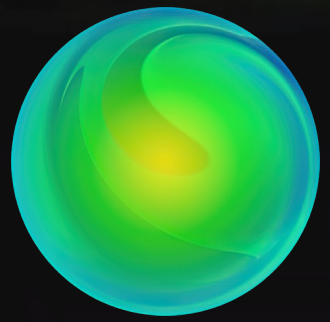
Android Runtime под капотом



Максим Сидоров, Максим Митюшкин

Системные сервисы, Salute TV

SBER DEVICES

Предыстория

Mobius 2025: доклад «Нюансы работы Android Runtime в сравнении с HotSpot VM»

— Исследование различий производительности коллекций и sequence в Android и HotSpot VM

— В большой Java коллекции ощутимо выигрывают у sequence (30%), а в Android наоборот значительно проигрывают

— Нам стало интересно, чем объясняется этот эффект...

Предыстория

Что нам удалось узнать

— Все различия в производительности объясняются нюансами работы компиляторов разных виртуальных машин

— В большой Java мощный оптимизатор, который гораздо лучше справляется с инлайнингом больших по размеру методов

— В Android параметры оптимизатора значительно слабее и он не может эффективно инлайнить большие методы

Предыстория

Где мы остановились

На самом деле нам это удалось, просто мы неправильно измеряли)

— Нам захотелось сделать компилятор Android таким же эффективным, как компилятор HotSpot VM

— Но чтобы сделать это, надо сначала разобраться, как работает компилятор Android

Как работает компиляция кода в Android Runtime

Компиляция в Android Runtime

В Android используется два вида компиляции

AOT компиляция – предварительно скомпилированный код

JIT компиляция – компиляция кода в runtime

Интерпретатор – исполнение кода без компиляции

Давайте разбираться, как все это уживается в Android...

Сначала весь код исполняется в режиме интерпретатора


```
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


```
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 исполняется 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)


```
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));
```

Baseline компиляция

Это упрощенная компиляция без затратных оптимизаций.

— Отключены все оптимизации

— Уменьшен порог инлайнинга (с 32 до 14)

— Уменьшена глубина инлайнинга

— Включено профилирование кода

Интерпретатор (ns)	Compiled Baseline (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

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::BaselineHotness(
    __ 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()
}
```

SBER DEVICES

Счетчик вызовов для скомпилированного метода

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::BaselineHotnessC
    __ 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()
}
```

Optimized компиляция

Добавляются следующие оптимизации:

- Inliner
- InstructionSimplifier
- SideEffectsAnalysis
- SelectGenerator
- CodeSinking
- InvariantCodeMotion
- GlobalValueNumbering
- InductionVarAnalysis
- LoadStoreElimination

Optimized компиляция

Добавляются следующие оптимизации:

- **Inliner**
- InstructionSimplifier
- SideEffectsAnalysis
- SelectGenerator
- CodeSinking
- InvariantCodeMotion
- GlobalValueNumbering
- InductionVarAnalysis
- LoadStoreElimination

Inliner – это базовая оптимизация, которая является началом для многих других оптимизаций.

before:

```
int add(int x, int y) {
   return x + y;
}

int r = add(a, b);
```

after:

```
int r = a + b;
```

Как инлайниг влияет на скорость работы смотрите в нашем предыдущем докладе

Optimized компиляция

Добавляются следующие оптимизации:

- Inliner
- **InstructionSimplifier**
- SideEffectsAnalysis
- SelectGenerator
- CodeSinking
- InvariantCodeMotion
- GlobalValueNumbering
- InductionVarAnalysis
- LoadStoreElimination

InstructionSimplifier - убирает избыточные и лишние конструкции, заменяя их более простыми.

before:

```
if (true) {
    goto a;
} else {
    goto a;
}
```

after:

```
goto a;
```

Optimized компиляция

Добавляются следующие оптимизации:

- Inliner
- InstructionSimplifier
- **SideEffectsAnalysis**
- SelectGenerator
- CodeSinking
- InvariantCodeMotion
- GlobalValueNumbering
- InductionVarAnalysis
- LoadStoreElimination

SideEffectsAnalysis - проверяет, может ли инструкция изменить состояние программы.

