedy porovnávat.

2.1 Vykreslování

Pro všechny jazyky, frameworky a většinu zařízení platí jeden údaj – **16 ms**.

Aby aplikace byla plynulá, respektive se tak zdála lidskému oku, tak musí splňovat vykreslování obrazovky s konstantní frekvencí. To je všeobecně známo jako FPS neboli kolik snímků je vykresleno za jednu sekundu. Kvůli hardwarovým požadavkům je záhodno hodnotu FPS synchronizovat s obnovovací frekvencí displeje, která se udává v jednotkách Hertz (Hz).

Drtivá většina zařízení využívá obnovovací frekvenci 60 Hz, což v praxi implikuje žádoucích 60 FPS.

$$1\ s = 1000\ ms$$
$$1000\ ms\ /\ 60\ frames = 16.666\ ms\ /\ frame$$

Z výše uvedeného výpočtu vychází, že na každé vykreslení obrazovky má zařízení 16.666 ms – většinou zkracováno na 16.7 ms či stejně jako v této práci na 16 ms. To v praxi znamená, že pokud špatná implementace zablokuje vykreslování na déle než 16 ms nebo je právě vykreslovaná obrazovka příliš složitá pro 16 ms, tak najednou klesne hodnota FPS. Kolísavá hodnota FPS se projevuje neplynulostí aplikace, zdánlivým zasekáváním či hovorově „lagováním“ – jank[1].

Pokud se zapne *Profile GPU rendering* jako sloupce na obrazovce (Nastavení > Možnosti pro vývojáře) a aplikace byla vytvořena pomocí Android SDK, tak se na spodu obrazovky objeví sloupcový graf jako na obrázku 2.1. Každý sloupec v tomto grafu znamená dobu vykreslování v jednom snímku. Zelená horizontální čára označuje „hranici plynulosti“ – 16 ms. Na obrázku 2.1 lze tedy spočítat, že došlo 16× k výskytu janku.

[1] Přiřazené slovo mobilními vývojáři pro situaci, kdy se v aplikaci projevuje náhlý pokles hodnoty FPS – například „zasekávání“

Obrázek 2.1. Sloupcový graf doby vykreslování přístupný na Android zařízeních

2.2 Kotlin JVM

Jméno tohoto jazyka je Kotlin. Pokud se o něm hovoří v kontextu systému Android, tak se jedná o jednu z jeho variací, která je cílená na JVM. Dalšími variacemi je kompilace do Javascriptu a kompilace do nativního kódu (pomocí LLVM), která lze taky použít v systému Android, ale tato možnost ještě nebyla zdaleka standardizována.

Jak již bylo zmíněno, Kotlin na Androidu se kompiluje do JVM bajtkódu, což prozrazuje fakt, že rozdíl mezi Kotlinem a Javou na Androidu je minimální. Napovídá tomu i skutečnost, že jazyky jsou vzájemně interoperabilní. Lze tedy usoudit, že není třeba porovnávat Javu a Kotlin zvlášť, neb výsledky by byly přiměřeně stejné.

> *Jazyk Java byl donedávna primárním a jediným oficiálním jazykem pro vývoj nativních aplikací. Na konci roku 2017 se druhým, avšak pro mnohé primárním, jazykem stal Kotlin s jeho modernější syntaxí a bohatým množstvím funkcí.*

2.2.1 Kompilace a běh

Kotlin (stejně jako Java) je prvně zkompilován do JVM kompatibilního bajtkódu. Ten se následně může obfuskovat a minimalizovat. Po provedení obou operací jsou výsledkem soubory `.class`. Nyní se soubory překompilují do Android bajtkódu `.dex`. Tyto soubory jsou následně zpracovány do jednoho instalačního balíčku `.apk`.

Instalace a běh aplikace se liší dle verze Androidu. Pokud se používá starší Dalvik VM, tak jsou použity soubory `.dex`, které se zároveň interpretují. Nadruhou stranu novější ART spouští nativní kód, který byl vytvořen při instalaci `.apk` balíčku.

Z předchozího odstavce lze vyvodit hlavní rozdíl mezi Dalvik a ART aneb spuštění kódu JIT respektive AOT. Výběr jedné z těchto možností přímo ovlivňuje rychlost aplikace, proto by bylo záhodno provést měření za použití obou možností.

> *Rok 2013 (Android 4.4 – Kitkat) přinesl podporu běhového prostředí ART, na které bylo možné systém přepnout v pokročilém nastavení. V roce 2014 (Android 5.0 – Lollipop) byl ART zvolen výchozím a kompletně nahradil dosavadní Dalvik. Nakonec, v roce 2016 (Android 7.0 – Nougat), byl Dalvik přiveden částečně zpět a začal být používán v kombinaci s ART.*

Ačkoliv ART zajišťuje rychlejší aplikace včetně spuštění, tak přináší nevýhody jako větší velikost aplikace nebo pomalejší instalace. Dalvik začal být používán, aby některé nevýhody zmizely. Například: Aplikace se nainstaluje pro Dalvik (rychlejší) a později v pozadí, když není zařízení využíváno, se nainstaluje aplikace pro ART.

Předním frameworkem je **Android SDK**, který obsahuje mj. potřebné knihovny, ladící nástroje a emulátor.

2.2.2 UI – XML

UI se zpravidla vytváří za pomocí značkovacího jazyka XML. Soubory, které obsahují definici toho jak má obrazovka nebo prvek obrazovky vypadat se ukládají do složky `/res/layout/`. Tam lze najít všechny typy rozvržení obrazovky, které v této práci testuji pro výkonnost – rychlost vykreslování UI.

Nejdůležitějším prvkem v jednom layoutu[1] je pravděpodobně nejvrchnější rodič – `layout view`. To určuje, jak se všechny prvky zaobalené v něm (dále jen „potomci“) mají organizovat a řadit. Android SDK od svého vzniku představil několik různých typů `layout view`, které lze na tomto místě využít:

1. AbsoluteLayout
2. TableLayout
3. FrameLayout
4. LinearLayout
5. RelativeLayout
6. ConstraintLayout

Lze nalézt více spadajících do této kategorie, ale již se nejedná o řešení, která lze vytvořit pouze za pomocí XML. První `layout view` je již dlouho deprecated[2]. Druhé `layout view` může najít uplatnění, ale v naprosté většině se od něj upouští, jelikož je příliš omezené. Zbylé čtyři `layout view` jsou denně používány.

FrameLayout je nejjednodušší a tedy nejlehčí z pohledu výkonu potřebného pro jeho syntaktickou analýzu, která předchází téměř každému vykreslení. Sám neurčuje žádnou organizaci, ale potomci samotní mohou definovat svou pozici vůči němu. Lze ale použít jen jednoduché pozicování typu vlevo, nahoře, nahoře-uprostřed a absolutně udávat mezeru mezi hranou rodiče a potomkem samotným. Pokud se žádné pozicování neuvede, potomci se vykreslí do levého horního rohu a každý další potomek překryje ten předchozí.

LinearLayout je jedním ze dvou nejpoužívanějších `layout view`. Umožňuje potomky řadit za sebe v pořadí uvedeném a to vertikálně nebo horizontálně.

RelativeLayout je druhým ze dvou nejpoužívanějších `layout view`. Umožňuje, aby se potomci řadili relativně k sobě navzájem a k rodiči. Například lze uvést, aby potomek byl umístěn horizontálně uprostřed rodiče a vertikálně pod prvkem jiným.

[1] layout – rozvržení UI prvků na obrazovce nebo v jinak rozměrově definované oblasti obrazovky
[2] zastaralé, nevhodné k použití, existuje pouze pro zpětnou kompatibilitu

ConstraintLayout je nejnovějším z `layout view`. Zda již je nebo bude nejpoužívanějším se nelze dopátrat, ale jistě je to jeho ambice. Teoreticky se dá přirovnat k RelativeLayoutu, ale prakticky má mnohem více možností. Rozdíl je totiž v tom, že RelativeLayout pozicuje potomka k potomku jinému, kdežto ConstraintLayout vytváří tzv. „constraint“, vazbu chcete-li, mezi potomky. A právě tato vazba přináší do ConstraintLayoutu více možností, mezi které patří například:

- Bias – procentuální pozicování na jedné ose (např.: existují horizontální vazby k rodiči a bias je nastaven na 25 % – prvek je v jedné čtvrtině rodiče)
- Chain – `x` potomků může mít jednu vazbu a té pak určit zda prvky na ní roztáhnout, seskupit atd.
- GoneMargin – velikost mezery, když potomek, ke kterému vede vazba, zmizí
- 3. rozměrová možnost – kromě `match_parent` (rozměr rodiče) a `wrap_content` (rozměr dle obsahu) přibývá `match_constraint` (roztáhnout dle vazeb)

Nejdůležitějším faktorem ale zůstává, proč bylo toto `layout view` vytvořeno. S tímto `layout view` lze vytvořit jakkoliv komplikované rozvržení obrazovky a to bez jediného vnoření. Viz kapitola 2.2.3, vnoření je problémem rychlosti vykreslování.

2.2.3 UI – Vykreslování

Jak vlastně vzniká obrazovka? Jak se z XML nebo z `new Button()` stanou pixely na displeji zařízení? Proces to není jednoduchý a přesto musí být jedním z nejefektivnějších a nejoptimalizovanějších, aby se aplikace dala bezstresově používat. Jednoduše lze říci, že:

a) CPU vezme objekt a transformuje jej na seznam příkazů pro kreslení – **DisplayList**
b) Tyto příkazy jsou nahrány do GPU skrze OpenGL-ES API
c) GPU provede rasterizaci

Tento jednoduchý proces je dále třeba rozložit na složitější seznam fází:

Obrázek 2.2. Fáze vykreslování aplikací vytvořených s Android SDK [17]

1. VSync Delay / Misc Time
2. Input Handling
3. Animation
4. Measure / Layout
5. Draw
6. Sync & Upload
7. Command Issue
8. Swap Buffers

Všechny tyto fáze se provádí každý snímek (frame), tedy běžně 60× (někdy 30× či 120×) za sekundu. Je podstatné, aby jejich délka trvání byla co nejkratší, neboť přímo ovlivňují plynulost aplikace.
Vsync Delay fáze je prací na UI vlákně mezi dvěma po sobě jdoucími snímky. Vysoká hodnota stráveného času v této fázi znamená, že se provádí příliš náročný výpočet, který by měl být přesunut na vedlejší vlákno.

Input Handling fáze je aplikace vykonávající logiku uvnitř callbacků pro vstup uživatele – `onTouch()`, `onClick()`, `onScroll()` atd.

Animation fáze vyhodnocuje všechny animační objekty, které ovlivňují prvky v aktuálním snímku.

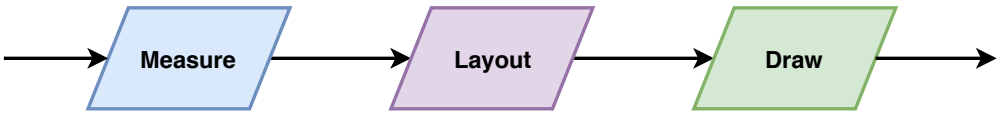
Measure / Layout fáze vyměřuje a pozicuje prvky, které mají být vykresleny na obrazovce.

Draw fáze vytváří nebo upravuje existující DisplayList. Ač se fáze jmenuje Draw, tak v tomto momentu se na obrazovku stále ještě nic nekreslí. V této fázi se pouze sbírají příkazy do DisplayListu, který bude zobrazen později.

Sync & Upload fáze je fází, kdy se přenáší bitmapové objekty z CPU paměti do GPU paměti. RAM paměti těchto jednotek, určené pro zpracovávání, jsou samozřejmě odlišné. Na obrazovku kreslí GPU, které musí všechny bitmapy nejdříve získat a následně si je uloží do své cache, aby se již tato fáze nemusela opakovat se stejnými bitmapami.

Command Issue fáze je prováděcí fází. Všechny příkazy pro vykreslování DiaplayListů se předávají OpenGL. Čas této fáze je přímo úměrný času potřebného pro vykreslení každého DisplayListu – čím více a složitějších DisplayListů, tím delší tato fáze.

Swap Buffers fáze je čekání na GPU. CPU řekne GPU, že má pro tento snímek hotovo a (**blokovaně**) čeká až GPU dokončí svou práci. Teprve potom může grafický ovladač zobrazit aktualizovaný snímek.



Obrázek 2.3. Hlavní CPU fáze vykreslování aplikací vytvořených s Android SDK

Ač je každá z výše uvedených fází důležitá, tak nejčastějšími fázemi, které je třeba kontrolovat a omezovat výskyt, jsou measure, layout a draw.

V **measure** části se musí určit velikost prvků. Velikost každého prvku obsahuje velikosti svých potomků a zároveň se musí vejít do svého rodiče. Každý prvek určí podmínky své velikosti buďto jako neurčitou, přesnou a nebo maximální. Tyto podmínky jsou pak poslány potomkům a proces se opakuje až k listům tohoto stromu.

Layout část získá přesnou velikost a pozici prvků. Každý rodič umístí své potomky dle změřených údajů v `measure` části.

Draw fáze se provádí po předchozím přesným zjištěním všech pozic a velikostí. Vygeneruje či upraví DisplayListy pro OpenGL.

Z předchozích bodů je očividné, že `layout` část je velice drahá. Faktem taky je, že potomek může v průběhu vyžadovat, aby se spustila `layout` část rodiče znova. Pokud se provádí `layout` část znova, pak se provádí i na všech potomcích. Toto je problém, který dokáže prodloužit vykreslování na obrazovku a způsobit jank. Například `RelativeLayout` (2.2.2) má garantovaně vždy dva `layout` průchody. První pro vypočítání pozic a velikostí všech potomků. Tyto vypočtené údaje poté využije při druhém průchodu, kdy určí finální pozice všech korelovaných potomků. Problémem tedy jsou vnořené `RelativeLayouty`, protože počet `layout` průchodů stoupá s každým vnořením exponenciálně. [16–18]

2.3 C/C++

2.3.1 Role v systému Android

Android aplikace jsou typicky napsány v Javě/Kotlinu. Nicméně, nastávají případy, kdy je třeba překonat limity Javy, jako jsou například paměťová správa či výkonnost.

Využívá se tedy hlavně v oblastech jako jsou hry nebo fyzikální emulace – výpočetně náročné aplikace.

2.3.2 Kompilace

Jazyky všem známý C a C++ jsou jazyky kompilované do nativního kódu neboli kód dle instrukční sady, kterému rozumí procesor na daném zařízení. Součástí nástrojů **Android NDK** je mj. kompilátor Clang (LLVM), který vývojářem napsaný kód zkompiluje do dynamické knihovny. Tato dynamická knihovna se pak zkopíruje do projektu a následně instalačního balíčku `.apk`.

> *GCC byl z Android NDK odstraněn v září 2018.*

2.3.3 JNI

JNI je rozhraní, které definuje možnost kódu spuštěným na virtuálním stroji Javy interagovat s nativními knihovnami a programy. [5]

Jde tedy o propojení, kdy vývojář z Java/Kotlin kódu volá funkce C/C++ kódu, což zajišťuje právě tato vrstva – rozhraní.

2.3.4 Instrukční sady

Android NDK podporuje několik architektur a s tím instrukčních sad, vyjmenované v tabulce 2.1, které se vyskytují na různých zařízeních se systémem Android. Nejrozšířenější je architektura ARM s malou percentuální účastí Intel x86 a zbytek je již nepodporovaný nebo zastaralý [2]. Malá variace použitých architektur je ku prospěchu velikosti aplikace, neb tato velikost roste lineárně s počtem podporovaných architektur. Každá podporovaná architektura implikuje jednu dynamickou knihovnu navíc.

ABI	Instrukční sada
armeabi	ARMV5TE and later Thumb-1
armeabi-v7a	armeabi Thumb-2 VFPv3-D16
arm64-v8a	AArch64
x86	x86 (IA-32) MMX SSE/2/3 SSSE3
x86_64	x86-64 MMX SSE/2/3 ...

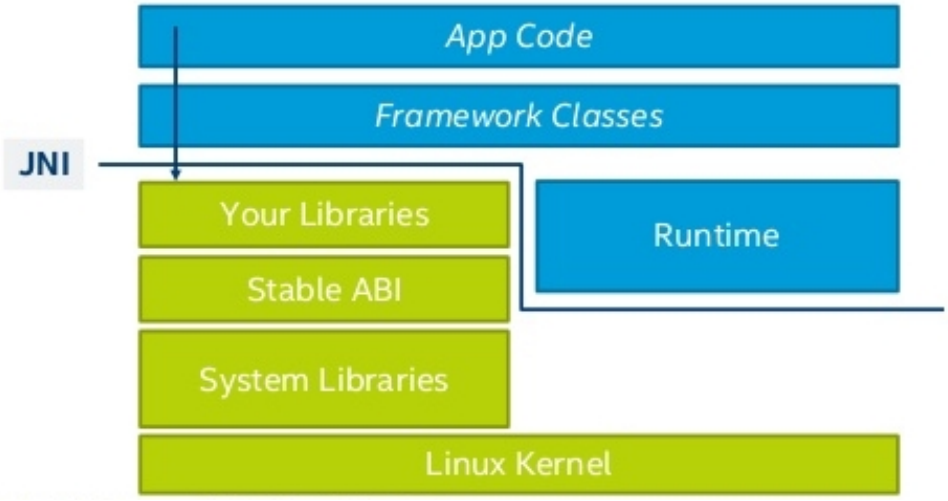
Tabulka 2.1. Instrukční sady, pro které lze zkompilovat C/C++ s Android NDK [1]

2.3.5 Běh aplikace

Běh aplikace se téměř neliší od standardní, avšak Kotlin funkce deklarované jako nativní mají implementaci v dynamické knihovně a odtud je kód též proveden. Na zjednodušeném obrázku 2.4 lze tento přístup rozpoznat. První vrstva *App Code* je spuštěná aplikace a druhá vrstva *Framework Classes* jsou komponenty a rozhraní v Javě poskytované vývojářům Kotlinu pro jednodušší psaní aplikací pro systém Android. Dále je tam vrstva runtime, která se stará o běh aplikace. Tyto tři vrstvy vzájemně spolupracují pro běžnou aplikaci napsanou čistě v Kotlinu.

Pod těmito vrstvami existuje rozhraní *JNI* z kapitoly 2.3.3. To zajišťuje komunikaci mezi první vrstvou a vrstvou *Your Libraries*. Tato vrstva je vývojářův zdrojový kód v C/C++ zkompilovaný do dynamické knihovny `.so`. Tento zdrojový kód může pracovat s vrstvou pod ní, *Stable ABI*, což jsou stabilní komponenty systému napsané v C/C++.

Výhoda je zřejmá – kód, který musí být výkonný/rychlý se vůbec nedotkne Javy a tedy je inherentně výkonnější/rychlejší.



