САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-механический факультет
Кафедра системного программирования

Гордиенко Егор Александрович

Энергопрофилирование многопоточных Android приложений

Курсовая работа

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

Санкт-Петербург, 2020 г.

Содержание

Введение

Смартфоны, планшеты и иные мобильные устройства стали повсеместным явлением в современном мире. Уже в 2016 году число пользователей смартфонов составляло 2.1 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 руководством Станислава Юрьевича Сартасова, решили изучить существующие решения и подходы анализа энергопотребления приложений для мобильных устройств и реализовать плагин для Android Studio, который поможет отследить энергоэффективность программного кода, имеющего возможность многопоточного исполнения.

1 Цели и задачи

Целью нашей работы является разработка плагина по профилированию энергопотребления для официальной среды разработки под ОС Android - Android Studio.

Для достижения общей цели передо мной были поставлены следующие задачи:

- Сделать обзор решений по уменьшению энергопотребления в ОС Android
- Реализовать Gradle-плагин для оценки энергопотребления, который будет интегрирован в Android Studio-плагин
  - Научиться получать информацию о работе процессора на разных частотах
  - Обеспечить корректную инструментовку исходного кода тестируемого приложения
  - Обработать логи о потреблении энергии и представить их в удобном формате для последующей интеграции с Android Studio плагином
- Добавить в Android Studio плагин возможность загружать Power Profile, файл со значениями энергии различных компонентов смартфона, и использовать их в вычислении общих энергозатрат

2 Обзор предметной области

2.1 Обзор существующих решений

Для поиска существующих решений использовался сервис “Google Scholar”. Из множества работ было выделено несколько подходящих решений, в которых используются разные подходы к измерению энергопотребления.

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. 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.

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

2.1.1 GreenDroid

GreenDroid[3] — инструмент для замера и анализа энергопотребления устройств Android. Он использует модель Power Tutor[4] — 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 статуса. Эти переменные характери-

зуют 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 путем разбора файлов исходного кода приложения, которое требуется проанализировать. Стоит отметить, что создатели данного инструмента подразумевают, что тестовые сценарии также будут предоставлены разработчиками. 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[5] — 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 функци-

ональностей 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 (для логирования входов и выходов из методов), 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.

2.1.3 ANEPROF

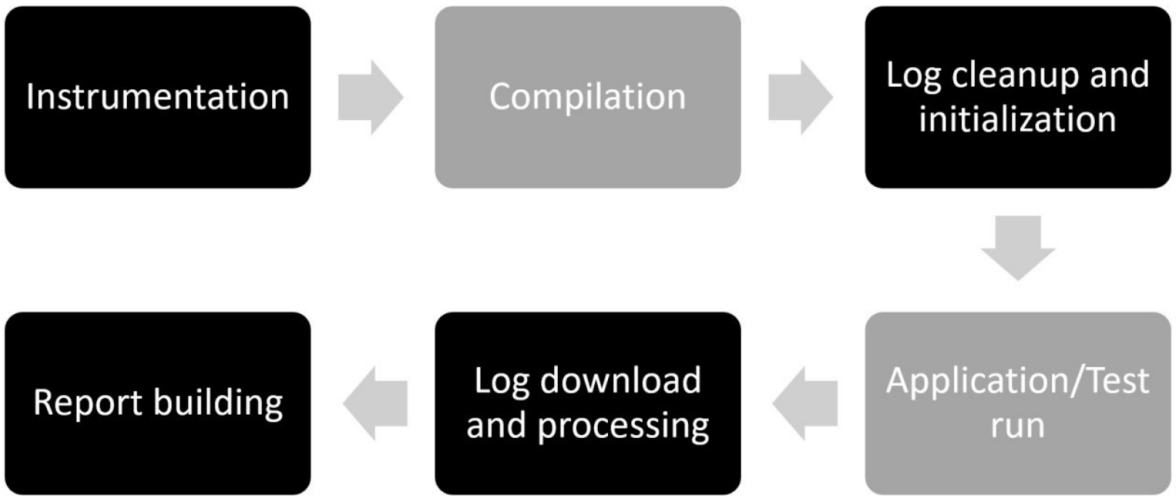
В ANEPROF[6] для профилирования энергии используется подход, основанный на прямых замерах с помощью тестового стенда PXA270 с Android 2.0. Для получения данных стенд подключается к считывателю данных NI DAQ. Информация о времени начала методов извлекается через Android TraceView, также используется инструмент 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, было решено реализовать фреймворк со следующей архитектурой:



