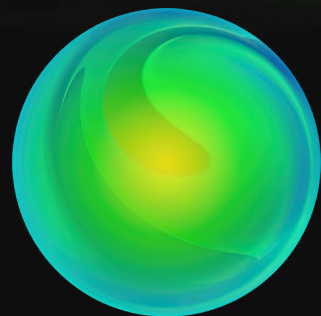


# Гибридная модель компиляции кода JIT + AOT в Android Runtime



Максим Сидоров, Максим Митюшкин

Системные сервисы, Salute TV

- Как в Android устроена гибридная модель компиляции кода JIT + AOT
- Как Google пришли к этой модели в 2016 году через боль и страдания
- Какой профит дает этот подход в реальных цифрах
- А что же в Java (проект Leyden)

## В Android используется сразу два вида компиляции

AOT компиляция – предварительно скомпилированный код

JIT компиляция – компиляция кода в runtime

Давайте сначала разберемся с JIT компиляцией...

# Сначала весь код исполняется в режиме интерпретатора

4 / 42

```
inline void PerformCall(Thread* self,
                        const CodeItemDataAccessor& accessor,
                        ArtMethod* caller_method,
                        const size_t first_dest_reg,
                        ShadowFrame* callee_frame,
                        JValue* result,
                        bool use_interpreter_entrypoint)
    REQUIRES_SHARED(Locks::mutator_lock_) {
    if (LIKELY(Runtime::Current()->IsStarted())) {
        if (use_interpreter_entrypoint) {
            interpreter::ArtInterpreterToInterpreterBridge(self, accessor, callee_frame, result);
        } else {
            interpreter::ArtInterpreterToCompiledCodeBridge(
                self, caller_method, callee_frame, first_dest_reg, result);
        }
    } else {
        interpreter::UnstartedRuntime::Invoke(self, accessor, callee_frame, result, first_dest_reg);
    }
}
```

# При этом для каждого метода заводится счетчик его вызовов

```
class ArtMethod final {  
    ...  
  
    protected:  
    ...  
  
    union {  
        // Non-abstract methods: The hotness we measure for this method. Not atomic,  
        // as we allow missing increments: if the method is hot, we will see it eventually.  
        uint16_t hotness_count_;  
        // Abstract methods: IMT index.  
        uint16_t imt_index_;  
    };  
};
```

Начальное значение счетчика равно 16 383

# При каждом вызове метода декрементируется счетчик ВЫЗОВОВ

6 / 42

```
void Jit::MethodEntered(Thread* thread, ArtMethod* method) {  
    Runtime* runtime = Runtime::Current();  
    if (UNLIKELY(runtime->UseJitCompilation() && JitAtFirstUse())) {  
        ArtMethod* np_method = method->GetInterfaceMethodIfProxy(kRuntimePointerSize);  
        if (np_method->IsCompilable()) {  
            JitCompileTask compile_task(  
                method, JitCompileTask::TaskKind::kCompile, CompilationKind::kOptimized);  
            ScopedSetRuntimeThread ssrt(thread);  
            compile_task.Run(thread);  
        }  
        return;  
    }  
    AddSamples(thread, method);  
}
```

# При каждом вызове метода декрементируется счетчик ВЫЗОВОВ

```
inline void Jit::AddSamples(Thread* self, ArtMethod* method) {  
    if (method->CounterIsHot()) {  
        if (method->IsMemorySharedMethod()) {  
            if (self->DecrementSharedMethodHotness() == 0) {  
                self->ResetSharedMethodHotness();  
            } else {  
                return;  
            }  
        } else {  
            method->ResetCounter(Runtime::Current()->GetJITOptions()->GetWarmupThreshold());  
        }  
        MaybeEnqueueCompilation(method, self);  
    } else {  
        method->UpdateCounter(1);  
    }  
}
```

```
inline bool ArtMethod::CounterIsHot() {  
    DCHECK(!IsAbstract());  
    return hotness_count_ == 0;  
}
```

Первые 16 000 вызовов метод выполняется в режиме интерпретатора

После 16 000 исполнений в режиме интерпретатора,  
запускается компиляция метода

```
inline void Jit::AddSamples(Thread* self, ArtMethod* method) {  
    if (method->CounterIsHot()) {  
        if (method->IsMemorySharedMethod()) {  
            if (self->DecrementSharedMethodHotness() == 0) {  
                self->ResetSharedMethodHotness();  
            } else {  
                return;  
            }  
        } else {  
            method->ResetCounter(Runtime::Current()->GetJITOptions()->GetWarmupThreshold());  
        }  
        MaybeEnqueueCompilation(method, self);  
    } else {  
        method->UpdateCounter(1);  
    }  
}
```

