МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Механико-математический факультет

Кафедра программирования

Направление подготовки 02.06.01 Компьютерные и информационные науки

Специализация: 05.13.11 Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

НАУЧНЫЙ ДОКЛАД ОБ ОСНОВНЫХ РЕЗУЛЬТАТАХ ПОДГОТОВЛЕННОЙ НАУЧНО-КВАЛИФИКАЦИОННОЙ РАБОТЫ (ДИССЕРТАЦИИ) АСПИРАНТА

Трепаков Иван Сергеевич

Тема работы: **Методы и средства оптимизирующей генерации служебных метаданных в управляемых средах**

«К защите допущена»	**Научный руководитель**
Д.ф.-м.н., профессор	К.ф.-м.н.
Заведующий кафедрой Прог	ИСИ СО РАН, зав. лаб.
Марчук А. Г./……	Бульонков М. А./……
(фамилия, И., О.) / (подпись, МП)	(фамилия, И., О.) / (подпись, МП)
«……»……20…г.	«……»……20…г.

Дата защиты: «……»……20…г.

Новосибирск, 2021

Оглавление

Введение

Разработка встраиваемых систем лежит в основе многих современных технологических направлений: от телекоммуникационных сетей и промышленного оборудования, до умного дома и беспилотных автомобилей. Если раньше под встраиваемыми системами понимали только небольшие микроконтроллеры, для которых приходилось писать программы на ассемблере, то сейчас многие устройства работают под управлением полноценной операционной системы [1], и все больше программного обеспечения пишется на высокоуровневых языках программирования с управляемой средой, таких как Java [2], Python [3] и C# [4]. При этом проблема ограниченности ресурсов до сих пор остается актуальной для встраиваемых систем, и ее решение требует эффективной реализации языков программирования и их сред исполнения.

В частности, требуется эффективная генерация служебных метаданных, которые требуются управляемой среде для эффективного исполнения кода программы. Эти метаданные представляют собой низкоуровневые структуры данных, вычисленные в результате компиляции программы, которые используются во время исполнения для поддержки таких возможностей, как динамическая диспетчеризация вызовов, проверки соответствия типа и точная сборка мусора. Однако размер порождаемых служебных метаданных велик и сравним с суммарным размером машинного кода программы [1]. Из-за этого при статической компиляции программы, когда все метаданные пишутся в исполняемый файл, неоправданно увеличивается размер файла на диске.

Одними из самых крупных структур служебных метаданных являются таблицы виртуальных и интерфейсных методов, которые используются для реализации динамической диспетчеризации полиморфных вызовов в объектно-ориентированных языках с интерфейсным наследованием (например, Java [5] и C# [6]). В этих языках, одиночное наследование классов позволяет эффективно реализовать виртуальные вызовы с помощью таблицы виртуальных методов, которая создается для каждого класса. Однако в случае множественного наследования интерфейсов, таких таблиц требуется несколько — по одной на каждый интерфейс, реализуемый классом.

[1] В качестве примера рассмотрим стандартную библиотеку для языка программирования Scala версии 2.13.0, включающую в себя около 2500 классов и интерфейсов. Если статически скомпилировать эту библиотеку, то на выходе получится исполняемый файл размером в 5 MB, из которых 2 MB (40 %) занимают таблицы виртуальных и интерфейсных методов.

Из-за этого возникает избыточность информации между ячейками разных таблиц, что и приводит к увеличению суммарного размера метаданных и, как следствие, исполняемого файла.

Большинство существующих реализаций таблиц интерфейсных методов жертвуют размером таблиц в пользу производительности. Так, например, реализации в CACAO [7] и SableVM [8], используют большие разреженные таблицы [9,10], требующие довольно трудоемкого процесса назначения глобальных номеров методам [11–14], без которого размер таблиц может неконтролируемо возрастать. В Jikes RVM [15] единственная интерфейсная таблица, несмотря на фиксированный размер, также может неконтролируемо расти за счет динамически генерируемых процедур разрешения конфликтов [16].

Большее внимание проблеме размера таблиц было уделено в системе Marmot [17], в которой используется схема вложенной раскладки интерфейсных таблиц, позволяющая уменьшить избыточность информации между ними. Однако, как отмечают сами авторы, данный подход не всегда дает уменьшение размера, и иногда может наоборот, увеличить суммарный размер таблиц. Аналогичная схема была реализована в SHAP [18] вместе с дополнительными оптимизациями скорости интерфейсного вызова [19].

Ни одна из существующих реализаций не дает гарантированного уменьшения размера таблиц по сравнению с базовой реализацией, что может оказаться неприемлемо при исполнении программы на встраиваемых системах. На разрешение данной проблемы и направлено диссертационное исследование.

Целью данной работы является сокращение занимаемого места на диске приложением путем уменьшения суммарного размера крупнейшей структуры служебных метаданных управляемой среды — таблиц виртуальных и интерфейсных методов. Для достижения данной цели были поставлены и решены следующие **задачи**:

1. Проанализировать существующие подходы к уменьшению размера таблиц виртуальных и интерфейсных методов.
2. Выявить основные источники избыточности в структуре таблиц.
3. Разработать подход к генерации таблиц, минимизирующий суммарный размер таблиц в исполняемом файле.

