Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра системного программирования

Поздин Дмитрий Евгеньевич

Оптимизированная интерпретация языка PostScript

Выпускная квалификационная работа специалиста

Допущена к защите.
Зав. кафедрой:
д.ф.-м.н., проф. А.Н. Терехов

Научный руководитель:
к.ф.-м.н., доц. Д.Ю. Булычев

Рецензент:
к.ф.-м.н., доц. Д.В. Кознов

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY
Chair of Software Engineering

Dmitri Pozdin

Optimised PostScript interpretation

Graduation Thesis

Adnitted for defence.
Head of the chair:
Professor Andrey Terekhov

Scientific supervisor:
Dmitri Boulytchev

Reviewer:
Dmitry Koznov

Saint-Petersburg
2015

Оглавление

Введение

Язык PostScript — это специализированный кроссплатформенный язык программирования, разработанный в начале восьмидесятых годов в компании Adobe Systems для поддержки вывода на печать документов, содержащих векторную и растровую графику. Приложения на языке PostScript представляют собой программу для построения рисунков и текстов. Основная цель создания языка заключалась в том, чтобы обеспечить аппаратно-независимую поддержку сложной цифровой графики. Язык получил широкое распространение, интерпретаторы PostScript используются в принтерах, редакторах документов (TEX, Adobe Reader) и графических редакторах (Photoshop, Inkscape) в виде программных или аппаратных компонентов. На данный момент широко известны следующие промышленные реализации интерпретатора PostScript: GhostScript [1], Adobe PostScript [2].

Для того, чтобы исполнять на разных устройствах PostScript-программы, для каждого из них необходим свой интерпретатор или один кроссплатформенный интерпретатор. В качестве базы для кроссплатформенного интерпретатора можно взять Java Virtual Machine, поскольку она реализована для большого числа платформ и архитектур.

В лаборатории JetBrains математико-механического факультета СПбГУ был разработан интерпретатор на платформе Java для языка Postscript (далее — интерпретатор JB). В текущий момент на основе интерпретатора ведётся разработка оконного менеджера и библиотеки примитивов для реализации графического интерфейса приложений с помощью языка PostScript. Вся работа делится на три части: оптимизированная интерпретация языка PosctScript, механизм обработки событий и реализация графических примитивов и оконного менеджера.

Из-за двухуровневой модели исполнения (уровень Postscript и уровень Java) у интерпретатора JB недостаточно высокая скорость работы по сравнению с аналогами, и поэтому существует необходимость улучшить его производительность. В процессе анализа задачи оптимизации существующего интерпретатора было выявлено два узких места.

Первой проблемой было то, что процедуры в PostScript-программах исполняются много раз, при этом каждый раз они интерпретируются заново, что существенно замедляет работу интерпретатора. Данную проблему можно решить с помощью динамической компиляции процедур в байт-код JVM.

Следующей выявленной проблемой было повторное исполнение объектов-имен (англ. "name" [3]), во время которого происходит поиск сохранённого по имени значения в стеке словарей. Для оптимизации интерпретатора было принято предположение о том, что во многих случаях объекты-имена не перезаписываются, и для улучшения скорости работы интерпретатора можно использовать кэширование имен во время компиляции процедур во избежание повторного поиска по словарям, что называется спекулятивной компиляцией.

1 Постановка задачи

Цель данной дипломной работы заключалась в оптимизации работы интерпретатора JB для повышения скорости его работы. Для этого необходимо решить следующие задачи.

- Выполнить оптимизацию исполнения программ на языке PostScript в среде JVM посредством динамической компиляции (JIT).
- Выполнить оптимизацию исполнению программ на языке PostScript в среде JVM посредством спекулятивной компиляции.
- Провести эксперименты по замеру производительности и проанализировать полученные результаты.

2 Обзор

В данном разделе рассмотрены особенности языка PostScript, интерпретатора GhostScript, интерпретатора JB, проекта по созданию графической библиотеки в целом и инструменты, использовавшиеся при разработке.

2.1 Язык Postscript

Данный язык является динамически типизированным, стековым языком с обратной польской нотацией. Он предназначен для отрисовки сложной векторной и растровой графики. Программы на PostScript обычно создаются другими приложениями, но при этом на PostScript можно писать как на обычном языке программирования.

Объекты

Программа на языке PostScript состоит из токенов (простейших лексических единиц), каждый из которых является объектом языка. Простые объекты хранят свое непосредственное значение, в то время как сложные — только ссылку на него. В языке используются следующие примитивные типы: `int`, `real`, `boolean`, `name`, `operator`, `mark`. Поддерживаются также сложные типы объектов, такие как: `string`, `array`, `dictionary`, `save`. Все объекты могут быть либо литеральными (от англ. "literal"), и в этом случае объект просто кладется на стек во время исполнения, либо исполняемыми. Это свойство записано в соответствующем атрибуте объекта.

Имена (тип `name`) — это аналог переменных в других языках. Литеральное имя используется для записи, а исполняемое — для вызова соответствующего значения. Поскольку в PostScript используется стратегия вычислений `вызов-по-имени` (англ. "call-by-name"), то связывание имен со значением про-

исходит в момент исполнения это имени. Кроме того, любое имя можно переопределить в любом месте программы, в том числе и то, которое связано со стандартным оператором.

Массивы (тип `array`) бывают литеральными, и тогда они являются структурами данных, или исполняемыми, тогда они играют роль процедур. Процедуры — это набор объектов, которые необходимо исполнить. В тексте программы процедуры выглядят как блок кода, ограниченный фигурными скобками.

Архитектура

В интерпретаторе языка есть четыре стека: графический стек, стек операндов, стек словарей и стек вызовов. Каждый из них отвечает за свои функции. Стек операндов хранит аргументы для операторов. Стек словарей — это структура, которая хранит данные о всех переменных. В каждом словаре, который находится на этом стеке, имена ассоциированы со своими значениями. В графическом стеке хранятся настройки графики, например: цвет, текущий путь и др. Стек вызовов отвечает за исполнение программы, каждый его элемент является процедурой.