- **Instrumentation** — начальный этап профилирования, подразумевающий вставку логирования состояния компонентов устройства в начало и конец методов для отслеживания трассы программы.
- **Compilation** — компиляция исходного кода приложения, тестовых пакетов, зависимостей, и т.д., а также конвертация в APK файл, готовый к непосредственному запуску.
- **Log cleanup and initialization** — очистка предыдущих логов на устройстве, 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 передаются как параметры запуска профилирования.
- **Log download and processing** — выгрузка созданных логов трассы программы и данных о замерах энергопотребления, обработка полученных данных.
- **Report building** — отображение результатов, например, в виде графиков, на которых будет показано энергопотребление девайса в каждый момент времени и корреляция с трассой программы.

Однако по ходу работы в летней школе стало понятно, что на чистом Kotlin 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 фреймворка.

В итоге было решено создать Gradle плагин, который и позволит реализовать и соединить модули архитектуры с помощью скриптов. Впоследствии, для более удобного использования фреймворка, Gradle-плагин будет интегрирован в Android Studio плагин, разработкой которого занимался другой член команды.

3 Инструменты

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

3.1 Gradle

Gradle[7] — 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 ввиду 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 со стандартными задачами сборщика. Благодаря этому возможно создание сложных скриптов с последовательной структурой вызова задач.

3.2 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 приоритетнее Java[8]. Кроме того, Kotlin DSL является заменой Groovy как скриптового языка программирования в системе сборки Gradle.

3.3 Android Debug Bridge

Android Debug Bridge (adb)[9] — универсальный инструмент командной строки, позволяющий взаимодействовать с мобильным устройством и предоставляющий доступ к оболочке Unix, которую можно использовать для запуска различных команд на устройстве.

3.4 Logcat

Logcat[10] — инструмент командной строки, который выводит журнал системных сообщений, включая трассировку стека, когда устройство выдает ошибку, и сообщения, которые могут быть написаны из пользовательского приложения с помощью класса Log.

3.5 Transform API

Transform API[11] — 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.

3.6 Javassist

Javassist[12] — библиотека для изменения байткода JVM. Она предполагает два уровня программного интерфейса: source level (подразумевает работу с исходным кодом проекта) и bytecode level (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.

3.7 power_profile.xml

power_profile.xml[13] — XML-файл со значениями мощности компонентов телефона. Производители смартфонов должны предоставить данный файл в определенной директории внутри смартфона. В профиле мощность указывается в миллиамперах (мА) потребляемого тока при номинальном напряжении.

3.8 Android Studio

Android Studio — официальная интегрированная среда разработки под ОС Android. Было решено создать плагин для Android Studio с удобным интерфейсом и интегрировать в него ранее описанный Gradle-плагин “Navitas Framework” для получения оценки энергопотребления.

4 Реализация Gradle-плагина

4.1 Получение информации о компонентах устройства

Из всех компонентов смартфона, влияющих на потребление энергии, самым важным является процессор. Поэтому сначала было решено научиться измерять данные именно о процессоре и его ядрах. В процессе исследования возможных способов получения требуемой информации было обнаружено, что ОС Android содержит специальные файлы в директории /sys, которые показывают сведения о времени, проведенном на определенной частоте каждого ядра процессора. Такие файлы называются time_in_state и содержат набор пар вида “<частота> <время>”. Всего этих пар ровно столько, сколько различных частот поддерживает данное ядро процессора. Время указано в единицах измерения, равных 10 миллисекундам, и отсчитывается с момента установки соответствующего драйвера для измерения данных о процессоре.

Таким образом, обладая данными time_in_state, вычислить потребляемую процессором энергию довольно просто: для каждого ядра и для каждой частоты, на которой оно работает нужно взять значение времени в конце метода и в его начале. Вычитая одно из другого, мы получим время, которое проработал метод на каждой из частот. Далее, располагая данными из файла power_profile.xml, можно умножить это время на значения констант (считая номинальное напряжение равным единице), и получить общую энергию, затраченную процессором на выполнение измеряемого метода.