Научная новизна и результаты, выносимые на защиту:

1. Доказано отсутствие верхней границы суммарного размера таблиц для существующего подхода раскладки в Marmot, по сравнению с базовой версией раскладки. Показано существование иерархии для сколь угодно большого числа δ, на которой суммарный размер таблиц раскладки Marmot в δ раз больше размера таблиц базовой раскладки.
2. Разработан алгоритм совмещенной раскладки таблиц виртуальных и интерфейсных методов, включающий только непересекающиеся таблицы суперинтерфейсов в таблицу интерфейса-наследника. В отличие от существующих подходов, включающих абсолютно все таблицы суперинтерфейсов, для предложенного алгоритма доказана гарантированная верхняя оценка суммарного размера таблиц, не превышающая размера таблиц базовой раскладки.
3. Разработан алгоритм разностного кодирования таблиц виртуальных и интерфейсных методов в исполняемом файле с гарантированной верхней оценкой размера полученного закодированного представления и линейной временной сложностью восстановления исходной структуры таблиц.

Теоретическая значимость работы:

1. Доказана неограниченность размера таблиц при использовании раскладки Marmot, по сравнению с базовой раскладкой таблиц.
2. Доказано, что суммарный размер таблиц при использовании совмещенной раскладки не превосходит размера таблиц базовой раскладки.
3. Доказано, что размер закодированного представления таблиц с помощью разностного кодирования не превосходит исходного размера кодируемых данных при реалистичных ограничениях на количество типов и методов в программе.

Практическая значимость работы:

1. Разработанный алгоритм совмещенной раскладки таблиц позволяет добиться значительного уменьшения дублирования данных в таблицах, а также существенно уменьшить размер исполняемого файла на диске.
2. За счет того, что алгоритм совмещенной раскладки таблиц, как и реализация в Marmot, модифицирует только раскладку, данный подход не влияет на эффективность поиска таблицы для интерфейсного вызова и может применяться совместно с более эффективными методами поиска таблицы.

3. Разработанный алгоритм разностного кодирования позволяет значительно уменьшить размер исполняемого файла, и при этом, эффективно восстановить исходную структуру таблиц виртуальных и интерфейсных методов во время исполнения.
4. Предложенный метод разностного кодирования не зависит от деталей раскладки таблиц, и поэтому может применяться совместно с любым алгоритмом раскладки.

Апробация и реализация результатов исследования. Основные результаты работы докладывались и обсуждались на

- 56-й международной научной студенческой конференции “МНСК-2018” (Новосибирск, 2018);
- 58-й международной научной студенческой конференции “МНСК-2020” (Новосибирск, 2020);
- международной конференции “Иванниковские чтения” (Орел, 2020);
- семинарах кафедры программирования Новосибирского государственного университета в 2016-2020 годах.

Предложенные методы были применены в промышленной реализации языка Java “Excelsior JET Embedded 11.3 for Linux/ARM”, а также в экспериментальной сборке Huawei JDK с поддержкой статической компиляции.

1. Предметная область

Среди современных объектно-ориентированных языков программирования большую популярность получила концепция интерфейсного наследования, в которой все типы в программе делятся на два вида: *классы*, которые наследуются одиночно, и *интерфейсы*, которые наследуются множественно. При этом каждый класс может наследовать (*реализовывать*) произвольное количество интерфейсов, а интерфейсы не имеют состояния, то есть в них нельзя объявлять поля и от них нельзя создавать объекты. Все полиморфные вызовы в таких языках разделяются на два вида: *виртуальные*, если формальный тип объекта вызова является классом, и *интерфейсные*, если формальный тип — интерфейс. Пример иерархии с интерфейсным наследованием может быть записан следующим образом:

```
interface K                    { c() }
interface J                    { b() }
interface I extends J, K       { c() }
class B implements J           { a() }
class C extends B implements I { b() }
```

Данная иерархия будет использоваться во всех поясняющих примерах далее, если не сказано обратного.

1.1. Таблица виртуальных методов

В случае одиночного наследования классов, виртуальный вызов можно реализовать наиболее эффективно с помощью специальной структуры *таблицы виртуальных методов* (англ. *virtual method table*, *VMT*, *vtable*), которая применяется практически во всех исследовательских и промышленных реализациях языков, поддерживающих одиночное наследование. Таблица виртуальных методов для некоторого класса C представляет собой массив vmt_C адресов реализаций методов, доступный напрямую из любого объекта этого класса. Каждому методу m класса C назначается *виртуальный номер* $vnum_{C,m}$, соответствующий индексу ячейки таблицы vmt_C, в которой находится адрес реализации этого метода. При переопределении метода в наследнике, новая реализация получает такой же виртуальный номер, что и оригинальный метод. Фактически это означает, что раскладка VMT класса должна *расширять* раскладку VMT суперкласса.

Таким образом по виртуальному номеру $vnum_{C,m}$ всегда можно получить актуальную реализацию метода m не только в таблице самого класса C, но и в таблице

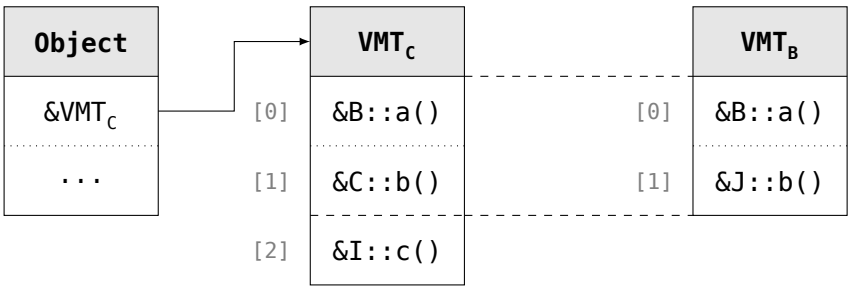


Рис. 1.1: Таблица виртуальных методов класса *C*

любого его наследника. Данное свойство позволяет генерировать виртуальный вызов `x.m()` как обычный косвенный вызов по таблице, полученной напрямую из объекта:

```
call x.vmt[vnum_C,m]
```