Obrázek 2.4. Architektura systému Android s aplikací podporující Android NDK [3]

2.4 Dart

2.4.1 Role v systému Android

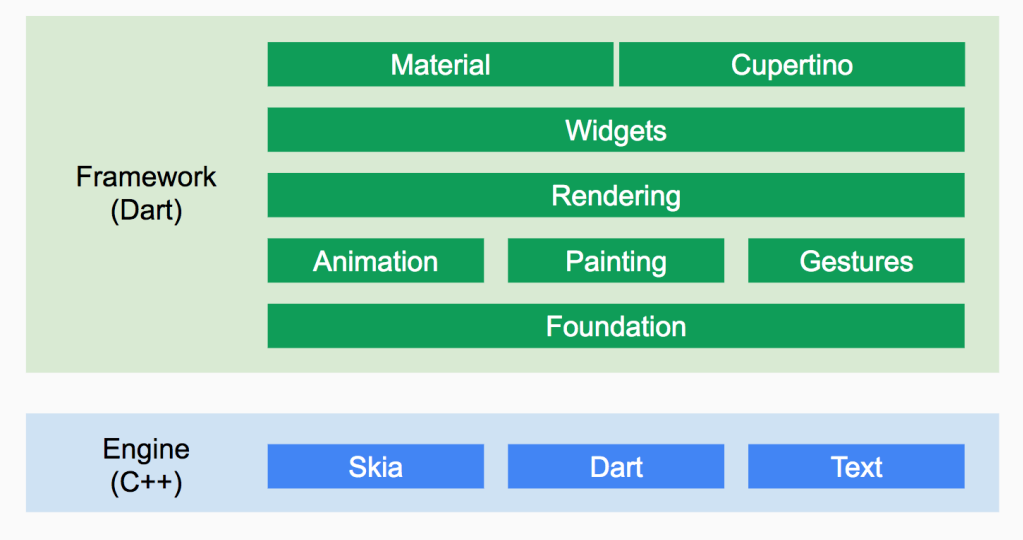
Když Google v roce 2011 odhalil jazyk Dart, tak jazyk neměl příliš velký ohlas. Přestože se dá využít v mnoha směrech, tak neměl žádnou oblast, kde by se svými vlastnostmi vítězil nad ostatními jazyky.

Avšak v roce 2015 odhalil Google nový framework **Flutter** a nástroje Flutter SDK pro vývoj mobilních aplikací, kde primárním jazykem je právě Dart.

Flutter byl vytvořen z důvodu absence frameworku, který by vytvářel stejně rychlé aplikace jako herní enginy – například Unity. Docíleno toho bylo skrze přístup velice efektivního a optimalizovaného kreslení na canvas. A protože canvas je k dispozici na většině platforem (iOS, Web, Desktop, ...), tak vedlejším produktem se stala i multiplatformnost, což bývá jedna z často zmiňovaných výhod tohoto frameworku. [4]

Další vlastností Flutteru je, že nepoužívá žádné komponenty poskytované Android frameworkem, konkrétně widgety. Widget ve Flutter terminologii je všechno, nicméně

zde se jedná o grafické prvky. Pro Flutter byly všechny grafické prvky vytvořeny od píky. V sekci nejčastěji pokládaných otázek pro Flutter [6] se lze dozvědět, že všechny komponenty (widgety ve Flutter terminologii) jsou napsány v Dartu a jen tenká vrstva kódu je napsaná v C/C++. [20]



Obrázek 2.5. Flutter architektura [20]

Na oficiálních stránkách lze najít propagovaný seznam výhod:

- Rychlý vývoj - zdánlivě irelevantní v této práci
- Výrazné a flexibilní UI - irelevantní v této práci
- Nativní výkonnost

Poslední bod říká, že aplikace má vykonávat práci srovnatelně s nativní úrovní. Toto tvrzení může být matoucí ze dvou důvodů. Prvně nevíme, co nativní znamená. Nativní jako nativní aplikace čili napsaná v Kotlinu/Javě? Nebo nativní jako napsaná v C/C++ a zkompilovaná do **nativního** kódu? Druhým důvodem pro zmatení je, že není známo, co má mít nativní výkonnost. Je to vykreslování, které má při správném programování garantováno dosáhnout 30/60 FPS [7]? Pravděpodobně. Druhou částí mince je výkonnost kódu. Vzhledem k tomu, že převážná většina frameworku je napsána v Dartu, lze tedy očekávat menší výkonnost než jakou dodává C/C++. Toto jsou otázky, které dávají rozměr této práci.

2.4.2 Kompilace

Flutter kompiluje zdrojový kód v Dartu do nativního kódu pro ARM architektury. Existují 3 módy kompilace. [8]

Debug. V tomto módu se běžně vyvíjí aplikace, protože se kompiluje metodou JIT a součástí aplikace jsou nejrůznější ladící nástroje. JIT zde umožňuje hlavní výhodu vývoje – „hot reload“; změny v kódu se projeví v aplikací téměř okamžitě a bez ztráty stavu aplikace. Na druhou stranu je přítomna nevýhoda pomalejší aplikace.

Release. V tomto módu jsou odstraněny všechny ladící nástroje, kód je minimalizován a optimalizován. Kompilováno je metodou AOT. Aplikace je tedy znatelně menší a rychlejší. O to déle trvá její kompilace a instalace.

Profile. Tento mód je kombinace dvou předešlých, kdy se využívá všeho z release módu, ale ponechá se většina ladících nástrojů.

Obrázek 2.6. Dart komponenty ve Flutteru v debug a release módu [29]

2.4.3 Běh aplikace

Jakmile je aplikace spuštěna, načte se Flutter plugin skrze který je jakékoliv vykreslování, vstup, události atd., delegováno přímo do AOT zkompilovaného zdrojového kódu aplikace či do Flutter komponent. Toto je velice podobný proces tomu, který provádějí herní enginy.

Rozdíl je pouze v debug módu kompilace, viz kapitola 2.4.2, ve kterém Dart VM zpracovává Dart kód. [6]

Flutter využívá čtyři vlákna pro vykonání práce za běhu aplikace. [28, 7]

- **Platform vlákno** je hlavním vláknem, ve kterém běží celý Flutter plugin.
- Na **UI vlákně** běží veškerý zdrojový kód aplikace. Vytváří se zde *layer tree*, který obsahuje příkazy pro vykreslování obrazovky (detailněji v pozdější kapitole).
- **GPU vlákno** přebírá *layer tree* a vykresluje jej na obrazovku skrze proces komunikace s GPU.
- **I/O vlákno** provádí nákladné operace (typicky I/O, například čtení souboru), které by jinak blokovaly UI a GPU vlákna.

2.4.4 Garbage Collector

Dalším rozdíl oproti Kotlinu (respektive ARTu) je **Garbage Collector**. Zatímco v Kotlin aplikacích se využívá hlavně algoritmus `(Concurrent) Mark&Sweep`, který je různě upravován a vylepšován pro Android, tak Dart využívá zároveň **dva** algoritmy – `Young Space Scavenger` a `Parallel Marking and Concurrent Sweeping`. Jeden z důvodů využití jazyku Dart ve Flutteru jsou právě tyto algoritmy Garbage Collectoru, které umožňují dodržet hypotézu, že všechny alokace by měly mít krátký život. Algoritmus `Young Space Scavenger` je blokový, leč mnohem rychlejší než `Mark&Sweep` a stará se tak o mazání všech krátkodobých objektů. Teprve když objekt přežije určitý počet uvolňování tímto algoritmem, tak je přesunut do části paměti, o kterou se stará `Mark&Sweep`. [30, 29]

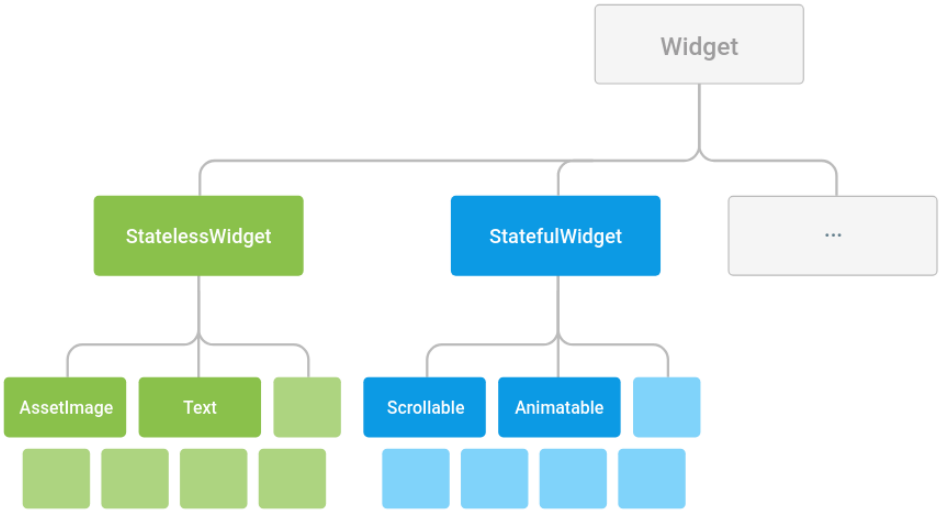
2.4.5 UI – Dart

Všechno je `widget`. `Widget` je základní stavební blok UI, tedy může být:

- Konstrukční prvek (např.: tlačítko)
- Stylistický prvek (např.: písmo, barva)
- Aspekt rozvržení (např.: padding)
- ...

Základní myšlenkou hierarchie `widget`ů je „Kompozice > Dědění“. Což je naprosto opačný přístup než v Android SDK. Rozdíl spočívá v tom, že v Android SDK, každé `View`[1] zdědilo několik základních parametrů jako jsou výška, šířka, barva, pozadí atd. Naproti tomu ve Flutteru, každý `widget` má pouze jednu malou úlohu; například umístit potomka na střed či nastavit pozadí. `Widget` může být i mnohem mocnější než jen jedna úloha, pak ale jde o kompozici několika menších `widget`ů s jednou úlohou.

Tento přístup zajišťuje přímočaré a jednodušší vykreslování prvků, viz následující kapitola 2.4.6.



Obrázek 2.7. Příklad stromu jednoho `widgetu` [20]

[1] View – základní prvek pro vytváření UI komponent v Android SDK

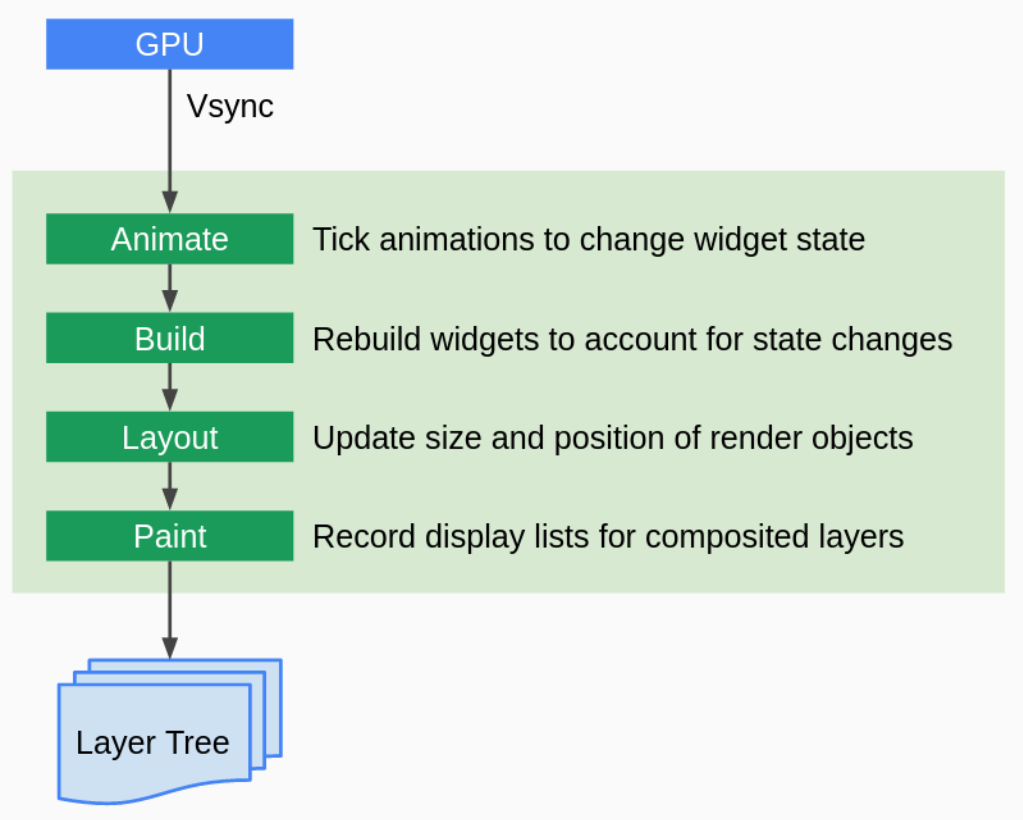
Existují dva základní stavební `widgety`, které slouží jako kořenový prvek pro strom `widgetů`. Nazývají se též `widget`, aneb všechno je `widget`, takže čtení jakéhokoliv textu včetně této práce může být velice matoucí. Tyto dva základní stavební `widgety` 2.7 se nazývají **StatelessWidget** a **StatefulWidget**. První je neměnitelný („immutable“), tedy všechny jeho proměnné by měly být finální. Druhý `widget` se stará o svůj stav po celou dobu své existence včetně rotace obrazovky apod. Při změně proměnných je třeba zavolat funkci `setState()`, která zapříčiní překreslení celého tohoto `widgetu`.

2.4.6 UI – Vykreslování

Vykreslování ve Flutteru probíhá skrze čtyři fáze:

1. Layout
2. Paint
3. Composite
4. Rasterize

Velice podobné fáze se vyskytují například v moderních prohlížečích.



Obrázek 2.8. Průchod fázemi vykreslování ve Flutteru (tzv. „Rendering Pipeline“) [20]

V první fázi, **layout**, se přesně vypočítá, jakou má mít každý prvek velikost a kde má být umístěn. Tento výpočet se provádí „dvěma“ průchody. Nejedná se o stejné dva průchody jako u `RelativeLayoutu` z Android SDK, nýbrž zdánlivě o kombinaci pre-order a post-order prohledávání do hloubky (DFS). Každý prvek prvně spočítá svá omezení (například svou maximální šířku), která předá svým potomkům. Potomci vypočítají svá omezení a proces se opakuje až k listům stromu. Listové prvky si poté

určí svou velikost a pozici tak, aby neporušily předaná omezení. Tuto velikost a pozici pošlou zpět rodiči. Rodič provede to samé a proces se opakuje až ke kořenu stromu.

V **paint** fázi se zjišťuje, jak má který prvek vypadat. Rodič dostane canvas, na který vykreslí sebe sama. Poté předá DFS průchodem potomkům stejný canvas a svůj offset. Proces vykreslování sebe sama dle offsetu a posílání předávání canvasu se opakuje. Je důležité zmínit, že na canvas se dá vykreslovat do vrstev. Zároveň proces vykreslování na canvas není viditelná změna obrazovky, ale pouze sběr příkazů pro kreslení.

Composite fáze zajistí, aby všechny vrstvy různých prvků se správně překrývaly a vše spojí do jedné scény. Tuto scénu pak předá *GPU vláknu*.

Rasterize fáze probíhá na *GPU vláknu* narozdíl od předchozích fází. Jedná se o proces konečného vykreslování prvků na obrazovce skrze komunikaci tohoto vlákna s GPU.

Kromě zmíněných fází lze ještě dohledat fáze **User Input**, **Animation** a **Build**. První a druhá fáze jsou totožné s fází v Android SDK a třetí, `Build`, fáze se stará o vytvoření stromu prvků, které mají být vykresleny na obrazovce.
[19–20]

Kapitola 3
Návrh

Aby se daly vytvořit návrhy pro změření výkonnosti, je zapotřebí nejdříve vědět, co to výkonnost je. Nejobvyklejší vysvětlení je většinou podíl práce a doby, za kterou byla daná práce vykonána. To však není výkonnost, jak ji rozumí vývojáři aplikací pro mobilní systémy. Hodnoty, které je třeba sledovat jsou mj.:

- Využití paměti RAM
- Využití CPU/GPU
- Čas prvního/dalšího spuštění aplikace
- Spotřeba baterie
- Ovlivňování jinými aplikacemi běžících paralelně
- Rychlost vykreslování
- I/O operace
- Tzv. jank výskyty
- Čas parsování odpovědí ze serveru a dalších `X` síťových záležitostí

Výkonnost aplikací často odkazuje na čistou rychlost a jak ovlivňuje plynulost aplikace. Měřítko je to spolehlivé a přímo i nepřímo ovlivňuje řadu výše zmíněných hodnot. To je důvodem, proč se vyskytuje v mnoha měřeních této práce.

Mnohé z výše uvedených bodů je zakomponováno do následujících návrhů.

3.1 Fibonacciho posloupnost

Pro začátek je dobré si udělat přehled o čisté rychlosti provedení nějakého algoritmu. Není nic jednoduššího než naivní implementace Fibonacciho posloupnosti. Je třeba dát si pozor, aby šlo o opravdu naivní implementaci, protože v opačném případě by mohly některé kompilátory provést například optimalizaci koncové rekurze a porovnání by pak nebylo ekvivalentního kódu.

Následuje rekurzivní verze Fibonacciho algoritmu v pseudokódu:

```
1  function fib(n)
2      if n <= 0 then ret 0
3      if n = 1 then ret 1
4
5      ret fib(n - 1) + fib(n - 2)
```

3.2 Vykreslování UI

Vykreslování obrazců na obrazovce a tím vytvářet UI je nejzákladnější funkce, kterou musí každý použitelný framework umět. Nejde ovšem o nic snadného, jak bylo zmíněno v kapitole 2.2.3. Nejdůležitější pro dobrý uživatelský dojem z aplikace je, když uživatel nezaznamená jediný jank. V následujících podkapitolách je nastíněno několik typů layoutu, které přiblíží výkonnostní problematiku.

3.2.1 Layout view

Pro začátek byl připraven nejjednodušší layout možný – 3 prvky pod sebou, nikterak pozicovaný kromě výchozího zarovnání doleva nahoru.

K tomu ještě jeden obyčejný layout se dvěma prvky, kde jeden je zarovnán k levému okraji a druhý k pravému. Oba zarovnány k hornímu okraji.

Třetím layoutem je jediný prvek vertikálně a horizontálně umístěný na střed.

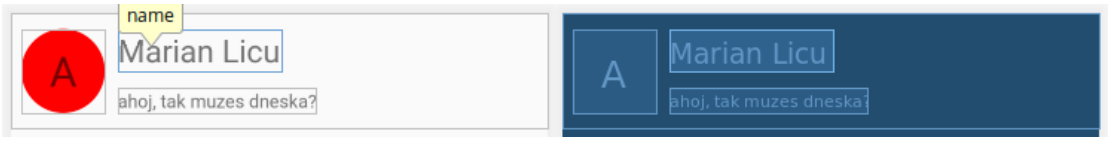
Poslední layout je tím pravým testem. Předchozí návrhy jsou spíše pro ověření, že vše funguje tak, jak je předpokládáno a nevyskytují se nepředvídatelné odchylky. Tento návrh však má již komplexní umístění prvků jak pozičně k okrajům obrazovky, tak k ostatním prvkům.

Celý obsah je zarovnán vertikálně na střed a ze stran oplývá malou mezerou. Doleva je umístěn obrázek sloužící jako kotva zbylých prvků. K hornímu okraji obrázku je zarovnán titulek, který je zároveň horizontálně umístěn na střed mezi obrázkem a datem. Datum je umístěno doprava se zarovnaným spodním účařím[1] k titulku. Ke spodnímu okraji obrázku je zarovnána skupina dvou tlačítek. Ta jsou od sebe vzdálená menší (pevnou) mezerou a dohromady jsou zarovnány na střed mezi obrázkem a okrajem obrazovky.

