Санкт-Петербургский государственный университет

Кафедра системного программирования

Богданов Егор Дмитриевич

Влияние оптимизаций компилятора на энергопотребление Android-приложений

Курсовая работа

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

Санкт-Петербург
2020

Оглавление

Введение

В наше время смартфоны, умные часы, ноутбуки и другие умные устройства, которые упрощают жизнь человеку, все сильнее проникают в нашу жизнь. Если эта тенденция сохранится, то в ближайшем будущем подавляющее большинство населения планеты будет иметь персональный компьютер в кармане. В настоящее время самая популярная используемая в таких устройствах операционная система - это Android [23].

В настоящее время одна из главных проблем в использовании мобильных устройств - малое время работы устройств в связи с высоким расходом энергии аккумулятора, от которого питается смартфон. Чем меньше расход аккумулятора, тем дольше работоспособность устройства. Производители данных устройств пытаются адаптироваться и улучшать аппаратную и программную составляющую устройств.

Увеличение времени работоспособности смартфона достигается разными способами: производители увеличивают ёмкость батарей, команды разработчиков Android проводят улучшения работы операционной системы в плане энергоэффективности, разработчики приложений пишут энергоэффективный код, например, избавляясь от энергетических анти-шаблонов кода [2]. Но в действительности код, исполняемый средой выполнения, отличается от кода, написанного программистом, в силу оптимизаций, которые проводит компилятор, тем самым видоизменяя код. Эти оптимизации могут повлиять не только на быстродействие, но и на энергопотребление приложений. Однако, в настоящее время есть данные о том, как эти оптимизации влияют на быстродействие, но не изучено, как они влияют на энергопотребление. Поэтому целью этой курсовой работы является оценка влияния этих оптимиза-

ций на энергопотребление приложений. Результаты исследования могут помочь разработчикам Android-приложений в условиях неоднозначного выбора между быстродействием и малым энергопотреблением.

1. Цель и задачи

Цель курсовой работы - оценить влияние оптимизаций, проводимых используемым в Android Studio по умолчанию компилятором, на энергопотребление.

Для достижения цели были поставлены следующие задачи:

- Провести обзор аналогичных исследований;
- Определить используемый в Android Studio по умолчанию компилятор и выполняемые им оптимизации;
- Выяснить, какие ключи компиляции отвечают за эти оптимизации и как эти ключи используются в Android Studio;
- Сформулировать методологию оценки влияния оптимизаций на энергопотребление;
- Провести эксперименты;
- Сделать выводы о влиянии оптимизаций из полученных результатов экспериментов.

2. Обзор

2.1. История компиляторов и процесс компиляции

В современном мире Android-приложения разрабатываются на языках программирования Java и Kotlin. Программы, написанные на этих языках, исполняются на Java Virtual Machine. Операционная система Android имеет собственную среду выполнения, которая использует собственный байт-код dex [9]. Поэтому до 2018 года использовался компилятор, именуемый DX, который переводил байт-код Java в байт-код dex. Перед тем, как скомпилировать Java-классы, их обрабатывало стороннее программное обеспечение ProGuard [19]. Эта программа занималась оптимизацией, обфускацией и удалением неиспользуемых частей кода.

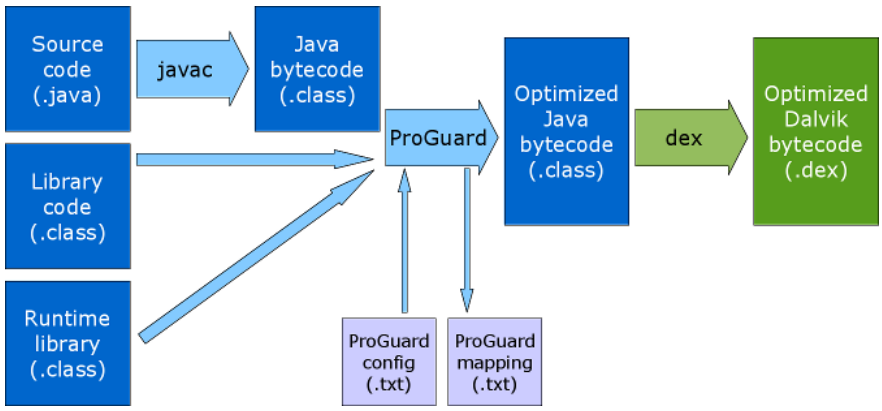

Рис. 1: Процесс компиляции Android-приложения в байт-код dex до 2018 года. [17]

Впоследствии Google выпустили новый компилятор dex под названием D8 [26], у которого время компиляции и размер итогового файла были меньше, чем у компилятора DX. Но главной особенностью нового компилятора являлся процесс компиляции, называемый desugaring, который преобразовывает ранее не поддерживаемые функции Java в байт-код, работающий на платформе Android.

Через некоторое время вышел новый компилятор dex - R8 [30]. Целью его создания было включение функциональности ProGuard в состав самого компилятора, поэтому особое внимание в нём уделялось оптимизациям, обфускации и удалению неиспользуемых частей кода. По состоянию на момент написания данной работы (2020 год) в официальной среде разработке Android-приложений Android Studio для сборки приложений по умолчанию используется R8 [1]. Также в ходе исследования предметной области были найдены компиляторы от сторонних разработчиков [27], но они не используются в Android Studio, поэтому в дальнейшем работа будет производиться с компилятором R8.

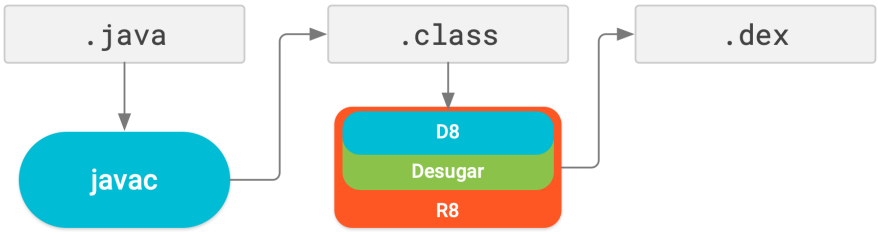

Рис. 2: Процесс компиляции Android-приложения в dex байт-код с использованием компилятора R8. [1]

2.2. Список оптимизаций

По состоянию на момент написания данной работы в документации, посвященной компилятору R8, нет полного списка и описания оптимизаций, но есть ссылка на блог Джейка Уортона [18], который рекомендован для получения большей информации о производимых оптимизациях. Также в документации отмечено, что компилятор не позволяет включать и отключать отдельные оптимизации или влиять на них. Это обосновано тем, что компилятор R8 продолжает улучшаться, поэтому поддержание стандартного поведения для оптимизаций помогает команде Android Studio устранять неполадки и решать проблемы,

с которыми пользователи могут столкнуться. Дальнейший список оптимизаций был получен путем изучения посвященных оптимизациям компилятора R8 статей в блоге Джейка Уортона.

2.2.1. Inlining

Эта оптимизация встраивает тело метода в место его вызова. Встраивание происходит, когда метод достаточно мал и/или вызывается достаточно редко, и становится выгодно скопировать содержимое тела метода на место вызова и удалить метод.

2.2.2. Staticization

Оптимизация принимает методы экземпляра, которые на самом деле не требуют доступа к экземпляру, и делает их статичными. В Kotlin отсутствуют статичные поля, методы и классы, поэтому для симуляции статических методов и полей используется конструкция языка, называемая вспомогательным объектом - одиночный экземпляр вложенного класса, инициализирующийся при загрузке класса. Компилятор анализирует байт-код вспомогательных объектов и часто может полностью исключить их, когда они используются только для симуляции статичных конструкций Java.