1.2. Таблица интерфейсных методов

Ограничения, накладываемые на интерфейсы, позволяют избежать типичных трудностей, сопровождающих поддержку множественного наследования [20] и использовать для реализации интерфейсных вызовов структуру *таблиц интерфейсных методов* (англ. *interface method table*, *IMT*, *itable*). Таблица интерфейсных методов (или *интерфейсная таблица*) для класса *C* и суперинтерфейса *I* представляет собой массив $imt_{C,I}$ адресов реализаций в классе *C* методов интерфейса *I*. По аналогии с таблицей виртуальных методов, каждому методу *m* интерфейса *I* назначается *виртуальный номер* $vnum_{I,m}$, соответствующий индексу ячейки таблицы $imt_{X,I}$, в которой находится адрес реализации этого метода в каждом классе *X*, который реализует данный интерфейс. Это позволяет сгенерировать интерфейсный вызов таким же образом, как виртуальный — с помощью косвенного вызова по интерфейсной таблице. Однако из-за того, что класс может наследовать произвольное количество интерфейсов, адрес IMT для вызова не всегда может быть известен во время компиляции. Поэтому в общем случае интерфейсный вызов требует динамического определения таблицы во время исполнения, например, с помощью линейного поиска IMT среди реализуемых классом интерфейсов.

Таким образом, сгенерированный код для интерфейсного метода `x.m()` состоит из некоторой процедуры поиска и косвенного вызова по полученной таблице, аналогично виртуальному вызову:

```
imt_C,I ← x.imts.find(&I)
call imt_C,I[vnum_I,m]
```

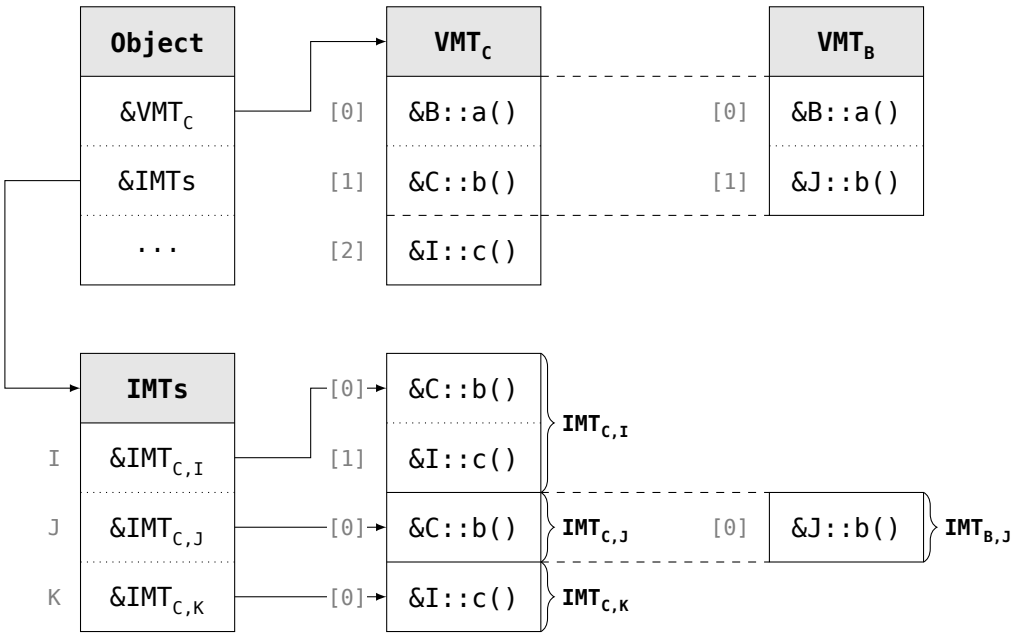


Рис. 1.2: Таблицы виртуальных и интерфейсных методов класса *C*

1.3. Существующие подходы

Многие табличные реализации интерфейсных вызовов жертвуют размером структур ради производительности самого вызова. Так, например, в проектах CACAO [9], SableVM [10] и Jikes RVM [16] удалось полностью избавиться от поиска таблицы с помощью одной глобально индексируемой таблицы интерфейсных методов в каждом классе. Но некоторые подходы делают больший акцент на уменьшении размера служебных данных и, в частности, таблиц интерфейсных методов.

Авторы системы Marmot [17], разработанной в лаборатории Microsoft Research, предложили инновационный способ наследования раскладки таблиц, при котором IMT каждого интерфейса включает в себя IMT всех своих непосредственных суперинтерфейсов как подтаблицы. Такой подход позволяет в каждом классе создавать только таблицы непосредственно реализуемых суперинтерфейсов, которые уже будут содержать все необходимые IMT. Однако, как отмечают авторы, при такой раскладке может возникнуть дублирование информации, которого не было в базовой раскладке. Несмотря на это, на практике, данный подход дает заметное уменьшение суммарного размера таблиц.

Независимо к похожей вложенной раскладке IMT пришли создатели платформы SHAP [18] — аппаратного процессора Java байткода для встраиваемых устройств. В отличие от классической структуры IMT, в SHAP все таблицы вклады-

ваются напрямую в VMT, позволяя переиспользовать ячейки IMT под виртуальные вызовы. Дополнительно каждая таблица (и подтаблица) имеет заголовок в виде ссылки на тип соответствующего интерфейса, который позволяет искать необходимую интерфейсную таблицу напрямую в VMT без отдельной структуры для поиска.

Также, авторам SHAP удалось во многом избавиться от необходимости линейного поиска при вызове за счет продвинутого статического анализа и специального механизма раскраски указателей [19], в котором указатели на объекты помечаются отступом до IMT, соответствующей одному из суперинтерфейсов настоящего класса этого объекта. Этот отступ кодируется напрямую в указателе, и часто может быть статически вычислен компилятором. Таким образом интерфейсный вызов превращается практически в виртуальный, кроме тех случаев, когда компилятор не может статически гарантировать соответствие пометки указателя нужному интерфейсу. В такой ситуации вставляется проверка соответствия цвета, которая может быть реализована эффективно, благодаря наличию ссылки на интерфейс в заголовке таблицы, с последующим поиском таблицы, если проверка провалилась.

Однако, у такого подхода есть свои недостатки: любое кодирование информации в указателях неизбежно приводит к усложнению управления памятью и отладки среды исполнения, а наличие заголовков у таблиц уменьшает эффект сжатия от вложенной раскладки таблиц.

1.4. Источники избыточности

Основной причиной большого размера таблиц является избыточность данных в них. В идеальной табличной структуре для диспетчеризации вызовов находилось бы ровно по одному адресу для каждой уникальной реализации метода. Но уже в таблицах виртуальных методов при одиночном наследовании возникает дублирование данных между таблицами класса и суперкласса, а при добавлении интерфейсных таблиц его становится еще больше. Для примера рассмотрим структуру таблиц, изображенную на рис. 1.2. Несмотря на то, что всего уникальных реализаций методов только четыре: `&B::a()`, `&J::b()`, `&C::b()` и `&I::c()`; каждая из них встречается в таблицах минимум два раза. Отсюда выделяются два вида избыточности таблиц:

- *межклассовая избыточность*, возникающая при наследовании реализаций,

определенных в суперклассе или суперинтерфейсе, и

- *внутриклассовая избыточность*, возникающая при множественном наследовании интерфейсов.

2. Совмещенная раскладка таблиц

Для уменьшения внутриклассовой избыточности был разработан новый алгоритм совмещенной раскладки интерфейсных таблиц внутри VMT, основанный на тех же идеях, что и вложенная раскладка в SHAP и Marmot. На рис. 2.1 проиллюстрированы основные преобразования базовой раскладки, вошедшие в итоговую совмещенную раскладку таблиц:

1. По аналогии с раскладкой в SHAP интерфейсные таблицы становятся частью раскладки VMT и наследуются при расширении раскладки VMT суперкласса (рис. 2.1б).
2. По аналогии с раскладкой в Marmot раскладка IMT включает таблицы суперинтерфейсов как подтаблицы (рис. 2.1в). Но в отличие от Marmot, в раскладку IMT добавляются только непересекающиеся по методам таблицы суперинтерфейсов. Таким образом размер IMT остается таким же как в базовой раскладке.
3. В некоторых случаях, последняя IMT суперкласса может быть расширена с помощью одной из таблиц класса-наследника (рис. 2.1г). Чтобы увеличить шансы такого расширения, подтаблицы VMT сортируются в порядке возрастания размера, а подтаблицы IMT сортируются в порядке убывания размера, позволяя наибольшей IMT расширяться наибольшей IMT класса-наследника.

Размер таблиц при использовании полученной совмещенной раскладки гарантированно не превосходит размера таблиц базовой раскладки, что подтверждается следующей теоремой 2.1.

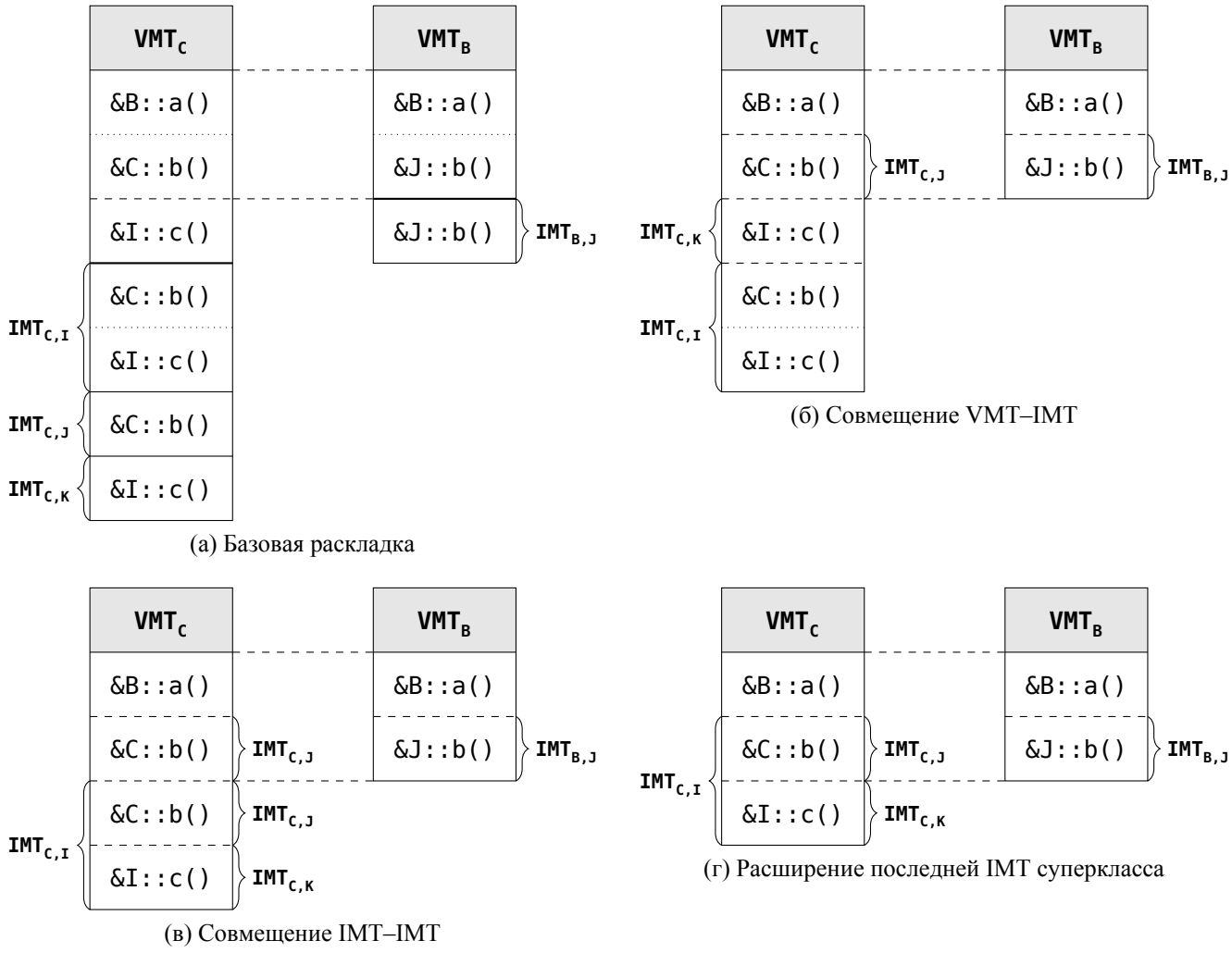
Теорема 2.1. *Для произвольной иерархии H суммарный размер таблиц в совмещенной раскладке не превосходит размера таблиц в базовой раскладке:*

$$\forall H : size_H^{Combined} \leq size_H^{Base}.$$

Доказательство. Каждое из рассмотренных преобразований не увеличивает суммарный размер таблиц. [2] □

Однако, для раскладки Marmot, в которой в IMT включаются все таблицы суперинтерфейсов, подобных гарантий нет. Более того размер таблиц в этой рас-

[2] Подробное доказательство представлено в тексте диссертации.



(а) Базовая раскладка

(б) Совмещение VMT–IMT

(в) Совмещение IMT–IMT

(г) Расширение последней IMT суперкласса

Рис. 2.1: Этапы перехода к совмещенной раскладке таблиц

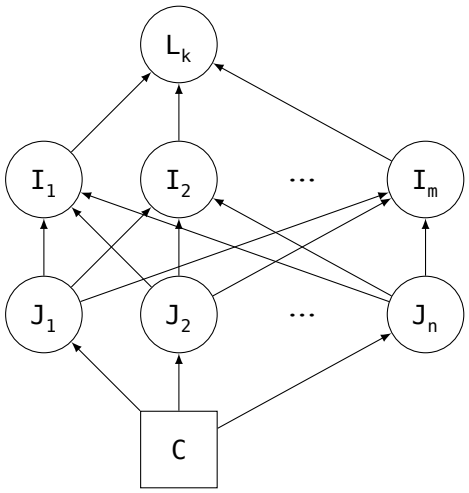


Рис. 2.2: Иерархия $H_{k,m,n}$

кладке может неограниченно превышать размер таблиц базовой раскладки, о чем говорит следующая теорема 2.2.

Теорема 2.2. *Для произвольной константы* $\delta > 1$ *существует иерархия* H*, в которой суммарный размер таблиц раскладки Marmot больше суммарного размера таблиц базовой раскладки в* δ *раз:*

$$\forall \delta > 1 \; \exists H \; : \; size_H^{Marmot} \geq \delta size_H^{Base}. \tag{2.1}$$

Доказательство. Рассмотрим следующую иерархию $H_{k,m,n}$, изображенную на рис. 2.2:

```
interface L_k                    { x_1(); ...; x_k() }
interface I_1 extends L_k        { y_1() }
...
interface I_m extends L_k        { y_m() }
interface J_1 extends I_1, ..., I_m { z_1() }
...
interface J_n extends I_1, ..., I_m { z_n() }
class C   implements J_1, ..., J_n {      }
```

VMT класса C в обеих раскладках будет состоять из всех объявленных в иерархии методов, поэтому ее размер будет равен:

$$|vmt_C| = |x_1 ... x_k y_1 ... y_m z_1 ... z_n| = k + m + n.$$

Размеры интерфейсных таблиц и суммарный размер таблиц вычисляются следующим образом:

Base

$$
\begin{aligned}
|imt_{C,L}| &= |x_1 ... x_k| = k \\
|imt_{C,I_i}| &= |x_1 ... x_k y_i| = k + 1 \\
|imt_{C,J_j}| &= |x_1 ... x_k y_1 ... y_m z_j| \\
&= k + m + 1
\end{aligned}
$$

$$
\begin{aligned}
size_H^{Base} &= |vmt_C| + |imt_{C,L}| \\
&+ \sum_{i=1,...,m} |imt_{C,I_i}| \\
&+ \sum_{j=1,...,n} |imt_{C,J_j}| \\
&= (k + m + n) + k + m(k + 1) \\
&+ n(k + m + 1)
\end{aligned}
$$

Marmot

$$
\begin{aligned}
|imt_{C,L}| &= |x_1 ... x_k| = k \\
|imt_{C,I_i}| &= |imt_{C,L} y_i| = k + 1 \\
|imt_{C,J_j}| &= |imt_{C,I_1} ... imt_{C,I_m} z_j| \\
&= m(k + 1) + 1
\end{aligned}
$$

$$
\begin{aligned}
size_H^{Marmot} &= |vmt_C| \\
&+ \sum_{j=1,...,n} |imt_{C,J_j}| \\
&= (k + m + n) \\
&+ n(m(k + 1) + 1)
\end{aligned}
$$

Тогда, для выполнения неравенства (2.1) достаточно выбрать целые значения k, m и n удовлетворяющие следующей системе неравенств: [3]

$$
\begin{cases}
m > \delta \\
k > \frac{(m+2)(\delta-1)}{m-\delta} \\
n \geq \frac{(k+m)(2\delta-1)+\delta km}{k(m-\delta)-(m+2)(\delta-1)}
\end{cases}
. \qquad (2.2)
$$

Например, при $\delta = 2$ неравенство (2.1) выполняется для иерархии $H_{6,3,63}$.

□

[3] Подробное доказательство представлено в тексте диссертации.

3. Разностное кодирование таблиц

Для уменьшения межклассовой избыточности был разработан алгоритм разностного кодирования таблиц, при котором в исполняемый файл записывается компактное представление разницы таблиц класса и его суперкласса, а затем восстанавливается во время исполнения в полноценную структуру таблиц виртуальных и интерфейсных методов. Закодированная разность таблиц представляет собой последовательность инструкций $Put_1(mid)$, $Put_2(cid, mid)$ и $Skip(k)$, описанных в таблице 3.1.

Таблица 3.1: Инструкции разностного кодирования

Инструкция	Код	Интерпретация
$Put_1(mid)$	0x0	Положить значение текущей ячейки VMT равным адресу метода *mid* в данном классе и перейти к следующей ячейке.
$Put_2(tid, mid)$	0x1	Положить значение текущей ячейки VMT равным адресу метода *mid* в супертипе *tid* и перейти к следующей ячейке.
$Skip(k)$	0x2	Пропустить следующие *k* ячеек.

Каждая инструкция представляется в виде одного или двух (в зависимости от количества аргументов) целых 32-битных чисел. При этом *Код* инструкции закодирован вместе с первым аргументом: `(Код | Арг`$_0$ `<< 1) [, Арг`$_0$`]`. Полученная последовательность целых чисел дополнительно кодируется в формате Unsigned LEB128 [21]. В предположении, что IMT включаются как подтаблицы в VMT, порождение инструкций кодирования и восстановление VMT класса *C* описаны далее в алгоритмах 3.1 и 3.2.

Алгоритм 3.1. *Кодирование* vmt_C.

1. *Инициализировать* $cur \leftarrow 0$, $k \leftarrow 0$;
2. *Пока* $cur < |vmt_C|$:
   - 2.1 *Если у класса C нет суперкласса или реализации по индексу cur отличаются между* vmt_C *и VMT суперкласса, то:*
     - 2.1.1 *Если* $k > 0$, *то породить инструкцию* $Skip(k)$ *и задать* $k \leftarrow 0$;
     - 2.1.1 *Если реализация объявлена в классе C, то породить инструкцию* $Put_1(mid)$, *где mid — идентификатор реализации;*

2.1.1 Иначе породить инструкцию $Put_2(tid, mid)$, где tid и mid — идентификаторы супертипа, в котором объявлена реализация, и реализации в этом типе;

2.1.1 Увеличить $cur \leftarrow cur + 1$.

2.2 Иначе увеличить $k \leftarrow k + 1$.

□

Алгоритм 3.2. *Восстановление vmt_C.*

1. *Инициализировать $cur \leftarrow 0$;*
2. *Если у класса C есть суперкласс, то скопировать его VMT в начало vmt_C;*
3. *Пока остались необработанные инструкции:*

3.1 Если следующая инструкция — $Skip(k)$, то увеличить $cur \leftarrow cur + k$;

3.2 Если следующая инструкция — $Put_1(mid)$, то:

3.2.2 Инициализировать ячейку по индексу cur адресом метода mid в классе C;

3.2.2 Увеличить $cur \leftarrow cur + 1$.

3.3 Если следующая инструкция — $Put_2(tid, mid)$, то:

3.3.3 Инициализировать ячейку по индексу cur адресом метода mid в супертипе tid;

3.3.3 Увеличить $cur \leftarrow cur + 1$.

□

Для полученного кодирования справедлива следующая теорема.

Теорема 3.1. *Если у класса не более 2^5 предков и в каждом из них, включая сам класс, не более 2^{21} методов, то при разностном кодировании VMT этого класса на 32-битной архитектуре размер закодированного представления не превосходит размера исходной таблицы.*

Доказательство. Достаточно показать, что для каждой ячейки (или последовательности ячеек), соответствующая инструкция кодируется не более чем четыремя байтами в кодировке Unsigned LEB128. В этой кодировке каждый байт закодированного представления содержит 7 значащих бит. Заметим, что размер инструкции $Skip(k)$ всегда строго меньше кодируемой последовательности ячеек (в худшем случае, когда $k = 1$, инструкция кодируется одним байтом), а также, что *Код* инструкции занимает два бита.

Условия теоремы гарантируют, что $tid < 2^5$ и $mid < 2^{21}$, а значит кодируются 5-ю и 21-м битом соответственно. Тогда вместе с *Кодом* инструкция $Put_1(mid)$ представляется 23-битным числом, которое кодируется не более чем четыремя байтами в Unsigned LEB128. Инструкция $Put_2(tid, mid)$ имеет два аргумента, которые вместе с *Кодом* занимают 7 и 21 бит соответственно. В кодировке Unsigned LEB128 первый аргумент всегда занимает один байт, а второй аргумент — не более трех байт. □

4. Экспериментальные результаты

Оба алгоритма совмещенной раскладки и разностного кодирования таблиц были реализованы в экспериментальной сборке Huawei JDK с поддержкой статической компиляции для 32-битной платформы ARMv7. Набор тестируемых приложений состоит из стандартных библиотек и сред исполнения четырех языков программирования, перечисленных в таблице 4.1. Каждая библиотека была скомпилирована статическим компилятором Huawei JDK. Помимо итогового алгоритма совмещенной раскладки, были протестированы также базовая раскладка и существующие подходы Marmot и SHAP:

- *Base* — базовая раскладка (рис. 2.1а);
- *Combined* — совмещенная раскладка (рис. 2.1г);
- *Marmot* — оригинальная раскладка IMT из Marmot;
- *SHAP* — оригинальная раскладка IMT из SHAP.

Таблица 4.1: Тестируемые языки и их стандартные библиотеки

Язык	Количество классов и интерфейсов	Размер исполняемого файла стандартной библиотеки, KB
Kotlin 1.4.32	2 518	2 171
Groovy 3.0.8	4 891	4 845
Scala 2.13.0	2 587	5 174
Java 1.8.0_282 (cp1)	6 500	10 285

В таблице 4.2 представлены коэффициенты уменьшения избыточности в таблицах, по сравнению с базовым алгоритмом раскладки. Использование совмещенной раскладки таблиц позволяет удалить более 60 % избыточности на всех тестируемых приложениях, показывая лучшие результаты, чем существующие алгоритмы. Подходу Marmot удается удалить до 21 % избыточности только за счет совмещения IMT, а подходу SHAP — еще улучшить этот результат на трех из четырех приложений за счет совмещения ячеек VMT и IMT.

Таблица 4.2: Коэффициент уменьшения (< 1) или увеличения (> 1) избыточности таблиц стандартной библиотеки

Язык	Base	Combined	Marmot	SHAP
Kotlin 1.4.32	1.00	**0.25**	0.79	0.79
Groovy 3.0.8	1.00	**0.04**	0.81	0.23
Scala 2.13.0	1.00	**0.40**	0.86	0.73
Java 1.8.0_282 (cp1)	1.00	**0.17**	0.82	0.51

В таблице 4.3 отражены значения размера исполняемого файла приложения, вместе с указанным в скобках коэффициентом уменьшения относительно схемы базовой раскладки таблиц.

На всех приложениях снижение избыточности VMT и IMT влечет уменьшение размера исполняемого файла. Однако, интересно отметить, что на приложениях с наибольшим снижением избыточности таблиц, написанных на Kotlin, Groovy и Java, получилось наименьшее уменьшение размера исполняемого файла, в то время как размер приложения, написанного на Scala, уменьшился значительно, несмотря на меньшее относительное снижение избыточности. Это объясняется тем, что стандартная библиотека языка Scala имеет богатую иерархию коллекций, основанную на довольно крупных интерфейсах, из-за чего таблицы виртуальных и интерфейсных методов занимают почти 40 % всего исполняемого файла.

Таблица 4.3: Размер исполняемого файла стандартной библиотеки (без кодирования), KB

Язык	Base	Combined	Marmot	SHAP
Kotlin 1.4.32	2 171 (1.00)	2 128 (0.98)	2 159 (0.99)	2 159 (0.99)
Groovy 3.0.8	4 845 (1.00)	4 726 (0.98)	4 821 (1.00)	4 750 (0.98)
Scala 2.13.0	5 174 (1.00)	**4 263 (0.82)**	4 961 (0.96)	4 769 (0.92)
Java 1.8.0_282 (cp1)	10 285 (1.00)	10 211 (0.99)	10 269 (1.00)	10 242 (1.00)

Результаты применения разностного кодирования к каждой из рассматриваемых раскладок представлены в таблице 4.4. Видно, что кодирование таблиц существенно уменьшает размер метаданных, даже с самым базовым алгоритмом

раскладки. Более компактные раскладки также положительно влияют на сжатие закодированного представления.

Таблица 4.4: Размер исполняемого файла стандартной библиотеки (с кодированием), KB

Язык	Base	Combined	Marmot	SHAP
Kotlin 1.4.32	2 048 (0.94)	2 043 (0.94)	2 046 (0.94)	2 045 (0.94)
Groovy 3.0.8	4 295 (0.89)	4 286 (0.88)	4 294 (0.89)	4 287 (0.88)
Scala 2.13.0	**3 497 (0.68)**	**3 424 (0.66)**	**3 490 (0.67)**	**3 471 (0.67)**
Java 1.8.0_282 (cp1)	9 956 (0.97)	9 940 (0.97)	9 951 (0.97)	9 943 (0.97)

Заключение

В настоящей работе были получены следующие результаты:

1. Разработан алгоритм совмещенной раскладки таблиц с гарантированной верхней оценкой размера таблиц, позволяющий добиться значительного уменьшения дублирования данных в таблицах. Получено существенное уменьшение размера исполняемого файла на диске.
2. Разработан алгоритм разностного кодирования, позволяющий значительно уменьшить размер исполняемого файла, и при этом, эффективно восстановить исходную структуру таблиц виртуальных и интерфейсных методов во время исполнения.
3. Продемонстрирована применимость разностного кодирования при произвольном алгоритме раскладки таблиц.
4. Доказано, что размер закодированного представления таблиц с помощью разностного кодирования не превосходит исходного размера кодируемых данных при реалистичных ограничениях на количество типов и методов в программе.
5. Оба разработанных алгоритма были реализованы в экспериментальной сборке Huawei JDK и апробированы на представительном наборе приложений.
6. Доказана неограниченность размера таблиц при использовании существующей раскладки таблиц в Marmot, по сравнению с базовым алгоритмом раскладки таблиц.

Для разработанных алгоритмов доказана только верхняя оценка размера таблиц и закодированного представления, однако, она не дает понимания эффективности сжатия на практике. Известно, что подход Marmot может давать более компактную раскладку, несмотря на отсутствие верхней оценки размера таблиц, что задает направления для дальнейших исследований в области раскладки интерфейсных таблиц. Также, имея всего три инструкци разностного кодирования, предложенный подход демонстрирует весьма положительные результаты, но оставляет свободу для расширения набора инструкций и кодируемых метаданных.

Публикации автора по теме диссертации

Статьи:

1. Trepakov I., Pavlov P. Compact Interface Method Table Layout // 2020 Ivannikov Memorial Workshop (IVMEM). IEEE, 2020. Сс. 62–68.

Свидетельства о государственной регистрации программы для ЭВМ:

2. Свидетельство о государственной регистрации программы для ЭВМ №2017613872. Excelsior JET Embedded 11.3 for Linux/ARM / А.Л. Гигуз и др. – Заявка №2016663925. Дата регистрации 19 декабря 2016 г. Дата публикации 3 апреля 2017 г.

Патентные заявки:

3. Заявка PCT/RU2020/000406. Application Service Metadata Encoding and Initialization / И.С. Трепаков, П.Е. Павлов; заявитель Huawei Technologies Co., Ltd.; заявл. 03.08.2020.

Тезисы докладов:

4. Трепаков И.С. Эффективная реализация таблиц виртуальных методов в языках с поддержкой ограниченного множественного наследования // Материалы 56-й Международной научной студенческой конференции МНСК-2018: Математика. 2018. С. 183.

5. Трепаков И.С. Эффективная генерация служебных метаданных для таблиц виртуальных методов в статических компиляторах // Материалы 58-й Международной научной студенческой конференции МНСК-2020: Математика. 2020. С. 133.

Список использованных источников

1. Сигаев А. Операционные системы для встраиваемых применений // Компоненты и технологии. Общество с ограниченной ответственностью «Издательство Файнстрит», 2000. № 5.
2. Courington B., Collins G. Getting Started with Java SE Embedded on the Raspberry Pi. 2012.
3. Donat W., Krause C. Learn Raspberry Pi Programming with Python. Springer, 2014.
4. Caracas A. и др. Mote runner: A multi-language virtual machine for small embedded devices // 2009 Third International Conference on Sensor Technologies and Applications. IEEE, 2009. С. 117–125.
5. Gosling J. The Java language specification. Addison-Wesley Professional, 2000.
6. Hejlsberg A., Wiltamuth S., Golde P. C# language specification. Addison-Wesley Longman Publishing Co., Inc., 2003.
7. Официальный сайт проекта CACAO JVM [Электронный ресурс]. URL: http://www.cacaojvm.org/ (дата обращения: 28.03.2021).
8. Официальный сайт проекта SableVM [Электронный ресурс]. URL: http://www.sablevm.org/ (дата обращения: 28.03.2021).
9. Krall A., Grafl R. CACAO - A 64-bit JavaVM Just-in-Time Compiler // Concurrency Practice and Experience. Chichester, Sussex: J. Wiley, c1989-c2000., 1997. Т. 9, № 11. С. 1017–1030.
10. Gagnon E.M., Hendren L.J. Sable VM: A research framework for the efficient execution of Java bytecode // Java Virtual Machine Research and Technology Symposium. 2001. С. 27–40.
11. Cox B.J. Object-oriented programming: an evolutionary approach. 1986.
12. Dixon R. и др. A fast method dispatcher for compiled languages with multiple inheritance // Conference proceedings on Object-oriented programming systems, languages and applications. 1989. С. 211–214.
13. André P., Royer J.-C. Optimizing method search with lookup caches and incremental coloring // ACM Sigplan Notices. ACM, 1992. Т. 27. С. 110–126.
14. Vitek J., Horspool R.N. Compact dispatch tables for dynamically typed object oriented languages // International Conference on Compiler Construction. Springer, 1996. С. 309–325.

15. Официальный сайт проекта Jikes RVM [Электронный ресурс]. URL: https://www.jikesrvm.org/ (дата обращения: 28.03.2021).
16. Deutsch L.P., Schiffman A.M. Efficient implementation of the Smalltalk-80 system // Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. 1984. С. 297–302.
17. Fitzgerald R. и др. Marmot: An optimizing compiler for Java // Software-Practice and Experience. Citeseer, 2000. Т. 30, № 3. С. 199–232.
18. Preußer T. Increasing the Performance and Predictability of the Code Execution on an Embedded Java Platform. 2011.
19. Preußer T.B., Zabel M., Spallek R.G. Enabling constant-time interface method dispatch in embedded Java processors // Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems. ACM, 2007. С. 196–205.
20. Stroustrup B. Multiple inheritance for C++ // Computing Systems. 1989. Т. 2, № 4. С. 367–395.
21. LEB128 - Wikipedia [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/LEB128#Unsigned_LEB128 (дата обращения: 01.05.2021).