# Первая компиляция метода (Baseline)

9 / 42

```
void Jit::MaybeEnqueueCompilation(ArtMethod* method, Thread* self) {  
    ...  
    if (it == shared_method_counters_.end()) {  
        shared_method_counters_[method] = kIndividualSharedMethodHotnessThreshold;  
        return;  
    } else if (it->second != 0) {  
        DCHECK_LE(it->second, kIndividualSharedMethodHotnessThreshold);  
        shared_method_counters_[method] = it->second - 1;  
        return;  
    } else {  
        shared_method_counters_[method] = kIndividualSharedMethodHotnessThreshold;  
    }  
}  
  
if (!method->IsNative() && GetCodeCache()->CanAllocateProfilingInfo()) {  
    thread_pool_->AddTask(  
        self,  
        new JitCompileTask(method, JitCompileTask::TaskKind::kCompile, CompilationKind::kBaseline));  
} else {  
    thread_pool_->AddTask(  
        self,  
        new JitCompileTask(method,  
            JitCompileTask::TaskKind::kCompile,  
            CompilationKind::kOptimized));  
}
```

Это упрощенная компиляция без затратных оптимизаций.

- Отключены все оптимизации
- Уменьшен порог инлайнинга (с 32 до 14)
- Уменьшена глубина инлайнинга
- Включено профилирование кода

| Интерпретатор<br>(ns) | Compiled Baseline<br>(ns) |
|-----------------------|---------------------------|
| 85 000                | 11 750                    |

## После Baseline компиляции метода

- Начинает исполняться скомпилированная версия метода (ассемблерный код)
- Но счетчик продолжает тикать...

# Счетчик вызовов для скомпилированного метода

При каждом вызове компиляции метода, в памяти создается новый экземпляр класса ProfilingInfo

```
class ProfilingInfo {
public:
    uint16_t GetBaselineHotnessCount() const {
        return baseline_hotness_count_;
    }

private:
    // Hotness count for methods compiled with the JIT baseline compiler.
    uint16_t baseline_hotness_count_;

    // Method this profiling info is for.
    ArtMethod* method_;

    // Number of instructions we are profiling in the ArtMethod.
    const uint32_t number_of_inline_caches_;

    // When the compiler inlines the method associated to this ProfilingInfo,
    // it updates this counter so that the GC does not try to clear the inline caches.
    uint16_t current_inline_uses_;
}
```

Хранит информацию профайлера для скомпилированного метода:

- КОЛИЧЕСТВО ВЫЗОВОВ метода
- статистика для инлайнинга

# Счетчик вызовов для скомпилированного метода

При инициализации объекта ProfilingInfo счетчик вызовов устанавливается в начальное значение

```
ProfilingInfo::ProfilingInfo(ArtMethod* method, const std::vector<uint32_t>& entries)
    : baseline_hotness_count_(GetOptimizeThreshold()),
      method_(method),
      number_of_inline_caches_(entries.size()),
      current_inline_uses_(0) {
    memset(&cache_, 0, number_of_inline_caches_ * sizeof(InlineCache));
    for (size_t i = 0; i < number_of_inline_caches_; ++i) {
        cache_[i].dex_pc_ = entries[i];
    }
}
```

GetOptimizedThreshold (по умолчанию = 65 535)

# Счетчик вызовов для скомпилированного метода

14 / 42

Обновление счетчика встраивается в ассемблерный код метода

```
void CodeGeneratorARM64::MaybeIncrementHotness(bool is_frame_entry) {  
    ...  
  
    if (GetGraph()->IsCompilingBaseline() && !Runtime::Current()->IsAotCompiler()) {  
        SlowPathCodeARM64* slow_path = new (GetScopedAllocator()) CompileOptimizedSlowPathARM64();  
        AddSlowPath(slow_path);  
        ProfilingInfo* info = GetGraph()->GetProfilingInfo();  
        DCHECK(info != nullptr);  
        DCHECK(!HasEmptyFrame());  
        uint64_t address = reinterpret_cast64<uint64_t>(info);  
        vixl::aarch64::Label done;  
        UseScratchRegisterScope temps(masm);  
        Register temp = temps.AcquireX();  
        Register counter = temps.AcquireW();  
        __ Ldr(temp, DeduplicateUint64Literal(address));  
        __ Ldrh(counter, MemOperand(temp, ProfilingInfo::BaselineHotnessCountOffset().Int32Value()));  
        __ Cbz(counter, slow_path->GetEntryLabel()); ----->  
        __ Add(counter, counter, -1);  
        __ Strh(counter, MemOperand(temp, ProfilingInfo::BaselineHotnessCountOffset().Int32Value()));  
        __ Bind(slow_path->GetExitLabel());  
    }  
}
```

```
if (counter == 0) {  
    // call Jit::EnqueueOptimizedCompilation(ArtMethod* method,  
    goto slow_path->GetEntryLabel()  
}
```