2.2.3. Null Data Flow Analysis

Компилятор в промежуточном представлении проверяет досягаемость всех частей графа разбора программы и удаляет ветки, на которые нельзя выйти при проверке данных на null. Если какая-либо переменная успешно прошла первую проверку на null, то дальнейшие проверки этой переменной на null между первой проверкой и изменением

значения переменной удалятся. Несмотря на то, что в Kotlin существует концепция Null safety, что позволяет при работе внутри Kotlin не использовать проверку аргументов на null, функциям можно извне передать null в качестве аргумента, пусть это не предусмотрено в объявлении функции. Поэтому для каждого аргумента происходит неявный вызов `Intrinsics.checkParameterIsNotNull`. С точки зрения сгенерированного байт-кода такие вызовы используют больше инструкций, чем при простой проверке на null. Поэтому компилятор заменяет все вызовы `Intrinsics.checkParameterIsNotNull` на обычную проверку.

2.2.4. Value Assumption

Среди списка всех правил, которые принимает R8, правило `-assumevalues` позволяет устанавливать диапазон возможных значений выбранных полей и результатов, возвращаемых методами. Это позволяет удалять однозначные проверки и недосягаемые ветки. Когда R8 удаляет эти проверки, он явно сохраняет вызов метода или чтение поля, несмотря на то, что значение не требуется. Вызов метода может вызвать некоторый другой побочный эффект, который приведет к изменению поведения в случае его удаления. Чтение поля может привести к тому, что класс будет загружен в первый раз, когда статический инициализатор может иметь побочные эффекты.

2.2.5. Method Outlining

Оптимизация применяется для последовательности инструкций, выделяя их в отдельный метод и вызывая этот метод в местах, где была эта последовательность инструкций. Она работает для последовательностей инструкций размером от 3 до 99 байт, которые встречаются не

менее 20 раз.

2.2.6. String Constant Operations

При записи строкового литерала в коде на Java или Kotlin, содержимое этой строки кодируется в специальном разделе байт-кода. Для байт-кода Java это называется Constant Pool. Для байт-кода dex это называется секцией строковых данных. В дополнение к строковым литералам, которые присутствовали в исходном коде, в этот раздел включены строки для имен типов, методов, полей и других структурных элементов. Так как эти строки являются константами, то общие строковые операции, такие как `length`, `startWith`, `indexOf`, `substring` и подобные, могут быть вычислены на этапе компиляции при условии, что их аргументы также являются константами. Но вычисление подстроки или выполнение конкатенации во время компиляции может увеличить размер секции строковых данных. Если входные строки все еще используются в других частях приложения, они не будут удалены. Новая строка, однако, всегда будет добавлена. По этой причине пока эта оптимизация не выполняется.

2.2.7. Forced Inlining

Некоторые методы слишком большие для того, чтобы R8 автоматически их встраивал, но если бы встраивание происходило, то в полученном коде могли сработать другие оптимизации. Чтобы не упустить эти оптимизации, существует правило `-alwaysinline`, которое всегда встраивает выбранный метод.

2.2.8. Enum Ordinals and Names

Каждая константа перечисляемого типа имеет методы `ordinal()`, который возвращает позицию этой константы в списке всех констант типа, и `name()`, который возвращает строку, содержащую имя константы. Если вызывать эти методы на константах, известных во время компиляции, то компилятор заменит эти вызовы на непосредственно результаты этих методов, поскольку эти результаты также известны на момент компиляции.

2.2.9. Выводы

Данные оптимизации, будучи применённые как вместе, так и по отдельности, заметно меняют код программы, и, как следствие, оптимизированные и неоптимизированные программы могут оказаться не эквивалентыми с точки зрения затрат энергии. Это служит причиной для исследования влияния этих оптимизаций на энергопотребление.

2.3. Аналогичные исследования

2.3.1. Поиск литературы

Для поиска аналогичных исследований, связанных с компилятором R8, использовался сервис “Google Scholar” со следующими поисковыми запросами:

- android compiler optimizations;
- android compiler optimizations energy consumption;
- android compiler optimizations consumption;
- android compiler optimizations energy efficiency;

- android compiler optimizations efficiency;
- android compiler energy;
- android compiler energy consumption;
- android compiler energy efficiency;
- android compiler power;
- android compiler power consumption;
- android compiler power efficiency;
- android compiler power reduction optimizations
- r8 android compiler optimizations;
- r8 android compiler optimizations energy consumption;
- r8 android compiler optimizations consumption;
- r8 android compiler optimizations energy efficiency;
- r8 android compiler optimizations efficiency;
- r8 android compiler energy;
- r8 android compiler energy consumption;
- r8 android compiler energy efficiency;
- r8 android compiler power;
- r8 android compiler power consumption;
- r8 android compiler power efficiency;

- r8 android compiler power reduction optimizations.

По каждому запросу было отобрано 100 первых работ. Если работа не подходила по названию или разделу "Abstract" (т.е. тема работы не была связана с исследованием оптимизаций компилятора R8, с их влиянием на энергопотребление или с измерением энергопотребления прочих аспектов платформы Android) или была рассмотрена ранее в другом запросе, то эта работа пропускалась [1].

2.3.2. Изучение аналогичных работ

В процессе изучения предметной области не было найдено прямых аналогов, поэтому рассматривались статьи по схожей тематике, а именно измерение энергопотребления кода программ на устройствах Android. Эти статьи рассматривались для того, чтобы выяснить методологию проводимых в них экспериментов и на основе полученных результатов сформулировать методологию собственных экспериментов. Для этого статьи исследовались на наличие ответов на следующие вопросы: какие предварительные действия со смартфоном совершались перед запуском замеров; как и где производилось считывание данных о текущем заряде батарей или расходе заряда; как происходило считывание замеров; оценивалась ли точность измерений. В этих статьях исследователи проводили 2 типа замеров - прямые и косвенные. Методы прямого измерения периодически собирают данные тока и напряжения с использованием внешних измерителей мощности или встроенных датчиков мощности. Использование внешних измерителей мощности позволяет получить энергопотребление мобильных устройств, но с их помощью сложно измерить точное энергопотребление конкретного приложения. Косвен-

[1] https://docs.google.com/spreadsheets/d/1T2jcXp9WCYAj9dV8ZuZDLVt4gXm71g1SJEhgbM1Q0pE/edit

ные замеры оценивают энергопотребление с использованием модели, которая коррелирует мощность с аппаратными датчиками производительности или другими событиями. Эти модели для оценки энергопотребления могут быть развернуты с низкими затратами и применены к коротким программам. Однако параметры этих моделей специфичны для каждого устройства, на котором происходят измерения. Ниже приведен неисчерпывающий список работ, которые показались нам интересными с точки зрения методологии исследования.

Синьбо Чен и соавт. [6] исследовали влияние на энергопотребления множества факторов, таких как язык программ, среда выполнения и компиляторы. В своей работе они представили свой инструмент для измерений, называемый Android Energy Profiler. Он основан на использовании внутренних вольтметров, данные которых сохраняются в системных файлах регистрации. Для получения информации из этих файлов использовался инструмент командной строки logcat.