Если инструкция не меняет состояние, то компилятор может применить кэширование или вообще убрать вызов инструкции.

Optimized компиляция

Добавляются следующие оптимизации:

- Inliner
- InstructionSimplifier
- SideEffectsAnalysis
- **SelectGenerator**
- CodeSinking
- InvariantCodeMotion
- GlobalValueNumbering
- InductionVarAnalysis
- LoadStoreElimination

SelectGenerator – заменяет простые ветвления.

before:

```
if (cond) {
    x = a;
} else {
    x = b;
}
```

after:

```
x = cond ? a : b;
```

Optimized компиляция

Добавляются следующие оптимизации:

- Inliner
- InstructionSimplifier
- SideEffectsAnalysis
- SelectGenerator
- **CodeSinking**
- InvariantCodeMotion
- GlobalValueNumbering
- InductionVarAnalysis
- LoadStoreElimination

CodeSinking - переносит инструкции ближе к месту ее реального использования.

before:

```
y = calculate();
...

if (cond) {
   x = y;
}
```

after:

```
if (cond) {
   y = calculate();
   x = y;
}
```

Тут используется **SideEffectsAnalysis**

calculate() – это чистая функция, не меняющая состояния

Optimized компиляция

Добавляются следующие оптимизации:

- Inliner
- InstructionSimplifier
- SideEffectsAnalysis
- SelectGenerator
- CodeSinking
- **InvariantCodeMotion**
- GlobalValueNumbering
- InductionVarAnalysis
- LoadStoreElimination

InvariantCodeMotion - выносит вызов инструкции из цикла, если нет необходимости ее постоянно вызывать.

before:

```
while (...) {
   x = a + b;
   calculate(x)
}
```

after:

```
x = a + b;
while (...) {
   calculate(x)
}
```

Тут также используется **SideEffectsAnalysis**

Optimized компиляция

Добавляются следующие оптимизации:

- Inliner
- InstructionSimplifier
- SideEffectsAnalysis
- SelectGenerator
- CodeSinking
- InvariantCodeMotion
- **GlobalValueNumbering**
- InductionVarAnalysis
- LoadStoreElimination

GlobalValueNumbering - убирает избыточные, дублирующие друг друга вычисления.

before:

```
a = b + c
...
d = b + c
```

after:

```
a = b + c
...
d = a
```

Optimized компиляция

Добавляются следующие оптимизации:

- Inliner
- InstructionSimplifier
- SideEffectsAnalysis
- SelectGenerator
- CodeSinking
- InvariantCodeMotion
- GlobalValueNumbering
- **InductionVarAnalysis**
- LoadStoreElimination

InductionVarAnalysis - анализ индукционных переменных и зависимостей от них.

before:

```
for (int i = 0; i < n; i++) {
    int offset = i * 4 + 12;
    sum *= arr[offset];
}
```

after:

```
int offset = 12;
for (int i = 0; i < n; i++) {
    sum *= arr[offset];
    offset += 4;
}
```

Из кода это не очевидно,
но переменная offset увеличивается
строго на 4 при каждой итерации цикла

Optimized компиляция

Добавляются следующие оптимизации:

- Inliner
- InstructionSimplifier
- SideEffectsAnalysis
- SelectGenerator
- CodeSinking
- InvariantCodeMotion
- GlobalValueNumbering
- InductionVarAnalysis
- **LoadStoreElimination**

LoadStoreElimination - убирает лишние операции чтения и записи в память.

before:

```
x = obj.field;
...

y = obj.field;
```

after:

```
x = obj.field;
...

y = x;
```

Optimized компиляция

А также добавляются более простые оптимизации:

— LoopOptimization (оптимизация циклов)

— ConstantFolding (вычисляет константы на этапе компиляции)

— DeadCodeElimination (удаление не используемого кода)

— BoundsCheckElimination (убирает лишние проверки границ массивов)

— Scheduling (изменение порядка инструкций)

— ConstructorFenceRedundancyElimination (убирает из кода лишние синхронизации)

— InstructionSimplifierArm64 (оптимизация инструкций под ARM64)

