САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-механический факультет
Кафедра системного программирования

Шапошников Алексей Игоревич

Энергопрофилирование многопоточных Android приложений

Курсовая работа

Научный руководитель :
ст. преп. С. Ю. Сартасов

Санкт-Петербург, 2020г.

Содержание

Введение

Смартфоны, планшеты и иные мобильные устройства стали повсеместным явлением в современном мире. Уже в 2016 году число пользователей смартфонов составляло 2.5 миллиарда людей, а к 2021 ожидается рост до 3.8 миллиардов[1]. Но мобильные устройства требуют энергоресурсов — чем сложнее устроены приложения, тем быстрее разряжается батарея, и время работы смартфона сокращается. Есть несколько подходов к решению этой проблемы, и среди них:

- Физическое увеличение батареи
- Увеличение энергоэффективности работы компонент девайсов (процессора, работы с памятью, периферийных устройств)
- Разработка приложений, потребляющих как можно меньше энергии

На разряд также влияют такие факторы, как параллельное выполнение программ или приложений, внешние переменные и износ батареи ввиду срока жизни устройства.

В последнее время рынок электронных устройств развивался, стали распространены литиумные батареи с большей вместимостью, появилась новая архитектура CPU (big.LITTLE2), что положительно сказалось на времени жизни современных девайсов. Однако было выявлено, что улучшения батареи и технологий в области аппаратных средств имеют предел[2]. Поэтому разработчикам приходится уделять пристальное внимание улучшению работы самих приложений.

Но как узнать, сработала оптимизация или нет, стал ли продукт более энергоэффективным? Для этого разработчики ис-

пользуют профилирование — сбор характеристик работы приложения. Но измерить потребление ресурсов в течение какого-то времени и связать с протекающими в это время операциями — не самая тривиальная задача.

Стоит также учитывать, что на сложность профилирования оказывает влияние и многопоточность — разделение задач, реализуемых приложением, на цельные блоки, которые могут одновременно исполняться и работать с информацией.

Поэтому мы, группа студентов СПБГУ направления Программная Инженерия под руковдством старшего преподавателя Станислава Юрьевича Сартасова, решили изучить существующие решения и подходы анализа энергопотребления приложений для мобильных устройств и реализовать утилиту, которая поможет отследить энергоэффективность программного кода, имеющего возможность многопоточного исполнения.

1 Цели и задачи

Так как над фреймворком работала команда (5 человевк), мы решили также получить опыт разработки с гибкой методологией, пользуясь некоторыми принципами scrum[3]: С. Ю. Сартасов исполнял роль scrum master, помогая улучшать эффективность работы develeopment team (студентов) с помощью устранения препятствий, рекомендаций и мотивирования, на еженедельных встречах обсуждался прогресс по проекту у каждого студента, решался вопрос о том, кто берет (или продолжает) какие задачи с доски проекта[4].

Целью данного проекта являлась разработка фреймворка, который поможет или будет использоваться для профилирования энергопотребления. Для достижения этой цели передо мной были поставлены (распределены) следующие задачи:

- Сделать обзор решений по инструментовке кода в существующих решениях для энергопрофилирования
- Реализовать инструментирующий Gradle плагин
- Интегрировать разработки других участников development team
- Разработать и реализовать API плагина в виде gradle задач (скриптов)
- Добиться воспроизводимости плагина на других машинах (написать инструкцию)
- Опубликовать плагин для доступа пользователям

2 Обзор предметной области

2.1 Существующие решения

Для поиска существующих решений по профилированию энергопотребления использовался сервис “Google Scholar”. Из множества работ было выбрано несколько подходящих решений, в которых используются разные подходы к замеру энергии.

Можно выделить два основных подхода к оценке энергопотребления программного обеспечения: прямое и косвенное измерение мощности.

Первый из них заключается в получении прямых замеров тока, напряжения или мощности с помощью внешнего устройства (мультиметра) или встроенных в устройство датчиков. Такой подход легко может быть расширен на случай многопоточного исполнения путем разделения трассы выполнения программы по идентификаторам потоков. Стоит учитывать, что при использовании внешнего мультиметра необходимо синхронизировать часы смартфона и этого устройства до начала замеров.