Марко Коуту и соавт. [10] представили приложение, позволяющее разработчикам отследить энергопотребление отдельных частей кода своих программ. Для измерения энергопотребления исследователи создали свою модель, основывающуюся на состояниях внутренних компонентах устройства. Также они сделали приложение для автоматической калибровки этой модели для любого устройства и инструмент, позволяющий разработчикам взаимодействовать с моделью на уровне исходного кода.

В силу различных обстоятельств некоторые исследователи не проводили косвенных или прямых замеров самостоятельно и для оценки энергопотребления использовали разные профилировщики энергии. Одним из таких является PowerTutor [29] - это приложение для Android-

устройств, которое отображает мощность, потребляемую основными компонентами системы, такими как процессор, сетевой интерфейс, дисплей, приемник GPS и различные приложения. Приложение позволяет разработчикам программного обеспечения увидеть влияние изменений дизайна на энергоэффективность. Пользователи приложения также могут использовать его, чтобы определить, как их действия влияют на срок службы батареи. PowerTutor использует модель энергопотребления, построенную на основе прямых измерений во время тщательного контроля состояний управления питанием устройства. Эта модель обычно дает оценки энергопотребления в пределах 5% от фактических значений, полученных с помощью прямых замеров. Он также предоставляет пользователям вывод на основе текстового файла, содержащий подробные результаты.

Луис Коррал и соавт. [22] исследовали производительность и энергопотребление аналогичных программ, написанных на Java, Native Code и обычном C. Эксперименты проводились на Google HTC Nexus One, на котором запускались тестовые программы по 100 раз, а измерения производились с помощью PowerTutor, показания которого использовались для дальнейшего анализа.

Ахмед Хуссейн и соавт. [15] исследовали влияние работы сборщика мусора на энергопотребление устройства, а также предложили свои идеи модификации для оптимизации энергопотребления. Для измерения использовался специальный стенд APQ8074 DragonBoard Development Kit, к которому присоединили датчик тока Pololu-ACS714. Также они использовали данные, предоставляемые модулем ядра Linux CPUfreq для углубления своих результатов.

Кагри Сахин и соавт. [14] исследовали влияние обфускации исходно-

го кода на энергопотребление, а именно рассматривались примененные обфускации и количество потраченной энергии. В качестве образцов кода использовались приложения с открытым исходным кодом. Тестировались разные сценарии использования. Для схожести тестов между собой использовалось приложения для захвата и воспроизведения действий RERAN. Сами тесты производились на Nexus 3, Nexus 4, Samsung Galaxy S II и Samsung Galaxy S5, которые были подключены к сети из внешнего источника энергии и мультиметра. Данная методология вызывает сомнения в точности своих результатов, потому что приложение для захвата действий потребляет энергию, сами сценарии использования проводились вручную, а не программно, а данные датчиков показывали потребление энергии всего устройства, на что могли повлияет неописанные и незафиксированные в этой статье факторы, такие как энергопотребление сторонних и служебных приложений, включенные модули Wi-Fi и GPS и прочее.

Антон Георгиев и соавт. [13] сравнивали энергопотребление сред исполнения Android-приложений Dalvik и ART. Эксперимент проводился на Nexus 4, в качестве тестовых программ использовались тесты производительности, такие как AndEBench, Antutu Benchmark, GFXBench 3.0, Pi Benchmark, Quadrant Standard Edition, RL Benchmark: SQLite и Vellamo Mobile Benchmark, а измерения производились с помощью PowerTutor и CharM (инструмент, который отслеживает цикл разрядки аккумулятора). Устройство перед каждым тестом заряжалось и остужалось.

Также был найден проект OSCAR [27] - компилятор, который ориентирован на снижение энергопотребления кода, исполняемого на многоядерных процессорах. В то время как оптимизации R8 наравлены на

снижение объёма кода скомпилированной программы и на улучшение её быстродействия путём рефакторинга исходного кода, OSCAR имеет собственные оптимизации по автоматическому распараллеливанию кода для уменьшения затрат энергии. Создатели тестировали компилятор на полноценных программах и использовали внешний мультиметр для замеров.

2.3.3. Выводы

У обоих способов замеров есть свои недостатки: для прямых замеров необходимо разбирать устройство для подключения внешних датчиков или использовать специальные установки; в случае косвенных, нет хорошей общепризнанной модели снятия показаний. Эксперименты следует производить с помощью программных тестов, а не с помощью программ для захвата и воспроизведения действий. Также перед экспериментами необходимо минимизировать влияние внешних факторов на результаты экспериментов, например, отключить фоновые приложения, модули Wi-Fi, GPS и пр.

2.4. Методология эксперимента

2.4.1. Подготовка устройства к экспериментам

Для повышения точности экспериментов был продуман ряд мер, которые минимизируют влияние внешних факторов на результаты. Перед проведением экпериментов должны быть отключены лишние модули (Wi-Fi, 3G, GPS), отключена работа фоновых приложений.

Android основан на модифицированной версии ядра Linux, а оно в свою очередь поддерживает алгоритмы DVFS (Dynamic Voltage and Frequency Scaling). DVFS - это метод, направленный на снижение ди-

намического энергопотребления за счет динамической регулировки напряжения и частоты процессора. Этот метод использует тот факт, что процессоры имеют дискретные настройки частоты и напряжения [5, 20]. Ядро Linux поддерживает масштабирование производительности процессора с помощью подсистемы CPUFreq (CPU Frequency scaling) [4], которая состоит из трех уровней кода: ядро, регуляторы масштабирования и драйверы масштабирования.

Ядро CPUFreq обеспечивает общую инфраструктуру кода и интерфейсы пользовательского пространства для всех платформ, которые поддерживают масштабирование производительности процессора. Он определяет базовую структуру, в которой работают другие компоненты.

Регуляторы масштабирования реализуют алгоритмы для оценки необходимой мощности процессора. Как правило, каждый регулятор реализует один, возможно, параметризованный алгоритм масштабирования.

Драйверы масштабирования общаются с оборудованием. Они предоставляют регуляторам масштабирования информацию о доступных P-состояниях (пары напряжение-частота, которые задают скорость и энергопотребление сопроцессора) или диапазонах P-состояний и получают доступ к аппаратным интерфейсам платформы для изменения P-состояний ЦП в соответствии с запросами регуляторов масштабирования.

Все тесты должны быть максимально идентичны между собой и поэтому необходимо установить одну определенную частоту процессора при проведении этих тестов. Поэтому в будущих экспериментах тесты будут проводиться на максимальной частоте.

Также для улучшения точности эксперимента необходимо умень-

шить влияние опреационной системы путем проведения вычисления на одном ядре с помощью технологии привязки к процессору. В операционной системе Linux существует системный вызов `sched_setaffinity(2)`, который позволяет для данного потока определить множество ядер процессора, которым планировщик задач может назначить выполнение этого потока [21]. С его помощью в будущих экспериментов выполнение потока с тестовым кодом будет привязано к первому ядру процессора.

2.4.2. Набор тестовых программ

Качество работы компиляторов сравнивают на основании набора тестов. Этот процесс называется бенчмаркингом. Наборы тестов могут быть направлены на выявление различных особенностей работы компилятора. Например, скорость полученного кода, размер полученного кода, время работы самого компилятора. В мире Java существует несколько наборов тестов, которые широко используются в академических кругах, одними из самых влиятельных являются SPEC [32] и DaCapo Benchmark Suite [7, 8]. Также был изучен набор тестов под названием CLBG (Computer Language Benchmarks Game), который тоже использовался в предыдущих исследованиях [31].