Механизм исполнения

Исполнение программы на PostScript происходит "объект за объектом". Например, если очередной объект — исполняемый массив, то он кладется на стек вызовов в качестве новой процедуры, и после этого исполняются объекты уже в этой процедуре. Таким образом, на стеке вызовов находятся прерванные процедуры, и на вершине стека — текущая исполняемая процедура.

Виды операторов

В PostScript существует большое количество операторов, которые находятся в одном словаре, называющимся системным. Среди операторов можно выделить несколько больших групп: арифметические, логические, для работы с стеками и разными типами объектов, для построения пути, рисования, и прочие.

2.2 Обзор промышленного интерпретатора Ghostscript

Одной из промышленных реализаций интерпретатора PostScript является GhostScript, продукт компании Artifex Software [4]. Первая версия была выпущена в 2000 году, и новые версии выходят до сих пор. GhostScript перенесён для операционных систем Linux, Mac OS X, Unix-подобных и Windows.

2.3 Описание интерпретатора языка PostScript

В качестве платформы для интерпретатора JB выбрана Java. Этот выбор был обусловлен тем, что эта платформа обеспечивает кроссплатформенность. Интерпретатор JB позволяет исполнять файлы PostScript, выводя на экран изображение, которое является результатом работы программы. Реализованы значительное подмножество языка, общая архитектура, все основные типы данных, механизм исполнения.

Проект по созданию интерпретатора делился на три основные части:

- Реализация общей поддержки времени исполнения для интерпретатора языка PostScript [5].
- Реализация графической части интерпретатора языка PostScript [6].

- Архитектура интерпретатора для исполнения программ на языке PostScript в JVM [7].

2.4 Интерпретатор JB и GhostScript

Интерпретатор GhostScript рассматривался в качестве эталонного при разработке интерпретатора JB. В GhostScript реализованы все системные операторы из спецификации. Поэтому именно он использовался для проверки корректности работы операторов и построения рисунка в целом в интерпретаторе JB.

2.5 Описание проекта по созданию графической библиотеки на основе PostScript

На основе интерпретатора JB в лаборатории JetBrains разрабатывается графическая библиотека. При этом все графические примитивы (кнопки, окна и т.д.) хранятся и отрисовываются при помощи PostScript. Оконный менеджер является интерактивным, т.е. в нем перехватываются события мыши и клавиатуры, происходит взаимодействие и перерисовка изменившихся элементов интерфейса. Преимущество данного подхода заключается в гибкости построения графического интерфейса, при этом в разных системах интерфейс будет выглядеть одинаково. Кроме того, в оконном менеджере можно добавить свои эффекты, например, переворот окна или эффект «волны».

Для обеспечения высокой производительности оконного менеджера проводится оптимизация его работы, описанная в данной дипломной работе. Сам проект разделяется на два основных направления работы: механизм обработки событий в графической библиотеке и реализация примитивов и оконного менеджера (см. рис. 1).

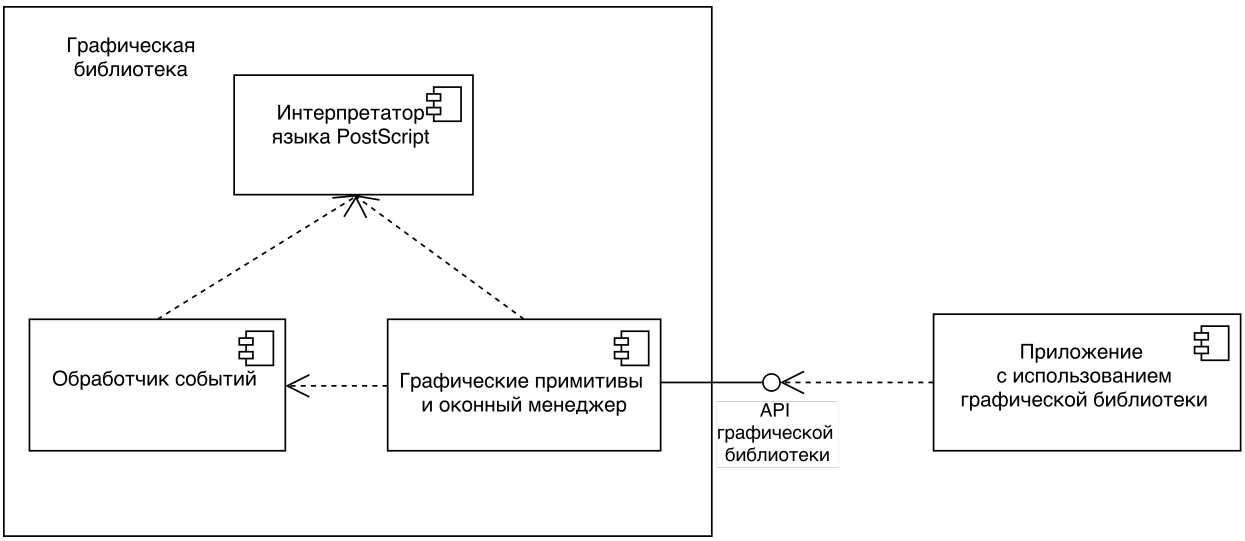

Рис. 1: Схема проекта

Контексты и многопоточность

В первоначальной реализации интерпретатора JB все основные компоненты: все стеки, локальная память и интерфейс к ним — находились в главном классе `Runtime`, в котором был реализован шаблон ООП «одиночка» (англ. "singleton") [8], и из любого места интерпретатора можно было к нему обратиться. Данная схема подходила только для однопоточного исполнения программ на PostScript. Спецификация языка вообще не предусматривает многопоточность, которая необходима для реализации графических эффектов в графической библиотеке.