3.2.2 Posuvný seznam prvků

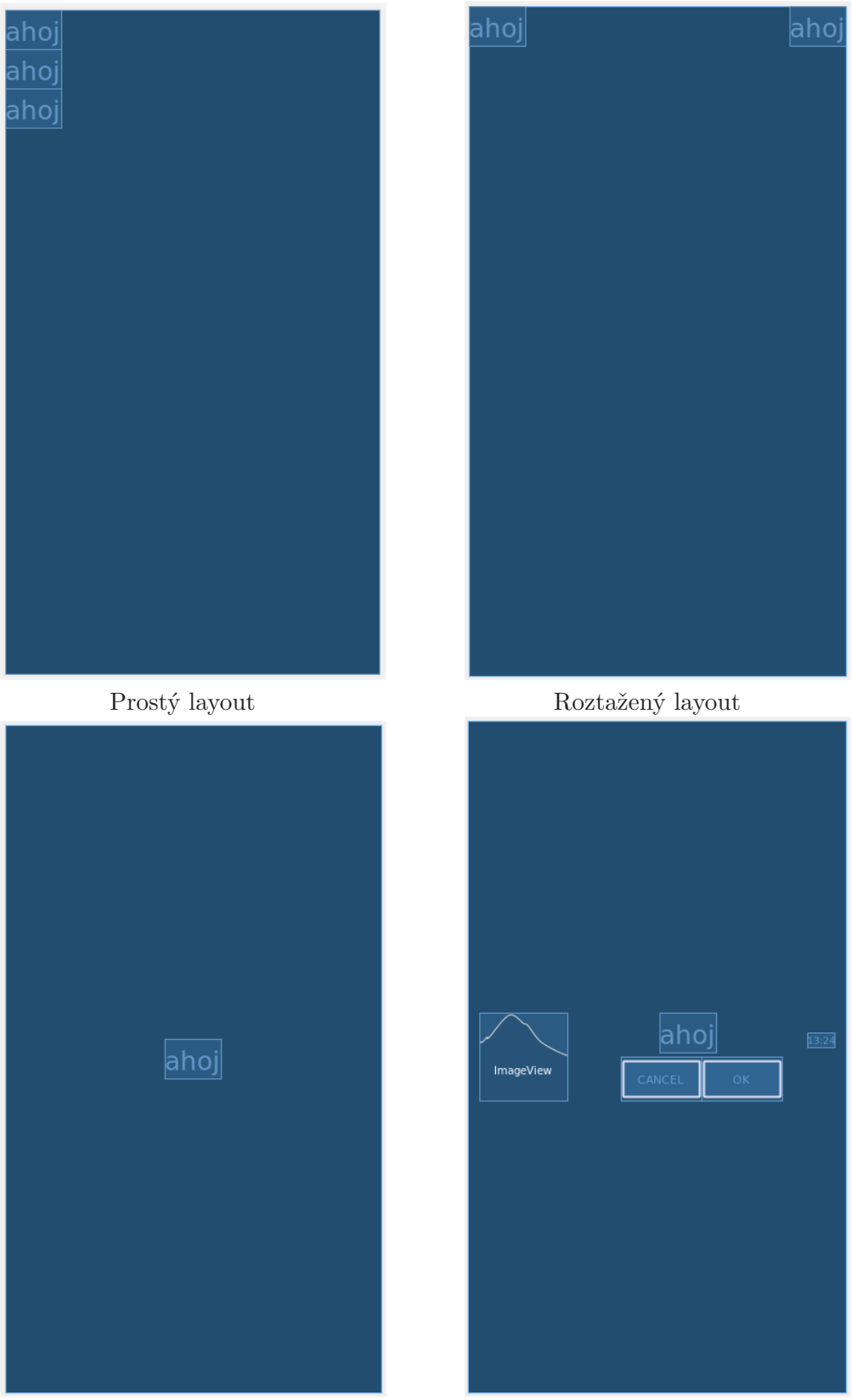
Kromě měření vykreslení jednoho statického `layoutu` je záhodno též vyzkoušet tzv. „scrollování“. Pod Android SDK existoval vždy jeden problematický případ a tím je seznam prvků, který se dá posouvat.

Návrhem tedy budiž jednoduchý posuvný seznam stejného prvku, kterým je barevné kolečko s textem umístěné vlevo a napravo od něj dva texty. Půjde tedy o jakýsi seznam konverzací, ale s náhodně vygenerovanými texty a barvou.



Obrázek 3.2. Návrh konverzace pro posuvný seznam prvků

[1] Písmová osa



Prostý layout | Roztažený layout

Vycentrovaný layout | Komplexní layout

Obrázek 3.1. Hrubé návrhy pro rozložení layoutů

3.3 Zpracování dat

I když Fibonacciho algoritmus (kapitola 3.1) prozradí čistou rychlost, tak se nejedná o kód vyskytující se v běžných aplikací. Z tohoto důvodu je zde ještě jeden test podobného rázu, ale s větší šancí na implementaci v běžných aplikacích.

Prvně se provede I/O operace otevření souboru a jeho přečtení. Pro jednoduchost je použit soubor s daty oddělenými řádky a středníky; jeden řádek obsahuje datum a změřenou hodnotu teploty na Celsiově stupnici.

```
1778;7;8;20.3       // 20.3 °C dne 08. 07. 1778
1831;12;29;-8.7     // -8.7 °C dne 29. 12. 1831
2007;11;15;1.0      //  1.0 °C dne 15. 11. 2007
...
```

Po přečtení se řádek rozdělí podle středníků a vytvoří objekt s 3 celými čísly pro datum a jedním reálným číslem pro teplotu. Objekt se uloží do pole dynamické velikosti.

```
1  struct Record {
2      int year, month, day
3      double temperature
4  }
5
6  Array<Record> records = new Array()
7  Array<String> lines = File("/path/to/weather/file").readLines()
8
9  for (int i = 0; i < lines.size; ++i)
10     Array<String> data = lines[i].split(';')
11     records.add(new Record(data[0], data[1], data[2], data[3]))
```

Následují transformační operace.

1. Všem objektům se změní hodnota teploty ve stupních Celsia na stupně Fahrenheita
2. Objekty se seskupí dle unikátního měsíce a roku do hash tabulky
3. Všechna pole, ukrývající se v hodnotových částech hash tabulky, se seřadí dle teploty

```
12 HashMap<Int, Array<Record>> map = new HashMap()
13
14 for (int i = 0; i < records.size; ++i)
15     records[i] = records[i].copy(temperature = temperature * 1.8 + 32)
16
17 for (int i = 0; i < records.size; ++i)
18     Record record = records[i]
19
20     if (map[unique(record.year, record.month)] = null)
21         map[unique(record.year, record.month)] = new Array()
22
23     map[unique(record.year, record.month)].add(record)
24
25 for (int i = 0; i < map.size; ++i) map[i].sort()
```

Poslední částí je opět I/O operace, tentokrát uložení do relační databáze. Relační databáze obsahuje tabulku s následujícími sloupci:

Název sloupce	Typ hodnot sloupce
id	Primární klíč
year	Celé číslo
month	Celé číslo
lowestTemperature	Reálné číslo
highestTemperature	Reálné číslo
measureCount	Celé číslo

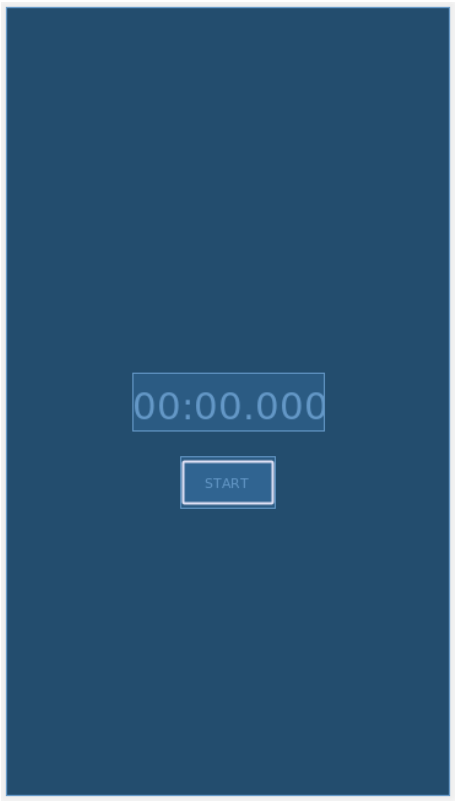
Tabulka 3.1. Návrh tabulky počasí pro test zpracování dat

Do této tabulky se transakčně uloží všechny údaje z hash tabulky a to v podobě unikátního měsíce, jeho nejnižší a nejvyšší teplota a nakonec kolikrát se v daném měsíci teplota změřila.

```
26  db.transaction
27      for (int i = 0; i < map.size; ++i)
28          db.insert(TABLE_NAME,
29              map[i].first().year,
30              map[i].first().month,
31              map[i].last().temperature,
32              map[i].first().temperature,
33              map[i].count()
34          )
```

3.4 Stopky

Posledním testem jsou jednoduché stopky. Stopky mají kromě tlačítka (re)start ještě 3 části: minuty, sekundy a milisekundy. Právě poslední část je překreslována velice často, což se bude hodit pro porovnání vytížení CPU a využití paměti RAM – faktory přímo ovlivňující spotřebu baterie.



Obrázek 3.3. Hrubý návrh obrazovky pro stopky

```
1  function start()
2      startTime = Time.now()                    // milisekundy
3      while true do
4          sleep 30                              // milisekundy
5          long diff = Time.now() - startTime
6          int ms = diff mod 1000
7          int sec = (diff / 1000) mod 60
8          int min = diff / 1000 / 60
9
10         clock.text = "$min:$sec.$ms"
```

Kapitola 4
Implementace

4.1 Android SDK

Výchozí instalace programu Android Studio podporuje vývoj pomocí Android SDK bez jakéhokoliv ladění nebo dodatečných instalací pluginů.

4.1.1 Struktura

Z pohledu implementace jsou ve struktuře projektu hlavní tyto části:

a) Složka **/src/main/java**
b) Složka **/src/main/res**
c) Složka **/src/main/assets**
d) Soubor **/src/main/AndroidManifest.xml**
e) Soubor **/src/build.gradle**

První bod, složka **/src/main/java**, obsahuje téměř veškerou logiku aplikace čili zdrojové soubory. Kromě pomocných tříd lze zde definovat dva hlavní stavební komponenty aplikace – `Activity` (aktivita) a `Fragment`.

`Activity` je základní komponenta reprezentující jednu obrazovku s UI. Běžně programy obsahují vstupní bod v podobě funkce `main()`, což ale není stejný případ v Androidu. Namísto toho je zde aktivita, která obsahuje metody životního cyklu jako `onCreate`, `onStart`, `onResume` a `onStop`. Skrze tyto metody se ovládá logika aplikace a reaguje na jiné okolnosti typu minimalizace aplikace do pozadí apod. V tomto projektu existuje hlavní aktivita zvaná `MainActivity.kt` a následně menší aktivity pro každý test. Hlavní aktivita obsahuje zjištění verze SQLite databáze, vytvoření struktury databáze a logiku pro navigaci mezi testy.

Fragment je část aktivity, která zajišťuje lepší modulární design. Lze říci, že se jedná o „sub-aktivitu“ s vlastním `layout`em a životním cyklem. Obsahuje podobné řídicí metody jako aktivita. V tomto projektu nebyl použit žádný fragment.

> *Uvedená složka* **/src/main/java** *neobsahuje zdrojové soubory přímo. Je třeba se zanořit do několika vnořených složek pojmenovaných podle unikátního identifikátoru aplikace. Například hlavní aktivita* `MainActivity` *je umístěna v* `/src/main/java/eu/lepicekmichal/android_sdk_ndk/MainActivity.kt`.

Složka **/src/main/res** obsahuje veškeré aplikační prostředky – **resources**. Ku příkladu bitmapy, překladové proměnné, `layouty`, barvy, styly apod. Většina těchto prostředků jsou psané ve značkovacím jazyce XML. Tyto prostředky se následně dynamicky načítají ze zdrojového kódu. V tomto projektu jsou nutně využity pouze `layouty` a dvě bitmapy.

Složka **/src/main/assets** je výchozí složkou pro umístění libovolných datových souborů, které aplikace využívá. Složka je srovnatelná se složkou **raw** v `resources` s rozdílem, že

pro soubory v `assets` není vygenerováno dynamické ID. V tomto projektu se využívá složka pouze pro datový soubor s naměřenými teplotami v Praze [31].

Soubor **/src/main/AndroidManifest.xml** mimo jiné slouží pro registraci všech aktivit.

Soubor **/src/build.gradle** slouží pro nastavení kompilovacího procesu, zajištění závislostí, defaultních parametrů aplikace (např.: minimální podporovaná verze systému) apod.

4.1.2 XML layout

Pro vytváření UI se primárně využívá značkovací jazyk XML. Díky tomu je alespoň částečně odděleno vytváření UI od logiky aplikace. Tyto `layouty` jsou uloženy v `resources` a jsou načítány dynamicky ze zdrojového kódu. Načítat je lze libovolně, ale prakticky se hlavně využívá funkce `setContentView()` v `onCreate()` aktivity a `LayoutInflater.inflate()` v `onCreateView()` fragmentu; oba případy udělají syntaktický rozbor XML a vytvoří strom objektů (UI komponent) pro UI dané aktivity respektive fragmentu.

4.2 Android NDK

Pro nerušený vývoj s C/C++ je potřeba do programu Android Studio nainstalovat tři komponenty. NDK samotný, CMake a LLDB (ladící nástroj pro C/C++).

Existuje druhá možnost, kdy nejsou potřeba poslední dvě komponenty. Jedná se o samostatný vývoj a kompilaci C/C++ mimo Android Studio a poté pouze přesunutí vygenerovaných dynamických knihoven na správné místo v projektu. Pro tuto možnost je ale potřeba speciální skript **ndk-build**, který zajistí použitelnou kompilaci pro Android.

V této práci je využita pouze první možnost.

4.2.1 Struktura

Z pohledu vývoje přináší podpora NDK pouze jedinou složku – **/src/main/cpp**. Do této složky se umisťují soubory se zdrojovým kódem a soubor `CMakeLists.txt` obsahující direktivy a instrukce popisující zdrojové soubory projektu a cíl kompilace (spustitelný soubor, knihovna nebo obojí).

V tomto projektu jsou zde umístěny jak vlastní soubory se zdrojovým kódem, tak SQLite knihovna (podrobněji v kapitole 7.5.2).

4.2.2 JNI komunikace

Aby spolu mohly strany Kotlin a C/C++ komunikovat, tak je zapotřebí nějakého prostředníka. Tím je JNI. Implementační požadavky tohoto rozhraní a celkové zajištění bezproblémové komunikace jsou rozebrány v následujících odstavcích.

Statické načtení dynamické knihovny, pomocí útržku kódu níže. Na místo zástupného slova *XXX* patří název knihovny, tak jak byla pojmenována v kompilačních instrukcích; není tím myšlen název souboru i když je zpravidla shodný.

```
1  companion object {
2      init {
3          System.loadLibrary("XXX")
4      }
5  }
```

Dalším implementačním detailem je definování externí funkce ve stejném souboru, ve kterém se načítá dynamická knihovna.

```
6  external fun ahoj(): String
```

Nejzajímavější požadavky se týkají volané funkce v C/C++. Prvně musí mít specificky daný název, který se sestavuje následně:

1. Vždy se začíná prefixem **Java_**
2. **Cesta** k souboru relativně k `/src/java/`[1]
3. **Název souboru**, kde je definována externí funkce (bez přípony souboru)
4. **Název funkce**

Definovaným parametrům musí předcházet parametr `JNIEnv* env`, což je ukazatel na VM a parametr `jobject thiz`, což je ukazatel na implicitní `this` objekt ze strany Kotlinu/Javy.

Třetím požadavkem je přidání direktivy `JNIEXPORT` před návratový datový typ a direktivy `JNICALL` před název funkce.

Nejdůležitějším požadavkem však je, obalení funkce blokem **extern** `C` . To zajistí, aby C++ kompilátor nezměnil název funkce při kompilaci, což je základní funkcionalita nutná pro přetěžování funkcí.

Výsledná volaná funkce v C/C++ vypadá například takto:

```
1  extern "C" JNIEXPORT jstring JNICALL
2  Java_eu_lepicek_dip_slozka_MainActivity_ahoj(JNIEnv *env, jobject thiz) {
3      ...
4  }
```

4.2.3 Kompilace

V souboru `CMakeLists.txt` se:

- Nastavuje minimální verze CMake – `cmake_minimum_required()`
- Přidávají knihovny, které se mají vygenerovat – `add_library()`
- Nalézají předkompilované knihovny – `find_library()`
- Specifikuje, které knihovny se mají slinkovat – `target_link_libraries()`

V tomto projektu je třeba zkompilovat pouze jednu dynamickou knihovnu z vlastních zdrojových souborů a SQLite. Taktéž se nalezne `logovací` knihovna, která umožňuje vypisování zpráv do konzole. Ve výsledku se nakonec slinkují pouze dvě knihovny.

Pro funkčnost je ale zapotřebí ještě upravit soubor **/src/build.gradle**. Je zde třeba definovat cestu k `CMakeLists.txt` a dobrovolně parametry pro příkaz kompilace.

```
1  defaultConfig {
2      externalNativeBuild {
3          cmake {
4              path "src/main/cpp/CMakeLists.txt"
5          }
6      }
7  }
```

[1] Lomítka (`/`) v cestě jsou nahrazena podtržítkem (`_`)

```
13  externalNativeBuild {
14      cmake {
15          cppFlags "-O3"
16          arguments "-DCMAKE_BUILD_TYPE=Release"
17      }
18  }
```

Ve druhém úryvku je definována 3. úroveň optimalizace, což je považováno za standard a vyrovnávají se tím optimalizace Kotlinu/Javy, které nelze nastavit. Poté je definován také argument, který určuje verzi buildu. Ten je nastaven na „release“, protože jinak kompilace nevygeneruje nejefektivnější kód, kvůli možnému ladění. Argument to není nutný, jelikož stačí pro stejný efekt vytvořit „release“ verzi místo „debug“ verze.

4.3 Flutter SDK

Android Studio nepodporuje Flutter a Dart out-of-box. Je třeba doinstalovat plugin zvaný příhodně *Flutter*. Nutné je též stáhnutí a instalace Flutter SDK, což nelze provést skrze Android Studio.

4.3.1 Struktura

Z pohledu implementace jsou ve struktuře projektu hlavní tyto části:

a) Složka **/android**
b) Složka **/ios**
c) Složka **/lib**
d) Soubor **/pubspec.yaml**

První složka, **/android**, obsahuje klasickou strukturu a soubory nativního vývoje v Kotlinu/Javě pro Android. Zde lze implementovat funkcionality, které zatím nejsou možné ve Flutteru a poté je z Flutteru volat/využívat. Pro tuto práci není tato možnost potřebná.

Druhá složka, **/ios**, není součástí této práce, ale zřejmě se jedná o stejný princip jako první bod; pouze pro platformu iOS.

Třetí složka, **/lib**, je určená pro zdrojové soubory psané v Dartu, které popisují UI a logiku aplikace. Zde lze nalézt všechny zdrojové kódy této práce, které se týkají části Flutter.

Posledním bodem je soubor **/pubspec.yaml**, který plní podobnou úlohu jako **/src/build.gradle** v Android SDK. V tomto projektu se zde nastavují zejména závislosti a cesta k `assets`.