В силу уникальности оптимизаций, производимых компилятором R8, эти наборы тестов не подошли. Наборы тестов ориентированы на интенсивное потребление ресурсов. Рассмотренные оптимизации довольно специфичны для платформы Android, потому что в приведенном выше списке лишь малая часть оптимизаций направлена на непосредственное увеличение производительности, в то время как остальная часть ориентирована на уменьшение размеров скомпилированного кода. В ходе изучения исходного кода вышеперечисленных наборов тестов

было выявлено, что ряд оптимизаций не влияет на скомпилированный код из-за отсутствия возможных случаев их применения. Например, оптимизация Inlining не применяется в коде тестов, так как код, несущий непосредственную вычислительную нагрузку, обычно расположен внутри одного метода и не содержит вызовов других методов. Оптимизация Method Outlining не применяется, так как в тестах код расположен лишь в одном классе. В вычислительных тестах, тестах на работу с памятью и подобных не использовались классы enum, поэтому влияние оптимизаций, связанных с ними, также невозможно оценить. Оптимизация Staticization применима только для кода на языке Kotlin. Сами тесты тщательно продумывались и проверялись авторами, поэтому в этих тестах отсутствуют избыточные проверки ссылок на null и оптимизация Null Data Flow Analysis тоже неприменима. Аналогично, оптимизация Value Assumption направлена на уменьшение скомпилированного кода путем удаления неиспользованного, который в тестах отсутствует.

Кроме всего прочего, из-за того, что нельзя отключать определенные оптимизации, тестируемый код должен быть ориентирован на каждую конкретную оптимизацию. Поэтому было принято решение самостоятельно реализовать тестируемый код для дальнейших экспериментов. Основной целью этого решения является реализация тестов, которые будут оптимизироваться с учетом изоляции каждой оптимизации. После проведения замеров на искусственных приложениях следует проверить влияние оптимизаций в полноценных приложениях.

2.4.3. Navitas Framework

Для измерения энергопотребления в экспериментах было принято решение использовать разработку кафедры системного программирования Navitas Framework [24]. Navitas Framework — инструмент для оценки энергопотребления Android-приложений, который выполняет инструментовку исходных кодов программы, запуск тестов на устройстве и выгрузку собранных логов на компьютер и их интерпретацию с точки зрения потреблённой энергии.

3. Эксперименты

3.1. Библиотека для измерений

В ходе проведения экспериментов было обнаружено, что Navitas Framework не совместим с устройством, на котором производились эксперименты: `power_profile.xml`, уникальный для каждой модели устройств и в котором содержатся необходимые для вычисления потреблённой энергии значения силы тока для каждой допустимой частоты, не соответствовал актуальному формату `power_profile.xml` от Google[2], с которым работает программное обеспечение. Также в процессе подготовки Navitas Framework к использованию возникал ряд технических проблем, одной из которых была инструментовка кода путём встравивания в тело каждого метода кода для замеров. Из-за того, что методы, написанные на C++ и используемые в тестах, не имели тела в файлах `.java`, Navitas Framework не мог произвести инструментовку. По этим причинам была реализована отдельная библиотека для замеров, метод снятия показаний которой идентичен методу снятия показаний Navitas Framework [25]. Соответствующий запрос об изменении был направлен команде Navitas Framework.

3.2. Описание экспериментальной установки

Эксперимент проводился на устройстве Samsung Galaxy A3 (2016) с процессором Exynos 7578, который имеет 4 ядра Cortex-A53 с максимальной тактовой частотой 1.5 ГГц. В силу схемотехнических особенностей процессора интерфейс CPUFreq имелся только у одного ядра. Также этот интерфейс имел лишь один доступный регулятор `interactive`.

[2]https://android.googlesource.com/platform/frameworks/base/+/master/core/res/res/xml/power_profile.xml

Для того, чтобы добиться одной частоты процессора в ходе проведения экспериментов, `scaling_min_freq` присваивалось значение `cpuinfo_max_freq`, тем самым сужая диапазон возможных частот процессора до одного значения.

3.3. Inlining

Был реализован тестируемый код, который представлял из себя цикл, в теле которого находилось 10 последовательных вызовов тестовой функции `calcFunc()`. Тело этой функции состояло только из вызова метода `System.currentTimeMillis()`[16].

В созданном приложении тестируемый код запускался в отдельном потоке по нажатию кнопки. Тестируемый код был реализован в 3 версиях.

Первая версия представляла собой отдельно реализованный класс `CalcTask` с интерфейсом `Runnable`. В этом класса был переопределен метод `run()`, в котором были вспомогательные функции для замеров и тестируемый код. Тестируемый код вызывал метод `calcFunc()`, определённый в теле класса `CalcTask`.

Во второй версии использовался анонимный класс. В конструктор нового потока передавался объект анонимного класса с интерфейсом Runnable, который был устроен аналогично классу из первой версии: переопределен метод `run()` и определен метод `calcFunc()`.

В третьей версии также использовался анонимный класс, но в отличие от второй версии метод `calcFunc()` вынесен из анонимного класса во внешний класс `MainActivity`. Таким образом в методе `run()` производился вызов метода `MainActivity.calcFunc()`.

В ходе испытаний было обнаружено, что при выключенных оптими-

зациях вторая версия работает быстрее третьей. При анализе сгенерированного байт-кода было выявлено, что во втором случае происходил одиночный явный вызов метода `calcFunc()`, когда в третьем случае происходила цепь вызовов искусственно сгенерированных методов, что и замедляло процесс работы. При включённых оптимизациях во второй версии вызов `System.currentTimeMillis()` встраивался в место вызова `calcFunc()`, но в третьей версии никаких подобных вставок не происходило. Таким образом третья версия в дальнейшем не рассматривалась. [3]

Рассмотренные случаи:

- Отдельный класс с включёнными оптимизациями
- Отдельный класс с выключенными оптимизациями
- Анонимный класс с включёнными оптимизациями
- Анонимный класс с выключенными оптимизациями

[3] С результатами всех экспериментов можно ознакомиться в этом документе: https://docs.google.com/spreadsheets/d/1sPdFw5B4h86HQbJ7r9PM5MkzNrIcHq7nj_FS3iY9LBc/edit

Оптимизации	выкл.	вкл.	выкл.	вкл.
Класс	отдельный	отдельный	анонимный	анонимный
Количество циклов	1000000	1000000	1000000	1000000
Тест №1, мДж	112377	111189	113223	112236
Тест №2, мДж	110826	111249	110544	110967
Тест №3, мДж	112236	112095	110967	112236
Тест №4, мДж	112095	112518	112659	112236
Тест №5, мДж	110544	111249	111672	112236
Тест №6, мДж	111390	111249	111390	111390
Тест №7, мДж	112236	112236	110544	112236
Тест №8, мДж	112377	112236	110826	112377
Тест №9, мДж	111813	111108	110967	110967
Тест №10, мДж	110544	111249	111672	111108
Тест №11, мДж	112236	112377	110967	111954
Тест №12, мДж	112095	112518	111813	112941
Тест №13, мДж	110967	111108	111249	112095
Тест №14, мДж	111108	111249	110967	111249
Тест №15, мДж	111813	112236	110967	111813
Среднее, мДж	111644	111724	111362	111869
Δ в %		-0,07		-0,46

Таким образом, установлено, что данная оптимизация слабо влияет на быстродействие приложения. Возможно, причина была в оптимизациях ahead-of-time компилятора ART: он производит анализ кода и может провести вставку методов [36, 3]. Метод `calcFunc()` в неоптимизированной версии встраивался во время работы ahead-of-time компилятора. Побочным результатом является наблюдение, что методы, с которыми работает анонимный класс, лучше прописывать в объявле-

нии анонимного класса, насколько это возможно.