Таким образом, возникла необходимость расширить язык и оптимизировать его для многопоточного исполнения. Каждый поток должен иметь свой набор стеков и оперировать ими. Теперь каждый поток хранит свой контекст исполнения, в котором содержатся стеки, интерфейсы к ним и другие данные, необходимые потоку. В классе `Runtime` остались только локальная память и механизм управления потоками. (см. рис. 2)

При такой многопоточной схеме в момент выполнения каждому объекту нужно передать контекст, в котором он исполняется. В однопоточной схеме,

которая обсуждалась выше, это было не нужно.

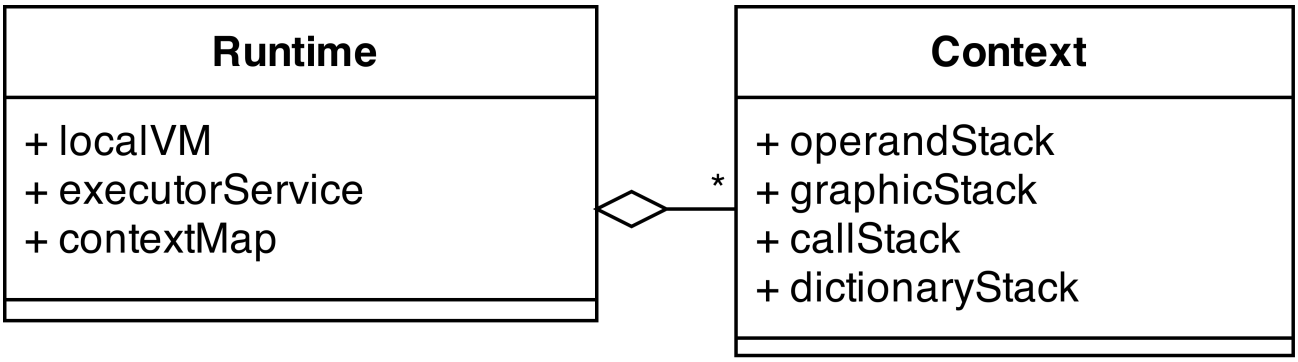

Рис. 2: Схема Runtime и контекстов

2.6 Библиотека ASM для генерации и изменения байт-кода JVM

Библиотека ASM предназначена для создания, изменения и анализа байт-кода Java. Данная библиотека разработана в компании OW2 Consortium [9]. Создание классов происходит вызовом соответствующих методов из библиотеки для добавления очередной инструкции JVM. Работа Java Virtual Machine и использование её инструкций подробно описано в спецификации JVM [10]. Этот инструмент очень полезен, если необходимо создавать скомпилированные Java-классы уже после запуска Java-приложения. Кроме того, собранные классы можно изменять перед загрузкой. Данная функциональность библиотеки активно используется в оптимизации методом динамической компиляции.

3 Динамическая компиляция процедур

Для повышения производительности интерпретатора JB были проанализированы узкие места, в которых тратилось большее количество времени. Таковыми оказались процедуры или по-другому исполняемые массивы, которые составляют большую часть программ на PostScript. Они используются в циклах, условных операторах и при их вызове по имени, то есть как вызов функции. В интерпретаторе JB каждый раз при повторном выполнении процедуры происходит повторная интерпретация этой процедуры, например в очередной итерации цикла. На повторную интерпретацию процедур тратится большое количество времени, в то время как его можно уменьшить за счет компиляции процедур. Оптимизация интерпретации программ на языке PostScript посредством динамической компиляции процедур исключает проблему повторной интерпретации, динамически компилируя процедуры при их объявлении и после используя скомпилированный байт-код для вызовов процедур. Эта оптимизация изменила процесс интерпретации: теперь стек вызовов не используется, а вместо процедур создаются «на лету» динамически скомпилированные классы в байт-коде JVM. Далее будет подробно рассмотрена схема создания и использования байт-кода (см. рис. 3) и особенности реализации некоторых операторов в данной оптимизации.

3.1 Генерация байт-кода

`BytecodeGenerator` — главный класс для создания класса в байт-коде для процедуры. Он отвечает за имя класса, наследование, поля, создание конструктора, статическую инициализацию и за объявление начала и конца метода. Содержимое самих методов заполняется инструкциями JVM во время компиляции конкретных объектов. Для добавления инструкций в метод