Kapitola 5
Implementace - Fibonacci

V matematice, Fibonacciho čísla či Fibonacciho posloupnost, jsou taková čísla, která jsou součtem dvou předchozích. Posloupnost začíná čísly 0 a 1.

$$F_0 = 0, F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

5.1 Kotlin

Pro obyčejné měření rychlosti provedení kódu není třeba dodržovat jakékoliv principy správného programování pro systém Android. Stačí tedy nejjednodušší vytvoření vedlejšího vlákna za pomocí funkce `Thread()` a jeho spuštění. Tento kód se provádí v aktivitě `/tests/fibonacci/FibonacciTest.kt`.

V uvedeném souboru lze najít metody nutné pro běžný chod aplikace a jedna z nich, `onStart()`, obsahuje spuštění zmíněného vedlejšího vlákna. Uvnitř tohoto vlákna je volání funkce `fib(n: Int)`, která je právě rekurzivní implementací výpočtu n-tého čísla ve Fibonacciho posloupnosti. Toto volání je ještě zabaleno do čas měřící funkce a cyklu pro výpočet vícera čísel najednou.

5.2 C/C++

5.2.1 Kód algoritmu

Samotný algoritmus sestává pouze z rekurzivní funkce výpočtu n-tého čísla Fibonacciho posloupnosti a jedné funkce navíc, která tuto funkci bude volat. Zde není nad čím se pozastavit kromě speciálního datového typu v těchto funkcích – `jint`. Aby spolu mohly jazyky Java/Kotlin a C/C++ komunikovat, potřebují společné datové typy. Jedním z nich je právě `jint`, který je v tomto případě naprosto stejný jako `int` v C/C++; není tedy nutné zbytečně alokovat další proměnnou jen pro překopírování hodnoty.

Celý tento kód lze najít v souboru `fibonacci.cpp`.

5.2.2 Spuštění kódu

Spuštění kódu musí být provedeno ze strany Kotlinu, ať už ihned po spuštění aplikace nebo kdykoliv později. Poté, co se definují všechny důležité prvky pro bezproblémovou komunikaci (viz kapitola 4.2.2), je možné zavolat funkci implementovanou v jazyce C/C++.

Zde nastává rozhodnutí, kam vložit cyklus pro spuštění `x` výpočtů Fibonacciho posloupnosti. Na stranu Kotlinu nebo na stranu C/C++. V prvním případě je do výsledku měření zahrnut čas nutný pro komunikaci mezi těmito dvěma jazyky, tedy v druhém

případě jsou pravděpodobně na dosah vyšší rychlosti. Byl zvolen první přístup, jelikož dává čas, který se bude hojněji vyskytovat v reálné aplikaci.

Spuštění kódu ze strany Kotlinu a definici nativní funkce lze najít ve stejném souboru jako spuštění Kotlinovské Fibonacciho funkce, `/tests/fibonacci/FibonacciTest.kt`.

5.3 Dart

Pro výpočet n-tého čísla Fibonacciho posloupnosti byl vytvořen State[1]<FibonacciTest> `FibonacciTestState`. Uvnitř tohoto stavu je `build()` metoda, která vytvoří UI dané obrazovky – točící se kolečko pro stav pracování a schování jakmile test proběhne. Toto se zařizuje přes prvek `FutureBuilder`.

> *FutureBuilder je prvek, který má v konstruktoru dva hlavní parametry – future a builder. Do parametru builder se předává funkce přijímající snapshot neboli data vypočtená asynchronně a v dané funkci se vytváří UI podle toho, jaká data se ve snapshotu objeví. Do parametru future se předává asynchronní funkce, která provádí práci na jiném vlákně a může trvat delší dobu.*

Prvku `FutureBuilder` je předáno vytvoření UI a `compute()` funkce pro spuštění testu. Uvnitř funkce pro spuštění testu lze nalézt opět pouze cyklus a volání rekurzivní funkce pro výpočet n-tého čísla Fibonacciho posloupnosti. Obě tyto funkce musí být definovány staticky nebo jako top-level funkce, toť požadavek Isolate[2] v Dartu.

> *Funkce compute() v jazyce Dart je pomocná funkce pro vytvoření Isolate s nejjednodušším případem asynchronního výpočtu – pouze jeden vstup a jediný výstup. Přebírá dva parametry, funkci, která se má provést na jiném vlákně a hodnotu vstupu.*

[1] Prvek reprezentující stav widgetu, může se měnit a slouží k perzistenci stavu widgetu
[2] Nezávislý worker podobný vláknům, ale bez sdílené paměti – komunikace je možná pouze skrze zprávy

Kapitola 6
Implementace - Vykreslování

6.1 Java (Kotlin)

Přestože je v této práci porovnáván jazyk Kotlin, ne Java, tak většina zdrojového kódu Android SDK je stále napsaná v Javě. Včetně funkcí starajících se o vykreslování. Odsud název této kapitoly a dalších.

6.1.1 Layout view

FrameLayout. Běžným použitím lze implementovat pouze roztažený a vycentrovaný `layout`. 3.1

Layout soubory:

- `activity_advanced_test_f.xml`
- `activity_center_test_f.xml`

S mnoha vnořenými **LinearLayout**y lze implementovat všechny navržené layouty včetně komplexního.

Layout soubory:

- `activity_simple_test_l.xml`
- `activity_advanced_test_l.xml`
- `activity_center_test_l.xml`
- `activity_complex_test_l.xml`

S **RelativeLayout**em lze bezproblémově implementovat všechny navržené layouty, kromě komplexního, ve kterém je zapotřebí jedno vnoření.

Layout soubory:

- `activity_simple_test_r.xml`
- `activity_advanced_test_r.xml`
- `activity_center_test_r.xml`
- `activity_complex_test_r.xml`

Jak bylo již zmíněno, **ConstraintLayout** byl vytvořen pro návrh jakkoliv komplikované obrazovky a tak není překvapením, že lze všechny testovací návrhy rozvržení v tomto `layout view` implementovat.

Layout soubory:

- `activity_simple_test_c.xml`
- `activity_advanced_test_c.xml`
- `activity_center_test_c.xml`
- `activity_complex_test_c.xml`

6.1.2 Posuvný seznam prvků

Implementace posuvného seznamu prvků v Kotlinu primárně zajišťuje **RecyclerView**. Ten je umístěn v souboru `activity_scroll_test.xml`. Jeho použití, ke kterému patří nastavení `LayoutManager`a a `Adapter`u, lze pak najít v souboru `/tests/scroll/ScrollTest.kt`.

`LayoutManager` je nutný pouze pro určení typu seznamu – lineární, mřížkový, vertikální, horizontální, počet prvků na řádku apod. Zvolen je výchozí lineární a vertikální typ.

Oproti tomu `Adapter` má mnohem důležitější roli. V `Adapter`u se drží prvky samotné, inflatování `layotu`, který je použit pro vykreslení položek a zařizuje se zde též jejich recyklace. Recyklace je úkon, který pomáhá zmenšit čas vykreslení pro položky, které již na obrazovce byly, poté se dostaly mimo obrazovku a opět zpět na obrazovku.

Celkově je tedy potřeba implementovat pět různých částí včetně `data class` objektu a `layout`u jednotlivé položky – konverzace (`/tests/scroll/Conversation.kt`, `item_conversation.xml`). Každá konverzace má část s kruhovým pozadím, což si vyžaduje ještě jeden soubor `conversation_circle_bg.xml` ve složce *drawable*.

6.2 Dart

V souboru `rendering.dart` jsou definovány stateless widgety `SimpleTest`, `CenterTest`, `AdvancedTest`, `ComplexTest`. Všechny obsahují co nejjednodušší UI widgety pro dosažení cílového rozvržení a zároveň, aby se dalo usoudit, že jsou XML i Dart `layouty` ekvivalentní.

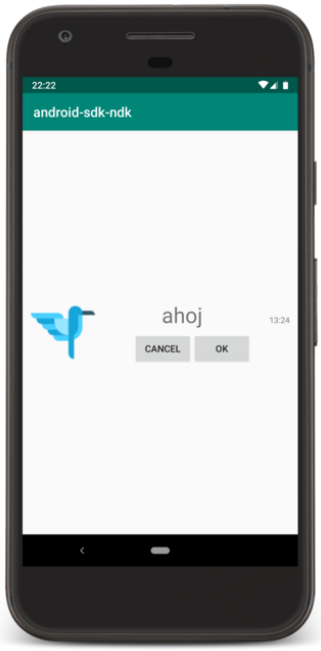
6.2.1 Posuvný seznam prvků

Oproti Kotlinu/XML je implementace v Dartu mnohem přímočařejší. Využije se `widget` zvaný **ListView**, kterému se předá počet prvků (pokud je fixní) a funkce, která vrátí vytvoření jedné položky dle předaného indexu. Veškerou implementaci tak lze najít v jediném souboru `scroll.dart`.

Prostý layout

Roztažený layout

Vycentrovaný layout



Komplexní layout

Obrázek 6.1. Výsledné podoby obrazovky pro měření vykreslování

Kapitola 7
Implementace - Zpracování dat

7.1 Čtení ze souboru

V průběhu implementace se naskytl menší problém. Předpřipravené funkce pro čtení souboru v Dartu (např.: `File.readAsLines()`) nepracují s bufferem, ale načítají rovnou celý soubor. Vzhledem k tomu, že výchozí funkce Kotlinu i C++ buffer používají, tak bylo zapotřebí hledat možnost podobné implementace. Dart nabízí alternativu v podobě streamu, která má pevně daný buffer o velikosti 64 KiB. To je značně omezující, ale ve zbylých porovnávaných jazycích si lze buffery upravit na stejnou velikost.

7.1.1 Kotlin

V Kotlinu jde o přímočaré vytvoření objektu `File` a z toho získání objektu `BufferedReader`, kterému se nastaví velikost bufferu. Nakonec se zavolá funkce `readLines()`, která rozdělí data na řádky.

Velikost bufferu se určí parametrem funkce pro získání `BufferedReader`u.

```
File(...).bufferedReader(bufferSize = 64 * 1024)
```

Přečtená data jsou uložena do `data class` objektu v dynamickém poli `ArrayList`.

7.1.2 C/C++

Ač otevření a čtení souboru v C++ není nikterak složité, tak úprava bufferu už vyžaduje jistou znalost. Prvně je třeba si definovat `ifstream` objekt pro čtení souboru. Následně se musí vytvořit objekt použitelný jako buffer, toť objekt typu `char*`. Poslední operací před čtením souboru je nastavení tohoto bufferu do `ifstream`u – to se provádí následujícím úryvkem kódu.

```
infile.rdbuf()->pubsetbuf(buf, sizeof buf); //buf = new char[64 * 1024]
```

Funkce `rdbuf()` vrací ukazatel na vnitřní buffer a funkce `pubsetbuf()` nahradí výchozí buffer vlastním.

Přečtená data jsou uložena do `struct` objektu v dynamickém poli `std::vector`.

7.1.3 Dart

Jak již bylo zmíněno, implementace v Dartu nebyla přímočará. Nicméně, kromě problémů s bufferem se dostalo i na podivné rychlostní chování alternativních implementací. Dart je ze své podstaty navržen pro reaktivní a asynchronní chování, ale pro korektní chování bylo zde potřeba počkat na přečtení celého souboru a získání všech dat do pole. Stream samotný je asynchronní a trvá konstantní dobu, ale vyčkání na výsledek za pomocí sekvence `await` je různé.

Následně uvádím dvě varianty, kde první prodlouží celý čas funkcionality na dvojnásobek, kdežto druhá si zachovává podobný čas jako alternativní implementace s přečtením celého souboru naráz. Vzhledem k tomu, že simulace podobné situace v Kotlinu i C/C++ vykazuje taktéž podobné časy, tak jsem se rozhodl pro zachování druhého řešení.

```
Stream<String> lines =
    stream.transform(const LineSplitter());
await for (String line in lines) { ... }
```

```
List<String> lines =
    await stream.transform(const LineSplitter()).toList();
for (String line in lines) { ... }
```

Přečtená data jsou uložena do `class` objektu v dynamickém poli `List`.

7.2 Transformace dat – Hluboká kopie

Tento úkon má za cíl převést tepelné hodnoty z Celsia na Fahrenheity. Ač by pro vykonání tohoto úkonu stačilo mít proměnnou teploty měnitelnou (mutable), tak se to neshoduje s moderním přístupem programování, kdy se preferuje mít vše neměnitelné (immutable). Taktéž bude mnohem zajímavější sledovat chování algoritmu s hlubokou kopií objektu než s pouhou změnou jedné členské proměnné.

Je tedy třeba provést projetí celého pole záznamů teploty a každý záznam nahradit jeho hlubokou kopií s upravenou teplotou.

7.2.1 Kotlin

Kotlin `ArrayList` obsahuje funkci `replaceAll()`, která by jednoduše pomohla nahradit všechny objekty v poli záznamů. Nicméně, implementace této funkce obsahuje režii navíc pro kontrolu chyb při více-vláknovém programování. Nezbývá tedy nic jiného, než funkci duplikovat do podoby bez této režie.

Další překážkou je Kotlin funkce `copy()`, kterou má každý `data class` objekt. Funkce snadno vytváří kopii, ale ne hlubokou. Naštěstí jsou všechny členské proměnné neměnné a jednoduchého typu, což zajistí v tomto případě kopii hlubokou. Kotlin sice neposkytuje vývojáři možnost rozhodnout se mezi jednoduchým typem (např.: `int`) a výchozím složeným typem (např.: `Integer`), ale při dekompilaci bajtkódu lze zjistit, že v tomto jednoduchém použití se kompilátor rozhodl pro nahrazení složeného typu typem jednoduchým.

Poslední menší překážkou je absence klasického `for` cyklu v Kotlinu. Kotlin obsahuje pouze `range-based for` cykly, které vyžadují nějakou formu iterovatelného pole. Ač by vytvoření tohoto pole výsledky porovnání nijak zvlášť neovlivnilo, byl pro jistotu použit ekvivalentní `while` cyklus.

Implementaci lze nalézt ve funkci `transform()` v souboru `/tests/data/DataTest.kt`.

7.2.2 C/C++

Na rozdíl od Kotlinu, C `struct` objekty explicitně neobsahují funkci pro kopírování objektu se změnou hodnot členských proměnných. Nelze tedy využít kopírovacího konstruktoru. Zároveň C/C++ neobsahuje pojmenované parametry a tak nezbývá, než

implementovat vlastní `copy()` funkci pro kopii objektu se změnou pouze jedné členské proměnné – teploty.
Využitím přetěžování se lze dostat na následující implementaci kopírovací funkcionality:

```
record copy(double temperature) {
    return copy(this->year, this->month, this->day, temperature);
}

record copy(int year, int month, int day, double temperature) {
    return {year, month, day, temperature};
}
```

Implementaci lze nalézt ve funkci `transform()` v souboru `data.cpp`.

7.2.3 Dart

Dart podporuje pojmenované parametry a stačí tedy doplnit záznamový objekt o jednu `copy()` funkci, která vytvoří nový objekt s upravenou teplotou. Pojmenované argumenty, které nebyly při volání funkce specifikovány, obsahují `null` a lze tedy využít pouze konstruktoru a jednoho `null-aware` operátoru.

Ekvivalent jednoho takového operátoru je následující:

```
‘x ?? 3‘ je ekvivalentní k ternárnímu operátoru ‘x = null ? x : 3‘
```

```
Record copy({int year, int month, int day, double temperature}) =>
    Record(
        year ?? this.year,
        month ?? this.month,
        day ?? this.day,
        temperature ?? this.temperature
    );
```

Implementaci lze nalézt ve funkci `transform()` v souboru `data.dart`.

7.3 Transformace dat – Seskupení

Účelem této části kódu je rozřazení všech tepelných záznamů do skupin podle unikátního měsíce. Navrženo je využití hash tabulky s klíčem unikátním pro každý měsíc a s hodnotou dynamické pole teplotních záznamů. Tento návrh dovolí vyhledání existence klíče i vložení páru klíč-hodnota v průměrné konstantní složitosti – $O(1)$. A jelikož se dle návrhu nebude přidávat více hodnot pod jeden klíč, tak se z této průměrné složitosti stává složitost v nejhorším možném případě.

Aby byl každý měsíc unikátní, je třeba vymyslet unikátní sestavení klíče. První naskytnutou možností je obyčejné datum ve `Stringu`, ale v tom případě by existovala zbytečná režie pro spojení `Stringů` a kopírování složeného datového typu jakým je `String`. Lze tedy vymyslet jednoduchou matematickou operaci – $rok * 100 + měsíc$. Vynásobením roku 100 se zajistí, že nevzniknou duplicity typu rok 2017, 2. měsíc a rok 2018, 1. měsíc.

7.3.1 Kotlin

Datové typy dědící od `MutableMap` v Kotlinu nabízí funkci `getOrPut()`, která zajistí vrácení hodnoty dynamického pole pro zadaný klíč. Pokud ještě klíč v `map`ě neexistuje, tak se vytvoří zadaná výchozí hodnota, kterou je nové prázdné dynamické pole.

Ve zdrojovém kódu funkce `getOrPut()` si lze ověřit, že se opravdu jedná o $O(1)$ operaci. Přidání do pole je též $O(1)$, amortizovaně.

Implementaci lze nalézt ve funkci `group()` v souboru `/tests/data/DataTest.kt`.

7.3.2 C/C++

Kontejner `std::unordered_map` poskytuje podobnou funkcionalitu. Je jím operátor `[]`. Ten sice neposkytuje možnost volby pro výchozí hodnotu nenalezeného klíče, ale vytváří výchozí hodnotu dle defaultního konstruktoru daného objektu. Složitost je stejná jako v případě Kotlinu.

Implementaci lze nalézt ve funkci `group()` v souboru `data.cpp`.

7.3.3 Dart

Kód v Dartu je naprosto identický tomu v Kotlinu. Jedinou změnou je název funkce pro vrácení/vkládání páru klíč-hodnota do `map`y, místo `getOrPut()` je zde `putIfAbsent()`. Složitost se opět neliší.

Implementaci lze nalézt ve funkci `group()` v souboru `data.dart`.

7.4 Transformace dat – Seřazení

Třetí a poslední částí je seřazení záznamů v poli dle teploty, a to u všech párů klíč-hodnota v hash tabulce dle měsíce. To později umožní získání nejvyšší a nejnižší teploty měsíce a hlavně předvede rychlost řazení. Řazení by mělo dosáhnout složitosti alespoň $O(n * logn)$.

7.4.1 Kotlin

Kotlin nabízí funkci `sortBy()`, která vyžaduje parametr určující podle čeho (typicky podle členské proměnné) pole seřadit. Kotlin pro řazení využívá řazení z Javy[10], což jsou dva řadící algoritmy:

1. Dual-Pivot QuickSort – pro pole prvků primitivního typu
2. TimSort – pro pole objektů

V tomto případě se tedy použije druhý řadící algoritmus, **TimSort**. Jde o adaptivní hybridní algoritmus `merge sortu` a `insertion sortu`. Do určitého menšího počtu řazených prvků se použije `insertion sort`, jinak `merge sort`. Časová složitost je $O(n * logn)$. [9, 11]

Implementaci lze nalézt ve funkci `sort()` v souboru `/tests/data/DataTest.kt`.

7.4.2 C/C++

V C/C++ bylo využito řadící funkce `std::sort()`. Řadící algoritmus v tomto případě je **IntroSort**, což je opět hybridní řadící algoritmus, tentokrát spojený ze tří. `Quick sort`, `heap sort` a `insertion sort`. Časová složitost je opět, v nejlepším, v průměrným i v nejhorším případě, $O(n * logn)$.

Požadavkem této řadící funkce je buďto argument `comparator` nebo aby řazený objekt měl přetížený operátor `<` (`less`). Zvolena byla možnost druhá z čistě estetických důvodů.

Implementaci lze nalézt ve funkci `sort()` v souboru `data.cpp`.

7.4.3 Dart

Dartovské dynamické pole `List` má funkci pro řazení zvanou `sort()`. Přebírá lambda funkci jíž je `comparator`, který určuje podle čeho se mají prvky seřadit.

Dokumentace není příliš výmluvná, ale při pohledu na zdrojový kód a po analýze funkce `_doSort()` ([32]) lze jednoznačně usoudit, že jde opět o **TimSort**. Řazení je tedy stejné jako v případě Kotlinu.

Implementaci lze nalézt ve funkci `sort()` v souboru `data.dart`.

7.5 Zápis do úložiště

7.5.1 Kotlin

Kotlin má out-of-box podporu SQLite databáze a lze ji využít pro účel této práce. Spojení s SQLite databází zajišťuje třída `SQLiteOpenHelper`. Ta má čtyři důležité vlastnosti. První je název databáze, jenž se použije jako název souboru, ve kterém je databáze uložena. Druhou vlastností je celé číslo označující verzi databáze. Pro účel této práce je následně zapotřebí už jen přepsat funkci `onCreate()`, ve které se zavolá vytvoření vlastní tabulky `weather`.

> *Je důležité získat referenci na databázi (*`getReadableDatabase()` *nebo* `getWritableDatabase()`*) před tím, než je spuštěno měření, protože databáze se vytváří až s prvním získáním této reference.*

Po získání databáze je zavolána transakční trojice funkcí

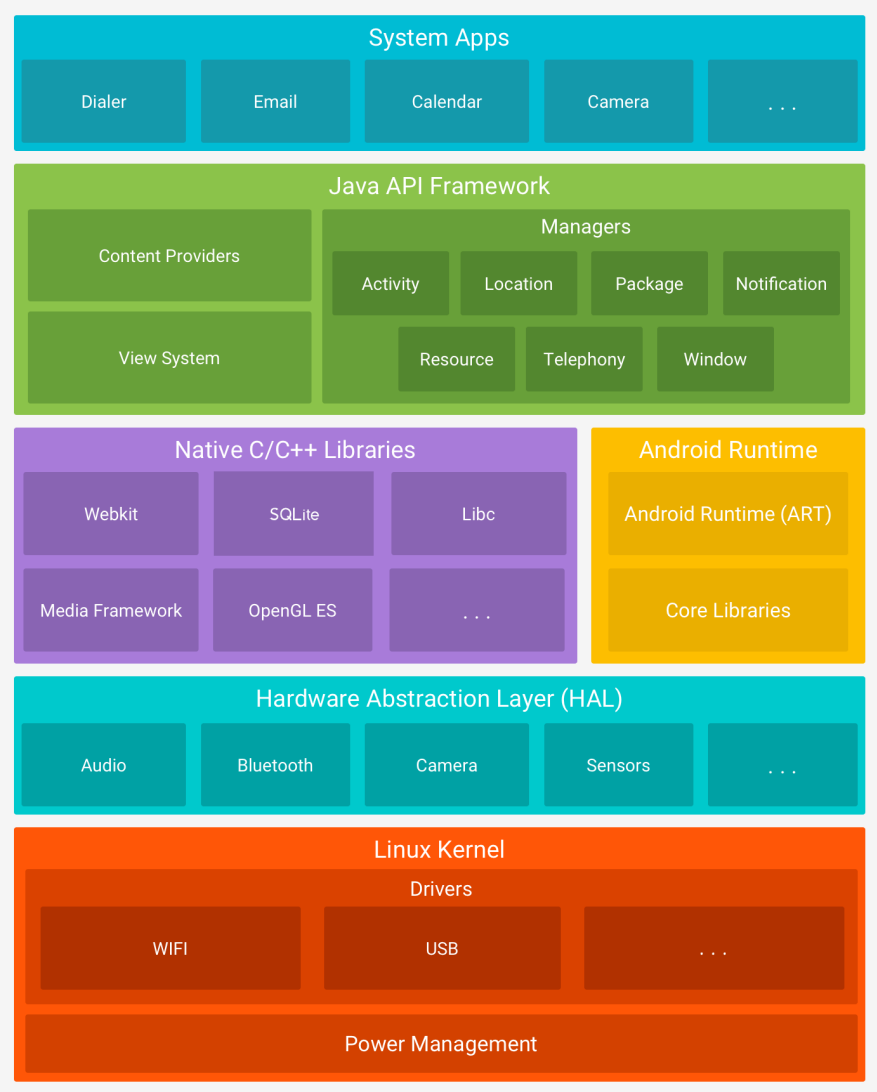
1. beginTransaction()
2. setTransactionSuccessful()
3. endTransaction()

mezi kterými jsou umístěny `insert` příkazy pro vložení řádků tabulky. Data řádků tabulky jsou získány z hash tabulky a všech jejích uložených párů klíč-hodnota.

Implementaci lze nalézt ve funkci `save()` v souboru `/tests/data/DataTest.kt`.

7.5.2 C/C++

Pracovat s databází SQLite skrze Android NDK je mnohem zajímavější část práce. Prvně lze rovnou vyřadit SQLite nadstavbu používanou na straně Kotlinu/Javy. Je tedy třeba využít přímo nativní API poskytované knihovnou SQLite. Jenže zde nastává první problém. Žádný import SQLite knihovny nebo její nadstavba, framework, wrapper neexistuje[14]. Je zřejmé, že někde v Androidu být musí, jelikož se využívá v Kotlinu/Javě. Z obrázku platformy 7.1 lze vyčíst, že SQLite knihovna patří do komponenty „Native C/C++ Libraries“. Tuto komponentu lze napasovat do obrázku NDK architektury 2.4, z čehož lze usoudit, že SQLite nebyla zakomponována do poskytovaného stabilního API.



Obrázek 7.1. Architektura platformy Android [13]

SQLite knihovna je ovšem veřejnosti přístupná, jelikož je šířená jako volné dílo [15]. Stačí tedy knihovnu stáhnout v C/C++ formě, importovat a zařadit do kompilovacího procesu.

Prvně je však třeba zjistit jakou verzi SQLite Kotlin/Java využívá. Není totiž žádoucí získat lepší výsledky jen z důvodu novější verze knihovny. Zjistit verzi knihovny

může být problém, protože verze Androidu sice přináší pevnou verzi SQLite knihovny, ale výrobci telefonů mohou tuto verzi změnit. Je tedy třeba zjistit verze na každém testovaném zařízení, což lze provést následujícím úryvkem Kotlin kódu nalezitelným v `MainActivity.kt`.

```
1  val cursor = SQLiteDatabase
2      .openOrCreateDatabase(":memory:", null)
3      .rawQuery("select sqlite_version() AS sqlite_version", null)
4
5  cursor.moveToNext()
6  print("SQLITE VERSION \${cursor.getString(0)}")
7  cursor.close()
```

Implementaci lze nalézt ve funkci `save()` v souboru `data.cpp`.

7.5.3 Dart

Implementace uložení do databáze SQLite je ve Flutteru téměř stejná jako v Kotlinu. Rozdílem je pouze syntaxe a fakt, že SQLite není ve Flutteru podporovaný out-of-box. Nicméně, Flutter tým vyvinul plugin *SQFLite*, který tuto mezeru zaplňuje.

V průběhu měřící fáze této práce se ukázalo, že existuje další faktor, který je třeba vzít v potaz. Je jím fakt, že SQFLite pouze komunikuje s SQLite v Kotlin prostředí a skrze vrstvu Flutter-services, která značně zpomaluje celý proces. Aby se zamezilo tzv. „ping-pong“ efektu mezi Dart a Kotlin prostředím, tak je třeba využít „batch“, který zabalí všechny SQL příkazy do jednoho „balíku“ a až ten se přenese mezi prostředími. [26]

Implementaci lze nalézt ve funkci `save()` v souboru `data.dart`.

Kapitola 8
Implementace - Stopky

8.1 Kotlin

Vytvoření stopek v Kotlinu je snadné. Stačí pouze jedno textové `view`, `TextView`, a tlačítko pro spuštění/zastavení/restartování stopek. Po spuštění stopek se každých 30 milisekund spočítá rozdíl času při kliknutí na tlačítko a aktuálního času. Rozdíl se následně zpracuje do minut, sekund a milisekund a zformátovaný přepíše původní rozdíl v `TextView`. Výsledkem je tedy aktualizace jednoho `view` každých 30 milisekund.

Celý tento proces se provádí na hlavním vlákně. Důvodem je, že UI se smí dotknout pouze hlavní vlákno a zbytek práce je příliš malý na to, aby odůvodnil vytváření vedlejšího vlákna či přepínání kontextu.

Implementaci lze nalézt v souboru `/tests/data/StopwatchTest.kt`.

8.2 Dart

8.2.1 Naivní přístup

Postup pro implementaci stopek ve Flutteru se zdá být učebnicový. Nejdříve se definuje `widget`, který vykresluje uběhlý čas a tlačítko pro ovládání stopek. Tento widget musí držet hodnotu, kdy byly stopky spuštěny, takže jde o `Stateful widget`. Při spuštění stopek se vytvoří instance `Timeru`, která periodicky každých 30 milisekund volá `setState()` neboli oznamuje změnu stavu a žádá o překreslení.

Toto se ale během implementace a zkušebního měření ukázalo jako naivní přístup. Problémem tohoto řešení je, že se žádá o překreslení daného `widgetu`, v tomto případě celé obrazovky, přestože je zapotřebí upravit pouze textovou část. Výkonnost vykreslování je tedy v tomto případě velmi ovlivněna. Obzvláště, když se překresluje obrazovka každých 30 milisekund.

8.2.2 Vylepšená verze

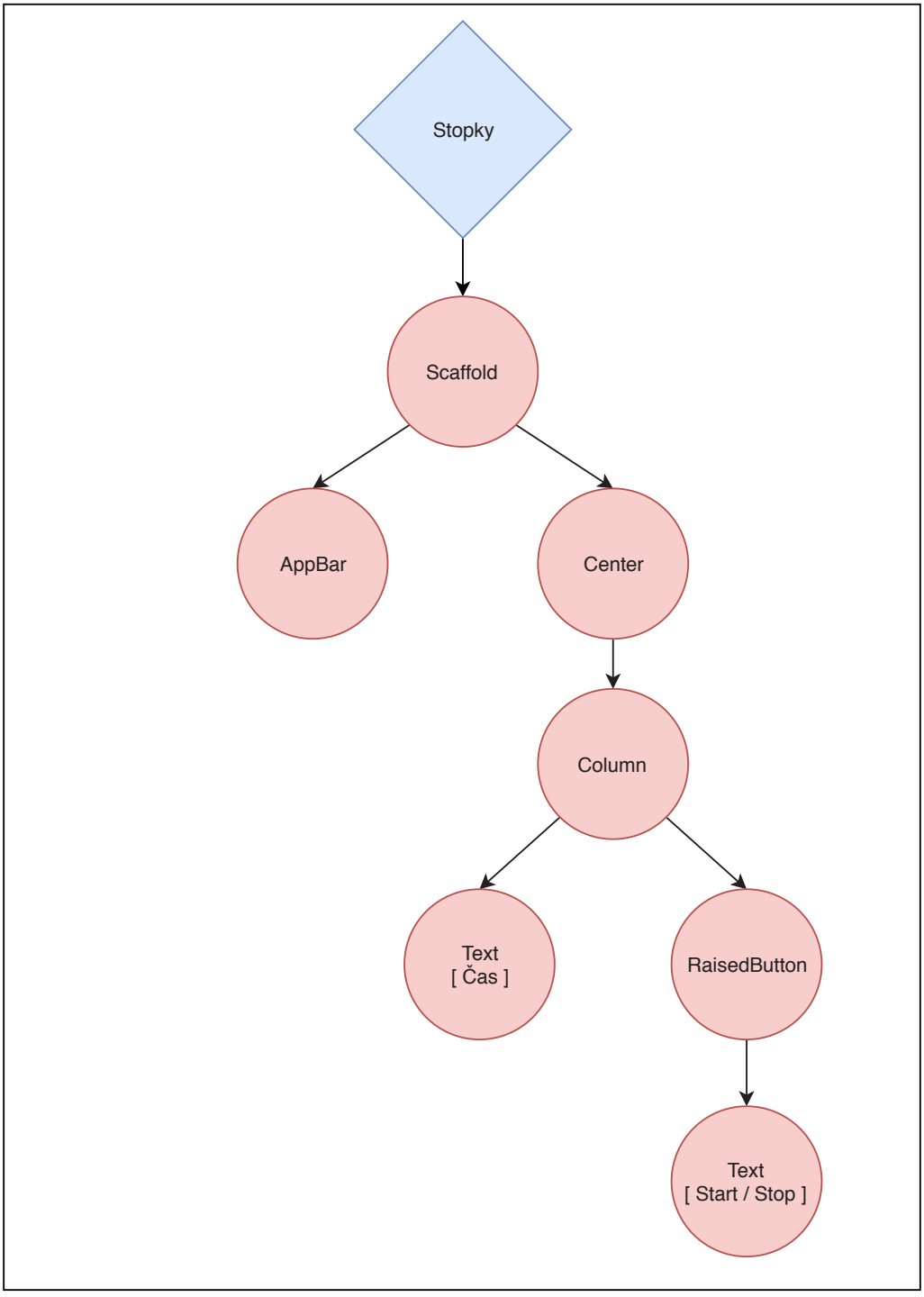
Po uvědomění si problému, je řešení zjevné. Stačí původní `widget` rozdělit na dva či rovnou na tři. Jeden `widget` starající se o text zobrazující čas stopek a druhý o zbytek obrazovky. Třech `widgetů` a zároveň nejlepšího řešení v této práci je dosáhnuto skrze rozdělení prvního `widgetu` na dva, jeden zajišťující minuty a sekundy, druhý zajišťující milisekundy.

Na obrázcích 8.1 a 8.2 lze vidět rozdíl překreslování UI. Modře podbarvené kosočtverce jsou oddělené samostatné `widgety`, které reagují na žádost překreslení. Žlutě podbarvené jsou UI `widgety`, které se každých 30 milisekund **nepřekreslují** a červeně ty, které se překreslují. Světle červenou barvou je označen textový `widget`, který zobrazuje minuty a sekundy; ten se překresluje pouze každou vteřinu neboli zhruba 1 z 34 žádostí

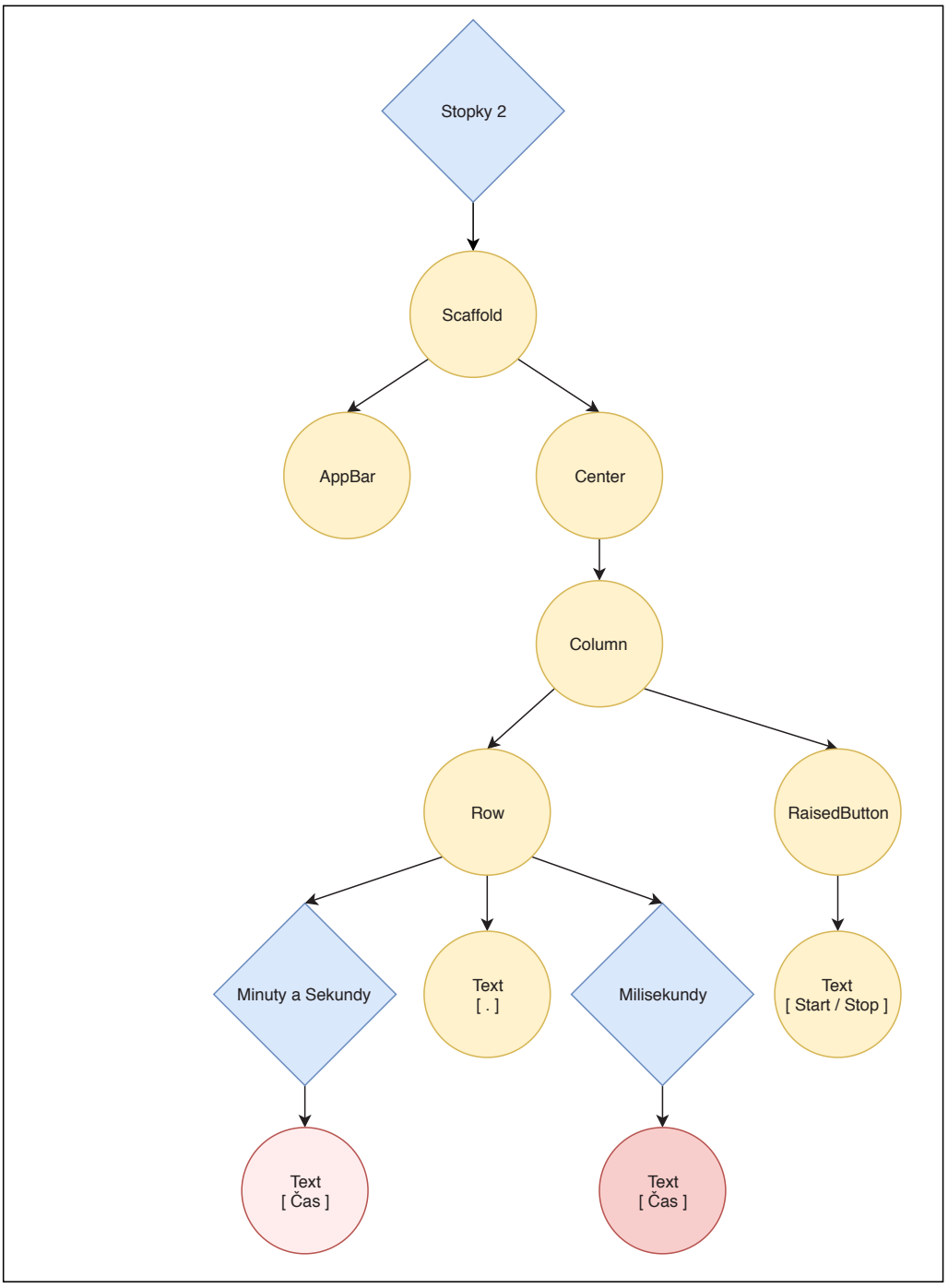
o překreslení. Pro úplnost je na obrázcích uveden i hlavní `widget Scaffold`, který implementuje základní vizuální rozložení obrazovky dle doporučení `Material Design`u.

Rozdíl je tedy patrný, ale lze jej i definovat. Buď T jednotný čas potřebný pro vykreslení jednoho UI `widget`u. Zatímco v první implementaci je třeba překreslovat sedm `widget`ů každých 30 milisekund ($7T$), tak v druhé implementaci je jich mnohem méně. V horším případě $2T$, ale amortizovaně jde o hodnotu $1T$. Na vykreslení má aplikace maximálně 16 ms (2.1) čili pokud by T bylo 3 ms, tak v první implementaci, aplikace už **obsahuje junk**, kdežto v druhé implementaci by T mohlo být **8 ms** respektive **16 ms** – tedy téměř trojnásobek respektive více než pětinásobek.

> *Vykreslení jednoho* `widget`*u trvá mnohem kratší dobu než je zmíněno v předchozím odstavci. Je tím pouze nastíněn hrubý výpočet aplikovatelný na mnohem složitější a komplexnější* `widget`*y.*



Obrázek 8.1. Naivní přístup implementace UI stopek



Obrázek 8.2. Efektivní implementace UI stopek

Kapitola 9
Porovnání

Rychlosti v „debug“ a „release“ verzích u Kotlinu jsou téměř stejné, přesto je vhodné využít pro měření „release“ verzi. Nicméně, kvůli C++ je využití „release“ verze nutné, aby se jistě aplikovaly všechny optimalizace bez omezení pro případné ladění.

Flutter s Dartem naopak přímo navádí k použití „profile“ nebo „release“ verze, protože v těchto zmíněných využívá AOT namísto JIT. Co je však zajímavější, je fakt, že „profile“ verze nelze spustit na emulátorech; pouze na fyzických zařízeních. Takto rozhodli, protože výkonnost na emulátorech není reprezentací reálné výkonnosti [8].

Hlavním zařízením pro měření se stal „chytrý“ telefon **Google Pixel** první generace. Důvodem je aktuálnost verze systému Android a osobní preference.

- Verze systému: 9 (Oreo)
- Velikost displeje: 5.0”
- Rozlišení: 1920 x 1080 px
- RAM: 4 GiB
- Procesor: Snapdragon 821
- Runtime: ART

Pro zkoušku obecné funkčnosti a vyvrácení existence specifického chování na primárním zařízení bylo použito zařízení **LG Nexus 5**.

- Verze systému: 6.0.1 (Marshmallow)
- Velikost displeje: 4.95”
- Rozlišení: 1920 x 1080 px
- RAM: 2 GiB
- Procesor: Snapdragon 800
- Runtime: ART

Posledním zařízením, pro vyzkoušení staršího běhového prostředí Dalvik, se stal originální **HTC Desire**.

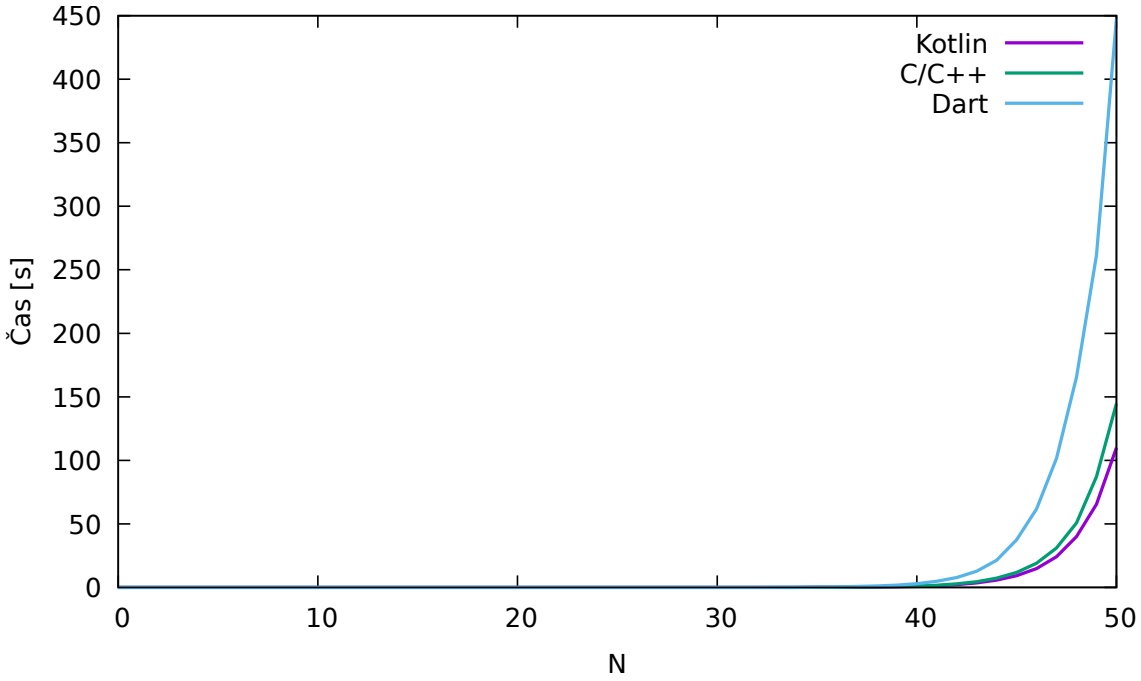
- Verze systému: 4.4 (Kitkat, CyanogenMod)
- Velikost displeje: 3.7”
- Rozlišení: 480 x 800 px
- RAM: 576 MiB
- Procesor: Snapdragon S1
- Runtime: Dalvik

9.1 Fibonacciho posloupnost

Rychlostní porovnání bylo provedeno vícekrát pro `N` hodnoty 0 až 50 a z těchto hodnot byl vypočítán průměr a medián. Průměr a medián se výrazně neliší a tak je dále zpracováván pouze medián.

N	Kotlin	C/C++	Dart
0	2	4	1
10	3	2	3
20	74	88	234
30	7239	8513	24814
40	826433	1053855	2998780
50	109906814	144693266	447650015

Tabulka 9.1. Výsledky měření rychlosti naivní rekurzivní implementace Fibonacciho algoritmu v mikrosekundách



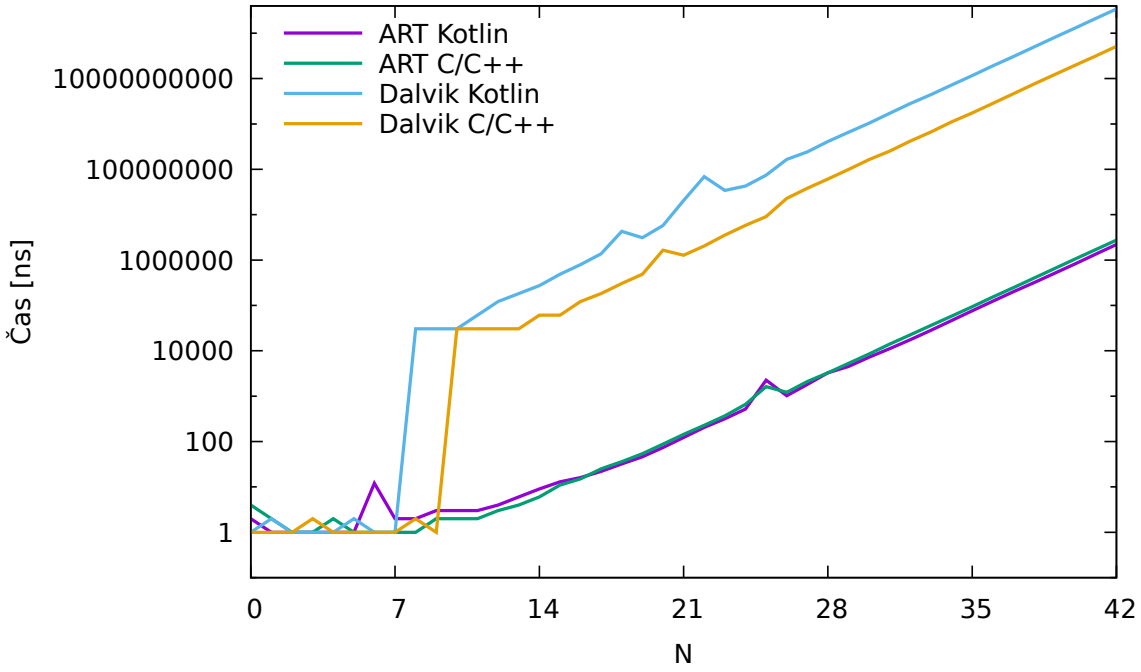
Obrázek 9.1. Medián rychlosti naivní rekurzivní implementace Fibonacciho algoritmu v mikrosekundách

Z tabulky 9.1 a grafu 9.1 lze jasně vyčíst, že pro větší `N` hodnoty je nejrychlejší Kotlin. Těsně za Kotlinem se drží C/C++, ale nejhůř je na tom Dart. Dart je oproti Kotlinu **4**× pomalejší.

Překvapením se může zdát menší rychlostní výhoda Kotlinu oproti C/C++. Je to dáno tím, že oba jazyky jsou v konečném výsledku AOT zkompilovány do nativního kódu. A tak pravděpodobně jedinou zbývající větší překážkou ve výkonnosti Kotlinu je GC, který v tomto zřejmě jednoduchém případě nehrál roli. Kompilace zdrojového kódu Kotlinu se zdá být velice efektivní, což může být dáno rokem 2015, kdy *ARM Holdings* přispělo mnoha optimalizacemi tohoto kompilačního procesu [22].

> *„ARM contributed large amounts of code to the Android Open Source Project to improve the efficiency of the bytecode compiler in ART.“ [22]*

Pro úplnost by bylo záhodno změřit i rozdíl pro zařízení s JIT (Dalvik VM), jelikož podíl na trhu takovýchto zařízení je stále 11 % [23].



Obrázek 9.2. Srovnání Kotlinu ART/Dalvik proti C/C++ s log. stupnicí na ose `Y`

Do grafu 9.2 (s logaritmickou stupnicí na ose `Y`) byly zaneseny údaje jak z ART, tak z běhového prostředí Dalvik. V tomto grafu nehraje čas žádnou roli mezi Kotlin implementacemi, jelikož jde o různá zařízení, jiné procesory a celkově obří rozdíl ve stáří všech komponent. Důležitým faktorem je zde rozdíl oproti C/C++. Z grafu je vidět, že všechny trendy jsou exponenciální. Avšak pro ART je stoupání stejné pro oba jazyky (jak bylo vysvětleno výše), kdežto pro Kotlin na Dalvik VM (JIT) je stoupání mnohem rychlejší než pro jeho C/C++ protějšek. Pro specifičnost následuje tabulka 9.2 s extrapolovanými vzorci, ze kterých lze vyčíst, že konstanta e u Kotlinu (Dalvik) je **6×** větší.

	Kotlin	C/C++
ART	$0.00395e^{0.479x}$	$0.00461e^{0.481x}$
Dalvik	$570e^{0.481x}$	$94.6e^{0.479x}$

Tabulka 9.2. Extrapolované vzorce trendů pro exponenciální růst Kotlinu a C/C++

Z důvodu 90% zastoupení na trhu, očividné nadřazenosti a dalších menších detailů ohledně implementace je ve zbylých měřeních využito pouze zařízení s ART.

9.2 Rychlost zpracování dat

Do grafu 9.3 jsou promítnuty dílčí části jednoho většího úkonu, které se mohou vyskytnout v přibližné podobě v reálné aplikaci. Kvůli větší časové odchylce jedné z implementace jsou stejná data promítnuta též do grafu 9.4 s logaritmickou stupnicí na ose `Y`.

První část, **read**, je čtení ze souboru s bufferem o velikosti 64 KiB. Rozdíl mezi Kotlinem a C/C++ je minimální a rozdíl oproti nejvhodnější implementaci v Dartu je taktéž zanedbatelný. Pro čtení souboru, který je uložen v interní paměti není z pohledu výkonnosti žádný jazyk jasným vítězem.

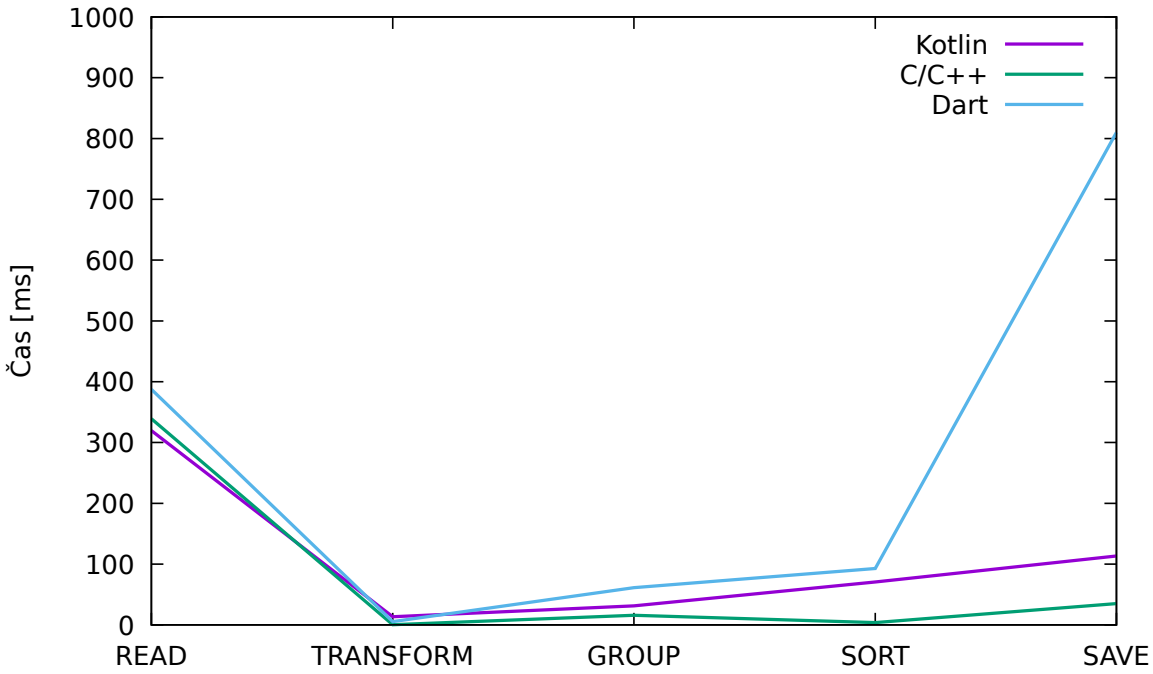
Další dílčí částí je **transform**, která vytváří hluboké kopie 89119 struktur a při kopírování změní hodnotu teplotní proměnné na stupnici Fahrenheita z původní stupnice Celsia. V této části lze najít jediný výskyt rychlejšího Dartu oproti Kotlinu, který zde rozhodně tratí. Nicméně, C/C++ je zde zdaleka nejefektivnější.

Group část překopírovává teplotní data do jiné struktury, která je rozdělena dle unikátního měsíce. Konkrétně se jedná o hashovací tabulku, obsahující seznam teplotních údajů v daném měsíci. C/C++ opět vítězí v rychlosti.

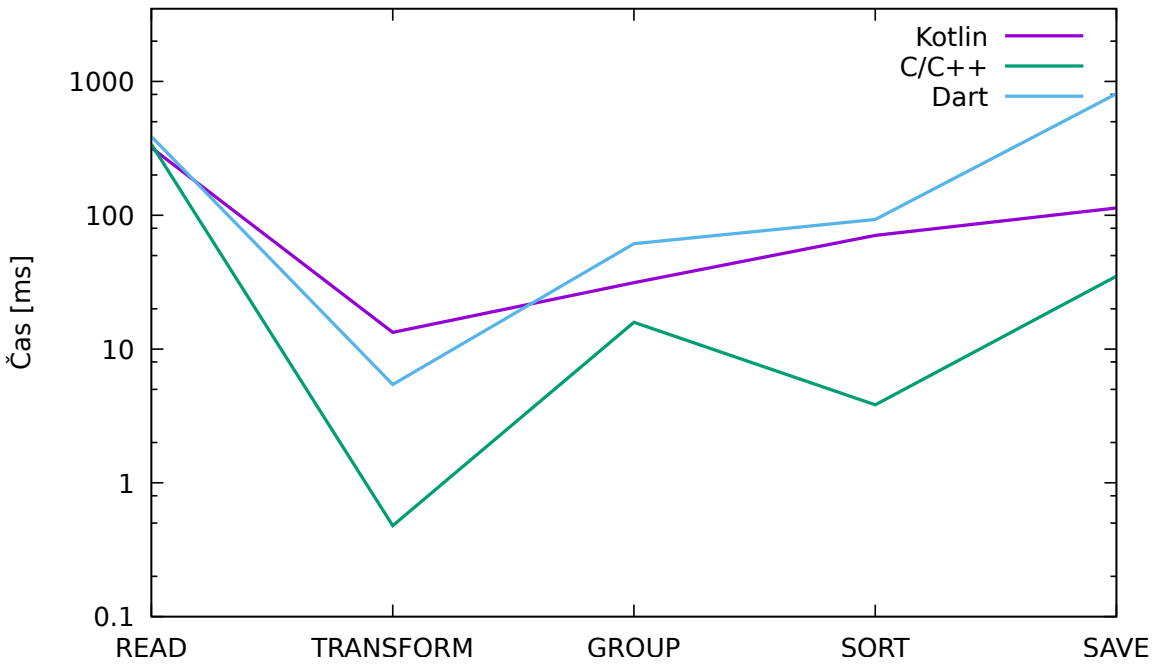
Čtvrtá část, **sort**, je již zajímavější. C/C++ je nejrychlejší a kromě obecně efektivnějšího jazyku, se zde též projevuje jiný algoritmus pro řazení – IntroSort. Dart i Kotlin využívají řadící algoritmus TimSort, což lze potvrdit i z grafů (9.3, 9.4), kde rychlostní údaje jsou velice blízko jedné hodnoty. Přesto se Kotlin stále projevil, zanedbatelně, rychlejší.

Nejzajímavější je poslední část, **save**. Přestože jsou využity stejné verze knihovny SQLite, tak výsledky jsou velmi rozdílné. V C/C++ se přistupuje přímo k veřejnému API knihovny SQLite, kdežto v Kotlinu existuje několik prostředních metod než se zavolá stejná nativní knihovna zabudovaná v platformě. Tento menší rozdíl mezi Kotlinem a C/C++ lze takto odůvodnit. Odstrašujícím případem je však SQLite (plugin SQFLite) v Dartu pro Flutter. Přestože je plugin SQFLite doporučován na oficiálních stránkách Flutteru [24], tak je zřejmé, že výkonnostně má velké nedostatky. Nalezeným důvodem je pouze vrstva Flutter-services, která slouží jako prostředník mezi Dartem a Javou. Tedy k času, který je potřeba pro ekvivalentní úkon v Kotlinu je třeba přičíst režii několikanásobného volání, předání dat a zároveň transformaci celého příkazu, jelikož Flutter-services obsahují speciální API pro komunikaci mezi těmito dvěma světy. V tomto případě je tedy Flutter nevhodným kandidátem pro specifické aplikace a je třeba vyčkat na podporu přímější komunikace mezi nativní SQLite knihovnou a SQFLite pluginem, nebo využít alternativu pro perzistenci dat.

> *Při dokončování této práce se objevil nový problém na oficiálním githubu pluginu SQFLite, který zmiňuje, že problém se nevyskytuje na iOS, pouze na Androidu. Samotný vývojář pluginu prozatím neobjevil řešení ani jádro problému. [25]*



Obrázek 9.3. Medián doby zpracování dat v milisekundách



Obrázek 9.4. Medián doby zpracování dat v milisekundách s log. stupnicí na ose `Y`

9.3 Vykreslování UI

Není-li uvedeno jinak, rozumí se, že všechny následující testy byly provedeny vícenásobně a dále byl zpracováván vypočtený medián hodnot.

9.3.1 Inflatování v Kotlinu

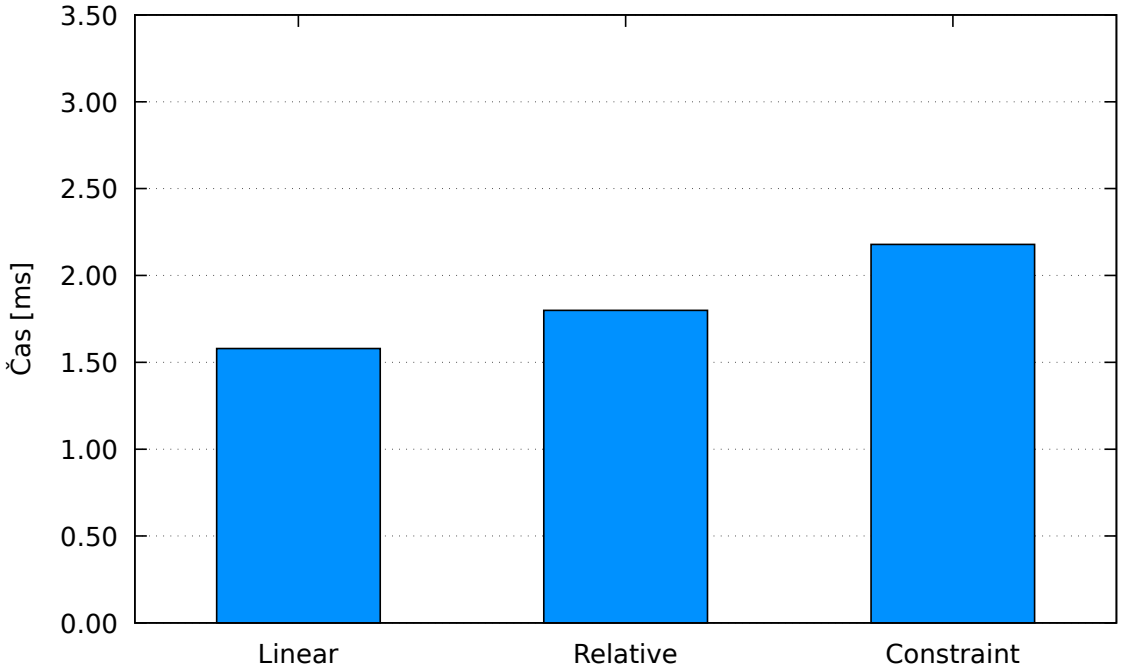
Ač je volání `inflate` mimo fáze vykreslování, tak v konečném důsledku téměř vždy předchází vykreslení daného `layoutu`. Je proto zajímavé a účelné zjistit, který typ `layout view` je nejvhodnější pro který případ. Obecně jde hlavně o statickou analýzu XML, kterým je `layout` popsán a tak lze usoudit, že čím méně řádků a složitějších parametrů daného XML, tím rychlejší `inflate` bude.

Na grafech 9.5, 9.6, 9.7 a 9.8 lze najít časy potřebné pro vykreslení všech navržených rozmístění v jednom `layoutu`.

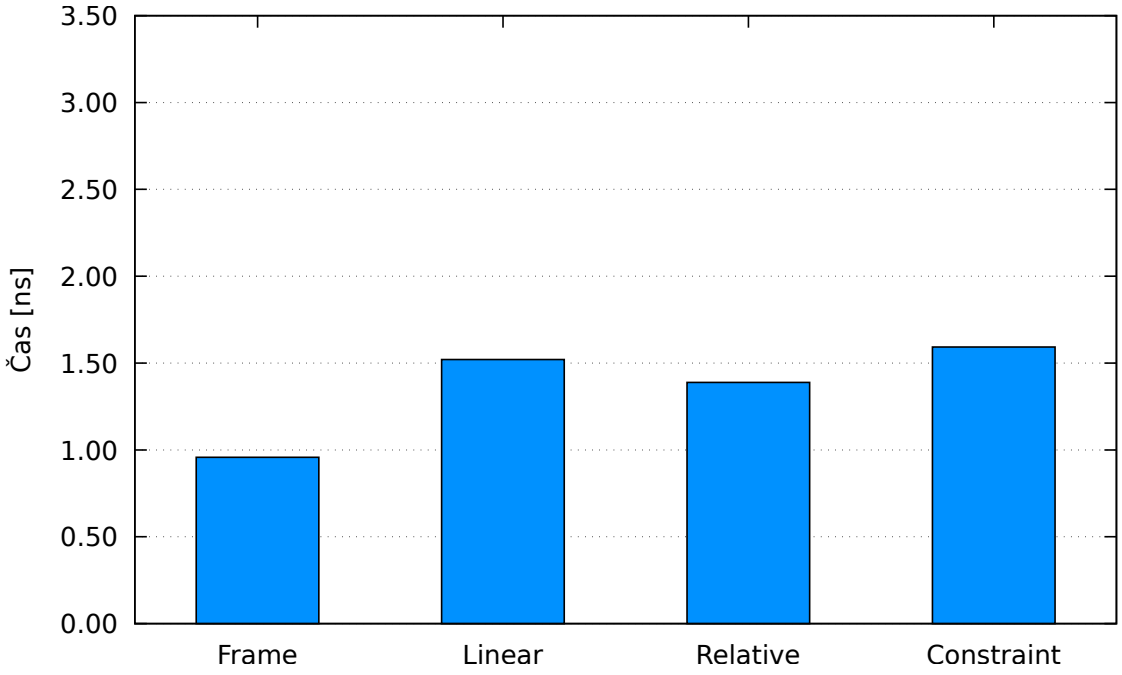
Není překvapením, že `ConstraintLayout` vyšel ve všech čtyřech testech jako nejhorší. Jeho síla spočívá právě v komplexních parametrech každého prvku, takže statická analýza logicky trvá nejdéle.

Zajímavostí je graf 9.7, který znázorňuje inflate jediného prvku uprostřed obrazovky. Jelikož `FrameLayout` je nejjednodušší, tak byl předpokládaným vítězem, ale namísto něho zvítězil `RelativeLayout`. Že je měření v pořádku a čas koreluje se složitostí XML lze vyčíst z grafu 9.6, ve kterém `RelativeLayout` má o něco více parametrů. Závěrem tedy je, že vytvoření stromu objektů je pro `RelativeLayout` snazší než pro `FrameLayout`.

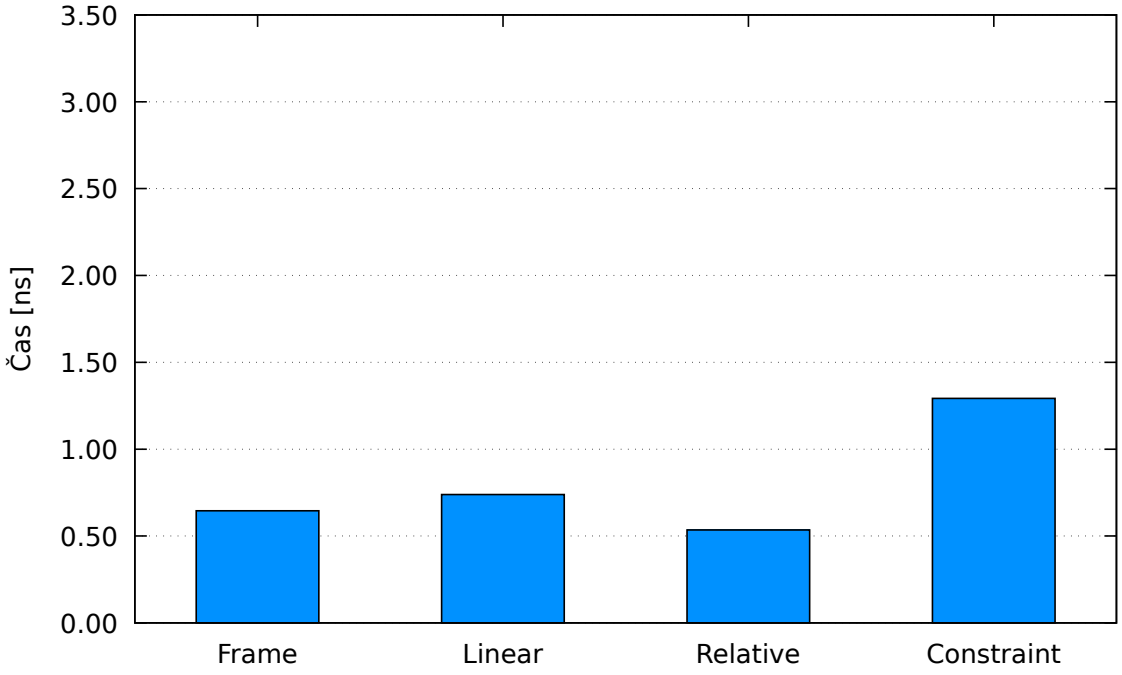
Nicméně je třeba dodat, že ač je tento test zajímavý, tak by neměl ovlivňovat vývojářovu volbu, jelikož z pohledu výkonnosti nemá téměř žádnou roli. Doba vykreslení nejrychlejšího layoutu z tohoto testu může trvat mnohem déle než u nejpomalejšího. Celkový čas by se pak ve stejných grafech mohl promítnout v naprosto opačném pořadí. Závěrem tedy je, že vývojář by měl zvolit takový `layout view`, aby účel odpovídal požadované skladbě prvků.



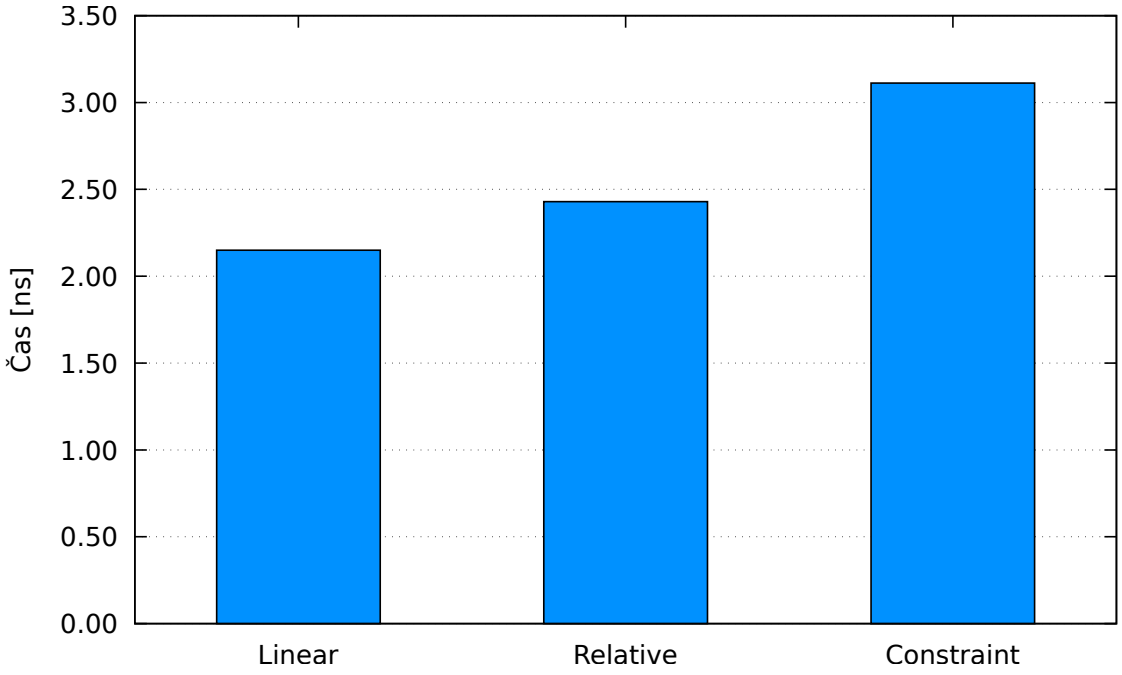
Obrázek 9.5. Medián doby inflatování prostého layoutu v milisekundách



Obrázek 9.6. Medián doby inflatování roztaženého layoutu v milisekundách



Obrázek 9.7. Medián doby inflatování vycentrovaného layoutu v milisekundách



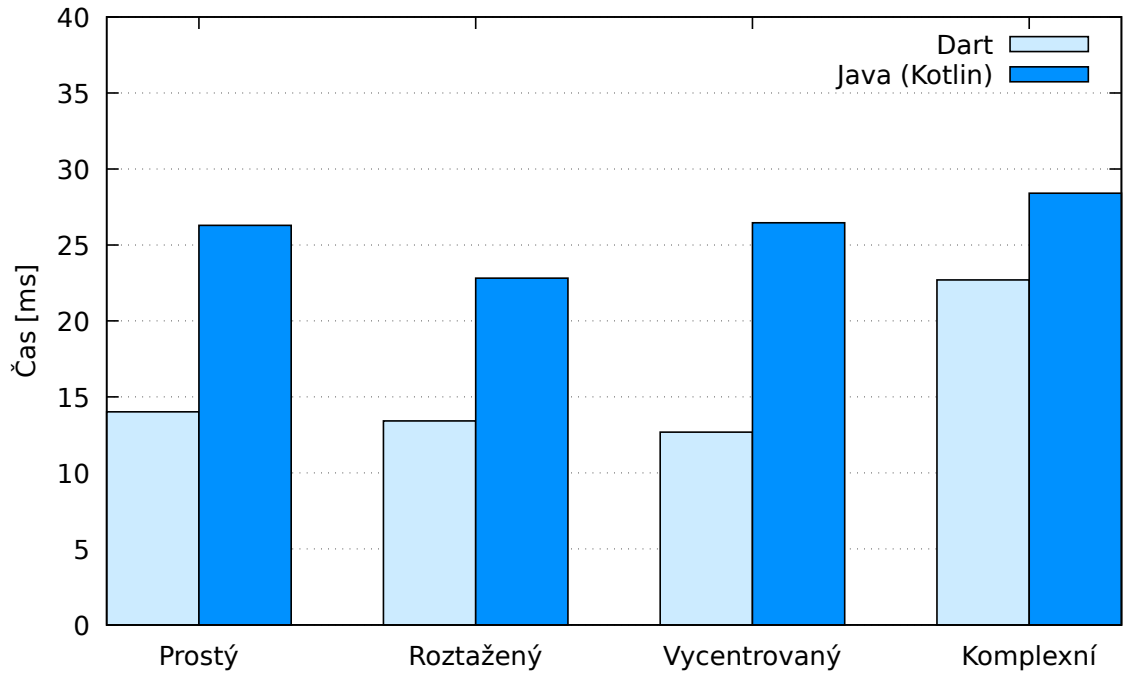
Obrázek 9.8. Medián doby inflatování komplexního layoutu v milisekundách

9.3.2 Vykreslení

Do grafu 9.9 byl zanesen čas potřebný pro vykreslení všech navržených `layoutů`. Daný čas sice neobsahuje dobu inflatování, ale na straně Kotlinu bylo vždy vybráno nejvhodnější `layout view` podle celkového času včetně inflatování.

Ze zmíněného grafu lze vyčíst jasnou nadřazenost Dartu. Zatímco i v nejjednodušších případech Kotlin přesahuje limitní hranici 16 ms, tak Dart se drží pod ní.

Důvodem pro obecně pomalé časy v tomto testu je návrh testu samotný. Kvůli oddělení vykreslovacího procesu od jakéhokoliv jiného byl začátek vykreslování posunut o 5 sekund dopředu a tím dosaženo neexistující interference od animací, přechodů atd. Zároveň se tím zrušily veškeré optimalizace, na které jsou frameworky připraveny – například optimalizace vykreslovacího procesu při přechodu mezi obrazovkami.

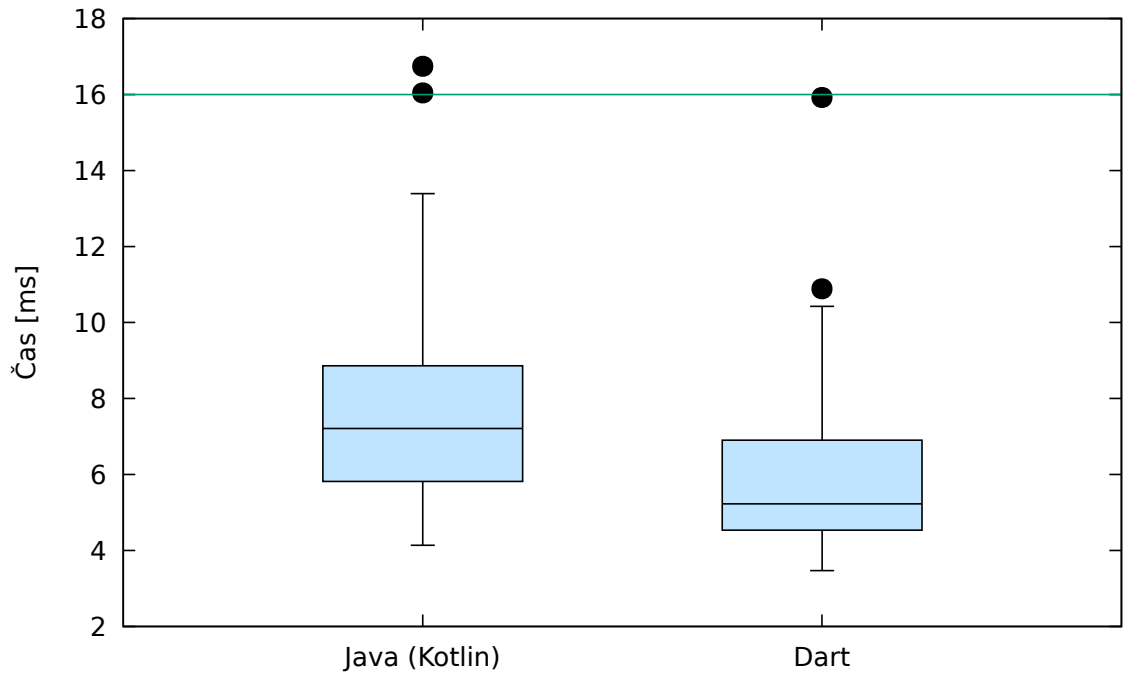


Obrázek 9.9. Medián doby vykreslení v milisekundách

9.3.3 Posuvný seznam prvků

I v tomto případě vykreslování byl Dart rychlejší, jak lze vyčíst z krabicového grafu 9.10. Tento test spočíval ve vytvoření 300 prvků v posuvném seznamu a skrze aplikaci třetí strany simulovat ekvivalentní posun. Z tohoto posunu byly následně zpracovány časy vykreslení prvních 150 snímků, které jsou zaneseny do zmíněného grafu.

Jednotlivý prvek seznamu není složitým či náročným `layout`em, ale i tak lze vidět rozdíl. Kromě tohoto evidentního posunu lze také najít několik outlierů, které přesáhly hranici 16 ms pouze na straně Kotlinu.



Obrázek 9.10. Doba vykreslování při posunu listu prvků v milisekundách

9.4 Využití hardwarových prostředků

Pomocí stopek jsem porovnal využití hardwarových prostředků CPU a RAM. V Kotlinu i Dartu bylo využito jednoduchého rozdílu času v milisekundách, který byl následně rozdělen na minuty, sekundy a milisekundy. Tyto hodnoty byly promítnuty do textové reprezentace na obrazovce periodicky každých 30 milisekund. Původně byl plánován i rozbor rychlosti vykreslování, ale ten se neprojevil jako užitečný, jelikož se data téměř nelišila a většina spadala do limitu 16 ms na každý snímek.

Změřená data pocházejí ze 2 minut spuštěných stopek a ze třech různých instancí. Instancemi jsou stopky v Kotlinu, stopky v naivní implementaci v Dartu a stopky v optimalizované implementaci v Dartu – „Kotlin“, „Dart v1“ a „Dart v2“.

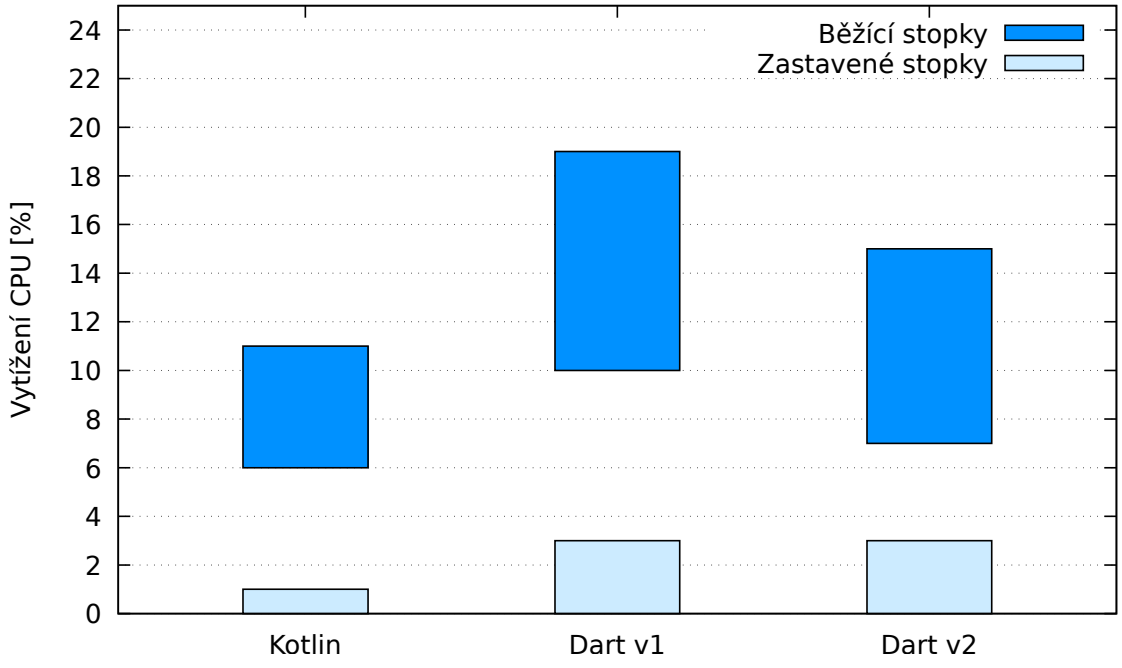
9.4.1 CPU

Pro měření CPU byl využit nástroj Profiler, který je součástí Android Studia. Ten umožňuje měřit nativní aplikace i ostatní procesy, pod které spadá aplikace vyvíjená ve Flutteru.

Z grafu 9.11 lze vyčíst, že vytížení CPU je v Dartu vyšší než v Kotlinu; zejména tak v neoptimalizované implementaci. Minimální vytížení optimalizované implementace v Dartu se již blíží minimu v Kotlinu.

Procentuálně, je vytížení CPU v Dartu 140 % vytížení v Kotlinu. Nicméně, z pohledu absolutních čísel jde o maximální rozdíl vytížení mezi hodnotami 11 a 15 procent. Ač jsou tyto hodnoty přímo korelující s výdrží baterie, tak lze tento rozdíl zanedbat. Zejména v porovnání s jinými multiplatformními řešeními, které povětšinou využívají Javascript, čímž využívají CPU mnohem více [21], jsou tyto výsledky pozitivní.

V grafu 9.11 je též uvedené vytížení při zastavených stopkách. Zde lze vidět třínásobné vytížení, které už tak pozitivní není. Pokud aplikace neprovádí žádný úkon ani překreslování obrazovky, tak optimisticky i realisticky by vytížení CPU mělo být co nejnižší.



Obrázek 9.11. Stopky – Vytížení CPU v procentech

9.4.2 RAM

Měření využití RAM je též prováděno skrze nástroj Profiler, jenž je součástí Android Studia, a zároveň je využit nástroj DevTools, který byl vytvořen za stejným účelem pro Dart a Flutter. Zatímco v prvním nástroji lze vyčíst absolutní využití RAM, druhý nástroj se vymezuje pouze na reálné využití zdrojovým kódem aplikace – stopek.

Obrázek 9.12. Kategorie alokací v paměti podle účelu či vlastníka v Android aplikacích

Profiler přidává navíc rozdělení alokací dle kategorií 9.12.

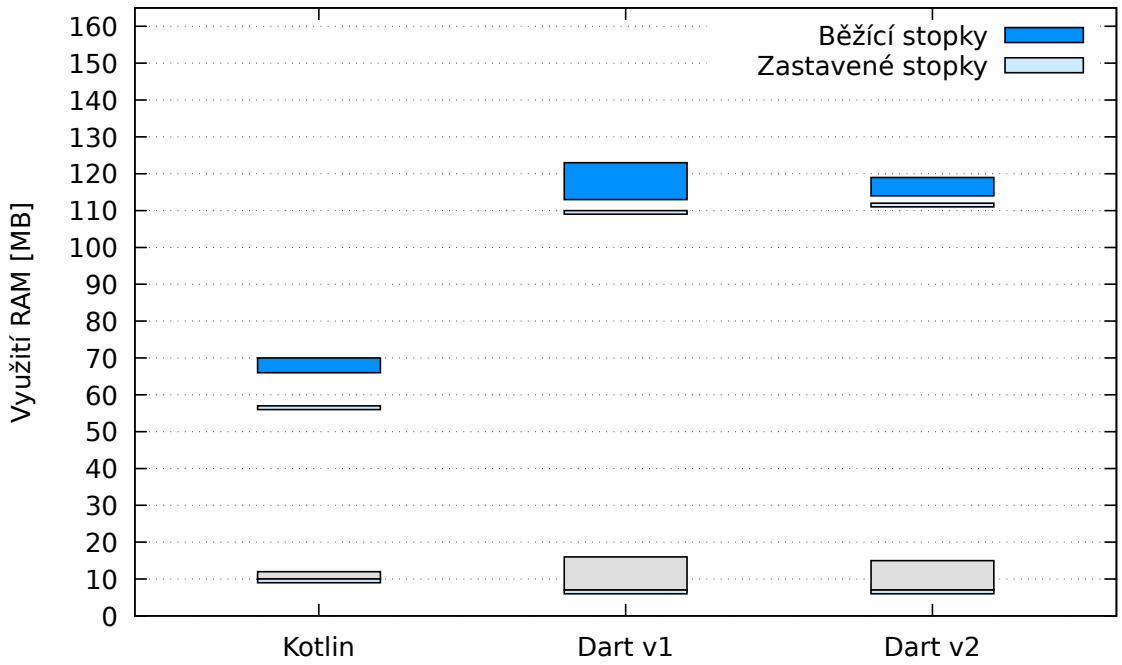
- **Java**: Objekty alokované z Java nebo Kotlin zdrojového kódu.
- **Native**: Objekty alokované z C/C++ zdrojového kódu. Včetně zdrojového kódu Android frameworku, který některé úkony implementuje právě v C/C++; například manipulace s obrázky.
- **Graphics**: Alokace grafických `buffer`ů za účelem zobrazování pixelů na obrazovce.
- **Stack**: Paměť využívaná nativním a Java zásobníkem – většinou se vztahuje k počtu běžících vláken.
- **Code**: Paměť používaná pro kód a prostředky jako `dex` bajtkód, sdílené knihovny (`.so`) a písma.
- **Others**: Nespecifikované objekty v paměti
- **Allocated**: Počet Java/Kotlin objektů alokovaných aplikací. Neobsahuje počet objektů alokovaných v C/C++.

Do grafu 9.13 je promítnut rozsah využití RAM, tedy minimální respektive maximální hodnota je typicky před respektive po probuzení GC.

I přes optimalizaci implementace stopek, je využití RAM oproti vytížení CPU mnohem méně pozitivní. Zatímco v nativní aplikaci, psanou v Kotlinu, je využití kolem 70 MB, tak v Dartu hodnota začíná na 110 MB. Ač procentuálně jde opět o nárůst „pouhých“ 57 %, tak absolutní čísla nemusí být vždy zanedbatelná. Obzvláště z pohledu korelace s výdrži baterie.

Na druhou stranu je třeba se zaměřit i na šedou část grafu 9.13. Šedé sloupce určují reálné využití RAM zdrojovým kódem stopek. V Profileru je využití rozděleno do několika kategorií, z nichž byly vybrány kategorie „Java“ a „Others“, které byly sečteny a promítnuty do šedého sloupce Kotlinu. OpenDev tools se soustředí zejména na zdrojový kód aplikace samotný a údaje jsou tedy z tohoto nástroje jednoznačné. Na zmíněném grafu lze vidět, že ač rozpětí využití je u Dartu větší, tak se pohybuje na stejné hranici jako Kotlin. Což lze považovat za logický úsudek vzhledem k ekvivalentní implementaci, nicméně jiné zdroje tuto informaci explicitně nesdílí a ukončují měření u absolutních čísel.

Pro úplnost bylo zapotřebí zjistit, v čem se alokace paměti RAM liší. Při pohledu na rozdělení alokací dle kategorií, lze jednoznačně vyvodit závěr, že jedinou objemnější kategorií je kategorie „Graphics“. Do této kategorie spadají alokace grafických `buffer`ů pro zobrazení pixelů na obrazovce. Vzhledem k tomu, že primární cíl, pro který byl Flutter vytvořen, se týká právě vykreslování pixelů na obrazovce, je tato objemnější alokace očekávatelná.



Obrázek 9.13. Stopky – Využití RAM v MB

Vedlejším pozorováním bylo též změřeno, že za běhu stopek je GC probuzen v Kotlinu každých 50 sekund, v neoptimalizovaných stopkách v Dartu každých 8 sekund a v optimalizovaných stopkách v Dartu každých 14 sekund.

Kapitola 10
Závěr

Cílem této diplomové práce bylo seznámení se s novým frameworkem Flutter a jeho použitím na vývoj aplikací pro mobilní systém Android. S ohledem na toto seznámení bylo zadáno provést výkonnostní porovnání jazyka Dart s konvenčně a dlouhodobě využívanými jazyky Kotlin a C/C++ pro vývoj aplikací pro systém Android.

V první fázi jsem se zabýval analýzou systému Android jako takového. Jak se aplikace kompilují, vykreslují a konečně jejich celkový běh na systému.

Následovala primární fáze návrhu a implementace takových úloh, abych mohl porovnat výkonnost aplikací napsaných ve zmíněných jazycích Dart, Kotlin a C/C++. Mezi tyto úlohy patří hlavně vykreslování UI a zpracování velkého objemu dat, tedy čtení ze souboru, analýza či transformace a zápis do úložiště.

Poslední fází bylo změřit výkonnost na těchto úlohách dle zjištěných výkonnostních metrik. Provedl jsem několik porovnání zmíněných jazyků v podobě rychlosti rekurzivního algoritmu, rychlosti a průběhu vykreslování obrazovky, rychlosti běžných operací jakými jsou čtení ze souboru, hluboká kopie, řazení a ukládání do perzistentního úložiště (relační SQLite databáze). Kromě rychlostních porovnání jsem též provedl měření využití hardwarových prostředků jakožto využití paměti RAM nebo vytížení CPU.

10.1 Úsudek

Jazyk C/C++ pod záštitou Android NDK je stále nejefektivnějším jazykem, který lze použít na vývoj aplikací pro mobilní systém Android. Stále je však určen pouze pro datové manipulace, náročné výpočty či herní enginy. C/C++ není vhodným kandidátem pro vývoj běžné aplikace; zejména UI.

Kotlin pod záštitou Android SDK poskytuje vše, co je potřeba pro vývoj plnohodnotné aplikace pro Android za cenu lehce menší výkonnosti.

Dart a Flutter nastupuje jako mladý a v několika směrech nedokonalý, ba dokonce nedokončený způsob vývoje aplikací pro Android. Nicméně, výkonnostně na tom není nijak zvlášť pozadu oproti dříve zmíněným jazykům a k tomu přináší rychlejší vykreslování implikující mnohem plynulejší UI. Pokud k tomu připočtu v určitých směrech snazší a vždy mnohem rychlejší vývoj **multiplatformního** řešení, tak se Dart stává žádoucím kandidátem pro vývoj mnoha aplikací.

Literatura

[1] Android Developers. *ABI Management* [online]. [cit. 2019-04-08]. Dostupné z: https://developer.android.com/ndk/guides/abis

[2] Unity. *Mobile (Android) Hardware Stats* [online]. [cit. 2019-04-08]. Dostupné z: https://web.archive.org/web/20170808222202/http://hwstats.unity3d.com:80/mobile/cpu-android.html

[3] Dan Galpin. *Using the NDK Performantly (Big Android BBQ 2015)* [online]. 2015 [cit. 2019-04-08]. Dostupné z: https://www.youtube.com/watch?v=Wb5HAI73QRE

[4] Pascal Welsch. *Heavy lift work in Flutter get started with Isolates* [online]. 2018 [cit. 2019-04-08]. Dostupné z: https://youtu.be/M8jGSkACneE?t=115

[5] Google. *JNI Tips* [online]. [cit. 2019-04-08]. Dostupné z: http://developer.android.com/training/articles/perf-jni.html

[6] Flutter. *Flutter FAQ* [online]. [cit. 2019-04-08]. Dostupné z: https://flutter.dev/docs/resources/faq

[7] Flutter. *Flutter performance profiling* [online]. [cit. 2019-04-08]. Dostupné z: https://flutter.dev/docs/testing/ui-performance

[8] Flutter. *Flutter's build modes* [online]. [cit. 2019-04-08]. Dostupné z: https://flutter.dev/docs/testing/build-modes

[9] OpenJDK 8. *Arrays* [online]. [cit. 2019-04-17]. Dostupné z: https://devdocs.io/openjdk~8/java/util/arrays#sort-java.lang.Object:A-

[10] JetBrains. *ArraysJVM* [online]. [cit. 2019-04-18]. Dostupné z: https://github.com/JetBrains/kotlin/blob/master/libraries/stdlib/jvm/src/generated/_ArraysJvm.kt#L1789-L1798

[11] AndroidSDKSources. *Arrays* [online]. [cit. 2019-04-18]. Dostupné z: https://github.com/AndroidSDKSources/android-sdk-sources-for-api-level-28/blob/master/java/util/Arrays.java#L1424

[12] Dart. *sdk/sort.dart* [online]. [cit. 2019-04-17]. Dostupné z: https://github.com/dart-lang/sdk/blob/da0363172adaa266ef2d44cf87e09b61a185e79d/sdk/lib/internal/sort.dart

[13] Android Developers. *Platform Architecture* [online]. [cit. 2019-04-18]. Dostupné z: https://developer.android.com/guide/platform

[14] Android Developers. *Android NDK Native APIs* [online]. [cit. 2019-04-18]. Dostupné z: https://developer.android.com/ndk/guides/stable_apis.html

[15] SQLite. *SQLite Copyright* [online]. [cit. 2019-04-18]. Dostupné z: https://www.sqlite.org/copyright.html

[16] Android Developers. *Inspect GPU rendering speed and overdraw* [online]. [cit. 2019-04-24]. Dostupné z: https://developer.android.com/studio/profile/inspect-gpu-rendering

[17] Android Developers. *Analyze with Profile GPU Rendering* [online]. [cit. 2019-04-24]. Dostupné z: https://developer.android.com/topic/performance/rendering/profile-gpu

[18] Android Developers. *Performance and view hierarchies* [online]. [cit. 2019-04-24]. Dostupné z: https://developer.android.com/topic/performance/rendering/optimizing-view-hierarchies

[19] Adam Barth. *Flutter's Rendering Pipeline* [online]. 2016-5-5 [cit. 2019-04-25]. Dostupné z: https://youtu.be/UUfXWzp0-DU

[20] Google. *Flutter System Architecture* [online]. 2017-4-23 [cit. 2019-04-25]. Dostupné z: https://docs.google.com/presentation/d/1cw7A4HbvM_Abv320rVgPVGiUP2msVs7tfGbkgdrTy0I/

[21] Purvak Pathak. *Solving Over 35 performance Issues in React Native* [online]. 2017-11-30 [cit. 2019-04-30]. Dostupné z: https://dev.to/purvak_pathak/react-native-performance-issues-and-insights-on-improving-it-b39

[22] Gary Sims. *Java vs C app performance – Gary explains* [online]. 2016-05-03 [cit. 2019-05-01]. Dostupné z: https://www.androidauthority.com/java-vs-c-app-performance-689081/

[23] Android Developers. *Distribution dashboard* [online]. 2018-10-26 [cit. 2019-05-01]. Dostupné z: https://developer.android.com/about/dashboards

[24] Flutter. *Persist data with SQLite* [online]. [cit. 2019-05-01]. Dostupné z: https://flutter.dev/docs/cookbook/persistence/sqlite

[25] Alex Miller. *Raw sql query is way slower on Android than iOS* [online]. 2019-04-30 [cit. 2019-05-01]. Dostupné z: https://github.com/tekartik/sqflite/issues/202

[26] Alexandre Roux. *SQFLite documentation* [online]. 2018-02-26 [cit. 2019-05-01]. Dostupné z: https://github.com/tekartik/sqflite/blob/master/sqflite/README.md

[27] Google Developers. *Advanced Android 04.1 Part A: Profile GPU Rendering tool* [online]. [cit. 2019-05-01]. Dostupné z: https://codelabs.developers.google.com/codelabs/advanced-android-training-profile-gpu/index.html

[28] Flutter. *The Engine architecture* [online]. 2018-11-05 [cit. 2019-05-01]. Dostupné z: https://github.com/flutter/flutter/wiki/The-Engine-architecture

[29] Matt Sullivan. *Flutter: Don't Fear the Garbage Collector* [online]. 2019-01-04 [cit. 2019-05-02]. Dostupné z: https://medium.com/flutter-io/flutter-dont-fear-the-garbage-collector-d69b3ff1ca30

[30] Google. *Debugging ART Garbage Collection* [online]. [cit. 2019-05-02]. Dostupné z: https://source.android.com/devices/tech/dalvik/gc-debug

[31] Český hydrometeorologický ústav. *Praha Klementinum* [online]. [cit. 2019-05-03]. Dostupné z: http://portal.chmi.cz/historicka-data/pocasi/praha-klementinum

[32] Dart. *sdk/sort.dart* [online]. [cit. 2019-05-03]. Dostupné z: https://github.com/dart-lang/sdk/blob/master/sdk/lib/internal/sort.dart#L55

Příloha A
Zkratky

- AOT – Ahead Of Time
- API – Application Programming Interface
- ART – Android Runtime
- CPU – Central Processing Unit
- DFS – Depth-first search
- FPS – Frames Per Second
- GC – Garbage Collector
- GCC – GNU Compiler Collection
- GPU – Graphics Processing Unit
- I/O – Input/Output; vstupní a výstupní operace – čtení a zápis ze souboru, síťové operace apod.
- JIT – Just In Time
- JNI – Java Native Interface
- JVM – Java Virtual Machine – VM pro zpracování Java bajtkódu
- LLVM – Kolekce modulárních a znovupoužitelných kompilátorských technologií
- NDK – Native Development Kit
- RAM – Random Access Memory
- SDK – Software Development Kit
- UI – User Interface
- VM – Virutal Machine – Virtuální stroj je modul emulující počítačový systém a jeho účelem je zpracovávat tzv. mezikód

Příloha B
Elektronická příloha práce

- **apk/** – adresář se spustitelnou formou implementace
- **src/**
  - **android_sdk_ndk/** – zdrojové kódy implementace v jazycích Kotlin a C/C++
  - **flutter_sdk/** – zdrojové kódy implementace v jazyku Dart
  - **thesis/** – zdrojová forma práce ve formátu TEX
- **thesis.pdf** – text práce ve formátu PDF