Второй метод основывается на использовании модели энергопотребления, содержащей некоторые коэффициенты потребления мощности определенных инструкций. Такой подход позволяет оценить энергопотребление тестового прогона приложения без запуска на реальном устройстве. Однако одного коэффициента для каждой инструкции недостаточно для точной оценки в многопоточном случае, что значительно усложняет вычисление модели.

Для подробного обзора было выделено несколько характерных решений с применением разных подходов и наиболее удачной реализацией.

2.1.1 GreenDroid

GreenDroid[5] — инструмент для замера и анализа энергопотребления устройств Android. Он использует модель Power Tutor[6] — отдельное приложение, которое показывает текущее потребление энергии смартфоном. В свою очередь Power Tutor работает по следующему принципу: каждому компоненту смартфона (CPU, Display, GPS, Wi-Fi и т.д.) сопоставляет определенные переменные статусы. Эти переменные характеризуют возможные состояния компонентов. Таким образом, сложив их и умножив на некоторые коэффициенты, полученные в результате запуска тестовых сценариев на данном устройстве, можно получить приблизительный расход энергии на текущий момент времени. GreenDroid выделяет API Power Tutor для взаимодействия с другими частями программы. В начало и конец каждого метода вставляются вызовы соответствующих методов для старта и завершения замеров (иначе говоря, осуществляется инструментовка). Это производится с помощью Java фреймворка javaparser[7]: файлы с расширением .java рабочей директории парсятся с помощью ферймворка, а затем в начало и конец методов вставляются инструкции; тоже самое делается с тестами. Далее на инструментированных тестах запускается приложение и генерирутеся несколько графиков и диаграмм для визуализации результатов.

GreenDroid работает только с Java файлами и не поддерживает Kotlin, который широко используется в современной мобильной разработке. Кроме того, последние коммиты в master ветке официального репозитория были сделаны 3 года назад, что свидетельствует о прекращении активной поддержки инструмента. Также модель Power Tutor (которая, к слову, последний

раз была обновлена в 2011 году) ориентирована под смартфоны HTC G1, HTC G2 и Nexus One, а на остальных данные об энергии могут быть довольно неточными.

2.1.2 PETrA

PETrA[8] — инструмент для профилирования энергии, разработанный в 2017 году. В отличие от предыдущего решения, где использовалась некоторая модель, здесь применяется подход, основанный на данных из встроенных в систему функциональностей для определения потребления энергии, а именно, Project Volta. Это официальное расширение для разработчиков Android, которое предоставляет инструменты, показывающие историю событий аккумулятора, общую статистику по работе устройства и т.д. PETrA полагается на данные dmtracedump (эта утилита создает call-stack диаграмму из трассы работы приложения), BatteryStats (для определения состояния каждого компонента в конкретный момент времени) и Systrace (для анализа изменения частоты процессора).

Существенным минусом данного инструмента является отсутствие открытого исходного кода, а также каких-либо данных о последующей разработке. Это свидетельствует о том, что данный инструмент так и не вышел за пределы исследовательской работы. Кроме того, PETrA собирает информацию из разных утилит Project Volta, а затем их объединяет, что может привести к уменьшению точности относительно маленьких временных интервалов; этот инструмент также не поддерживает профилирование нескольких потоков, а для использования требуется скачать jar и запустить через командную строку (не самый удобный способ).