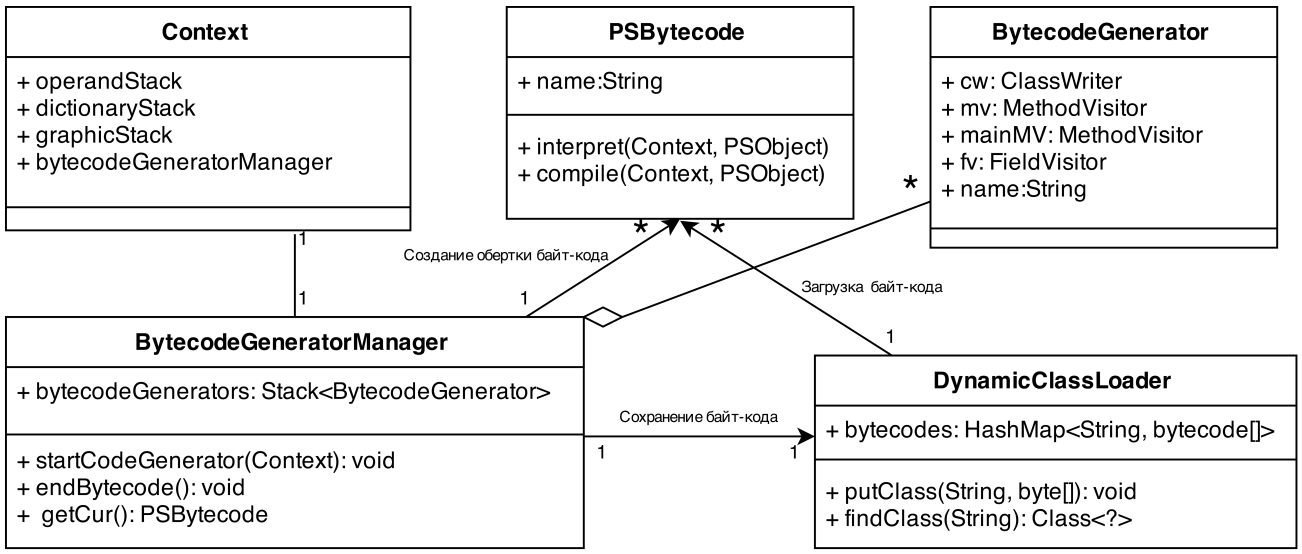

Рис. 3: Схема классов для создания байт-кода

используется соответствующий метод класса `MethodVisitor` (из библиотеки ASM), например, как в листинге 1.

```
MethodVisitor mv = context.bcGenManager.mv;
mv.visitInsn(DUP);
mv.visitTypeInsn(NEW, "psObjects/values/simple/numbers/PSInteger");
mv.visitInsn(DUP);
mv.visitLdcInsn(value);
mv.visitMethodInsn(INVOKESPECIAL,
  "psObjects/values/simple/numbers/PSInteger", "<init>", "(I)V", false);
mv.visitMethodInsn(INVOKESPECIAL,
  "psObjects/PSObject", "<init>", "(LpsObjects/values/Value;)V", false);
mv.visitFieldInsn(GETSTATIC, name, "context", "Lruntime/Context;");
mv.visitInsn(ICONST_0);
mv.visitMethodInsn(INVOKEVIRTUAL,
        "psObjects/PSObject", "interpret", "(Lruntime/Context;I)Z", false);
```

Листинг 1: Компиляция объектов `Integer`

Для работы с полями класса используется `FieldVisitor`. Создание класса осуществляется с помощью `ClassWriter`. После завершения компоновки клас-

са в байт-коде `BytecodeGenerator` возвращает массив байтов — скомпилированный класс.

Процедуры в PostScript могут быть в вложенными, т.е. объявление одной процедуры может быть внутри другой. Компиляция происходит «на лету», т.е. в момент объявления процедуры, из-за этого во время объявления вложенной процедуры создание для внешней ещё не закончено. Поэтому в один и тот же момент времени может быть не закончена интерпретация (а вместе с ней и создание массива) для нескольких процедур.

Для управления процедурами, для которых не завершена компиляция, используется класс `BytecodeGeneratorManager`. Каждому контексту исполнения (`Context`) соответствует один менеджер. В нем содержится стек из генераторов байт-кода (`BytecodeGenerator`), каждый из которых отвечает за свою незавершенную процедуру. На вершине стека находится тот генератор байт-кода, работа с которым идёт в данный момент. Через менеджер можно получить доступ к имени текущего класса и `MethodVisitor`, в котором определяется текущий метод класса. С помощью менеджера создаются новые генераторы (`startBytecodeGenerator(Context)`) и завершается создание байт-кода для процедуры (`endBytecode()`), после которого происходит сохранение этого байт-кода в `DynamicClassLoader` с помощью метода `putClass(String,byte[])`. Каждому имени скомпилированного класса сопоставляется массив байтов (`bytecodes: HashMap<String,byte[]>`).

Создание байт-кода по процедуре начинается, когда во входном потоке встречается токен `"{"`, означающий начало исполняемого массива. В этот момент создаётся новый генератор байт-кода и кладется на вершину стека генераторов. После этого для каждого следующего токена в байт-код добавляется несколько инструкций JVM, соответствующих интерпретации данного токена. Для разных типов объектов это происходит по-разному. Когда встре-

чается токен "}", то завершается создание байт-кода, и его генератор снимается со стека. При этом происходит сохранение скомпилированного класса в виде массива байтов в специальном хранилище, а на стек операндов кладётся объект `PSBytecode`, в котором хранится только имя скомпилированного класса.

Исполнение байт-кода

Исполнение байт-кода соответствует исполнению процедур в интерпретаторе JB, но реализуется по-другому. В момент вызова байт-код объекта из хранилища скомпилированных классов достаётся массив байтов, соответствующий этому объекту. Затем массив загружается загрузчиком классов (`ClassLoader`). С этого момента можно уже использовать динамически созданный класс: создавать объекты, вызывать методы и изменять поля у этого класса. Все создаваемые нами классы имеют главный статический метод `run()`, который выполняет всю работу. Именно этот метод вызывается для выполнения скомпилированной процедуры. После этого происходит исполнение всех инструкций JVM, которые были добавлены в этот класс в момент его генерации.

Во время выполнения байт-кода необходимо использовать внутреннее состояние контекста, а также его изменять (например, брать объект со стека операндов или класть на него). В момент создания байт-кода ещё неизвестно, в каком контексте и потоке он будет вызываться. Это происходит из-за того, что объекты можно передавать между потоками. Поэтому в момент вызова байт-кода передаётся в качестве единственного аргумента текущий контекст, он записывается в качестве поля.

Таким образом, исполнение байт-кода можно разбить на три пункта.

- Загрузка скомпилированного класса загрузчиком классов.
- Установление контекста выполнения для загруженного класса.
- Вызов статического метода `run()` у загруженного класса.

3.2 Отдельные сложные операторы

Из-за изменения схемы интерпретации некоторые стандартные операторы языка, связанные с массивами, реализуются гораздо сложнее, чем первоначально в интерпретаторе JB.