# Счетчик вызовов для скомпилированного метода

15 / 42

Обновление счетчика встраивается в ассемблерный код метода

```
void CodeGeneratorARM64::MaybeIncrementHotness(bool is_frame_entry) {  
    ...  
  
    if (GetGraph()->IsCompilingBaseline() && !Runtime::Current()->IsAotCompiler()) {  
        SlowPathCodeARM64* slow_path = new (GetScopedAllocator()) CompileOptimizedSlowPathARM64();  
        AddSlowPath(slow_path);  
        ProfilingInfo* info = GetGraph()->GetProfilingInfo();  
        DCHECK(info != nullptr);  
        DCHECK(!HasEmptyFrame());  
        uint64_t address = reinterpret_cast64<uint64_t>(info);  
        vixl::aarch64::Label done;  
        UseScratchRegisterScope temps(masm);  
        Register temp = temps.AcquireX();  
        Register counter = temps.AcquireW();  
        __ Ldr(temp, DeduplicateUint64Literal(address));  
        __ Ldrh(counter, MemOperand(temp, ProfilingInfo::BaselineHotnessCountOffset().Int32Value()));  
        __ Cbz(counter, slow_path->GetEntryLabel()); -----  
        __ Add(counter, counter, -1);  
        __ Strh(counter, MemOperand(temp, ProfilingInfo::BaselineHotnessCountOffset().Int32Value()));  
        __ Bind(slow_path->GetExitLabel());  
    }  
}
```

Когда счетчик достигает нуля, запускается  
компиляция в режиме optimized

```
if (counter == 0) {  
    // call Jit::EnqueueOptimizedCompilation(ArtMethod* method,  
    goto slow_path->GetEntryLabel();  
}
```

В Optimized компиляции добавляется огромное количество оптимизаций, недоступных при baseline и статической компиляции

Optimized компиляция может быть намного более затратной и медленной по сравнению с Baseline компиляцией

Но может давать двухкратный прирост скорости

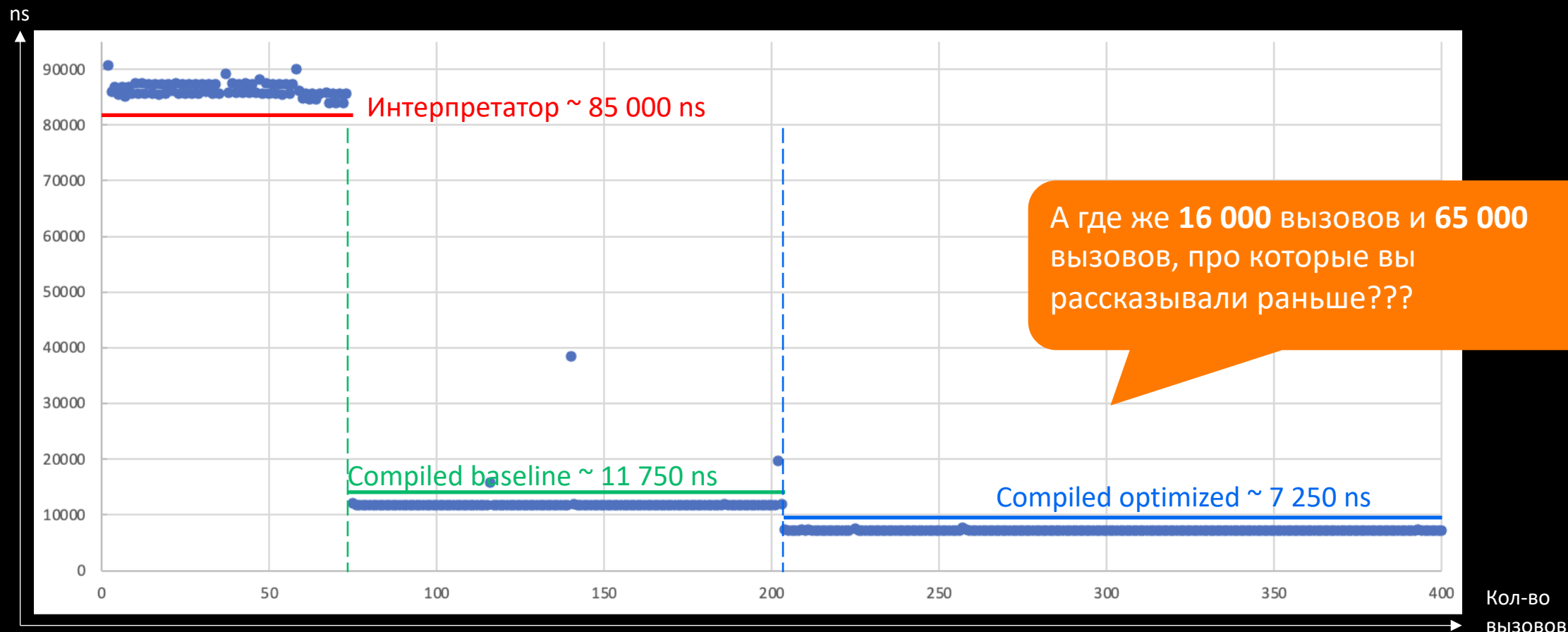
| Интерпретатор<br>(ns) | Baseline<br>(ns) | Optimized<br>(ns) |
|-----------------------|------------------|-------------------|
| 85 000                | 11 750           | 7 250             |