Optimized компиляция

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

Но может давать двухкратный прирост скорости

Baseline (ns)	Optimized (ns)
11 750	7 250

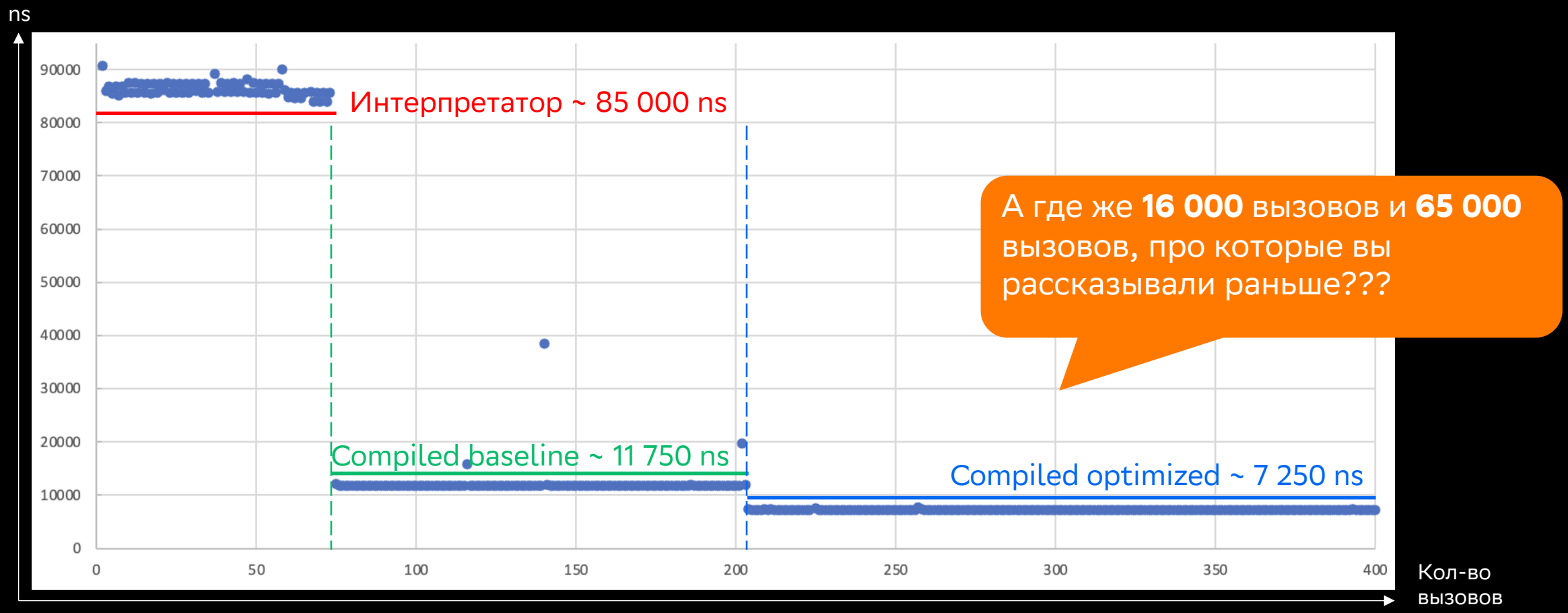
Компиляция метода в Android происходит в три этапа

— 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



Наш метод скомпилировался уже на 75 вызовах

Компиляция с оптимизациями выполнилась после 200 вызовов функции

Как уменьшается счетчик вызовов

Наша тестовая функция

```
fun heavyFunction(start: Int): Int {
    var value = start

    repeat(1000) {
        if (value % 2 == 0) {
            value += 1
        } else {
            value -= 1
        }
    }

    return value
}
```