Оператор связывания `bind`

Данный оператор является системным и заменяет в исполняемом массиве все имена, ссылающиеся на стандартные операторы, на сами операторы. Во-первых, он предотвращает лишний поиск по словарям. Во-вторых, если после вызова `bind` переопределится какой-то из операторов, то процедура не изменит хода своего исполнения и будет работать как раньше. Таким образом обеспечивается безопасность в том смысле, что вне зависимости от того, переопределятся ли стандартные операторы или нет, процедура не изменит своего поведения. В примере 2 перезаписывается оператор `add`, но это не влияет на работу программы — в конце работы программы на стеке лежит число 5. Если бы оператора `bind` не было в программе, то на вершине стека было бы число 6. Напомним, что оператор `def` записывает по имени объект, а префикс "/" используется для обозначения литеральных имен.

Реализация оператора `bind` осложняется в данной оптимизации тем, что вместо исполняемых массивов используется байт-код, то есть собранный и скомпилированный класс. Причём байт-код нельзя изменять после загрузки

```
/a {2 3 add} bind def
/add {mul} def
a
```

Листинг 2: пример на PostScript на оператор `bind`

в JVM. Таким образом, нужно изменять динамически созданный байт-код до загрузки в JVM, что накладывает некоторые ограничения на данную оптимизацию. Оператор `bind` должен быть вызван до первого вызова процедуры, к которой он относится. Библиотека ASM позволяет модифицировать методы в байт-коде, добавлять и удалять инструкции JVM, то есть инструментировать байт-код.

Для удобства модификации выделим все «подозрительные» имена в отдельные методы. Здесь «подозрительными» мы назвали те имена, которые ссылаются на системные операторы в момент создания байт-кода. При этом на момент выполнения оператора `bind` совсем не обязательно, что эти имена всё еще ссылаются на системные операторы. Они могут быть перезаписаны. Именно по этому они «подозрительные».

Поскольку теперь главный метод `run()` разбит на части, то он выглядел бы следующим образом (см. листинг 3), если бы был он написан на Java. Каждый отдельный метод называется `run_<число>`.

```
public static void run(){
        run_1();
        run_2();
        ...
        run_m();
}
```

Листинг 3: Структура метода run()

В момент компиляции мы запоминаем, какие методы были «подозритель-

ными», то есть содержали вызов «подозрительного» имени. Запись производится в хеш-таблицу (Hash Map). Ключ — это пара из номера класса (все классы байт-кода пронумерованы) и номера «подозрительного» метода. А значение, ассоциированное с ключом, является названием того оператора, на который ссылается это подозрительное имя.

Итак, общий алгоритм оператора `bind` состоит из трёх пунктов:

1. Обнаружение всех «подозрительных» методов в классе с помощью хеш-таблицы.

2. Выполнение проверки того, перезаписались ли соответствующие им имена. Используются только те, которые всё ещё ссылаются на системные операторы.

3. Модификация найденных методов:

   (a) удаление всех инструкций JVM в этих методах кроме последних трёх служебных инструкций;

   (b) добавление инструкций, которые вызывают нужный оператор.

После модификации получается уже искомый байт-код, в котором заменены вызовы имен на операторы.

Оператор `cvx`

В языке PostScript системный оператор `cvx` изменяет атрибут объекта: литеральный делает исполняемым. Поскольку схема интерпретации изменена только для процедур (исполняемых массивов), то с ними возникают некоторые сложности, когда из массива необходимо сделать байт-код. Необходима компиляция массива целиком со всеми объектами. Отличие компиляции на

«лету» от компиляции байт-кода по массиву состоит в том, что в первом случае компилируются только отдельные токены, и поэтому компиляция составных объектов, таких как массивы и словари, не происходит. Они создаются уже на этапе выполнения байт-кода, поскольку состоят из нескольких токенов. Во время компиляции байт-кода по массиву, необходимо компилировать любые входящие в массив объекты, в том числе составные.

Таким образом, для оператора `cvx` необходимо дополнительно реализовывать компиляцию сложных объектов, значения которых отделены от объекта и загружены в память, локальную или глобальную. Проблема состоит в том, что в контекст байт-кода можно передавать только неизменяемые объекты и примитивы (с точки зрения Java), которые хранятся в пуле констант Java, а именно: строки и числа. Любые композитные объекты нельзя передавать, т.к. они могут быть изменены.

В случае локальной памяти, где значения сложных объектов хранятся по номеру, данная проблема решается просто. Для восстановления сложного объекта в контексте байт-кода нужно передать его номер в локальной памяти и атрибуты, закодированные с помощью чисел. Так при исполнении байт-кода создаются новые объекты со старыми значениями и атрибутами.

В случае глобальной памяти, где значения хранятся не по номеру в таблице, а по ссылке, решить проблему тем же способом не удаётся. Поэтому самый простой и очевидный способ — это завести специальную таблицу для сложных объектов, находящихся в глобальной памяти, которые загружаются в байт-код оператором `cvx`. Из-за того что это довольно редкое событие (оператор `cvx` нечасто используется, и, в основном, используют локальную память), данное решение не влечет бóльших накладных расходов. В этой таблице по номеру можно найти значение из глобальной памяти, которое нужно загрузить в байт-код.

Таким образом осуществляется компиляция сложных объектов, которая потребовалась в реализации оператора `cvx`. Простые же операторы компилируются аналогично тому, как это происходит в компиляции на «лету», передавая в байт-код своё примитивное значение.

Оператор выгрузки массива `aload`

Данный оператор в некотором смысле противоположен оператору `cvx`. Он разгружает массив (при этом не важно, исполняемый или литеральный), т.е кладёт на стек операндов все элементы массива и в конце сам массив. Сложности возникают в случае исполняемого массива, т.е байт-кода в данной оптимизации.