## Компиляция метода в Android происходит в три этапа

- Первые 16 000 исполнений происходят в режиме интерпретатора
- Следующие 65 000 исполнений происходят для кода, скомпилированного в baseline режиме (аналог C1)
- И только после 81 000 исполнений метода запускается тяжелая компиляция в режиме optimized со всеми доступными оптимизациями (аналог C2)

# Замеры этапов компиляции метода

18 / 42



Наш метод скомпилировался уже на 75 вызовах

Компиляция с оптимизациями выполнилась после 200 вызовов функции

## Наша тестовая функция

```
fun heavyFunction(start: Int): Int {  
    var value = start  
  
    repeat(1000) {  
        if (value % 2 == 0) {  
            value += 1  
        } else {  
            value -= 1  
        }  
    }  
  
    return value  
}
```

**Hotness\_count** — это не счетчик вызовов

## Интерпретатор

- Каждый вызов метода уменьшает счетчик на 1
- Каждая операция ветвления уменьшает счетчик на 1 (условный, безусловный переход)

В нашем случае каждый вызов уменьшает счетчик на 3 001  
 $= 1 + 1000 * (1 + 1 + 1)$

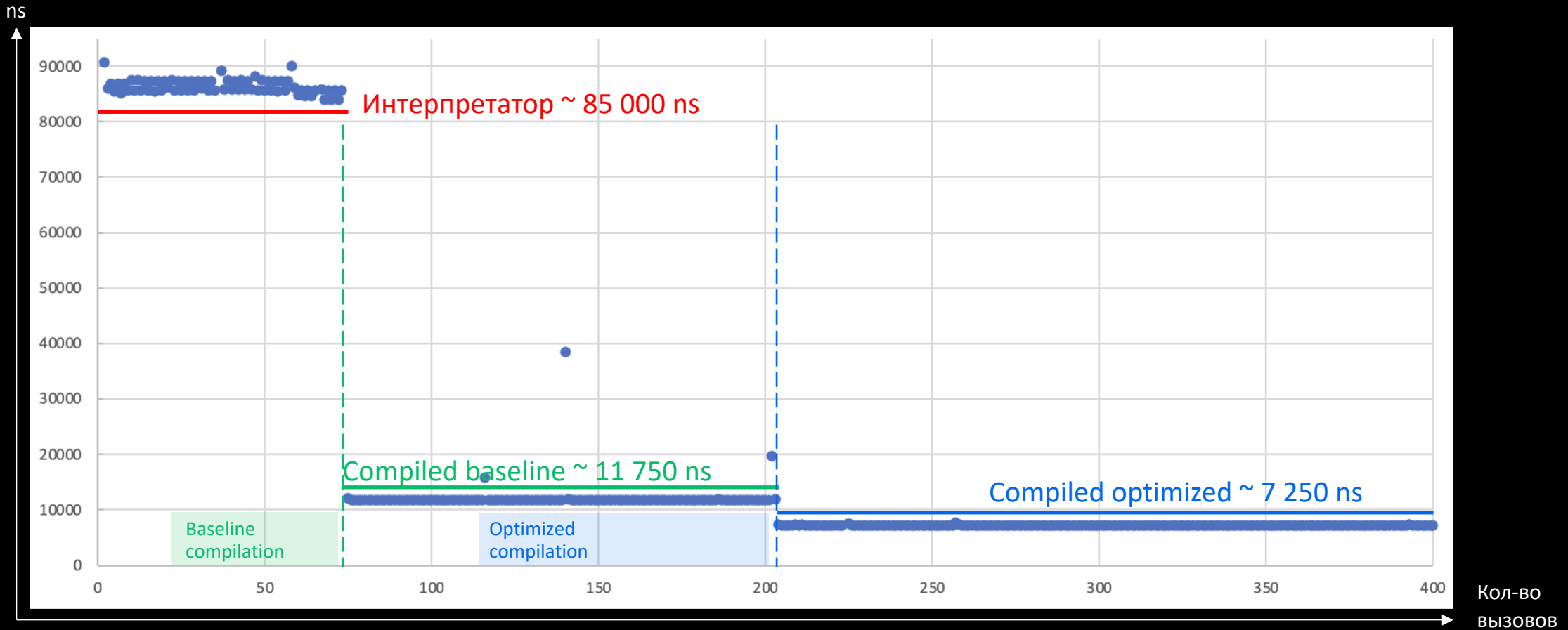
## Baseline

- Каждый вызов метода уменьшает счетчик на 1
- Каждый BackEdge уменьшает счетчик на 1

В нашем случае каждый вызов уменьшает счетчик на 1 001  
 $= 1 + 1000$  (repeat = BackEdge)

# Этапы компиляции метода

20 / 42



Задача Baseline компиляции создаётся после 17 вызова и выполнялась 966 микросекунд