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

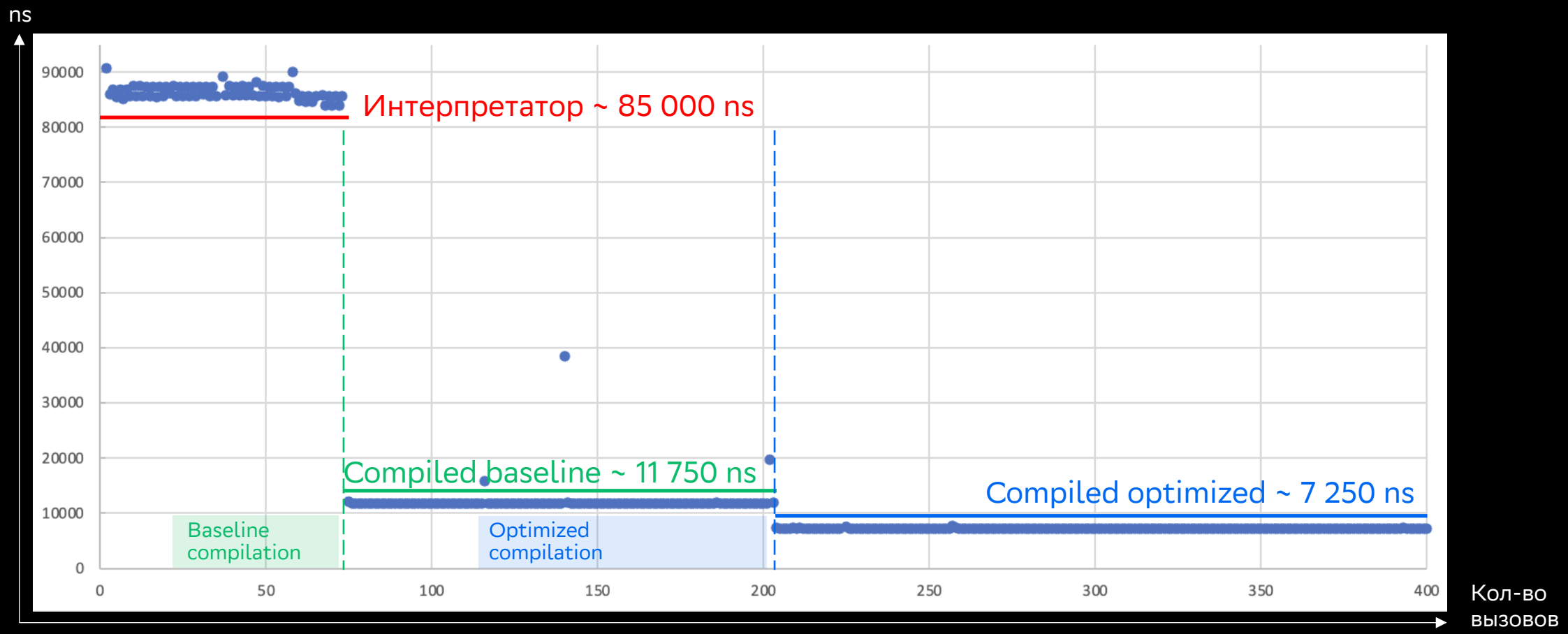
— Каждый вызов метода уменьшает счетчик на 1

— Каждый BackEdge уменьшает счетчик на 1

> BackEdge – возврат к уже выполненным инструкциям (goto назад)

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

Этапы компиляции метода



Таска Baseline компиляции создалась после 17 вызова и выполнялась 966 микросекунд

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

Но это все в рантайме

Кадр из **сериала** "**Кухня**"

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

При каждом рестарте процесса мы теряем эту информацию

С жизненным циклом Android приложений это работа практически вхолостую...

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

Виды компиляции

JIT (just-In-time) компиляция

Код компилируется динамически, по мере его выполнения в среде исполнения

Доступны более глубокие оптимизации (выигрыш до 50%)

Компилируется только тот код, который реально исполняется

AOT (ahead-of-time) компиляция

Весь код компилируется заранее

Объем скомпилированного кода в 3-4 раза больше объема байт-кода

Недоступны глубокие оптимизации

Метания Google

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

— Во время работы приложения, JIT передает в ProfileSaver информацию о всех вызванных и скомпилированных в runtime методах

— ProfileSaver сохраняет эту информацию в специальный файл профиля приложения

speed-profile компиляция

Когда система бездействует, запускается фоновая задача BackgroundDexoptJob

— BackgroundDexoptJob запускает полную dex2oat компиляцию приложения в режиме speed-profile

— speed-profile это режим компиляции, при котором компилируются только горячие методы из baseline профиля

Baseline профили

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

Это ускоряет старт приложения и позволяет экономить ресурсы JIT компилятора

Метания Google

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),
Android 7	ART (JIT + AOT), перекомпиляция после использования на основе JIT профилирования
Android 9	ART (JIT + AOT + baseline профили), install-time компиляция + перекомпиляция после использования

Здесь ART стал настоящим Android Runtime

Маркетинг ушел, пришли технологии...