Казалось бы, всё что можно сделать с байт-кодом — это вызвать его, что значит исполнить все скомпилированные в нем объекты. Но это не то, что требуется в данном операторе. Поэтому реализовано некое «литеральное исполнение» объекта, когда он просто кладётся на стек. Во время вызова любого объекта будем проверять специальный флаг (`aloading`). Если в нём записана истина, то вместо исполнения конкретного значения, этот объект сразу кладётся на стек.

Поэтому реализация оператора `aload` выглядит следующим образом.

1. Установить флаг `aloading = true` в текущем контексте.
2. Вызвать байт-код.
3. Положить байт-код на стек операндов.
4. Установить флаг `aloading = false` в текущем контексте.

Для правильного исполнения оператора `aload` необходимо, чтобы из байт-кода вызывалось исполнение каждого объекта (т.е. централизованно через

один метод). Из-за этого компиляция всех объектов состоит из двух основных частей: «сборка» объекта (внутри него «сборка» его значения) и вызов объекта с помощью метода `iterpret()` как показано в листинге 1 на странице 14. Там происходит создание `PSInteger` по переданному значению, создание объекта `PSObject` на его основе и вызов интерпретации этого объекта с нужными параметрами.

4 Спекулятивная компиляция

Объекты-имена встречаются в программах на языке PostScript очень часто, в них записываются либо конкретные неисполняемые значения (в этом случае имена выполняют роль переменных), либо процедуры, исполняемые массивы (здесь имена выполняют роль вызова процедур). Кроме того, в этих двух случаях можно выделить следующие особенности. Имена, ссылающиеся на процедуры, крайне редко перезаписываются, в то же время имена, ссылающиеся на неисполняемые объекты, довольно часто перезаписываются. При этом последних обычно значительно меньше.

Исполнение объекта-имени в интерпретаторе JB происходит следующим образом. Перебираются словари, находящиеся на стеке словарей. Если в текущем словаре есть данное имя и ассоциированное с ним значение, то поиск прекращается, и исполняется найденное значение. При этом словарь является деревом, как структура данных. Получается, что поиск ассоциированного с именем значения — долгая и дорогостоящая операция.

В данной оптимизации можно предотвратить повторный поиск по имени при многократном исполнении процедуры, когда ассоциированное значение уже найдено при определении этой процедуры, и после оно не менялось. В этом и состоит спекулятивная компиляция.

4.1 Особенности реализации

Для этой оптимизации изменён способ компиляции для исполняемых объектов-имён. Сначала происходит поиск ассоциированного с именем значения. Если оно не найдено, то компиляция для этого имени происходит без данной оптимизации, т.е. генерируется код, отвечающий за поиск и вызов значения. Поскольку в PostScript используется стратегия вычислений `вызов`

`по имени` (англ. "call-by-name"), то значение имени может быть определено позже, чем определение процедуры, использующее это имя.

Если ассоциированное значение найдено, то оно записывается в переменную и используется далее при кэшировании. Для проверки актуальности закэшированного значения сравнивается «версии имён» на момент компиляции и на момент исполнения. Во время новых определений для имен и в момент обновления стека словарей сохраняется номер текущей версии для каждого имени. Сравнение версий происходит внутри байт-кода. Если версии имен совпадают, то исполняется закэшированное значение, иначе имя исполняется без оптимизации. Повторного кэширования не происходит, поэтому имена, у которых изменено значение хотя бы один раз при вызове какой-то процедуры (например, счетчик), исполняются в этой процедуре без оптимизации.

5 Тестирование

Во процессе разработки оптимизаций регулярно выполнялось регрессионное тестирование с целью проверки корректности работы основных операторов языка PostScript, используя unit-тесты. Также вместе с разработкой пополнялась база тестов. Каждый из этих тестов содержал короткую программу на PostScript, использовавшая только определенный набор операторов. В результате работы этой программы на стеке операндов появляются конкретные значения. Производилось сравнение состояния стека операндов после работы интерпретатора в двух режимах: с оптимизациями и без них. Поскольку все тесты прошли проверку успешно, то можно сделать вывод, что не смотря на то, что в данных оптимизациях изменен способ интерпретации, семантика выполнения программ осталась неизменна.

Далее была протестирована производительность интерпретатора JB с учетом оптимизаций на значительном наборе тестов-рисунков. Каждый тест — это программа на языке PostScript, в результате работы которой на экране показывается изображение. Измерено время работы интерпретатора в трёх режимах: без оптимизаций, с одной оптимизацией (динамическая компиляция) и сразу с двумя оптимизациями (спекулятивная компиляция работает только в вместе с динамической компиляцией). Все тесты можно условно поделить на две большие группы: простые и сложные. Простые — это тесты, в которых содержится мало трудоёмких операций и время их выполнения не превышает двух секунд с использованием оптимизации (это разделение довольно условное). Остальные — сложные. Из-за большого количества трудоёмких операций они выполняются значительно дольше.

Сравним быстродействие интерпретатора JB с оптимизациями и без них на компьютерах с системными параметрами, указанными в таблице 1.

Процессор	Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
Оперативная память	16 GB
Операционная система	Windows 8 x64

Таблица 1: Конфигурация компьютера, на котором проводилось тестирование

Простые тесты

Результаты тестирования набора простых тестов приведены в табл. 2 и на двух графиках на рис. 4. Между собой тесты не связаны и упорядочены в порядке увеличения времени. На оси абсцисс отмечены номера тестов. Два графика понадобилось для лучшего визуального представления, поскольку диапазон времени исполнения слишком большой: от двух миллисекунд до 4 секунд.

На первом графике, где представлены тесты с наименьшим временем, видно, что интерпретация без оптимизации почти везде работает быстрее, чем с оптимизациями. Это связано с накладными расходами, возникающими при компиляции. На втором графике, на котором представлены тесты с чуть большим временем выполнения, можно заметить, что оптимизации дают выигрыш во времени, но, в основном, незначительный. Из этого можно сделать вывод, что оптимизация интерпретатора на таких простых и коротких программах не требуется.