2.1.3 ANEPROF

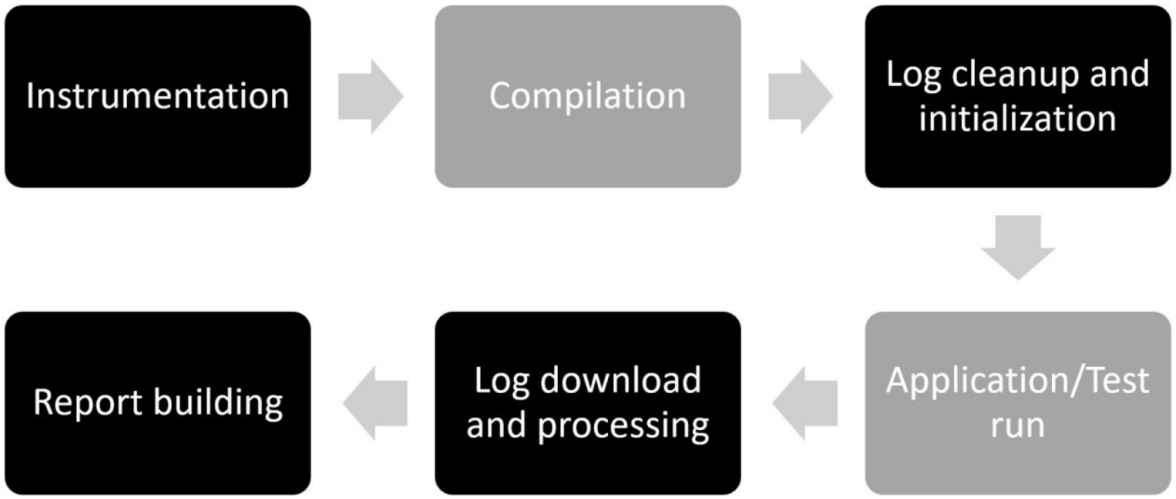
В ANEPROF[9] для профилирования энергии используется подход, основанный на прямых замерах с помощью тестового стенда PXA270 с Android 2.0. Для получения данных стенд подключается к считывателю данных NI DAQ. В OS scheduler кладутся коды инструментовки, чтобы сохранять системное время и id процесса, когда происходит переключение контекста (такая инструментовка часто вызывает накладные расходы); информация о времени начала методов извлекается через Android TraceView, также используется инструмент DVMTI для извлечения нужной информации и исключения не относящейся к непосредственно вызову методов (например, управление кучей и сборка мусора). Данные замеров энергии и трассы программы коррелируются между собой после синхронизации по времени, после чего генерируется отчет, показывающий энергопотребление методов в джоулях и процентах.

Этот инструмент не имеет открытого исходного кода и был разработан еще в 2011 году, поэтому непонятно, как он будет работать на современных устройствах с более новыми версиями ОС Android. Кроме того, может возникнуть проблема синхронизации часов, если данные о потреблении энергии получаются на одном девайсе, а трасса программы — на другом.

2.2 Navitas Framework

После изучения существующих решений и подходов к профилированию энергоэффективности приложений, мы решили создать собственный инструмент для энергопрофилирования, который учтет недостатки предыдущих работ, будет легко-подключаемым, с открытым кодом для доработки и возможностью дальней-

шей интеграции; для этого было решено реализовать Navitas Framework со следующей архитектурой:



- **Instrumentation** — начальный этап профилирования, подразумевающий вставку вызовов нужных нам методов (создание логов о начале и конце выполнения метода, информации об активности ЦПУ и состоянии других компонент, последнее еще находится в разработке) в начало и конец методов, чтобы отследить трассу программы и частоты каждого процессора с затраченным на каждую частоту временем.
- **Compilation** — компиляция исходного кода приложения, тестовых пакетов, зависимостей etc.; конвертация в android package файл, готовый к непосредственному запуску.
- **Log cleanup and initialization** — очистка предыдущих логов на устройстве, чтобы не получить лишней информации; загрузка приложения на девайс.
- **Application/Test run** — запуск всех тестовых пакетов для

обхода разных активных компонент приложения. Подразумевается, что тестовые пакеты создаются самим разработчиком и либо передаются отдельным параметром, либо берутся в дефолтном пути (см. API).

- **Log download and processing** — выгрузка созданных логов трассы программы и данных о замерах энергопотребления, обработка полученных данных в json.