3.4. Outlining

В этом эксперименте (а также и в последующих) за основу был взят код предыдущего эксперимента, а именно код класса `MainActivity`, связанного с ним `activity_main.xml`, запуск потоков и код для замеров, т.е. вспомогательный код [28].

```
for (int i = 0; i < cycles; i++) {
    arr[0] = new StringBuilder("1st mill: ")
            .append(System.currentTimeMillis())
            .append("\n2nd mill: ")
            .append(System.currentTimeMillis())
            .toString();

    ...

    arr[19] = new StringBuilder("1st mill: ")
            .append(System.currentTimeMillis())
            .append("\n2nd mill: ")
            .append(System.currentTimeMillis())
            .toString();

}
```

Listing 1: содержимое тела цикла с тестируемым кодом

В оптимизированном байт-коде dex генерируется специальный класс `GeneratedOutlineSupport`. В этом классе, помимо сгенерированного метода `outline1`, который копировал по инструкциям повторяющийся части тестируемого кода, были и другие методы, которые он вызывал в таких местах, как передача аргументов методу логирования, если аргументы в исходном коде являлись результатом конкатенации строк.

```
.method public static outline1(Ljava/lang/StringBuilder;
    Ljava/lang/String;)Ljava/lang/String;
    .registers 4

    invoke-static {}, Ljava/lang/System;->currentTimeMillis()J

    move-result-wide v0

    invoke-virtual {p0, v0, v1}, Ljava/lang/StringBuilder;
        ->append(J)Ljava/lang/StringBuilder;

    invoke-virtual {p0, p1}, Ljava/lang/StringBuilder;
        ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

    invoke-static {}, Ljava/lang/System;
        ->currentTimeMillis()J

    move-result-wide v0

    invoke-virtual {p0, v0, v1}, Ljava/lang/StringBuilder;
        ->append(J)Ljava/lang/StringBuilder;

    invoke-virtual {p0}, Ljava/lang/StringBuilder;
        ->toString()Ljava/lang/String;

    move-result-object p0

    return-object p0
.end method
```

Listing 2: байт-код сгенерированного метода outline1

В первой реализации тестов результаты оптимизированной и неоптимизированной версий не отличались в пределах погрешности. Во второй реализации количество повторяющегося кода увеличилось, из-за чего компилятор выделил его в 2 метода: часть байт-кода, которая по размеру не превышала 99 байт, была выделена в первый метод, затем результат этого метода передавался во второй метод, где располагалась оставшаяся часть повторяющегося кода. Но на разницу в результатах это также не повлияло.

Оптимизации	выкл.	вкл.
Количество циклов	10000	10000
Тест №1, мДж	70641	69372
Тест №2, мДж	69654	70218
Тест №3, мДж	71064	70218
Тест №4, мДж	69372	69795
Тест №5, мДж	70641	69795
Тест №6, мДж	69654	70359
Тест №7, мДж	70077	69654
Тест №8, мДж	70218	70218
Тест №9, мДж	73359	69936
Тест №10, мДж	70077	69795
Тест №11, мДж	71064	70077
Тест №12, мДж	69513	70218
Тест №13, мДж	69654	70077
Тест №14, мДж	70077	70641
Тест №15, мДж	70359	69372
Среднее, мДж	70162	69983
Δ в %		0,26

Возможно, причина, как и в первом эксперименте, была в оптимизациях встравивания ahead-of-time компилятора ART.

3.5. Enum Ordinal and Names

В первой реализации для этой оптимизации был добавлен класс enum с тремя элементами. Внутри тела цикла в качестве тестируемого кода были вызовы методов `name()` и `ordinal()` [12].

```
for (int i = 0; i < cycles; i++) {
    tmpstr.delete(0, tmpstr.capacity());
    tmpstr.append(Thirds.FIRST.name() +
            Thirds.SECOND.name() +
            Thirds.THIRD.name()
    );

    ...

    tmpint += Thirds.FIRST.ordinal() +
            Thirds.SECOND.ordinal() +
            Thirds.THIRD.ordinal();

    ...
}
```

Listing 3: содержимое тела цикла с тестируемым кодом

При анализе сгенерированного оптимизированного байт-кода для исходного кода было выявлено, что вместо подстановки значения методов `names()` и дальнейшей конкатенации их для передачи в метод `append()`, в метод `append()` сразу передавалась константная строка `FIRSTSECONDTHIRD`. Это означало, что оптимизация String Constant Operations оказалась включенной, хотя Джейк Уортон в своём блоге отмечал, что эта оптимизация не применяется. Поэтому в дальнейших экспериментах эта оптимизация исследовалась отдельно.

```
:goto_35
iget v7, p0, Lcom/example/eoan/MainActivity$CalcTask;->cycles:I

if-ge v5, v7, :cond_54

invoke-virtual {v0}, Ljava/lang/StringBuilder;->capacity()I

move-result v7

invoke-virtual {v0, v4, v7}, Ljava/lang/StringBuilder;
    ->delete(II)Ljava/lang/StringBuilder;

const-string v7, "FIRSTSECONDTHIRD"
```

```
invoke-virtual {v0, v7}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

invoke-virtual {v0, v7}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

invoke-virtual {v0, v7}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

add-int/lit8 v6, v6, 0x3

add-int/lit8 v6, v6, 0x3

add-int/lit8 v6, v6, 0x3

add-int/lit8 v5, v5, 0x1

goto :goto_35
```

Listing 4: оптимизированный байт-код тела цикла с включенной оптимизацией String Constatnt Operations

Во второй реализации рассматривались два класса с enum: с тремя и пятнадцатью элементами соответственно.

```
for (int i = 0; i < cycles; i++) {
    tmpstr.delete(0, tmpstr.capacity());

    tmpstr.append(Thirds.FIRST1.name())
            .append(Thirds.SECOND2.name())
            .append(Thirds.THIRD3.name());

    ...

    tmpint += Thirds.FIRST1.ordinal();
    tmpint += Thirds.SECOND2.ordinal();
    tmpint += Thirds.THIRD3.ordinal();

    ...
}
```

Listing 5: содержимое тела цикла с тестируемым кодом для enum с тремя элементами

```
for (int i = 0; i < cycles; i++) {
    tmpstr.delete(0, tmpstr.capacity());

    tmpstr.append(Fifteens.FIRST.name())
            .append(Fifteens.SECOND.name())
            .append(Fifteens.THIRD.name());

    ...

    tmpstr.append(Fifteens.THIRTEENTH.name())
            .append(Fifteens.FOURTEENTH.name())
            .append(Fifteens.FIFTEENTH.name());

    tmpint += Fifteens.FIRST.ordinal();
    tmpint += Fifteens.SECOND.ordinal();
    tmpint += Fifteens.THIRD.ordinal();

    ...

    tmpint += Fifteens.THIRTEENTH.ordinal();
    tmpint += Fifteens.FOURTEENTH.ordinal();
    tmpint += Fifteens.FIFTEENTH.ordinal();
    }
```

Listing 6: содержимое тела цикла с тестируемым кодом для enum с пятнадцатью элементами

В случае с тремя элементами в сгенерированном оптимизированном байт-коде в начале тела цикла в 3 различных регистра помещались соответственные имена элементов. Затем значения этих регистров каждый раз передавались в места, где раньше был вызов метода `name()`. В случае с пятнадцатью элементами в сгенерированном оптимизированном байт-коде перед каждым вызовом метода, аргументом которого являлся результат метода `name()`, в один и тот же регистр записывалась соответсвующая строковая константа. Поэтому из-за того, что в первом случае количество записей в регистр меньше, чем во втором, энергопо-

требление оптимизированной версии с тремя элементами меньше энергопотреблении оптимизированной версии с пятнадцатью элементами.

Оптимизации	выкл.	вкл.	выкл.	вкл.
количество элементов	3	3	15	15
количество циклов	10000000	10000000	10000000	10000000
Тест №1, мДж	295677	267054	302022	270156
Тест №2, мДж	294549	266913	300189	270015
Тест №3, мДж	294549	266913	300612	270156
Тест №4, мДж	293421	266913	300894	270297
Тест №5, мДж	294408	267054	300330	270156
Тест №6, мДж	294267	267054	300189	270156
Тест №7, мДж	293562	266913	300048	269874
Тест №8, мДж	293562	267054	299907	269874
Тест №9, мДж	293562	266913	300048	270015
Тест №10, мДж	293562	267056	299907	269874
Тест №11, мДж	293562	266772	300048	270579
Тест №12, мДж	293703	266490	300471	269874
Тест №13, мДж	293562	267195	299907	270015
Тест №14, мДж	293703	266913	300330	270156
Тест №15, мДж	293421	267195	300189	269874
Среднее потребление, мДж	293938	266960	300339	270071
Δ в %		9,18		10,08

3.6. String Constant Operations

Для тестирования этой оптимизации был создан класс с приватными строковыми константами и методами чтения для каждой из них. В теле цикла тестируемого кода вызывались эти методы и над их результатами выполнялись различные строковые вычисления по типу конкатенации, нахождения длины строки и т.п. [34]

```
for (int i = 0; i < cycles; i++) {
    tmpstr.delete(0, tmpstr.capacity());

    tmpstr.append(SCOHelper.getFirst() +
        SCOHelper.getSecond() +
        SCOHelper.getThird());

    tmpstr.append(SCOHelper.getSub() +
        (SCOHelper.getSubstr()
            .startsWith(SCOHelper.getSub()) ?
            SCOHelper.getStr() :
            SCOHelper.getSubstr()));

    tmpint += SCOHelper.getFirst().length() +
            SCOHelper.getSecond().length() +
            SCOHelper.getThird().length() +
            SCOHelper.getSubstr().length() +
            SCOHelper.getSub().length() +
            SCOHelper.getStr().length();
}
```

Listing 7: содержимое тела цикла с тестируемым кодом

В сгенерированном оптимизированном байт-коде в переменные действительно присваивался готовый результат вычислений, из-за чего размер байт-кода сильно уменьшился в сравнении с неоптимизированной версией.

```
:goto_33
iget v6, p0, Lcom/example/sco/MainActivity$CalcTask;->cycles:I

if-ge v4, v6, :cond_c6

invoke-virtual {v0}, Ljava/lang/StringBuilder;->capacity()I

move-result v6

invoke-virtual {v0, v3, v6}, Ljava/lang/StringBuilder;
    ->delete(II)Ljava/lang/StringBuilder;

new-instance v6, Ljava/lang/StringBuilder;

invoke-direct {v6}, Ljava/lang/StringBuilder;-><init>()V

invoke-static {}, Lcom/example/sco/SCOHelper;->getFirst()Ljava/lang/String;

move-result-object v7

invoke-virtual {v6, v7}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

invoke-static {}, Lcom/example/sco/SCOHelper;->getSecond()Ljava/lang/String;

move-result-object v7

invoke-virtual {v6, v7}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

invoke-static {}, Lcom/example/sco/SCOHelper;->getThird()Ljava/lang/String;

move-result-object v7

invoke-virtual {v6, v7}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

invoke-virtual {v6}, Ljava/lang/StringBuilder;
    ->toString()Ljava/lang/String;

move-result-object v6

invoke-virtual {v0, v6}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

new-instance v6, Ljava/lang/StringBuilder;

invoke-direct {v6}, Ljava/lang/StringBuilder;-><init>()V

invoke-static {}, Lcom/example/sco/SCOHelper;->getSub()Ljava/lang/String;

move-result-object v7

invoke-virtual {v6, v7}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

invoke-static {}, Lcom/example/sco/SCOHelper;->getSubstr()Ljava/lang/String;
```

```
move-result-object v7

invoke-static {}, Lcom/example/sco/SCOHelper;->getSub()Ljava/lang/String;

move-result-object v8

invoke-virtual {v7, v8}, Ljava/lang/String;->startsWith(Ljava/lang/String;)Z

move-result v7

if-eqz v7, :cond_7e

invoke-static {}, Lcom/example/sco/SCOHelper;->getStr()Ljava/lang/String;

move-result-object v7

goto :goto_82
```

Listing 8: неоптимизированный байт-код тела цикла

```
:goto_31
iget v7, p0, Lcom/example/sco/MainActivity$CalcTask;->cycles:I

if-ge v5, v7, :cond_4b

invoke-virtual {v0}, Ljava/lang/StringBuilder;->capacity()I

move-result v7

invoke-virtual {v0, v4, v7}, Ljava/lang/StringBuilder;
    ->delete(II)Ljava/lang/StringBuilder;

const-string v7, "FirstSecondThird"

invoke-virtual {v0, v7}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

const-string v7, "substr"

invoke-virtual {v0, v7}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

add-int/lit8 v6, v6, 0x1c

add-int/lit8 v5, v5, 0x1

goto :goto_31
```

Listing 9: оптимизированный байт-код тела цикла

Оптимизации	выкл.	вкл.
Количество циклов	3000000	3000000
Тест №1, мДж	113364	17061
Тест №2, мДж	113928	17343
Тест №3, мДж	115056	17766
Тест №4, мДж	115056	17625
Тест №5, мДж	116043	17061
Тест №6, мДж	115761	17061
Тест №7, мДж	115761	17202
Тест №8, мДж	115056	17202
Тест №9, мДж	121260	17343
Тест №10, мДж	115902	17202
Тест №11, мДж	116184	17484
Тест №12, мДж	115479	17202
Тест №13, мДж	115479	17061
Тест №14, мДж	111108	17202
Тест №15, мДж	113364	17061
Среднее потребление, мДж	115253	17258
Δ в %		85,03

Из-за уменьшенного количества исполняемого кода уменьшилось количество энергии, потреблённой процессором.

3.7. Enum Ordinal And Names и String Constant Operations

Случай, выявленный в эксперименте с Enum ordinal and names был рассмотрен отдельно [11]. Также, как и в предыдущем эксперименте, был объявлен класс enum с тремя элементами. В теле цикла переменной tmpstr добавлялся результат конкатенации вызовов методов `name()` у каждого элемента enum. Из-за того, что оптимизация заменила вызовы метода `name()` на константный результат, с этим константным результатом могла работать оптимизация String Constant Operations. Поэтому в сгенерированном оптимизированном байт-коде к переменной `tmpstr` вместо результата цепочки вызовов добавлялась константная строка. Из-за этого энергопотребление оптимизированной версии оказалось меньше энергопотребления неоптимизированной версии.

```
:goto_32
iget v5, p0, Lcom/example/eoansco/MainActivity$CalcTask;->cycles:I

if-ge v4, v5, :cond_1c7

invoke-virtual {v0}, Ljava/lang/StringBuilder;->capacity()I

move-result v5

invoke-virtual {v0, v3, v5}, Ljava/lang/StringBuilder;
    ->delete(II)Ljava/lang/StringBuilder;

new-instance v5, Ljava/lang/StringBuilder;

invoke-direct {v5}, Ljava/lang/StringBuilder;-><init>()V

sget-object v6, Lcom/example/eoansco/MainActivity$CalcTask$Thirds;
    ->FIRST:Lcom/example/eoansco/MainActivity$CalcTask$Thirds;

invoke-virtual {v6}, Ljava/lang/Enum;->name()Ljava/lang/String;

move-result-object v6

invoke-virtual {v5, v6}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

sget-object v6, Lcom/example/eoansco/MainActivity$CalcTask$Thirds;
```

```
    ->SECOND:Lcom/example/eoansco/MainActivity$CalcTask$Thirds;

invoke-virtual {v6}, Ljava/lang/Enum;->name()Ljava/lang/String;

move-result-object v6

invoke-virtual {v5, v6}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

sget-object v6, Lcom/example/eoansco/MainActivity$CalcTask$Thirds;
    ->THIRD:Lcom/example/eoansco/MainActivity$CalcTask$Thirds;

invoke-virtual {v6}, Ljava/lang/Enum;->name()Ljava/lang/String;

move-result-object v6

invoke-virtual {v5, v6}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

invoke-virtual {v5}, Ljava/lang/StringBuilder;
    ->toString()Ljava/lang/String;

move-result-object v5

invoke-virtual {v0, v5}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

...
```

Listing 10: неоптимизированный байт-код тела цикла

```
:goto_30
iget v6, p0, Lcom/example/eoansco/MainActivity$CalcTask;->cycles:I

if-ge v5, v6, :cond_5e

invoke-virtual {v0}, Ljava/lang/StringBuilder;->capacity()I

move-result v6

invoke-virtual {v0, v4, v6}, Ljava/lang/StringBuilder;
    ->delete(II)Ljava/lang/StringBuilder;

const-string v6, "FIRSTSECONDTHIRD"

invoke-virtual {v0, v6}, Ljava/lang/StringBuilder;
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

Listing 11: оптимизированный байт-код тела цикла

Оптимизации	выкл.	вкл.
Количество циклов	1000000	1000000
Тест №1, мДж	178909	21432
Тест №2, мДж	179350	21150
Тест №3, мДж	181326	21291
Тест №4, мДж	177660	21009
Тест №5, мДж	178506	21150
Тест №6, мДж	180621	21150
Тест №7, мДж	179493	21150
Тест №8, мДж	178083	21432
Тест №9, мДж	181326	21150
Тест №10, мДж	180057	21009
Тест №11, мДж	179775	21009
Тест №12, мДж	178788	21150
Тест №13, мДж	179916	21009
Тест №14, мДж	180198	21573
Тест №15, мДж	180762	21150
Среднее потребление, мДж	179653	21188
Δ в %		88,21

Таким образом совместное применение оптимизаций может оказать значительное влияние на сгенерированный байт-код и потребление энергии при его исполнении.

3.8. Staticization

Вспомогательный код, используемый в предыдущих экспериментах, был воссоздан на языке Kotlin [33]. Был объявлен класс `StatHelper` с тремя вычислительными методами `triangularNumber()`, `fibonacci()` и `gcd()`. В теле цикла к переменной `tmpint` прибавлялся результат вызова каждого метода. Байт-код неоптимизированной версии для каждого такого прибавления содержал четыре инструкции: получение экземпляра класса `StatHelper`, вызов соответствующего метода у экземпляра, запись результата вызова в регистр, прибавление значения этого регистра к переменной `tmpint`.

```
object StatHelper {
    fun triangularNumber(n: Int): Int {
        var res = 0
        for (i in 0 until n) res += i
        return res
    }
    fun fibonacci(n: Int): Int {
        val cache = intArrayOf(1, 1)
        for (i in 2..n) {
            cache[i % 2] = cache[0] + cache[1]
        }
        return cache[n % 2]
    }
    fun gcd(v1: Int, v2: Int): Int {
        var a = v1
        var b = v2
        while (b != 0) {
            a %= b
            a += b
            b = a - b
            a -= b
        }
        return a
    }
```

```
}
```

Listing 12: содержимое тела класса StatHelper

```
for (i in 0 until cycles) {
    tmpint += StatHelper.triangularNumber(i)
    tmpint += StatHelper.fibonacci(i)
    tmpint += StatHelper.gcd(i, cycles - i)
}
```

Listing 13: содержимое тела цикла с тестируемым кодом

Но в оптимизированной версии тела методов вставлялись в места их вызовов из-за оптимизации встраивания. Поэтому количество вызовов методов в теле цикла увеличилось, а также увеличился код этих методов путем добавления повторяющихся инструкций. Когда количество вызовов увеличилось до 10 для каждого метода и тело каждого метода содержало 40 дополнительных операций, к коду перестала применяться оптимизация встраивания и сработала оптимизация статизации. Методы в байт-коде класса `StatHelper` стали объявлены статичными. Байт-код оптимизированной версии для каждого прибавления стал содержать три инструкции: статический вызов соответствующего метода класса, запись результата вызова в регистр, прибавление значения этого регистра к переменной `tmpint`.

```
:goto_52
if-ge v5, v0, :cond_12b

.line 147
sget-object v7, Lcom/example/staticization/StatHelper;
    ->INSTANCE:Lcom/example/staticization/StatHelper;

const/4 v8, 0x1

invoke-virtual {v7, v8}, Lcom/example/staticization/StatHelper;
    ->triangularNumber(I)I

move-result v7
```

```
add-int/2addr v6, v7

.line 148
sget-object v7, Lcom/example/staticization/StatHelper;
    ->INSTANCE:Lcom/example/staticization/StatHelper;

invoke-virtual {v7, v8}, Lcom/example/staticization/StatHelper;->fibonacci(I)I

move-result v7

add-int/2addr v6, v7

.line 149
sget-object v7, Lcom/example/staticization/StatHelper;
    ->INSTANCE:Lcom/example/staticization/StatHelper;

invoke-virtual {v7, v8, v8}, Lcom/example/staticization/StatHelper;->gcd(II)I

move-result v7

add-int/2addr v6, v7
```

Listing 14: неоптимизированный байт-код тела цикла

```
:goto_54
if-ge v6, v0, :cond_f1

const/4 v8, 0x1

.line 18
invoke-static {v8}, Lcom/example/staticization/StatHelper;
    ->triangularNumber(I)I

move-result v9

add-int/2addr v9, v7

.line 19
invoke-static {v8}, Lcom/example/staticization/StatHelper;->fibonacci(I)I

move-result v7

add-int/2addr v7, v9

.line 20
invoke-static {v8, v8}, Lcom/example/staticization/StatHelper;->gcd(II)I

move-result v9

add-int/2addr v9, v7
```

Listing 15: оптимизированный байт-код тела цикла

Оптимизации	выкл.	вкл.
Количество циклов	1000000	1000000
Тест №1, мДж	53721	52311
Тест №2, мДж	53298	52593
Тест №3, мДж	53721	52170
Тест №4, мДж	53298	53016
Тест №5, мДж	53721	52452
Тест №6, мДж	53580	52311
Тест №7, мДж	53157	52311
Тест №8, мДж	53580	52311
Тест №9, мДж	53157	52311
Тест №10, мДж	53439	52593
Тест №11, мДж	53580	52311
Тест №12, мДж	53439	52311
Тест №13, мДж	53862	53157
Тест №14, мДж	53721	52734
Тест №15, мДж	53289	52170
Среднее потребление, мДж	53509	52471
Δ в %		1,94

Из-за того, что инструкция создания экземпляра класса была убрана, уменьшились количество исполняемого кода и уменьшилось количество энергии, потреблённой процессором.

4. Заключение

Оптимизации, проводимые компилятором R8, влияют на код приложения и уменьшают его энергопотребление, но даже в синтетических тестах влияние большинства оптимизаций мало - порядка единиц или десятых долей процентов. Тестирование оптимизаций на полноразмерных приложениях не удалось реализовать (попытки принимались [35]), но исходя из того, что тестируемые программы крайне специфичны и на практике чаще всего подобных им нельзя встретить, наша рабочая гипотеза состоит в том, что на энергопотребление реальных приложений эти оптимизации практически не влияют.

Список литературы

[1] Android Studio release notes. — URL: https://developer.android.com/studio/releases (online; accessed: 14.09.2020).

[2] Anti-patterns and the energy efficiency of Android applications / Rodrigo Morales, Ruben Saborido, Foutse Khomh et al. // arXiv preprint arXiv:1610.05711. — 2016.

[3] Avoiding Vendor- and Version-Specific VM Bugs. — URL: https://jakewharton.com/avoiding-vendor-and-version-specific-vm-bugs/ (online; accessed: 22.11.2020).

[4] CPUFreq Documentation. — URL: https://www.kernel.org/doc/Documentation/cpu-freq/ (online; accessed: 14.09.2020).

[5] Cardoso João Manuel Paiva, de Figueiredo Coutinho José Gabriel, Diniz Pedro C. Embedded Computing for High Performance: Efficient Mapping of Computations Using Customization, Code Transformations and Compilation. — Morgan Kaufmann, 2017.

[6] Chen Xinbo, Zong Ziliang. Android app energy efficiency: The impact of language, runtime, compiler, and implementation // 2016 IEEE international conferences on big data and cloud computing (BDCloud), social computing and networking (socialcom), sustainable computing and communications (sustaincom)(BDCloud-socialcom-sustaincom) / IEEE. — 2016. — P. 485–492.

[7] The DaCapo Benchmarking Suite. — URL: http://dacapobench.org/ (online; accessed: 14.09.2020).

[8] The DaCapo benchmarks: Java benchmarking development and analysis / Stephen M Blackburn, Robin Garner, Chris Hoffmann et al. // Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. — 2006. — P. 169–190.

[9] Dalvik bytecode. — URL: `https://source.android.com/devices/tech/dalvik/dalvik-bytecode` (online; accessed: 14.09.2020).

[10] Detecting anomalous energy consumption in android applications / Marco Couto, Tiago Carção, Jácome Cunha et al. // Brazilian Symposium on Programming Languages / Springer. — 2014. — P. 77–91.

[11] Enum Ordinal And Names и String Constant Operations test. — URL: `https://github.com/thofyb/coursework/tree/master/eoansco` (online; accessed: 23.11.2020).

[12] Enum Ordinal and Names test. — URL: `https://github.com/thofyb/coursework/tree/master/eoan` (online; accessed: 23.11.2020).

[13] Georgiev Anton B, Sillitti Alberto, Succi Giancarlo. Open source mobile virtual machines: an energy assessment of Dalvik vs. ART // IFIP International Conference on Open Source Systems / Springer. — 2014. — P. 93–102.

[14] How does code obfuscation impact energy usage? / Cagri Sahin, Mian Wan, Philip Tornquist et al. // Journal of Software: Evolution and Process. — 2016. — Vol. 28, no. 7. — P. 565–588.

[15] Impact of GC design on power and performance for Android / Ahmed Hussein, Mathias Payer, Antony Hosking, Christopher A Vick // Proceedings of the 8th ACM International Systems and Storage Conference. — 2015. — P. 1–12.

[16] Inline test. — URL: `https://github.com/thofyb/coursework/tree/master/inlining` (online; accessed: 23.11.2020).

[17] Jack & Jill compilers. — URL: `https://www.guardsquare.com/en/blog/the_upcoming_jack_and_jill_compilers_in_android` (online; accessed: 14.09.2020).

[18] Jake Wharton's blog. — URL: `https://jakewharton.com/blog/` (online; accessed: 14.09.2020).

[19] Java Obfuscator and Android App Optimizer. — URL: `https://www.guardsquare.com/en/products/proguard` (online; accessed: 14.09.2020).

[20] Jha Niraj K. Low power system scheduling and synthesis // IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281) / IEEE. — 2001. — P. 259–263.

[21] Linux manual page, sched_setaffinity(2). — URL: `https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html` (online; accessed: 14.09.2020).

[22] Method reallocation to reduce energy consumption: An implementation in android os / Luis Corral, Anton B Georgiev, Alberto Sillitti, Giancarlo Succi // Proceedings of the 29th Annual ACM Symposium on Applied Computing. — 2014. — P. 1213–1218.

[23] Mobile Operating System Market Share Worldwide. — URL: https://gs.statcounter.com/os-market-share/mobile/worldwide (online; accessed: 14.09.2020).

[24] Navitas Framework. — URL: https://github.com/Stanislav-Sartasov/Navitas-Framework (online; accessed: 14.09.2020).

[25] Navitas Lite Library. — URL: https://github.com/thofyb/NavitasLiteLibrary (online; accessed: 23.11.2020).

[26] Next-generation Dex Compiler Now in Preview. — URL: https://android-developers.googleblog.com/2017/08/next-generation-dex-compiler-now-in.html (online; accessed: 14.09.2020).

[27] OSCAR compiler controlled multicore power reduction on android platform / Hideo Yamamoto, Tomohiro Hirano, Kohei Muto et al. // International Workshop on Languages and Compilers for Parallel Computing / Springer. — 2013. — P. 155–168.

[28] Outline test. — URL: https://github.com/thofyb/coursework/tree/master/outlining (online; accessed: 23.11.2020).

[29] PowerTutor: A Power Monitor for Android-Based Mobile Platforms. — URL: http://ziyang.eecs.umich.edu/projects/powertutor/ (online; accessed: 14.09.2020).

[30] R8, the new code shrinker from Google, is available in Android studio 3.3 beta. — URL: https://android-developers.googleblog.com/2018/11/r8-new-code-shrinker-from-google-is.html (online; accessed: 14.09.2020).

[31] Schwermer Patrik. Performance Evaluation of Kotlin and Java on Android Runtime. — 2018.

[32] Standard Performance Evaluation Corporation, SPEC Benchmarks. — URL: https://www.spec.org/benchmarks.html (online; accessed: 14.09.2020).

[33] Staticiation test. — URL: https://github.com/thofyb/coursework/tree/master/staticiation (online; accessed: 23.11.2020).

[34] String Constatn Operations test. — URL: https://github.com/thofyb/coursework/tree/master/sco (online; accessed: 23.11.2020).

[35] Vlc test. — URL: https://github.com/thofyb/coursework/tree/vlc_test (online; accessed: 23.11.2020).

[36] compiler/optimizing/inliner.cc - platform/art - Git at Google. — URL: https://android.googlesource.com/platform/art/+/nougat-release/compiler/optimizing/inliner.cc (online; accessed: 22.11.2020).