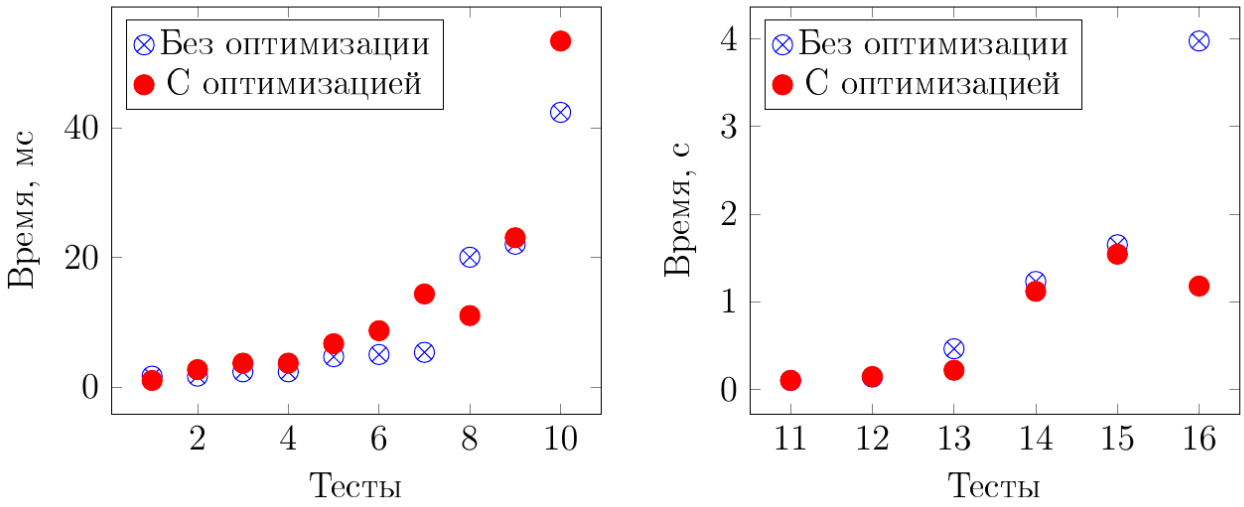

Рис. 4: Время работы простых тестов

Название теста	**Без оптимизации, мс**	**С оптимизацией, мс**
7_ellipses.ps	1,667	2,667
6_arcs.ps	1,667	1,000
1_rectangles.ps	2,333	3,667
5_star.ps	2,333	3,667
flower.ps	4,667	6,667
2_trapezoid.ps	5,000	8,667
drop.ps	5,333	14,333
1_clip.ps	20,000	11,000
4_circles.ps	22,000	23,000
tiger.eps	42,333	53,333
colorcir.ps	100,667	101,000
doretree.ps	139,333	145,333
WireFrame.eps	462,333	217,000
gingerbread.ps	1649,333	1541,667
6_Fractal_Arrow.ps	1230,667	1118,333
chupcko.ps	3975,333	1178,000

Таблица 2: Время работы быстрых тестов

Сложные тесты

Программы на PostScript, которые входят в набор сложных тестов, работают продолжительное время, поскольку в них используется большое количество математических операций и вычислительно-сложные алгоритмы, например, построение фрактальных рисунков, рендеринг методом трассировки лучей для каждой точки.

Для сложных тестов выполнены замеры времени выполнения интерпретатора в трех режимах: оригинальная интерпретация (без оптимизаций, рассмотренных в данной работе), интерпретация с одной оптимизацией (динамическая компиляция) и интерпретация с двумя оптимизациями (динамическая и спекулятивная компиляция). Отдельно оптимизация посредством спекулятивной компиляции не представлена, поскольку она работает только вместе с динамической компиляцией. Результаты тестирования представлены на рис. 5 и в табл. 5. В таблице указано для каждого из восьми тестов среднее время выполнения и среднее отклонение во всех режимах, в каждом из которых тесты были запущены по 15 раз. На графике указано, в какое количество раз увеличилась скорость работы интерпретатора JB при использовании одной или двух оптимизаций по сравнению с оригинальной интерпретацией.

Для сложных тестов заметна разница между временем исполнения программ в интерпретаторе в разных режимах. Это происходит из-за того, что в этих тестах в большом количестве повторно используются процедуры. Процедуры в оптимизации посредством динамической компиляции интерпретируются один раз, при этом создается байт-код по процедуре, который исполняется каждый раз при вызове этой процедуры. В то время как в оригинальной интерпретации процедуры интерпретируются каждый раз заново. Спекулятивная компиляция ещё сокращает время работы для большинства тестов.