Optimized компиляция запустилась примерно после 40 вызовов скомпилированного метода и выполнялась 1026 микросекунд



Jit хранит весь скомпилированный код и статистику по горячим методам в памяти процесса

При каждом рестарте процесса мы теряем эту информацию

А хочется ведь переиспользовать это при последующих запусках приложения...

В Android используется гибридная модель  
компиляции кода

JIT (just-In-time) компиляция

Код компилируется динамически, по мере его выполнения в среде исполнения

+

AOT (ahead-of-time) компиляция

Весь код компилируется заранее

|             |                                                                |
|-------------|----------------------------------------------------------------|
| Android 1   | Dalvik, в начале был только интерпретатор                      |
| Android 2.2 | Dalvik (JIT for hot-paths), появляется JIT для горячих методов |
| Android 5   | ART (only AOT), полный отказ от JIT компиляции                 |

Здесь ART – это Android Run Time  
Ускорено время запуска приложений

Настоящий Runtime остался в Dalvik)

## AOT компиляция — это наше все

### Но есть и минусы

- Размер приложений существенно вырос
- Предварительная компиляция стала занимать ощутимое время
- На это стали жаловаться пользователи

И тогда инженеры Google придумали как скрестить AOT и JIT компиляцию



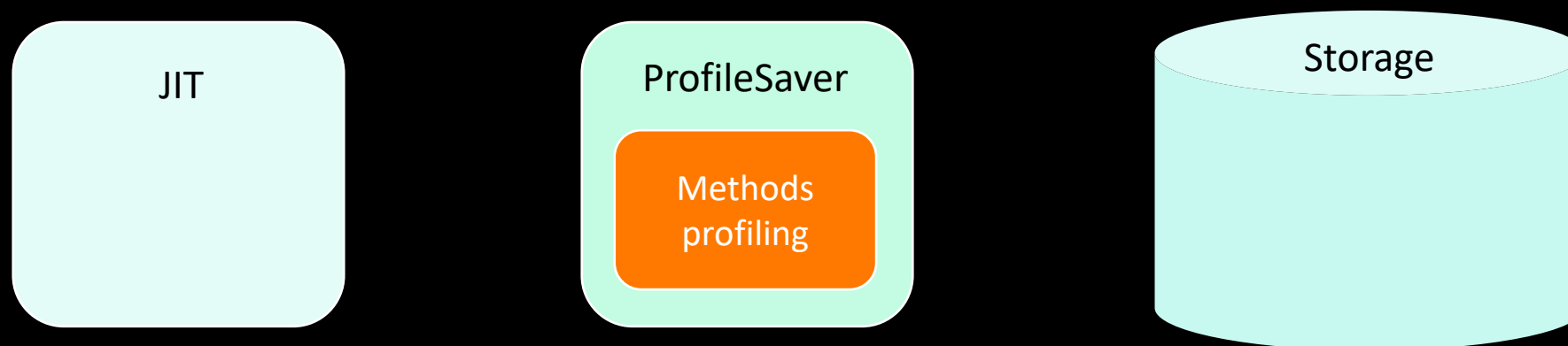
— JIT собирает статистику по профилированию методов

И тогда инженеры Google придумали как скрестить AOT и JIT компиляцию



- JIT собирает статистику по профилированию методов
- ProfileSaver забирает эту статистику из JIT

И тогда инженеры Google придумали как скрестить AOT и JIT компиляцию



- JIT собирает статистику по профилированию методов
- ProfileSaver забирает эту статистику из JIT
- ProfileSaver сохраняет ее на диск в специальный файл профиля приложения

# Как устроен baseline профиль

28 / 42

Baseline профиль – это обычный файл, содержащий список методов

```
HSP_Landroidx/fragment/app/FragmentManager;->addFragment(Landroidx/fragment/app/Fragment;)V
HSPLandroidx/fragment/app/FragmentManager;->burpActive()V
HSPLandroidx/fragment/app/FragmentManager;->containsActiveFragment(Ljava/lang/String;)Z
HSPLandroidx/fragment/app/FragmentManager;->dispatchStateChange(I)V
HSPLandroidx/fragment/app/FragmentManager;->findActiveFragment(Ljava/lang/String;)Landroidx/fragment/app/Fragment;
HSPLandroidx/fragment/app/FragmentManager;->findFragmentById(I)Landroidx/fragment/app/Fragment;
HSPLandroidx/fragment/app/FragmentManager;->findFragmentIndexInContainer(Landroidx/fragment/app/Fragment;)I
HSPLandroidx/fragment/app/FragmentManager;->getActiveFragmentManagerStateManagers()Ljava/util/List;
HSPLandroidx/fragment/app/FragmentManager;->getActiveFragments()Ljava/util/List;
HSPLandroidx/fragment/app/FragmentManager;->getFragmentManager(Ljava/lang/String;)Landroidx/fragment/app/FragmentManager;
HSPLandroidx/fragment/app/FragmentManager;->getFragments()Ljava/util/List;
HSPLandroidx/fragment/app/FragmentManager;->makeActive(Landroidx/fragment/app/FragmentManager;)V
HSPLandroidx/fragment/app/FragmentManager;->moveToExpectedState()V
```

Для каждого метода указаны его класс, имя метода и аргументы

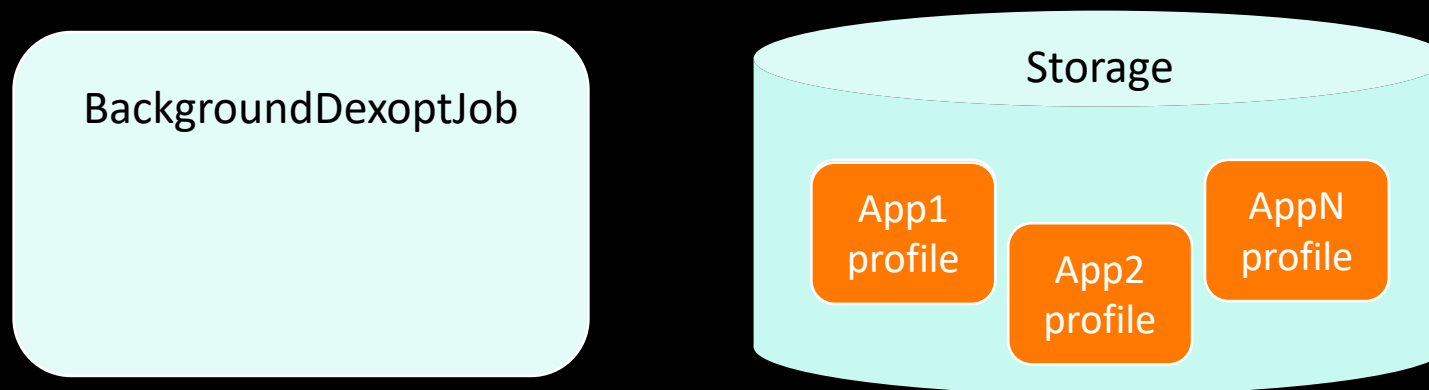
А также специальные флаги в заголовке метода