Однако по ходу обзора решений и инструментов стало понятно, что на чистом языке программирования тяжело и громоздко реализовывать интеграцию перечисленных выше модулей. Без использования существующих технических решений даже стадии компиляции и инструментовки представляют большую сложность в рамках учебной практики, поэтому был произведен поиск актуальных инструментов, способных помочь в реализации проекта Navitas Framework.

В итоге было решено реализовывать Gradle плагин, который и позволит создать и соединить модули архитектуры с помощью скриптов.

3 Инструменты

В данной секции описаны программные инструменты, которые использовались в процессе разработки.

3.1 Gradle

Gradle[10] — open-source система автоматической сборки, фокусирующаяся на гибкости разработки и функциональности. Gradle скрипты сборки могут быть написаны как с помощью Groovy, так и Kotlin DSL. Мы остановились на Gradle в виду того, что он полностью поддерживается в Android Studio (а значит, будет работать для каждого проекта андроид приложения) и имеет большой потенциал создания пользовательских плагинов, так как облегчает процесс взаимодействия пользовательских задач (task-ов) с стандартными задачами сборщика (e.g. buildDebug, assemble, classes, jar, testClasses). Благодаря этому возможно создание сложных скриптов с последовательной структурой вызова задач (что актуально для нашего проекта).

3.2 Android Debug Bridge

Android Debug Bridge (adb)[11] — универсальный инструмент командной строки, позволяющий взаимодействовать с мобильным устройством и предоставляющий доступ к оболочке Unix, которую можно использовать для запуска различных команд на устройстве. adb входит в пакет Android SDK Platform-Tools, который можно загрузить с помощью SDK Manager.

3.3 Kotlin

Kotlin — статически-типизированный язык программирования, разрабатываемый компанией JetBrains. Kotlin направлен

на JVM, но также может быть скомпилирован в JavaScript или машинный код. Kotlin был выбран как основный язык фреймворка, т.к. Google объявили Android разработку
Kotlin-ориентированной, что подразумевает поддержку нововведений Kotlin'ом приоритетнее Java[12]. Кроме того, Kotlin DSL является заменой Groovy как скриптового языка программирования в системе сборки Gradle.

3.4 Transform API

Transform API[13] — библиотека, которая включена в Gradle plugin с версии 1.5.0-beta1; она позволяет сторонним плагинам манипулировать скомпилированными .class файлами до их конвертации в .dex файлы (они содержат код, который потом непосредственно будет исполнен Android Runtime). Такой процесс манипулирования называется инструментовкой, и в нашем плагине мы реализовали его с помощью библиотеки Javassist.

3.5 Javassist

Javassist[14] — библиотека для изменения байткода (скомпилированного кода) Java (соответственно, и Kotlin). Она предполагает два уровня программного интерфейса: source level (подразумевает работу с исходным кодом проекта) и bytecode level (i.e. работу с упомянутыми .class файлами). Для реализации плагина была выбран байткод уровень, т.к. создание специальной инструментованной сборки, не влияющей на исходное приложение, предпочтительнее для пользователя. Javassist предоставил возможность обхода всех классов, добавления в начало и конец каждого метода вызовов логирования и замеров частоты центрального процессора.

4 Реализация

4.1 Инструментирующий плагин

Navitas Framework начинался как проект в рамках летней школы Ланит-Терком под руководством С. Ю. Сартасова. После обзора подходящих для нашей задачи решений по профилированию энергопотребления я решал задачу инструментовки. Изначально с помощью библиотеки Javassist был реализован javaagent (отдельный файл, который подключается через указание в файле-манифесте и параметр командной строки, и позволяет добавить инструментовку байт-кода)[15].

Но далее оказалось, что такой вариант не подходит, потому что javaagent позволяет инструментовать только скомпилированные jar файлы, а в случае андроид-приложений процесс сборки сложнее. В нем исходный код компилируется и конвертируется в dex файлы и вместе с подгруженными библиотеками и ресурсами упаковывается в apk (android package) файл.

В частности, поэтому было решено перейти на архитектуру Gradle-плагина, который позволяет гибко вмешиваться в скрипты автоматической сборки; следом были изучены способы создания пользовательского Gradle плагина[16] и использования TransformAPI для манипулирования байткодом[17].

Итого, к концу летней школы был реализован ProfilingPlugin[18], который при запуске скриптов автоматический сборки с суффиксом “build” инструментировал полученный скомпилированный код методами вызовов логирования. Дизайн плагина подразумевал создание локального репозитория (с помощью задачи “uploadArchives”), указание его в зависимостях project-уровня *build.gradle* и указания пути к локальному репозиторию на дис-

ке.

4.2 NaviProf

С началом учебного года были переписаны в Kotlin DSL (или созданы) и интегрированы в существовавший *ProfilingPlugin* следующие модули: загрузка приложения на устройство, очистка логов, запуск тестовых пакетов, считывание данных (логирование и cpuFreqs при вызове и выходе каждого метода), затем их вывод в txt файлы и парсинг в json с информацией о всех вызванных методах и активности центрального процессора с временными метками. Предполагалось, что плагин будет предоставлять пользователю возможность автоматизированного профилирования с выходными сырыми данными, которые в дальнейшем могут быть обработаны или визуализированы. Для примера смотрите работу другого участника команды, Владислава Мясникова[19].

Отдельной проблемой по созданию скриптов стал интерфейс работы с *Tasks* и *TasksContainer* в методе "apply" плагина: пользовательские задачи должны быть зарегистрированы, а не созданы (иначе в момент инициализации плагина эти задачи действительно будут сконфигурированы, что может привести к проблемам, например, если мобильное устройство не найдено — и синхронизация gradle-файлов закончится ошибкой). "tasks.register" же позволяет сделать ссылку (reference) на задачу, а создавать копию только при непосредственном вызове скрипта[20].

Так как проект предоставляет методы для профилирования, то было решено один сложный и целостный скрипт превратить в последовательность нескольких для более гибкой функциональности и удобства пользователя. И так, я разделил рабо-

ту плагина на более мелкие задачи с разными конфигурациями (зависимостями и принимаемыми параметрами), и родился пользовательский интерфейс **NaviProf** (новое название уже более целостного gradle-плагина):

>**profileBuild** — собирает *app-debug.apk* и *androidTest.apk* с инструментовкой байткода и устанавливает их на девайс. Не требует параметров.

>**runTests** — запускает все методы в указанных тестовых классах для стандартного пути к *test_ apk_ path*.

- параметр *test_ paths* принимает имена тестовых классов (разделенные через ','), тесты из которых должны быть запущены
- если параметр *granularity* установлен на "class" или не указан, то выходные данные будут поделены по названиям тестовых классов
- если параметр *granularity* установлен на "methods" , то выходные данные будут поделены по каждому тестовому методу и необходимо после названия тестового класса указать '#' и перечислить названия тестовых методов через ':'.

>**runCustomTests** — реализует ту же функциональность, что и скрипт "**runTests**" , но принимает параметр *test_ apk_ path*, в котором указывается путь к тестовому apk, тесты которого будут вызываться.

>**defaultProfile** — это задача полного цикла, включающая

в себя компилирование с инструментовкой и упаковку, установку на девайс, запуск тестов, выгрузку логов и обработку их в json файл. Apk приложения и тестовое apk генерируется из *src* директории модуля. Этому скрипту нужно передавать только один параметр - *test_paths* (и при необходимости *granularity*), чтобы определить необходимые для запуска тесты.

>**customProfile** — в отличие от "**defaultProfile**" , он не компилирует и не устанавливает apk на устройство; предполагается, что пользователь выбрал и установил нужные apk и путь к ним нужно передать параметрам *apk_path* и *test_apk_path* (остальное аналогично предыдущему таску).

Перечисленные команды запускается с помощью "./gradlew :{appModule}:{chosenTask}" , где указывается модуль приложения и нужная задача. Выходные данные генерируются в директории модуля в папке "profileOutput". Для более подробной документации, можно обратиться на github[21].

4.3 Публикация плагина

В определенный момент стало очевидно, что перемещать локальный репозиторий с плагином в проект и указывать к нему путь некрасиво, долго, не легко доступно для всех пользователей. Поэтому я решал задачу размещения NaviProf на удаленном сервере.

Сначала я реализовал публикацию на github-packages с помощью плагина 'maven-publish' — это предоставило возможность загружать и скачивать файлы раздела *package* репозитория на github[22]. Но это могло быть лишь временным решением, по-

тому как требовало аутентификации на github через имя пользователя и github-токен не только для загрузки пакетов, но и для скачивания.

Вследствие, я перешел к реализации публикации плагина на Maven Central, один из самых популярных репозиториев, в том числе и для плагинов. Однако в него нельзя напрямую загружать артефакты, сначала их необходимо опубликовать в подтвержденный (approved) репозиторий. Для этого был выбран Sonatype OSS Nexus[23], бесплатный для опен-сорс проектов подтвержденный репозиторий от той же компании, что управляет Maven Central.

Чтобы в свою очередь загрузить артефакт на OSS Nexus, нужно зарегистрировать в Sonatype и подать заявку на Jira. Важный момент, что namespace (название организации) должно быть подтвержденным — необходимо либо владеть доменом, либо являться хозяином репозитория, чтобы использовать название в качестве *group id*. Так мы остановились на "com.github.stanislav-sartasov:NaviProf" вместо "com.Navitas-Framework:NaviProf". Для загрузки, закрытия и релиза артефакта (при всех пройденных проверках) через командную строку использовался плагин 'com.bmuschko.nexus'[24].

Подробная инструкция для разработчика по публикации нашего плагина находится так же в его документации на github[21]. Пользователю же для подключения необходимо иметь mavenCentral в списке репозиториев *build.gradle* и прописать в зависимостях "classpath 'com.github.stanislav-sartasov:NaviProf:1.19' " а затем просто подключить в нужном модуле — "apply plugin: 'NaviProf'".

4.4 Ограничения и дальнейшее развитие

Для работы плагина необходима актуальная для android tools версия Gradle[25], корректный Android SDK Platform-Tools, запускающиеся на устройстве тестовые пакеты.

Однако на данный момент (версия 1.19) плагин не поддерживает версии "com.android.tools.build:gradle"выше 3.4.2: при вызове тестовых методов через командную строку появляется ошибка "java.lang.ClassNotFoundException: Didn't find class "androidx.appcompat.R$drawable". Для этой ошибки наиболее частый рабочий ответ был откат до стабильный версии Android Gradle plugin. В дальнейшем планируется решить эту проблему.

И говоря о развитии проекта, также планируется протестировать плагин на различных android-девайсах для анализа данных о поведении на различных моделях и версиях android (что неосуществимо в настоящее время в условиях карантина), добиться работоспособности для наибольшего количества устройств, научиться извлекать из файлов устройства power_profile.xml — файл с указанными константами потребления энергии разными модулями (wi-fi, bluetooth, etc.) для конкретной конфигурации оборудования устройства. А также планируется извлекать данные о состоянии телефона: геолокация, bluetooth, wi-fi; это позволит более точно показать потребление энергии на телефоне (в акутальной версии считываются частоты всех процессоров и яркость экрана).

Заключение

В результате курсовой работы были выполнены следующие задачи:

- Произведен обзор существующих решений и методологий по уменьшению энергопотребления и инструментовке кода
- Разработана архитектура и первая версия инструментирующего Gradle плагина
- Интегрированы модули участников команды Navitas Framework и реализован скрипт полного цикла профилирования
- Реализован более гибкий пользовательский интерфейс плагина и написана документация
- Реализовано несколько способов публикации плагина и написана инструкция

Незаконченной осталась задача поддержки NaviProf при последних версиях Android Gradle plugin (3.5.* и выше). Также видны пути дальнейшего улучшения и развития проекта Navitas Framework, включающие добавление новых компонент в анализ энергопотребления, создания новых видов отчетов и способов указания значений power_profile.xml (для плагина Android Studio, который тоже разрабатывался в нашей команде и подключает и использует Gradle-плагин NaviProf).