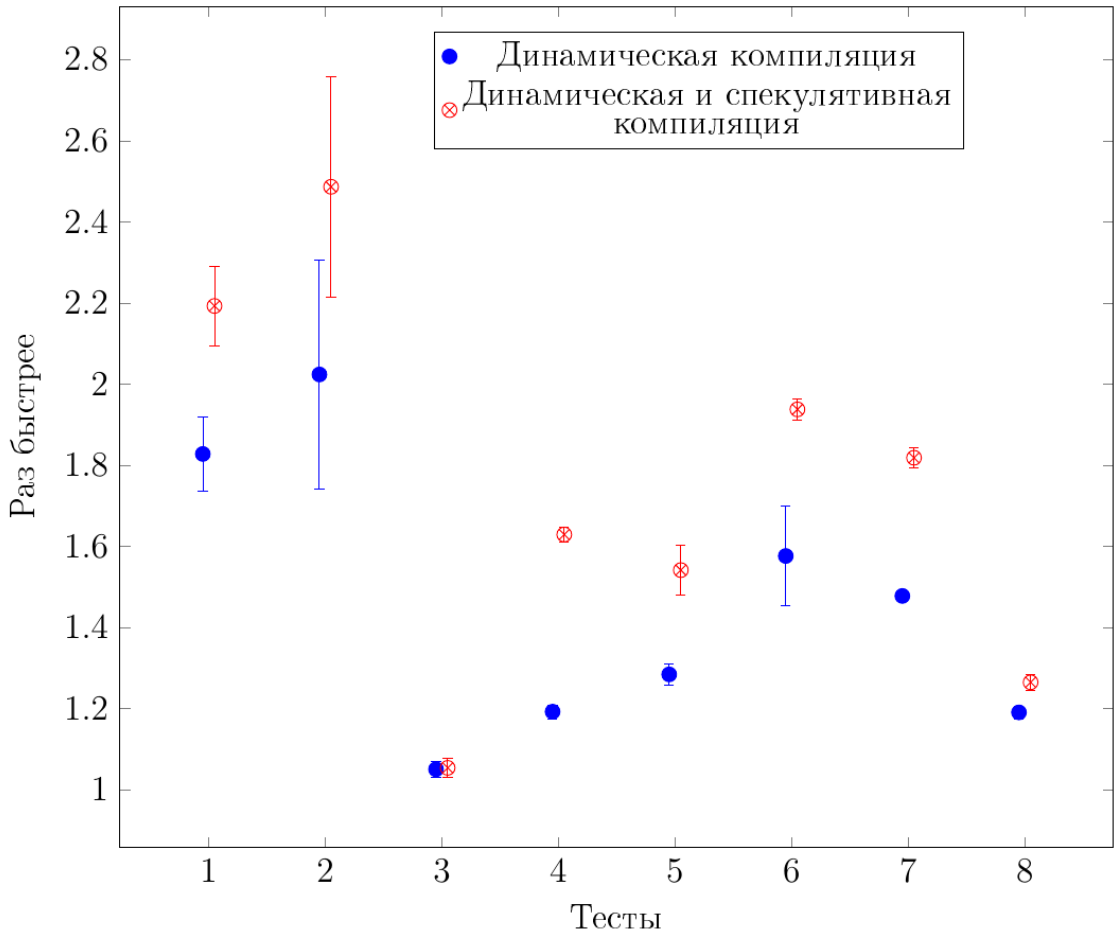

Рис. 5: Ускорение оптимизаций на сложных тестах

Номер	**Название теста**	**Оригинальная интерпретация, с**	**Динамическая компиляция, с**	**Динамическая и спекулятивная компиляция, с**
1	chupcko.ps	2.427 ± 0.073	1.328 ± 0.082	1.107 ± 0.075
2	bubbles.ps	10.929 ± 1.399	5.401 ± 0.837	4.396 ± 0.633
3	henon.ps	12.622 ± 0.088	12.017 ± 0.144	11.977 ± 0.204
4	mandelbrotset.ps	26.260 ± 0.105	22.025 ± 0.286	16.120 ± 0.226
5	julia.ps	123.845 ± 2.477	96.437 ± 0.579	80.351 ± 3.375
6	psRay.ps	150.695 ± 0.753	95.612 ± 11.187	77.780 ± 1.633
7	mandel.ps	176.550 ± 0.353	119.486 ± 0.717	97.100 ± 2.233
8	1dca.ps	323.739 ± 1.942	271.925 ± 2.719	255.973 ± 3.328

Таблица 3: Время работы сложных тестов

По результатам тестирования сложных тестов можно сделать вывод, что рассмотренные в данной работе оптимизации позволяют получить выигрыш во времени до двух раз на некоторых сложных, затратных задачах.

Заключение

В рамках дипломной работы получены следующие результаты.

- Выполнена оптимизация исполнения программ на языке PostScript в среде JVM посредством динамической компиляции (JIT).
  - Проведена динамическая компиляция процедур «на лету» (при их инициализации) способом генерации байт-кода JVM (библиотека ASM).
  - Осуществлена замена стека исполнения стеком генераторов байт-кода.
- Выполнена оптимизация исполнения программ на языке PostScript в среде JVM посредством спекулятивной компиляции.
  - Компиляция проводится при предположении, что значения большинство имен, ссылающихся на процедуры, не меняются.
  - Производится кэширование имен при компиляции, если по имени записано значение.
- Выполнены эксперименты по замеру производительности, показавшие увеличение быстродействия в два раза по сравнению со стандартной реализацией.

Список литературы

[1] Официальная страница интерпретатора GhostScript.
`http://www.ghostscript.com/`

[2] Официальная страница интерпретатора Adobe PostScript.
`http://www.adobe.com/products/postscript/`

[3] Спецификация языка Postscript. PostScript Language reference.
Adobe Systems. 1999
`http://www.adobe.com/products/postscript/pdfs/PLRM.pdf`

[4] Официальная страница компании Artifex Software.
`http://www.artifex.com/`

[5] Дмитрий Поздин. Реализация общей поддержки времени исполнения для интерпретатора языка PostScript.// Труды лаборатории языковых инструментов. Выпуск 2. 2014. с. 276-296.

[6] Артур Гудиев. Реализация графической части интерпретатора языка PostScript.// Труды лаборатории языковых инструментов. Выпуск 2. 2014. с. 297-312.

[7] Рустам Макулов. Архитектура интерпретатора для исполнения программ на языке PostScript в JVM. // Труды лаборатории языковых инструментов. Выпуск 2. 2014. с. 259-275.

[8] Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидс.
Приёмы объектно-ориентированного проектирования. Паттерны проектирования. 1994

[9] Официальная страница компании OW2 Consortium.
www.ow2.org/

[10] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley. The Java Virtual Machine Specification. Java SE 7 Edition, 2013.
docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf