САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

На правах рукописи

Пилипенко Артур Витальевич

Разработка и реализация механизмов сокращения размера Java-приложений для встраиваемых систем в закрытой модели

Специальность 05.13.11 —
«Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей»

Диссертация на соискание учёной степени
кандидата технических наук

Научный руководитель:
кандидат физико-математических наук, доцент
Кияев Владимир Ильич

Санкт-Петербург — 2018

Оглавление

Стр.

Введение 4

Глава 1. Обзор существующих механизмов сокращения размера 13
1.1 Анализ достижимости методов 13
1.2 Удаление неиспользуемых полей 22
1.3 Специализация иерархии классов 23
1.4 Ромизация 24
1.5 Алгоритм Reachable Member Analysis 26
1.6 Понижение избыточности Java-программ 28
1.7 Анализ рефлексии 32
1.8 Специализация набора инструкций 34
1.9 Оптимизация кодирования инструкций 41
1.10 Сокращение размера класс-файлов 42
1.11 Выводы 42

Глава 2. Понижение избыточности при выборочной инициализации классов 45
2.1 Инициализация классов 45
2.2 Финализация объектов 48
2.3 Достижимость объектов в куче 50
2.4 Описание дополнительных зависимостей 52
2.5 Анализ достижимости методов 55
2.6 Анализ удалимости полей 58
2.7 Анализ достижимости неудалимых объектов 60
2.8 Межпроцедурный анализ указателей 62
2.9 Анализ удалимости классов 68
2.10 Автоматический анализ нативных зависимостей 69
2.11 Выводы 71

Глава 3. Специализация набора инструкций 73
3.1 Специализация набора инструкций 73

Стр.

3.2 Шаблоны последовательностей инструкций . . . 74
3.3 Упрощение исходного набора инструкций . . . 75
3.4 Построение словаря последовательностей . . . 77
3.5 Инструкции шаблона . . . 79
3.6 Перебор шаблонов . . . 80
3.7 Выбор набора инструкций . . . 82
3.8 Выводы . . . 84

Глава 4. Программная реализация и экспериментальное исследование 87
4.1 Реализация предложенных алгоритмов . . . 87
4.2 Сравнение алгоритмов понижения избыточности . . . 89
4.3 Экспериментальная оценка алгоритмов понижения избыточности . 92
4.4 Сравнение алгоритмов сжатия . . . 103
4.5 Экспериментальная оценка алгоритма сжатия . . . 105
4.6 Ограничения . . . 109
4.7 Направления дальнейшего исследования . . . 110
4.8 Выводы . . . 111

Заключение . . . 118

Список литературы . . . 121

Список рисунков . . . 133

Список таблиц . . . 134

Приложение А. Формальное описание алгоритмов . . . 135
А.1 Reachable Member Analysis . . . 135
А.2 Алгоритмы понижения избыточности . . . 136
А.3 Алгоритм сжатия Java байт-кода . . . 142

Приложение Б. Акты о внедрении результатов диссертационного исследования . . . 148

Введение

Актуальность темы.

Качество программного обеспечения является критически важным аспектом промышленной разработки [1]. Использование высокоуровневых языков программирования, таких как Java, C#, позволяет повысить эффективность работы программиста, сократить время и стоимость разработки и в конечном итоге повысить качество программного обеспечения. Автоматическое управление памятью позволяет избавиться от целого класса ошибок, связанных с управлением памятью. Строгая типизация, автоматические проверки на этапе исполнения на уровне языка обеспечивают безопасность исполняемого кода. Более высокоуровневые абстракции, богатые стандартные библиотеки позволяют сократить количество ошибок за счет переиспользования кода. Удобный инструментарий, интегрированные среды разработки повышают эффективность работы программиста, упрощают рефакторинг кода.

В то же время основными языками программирования для встраиваемых устройств были и остаются низкоуровневые языки, например C или C++. Вычислительная мощность, объем оперативной и энергонезависимой памяти встраиваемых устройств на несколько порядков меньше, чем у персональных компьютеров и серверов. Низкоуровневые языки программирования позволяют эффективно использовать ограниченные возможности целевых устройств. Недостатком таких языков является высокая сложность разработки.

Задачи, решаемые при разработке приложений для встраиваемых устройств, имеют много общего, при этом применяемые технологии часто не позволяют переиспользовать существующие решения. Все, кто сталкивается с разработкой приложений для встраиваемых устройств, вынуждены реализовывать одну и ту же функциональность, прежде чем начать реализовывать бизнес-логику приложения. Примером такой функциональности является взаимодействие с периферией с помощью стандартных интерфейсов, сетевое взаимодействие, шифрование и тому подобное [2; 3]. Какое-то время назад эти проблемы были актуальны и при разработке для персональных компьютеров и серверов. Решением данных проблем стало применение более высокоуровневых и безопасных языков программирования и платформ, таких как Java и .NET.

Применение языка Java для разработки программного обеспечения для встраиваемых устройств позволяет существенно сократить время и стоимость разработки и повысить качество целевого продукта.

Возможность применения языка Java для встраиваемых устройств рассматривалась с самого начала его разработки. Однако этот язык предъявляет более высокие требования к ресурсам, по сравнению с языками C или C++. Это плата, которую приходится платить за высокоуровневые абстракции, безопасность и богатую стандартную библиотеку. Реализация Java-платформы для ограниченных в ресурсах устройств должна быть оптимизирована по размеру, а не по скорости исполнения.

Начиная с 2000 года опубликовано большое количество академических работ на тему использования языка Java для встраиваемых применений [4—12]. В настоящее время язык начинает активно использоваться на практике при разработке встраиваемых систем, что заставляет возвращаться к вопросам оптимизации.

Стандартная модель распространения Java-приложений подразумевает, что на устройство может быть загружено и исполнено произвольное приложение. При этом реализация Java-платформы должна поддерживать всё, что описано в её спецификации. Такая модель называется открытой. Для встраиваемых устройств набор исполняемых приложений обычно определен заранее, и нет необходимости исполнять произвольные приложения. Модель, в которой весь исполняемый код известен заранее, называется закрытой. В закрытой модели реализация Java-платформы может быть специализирована для заданного приложения. Так, например, в закрытой модели можно проанализировать, какие возможности платформы используются приложением, и удалить неиспользуемые. Такую специализацию будем называть понижением избыточности. Кроме того, в закрытой модели для заданного приложения можно разработать специализированный набор инструкций, который уменьшит суммарный размер исполняемого кода и интерпретатора, необходимого для его выполнения.

Степень разработанности темы.

Задача понижения избыточности программ хорошо проработана. Изначально алгоритмы анализа достижимости методов, удалимости полей и классов были описаны для языка C++ в середине 90-х годов [13—17]. Впоследствии идеи этих работ получили развитие для программ на языке Java. Алгоритмы анализа достижимости методов, удалимости полей и классов часто используются для выделения

минимального подмножества Java-платформы, необходимого для выполнения заданного приложения в закрытой модели [18—25].

Реализации виртуальных машин для встраиваемых применений часто используют раздельную инициализацию, при которой часть работы по инициализации виртуальной машины переносится с целевого устройства на хостовое устройство [26—30]. В этом случае стандартные алгоритмы понижения избыточности не всегда применимы. В инициализированном состоянии виртуальной машины существуют зависимости между методами, полями и объектами, созданными при инициализации, которые не разрешаются классическими алгоритмами. Несмотря на большое количество публикаций по теме понижения избыточности, большинство алгоритмов анализируют только код программы и не имеют дела с её состоянием. На момент написания работы соискателю известно только о двух статьях, обсуждающих применимость существующих алгоритмов анализа достижимости методов при раздельной инициализации [28; 29]. Таким образом, исследование возможности применения существующих алгоритмов понижения избыточности при раздельной инициализации является актуальной задачей.

Другим направлением для сокращения размера является использование более компактного представления программы [31—35]. Сокращение размеров бинарного кода приложения актуально во многих областях применения. Сюда входят разработка программного обеспечения для ограниченных в ресурсах устройств; ускорение загрузки и уменьшение размера передаваемых по сети файлов. В зависимости от области применения к механизмам сжатия предъявляются различные требования. Так, например, встраиваемые устройства с ограниченными ресурсами зачастую не имеют возможности предварительно распаковать исполняемую программу. Для таких применений приходится использовать компактные исполняемые представления.

Специализация набора инструкций – один из способов получить компактное исполняемое представление [27; 36—40]. Применение этого подхода для Java ограничено тем, что в открытой модели итоговый набор инструкций должен быть совместим со стандартным. В закрытой модели такие ограничения отсутствуют. Таким образом, применение специализации набора инструкций для сжатия Java байт-кода в закрытой модели является перспективным направлением.

Целью данной работы является сокращение аппаратных требований Java-платформы для встраиваемых систем при исполнении заданного набора приложений в закрытой модели. Цель диссертационной работы достигается

решением следующей научной **задачи**: разработать и реализовать методы, алгоритмы и программные средства, позволяющие сократить размер Java-приложений для встраиваемых систем в закрытой модели, посредством специализации Java-платформы для заданных приложений. Данная задача разбивается на следующие частные задачи исследования:

1. Изучить существующие алгоритмы понижения избыточности программ с точки зрения их применимости при использовании раздельной инициализации.
2. Исследовать существующие алгоритмы сжатия бинарного кода. Изучить их применимость для сжатия Java байт-кода в закрытой модели.
3. Разработать алгоритмы понижения избыточности Java-программ, применимые при раздельной инициализации.
4. Разработать алгоритм сжатия Java байт-кода в закрытой модели, применимый для встраиваемых систем с ограниченными ресурсами.
5. Реализовать предложенные алгоритмы и экспериментальным путем измерить их эффективность.

Научная новизна:

1. Разработан алгоритм анализа Java-программ, определяющий достижимость методов Java-классов и осуществляющий выборочную инициализацию используемых классов. Алгоритм не инициализирует классы, которые не могут быть инициализированы в процессе работы приложения. Такой подход, в отличие от существующих, не нарушает поведения приложения и позволяет сократить размер инициализированного образа.
2. Сформулирован критерий удалимости ссылочных полей Java-классов, позволяющий удалять инициализируемые, но неиспользуемые поля, не нарушая семантику финализации объектов, в отличие от описанных ранее алгоритмов, которые либо не позволяли удалять инициализируемые, но неиспользуемые поля, либо не учитывали, что удаление ссылочных полей может нарушить семантику финализации. Такой подход позволяет сократить количество полей в образе и размер образа, не нарушая поведения программы.
3. Предложен оригинальный метод анализа программ, выявляющий межъязыковые зависимости между кодом на языках Java и C++ для алгоритмов понижения избыточности. Применение данного метода позволяет не описывать такие зависимости вручную.

4. Разработан алгоритм сжатия Java байт-кода с помощью специализации набора инструкций, осуществляющий свертку и укорачивание аргументов совместно со сворачиванием последовательностей инструкций, в отличие от ранее описанных алгоритмов сжатия Java байт-кода, которые не используют данные оптимизации совместно. Совместное применение данных методов позволяет добиться более высокой степени сжатия.
5. Впервые применено упрощение исходного набора инструкций Java байт-кода для словарного сжатия. Получены экспериментальные результаты, показывающие эффективность такого преобразования для Java байт-кода.

Теоретическая и практическая значимость. Теоретическая значимость работы заключается в расширении области применимости существующих алгоритмов понижения избыточности для анализа преинициализированного состояния программы, уточнении критериев удалимости неиспользуемых поля для языка Java, а также в новом применении существующих подходов для компактного кодирования Java-программ.

Применение предложенных алгоритмов на практике позволяет сократить размер приложений, не нарушая их поведения. Сокращение размера приложений позволяет и снизить аппаратные требования к целевым устройствам. Это, в свою очередь, расширяет область применения языка Java. Применения языка Java при разработке программного обеспечения для встраиваемых устройств позволяет повысить эффективность процесса разработки и надежность программного обеспечения.

Задачи диссертации могут решаться путем построения математических моделей и применения формальных математических методов. Однако практическая направленность данного исследования определяет фокус на практическую **методологию и методы исследования**. Методология исследования основана на анализе существующей литературы, формулировании целей и задач, разработке алгоритмических решений и их программной реализации, экспериментальной оценке с помощью численного эксперимента, апробации и анализе результатов. В качестве методов исследования используются методы теории компиляторов, теории множеств и теории графов, теории алгоритмов и математической логики, а также методы разработки программного обеспечения.

Основные положения, выносимые на защиту:

1. Алгоритм анализа Java-программ, определяющий достижимость методов Java-классов и осуществляющий выборочную инициализацию используемых классов.
2. Алгоритм анализа Java-программ, определяющий удалимость полей Java-классов и применимый при раздельной инициализации.
3. Алгоритм анализа Java-программ, определяющий удалимость Java-классов и применимый при раздельной инициализации.
4. Метод анализа программ, выявляющий межъязыковые зависимости между кодом на языках Java и C++ для алгоритмов понижения избыточности.
5. Алгоритм эквивалентного преобразования Java-программы и интерпретатора, необходимого для ее выполнения, сокращающий их суммарный размер за счет специализации набора инструкций интерпретатора.

Положения, выносимые на защиту, соответствуют паспорту специальности 05.13.11 по пункту 1. «Модели, методы и алгоритмы проектирования и анализа программ и программных систем, их эквивалентных преобразований, верификации и тестирования».

Степень достоверности и апробация результатов. Достоверность полученных результатов обеспечивается проведенными логическими построениями, доказательствами и экспериментами. Результаты не противоречат ранее полученным данным, описанным другими авторами.

Основные результаты работы докладывались на:

1. Всероссийской научной конференции по проблемам информатики СПИСОК-2013 [41].
2. Конференции «Технологии Microsoft в теории и практике программирования 2014 (Новые подходы к разработке ПО на примере технологий Microsoft и EMC)» [42].
3. Конференции JavaOne San-Francisco 2014 [43].
4. Международной научно-практической конференции «Интеллектуальные и информационные технологии в формировании цифрового общества» 2017 [44].

Предложенные алгоритмы и методы были реализованы в инструменте для автоматической специализации Java-платформы Oracle Java ME Embedded для заданного приложения в закрытой модели. Реализованный инструмент был внедрен

на практике, что подтверждается актами о внедрении. Копии актов о внедрении приводятся в приложении Б.

Личный вклад. Все представленные в диссертации результаты получены автором лично. Соискателем разработаны и реализованы алгоритмы понижения избыточности, специализации набора инструкций, метод автоматического анализа межъязыковых зависимостей. Проведены эксперименты по оценке эффективности разработанных методов и алгоритмов.

Публикации. Основные результаты по теме диссертации изложены в 7 работах [41; 42; 44—48], 3 из которых [45; 47; 48] изданы в журналах из перечня российских рецензируемых научных журналов, в которых должны быть опубликованы основные научные результаты диссертаций на соискание учёных степеней доктора и кандидата наук. Работы [42; 44; 46—48] написаны в соавторстве. В статьях [46—48] соискателю принадлежат идея, описание и реализация алгоритмов специализации набора инструкций, анализа достижимости методов Java-программ при выборочной инициализации классов, анализа удалимости полей и классов, проведение и интерпретация экспериментов по оценке эффективности предложенных алгоритмов. В статьях [42; 44] соискателю принадлежит идея и реализация механизма, использующего межпроцедурный анализ потока данных для анализа достижимости методов и удалимости полей. Остальные результаты в данных статьях принадлежат соавторам.

Объем и структура работы. Диссертация состоит из введения, четырех глав, заключения и двух приложений. Полный объём диссертации составляет 150 страниц, включая 20 рисунков и 16 таблиц. Список литературы содержит 111 источников.

В **первой** главе приведен обзор существующих механизмов сокращения размера приложений. Рассматривается задача понижения избыточности путем удаления неиспользуемых методов, полей и классов. Приводится обзор стандартных алгоритмов анализа достижимости методов, полей и классов. Описывается механизм раздельной инициализации (ромизации). Отмечаются ограничения существующих алгоритмов понижения избыточности, которые не позволяют использовать их при ромизации.

Другой рассмотренный механизм сокращения размера приложения состоит в использовании интерпретируемого представления вместо машинного кода. Для заданного приложения в закрытой модели можно добиться более эффективного кодирования путем специализации набора инструкций интерпретируемого

представления. В главе приведен обзор существующих механизмов для специализации набора инструкций. В обзоре отмечаются недостатки существующих алгоритмов специализации Java байт-кода.

На основании выделенных недостатков существующих решений формулируются задачи для дальнейшего исследования.

Во **второй** главе предлагаются алгоритмы понижения избыточности Java-программ, применимые при раздельной инициализации. Классический подход к удалению недостижимого кода состоит в построении транзитивного замыкания методов, достижимых из точек входа программы, и удаления методов, не попавших в замыкание. При таком подходе возникает вопрос: в какой момент инициализировать классы? До построения замыкания нет информации о том, какие классы используются достижимым кодом. Инициализация классов и удаление статических инициализаторов после построения замыкания может сделать некоторые методы недостижимыми.

Для решения этой проблемы соискателем предложен алгоритм анализа достижимости методов, который расширяет алгоритм RTA выборочной инициализацией используемых классов. В отличие от классических алгоритмов, разработанный алгоритм учитывает объекты, созданные при инициализации классов. Соискателем также предложены алгоритмы анализа удалимости полей и классов, которые учитывают объекты, созданные при инициализации классов.

Результаты, изложенные в данной главе, были опубликованы соискателем в статьях [42; 44; 47; 48]

В **третьей** главе предлагается алгоритм сжатия Java байт-кода, который порождает компактное исполняемое представление путем специализации набора инструкций для заданного приложения. Специализированный набор инструкций сокращает суммарный размер программы и интерпретатора, необходимого для ее исполнения, за счет кодирования часто встречающихся шаблонов последовательностей инструкций новыми инструкциями. В качестве шаблонов используются последовательности инструкций, параметризованные значениями некоторых из аргументов. По существу, такая оптимизация является применением словарного сжатия к коду программы.

Предложенный алгоритм осуществляет свертку и укорачивание аргументов в процессе выбора словаря шаблонов, что позволяет добиться более эффективного кодирования за счет применения этих оптимизаций в контексте других

инструкций. Кроме того, предложенный алгоритм предварительно упрощает исходный набор инструкций, удаляя из него инструкции, которые могут быть представлены с помощью других инструкций. Такое преобразование освобождает дополнительные опкоды для специализированных инструкций.

Результаты, изложенные в данной главе, были опубликованы соискателем в статье [46].

Четвертая глава посвящена описанию программной реализации предложенных алгоритмов и экспериментальному исследованию данной реализации. Предложенные алгоритмы сравниваются с существующими решениями. Отмечаются недостатки и ограничения предложенных решений, выделяются направления для дальнейшего исследования.

Приложение А содержит формальное описание предложенных алгоритмов.

В **приложении Б** приводятся копии актов о внедрении результатов диссертационного исследования.

Глава 1. Обзор существующих механизмов сокращения размера

В главе приведен обзор существующих механизмов сокращения размера приложений. Рассматривается задача понижения избыточности путем удаления неиспользуемых методов, полей и классов. Приводится обзор стандартных алгоритмов анализа достижимости методов, полей и классов. Описывается механизм раздельной инициализации (ромизации). Отмечаются ограничения существующих алгоритмов понижения избыточности, которые не позволяют использовать их при ромизации.

Другой рассмотренный механизм сокращения размера приложения состоит в использовании интерпретируемого представления вместо машинного кода. Для заданного приложения в закрытой модели можно добиться более эффективного кодирования путем специализации набора инструкций интерпретируемого представления. В главе приведен обзор существующих механизмов для специализации набора инструкций. В обзоре отмечаются недостатки существующих алгоритмов специализации Java байт-кода.

На основании выделенных недостатков существующих решений формулируются задачи для дальнейшего исследования.

1.1 Анализ достижимости методов

Если метод f может быть вызван в результате выполнения метода g, метод f считается достижимым из метода g. Для нахождения множества методов, которые могут быть исполнены при выполнении программы, строится транзитивное замыкание методов, достижимых из точек входа программы. Анализ достижимости методов – ключевой анализ при понижении избыточности, так как все последующие анализы удалимости анализируют код методов, выделенных на данном этапе.

Для языков с динамической диспетчеризацией вычисление множества методов, достижимых из данного, является нетривиальной задачей. В таких языках вызываемый метод не всегда определяется статически. Наиболее распространенным примером динамической диспетчеризации являются виртуальные вызовы. Механизм виртуальных вызовов используется для реализации полиморфизма

во многих объектно-ориентированных языках программирования. При наличии динамической диспетчеризации множество достижимых методов приходится вычислять консервативно. Вычисленное множество достижимых методов является надмножеством множества методов, которые будут выполнены при исполнении программы. Количество «лишних» методов будет зависеть от точности анализа. При понижении избыточности от точности анализа достижимости методов зависит точность последующих анализов удалимости и итоговая эффективность алгоритма.

В языке Java есть два механизма динамической диспетчеризации – это виртуальные и интерфейсные вызовы. В обоих случаях вызываемый метод определяется на этапе исполнения в зависимости от типа аргумента получателя. Аналогично прямым вызовам, виртуальные и интерфейсные вызовы содержат статическое описание вызываемого метода, однако при выполнении такие вызовы связываются с реализацией указанного метода в классе объекта-получателя.

```
class Base {
  void foo() {
    System.out.println("Base.foo");
  }
}
class Derived {
  void foo() {
    System.out.println("Derived.foo");
  }
}
Base object = factory();
object.foo();
```

В данном примере в зависимости от типа объекта `object` может быть вызван либо метод `Base.foo`, либо метод `Derived.foo`.

При статическом анализе информация об объектах времени исполнения отсутствует. Можно считать, что вызов может быть связан с любым методом, совместимым по имени и сигнатуре. Однако такой подход крайне неэффективен. Обычно для анализа достижимости используется консервативное приближение множества типов объектов-получателей. Рассмотрим существующие подходы к вычислению этого множества.

Алгоритмы Class Hierarchy Analysis и Rapid Type Analysis

В самом простом случае можно считать, что множество типов объекта получателя ограничено статическим типом получателя. То есть для Java множество возможных типов – это множество всех подклассов статического класса объекта-получателя.[1] Так, в примере выше статический класс объекта получателя `object` – класс `Base`, множество возможных типов объекта получателя – {`Base`, `Derived`}. Такой подход называется Class Hierarchy Analysis (CHA) [49].

Алгоритм CHA неэффективен при анализе универсальных библиотек. Такие библиотеки часто содержат большое число различных реализаций базовых интерфейсов или классов. Обычно приложение использует небольшую часть этих реализаций, но для большей гибкости при работе с объектами используются интерфейсы базовых типов.

Примером может служить библиотека стандартных коллекций Java. В пакете `java.util` содержится 10 классов, реализующих интерфейс `List`. Для следующего кода достижимыми будут считаться реализации метода `add` всех 10 классов:

```
List<String> list = new ArrayList<String>();
list.add("one");
```

Точность анализа можно повысить, если учесть, что нестатический метод класса может быть вызван только тогда, когда существуют или могут быть созданы объекты данного класса или его подклассов. Уточненный алгоритм называется Rapid Type Analysis (RTA) [15]. В процессе работы алгоритма вычисляется множество косвенных вызовов и множество классов, экземпляры которых могут быть использованы для косвенных вызовов. Обозначим эти множества *IndirectInvocations* и *InstantiableClasses* соответственно. Итеративный алгоритм начинается с добавления в замыкание точек входа приложения. Алгоритм завершается по достижению неподвижной точки. На каждой итерации выполняются следующие действия.

1. Для каждого ранее не обработанного метода из замыкания просматривается его код:
    a) методы, вызываемые инструкциями прямого вызова, добавляются в замыкание;

[1] Здесь и далее будем считать, что сам класс принадлежит множеству его подклассов.

б) косвенные вызовы добавляются в множество *IndirectInvocations*;

в) классы, используемые для создания объектов с помощью инструкции `new`, добавляются в множество *InstantiableClasses*.

2. Для каждого вызова из множества *IndirectInvocations* вычисляется пересечение множества подклассов статического класса вызова с множеством *InstantiableClasses*. Полученное пересечение – это классы, экземпляры которых могут быть использованы в качестве объектов-получателей для данного вызова. Реализации вызываемого метода в данных классах добавляются в замыкание.

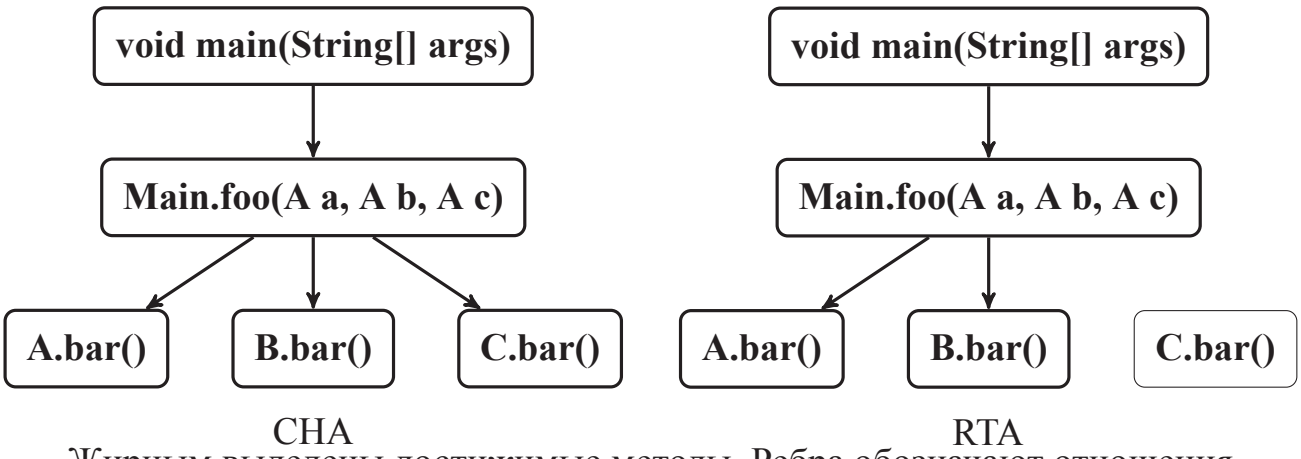
Классы, принадлежащие множеству *InstantiableClasses*, далее будем называть инстанциируемыми.

Продемонстрируем различие между алгоритмами CHA и RTA на примере из листинга 1.1. Результат анализов достижимости для данной программы приведен на рисунке 1.1

Листинг 1.1 Пример приложения для анализа достижимости методов

```
class A {
  void bar() {};
}
class B extends A {
  void bar() {};
}
class C extends A {
  void bar() {};
}
class Main {
  public static void main(String[] args) {
    new B();
    foo(new A(), new A(), null);
  }

  static void foo(A a, A b, A c) {
    c.bar();
  }
}
```



Жирным выделены достижимые методы. Ребра обозначают отношения достижимости.

Рисунок 1.1 — Результат анализов CHA и RTA для программы из листинга 1.1

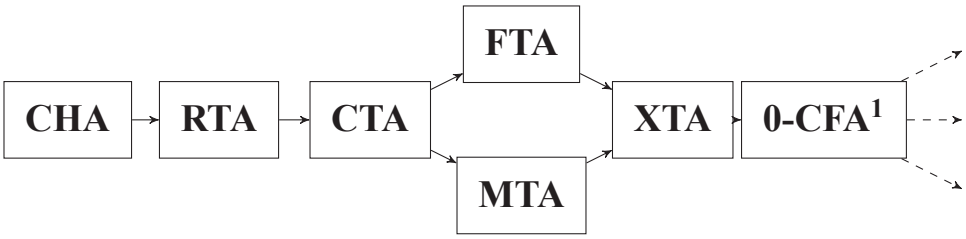
Алгоритм X Type Analysis

Для оценки возможных типов объектов в любой точке программы алгоритм RTA использует одно множество $InstantiableClasses$. Более точные алгоритмы строят граф потока данных программы и вычисляют множество достигающих типов для каждого узла в графе. Алгоритмы отличаются точностью моделирования потока данных. Например, алгоритм XTA, описанный в статье [50], моделирует поток данных между методами программы. В языке Java между методами объекты передаются через аргументы, возвращаемые значения и ссылки в куче (ссылки могут содержаться в полях объектов и элементах массивов). Еще одна ситуация, при которой объекты передаются между методами, – это бросание исключений. Поскольку брошенное исключение может быть обработано любым методом, находящимся выше по стеку вызовов, такие зависимости анализировать сложнее. Однако так могут быть переданы только объекты-исключения – экземпляры подклассов класса `java.lang.Throwable`. Учитывая, что множество подклассов класса `java.lang.Throwable` обычно невелико, предложенный алгоритм не моделирует бросание исключений точно, а использует одно множество для описания возможных подклассов класса `java.lang.Throwable` в любой точке программы.

Поток данных можно моделировать с разной точностью. Авторами рассмотрены несколько вариаций алгоритма.

1. CTA – для каждого класса используется одно множество, описывающее достигающие типы для всех полей и методов этого класса.
2. MTA – для каждого класса используется одно множество, описывающее достигающие типы для полей этого класса. Каждому методу соответствует одно множество достигающих типов.
3. FTA – для каждого класса используется одно множество, описывающее достигающие типы для методов этого класса. Каждому полю соответствует одно множество достигающих типов.
4. XTA – для каждого метода и поля исполюзуются отдельные множества.

На рисунке 1.2 отражено, как авторы позиционируют предложенные алгоритмы по сравнению с алгоритмами анализа достижимости методов, описанными ранее [50, с. 3, рисунок 1].



От менее точных слева к более точным справа.

Рисунок 1.2 — Алгоритмы анализа достижимости методов

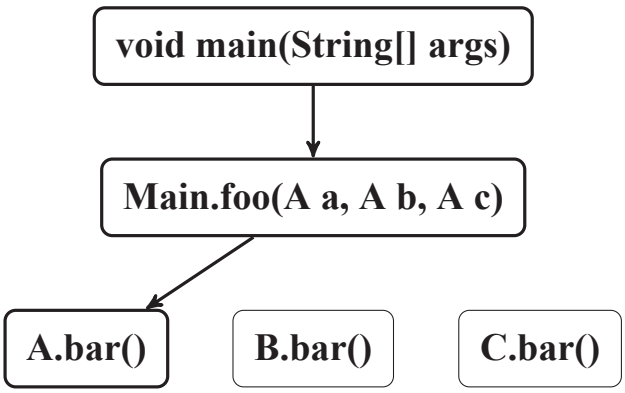
При моделировании потока данных алгоритм не учитывает порядок выполнения программы и состояние локальных переменных.

Результат анализа XTA для программы из листинга 1.1 приведен на рисунке 1.3. Авторы утверждают, что использование алгоритма XTA позволяет сократить количество достижимых методов в среднем на 1,6 % по сравнению с алгоритмом RTA.

Алгоритмы Declared Type Analysis, Variable Type Analysis

Другим примером более точного анализа является алгоритм Variable-Type Analysis (VTA) [51], который моделирует поток данных между переменными. Алгоритм VTA строит type propagation граф, узлы которого моделируют переменные ссылочных типов, а именно:

[1] Алгоритмы k-CFA рассматриваются далее в разделе «Алгоритмы k-Control Flow Analysis»



XTA

Жирным выделены достижимые методы. Ребра обозначают отношения достижимости.

Рисунок 1.3 — Результат анализа XTA для программы из листинга 1.1

– локальные переменные,
– поля,
– элементы массивов,
– аргументы,
– возвращаемые значения.

Разные экземпляры одной переменной моделируются с помощью одного узла в графе. Ребра в графе обозначают возможные присваивания между моделируемыми объектами при выполнении программы. Передача объектов через аргументы и возвращаемые значения моделируется аналогично присваиванию.

В отличие от алгоритма RTA, алгоритм VTA – пессимистичный. Для моделирования передачи объектов между функциями необходимо знать, с какими методами может быть связан каждый вызов. Для косвенных вызовов это заранее неизвестно. Для решения этой проблемы алгоритм VTA использует предварительно вычисленный консервативный граф вызовов. Такой граф может быть построен с помощью алгоритмов CHA или RTA. При моделировании передачи объектов через аргументы и возвращаемые значения косвенных вызовов предполагается, что косвенный вызов может быть связан с любым из методов из консервативного графа вызовов. Таким образом, эффективность алгоритма зависит от исходного консервативного графа вызовов.

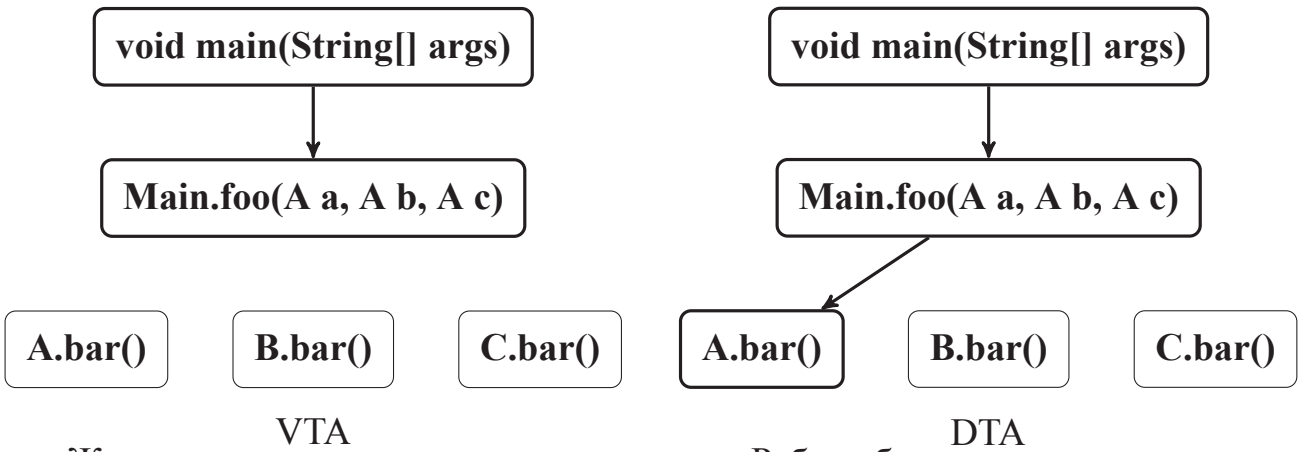
При наличии native-методов их операции также моделируются в type propagation графе, однако их код не анализируется автоматически, а описывается вручную.

С каждым узлом в type propagation графе ассоциируется множество достигающих типов *ReachingTypes*. Это множество классов, на экземпляры которых может указывать соответствующая переменная. Для вычисления множеств достигающих типов в коде методов анализируются присваивания вида `lhs = new A()`. Для каждого выражения такого вида в множество *ReachingTypes* узла, описывающего переменную lhs, добавляется класс `A`. Затем множество *ReachingTypes* для каждого узла вычисляется как объединение множеств *ReachingTypes* всех узлов достижимых из данного.

Авторами также предложена вариация алгоритма под названием Declared-Type Analysis (DTA), которая моделирует разные переменные одного типа с помощью одного узла в графе.

Описанные алгоритмы потоко-нечувствительны, то есть не учитывают порядок выполнения программы.

Результат анализов VTA и DTA для программы из листинга 1.1 приведен на рисунке 1.4. Согласно данным, приведенным авторами, использование алгоритма VTA сокращает количество достижимых методов по сравнению с алгоритмом RTA в среднем на 11,4 %. Для алгоритма DTA эта цифра значительно меньше – 3 %. Стоит отметить, что эти данные и цифры, приведенные для алгоритма XTA, получены разными авторами для разного набора приложений и не могут быть использованы для непосредственного сравнения.



Жирным выделены достижимые методы. Ребра обозначают отношения достижимости.

Рисунок 1.4 — Результат анализов VTA и DTA для программы из листинга 1.1

Алгоритмы k-Control Flow Analysis

В статье [52] представлен обобщенный механизм для построения графа вызова для языков с динамической диспетчеризацией. Используя обобщенный механизм, авторы рассматривают множество потоко-чувствительных и контекстно-чувствительных алгоритмов анализа, включая k-Control Flow Analysis (k-CFA), Cartesian Product Algorithm (CPA), Simple Class Set (SCS). В таких алгоритмах для каждой переменной в графе потока данных может существовать несколько узлов, соответствующих разным контекстам исполнения.

Авторы отмечают, что использование потоко-чувствительного анализа для удаления недостижимых методов сокращает итоговый размер приложения на 0–3 % по сравнению с потоко-нечувствительным анализом. Использование контекстно-чувствительного анализа не оказывает измеримого влияния на размер приложения.

Анализ указателей

Как было показано выше, задача анализа достижимости методов сводится к задаче оценки множества возможных типов объектов-получателей. В общем случае на вопрос о том, на какие объекты может указывать данная переменная, отвечает анализ указателей (points-to анализ). Описанные выше алгоритмы можно считать специализированными версиями анализа указателей. Классические алгоритмы анализа указателей, такие как анализ Андерсена [53] и анализ Стеенсгарда [54], были сформулированы для языков C и C++. Впоследствии эти алгоритмы были адаптированы для анализа Java-кода [55; 56]. Применение анализа указателей для анализа достижимости методов и построения графа вызовов описывается в статьях [57—60]

Многочисленные исследования [50—52; 61] показывают, что более точные алгоритмы незначительно уменьшают число достижимых методов, но существенно уменьшают количество ребер в графе вызовов. Сокращение числа ребер в графе позволяет более точно идентифицировать мономорфные вызовы, которые

впоследствии могут быть оптимизированы [62]. Кроме того, сокращение числа ребер повышает эффективность межпроцедурных оптимизаций [52].

1.2 Удаление неиспользуемых полей

Для известного набора достижимых методов можно вычислить множество полей, которые используются достижимым кодом, и удалить неиспользуемые. В статье [17] рассматривается задача удаления неиспользуемых полей для кода на C++. Для решения этой задачи анализируется код достижимых методов.

1. Если в коде встречается операция чтения поля `e'.m` или операция вычисления адреса поля `&e'.m`, соответствующее поле считается используемым.
2. Если в коде встречается операция вычисления смещения поля в объекте `&Z::m`, поле считается используемым.
3. Если в коде встречается небезопасное приведение типов в виде `(T)(e')`, все поля класса, описывающего тип выражения `e'`, считаются используемыми.

После анализа всех достижимых методов анализируются типы объединения (`union`). Если объединение содержит хотя бы одно используемое поле, то все поля, прямо или косвенно содержащиеся в объединении, считаются используемыми.

После анализа объединений поля, не помеченные как используемые, удаляются. Описанный алгоритм позволяет удалять поля, для которых в достижимом коде есть операции записи, но нет операций чтения. Авторы отмечают, что удаление таких полей важно на практике, так как неиспользуемые поля часто инициализируются в конструкторе класса.

Аналогичный подход широко используется для удаления неиспользуемых полей в Java-коде. В статьях [18—21] аналогичные алгоритмы используются в контексте понижения избыточности Java-программ в закрытой модели. Удаление неиспользуемых полей для Java упоминается в статье [63] наряду с другими механизмами уменьшения потребления памяти. В статьях [18—20; 63] авторы упоминают, что поля, для которых в достижимом коде нет операций чтения, могут быть удалены, операции записи таких полей также удаляются. Таким образом, аналогично алгоритму, описанному в [17], данные алгоритмы позволяют удалять

неиспользуемые поля, инициализируемые в конструкторах и статических инициализаторах классов.

Необходимо отметить, что в Java удаление объектного поля, для которого в достижимом коде есть операции записи, но нет операций чтения, может нарушить достижимость объектов, удаление которых приложение может отследить. Данная оптимизация нарушает семантику финализации объектов и может повлиять на видимое поведение программы. Подробнее семантика финализации объектов в Java рассматривается далее в разделе 2.2.

В статье [63] отмечается, что согласно спецификации языка при записи в поле выполняются проверки, которые могут приводить к бросанию исключений, например, проверка на `null`. При удалении записей неиспользуемых полей такие проверки должны сохраняться.

1.3 Специализация иерархии классов

В статье [14] рассматривается задача понижения избыточности иерархии C++ классов. Статья описывает алгоритм Class Hierarchy Slicing, который удаляет из иерархии классов программы поля, методы, классы и отношения наследования, не требуемые для выполнения программы. Для этого анализируется код программы и вычисляется используемое подмножество иерархии классов. При анализе кода программы используется пессимистичный подход, а именно изначально предполагается, что все методы выполнимы. Удаляется только то подмножество иерархии классов, которое нигде не используется.

Развитием вышеописанного алгоритма является алгоритм специализации иерархии классов (Class Hierarchy Specialization) [16]. Данный алгоритм анализирует использования переменных. Для разных переменных одного типа алгоритм создает специализированные версии классов, которые включают в себя только те члены, которые используются с помощью конкретной переменной. Такое преобразование позволяет сократить потребление памяти.

1.4 Ромизация

При каждом старте и инициализации виртуальной машины выполняются одни и те же действия. Создаются системные объекты, загружаются классы стандартных библиотек, разрешаются символические ссылки, инициализируются загруженные классы. Этот процесс можно оптимизировать, если сохранить состояние инициализированной виртуальной машины в виде загрузочного образа. Тогда при старте достаточно будет восстановить состояние из сохраненного образа. При таком подходе выполнение программы разделяется на инициализацию и исполнение. Инициализация осуществляется специальной версией виртуальной машины на хостовом устройстве. Затем инициализированный образ переносится для исполнения на целевое устройство. При этом, помимо инициализации, могут осуществляться различные оптимизации, например, удаление неиспользуемых методов, полей и классов. Такой подход называется ромизацией (romization) или раздельной инициализацией и часто используется для оптимизации виртуальных машин, предназначенных для работы на устройствах с ограниченными ресурсами [27—29; 64; 65]. Ромизация позволяет сократить время запуска и потребление памяти. Кроме того, при ромизации появляется возможность оптимизировать исполняемое представление программы на хостовом устройстве. Отметим, что такие оптимизации далеко не всегда возможны на целевых устройствах в связи с ограниченностью ресурсов.

Для языка Java, согласно спецификации языка и виртуальной машины загрузка, разрешение символических ссылок и инициализация должны производиться лениво при первом обращении к методу, полю или классу. Такое требование к реализации языка затрудняет проведение границы между инициализацией приложения и его исполнением. В большинстве случаев при ромизации загрузку и связывание осуществляют на этапе подготовки образа. Однако такая оптимизация не всегда безопасна и в некоторых случаях может привести к нарушению поведения программы. Например, в программе может существовать недостижимый код, который ссылается на несуществующий класс. При ленивом связывании программа исполнится без ошибок, однако раннее связывание приведет к ошибке.

В вышеупомянутых работах данная проблема либо не рассматривается, либо подразумевается, что несмотря на то, что такой код корректен с точки зрения спецификации виртуальной машины, правильно написанное приложение не должно содержать неразрешимых ссылок.

При подготовке образа на хостовом устройстве может осуществляться инициализация всех или некоторых классов программы [28; 29]. Это сокращает время инициализации приложения на целевом устройстве, ускоряет исполнение и в некоторых случаях позволяет уменьшить размер образа. В Java инициализация класса подразумевает исполнение статического инициализатора – специального метода с именем `<clinit>`. Этот метод не может быть вызван стандартными средствами, а значит, может быть удален у инициализированных классов.

Спецификация языка Java требует ленивой инициализации классов. В некоторых случаях ранняя инициализация может нарушить видимое поведение программы. Авторами [66] рассматривается задача определения последовательности инициализации классов, при которой ранняя инициализация не нарушает видимого поведения программы.

В статье [30] развивается идея раздельной инициализации для встраиваемых систем. Авторами предлагается перенести создание и инициализацию потоков приложения с этапа исполнения на этап подготовки загрузочного образа. В статье отмечается, что такой подход не только ускоряет запуск приложения, но и сокращает число достижимых методов, так как позволяет удалить с устройства код, который используется только при инициализации.

Механизм аналогичный раздельной инициализации используется для ускорения запуска приложений в платформе Android [67]. При старте устройства создаётся Zygote-процесс, который представляет собой преинициализированную виртуальную машину. При запуске каждого нового приложения порождается fork-копия Zygote-процесса. В данном случае преинициализация осуществляется на целевом устройстве, однако результат преинициализации используется для множества приложений.

1.5 Алгоритм Reachable Member Analysis

Несмотря на большое количество публикаций по теме анализа достижимости кода, большинство алгоритмов анализируют только код программы и не имеют дела с её состоянием. В инициализированном состоянии виртуальной машины существуют зависимости между методами, полями и объектами, которые не разрешаются классическими алгоритмами.

Впервые задача анализа достижимости методов при наличии инициализированного состояния была описана в статье [30]. Описывая систему раздельной инициализации для Java, авторы предлагают удалять неиспользуемые методы на этапе инициализации загрузочного образа, после того как инициализированы потоки приложения. Для анализа достижимости методов предложено использовать алгоритм CHA. Данный алгоритм предполагает, что в качестве объекта-получателя косвенного вызова может быть использован экземпляр любого статически совместимого типа. В отличие от более точных алгоритмов, алгоритм CHA анализирует только иерархию классов и не зависит от инициализированного состояния программы.

Более точный анализ достижимости методов требует анализа инициализированного состояния программы. Примером такого анализа может служить алгоритм RMA, используемый для удаления недостижимых методов в языке программирования Virgil [28]. Данный язык используется для разработки приложений для встраиваемых устройств. Вся инициализация программы на языке Virgil осуществляется на этапе компиляции. На этапе компиляции исполняются конструкторы компонент, которые могут содержать произвольный код. Язык Virgil не поддерживает динамического создания объектов, все используемые программой объекты должны быть созданы во время инициализации.

После инициализации программы на этапе компиляции осуществляется удаление недостижимых методов. В языке используется динамическая диспетчеризация в виде виртуальных вызовов и делегатов. Таким образом, при анализе достижимости методов возникает задача оценки множества возможных типов объектов-получателей. Так как язык не поддерживает динамического создания объектов, на момент анализа достижимости методов известны все доступные приложению объекты. Анализируемый код при этом не может создавать новых объектов.

Авторы рассматривают подход, при котором возможными объектами-получателями считаются все существующие в памяти объекты совместимого типа. Такой подход является адаптацией алгоритма RTA для ситуации, когда все объекты приложения известны заранее и анализируемый код не может создавать новые объекты. При этом отмечается, что при таком подходе после удаления неиспользуемых полей некоторые из объектов могут стать недостижимы и будут удалены сборщиком мусора. Повторное применение алгоритма позволяет удалить некоторые из методов ставшие недостижимыми. Однако, если между недостижимыми методами, полями и объектами существуют циклические зависимости, то они не будут удалены.

В приведенном ниже примере при инициализации программы будут созданы объекты классов `A`, `B` и `C`. При анализе достижимости методы `m` всех трех классов будут считаться достижимыми. При удалении неиспользуемых полей будет удалено поле `h`, объект класса `C` станет недостижим и может будет удален сборщиком мусора. Повторное применение алгоритма анализа достижимости позволит удалить метод `m` в классе `C`.

Примером циклической зависимости служит поле `g`, которое ссылается на объект класса `B`, и метод `m` в классе `B`, который ссылается на поле `g`. Несмотря на то, что они не используются достижимым кодом, они не будут удалены.

```
component Main {
 field f: A = new A();
 field g: A = new B();
 field h: A = new C();
 method entry() {
 while ( true ) f = f.m();
 }
}
class A {
 method m(): A { return this; }
}
class B extends A {
 method m(): A { return Main.g; }
}
class C extends A {
 method m(): A { . . . }
}
```

Отмечая недостатки итеративного алгоритма, авторы предлагают альтернативный подход, при котором возможными объектами-получателями считаются

только те объекты, которые достижимы через читаемые приложением поля. Авторы описывают алгоритм Reachable Member Analysis (RMA), который использует данный подход для анализа достижимости методов, полей и объектов.

В процессе работы алгоритма для каждого класса вычисляется три множества:

– множество используемых членов этого класса *members(class)*,
– множество инстанциированных подтипов класса *subtypes(class)*,
– множество объектов экземпляров класса *objects(class)*.

Кроме того, в процессе работы алгоритма явно вычисляется множество достижимых методов *methods*.

Алгоритм использует рабочее множество, элементами которого могут быть методы, классы, операции чтения полей, вызовы методов или объекты. При обработке элемента рабочего множества зависимости элемента добавляются в рабочее множество, а сам элемент добавляется в одно из соответствующих множеств.

– Методы добавляются в множество достижимых методов methods.
– Классы добавляются в множества *subtypes(class)* соответствующих надклассов.
– Вызовы методов и чтения полей добавляются в множества используемых членов *members(class)* соответствующих классов.
– Объекты добавляются в множества *objects(class)* соответствующих классов.

Формальное описание алгоритма приведено в Приложении A.1.

1.6 Понижение избыточности Java-программ

Вышеописанные алгоритмы часто применяют для понижения избыточности Java программ. Так, в статьях [18—20; 61] авторы описывают инструмент Jax Application Extractor, который осуществляет выделение минимального подмножества библиотек для заданного приложения. Основные возможности инструмента.

– Удаление недостижимых методов. Для анализа достижимости используется алгоритмы CHA, RTA, XTA.

– Удаление неиспользуемых полей с помощью алгоритма, описанного в разделе 1.2. Неиспользуемыми считаются поля, для которых в достижимом коде нет операций чтения.
– Удаление неиспользуемых классов.
– Объединение смежных классов в иерархии наследования при условии, что такое преобразование не увеличивает размер экземпляров класса.
– Переименование классов, методов и полей.

Авторы отмечают, что зависимости рефлексивного кода в общем случае не могут быть проанализированы статически. Данный инструмент позволяет описать такие зависимости вручную. Кроме того, отмечается, что анализируемый Java-код может содержать native-методы, реализация которых написана на другом языке, чаще всего C/C++. Native-методы могут создавать экземпляры классов, вызывать другие Java методы, обращаться к полям. Такие зависимости также должны быть описаны вручную.

Виртуальная машина JamaicaVM может быть специализирована для заданного приложения с помощью инструмента Jamaica Builder [68]. Результатом работы инструмента является исполняемый файл, включающий в себя виртуальную машину, приложение и библиотеки, необходимые для его исполнения. Во время специализации могут быть удалены методы, поля и классы, не используемые приложением. Документация виртуальной машины не описывает используемые алгоритмы, но некоторые свойства используемых алгоритмов были определены экспериментальным путем.

– Анализ достижимости методов использует алгоритм, аналогичный алгоритму RTA.
– Анализ удалимости полей позволяет удалять поля примитивных типов, для которых в коде достижимых методов есть операции записи, но нет операций чтения. Для полей объектных типов такая оптимизация не выполняется.

Jamaica Builder предоставляет возможность указать классы, которые должны быть безусловно включены в специализированную виртуальную машину. В итоговый исполняемый файл будут включены все методы и поля данных классов вне зависимости от результатов анализа достижимости. Данная возможность должна быть использована для описания зависимостей native-методов, и кода, использующего рефлексию.

В некоторых виртуальных машинах алгоритмы понижения избыточности используются для оптимизации загрузочного образа при раздельной инициализации. Механизм раздельной инициализации, описанный в [30], был реализован в системе Java In The Small (JITS). При подготовке образа на хостовом устройстве осуществляется инициализация потоков приложения, после чего удаляются методы, поля, объекты и классы, не используемые приложением. Для вычисления множества достижимых методов используется алгоритм CHA. При этом авторы не описывают алгоритмы анализа удалимости полей и классов.

Другой пример использования алгоритмов понижения избыточности для оптимизации загрузочного образа описан в статье [29]. Авторами описывается система ExoVM, автоматически специализирующая виртуальную машину для заданного приложения. Для ускорения запуска и сокращения потребления памяти ExoVM использует раздельную инициализацию. Приложение загружается и инициализируется полнофункциональной версией виртуальной машины. В процессе инициализации разрешаются все символические ссылки и инициализируются все загруженные классы. При этом в статье отмечается, что спецификация языка требует ленивой загрузки и инициализации, и что ранняя инициализация всех классов программы может нарушить ее поведение. Подробнее семантика инициализации классов рассмотрена далее в разделе 2.1. После того, как все классы приложения инициализированы, осуществляется анализ достижимости, называемый feature analyis. В отличие от Jax Application Extractor, система ExoVM анализирует достижимость не только для сущностей языка, но и для отдельных компонентов виртуальной машины. В статье определяются отношения достижимости между различными сущностями языка (методами, полями, объектами и классами) и компонентами виртуальной машины. В загрузочный образ специализированной виртуальной машины включаются только те сущности и компоненты, которые достижимы из точек входа приложения.

Так как анализ достижимости осуществляется после инициализации классов, на момент анализа в памяти могут существовать объекты, созданные в процессе инициализации. Эти объекты необходимо учитывать при анализе достижимости методов. Для анализа достижимости методов ExoVM использует подход, основанный на алгоритмах RTA и RMA. А именно возможными объектами-получателями косвенных вызовов считаются:

– объекты инстанциируемые достижимым кодом;

– инстанциированные объекты, достижимые через ссылки в полях, читаемых достижимым кодом.

Для удаление неиспользуемых полей ExoVM использует критерий удалимости, описанный в разделе 1.2. Удалимыми считаются поля, для которых в достижимом коде нет операций чтения. Данный критерий позволяет удалять неиспользуемые поля, инициализируемые в конструкторах или инициализаторах классов. Однако, как было отмечено ранее, такая оптимизация нарушает семантику финализации объектов и может изменить видимое поведение программы.

При анализе достижимости зависимости native-методов не анализируются автоматически. Вместо этого реализация анализа содержит описание зависимостей для известных native-методов стандартной библиотеки классов. Если в анализируемом приложении оказывается достижим native-метод, для которого неизвестны зависимости, анализ останавливается с ошибкой.

В статье [21] рассматривается еще один пример системы для выделения минимального подмножества библиотек для заданного Java-приложения. Выделение подмножества осуществляется с помощью графа зависимостей. Узлы в графе описывают сущности приложения и библиотек, такие как классы, методы, поля, интерфейсные и виртуальные вызовы. Ребра в графе отражают зависимости между ними, например, отношения наследования между классами, зависимости между методами и определяющими их классами, зависимости методов, определяемые их байт-кодом. Зависимости косвенных вызовов вычисляются с помощью анализа CHA. Впоследствии эти зависимости уточняются с помощью анализа инстанциируемых типов (RTA). Минимальное подмножество библиотек вычисляется как транзитивное замыкание графа зависимостей для заданных точек входа приложения. Зависимости native-методов и кода, использующего рефлексию, предлагается описывать в качестве дополнительных точек входа приложения. Стоит отметить, что зависимости, описанные таким образом, будут включены в замыкание вне зависимости от того, включен ли зависящий от них код в замыкание или нет.

Все вышеописанные реализации понижения избыточности реализуют консервативный анализ используемости компонентов. В итоговое приложение и библиотеки включаются те компоненты, для которых не удалось доказать неиспользуемость. Более сложные и точные алгоритмы анализа позволяют доказать неиспользуемость большего числа компонентов, то есть позволяют сократить

итоговый размер приложения и библиотек. В статьях [24; 25; 69] предлагается альтернативный подход, при котором целевое устройство может загружать отсутствующие компоненты по сети. Это позволяет удалять неиспользуемые компоненты более агрессивно. Даже если в результате анализа не удалось доказать неиспользуемость компонента, этот компонент может быть удален спекулятивно. Если впоследствии компонент потребуется для выполнения программы, он будет загружен по сети. Такая конфигурация позволяет сократить потребление памяти на целевом устройстве по сравнению с реализациями, использующими консервативный анализ.

1.7 Анализ рефлексии

Все механизмы понижения избыточности, перечисленные выше, требуют ручного описания зависимостей рефлексивного кода. Задача автоматического анализа таких зависимостей при анализе достижимости методов рассматривается в статье [70]. Для анализа достижимости методов используется оптимистичный алгоритм, построенный на базе анализа указателей [60].

В некоторых случаях анализ указателей позволяет автоматически вычислить зависимости рефлексивного кода. Так, например, с помощью анализа указателей вычисляются возможные значения строкового аргумента для каждого вызова `java.lang.Class.forName`. На этапе исполнения значение аргумента используется для поиска класса по имени. В тех случаях, когда все возможные значения аргумента являются константами, зависимости вызова однозначно определяются значениями этих констант. Если некоторые из возможных значений не являются константами, алгоритм требует ручного описания этих значений. В общем случае задача анализа возможных значений строковых переменных обсуждается в статье [71].

Для больших приложений количество точек, в которых требуется ручное описание значений, может быть велико. Авторы отмечают, что в случаях, когда рефлексия используется для создания экземпляров объектов, результирующий объект часто приводится к некоторому общему типу.

```
String className = ...;
Class c = Class.forName(className);
```

```
Object o = c.newInstance();
T t = (T) o;
```

Предполагая, что в корректной программе приведение типа всегда успешно, этот тип можно использовать как верхнюю границу возможных типов для вызова `Class.forName`. Данное наблюдение позволяет сократить количество точек, в которых требуется ручное описание значений аргументов. Статический анализ вышеописанной идиомы затрудняет тот факт, что данные операции могут находиться в разных методах. Для того чтобы ассоциировать приведение типа с вызовом `Class.forName`, используется анализ типов.

Обращения к полям и вызовы методов через рефлексию анализируются аналогичным образом. Анализ указателей используется для того, чтобы ассоциировать объекты классов `java.lang.reflect.Field`, `java.lang.reflect.Method`, `java.lang.reflect.Constructor` с точками их создания. Примерами точек создания могут служить вызовы методов `getMethod`, `getField` на объектах класса `Class`. Возможные значения аргументов в точках создания, вычисленные с помощью анализа указателей, определяют методы и поля, с которыми связаны соответствующие объекты.

Инструмент для статического анализа рефлексивного кода ELF [72] развивает идеи, предложенные в [70]. В тех случаях, когда анализ аргументов точек создания не позволяет однозначно связать экземпляры классов `Field`, `Method` и `Constructor` с соответствующими полями и методами, ELF анализирует использования данных объектов.

```
Method m = ...;
Field f1 = ...;
Field f2 = ...;

r = (X) m.invoke(o, arguments);
f2.set(o, (Y) f.get(o));
```

Так, для вызова `m.invoke(o, arguments)` статический тип объекта-получателя `o` определяет верхнюю границу множества классов, которым может принадлежать метод. Тип аргументов `arguments` и приведение типа результата вызова к классу `X` налагает ограничения на сигнатуру вызова. По заданным ограничениям можно статически определить множество методов, связываемых с данным вызовом.

Для операций доступа к полю тип объекта `o` определяет верхнюю границу множества классов, которым может принадлежать поле. Приведение результата

чтения к типу `Y` налагает ограничение на тип поля, аналогичным образом тип записываемого в поле значения ограничивает множество возможных типов поля. Заданные ограничения статически определяют множество используемых полей. В статьях [73—75] обсуждаются алгоритмы анализа, уточняющие данный подход.

Альтернативный подход к анализу рефлексивных зависимостей состоит в динамическом анализе зависимостей на этапе исполнения. Для этого необходимо инструментировать рефлексивные вызовы таким образом, чтобы в процессе исполнения сохранять информацию о методах, полях и классах, использованных через рефлексию. Затем информация, собранная в процессе исполнения, может быть использована для уточнения статического анализа [76; 77]. Подробный обзор механизмов для анализа рефлексивных зависимостей приводится в статье [78].

1.8 Специализация набора инструкций

Сокращение размеров исполняемого кода приложений актуально во многих областях применения. Сюда входят разработка программного обеспечения для ограниченных в ресурсах устройств; ускорение загрузки и уменьшение размера передаваемых по сети файлов. В зависимости от области применения к механизмам сжатия предъявляются разные требования. Например, ограниченное в процессорной мощности и объеме оперативной памяти устройство не может распаковать сжатую программу перед ее исполнением. В этих условиях использование интерпретируемого представления вместо машинного кода является удачным и часто применяемым подходом.

Интерпретируемое представление не требует предварительного декодирования, чем выгодно отличается от других механизмов. Такое представление может быть в несколько раз компактнее соответствующего машинного кода за счет более высокоуровневого набора инструкций [79]. Платой за компактность при этом являются накладные расходы на интерпретацию. В случае встраиваемых систем компромисс между скоростью и размером зачастую решается в пользу размера и накладные расходы на интерпретацию считаются приемлемыми [45]. Примером широко распространенного интерпретируемого представления может служить Java байт-код [80].

В закрытой модели можно достичь еще более компактного кодирования за счет специализации набора инструкций для заданного приложения. Новый набор инструкций может быть выбран таким образом, чтобы уменьшить суммарный размер заданного приложения и интерпретатора, необходимого для его выполнения. Кодирование часто встречающихся последовательностей байт-кодов одной инструкцией может ускорить исполнение программы за счет устранения накладных расходов на диспетчеризацию инструкций в сжатой последовательности [27; 45]. Кроме того, реализация последовательности в интерпретаторе может быть оптимизирована.

Рассмотрим существующие алгоритмы, использующие специализацию набора инструкций для сокращения размера исполняемого кода.

Clausen и другие, LZW-CC

В статье [36] рассматривается задача сокращения размера Java-приложений для встраиваемых устройств. Авторы отмечают, что большинство Java-программ содержат часто повторяющиеся последовательности инструкций, и предлагают расширить стандартный набор инструкций инструкциями для кодирования таких последовательностей. Набор специализированных инструкций зависит от приложения, так как в разных приложениях часто встречающиеся последовательности могут отличаться. Вместо того, чтобы генерировать специализированный интерпретатор для каждого приложения, предлагается расширить переносимый формат описанием специализированных инструкций, называемых макроинструкциями. Такие инструкции кодируются свободными опкодами и задаются последовательностью исходных инструкций. Изменения в интерпретаторе в таком случае сводятся к поддержке исполнения макроинструкций.

Описанный подход сохраняет совместимость со стандартным набором инструкций, так как только дополняет его. При этом количество свободных опкодов оказывается невелико: от 52 до 152 в зависимости от стандарта Java. Для того чтобы кодировать большее количество макроинструкций, предлагается использовать двухбайтовое кодирование для некоторых макроинструкций. Такое кодирование оказывается менее компактным и используется для последовательностей, встречающихся менее часто. Утверждается, что, несмотря на менее компактное

кодирование, использование двухбайтовых инструкций сокращает итоговый размера кода.

Алгоритм сжатия исполняемого представления для заданного приложения состоит из следующих шагов.

– Вычисляется множество всех встречающихся последовательностей.
– Из вычисленного множества последовательностей выбирается подмножество, которое будет закодировано макроинструкциями. Для выбора последовательностей используется жадный алгоритм. Из всех последовательностей выбирается такая, замена которой на макроинструкцию приведет к наибольшему сокращению размера. Соответствующая макроинструкция добавляется в набор инструкций и все вхождения последовательности в потоке инструкций заменяются на добавленную инструкцию. Процесс повторяется до тех пор, пока есть свободные опкоды.

Оптимизацию, при которой специализированная инструкция используется для кодирования последовательности инструкций, далее будем называть сверткой последовательности инструкций.

В статье [37] описан аналогичный алгоритм сжатия интерпретируемого кода под названием LZW-CC. Алгоритм, основанный на алгоритме словарного сжатия LZW [81], находит часто встречающиеся последовательности инструкций и расширяет набор инструкций интерпретатора инструкциями, кодирующими такие последовательности. В отличие от механизма, описанного в [36], для каждого приложения предлагается генерировать специализированную версию интерпретатора. При генерации специализированного интерпретатора предложено удалять реализации неиспользуемых инструкций. Это позволяет сократить размер интерпретатора, однако делает интерпретатор несовместимым со стандартным набором инструкций. Кроме того, практические исследования показывают, что расширение набора инструкций интерпретатора позволяет добиться более эффективного кодирования [82]. Алгоритм LZW-CC реализован для Java байт-кода и для языка Haskell.

TakaTuka

Специализация набора инструкций используется для сокращения размера в реализации Java для микроконтроллеров TakaTuka [27]. В закрытой модели, когда не требуется совместимость со стандартным набором инструкций, для каждого приложения генерируется специализированный интерпретатор. Из реализации интерпретатора удаляются инструкции, не используемые приложением. Кроме того, удаляются инструкции, реализации которых для различных типов аргументов не отличаются.

Свободные опкоды используются для новых инструкций, которые обеспечивают более компактное кодирование программы. Следующие оптимизации применяются для сокращения размера индивидуальных инструкций.

– Укорачивание аргумента. Специализированная инструкция используется для более компактного кодирования аргумента исходной инструкции. Например, стандартные инструкции переходов используют 2 байта для кодирования относительного смещения перехода. Специализированная инструкция кодирует смещение с помощью 1 байта.
– Свертка аргумента. Специализированная инструкция фиксирует значение аргумента, аргумент не кодируется в потоке инструкций. Например, инструкция `iload 0x0010`, занимающая 2 байта, может быть закодирована с помощью специализированной инструкции с фиксированным аргументом `iload_0x0010`, занимающей 1 байт.

Также предлагается использовать специализированные инструкции для кодирования последовательностей опкодов. В отличие от вышеописанных алгоритмов, такие инструкции не фиксируют значения аргументов инструкций, составляющих последовательность. Значения этих аргументов описываются аргументами специализированных инструкций. Таким образом, новые инструкции кодируют шаблоны последовательностей инструкций, параметризованные значениями аргументов. Выбор шаблонов для кодирования специализированными инструкциями осуществляется с помощью алгоритма, аналогичного алгоритмам, описанным в [36; 37].

Авторы не рассматривают возможность совместного использования оптимизаций кодирования индивидуальных инструкций и свертки последовательности инструкций. Совместное использование оптимизаций является неочевидной

задачей, так как каждая из этих оптимизаций сокращает количество свободных опкодов, доступных для другой оптимизаций.

Saougkos, Manis и другие

Saougkos, Manis и другие [38] исследовали возможность использования более сложных шаблонов для создания специализированных инструкций. В статье рассматриваются шаблоны произвольной длины, параметрами которых могут быть не только аргументы инструкций, но и сами инструкции. Для выделения набора параметризуемых шаблонов используется иерархическая кластеризация [83; 84].

Шаблоны в описанном алгоритме могут фиксировать произвольное количество аргументов. При кодировании шаблона специализированной инструкцией фиксированный аргумент не нужно кодировать в потоке инструкций. За счет этого осуществляется свертка аргументов. Свертка аргументов при этом является частью процесса построения множества шаблонов и может осуществляться в контексте последовательностей инструкций. В отличие от реализации TakaTuka, описанный алгоритм не укорачивает аргументы.

Применение параметризованных шаблонов позволяет достичь высокой степени сжатия, однако исполнение специализированных инструкций, кодирующих таких шаблоны, сопряжено с дополнительными накладными расходами. Кроме того, процесс перебора множества возможных шаблонов при сжатии занимает много времени. Согласно данным, приведенным в статье, перебор шаблонов для всех классов пакета java.lang платформы MIDP 2.0 занимает до 12731 секунды. Суммарный размер байт-кода в данных классах составляет 3998 байт. Данные результаты получены на компьютере с процессором AMD Athlon XP 2400+ (2 GHz, 256 Kb L2 cache).

Специализация представлений в виде деревьев

Все рассмотренные выше алгоритмы применяют идею словарного сжатия к исполняемому представлению в виде линейных последовательностей инструкций. Словарное сжатие может быть также применено к представлениям в виде деревьев. Fraser и Proebsting [39] описывают компилятор языка C, оптимизирующий размер исполняемого кода. Результатом компиляции является интерпретируемое представление и интерпретатор, необходимый для его исполнения. Набор инструкций интерпретатора выбирается таким образом, чтобы сократить размер кода приложения и интерпретатора.

Описанный компилятор построен на базе компилятора lcc [85]. Front end этого компилятора генерирует внутреннее представление в виде деревьев. Это представление анализируется для выбора набора инструкций. Алгоритм выбора набора инструкций выглядит следующим образом.

- Генерируется промежуточное представление программы.
- Перебираются все поддеревья промежуточного представления с числом узлов меньше N. Для каждого поддерева генерируется множество параметризованных шаблонов. Параметрами шаблонов являются значения аргументов операторов, входящих в шаблон. При этом шаблон может фиксировать значение одного или нескольких аргументов. Порожденные шаблоны являются кандидатами для кодирования специализированными инструкциями.
- Подмножество операторов lcc, используемое программой, итеративно расширяется специализированными инструкциями. На каждом шаге выбирается шаблон, кодирование которого с помощью специализированной инструкции обеспечивает наибольшее сокращение размера. Процесс завершается по исчерпанию доступных опкодов.

Аналогично алгоритму, описанному в Разделе 1.8, специализированные инструкции могут содержать произвольное количество фиксированных аргументов, что позволяет осуществлять свертку аргументов. Свертка аргументов является частью процесса построения множества шаблонов и может осуществляться в контексте последовательностей инструкций.

Упрощение исходного набора инструкций

В [40] описан механизм сжатия промежуточного кода виртуальной машины OmniVM путем специализации регистрового RISC набора инструкций. При расширении набора инструкций применяются следующие оптимизации:

– свертка одного из аргументов,
– свертка пар последовательных опкодов.

Итеративное расширение набора инструкций позволяет порождать инструкции, кодирующие последовательности произвольной длины с произвольным набором фиксированных аргументов.

Исходный набор инструкций OmniVM не является минимальным. Некоторые из инструкций могут быть закодированы последовательностью более простых инструкций. Например, инструкция чтения из памяти с непосредственным операндом может быть закодирована через чтение с косвенным операндом. Авторы отмечают, что избыточные инструкции могут быть не оптимальны для заданного приложения и могут негативно влиять на эффективность сжатия.

В статье сравнивается эффективность алгоритма для различных вариантов исходного набора инструкций. Следующие избыточные инструкции были удалены из стандартного набора инструкций OmniVM:

– инструкции с непосредственными операндами,
– инструкции чтения и записи в память с адресацией отличной от регистровой.

Удаление таких инструкций перед специализацией набора инструкций приводит к сокращению итогового размера кода на 9%.

Отметим, что Java байт-код содержит избыточные инструкции. Например, инструкции вида `if_icmp<cond>` могут быть закодированы с помощью последовательностей вида `isub if<cond>`. Ни один из рассмотренных механизмов сжатия Java байт-кода не упрощает исходный набор инструкций.

1.9 Оптимизация кодирования инструкций

В тех случаях, когда потребление памяти не является ограничивающим фактором, например, если сжатие применяется для сокращения объема данных передаваемых по сети, оправдано применение декодируемого представления. Такое представление требует предварительного декодирования перед исполнением и зачастую более компактно.

Распределение вероятностей, с которыми встречаются различные инструкции, очень неравномерно. Используя этот факт, можно кодировать часто встречающиеся инструкции более компактно. В статье [86] рассматривается сжатие JVM байт-кода с помощью кода Хаффмана [87]. Недостатком данного подхода является замедление исполнения за счет усложнения декодирования инструкций. Проблеме скорости декодирования инструкций, записанных с помощью кода Хаффмана, посвящена статья [88].

Авторами [40] описано декодируемое представление для языка C. Размер программы в описанном представлении может быть в 4,9 раз меньше оригинальной программы. Генерация представления выглядит следующим образом.

- Генерируется промежуточное представление компилятора lcc.
- Сгенерированные деревья разделяют на поток опкодов и потоки операндов для каждого из опкодов. Каждый из этих потоков кодируется отдельно.
  - Применяется move-to-front (MTF) кодирование [89].
  - MTF-индексы кодируются с помощью кода Хаффмана.
- Итоговый файл, содержащий все потоки, кодируется с помощью утилиты gzip.

Разделение опкодов и операндов на различные потоки позволяет сгруппировать семантически схожие потоки данных, что обеспечивает более эффективное кодирование.

В статье [90] описывается алгоритм автоматической генерации компактного представления программы на языке Scheme и виртуальной машины для её исполнения. Для сжатия исполняемого кода используется комбинированный подход. Набор инструкций расширяется инструкциями, которые соответствуют часто встречающимся шаблонам последовательностей исходных инструкций. Для кодирования инструкций используется код Хаффмана.

1.10 Сокращение размера класс-файлов

Задача сокращения размера Java байт-кода часто рассматривается как часть более общей задачи компрессии Java класс-файлов. Стандартный формат класс-файлов не очень эффективен с точки зрения размера. Большую часть размера класс-файлов составляет символическая информация из пула констант, описывающая имена методов, полей и классов. На практике реализации Java для встраиваемых устройств часто используют альтернативные форматы класс-файлов [91—96].

1.11 Выводы

На основании вышеизложенного обзора можно сделать следующие выводы:

1. Понижение избыточности путем удаления неиспользуемых полей методов и классов широко применяется для сокращения размера приложения в закрытой модели. Задача понижения избыточности хорошо исследована. Многие из известных алгоритмов были предложены для языка C++ и затем были адаптированы для Java.
2. Удаление неиспользуемых, но инициализируемых полей важно на практике. Однако, такая оптимизация для Java приводит к нарушению семантики финализации объектов, так как может нарушить достижимость объектов, удаление которых приложение может отследить.
3. При использовании раздельной инициализации алгоритмы понижения избыточности должны анализировать не только код приложения, но и его инициализированное состояние. Так, например, объекты, созданные во время инициализации могут быть использованы в качестве объектов-получателей косвенных вызовов. Классические алгоритмы анализа достижимости методов не учитывают эти объекты и не могут быть применены при использовании раздельной инициализации.
4. Алгоритм RMA – единственный описанный в литературе алгоритм анализа достижимости методов, который принимает во внимание объекты, созданные во время инициализации. Первоначально данный алгоритм

был описан для языка Virgil, впоследствии был адаптирован для языка Java в реализации виртуальной машины ExoVM. Реализация ExoVM, однако, имеет существенный недостаток: на этапе подготовки образа ExoVM инициализирует все классы, вне зависимости от того, используются ли они достижимым кодом или нет. При этом может быть нарушено видимое поведение приложения.

5. Native-методы затрудняют автоматический анализ зависимостей. Зависимости таких методов либо зафиксированы в реализации анализа как в ExoVM, либо описываются вручную. При большом количестве native-методов ручное описание зависимостей неэффективно и чревато ошибками.
6. Использование интерпретируемого кода позволяет сократить размер приложения по сравнению с машинным кодом. Специализация набора инструкций для заданного приложения позволяет добиться еще более компактного кодирования. Такое кодирование не требует предварительной декомпрессии и поэтому широко применяется для встраиваемых систем с ограниченными ресурсами.
7. Специализированные инструкции обеспечивают более эффективное кодирование за счет следующих оптимизаций.
   - Свертка аргумента – значение часто используемого аргумента зафиксировано инструкцией и не кодируется в потоке инструкций.
   - Укорачивание аргумента – аргумент инструкции кодируется в потоке инструкций более компактно, чем в оригинальной инструкции.
   - Сворачивание последовательности инструкций – одна инструкция кодирует последовательность нескольких опкодов.
8. Некоторые из рассмотренных работ осуществляют свертку аргументов в контексте последовательностей инструкций. Для Java контекстная свертка аргументов была реализована только в работе Saougkos. При этом исполнение специализированных инструкций в этой работе сопряжено с дополнительными накладными расходами.

   Ни в одной из рассмотренных работ не осуществляется контекстного укорачивания аргументов.

9. Упрощение исходного набора инструкций позволяет повысить эффективность сжатия. Ни в одной из рассмотренных работ, посвященных сжатию Java байт-кода, идея упрощения исходного набора инструкций не применялась.

В данной главе были решены следующие задачи диссертации:

1. Изучены существующие алгоритмы понижения избыточности программ с точки зрения их применимости при использовании раздельной инициализации.
2. Исследованы существующие алгоритмы сжатия бинарного кода. Изучена их применимость для сжатия Java байт-кода в закрытой модели.

На основании выводов по результатам обзора сформулируем следующие подзадачи задач диссертации:

1. Разработать и реализовать алгоритмы понижения избыточности Java-программ, применимые при раздельной инициализации.
    а) Разработать алгоритм понижения избыточности для Java, осуществляющий выборочную инициализацию классов и сохраняющий семантику финализации при удалении неиспользуемых, но инициализируемых полей.
    б) Разработать механизм автоматического анализа зависимостей native-методов.
2. Разработать и реализовать алгоритм сжатия Java байт-кода в закрытой модели, применимый для встраиваемых систем с ограниченными ресурсами.
    а) Разработать алгоритм специализации Java байт-кода, осуществляющий контекстную свертку аргументов и не имеющий накладных расходов на декодирование специализированных инструкций.

Применение разработанных алгоритмов при специализации Java-платформы для заданного приложения в закрытой модели позволит сократить аппаратные требования платформы, не нарушая поведения приложения.

Глава 2. Понижение избыточности при выборочной инициализации классов

В главе предлагаются алгоритмы понижения избыточности Java-программ при использовании ранней инициализации классов. Классический подход к удалению недостижимого кода состоит в построении транзитивного замыкания методов, достижимых из точек входа программы, и удаления методов, не попавших в замыкание. При таком подходе возникает вопрос: в какой момент инициализировать классы? До построения замыкания нет информации о том, какие классы используются достижимым кодом. Инициализация классов и удаление статических инициализаторов после построения замыкания может сделать некоторые методы недостижимыми.

Для решения этой проблемы соискателем предложен алгоритм анализа достижимости методов, который расширяет алгоритм RTA выборочной инициализацией используемых классов. В отличие от классических алгоритмов, разработанный алгоритм учитывает объекты, созданные при инициализации классов. Соискателем также предложены алгоритмы анализа удалимости полей и классов, которые учитывают объекты, созданные при инициализации классов.

Результаты, изложенные в данной главе, были опубликованы соискателем в статьях [42; 44; 47; 48]

2.1 Инициализация классов

Согласно спецификации языка Java инициализация класса происходит при первом обращении к классу. Обращением считается:

- выполнение одной из инструкций `new`, `invokestatic`, `getstatic`, `putstatic`;
- рефлексивный доступ к классу, например, с помощью метода `Class.forName`.

Перед инициализацией класса инициализируются его суперклассы. В процессе инициализации исполняется статический инициализатор класса – метод `<clinit>` [80].

При подготовке загрузочного образа некоторые классы могут быть инициализированы заранее. Такая оптимизация сокращает время инициализации приложения на целевом устройстве. Кроме того, ранняя инициализация ускоряет исполнение приложения. В общем случае инструкции `new`, `invokestatic`, `getstatic`, `putstatic` должны проверять инициализирован ли класс, к которому происходит обращение. После того, как класс инициализирован, инструкции, ссылающиеся на данный класс, могут быть заменены быстрыми версиями, в которых такая проверка отсутствует. Если код методов размещается в неизменяемой памяти, данная оптимизация невозможна на этапе исполнения.

Инициализация класса сокращает количество достижимых методов, так как метод `<clinit>` не может быть вызван стандартными инструкциями и после инициализации становится недостижим. В то же время при инициализации могут создаваться объекты, которые увеличивают размер образа. В некоторых случаях ранняя инициализация сокращает размер образа. Это происходит тогда, когда суммарный размер методов, ставших недостижимыми, оказывается больше, чем суммарный размер созданных при инициализации объектов.

В общем случае ранняя инициализация классов нарушает спецификацию языка и может изменить видимое поведение программы. Поэтому не все классы можно инициализировать заранее. Результат инициализации некоторых классов может зависеть от последовательности инициализации, которая в свою очередь определяется последовательностью исполнения программы. Инициализация класса может иметь побочные эффекты, например, инициализатор может осуществлять операции ввода-вывода. Результат работы инициализатора может зависеть от того, вызван он на хосте или на целевом устройстве. Например, инициализатор может использовать метод `java.lang.System.getProperty()`, результат работы которого отличается на разных устройствах. Исполнение инициализатора может приводить к бросанию исключения. Если инициализация произошла в процессе исполнения приложения при первом обращении к классу, брошенное исключение может быть обработано кодом, вызвавшим инициализацию. При ранней инициализации такие исключения нельзя обработать.

Не будем автоматически инициализировать класс, если в его инициализаторе присутствует хотя бы одна из следующих операций.

– Цикл. Инициализатор с циклами может никогда не завершиться.
– Вызов методов. В частности, запрещен вызов конструкторов объектов. Если в коде инициализатора присутствует вызов метода, то для анализа

безопасности инициализации требуется анализ кода вызываемых методов.

– Обращение к статическому полю класса, если этот класс не является инициализируемым классом или одним из его суперклассов. Данная операция может привести к небезопасной инициализации класса, определяющего поле. Если поле принадлежит инициализируемому классу или его суперклассу, то на момент исполнения операции соответствующий класс уже будет инициализирован.

Данная эвристика может оказаться слишком консервативной. Разработчику приложения предоставлена возможность отменить запрет на инициализацию при помощи аннотации `@InitAtBuild`.

В какой момент инициализировать классы? С одной стороны, инициализация классов влияет на достижимость методов. Методы, используемые исключительно для инициализации классов, становятся недостижимы после инициализации и могут быть удалены. С другой стороны, до анализа достижимости неизвестно, какие классы могут быть использованы достижимым кодом. Таким образом, инициализация классов до анализа достижимости может привести к инициализации классов, которые не будут использованы в процессе исполнения. Инициализация таких классов может нарушить поведение программы за счет исполнения мертвого кода. Кроме того, инициализация лишних классов может привести к созданию неиспользуемых, но достижимых объектов. Рисунок 2.1 иллюстрирует циклические зависимости между инициализацией классов и анализом достижимости методов. Для того чтобы разрешить эти зависимости, будем инициализировать классы в процессе построения замыкания.

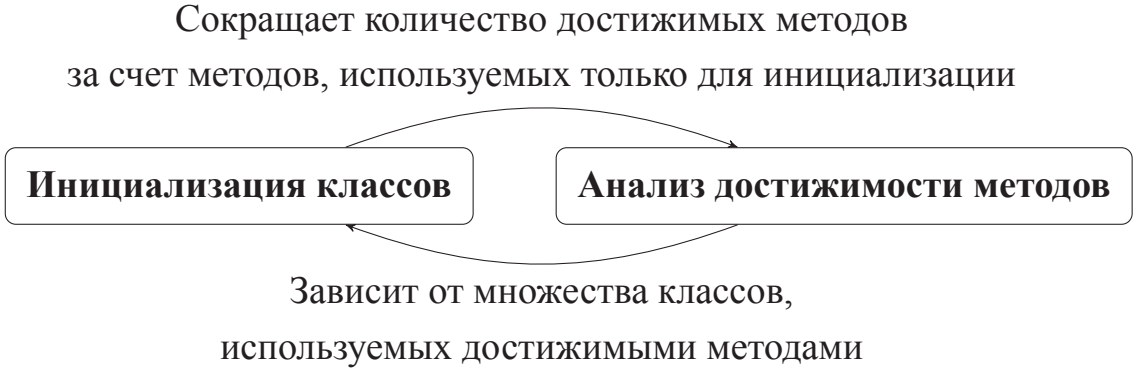


Рисунок 2.1 — Зависимости между инициализацией классов и анализом достижимости методов

Для каждого потенциально инициализируемого класса существует выбор: инициализировать класс в процессе построения замыкания или отложить инициализацию до этапа исполнения. В первом случае метод `<clinit>` должен быть исполнен, во втором – должен быть добавлен в замыкание, его зависимости должны быть проанализированы.

При добавлении метода в замыкание мы анализируем его код и консервативно оцениваем эффекты его исполнения. Например, анализируя инструкции `new` в теле метода, алгоритм RTA вычисляет множество классов, которые могут быть инстанциированы методом. Для методов, которые были исполнены в процессе анализа, вместо консервативной оценки мы можем точно проанализировать эффекты их исполнения. Так, например, для анализа достижимости косвенных вызовов можно точно вычислить множество объектов, созданных в процессе исполнения метода. Анализ созданных объектов обсуждается далее в разделе 2.3.

Если при исполнении инициализатора происходит бросание исключения, создание образа завершается ошибкой. Таким образом, для некоторых корректных с точки зрения спецификации приложений нельзя создать загрузочный образ.

2.2 Финализация объектов

Java – язык с автоматическим управлением памятью. Сборщик мусора (GC) отслеживает достижимость объектов в куче и удаляет недостижимые объекты. Достижимыми считаются объекты, которые могут быть использованы в каком-либо из возможных продолжений исполнения программы. Более строго достижимость объектов определяется как достижимость по ссылкам в объектных полях.

У приложения есть несколько способов отследить удаление объекта. Первый из них – определить метод `finalize` в классе объекта. Метод `finalize` определен в классе `java.lang.Object` и может быть переопределен в подклассах. Финализатором класса будем называть реализацию метода `finalize` в данном классе. Финализатором объекта называется финализатор класса объекта. Финализатор объекта вызывается при уничтожении объекта сборщиком мусора. Класс `Object` содержит пустую реализацию метода `finalize`. Объект с непустым финализатором будем называть финализируемым. Другой способ состоит

в установке на объект специальной ссылки. Специальные ссылки в Java представлены классами `WeakReference`, `SoftReference`, `PhantomReference`. Специальные ссылки не влияют на достижимость объектов и обнуляются, если объект, на который указывала ссылка, был уничтожен сборщиком мусора. Объекты, удаление которых приложение может отследить, будем называть неудалимыми, поскольку удаление таких объектов из образа может нарушить поведение программы.

Спецификация языка позволяет сокращать время жизни объекта в локальных переменных, однако явно запрещает такую оптимизацию для ссылок в куче. Такое ограничение позволяет реализовать идиому finalizer guaridan [97; 98]. Данная идиома решает проблему финализации суперкласса в случае, когда метод `finalize` переопределен в подклассе.

Рассмотрим следующую ситуацию. Класс `Base` определяет метод `finalize`.

```
class Base {
  protected void finalize() {
    /* Finalize Base class instance */
  }
}
```

Класс `Derived` наследуется от класса `Base` и переопределяет метод `finalize`.

```
class Derived extends Base {
  protected void finalize() {
    /* Finalize Derived class instance */
  }
}
```

При уничтожении экземпляра класса `Derived` будет вызван финализатор, объявленный в классе `Derived`. Если реализация финализатора `Derived` не вызывает финализатора класса `Base` явным образом, финализатор класса `Base` не будет вызван.

Идиома finalizer guaridan решает эту проблему следующим образом.

```
class Base {
    private final Object finalizerGuardian = new Object() {
        protected void finalize() {
            /* Finalize Base class instance */
        }
    }
}
```

При уничтожении объекта, реализующего класс `Base` или его подкласс, объект `finalizerGuardian` становится недостижим и также удаляется сборщиком мусора. При его уничтожении вызывается финализатор, который осуществляет финализацию класса `Base`. Финализатор объекта `finalizerGuardian` вызывается, даже если подкласс класса `Base` переопределяет метод `finalize` и не вызывает финализатор суперкласса явным образом.

2.3 Достижимость объектов в куче

Некоторые из объектов, созданных при инициализации классов, останутся достижимыми для приложения и могут быть использованы для косвенных вызовов. Для анализа достижимости методов необходимо проанализировать эффекты инициализации классов и определить, какие объекты останутся достижимыми для приложения. В простом случае можно считать, что это все объекты, пережившие сборку мусора после инициализации классов. Однако последующее удаление неиспользуемых полей может сделать некоторые из этих объектов недостижимыми. Рассмотрим следующий пример:

```
@InitAtBuild
class Main {
  static A a = new A();
  static B b = new B();
  static Main main = new Main();

  public static void main(String ... args) {
    main.foo();
  }

  void foo() {
    System.out.println("Main.foo");
  }
}

class A extends Main {
  void foo() {
    System.out.println("A.foo");
  }
}
```

```
class B extends Main {
  void foo() {
    System.out.println(Main.b.toString());
  }
}
```

На этапе инициализации классов инициализируется класс `Main`. Инициализатор класса `Main` создает экземпляр класса `A`. После сборки мусора класс `A` будет добавлен в множество *InstantiableClasses*, метод `A.foo()` будет включен в замыкание. Однако поле `Main.a`, через которое экземпляр класса `A` был достижим, не используется достижимым кодом. После удаления этого поля экземпляр класса `A`, на который больше нет ссылок, может быть уничтожен сборщиком мусора. Так как достижимый код не создает экземпляров этого класса, объекты этого типа не могут быть аргументами косвенных вызовов. Метод `A.foo()` не может быть вызван.

Проблему подобных зависимостей можно решить, повторяя поиск недостижимых методов после удаления неиспользуемых полей до тех пор, пока на очередной итерации не будет удалено ни одного метода. В худшем случае на каждой итерации из множества *InstantiableClasses* будет удаляться один класс. Общее количество итераций такого алгоритма ограничивается количеством анализируемых классов.

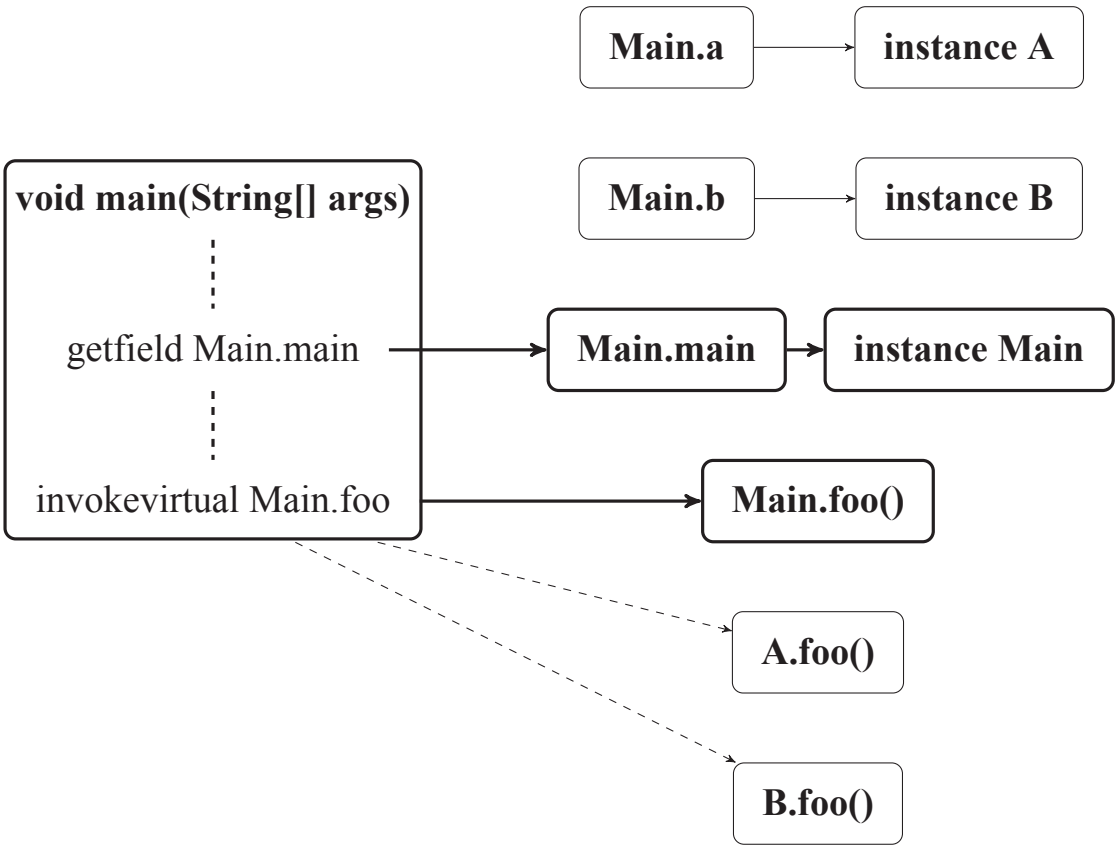
Однако итеративный подход не решает проблемы циклических зависимостей. Такая зависимость показана на примере класса `B` и поля `Main.b`. Поле `Main.b` не будет удалено, так как к нему существует обращение из метода `B.foo()`. Виртуальный метод `B.foo()`, в свою очередь, включен в замыкание только потому, что объект класса `A` достижим через поле `Main.b`.

Использование сборщика мусора для вычисления множества объектов достижимых для приложения неточно, так как сборщик мусора проходит по всем ссылочным полям, не учитывая, используются ли они достижимым кодом или нет. Для точного анализа необходимо построить множество читаемых полей и множество объектов, достижимых для приложения.

Читаемыми будем называть поля, для которых в коде достижимых методов есть инструкции чтения (`getstatic`, `getfield`). Чтобы вычислить множество объектов, достижимых для приложения, обойдем граф объектов начиная с корневых ссылок. При обходе специальные ссылки обрабатываются наравне с обычными ссылками. Не будем посещать объекты по ссылкам в нечитаемых

полях. Все посещенные объекты будем считать достижимыми для приложения. Кроме того, все финализируемые объекты достижимы для приложения через ссылку `this` при выполнении метода `finalize`.

Рисунок 2.2 иллюстрирует анализ достижимости методов, использующий описанный анализ достижимости объектов, на вышеприведенном примере.



Жирным выделены методы, поля и объекты, достижимые для приложения. Ребра, выделенные жирным, обозначают отношения достижимости между методами, полями и объектами. Пунктирные ребра обозначают отношения достижимости между методами согласно анализу CHA.

Рисунок 2.2 — Пример анализа достижимости объектов в куче

2.4 Описание дополнительных зависимостей

Для понижения избыточности необходимо анализировать зависимости между методами, полями, классами и объектами. В большинстве случаев необходимые зависимости могут быть извлечены из байт-кода. Однако анализ некоторых зависимостей из байт-кода либо сложен, либо вовсе невозможен. Так, например,

рефлексия позволяет обращаться к методам, полям и классам, используя динамический поиск по имени в процессе исполнения. Зависимости, возникающие при использовани рефлексии, сложно анализировать статически. Другой пример – native-методы. Такие методы не имеют Java байт-кода, их реализация написана на другом языке, чаще всего C/C++. Код этих методов может обращаться к Java-методам, полям и классам, используя специальный интерфейс, предоставляемый виртуальной машиной. Примером такого интерфейса может служить JNI [99].

Для того чтобы корректно обрабатывать такие ситуации, необходимо предоставить возможность вручную описывать дополнительные зависимости. В простом случае для корректной работы алгоритма можно предоставить возможность безусловно запретить удаление выбранных методов, полей и классов. Однако зачастую метод, поле или класс нельзя удалять потому, что существуют использующие их методы. Если эти методы окажутся недостижимы и будут удалены, их зависимости также могут быть удалены. Для предложенных алгоритмов реализован механизм описания дополнительных зависимостей отдельных методов. Для каждого метода могут быть указаны используемые поля, методы и классы. Такие зависимости анализируются наряду с зависимостями из байт-кода.

Дополнительные зависимости описываются с помощью конфигурационных файлов. Конфигурационные файлы состоят из записей, которые выглядят следующим образом:

```
Method <имя метода>
  <список зависимостей>
  ...
EndMethod
```

Такая запись перечисляет дополнительные зависимости метода `<имя метода>`. Для метода могут быть указаны следующие зависимости:

- `MethodCall <имя метода>` – методы, вызываемые из данного;
- `FieldRead <имя поля>` – поля, читаемые данным методом;
- `FieldWrite <имя поля>` – поля, записываемые данным методом;
- `ClassNew <имя класса>` – классы, экземпляры которых создаются в методе;
- `ClassAccess <имя класса>` – классы, на которые данный метод ссылается.

Для ручного описания зависимостей предоставлена возможность использовать Java-аннотации, которые впоследствии конвертируются в конфигурационный файл. Пример описания дополнительных зависимостей с помощью аннотаций приведен в листинге 2.1. Листинг 2.2 демонстрирует конфигурационный файл, сгенерированный из данных аннотаций.

Листинг 2.1 Пример описания дополнительных зависимостей

```
@Local(ClassAccess = {"some.class.A"})
Class getClassA() {
  return Class.forName("some.class.A");
}
@Local(ClassAccess = {"some.class.B"})
Class getClassA() {
  return Class.forName("some.class.B");
}
Class foo() {
  return getClassA();
}
```

Листинг 2.2 Зависимости из листинга 2.1 после конвертации в синтаксис конфигурационных файлов

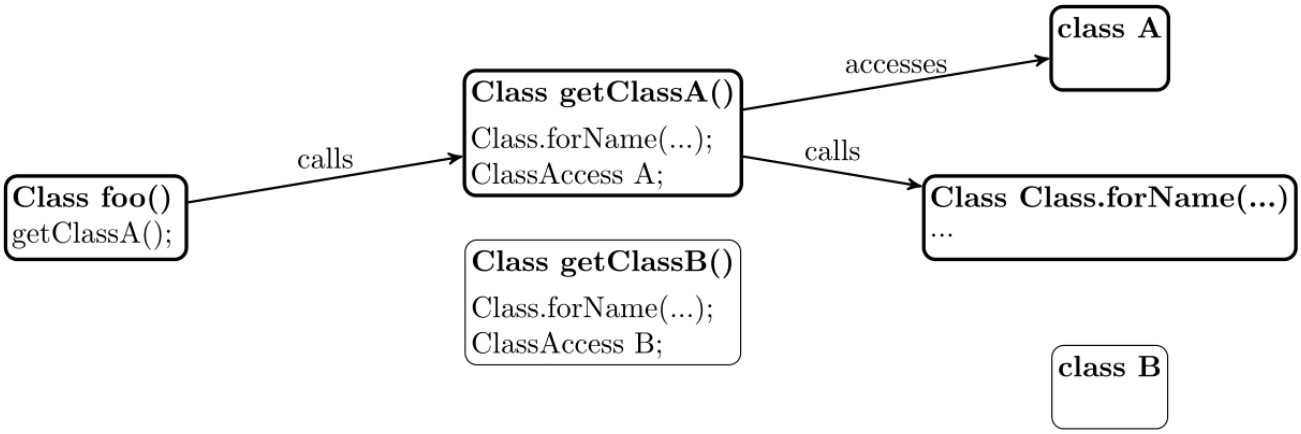
```
Method getClassA
  ClassAccess some.class.A
EndMethod

Method getClassB
  ClassAccess some.class.B
EndMethod
```

При анализе зависимостей указанного метода, поля, методы и классы, перечисленные в списке зависимостей, будут учтены наряду с зависимостями, извлеченными из байт-кода. Пример анализа метода `foo()` из листинга 2.1 приведен на рисунке 2.3. В данном примере дополнительные зависимости метода `getClassB()` не используются, поскольку сам метод оказывается недостижим.

Предложенные алгоритмы понижения избыточности не анализируют зависимости, возникающие при использовании рефлексии, автоматически. Такие зависимости должны быть описаны вручную с помощью конфигурационных файлов или Java-аннотаций. Механизм автоматического анализа зависимостей

native-методов, описанный в разделе 2.10, порождает описание зависимостей в формате конфигурационных файлов.



Жирным выделены достижимые методы и классы, ребра между методами и классами обозначают зависимости.

Рисунок 2.3 — Пример анализа дополнительных зависимостей из листинга 2.1

2.5 Анализ достижимости методов

Опишем алгорим анализа достижимости методов, осуществляющий выборочную инициализацию используемых классов. Для этого дополним алгоритм RTA выборочной инициализацией классов. Входными данными алгоритма будут являться:

– иерархия загруженных классов, включающая в себя байт-код методов данных классов;
– набор точек входа приложения;
– конфигурационные файлы.

На выходе алгоритм порождает преинициализированное состояние памяти виртуальной машины и множества:

– *ReachableMethods* – множество достижимых методов и его рабочее подмножество;
– *IndirectInvocations* – множество методов, используемых для косвенных вызовов;
– *InstantiableClasses* – множество классов, экземпляры которых доступны для приложения;

– *InitializableClasses* – множество классов, потенциально инициализируемых достижимым кодом;
– *ReadFields* – множество читаемых полей.

В процессе работы алгоритм поддерживает рабочие подмножества соответствующих множеств: *NewReachableMethods*, *NewIndirectInvocations*, *NewInstantiableClasses*, *NewInitializableClasses*. При добавлении нового элемента в множество, у которого есть рабочее подмножество, будем добавлять элемент в множество и соответствующее рабочее подмножество.

В начале работы алгоритма точки входа приложения добавляются в множество *ReachableMethods*. Пока хотя бы одно из множеств *NewReachableMethods* и *NewInitializableClasses* не пусто, выполняются следующие действия.

1. Для каждого метода из множества *NewReachableMethods* анализируем его зависимости. Вначале просматриваем код метода:
    а) методы, вызываемые инструкциями прямого вызова, добавляем в множество *ReachableMethods*;
    б) методы, используемые инструкциями `invokevirtual` и `invokeinterface`, запоминаем в множестве *IndirectInvocations*;
    в) классы, используемые для создания объектов с помощью инструкции `new`, запоминаем в множестве *InstantiableClasses*;
    г) классы, на которые ссылаются инструкции `new`, `invokestatic`, `getstatic`, `putstatic`, добавляем в множество *InitializableClasses*;
    д) поля, используемые инструкциями `getstatic`, `getfield`, добавляем в множество *ReadFields*.

    Затем обрабатываем зависимости метода, описанные в конфигурационных файлах:
    а) методы, указанные в зависимостях `MethodCall`, добавляем в множество *ReachableMethods*;
    б) классы, указанные в зависимостях `ClassNew`, запоминаем в множестве *InstantiableClasses*;
    в) классы, указанные в зависимостях `ClassNew` и `ClassAccess`, добавляем в множество *InitializableClasses*;
    г) поля, указанные в зависимостях `FieldRead`, добавляем в множество *ReadFields*.

Проанализированные методы удаляем из множества *NewReachableMethods*.

2. Обрабатываем классы из множества *NewInitializableClasses*. Инициализируем классы, которые могут быть инициализированы согласно эвристике из раздела 2.1, методы `<clinit>` остальных классов добавляем в множество *ReachableMethods*. Обработанные классы удаляем из множества *NewInitializableClasses*.
3. Выполняем сборку мусора, финализируем удаленные объекты.
4. Обходим объекты, достижимые для приложения. Для этого:
    а) обходим все финализируемые объекты,
    б) обходим граф объектов, достижимых из корневых ссылок по обычным и специальным ссылкам. По ссылке, содержащейся в поле, переходим, только если поле принадлежит множеству *ReadFields*.

   Классы посещенных объектов добавляем в множество *InstantiableClasses*.
5. В множество *ReachableMethods* добавляем методы, доступные через косвенные вызовы:
    а) добавляем методы классов из множества *InstantiableClasses*, достижимые через косвенные вызовы методов из множества *NewIndirectInvocations*;
    б) добавляем методы классов из множества *NewInstantiableClasses*, достижимые через косвенные вызовы методов из множества *IndirectInvocations*.
6. В множество *ReachableMethods* добавляем финализаторы классов из *NewInstantiableClasses*.
7. Обнуляем множества *NewInstantiableClasses*, *NewIndirectInvocations*.

Все множества, вычисляемые алгоритмом, растут монотонно. Количество элементов в множествах *ReachableMethods* и *InitializableClasses* ограниченно общим количеством анализируемых методов и классов соответственно. Таким образом гарантируется завершение итеративного алгоритма.

После завершения алгоритма методы, не принадлежащие объединению множеств *IndirectInvocations* и *ReachableMethods*, можно удалить, не нарушив поведения программы. Методы, принадлежащие разности множеств *IndirectInvocations* и *ReachableMethods*, являются эффективно абстрактными. Тела таких методов можно удалить.

Формальное описание алгоритма приведено в Приложении А.2

2.6 Анализ удалимости полей

После удаления недостижимых методов удаляются неиспользуемые поля. Поля, которые можно удалить, не нарушив поведения программы, назовем удалимыми. Определим следующие критерии неудалимости поля.

- В достижимом коде есть операции чтения поля. Заметим, что операции записи не являются достаточным критерием неудалимости. Нередко поля, инициализированные в конструкторе или статическом инициализаторе, не используются после инициализации.
- Существует доступ к полю, который может привести к инициализации класса.
- Поле неудалимо, если экземпляр поля может содержать ссылку, которая препятствует уничтожению неудалимого объекта. Удаление поля может нарушить достижимость объектов, достижимых через ссылки в поле.

Опишем алгоритм вычисления множества неудалимых полей согласно данным критериям. Входными данными алгоритма являются:

- иерархия загруженных классов, включающая в себя описание полей и байт-код методов данных классов;
- множество *ReadFields*;
- конфигурационные файлы;
- преинициализированное состояние памяти виртуальной машины.

На выходе алгоритм порождает множество *UnremovableFields*, которое содержит все поля, удаление которых может нарушить видимое поведение программы.

1. Инициализируем множество *UnremovableFields* полями из множества *ReadFields*, поскольку все читаемые поля неудалимы.
2. Добавим поля, доступ к которым может привести к инициализации класса. В Java байт-коде существуют две инструкции доступа к полям, которые обладают побочным эффектом инициализации класса: `getstatic` и `putstatic`. Все поля, используемые инструкциями

`getstatic`, были включены в множество *UnremovableFields* как читаемые поля. Остается проанализировать инструкции `putstatic` в коде достижимых методов.

В некоторых случаях можно гарантировать, что класс, содержащий поле, будет инициализирован на момент исполнения инструкции. Например:

- если класс, содержащий поле, инициализирован на момент анализа (то есть был инициализирован в процессе построения замыкания);
- если класс, содержащий анализируемый метод, является подклассом класса, содержащего поле.

В противном случае включим поле, используемое инструкцией `putstatic`, в множество *UnremovableFields*.

3. Добавим поля, которые могут содержать ссылки, препятствующие уничтожению неудалимых объектов. Будем считать, что объектное поле может содержать ненулевую ссылку в процессе исполнения, если поле было инициализировано ненулевой ссылкой в процессе анализа, либо в достижимых методах существуют операции записи поля. Вычислим множество *WrittenObjectFields* – множество объектных полей, для которых существуют операции записи в достижимых методах.
   - Проанализируем байт-код достижимых методов. Добавим объектные поля, используемые инструкциями `putstatic`, `putfield`.
   - Проанализируем зависимости достижимых методов. Добавим объектные поля, указанные в качестве `FieldWrite` зависимостей.

   В простом случае можно консервативно предположить, что любое поле, которое может содержать ненулевую ссылку, препятствует уничтожению неудалимых объектов. Более точный алгоритм, вычисляющий множество полей, которые могут содержать ссылки, препятствующие уничтожению неудалимых объектов, описан в следующем разделе.

2.7 Анализ достижимости неудалимых объектов

Будем считать, что поле нельзя удалять, если через ссылку в поле может быть достижим неудалимый объект. Ранее мы выделили два типа неудалимых объектов: финализируемые объекты и объекты, доступные посредством специальных ссылок. Для проверки достижимости финализируемых объектов применим анализ типов.

- Статическое поле может ссылаться на экземпляр класса A, если выполнено хотя бы одно из двух условий:
  - в процессе инициализации полю была присвоена ссылка на экземпляр класса A;
  - в коде достижимых методов есть операции записи в поле, и класс A принадлежит множеству подклассов объявленного типа поля.
- Нестатическое поле может ссылаться на экземпляр класса A, если выполнено хотя бы одно из двух условий:
  - в процессе инициализации экземпляру поля была присвоена ссылка на экземпляр класса A;
  - в коде достижимых методов есть операции записи в поле, класс A принадлежит множеству подклассов объявленного типа поля, и содержащий поле класс принадлежит множеству *InstantiableClasses*.
- Экземпляр класса A может ссылаться на экземпляр класса B, если класс B принадлежит множеству *InstantiableClasses*, и у класса A существует нестатическое поле, которое может ссылаться на экземпляр класса B.

Сложнее доказать недостижимость объектов, доступных посредством специальных ссылок. В Java-коде специальные ссылки представлены экземплярами классов `java.lang.WeakReference`, `java.lang.SoftReference`, `java.lang.PhantomReference`. В исходном коде тип специальной ссылки параметризован классом объекта, на который указывает ссылка. При компиляции в Java байт-код эта информация теряется из-за стирания типов. В простом случае можно считать, что если приложение создает экземпляры классов специальных ссылок, то они могут ссылаться на объекты любых инстанциируемых классов.

Опишем алгоритм, вычисляющий *RefToFinalizableFields* – множество полей, которые могут содержать ссылки, препятствующие уничтожению неудалимых объектов. Все поля из данного множества являются неудалимыми и должны быть включены в множество *UnremovableFields* по завершению алгоритма. Входными данными алгоритма являются:

– иерархия загруженных классов, включающая в себя описание полей и байт-код методов данных классов;
– множества *InstantiableClasses*, *ReadFields*, *WrittenObjectFields*;
– преинициализированное состояние памяти виртуальной машины.

Будем считать, что приложение использует специальные ссылки, если множество *InstantiableClasses* содержит какой-либо из классов специальных ссылок. Если приложение использует специальные ссылки, то множество *RefToFinalizableFields* должно быть дополнено объектными полями, которые могут содержать ненулевые ссылки в процессе исполнения. Для этого включим в множество *RefToFinalizableFields* поля, содержащие ненулевые ссылки, и поля из множества *WrittenObjectFields*.

Если приложение не использует специальные ссылки, построим множество полей, через ссылки в которых могут быть достижимы финализируемые объекты. Для каждого класса предварительно вычислим два множества.

– *MayReferTo(class)* – множество полей, объявленный тип которых совместим с классом *class*. То есть множество полей, которые могут содержать ссылки на экземпляры класса *class*. Для вычисления переберем все поля, каждое поле добавим в множества *MayReferTo(class)* подклассов объявленного типа поля.
– *ReferTo(class)* – множество полей, которые содержат ссылки на экземпляры класса *class*. Для вычисления обойдем граф объектов, достижимых из корневых ссылок. При посещении ссылки в поле добавим это поле в множество *ReferTo(class)*, где *class* – класс объекта, на который указывает ссылка.

В процессе работы алгоритма будем вычислять следующие множества:

– *RefToFinalizableFields*, *NewRefToFinalizableFields* – множество полей, которые могут прямо или косвенно ссылаться на финализируемые классы, и его рабочее подмножество;

– *RefToFinalizableClasses*, *NewRefToFinalizableClasses* – множество классов, экземпляры которых могут прямо или косвенно ссылаться на финализируемые классы, и его рабочее подмножество.

При добавлении нового элемента в множество *RefToFinalizableFields* или *RefToFinalizableClasses* будем добавлять элемент в множество и соответствующее рабочее подмножество.

Вначале инициализируем множество *RefToFinalizableClasses* инстанциируемыми классами, у которых определен метод `finalize`. Пока множество *NewRefToFinalizableClasses* не пусто, выполняем следующие действия.

1. Для каждого класса из множества *NewRefToFinalizableClasses* поля, которые могут ссылаться на экземпляр класса, добавляются в множество *RefToFinalizableFields*. Это поля, принадлежащие множеству *MayReferTo(class)* $\cap$ (*ReferTo(class)* $\cup$ *WrittenObjectFields*).
   Обработанные классы удаляются из множества *NewRefToFinalizableClasses*.
2. Для каждого нестатического поля из множества *NewRefToFinalizableFields* классы, экземпляры которых могут содержать поле, добавляются в множество *RefToFinalizableClasses*. Это подклассы класса, в котором определено поле, принадлежащие множеству *InstantiableClasses*
   Обработанные поля удаляются из множества *NewRefToFinalizableFields*.

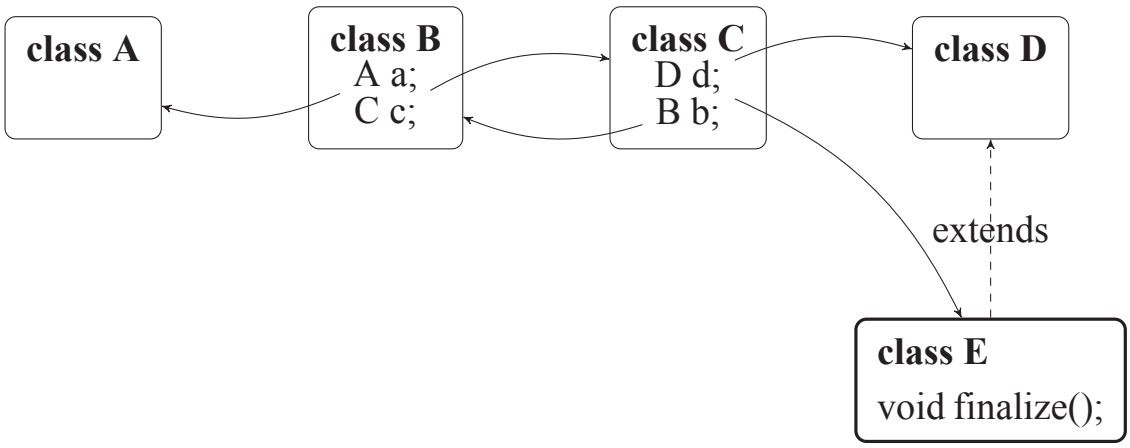
Рисунок 2.4 демонстрирует пример анализа типов.

Все множества, вычисляемые алгоритмом, растут монотонно. Количество элементов в множестве *RefToFinalizableClasses* ограничено общим количеством анализируемых классов. Таким образом гарантируется завершение итеративного алгоритма.

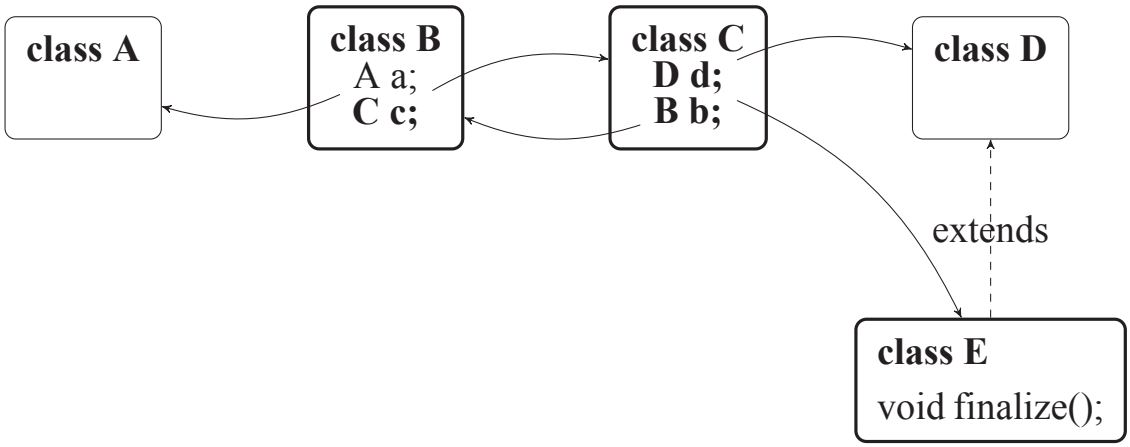
Формальное описание алгоритма приведено в Приложении А.2.

2.8 Межпроцедурный анализ указателей

Алгоритм анализа достижимости неудалимых объектов, описанный в разделе 2.7, не позволяет определить, на какие объекты могут указывать специальные ссылки. Если приложение инстанциирует какой-либо из классов специальных ссылок, то предполагается, что специальные ссылки могут указывать на объекты



(а) Начало алгоритма



(б) Конец алгоритма

Ребро между полем и классом обозначает, что поле может ссылаться на экземпляр класса. Классы, выделенные жирным, принадлежат множеству *RefToFinalizableClasses*. Поля, выделенные жирным, принадлежат множеству *RefToFinalizableFields*, и не могут быть удалены.

Рисунок 2.4 — Пример анализа достижимости неудалимых объектов

любых инстанциируемых типов. Таким образом, в присутствии специальных ссылок теряется возможность удаления неиспользуемых, но инициализированных ссылочных полей. Через такие поля могут оказаться достижимы большие подграфы неиспользуемых объектов. Невозможность удалить такие объекты негативно сказывается на размере образа.

Опишем алгоритм, который позволит более точно анализировать достижимость объектов, доступных посредством специальных ссылок. Входными данными алгоритма являются:

– иерархия загруженных классов, включающая в себя описание полей и байт-код методов данных классов;
– множество *InstantiableClasses*;

– конфигурационные файлы;
– преинициализированное состояние памяти виртуальной машины.

На выходе алгоритм порождает множество *ReferencedBySpecialRefs*, которое содержит все типы объектов, на которые могут быть установлены специальные ссылки.

Для того чтобы определить, на какие объекты могут указывать специальные ссылки, необходимо знать типы объектов, передаваемых в конструкторы классов специальных ссылок. Как было отмечено ранее, данная информация теряется при компиляции из-за стирания типов. Для вычисления множества возможных типов воспользуемся межпроцедурным анализом указателей [44].

В процессе работы алгоритма будем строить граф, который моделирует поток данных между операциями создания объектов и операциями создания специальных ссылок. Объекты в Java могут передаваться через аргументы и возвращаемые значения вызовов, через ссылки в полях и массивах. Таким образом, граф, моделирующий поток данных, будет содержать следующие узлы:

– *NewObject(class)* – узлы создания объектов (для моделирования разных точек создания объектов одного типа используется один модельный объект),
– *NewSpecialRef* – узел создания специальных ссылок (для удаления неиспользуемых полей достаточно использовать один модельный объект для всех точек создания специальных ссылок),
– промежуточные узлы:
    – *MethodArg(method, i)* – аргументы методов,
    – *MethodReturnValue(method)* – возвращаемые значения методов,
    – *Field(field)* – поля объектов (для моделирования разных объектов одного типа используется один модельный объект),
    – *Array* – элементы массивов (для моделирования массивов используется один модельный объект),
    – узлы слияния контекстов исполнения (phi-узлы).

Ребра в графе отражают операции присваивания в программе. Направленное ребро из узла A в B означает, что существует такая последовательность присваиваний, в результате исполнения которой значение узла A присваивается узлу B или используется узлом B.

Для построения графа потока данных с помощью абстрактной интерпретации проанализируем операции со ссылочными типами в коде достижимых

методов. Абстрактный интерпретатор оперирует узлами графа вместо значений времени исполнения. Контекст интерпретатора представляет собой состояние стека и локальных переменных. Операции присвоения в пределах контекста интерпретируются присваиванием значений в контексте. При слиянии контекстов исполнения для каждого элемента контекста (слота на стеке или локальной переменной ссылочного типа) выполняются следующие действия.

1. Создается phi-узел, который является значением элемента в объединенном контексте.
2. Создаются направленные ребра к phi-узлу от узлов, описывающих значения элемента в объединяемых контекстах.

Нелокальные присваивания, такие как записи и чтения полей, передача объектов через аргументы и возвращаемые значения, отражаются на модельном графе.

– Операция создания объекта моделируется с помощью узла *NewObject(class)*.
– Значение аргумента анализируемого метода моделируется с помощью узла *MethodArg(method, i)*.
– Результат чтения объектного поля моделируется с помощью узла, описывающего данное поле *Field(field)*.
– Результат чтения объекта из массива моделируется с помощью узла *Array*.
– Запись объектного поля моделируется с помощью направленного ребра между узлом, описывающим сохраняемое значение, и узлом *Field(field)*.
– Создание специальной ссылки моделируется с помощью направленного ребра между узлом, описывающим значение, на которое устанавливается ссылка, и узлом *NewSpecialRef*.

При анализе вызова необходимо знать, с какими методами может быть связан данный вызов. Для прямых вызовов связываемый метод известен статически. Для виртуальных и интерфейсных вызовов используем оценку из алгоритма достижимости методов, предложенного в разделе 2.5, а именно будем считать, что косвенный вызов может быть связан с реализацией метода в инстанциируемых классах совместимого типа.

– Для каждого аргумента объектного типа создается направленное ребро между узлом, описывающим значение передаваемого аргумента, и узлами *MethodArg(method, i)* всех связываемых методов.

– Возвращаемое значение вызова моделируется с помощью phi-узла, имеющего входящие ребра из узлов *MethodReturnValue(method)* всех связываемых методов.

Рисунок 2.5 демонстрирует пример модельного графа для метода `foo`, приведенного в листинге 2.3.

Листинг 2.3 Метод foo

```
class A {
    Object getObject(Object input) {
        return new Integer(input.hashCode());
    }
}
class B extends A {
    static Throwable t;
    Object getObject(Object input) {
        return t;
    }
}
class C extends A {
    Object getObject(Object input) {
        return input;
    }
}

void foo(A a) {
  B.t = new Throwable();
  new WeakReference(a.getObject(new A()));
}
```

После построения графа вычислим множество *ReferencedBySpecialRefs*. Это множество составляют типы в узлах создания объектов, достижимых по обращениям ребер из узла, моделирующего операции создания специальных ссылок. Для вычисления этого множества обойдем модельный граф по обращениям ребер. Начнем обход с узла, моделирующего операции создания специальных ссылок. При посещении узлов создания объектов добавим соответствующие типы в *ReferencedBySpecialRefs*.

Информация о типах объектов, на которые могут быть установлены специальные ссылки, позволяет вычислить множество полей, удаление которых может привести к удалению объектов, достижимых посредством специальных ссылок.

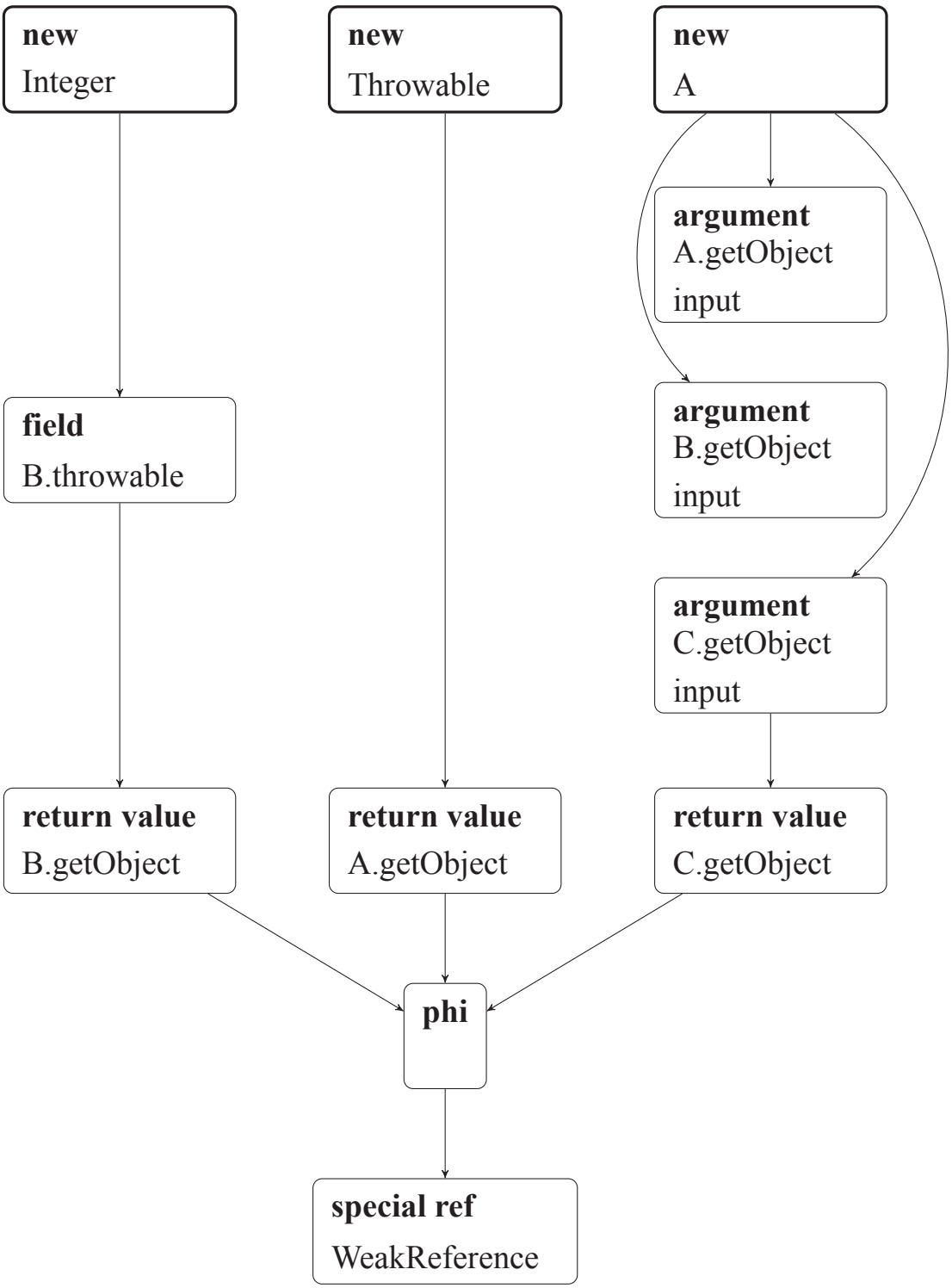


Рисунок 2.5 — Модельный граф для метода foo, приведенного в листинге 2.3

Для этого необходимо использовать анализ типов, аналогичный описанному в Разделе 2.7.

Отметим, что межпроцедурный анализ указателей позволяет вычислить множество возможных типов не только для аргументов конструкторов специальных ссылок, но и для любой другой точки программы. В частности, анализ

указателей может быть использован для вычисления множества типов объектов-получателей косвенных вызовов. Эта информация может быть использована для реализации более точного алгоритма анализа достижимости методов. Оптимистичный алгоритм анализа достижимости методов, использующий межпроцедурный анализ указателей, описан соискателем в статье [42].

2.9 Анализ удалимости классов

После удаления недостижимых методов и полей вычислим множество *UnremovableClasses* – множество неудалимых классов.

- Классы, на которые ссылаются достижимые методы, удалять нельзя. Проанализируем код достижимых методов и включим классы, на которые ссылаются инструкции `new`, `invokespecial`, `invokestatic`, `invokevirtual`, `invokeinterface`, `getstatic`, `putstatic`, `getfield`, `putfield`, `ldc`, `instanceof`, `checkcast`, в множество *UnremovableClasses*. Также добавим в множество классы, указанные в качестве зависимостей `ClassAccess` для достижимых методов.
- Классы исключений из таблиц обработчиков исключений достижимых методов неудалимы. Добавим их в множество *UnremovableClasses*.
- Класс нельзя удалять, если для приложения достижим экземпляр класса. Такие классы содержатся в множестве *InstantiableClasses*, поэтому дополним множество *UnremovableClasses* классами из множества *InstantiableClasses*.
- Класс нельзя удалять, если он содержит неудалимое поле. В коде достижимых методов может не оказаться использований неудалимого поля (и, соответственно, инструкций, ссылающихся на класс), если данное поле было инициализировано при построении замыкания. Дополним множество *UnremovableClasses* классами, содержащими неудалимые поля.

Классы, не принадлежащие множеству *UnremovableClasses*, можно удалить, не нарушив поведения программы.

Формальное описание алгоритма приведено в Приложении A.2

2.10 Автоматический анализ нативных зависимостей

Системные классы виртуальной машины, для которой реализованы предложенные алгоритмы, содержат большое количество native-методов. Для корректной работы алгоритмов понижения избыточности необходимо учитывать зависимости таких методов. Нередки ситуации, когда некоторые поля используются только native-методами. Например, поля, хранящие дескрипторы нативных ресурсов, могут не иметь использований в Java-коде. Удаление таких полей некорректно, если достижим хотя бы один native-метод, использующий поле. В разделе 2.4 был предложен механизм для ручного описания таких зависимостей. При большом количестве native-методов ручное описание зависимостей неэффективно и чревато ошибками. Опишем метод автоматического анализа зависимостей native-методов.

Нативный интерфейс виртуальной машины, для которой реализованы описанные алгоритмы, не позволяет вызывать Java-методы из native-методов, поэтому дальнейшее описание касается только анализа использования полей. Аналогичный подход может быть использован для анализа достижимости методов.

Для каждого native-метода необходимо знать, какие Java-поля используются реализацией этого метода. Для доступа к Java-сущностям интерфейс JNI использует динамический поиск по имени, что делает статический анализ его использования затруднительным. Однако для заданного набора классов можно сгенерировать статический нативный интерфейс, который будет моделировать Java-сущности с помощью сущностей C++. Анализировать использования такого интерфейса будет значительно проще.

Так, иерархию Java-классов можно продублировать иерархией C++ классов, сгенерировав для каждого Java-класса соответствующий ему C++ класс. Для каждого Java-поля генерируется accessor-метод в соответствующем C++ классе. Accessor-метод возвращает ссылку на поле и используется как для записи, так и для чтения Java-поля. При наличии возможности вызова Java-методов из нативного кода, Java-методы могут быть смоделированы аналогичным образом с помощью C++ методов.

В листинге 2.4 приведен фрагмент интерфейса, сгенерированного для класса `java.lang.String`, и пример его использования.

Листинг 2.4 Метод foo

```
// Java
package java.lang;
class String {
  char value[];
  int offset;
  int count;
  ...
}
```

```
// C++
struct jchar_array {
  jint length();
  jchar* elements();
};

struct Java_java_lang_String :
  public Java_java_lang_Object {
  jchar_array*& value();
  jint& offset();
  jint& count();
};

Java_java_lang_String* java_string;
char* char_base =
  java_string->value()->elements();
char_base += java_string->offset();
int length = java_string->count();
std::string str(char_base, length);
```

При использовании такого статического интерфейса анализ Java-зависимостей native-методов сводится к анализу достижимости C++ методов. Для каждого C++ метода, являющегося реализацией native-метода, необходимо вычислить набор достижимых из него методов-аксессоров. Зависимостями данного метода являются Java-поля, соответствующие найденным аксессорам.

Анализатор нативного кода реализован в виде самостоятельного приложения, которое запускается в процессе сборки среды исполнения и системных классов. Анализатор принимает на вход набор исходных C++ файлов вместе с флагами компиляции. На выходе анализатор генерирует конфигурационный

файл с описаниями зависимостей. Для разбора C++ кода используется библиотека libclang, которая предоставляет доступ к AST исходного кода [100]. Для каждого метода, являющегося реализацией native-метода, с помощью обхода в глубину вычисляется множество достижимых из него методов. Если множество методов-аксессоров не пусто, то в конфигурационный файл добавляется запись, объявляющая Java-поля, соответствующие достижимым аксессорам, зависимостями анализируемого метода.

Для анализа виртуальных вызовов используется алгоритм CHA, то есть для данного виртуального вызова достижимыми считаются все возможные реализации виртуального метода [49]. Анализ косвенных вызовов через указатели на функции не поддерживается. Если при вычислении множества достижимых методов встретился вызов через указатель на функцию, генерируется предупреждение о том, что зависимости анализируемого native-метода должны быть описаны вручную.

2.11 Выводы

В данной главе была рассмотрена задача понижения избыточности Java-программ при использовании ранней инициализации классов.

1. Предложено расширение алгоритма RTA выборочной инициализацией классов. В дополнение к множеству классов, потенциально инстанциируемых достижимым кодом, предложенный алгоритм вычисляет множество потенциально инициализируемых классов. Некоторые классы из данного множества инициализируются в процессе анализа. Для выбора классов, которые могут быть инициализированы при подготовке образа, предложена консервативная эвристика. Инициализаторы классов, которые не были инициализированы во время анализа, добавляются в множество достижимых методов.

   Инициализация классов может порождать объекты, которые влияют на разрешение косвенных вызовов. Для вычисления множества объектов, которые могут быть использованы в качестве объектов-получателей косвенных вызовов, используется подход, аналогичный алгоритму RMA.

2. Предложен алгоритм анализа удалимости полей, позволяющий удалять инициализированные, но неиспользуемые поля. Предложенный алгоритм сохраняет семантику финализации и не удаляет поля, если это может нарушить достижимость объектов, финализация которых видна приложению.
3. Предложен алгоритм анализа удалимости классов, учитывающий объекты, созданные во время инициализации классов.
4. Предложен механизм гранулярного описания дополнительных зависимостей, который позволяет удалять зависимости, если использующий их код недостижим. Для вычисления зависимостей native-методов предложен метод автоматического анализа.

В данной главе решена 3-я задача диссертации: «Разработать алгоритмы понижения избыточности Java-программ, применимые при раздельной инициализации». Реализация предложенных алгоритмов и методов позволит более точно анализировать зависимости приложения при специализации Java-платформы для заданного приложения в закрытой модели. Более точный анализ зависимостей позволит сократить размер приложения и аппаратные требования платформы, не нарушая поведения приложения.

Глава 3. Специализация набора инструкций

В главе предлагается алгоритм сжатия Java байт-кода, который порождает компактное исполняемое представление путем специализации набора инструкций для заданного приложения. Специализированный набор инструкций сокращает суммарный размер программы и интерпретатора, необходимого для ее исполнения, за счет кодирования часто встречающихся шаблонов последовательностей инструкций новыми инструкциями. В качестве шаблонов используются последовательности инструкций, параметризованные значениями некоторых из аргументов. По существу, такая оптимизация является применением словарного сжатия к коду программы.

Предложенный алгоритм осуществляет свертку и укорачивание аргументов в процессе выбора словаря шаблонов, что позволяет добиться более эффективного кодирования за счет применения этих оптимизаций в контексте других инструкций. Кроме того, предложенный алгоритм предварительно упрощает исходный набор инструкций, удаляя из него инструкции, которые могут быть представлены с помощью других инструкций. Такое преобразование освобождает дополнительные опкоды для специализированных инструкций.

Результаты, изложенные в данной главе, были опубликованы соискателем в статье [46].

3.1 Специализация набора инструкций

Как было отмечено в разделе 1.11, специализированные инструкции обеспечивают более эффективное кодирование за счет следующих оптимизаций:

- Свертка аргумента. Значение часто используемого аргумента зафиксировано инструкцией и не кодируется в потоке инструкций.
- Укорачивание аргумента. Аргумент инструкции кодируется в потоке инструкций более компактно, чем в оригинальной инструкции.
- Сворачивание последовательности инструкций. Одна инструкция кодирует последовательность нескольких опкодов.

При добавлении специализированных инструкций будем расширять интерпретатор поддержкой добавленных инструкций. Для этого реализация интерпретатора должна быть дополнена реализацией новых инструкций в машинном коде. Таким образом, специализированные инструкции сокращают размер байт-кода программы, но в то же время увеличивают размер интерпретатора. В предельном случае вся программа может быть закодирована одной специализированной инструкцией, реализация которой – реализация программы в машинном коде. Однако в большинстве случаев размер программы в машинном коде существенно больше размера исходного байт-кода. При выборе набора инструкций необходимо учитывать увеличение размера интерпретатора.