H – горячий метод

S – startup метод

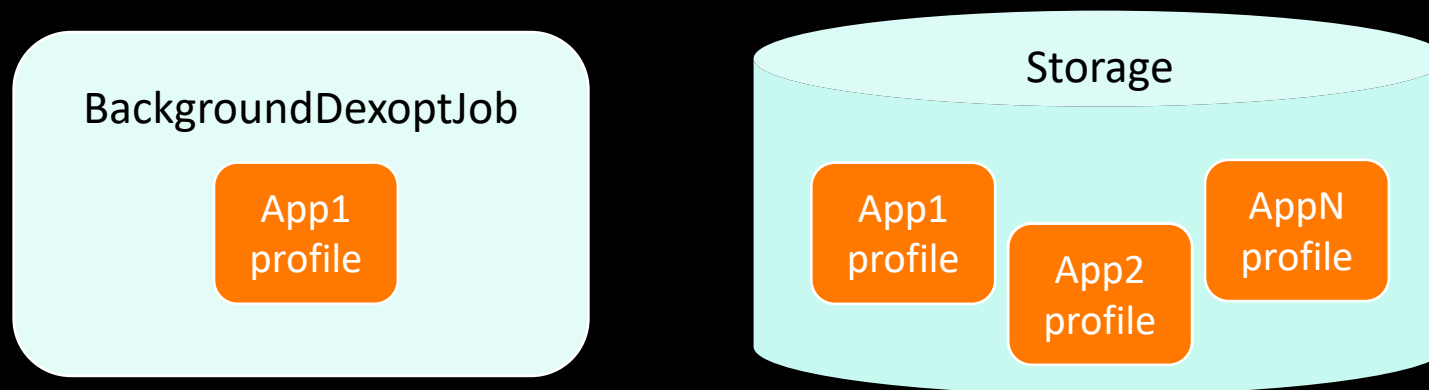
P – post startup метод

Когда система бездействует, происходит фоновая перекомпиляция приложений



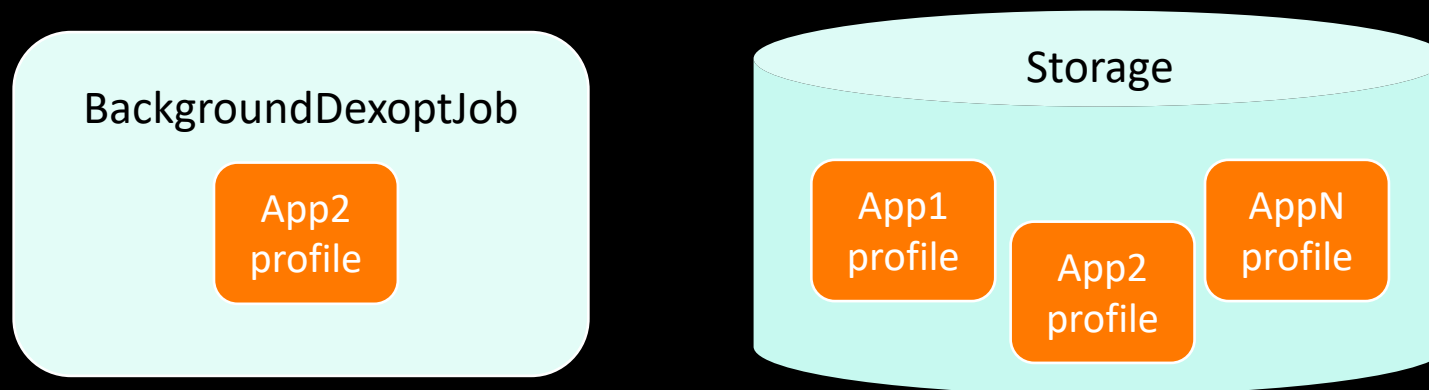
— BackgroundDexoptJob анализирует профили приложений на диске

Когда система бездействует, происходит фоновая перекомпиляция приложений



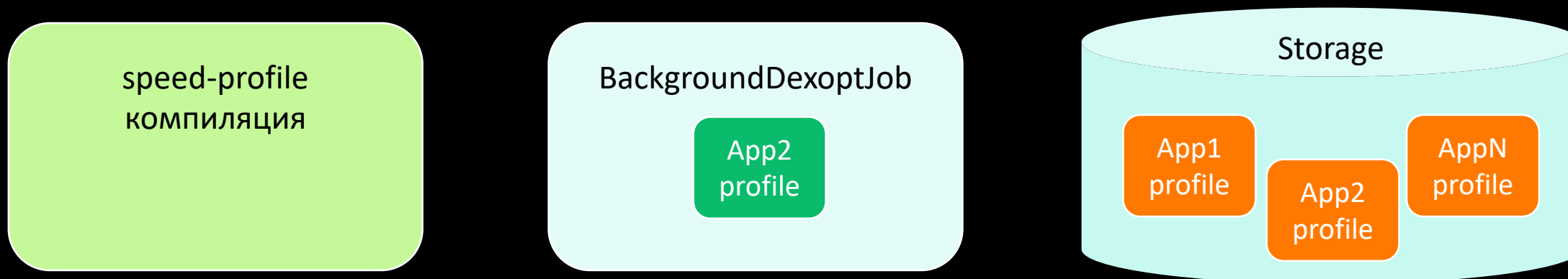
— BackgroundDexoptJob анализирует профили приложений на диске

## Когда система бездействует, происходит фоновая перекомпиляция приложений



- BackgroundDexoptJob анализирует профили приложений на диске
- Для приложений с измененными профилями производится полная AOT компиляция в режиме speed-profile

## Когда система бездействует, происходит фоновая перекомпиляция приложений



- BackgroundDexoptJob анализирует профили приложений на диске
- Для приложений с измененными профилями производится полная AOT компиляция в режиме speed-profile
- Speed-profile компиляция компилирует только Hot и Startup методы из профиля приложения

По сути Android просто заново компилирует ваше приложение после каждого запуска

С каждым запуском приложения скомпилированный код включает в себя все больше и больше горячих методов

При этом компилируется в среднем не больше 15-20% кода приложения

Соответственно объем скомпилированного кода не распухает также, как при обычной AOT компиляции

# Baseline профили - Android 9, 2018 г.

## Baseline профили разработчиков

Разработчики могут формировать свои стартовые **baseline профили**

Это делается с помощью обычных инструментальных тестов

Должны включать в себя только профилирование первого запуска приложения

Используются для первой компиляции приложения после его установки

Профиль не зависит от версий OS и железа и может использоваться на любом устройстве

## Инженеры Google придумали крутую штуку

Ускорение первого старта приложения после его установки

Исключение режима прогрева приложения и исполнения кода в интерпретаторе на старте

Постепенное пополнение профилей и снижение нагрузки на компиляцию во время работы приложения

При этом сохраняются почти все преимущества JIT компилятора и его оптимизаций

Google декларирует ускорение старта приложений на 15-20%

## Google декларирует ускорение старта на 15-20%

| Приложение | Время запуска |
|------------|---------------|
| IVI        | 17%           |
| Okko       | 16%           |
| Premier    | 18%           |
| Kinopoisk  | 19%           |
| Rutube     | 18%           |

Брали 75 перцентиль на 25 запусков приложения

Отличный результат, Google нас не обманул

Среднее ускорение: 15-20%

## Замеры вызовов интерпретатора

| Приложение | Время запуска | Время исполнения в интерпретаторе | Кол-во вызовов в интерпретаторе |
|------------|---------------|-----------------------------------|---------------------------------|
| IVI        | 17%           | 99%                               | 83%                             |
| Okko       | 16%           | 89%                               | 85%                             |
| Premier    | 18%           | 95%                               | 66%                             |
| Kinopoisk  | 19%           | 99%                               | 73%                             |
| Rutube     | 18%           | 96%                               | 62%                             |

Исполнение кода в интерпретаторе значительно сократилось

|         |                                       |
|---------|---------------------------------------|
| Java 9  | jaotc – AOT компилятор в нативный код |
| 2022    | CRaC – гибридная компиляция           |
| Java 26 | Leyden – сохранение JIT кеша в файл   |

С момента появления эти экспериментальные функции использовались крайне редко, а усилия, необходимые для их поддержки и улучшения были значительны

## Проект Leyden

Сначала производится training-запуск приложения

```
java -jar app.jar -XX:AOTCacheOutput=${aot-cache-file}
```

Затем на основе собранного AOT-кеша производится production-запуск приложения

```
java -jar app.jar -XX:AOTCache=${aot-cache-file}
```

## Сравнение с решением Google (ART)

### Плюсы:

- Хранится полная статистика JIT (счетчики вызовов, весь скомпилированный код и даже метадата загруженных классов)
- Поддерживается возможность деоптимизации. Методы, попавшие в AOT компиляцию могут быть перекомпилированы заново

### Минусы:

- Нет возможности постепенного пополнения AOT методов. AOT компиляция производится только один раз в специальном обучающем режиме
- AOT компиляция жестко привязана к железу и версии виртуальной машины. По сути AOT кеш является не переносимым

## Гибридная модель компиляции кода

Дает реальный профит в виде снижения нагрузки на компилятор

Предварительно компилируется только небольшая часть методов (Hot + Startup)

Решение от Google в проде с 2016 года и уже доказало свою эффективность

В большой Java ждем проект Leyden, который выглядит очень перспективно...

# Спасибо за внимание

linkedIn: [sidorov-max](#)



linkedIn: [kotlinovsky](#)



Максим Сидоров, Максим Митюшкин

Системные сервисы, Salute TV