Общие энергозатраты состоят не только из работы процессора. Также большое влияние оказывают яркость экрана, Wi-Fi,

Bluetooth, камера и т.д. Получением замеров данных остальных компонентов в нашей команде занимался другой человек, однако возникли некоторые сложности, и поэтому на текущем этапе работы во внимание берется только работа процессора.

4.2 Инструментовка исходного кода приложения

Изменение исходного кода приложения производилось с помощью библиотек Transform API и Javassist. Чтобы трансформировать код, нужно создать класс с реализацией интерфейса Transform и зарегистрировать его. Основная логика инструментовки происходит в переопределенном методе transform. Для каждого инструментуемого файла в начало добавляются необходимые импорты сторонних библиотек. Далее, для всех методов каждого класса применяются методы библиотеки Javassist - insertBefore и insertAfter. В качестве параметра передается строка, содержащая код на Java. Несмотря на то, что Kotlin сейчас является основным языком разработки под Android, многие приложения все еще написаны на Java, поэтому добавление Java-кода позволяет обеспечить корректную работу исходной программы на обоих языках (потому что Kotlin полностью поддерживает Java).

Таким образом, встраиваемый код находит необходимые файлы time_in_state в файловой системе Android и передает данные оттуда в Logcat. Logcat удобен тем, что при создании лога, кроме самого сообщения, отображается информация о текущем времени (вплоть до миллисекунд), идентификаторе процесса, а главное - идентификаторе потока.

Стоит отметить, что в процессе получения данных из Logcat я столкнулся со следующей проблемой: по умолчанию на неко-

торых телефонах размер буфера ограничен 64Кб. В зависимости от размера тестируемой программы, количество созданных логов также меняется, поэтому может произойти заполнение всего буфера, и часть логов не сохранится. Для решения проблемы необходимо на смартфоне в меню “для разработчиков” изменить размер буфера регистратора, поставив его как можно больше (для нашего тестового приложения хватило 256Кб, но для других приложений может потребоваться и выше).

4.3 Обработка логов и их представление в удобном формате

После выполнения всех тестов, необходимые логи выгружались в текстовые файлы таким образом, что для каждого тестового класса или метода создавался отдельный файл. Однако с такими “сырыми” данными неудобно работать, и поскольку было необходимо передать эти данные на сторону Android Studio плагина, где и подразумевался подсчет общей энергии с использованием констант power_profile.xml, то встал вопрос об удобном формате представления логов. Было предложено два варианта - CSV и JSON. Ввиду того, что формат CSV подразумевает хранение некоторой таблицы, он не совсем подходил для моей задачи. В то же время JSON предоставлял удобный формат с парами ключ-значение и массивами объектов.

В итоге был написан метод, проходящий по всем сгенерированным текстовым файлам и записывающий логи в общий JSON файл. На данном шаге работа нашего Gradle-плагина “Navitas Framework” завершалась и оставшаяся логика профилирования (подсчет энергии, визуализация результатов) оставалась на итоговом плагине для Android Studio.

5 Использование Power Profile для вычисления энергии

5.1 Android Studio плагин

Передо мной стояла задача реализации возможности загрузки пользовательского файла power_profile.xml и использования его констант для вычисления энергии. Так как эти действия выполнялись на стороне Android Studio плагина, то сначала было необходимо разобраться с общей структурой проекта.

Итак, плагин написан на языке Kotlin, с применением таких архитектурных паттернов, как Model-View-ViewModel (MVVM) и Repository. Для подписки на события используется библиотека RxJava 2. Сам плагин имеет слоистую архитектуру. Для добавления требуемой функциональности мне потребовалось работать со следующими слоями:

- **domain** — в нем хранятся модели данных, используемые в бизнес-логике плагина. Обычно такие данные отображаются в пользовательском интерфейсе. Также здесь объявлены интерфейсы репозиториев. Я добавил класс PowerProfile, который представляет непосредственно данные из профиля и объявил репозиторий PowerProfileRepository для удобной работы с данными из PowerProfile.
- **data** — здесь хранятся модели данных, не относящиеся к бизнес-логике напрямую, но при этом используемые в работе плагина. К примеру, из таких данных могут впоследствии создаваться данные для domain-слоя. Также здесь реализованы интерфейсы репозиториев, в частности, репозитория PowerProfile.