Но в результате курсовой работы я также получил опыт работы в команде, коммуникации и решении вопросов с другими разработчиками, опробовал некоторые из принципов agile разработки, такие как еженедельные собрания с обсуждением прогресса и дальнейших планов, разделение ответственности, обсуждение сложности и распределение текущих задач.

Список литературы

[1] Statista Inc. Number of smartphone users worldwide from 2014 to 2020 (in billions). https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, 2020. Дата обращения 3.06.2020.

[2] Jason Flinn and M. Satyanarayanan. Energy-aware Adaptation for Mobile Applications. *SIGOPS Oper. Syst. Rev.*, 34(2):13–14, April 2000.

[3] Scrum. https://ru.wikipedia.org/wiki/SCRUM. Дата обращения 3.06.2020.

[4] Navitas Framework's tasks board. https://github.com/Stanislav-Sartasov/Navitas-Framework/projects/1. Дата обращения 3.06.2020.

[5] Marco Couto, Jácome Cunha, João Fernandes, Rui Pereira, and Joao Saraiva. GreenDroid: A tool for analysing power consumption in the android ecosystem. pages 73–78, 11 2015.

[6] PowerTutor. http://ziyang.eecs.umich.edu/projects/powertutor/index.html. Дата обращения 3.06.2020.

[7] Javaparser. https://code.google.com/archive/p/javaparser/. Дата обращения 3.06.2020.

[8] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea Lucia. PETrA: A Software-Based Tool for Estimating the Energy Profile of Android Applications. 05 2017.

[9] Yi-Fan Chung, Chun-Yu Lin, and Chung-Ta King. ANEPROF: Energy Profiling for Android Java Virtual Machine and Applications. *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, pages 372–379, 12 2011.

[10] Gradle. `https://gradle.org/`. Дата обращения 3.06.2020.

[11] Android Debug Bridge. `https://developer.android.com/studio/command-line/adb`. Дата обращения 3.06.2020.

[12] Google I/O 2019. https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html. Дата обращения 3.06.2020.

[13] Transform API. `http://google.github.io/android-gradle-dsl/javadoc/3.1/com/android/build/api/transform/Transform.html`. Дата обращения 3.06.2020.

[14] Javassist. `https://www.javassist.org/html/javassist/package-summary.html`. Дата обращения 3.06.2020.

[15] Instrumenting Java agent. `https://github.com/Stanislav-Sartasov/Navitas-Framework/tree/code_instrumentation/Instrumentation`. Дата обращения 3.06.2020.

[16] Writing Gradle Plugins. `https://docs.gradle.org/current/userguide/custom_plugins.html`. Дата обращения 3.06.2020.

[17] Grandcentrix. Using the Transform API for byte code manipulation. https://github.com/grandcentrix/LogALot-TransformAPI-sample. Дата обращения 3.06.2020.

[18] Navitas framework. https://github.com/Stanislav-Sartasov/Navitas-Framework. Дата обращения 3.06.2020.

[19] Android Studio Navitas-Plugin. https://github.com/Stanislav-Sartasov/Navitas-Framework/tree/master/Navitas-Plugin. Дата обращения 3.06.2020.

[20] Task Configuration Avoidance. https://docs.gradle.org/current/userguide/task_configuration_avoidance.html. Дата обращения 3.06.2020.

[21] NaviProf documentation. https://github.com/Stanislav-Sartasov/Navitas-Framework/tree/master/NaviProf. Дата обращения 3.06.2020.

[22] NaviProf package. https://github.com/Stanislav-Sartasov/Navitas-Framework/packages. Дата обращения 3.06.2020.

[23] Sonatype OSS Nexus. https://www.sonatype.com/nexus-repository-oss. Дата обращения 3.06.2020.

[24] Gradle docker plugin. https://bmuschko.github.io/gradle-docker-plugin/. Дата обращения 3.06.2020.

[25] Android Gradle plugin. https://developer.android.com/studio/releases/gradle-plugin. Дата обращения 3.06.2020.