Ускорение запуска приложения

Результаты замеров холодного запуска приложений на наших устройствах

	No Baseline (msec)	Baseline (msec)	Ускорение
Kinopoisk	10 700	8 500	20%
Premiere	10 250	8 100	20%
Ivi	6 200	5 170	15%
Okko	9 510	8 200	16%

Как устроен baseline профиль

Baseline профиль – это текстовый файл, содержащий список методов

```
HSPLandroidx/fragment/app/FragmentStore;->addFragment(Landroidx/fragment/app/Fragment;)V
HSPLandroidx/fragment/app/FragmentStore;->burpActive()V
HSPLandroidx/fragment/app/FragmentStore;->containsActiveFragment(Ljava/lang/String;)Z
HSPLandroidx/fragment/app/FragmentStore;->dispatchStateChange(I)V
HSPLandroidx/fragment/app/FragmentStore;->findActiveFragment(Ljava/lang/String;)Landroidx/fragment/app/Fragment;
HSPLandroidx/fragment/app/FragmentStore;->findFragmentById(I)Landroidx/fragment/app/Fragment;
HSPLandroidx/fragment/app/FragmentStore;->findFragmentIndexInContainer(Landroidx/fragment/app/Fragment;)I
HSPLandroidx/fragment/app/FragmentStore;->getActiveFragmentStateManagers()Ljava/util/List;
HSPLandroidx/fragment/app/FragmentStore;->getActiveFragments()Ljava/util/List;
HSPLandroidx/fragment/app/FragmentStore;->getFragmentStateManager(Ljava/lang/String;)Landroidx/fragment/app/Fragm
HSPLandroidx/fragment/app/FragmentStore;->getFragments()Ljava/util/List;
HSPLandroidx/fragment/app/FragmentStore;->makeActive(Landroidx/fragment/app/FragmentStateManager;)V
HSPLandroidx/fragment/app/FragmentStore;->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

Формирование Baseline профиля для приложения