- в слое **presentation** лежат классы, связанные с пользовательским интерфейсом. Во view содержатся UI-компоненты, распределённые на пакеты. Во viewmodel хранятся классы ViewModel. Для PowerProfile был также создан класс ViewModel, который обеспечивает взаимодействие между view и репозиторием.
- **tooling** — слой для различных классов-инструментов, в том числе и для пасера профиля.

5.2 Добавление необходимой функциональности

Для загрузки профиля я создал диалоговое окно, где пользователь может выбрать файл со своего компьютера, либо же использовать профиль по умолчанию, в котором находятся значения компонент, взятые с тестового смартфона, но в таком случае итоговые затраты энергии могут получиться неточными.

С помощью библиотеки Swing[14] я добавил необходимые для диалога компоненты, такие как JFileChooser, JButton и JPanel. После выбора профиля файл передается в PowerProfileVM, где происходит парсинг и создается объект класса PowerProfile. Далее, этот объект сохраняется в Repository, и при необходимости другие компоненты плагина берут его именно оттуда.

Для получения константы для конкретной частоты в классе PowerProfile определен метод getPowerAtSpeed. В качестве параметров он принимает номер ядра и частоту, т.к. для разных ядер диапазон частот может отличаться, как и значения энергии. В этом методе происходит поиск необходимой частоты, и если таковая есть, то возвращается значение-константа. Если же частоты в профиле не нашлось, то находятся две со-

седние, между которыми лежит требуемая частота, и между ними “строится” прямая. Коэффициент энергии берется в точке, которой соответствует искомая частота.

Таким образом, для получения общей энергии, затраченной на выполнение того или иного метода, нужно взять сумму произведений константы, соответствующей данной частоте, на время, проведенное на ней. Это время вычисляется на основе JSON файла с логами в отдельном классе-помощнике, реализация которого была задачей другого члена команды.

В итоге, с использованием power_profile.xml можно получить приблизительное потребление энергии для каждого метода тестируемого приложения.

Заключение

К концу весеннего семестра были выполнены следующие задачи:

- Произведен обзор существующих решений и методологий по уменьшению энергопотребления
- Разработана архитектура и рабочая версия Gradle-плагина[1], который модифицирует исходный код тестового приложения, замеряет данные о компонентах смартфона и отправляет их в формате JSON
- Реализован Android Studio плагин[2] и модуль к нему, позволяющий загрузить собственный Power Profile или использовать стандартный

В будущем планируются работы по добавлению новых компонентов для измерения (таких, как яркость экрана, Wi-Fi и др.), обновление зависимостей Gradle-плагина до более новых версий, улучшение интерфейса Android Studio-плагина, добавление набора профилей, из который пользователь может выбрать подходящий, сопоставление результатов замеров с предыдущими запусками, публикация плагина Android Studio в свободном доступе.

[1] https://github.com/Stanislav-Sartasov/Navitas-Framework/tree/master/NaviProf
[2] https://github.com/Stanislav-Sartasov/Navitas-Framework/tree/master/Navitas-Plugin

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 15.05.2020.

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

[3] 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.

[4] PowerTutor. http://ziyang.eecs.umich.edu/projects/powertutor/index.html. Дата обращения 15.05.2020.

[5] 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.

[6] 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.

[7] Gradle. https://gradle.org/. Дата обращения 15.05.2020.

[8] Google I/O 2019: Empowering developers to build the best experiences on Android + Play. https://bit.ly/2WTdxwQ. Дата обращения 15.05.2020.

[9] Android Debug Bridge. https://developer.android.com/studio/command-line/adb. Дата обращения 15.05.2020.

[10] Logcat command-line tool. https://developer.android.com/studio/command-line/logcat. Дата обращения 15.05.2020.

[11] Transform API. http://google.github.io/android-gradle-dsl/javadoc/3.1/com/android/build/api/transform/Transform.html. Дата обращения 15.05.2020.

[12] Javassist. https://www.javassist.org/html/index.html. Дата обращения 15.05.2020.

[13] Measuring Power Values. https://source.android.com/devices/tech/power/values. Дата обращения 15.05.2020.

[14] javax.Swing. https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html. Дата обращения 15.05.2020.