— Вы можете настроить формирование Baseline профилей с помощью библиотеки Jetpack Macrobenchmark

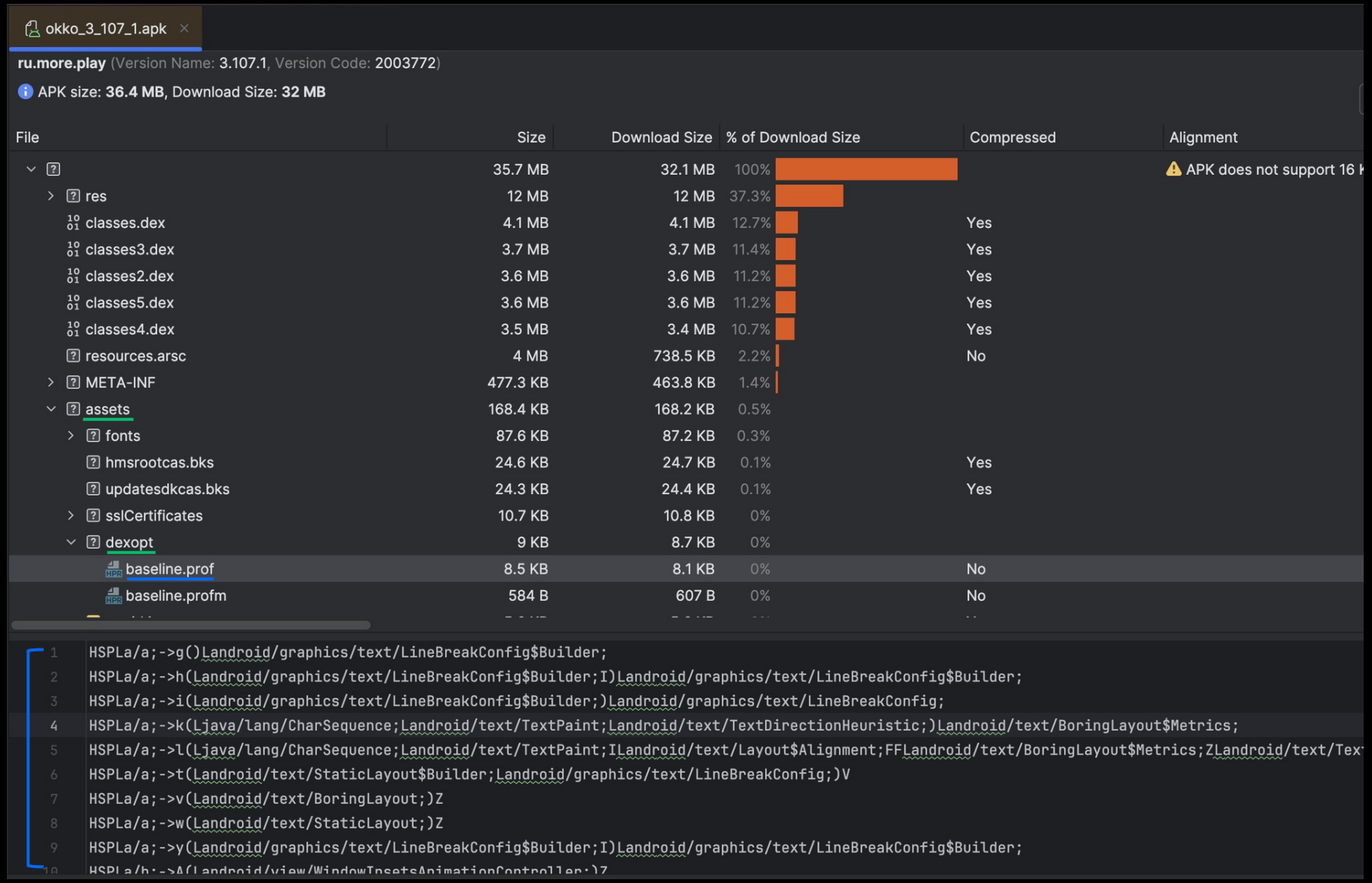
— Для формирования профилей нужно создать инструментальный тест и описать в нем основные сценарии работы приложения

— А можно просто использовать готовый шаблон, который автоматически создаст тест и сформирует базовый сценарий запуска приложения

— Дальше можно запускать эту таску на исполнение и она сама создаст и подложит в apk сформированный профиль

Ссылка на документацию Google

Baseline профиль в APK



SBER DEVICES

Анализ профилей приложений кинотеатров

Приложение	Профиль	Размер	Ускорение	Качество профиля
VK Video	нет			
Wink	есть	2.7 кб		Слабо (только Android методы)
Smotreshka	есть	3.2 кб		Слабо (только Android методы)
Okko	есть	8.5 кб	16%	Слабо (только Android методы)
IVI	есть	12.1 кб	15%	Слабо (примерно 20 собственных методов)
Kion	есть	10.9 кб	16%	Слабо (примерно 50 собственных методов приложения)
Premiere	есть	15.7 кб	20%	Отлично (более 4 000 собственных методов)
Kinopoisk	есть	31.2 кб	20%	Отлично (более 4 000 собственных методов приложения)

Инженеры Google придумали крутую штуку

При помощи Baseline профилей они скрестили JIT и AOT компиляцию

Baseline профили позволили взять все плюсы из JIT и AOT компиляции и почти избавиться от их минусов

Baseline профили реально ускоряют первые запуски приложений на 15-20%

Для нас это абсолютно бесплатно и этим надо активней пользоваться)

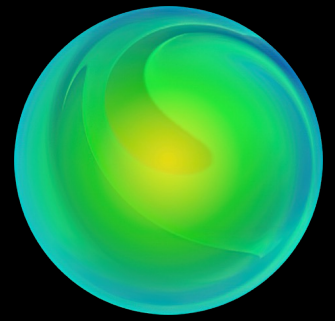
Спасибо за внимание

linkedIn:
sidorov-max

linkedIn:
kotlinovsky

Максим Сидоров,
Максим Митюшкин

Системные сервисы, Salute TV



SBER DEVICES