В закрытой модели, когда весь исполняемый на устройстве код известен заранее, не требуется совместимости со стандартным набором инструкций Java байт-кода. В таком случае инструкции, не используемые заданным приложением, могут быть удалены. Удаление неиспользуемых инструкций освобождает опкоды для специализированных инструкций и сокращает размер интерпретатора.

Специализация набора инструкций может быть использована в открытой модели для компактного кодирования предопределенных системных классов. В таком случае порожденный набор инструкций должен быть совместим со стандартным Java байт-кодом. Такое ограничение не позволяет удалять стандартные инструкции, что существенно ограничивает количество свободных опкодов. Из 256 возможных значений в стандартном байт-коде используются 202 значения, что оставляет 54 свободных значения для специализированных инструкций.

3.2 Шаблоны последовательностей инструкций

Назовем шаблоном последовательность инструкций, параметризованную значениями аргументов некоторых из входящих в нее инструкций. Значения аргументов остальных инструкций определяются шаблоном. Длина аргумента инструкции в параметре шаблона может быть меньше, чем исходная длина аргумента инструкции. Подстановка фактических значений параметризованных аргументов в шаблон порождает последовательность инструкций.

Будем говорить, что шаблон покрывает последовательность инструкций или последовательность инструкций соответствует шаблону, если существуют такие значения параметров, подстановка которых в шаблон порождает заданную последовательность.

Приведем пример шаблона: `getstatic_short8 * ldc 3 invokevirtual 4`. В данной последовательности аргумент инструкции `getstatic` является параметром шаблона, значения аргументов остальных инструкций зафиксированы. В отличие от аргумента исходной инструкции, который кодируется двумя байтами, значение аргумента инструкции `getstatic` в параметре шаблона кодируется с помощью одного байта. Такой шаблон покрывает последовательность инструкций `getstatic 12`, `ldc 3`, `invokevirtual 4`, так как подстановка 12 в качестве значения параметра шаблона порождает соответствующую последовательность.

Предлагаемый алгоритм расширяет набор инструкций специализированными инструкциями для кодирования вышеописанных шаблонов. При кодировании шаблона специализированной инструкцией параметры шаблона становятся аргументами инструкции. Специализированные инструкции осуществляют свертку аргументов за счет того, что аргументы некоторых из инструкций шаблона зафиксированы шаблоном. Укорачивание аргументов осуществляется за счет того, аргумент специализированной инструкции может быть короче, чем соответствующий аргумент оригинальной инструкции. Таким образом, специализированные инструкции могут сочетать в себе все три способа сокращения размера.

Описываемый алгоритм осуществляет свертку и укорачивание аргументов одновременно с выделением часто встречающихся последовательностей, что позволяет генерировать шаблоны, в которых свертка и укорачивание аргументов осуществляется в контексте других инструкций.

3.3 Упрощение исходного набора инструкций

Стандартный набор инструкций Java байт-кода не минимален, так как содержит инструкции, которые могут быть представлены с помощью других инструкций. Избыточные инструкции призваны обеспечить более эффективное

кодирование за счет использования вышеописанных оптимизаций. Примеры избыточных инструкций в стандартном Java байт-коде:

- Инструкции для доступа к первым четырем локальным переменным (`aload_0`, `aload_1`, `iload_0`, `iload_1` и т.п.) сокращают размер байт-кода за счет свертки аргумента. Эти инструкции могут быть представлены с помощью инструкций доступа к локальным переменным с аргументом (`aload 0`, `iload 0`).
- Короткие версии инструкций перехода (`goto`, `jsr`) сокращают размер байт-кода за счет укорачивания аргумента. Эти инструкции могут быть представлены с помощью длинных версий соответствующих инструкций (`goto_w`, `jsr_w`).
- Инструкции вида `if_icmp<cond>` реализуют свертку последовательности инструкций. Такие инструкции эквивалентны последовательностям `isub if<cond>`.

Избыточные инструкции в Java байт-коде выбраны таким образом, чтобы сократить размер типичного приложения. Однако для фиксированного набора приложений в закрытой модели данный набор оптимизированных инструкций может оказаться не оптимален. Такие инструкции сокращают количество свободных опкодов, которые могут быть использованы для специализированных инструкций.

Другая проблема состоит в том, что такие инструкции затрудняют выявление часто встречающихся шаблонов. Так, например, последовательности `aload_0`, `getfield` (0x2A 0xB4) и `aload 5`, `getfield` (0x19 0x05 0xB4) могут быть покрыты одним шаблоном `aload *`, `getfield` (0x19 * 0xB4), что неочевидно из исходных последовательностей.

Перед специализацией набора инструкций осуществим каноникализацию исходных данных, а именно заменим избыточные инструкции в коде методов на эквивалентные последовательности основных инструкций. Заменим инструкции доступа к локальным переменным (`aload_<n>`, `iload_<n>` и другие), инструкции сравнения и условного перехода (`if_icmp<cond>`) и некоторые другие. Такое преобразование упрощает выявление часто встречающихся шаблонов. После каноникализации избыточные инструкции не используются в коде методов, что позволяет удалить их из набора инструкций в закрытой модели. Это, в свою очередь, увеличивает количество свободных опкодов для новых специализированных инструкций. Так как описываемый алгоритм осуществляет оптимизации,

применяемые в избыточных инструкциях, автоматически, некоторые из удаленных инструкций могут быть восстановлены в результате работы алгоритма.

Упрощение исходного набора инструкций перед применением словарного сжатия ранее использовалось для набора инструкций виртуальной машины OmniVM [40]. В OmniVM удаление избыточных инструкций приводит к сокращению итогового размера кода на 9%.

3.4 Построение словаря последовательностей

Для создания новых инструкций будем использовать шаблоны, покрывающие последовательности инструкций и удовлетворяющие следующим ограничениям:

1. Длина последовательности не превышает $max\_pattern\_length$.
2. Последовательность не пересекает границы расширенных базовых блоков с одной точкой входа и множественными выходами. Другими словами, в коде не существует переходов внутрь последовательности.

Для начала построим словарь всех последовательностей исходной программы, удовлетворяющих данным ограничениям. Для каждой последовательности будем хранить число ее вхождений в коде программы. Реализуем соответствующий ассоциативный массив с помощью префиксного дерева. Ребра данного дерева помечаются инструкциями. Каждая вершина такого дерева описывает последовательность инструкций, которую можно восстановить, выписав инструкции всех ребер на пути от корня до данной вершины. В каждой вершине хранится число вхождений соответствующей последовательности. Корень дерева соответствует пустой последовательности. Такую последовательность будем обозначать ε. Пример префиксного дерева словаря последовательностей показан на рисунке 3.1

Опишем алгоритм построения словаря последовательностей. Разобьем тело программы на расширенные базовые блоки и будем анализировать код каждого из блоков отдельно. Обозначим инструкцию в i-ой позиции блока $instr(i)$. Для каждой позиции внутри блока вычислим множество $Sequences(i)$ – множество последовательностей, которые оканчиваются в i-ой позиции. Для этого используем динамическое программирование.

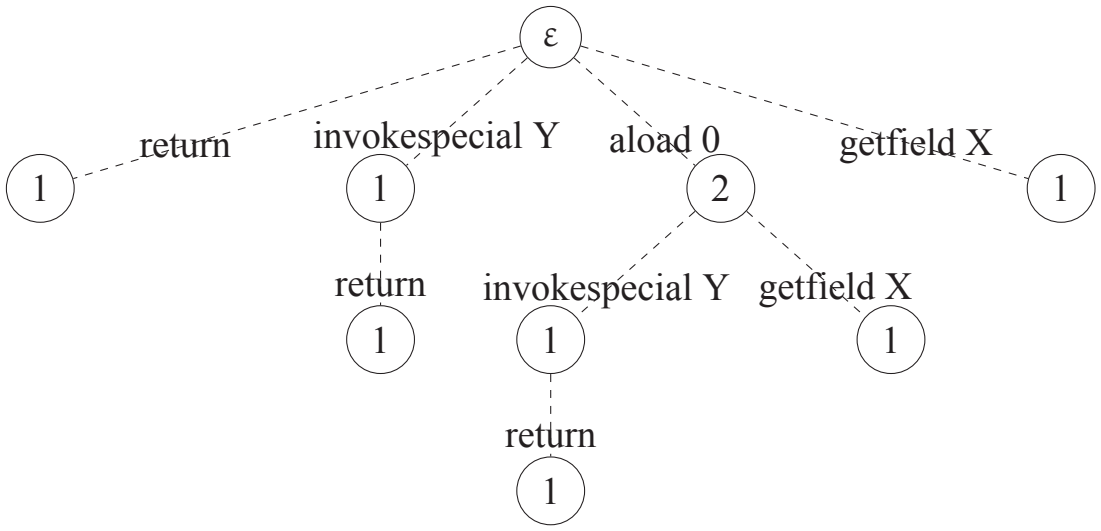


Рисунок 3.1 — Пример префиксного дерева словаря последовательностей для последовательностей aload 0 getfield X, aload 0 invokespecial Y return

– В начале базового блока это множество состоит из одного элемента – пустой последовательности.

$$Sequences(0) = \{\varepsilon\}$$

– Для каждой последующей позиции множество вычисляется на основании множества предыдущей позиции.
  – В данной позиции может начаться новая последовательность, поэтому в множество добавляется пустая последовательность – корень дерева.
  – Каждая последовательность, не длиннее $max\_pattern\_length$, из множества предыдущей позиции может быть продолжена текущей инструкцией, поэтому в множество добавляется расширенная последовательность. Для каждой расширенной последовательности счетчик вхождений увеличивается на единицу.

$$Sequences(i+1) = \{\varepsilon\} \cup \bigcup_{\substack{s \in Sequences(i) \\ length(s) < max\_pattern\_length}} \{extend(s, instr(i+1))\}$$

Операция $extend(s, i)$ продолжает последовательность, заданную вершиной префиксного дерева s, инструкцией i и возвращает расширенную последовательность. Если расширенная последовательность ранее не встречалась, то она добавляется в дерево путем добавления соответствующего ребра из вершины s.

Верхняя оценка количества последовательностей для программы длиной $length$ равна $\sum_{k=1}^{max\_pattern\_length}(length - k + 1)$.

Формальное описание алгоритма приведено в Приложении А.3.

3.5 Инструкции шаблона

Шаблон покрывает последовательность инструкций, если существуют такие значения параметров, подстановка которых в шаблон порождает заданную последовательность. Рассмотрим шаблоны, которыми можно покрыть заданную последовательность инструкций. Для каждой инструкции последовательности на соответствующей позиции в шаблоне должна находиться покрывающая ее инструкция. Инструкции в шаблоне могут быть трех типов.

1. Инструкции без аргумента. Инструкция последовательности без аргумента соответствует такой инструкции, если у них совпадают значения опкодов.
2. Инструкции с фиксированным аргументом. Инструкция последовательности с аргументом соответствует такой инструкции, если у них совпадают значения опкодов и аргументов.
3. Инструкции с параметризованным аргументом. Инструкция последовательности с аргументом соответствует такой инструкции, если у них совпадают значения опкодов, и аргумент инструкции последовательности можно закодировать параметром инструкции шаблона.

Для каждой инструкции последовательности может существовать несколько соответствующих инструкций шаблона. Количество возможных вариантов инструкции шаблона зависит от значения аргумента. Например, инструкция `getfield 16` соответствует следующим вариантам инструкции в шаблоне:

– `getfield 16` – инструкция без параметров,
– `getfield *` – инструкция с двухбайтовым параметром,
– `getfield_short8 *` – инструкция с однобайтовым параметром.

Значение аргумента инструкции `getfield 512` позволяет сопоставить её только с двумя инструкциями:

– `getfield 512` – инструкция без параметров,

– `getfield *` – инструкция с двухбайтовым параметром.

Количество шаблонов, которым соответствует последовательность длины length, равно $\prod_{i=1}^{length} p_i$, где p_i – количество вариантов инструкций шаблона для инструкции последовательности с индексом i. Это значение может быть оценено сверху как ${p_{max}}^{length}$, где p_{max} – максимальное количество вариантов инструкции шаблона, которым может соответствовать инструкция последовательности.

3.6 Перебор шаблонов

Множество шаблонов можно представить в виде префиксного дерева, аналогично тому, как представлено множество последовательностей. Мы не будем хранить префиксное дерево шаблонов в явном виде. Вместо этого используем префиксное дерево последовательностей для перебора множества шаблонов в том же порядке, как если бы мы обходили префиксное дерево шаблонов в глубину. Такой подход позволит существенно сократить объем памяти, необходимый для хранения словаря.

Перебор шаблонов будем осуществлять рекурсивно, начиная с пустого шаблона. На каждом шаге будем вычислять множество всех возможных продолжений текущего шаблона. Далее опишем процесс вычисления всех возможных продолжений для заданного шаблона.

Для каждого шаблона будем вычислять множество $MatchingSequences(p)$[1] – множество всех последовательностей, покрываемых шаблоном. Последовательности в этом множестве представлены вершинами префиксного дерева. Множество $MatchingSequences$ для пустого шаблона содержит один элемент – пустую последовательность.

Множество исходящих ребер для данной вершины префиксного дерева обозначим $Edges(s)$. Это множество определяет множество инструкций, которыми

[1] Здесь и далее символ s используется для обозначения последовательностей инструкций (sequence), символ p – для обозначения шаблонов (pattern).

может быть продолжена соответствующая последовательность. Следующее множество определяет множество инструкций программы, которыми может быть продолжен заданный шаблон p.

$$SIContinuations(p) = \{Edges(s) | s \in MatchingSequences(p)\}$$

По множеству $SIContinuations(p)$ определим множество инструкций шаблона, которыми может быть продолжен шаблон.

$$PIContinuations(p) = \{PI(si) | si \in SIContinuations(p)\},$$

где $PI(si)$ – множество инструкций шаблона, которыми можно покрыть инструкцию последовательности si.

Множество $SIContinuations(p)$ определяет все возможные продолжения шаблона p. Теперь для каждого продолжения необходимо вычислить множество последовательностей, покрываемых продолжением. Это множество вершин, в которые можно перейти из вершин текущего множества последовательностей по ребрам, соответствующим новой инструкции шаблона.

$$MatchingSequences(p, pi) = \bigcup_{s \in MatchingSequences(p)} \bigcup_{si \in SIContinuations(p)} \{extend(s, si) | matches(si, pi)\},$$

где $matches(si, pi)$ – функция, которая определяет, соответствует ли инструкция последовательности si инструкции шаблона pi.

Не будем расширять шаблоны инструкциями переходов с фиксированным аргументом, то есть не будем сворачивать аргументы таких инструкций. Это связано с тем, что значение аргумента инструкции перехода является смещением в теле метода и может меняться при введении специализированных инструкций. Укорачивать аргументы инструкций переходов при этом можно, так как значения смещений не могут увеличиваться.

На рисунке 3.2 показан процесс построения шаблона `aload * getfield X` для последовательностей `aload 0 getfield X`, `aload 1 getfield X return`.

3.7 Выбор набора инструкций

Задача выбора набора инструкций состоит в выборе шаблонов, которые будут закодированы специализированными инструкциями. На множество шаблонов при этом налагаются следующие ограничения:

- Общее количество шаблонов не должно превышать 256. Аналогично стандартному Java байт-коду описываемый алгоритм использует однобайтовое кодирование инструкций.
- Исходный код программы может быть покрыт шаблонами из этого множества.

Выбранный набор инструкций должен сократить размер приложения в новом байт-коде вместе с размером интерпретатора, необходимого для его исполнения. Известно, что задача выбора оптимального словаря для методов словарного сжатия является NP-полной [101]. На практике для выбора словаря используются различные эвристики, например, частотный подход, когда в словарь включаются самые часто встречающиеся подстроки исходного текста. Аналогичный подход используется в описываемом алгоритме.

Для выбора набора инструкций используем жадный итеративный алгоритм. На каждом шаге алгоритм выбирает шаблон, кодирование которого с помощью специализированной инструкции обеспечивает наибольшее сокращение размера. Сокращение размера, обеспечиваемое заданным шаблоном, будем называть весом шаблона и будем вычислять следующим образом:

$$weight(p) = count(p) * bytecode\_gain(p) - interpreter\_size(p),$$

где $count(p)$ – сколько раз данный шаблон встречается в коде программы;
$bytecode\_gain(p)$ – выигрыш от замены одного вхождения шаблона на новую инструкцию;
$interpreter\_size(p)$ – размер кода интерпретатора, реализующего данный шаблон, считается равным сумме размеров кода интерпретатора для каждой составляющей его инструкции.

Для заданного шаблона вычислим $count(p)$ как сумму вхождений последовательностей, покрываемых шаблоном. Вершины префиксного дерева

хранят количество вхождений соответствующих последовательностей, таким образом, количество вхождений шаблона равно сумме значений в узлах $MatchingSequences(p)$.

Для заданного шаблона $bytecode\_gain(p)$ считается следующим образом:

$$bytecode\_gain(p) = \sum_{i=1}^{length} bytecode\_gain_i - 1,$$

где $bytecode\_gain_i$ – выигрыш от инструкции в шаблоне с индексом i, который равен 1 + (размер оригинального аргумента инструкции минус размер параметра инструкции в шаблоне).

Итеративный алгоритм начинается со стандартного набора инструкций Java байт-кода. На каждом шаге в набор добавляется одна инструкция. Для выбора новой инструкции выполняются следующие шаги:

1. Строится словарь последовательностей, как описано в разделе 3.4.
   При построении словаря запоминаются используемые в коде программы опкоды. При генерации набора инструкций, совместимого со стандартным Java байт-кодом, опкоды всех стандартных инструкций считаются используемыми. Если использованы все 256 доступных значений опкодов, то алгоритм завершается.
2. С помощью алгоритма, описанного в разделе 3.6, перебирается множество всех шаблонов. Находится шаблон с максимальным весом.
   При поиске шаблона с наибольшим весом перебор множества шаблонов может быть сокращен. Если несколько различных продолжений шаблона покрывают один и тот же набор последовательностей, то среди них достаточно выбрать продолжение с наибольшим весом. Например, если продолжения шаблона короткой и длинной версией некой инструкции приводят к одному и тому же набору подходящих последовательностей, то нет необходимости продолжать шаблон длинной версией этой инструкции.
   Если вес найденного шаблона не положителен, то алгоритм завершается.
3. Набор инструкций расширяется инструкцией для кодирования шаблона, выбранного на предыдущем шаге. Для кодирования новой инструкции используется один из свободных опкодов. Все вхождения шаблона в программе заменяются специализированной инструкцией.

Формальное описание алгоритма приведено в Приложении А.3.

Верхняя оценка количества итераций алгоритма равна 256 – общему количеству возможных опкодов. Добавление новой инструкции является дорогой операцией, так как требует перестроения словаря последовательностей и перебора множества шаблонов.

Перестроение словаря шаблонов требуется по следующим причинам. Во-первых, добавление новой инструкции существенно изменяет актуальные веса оставшихся шаблонов. Большинство шаблонов с наибольшим весом являются либо разными шаблонами для одной и той же последовательности, либо шаблонами пересекающихся последовательностей. Замена шаблона с наибольшим весом на специализированную инструкцию существенно сокращает количество вхождений других шаблонов. Во-вторых, после замены вхождений шаблона новой инструкцией некоторые из ранее используемых опкодов могут перестать использоваться. На следующей итерации освободившиеся значения опкодов могут быть использованы для новых инструкций.

3.8 Выводы

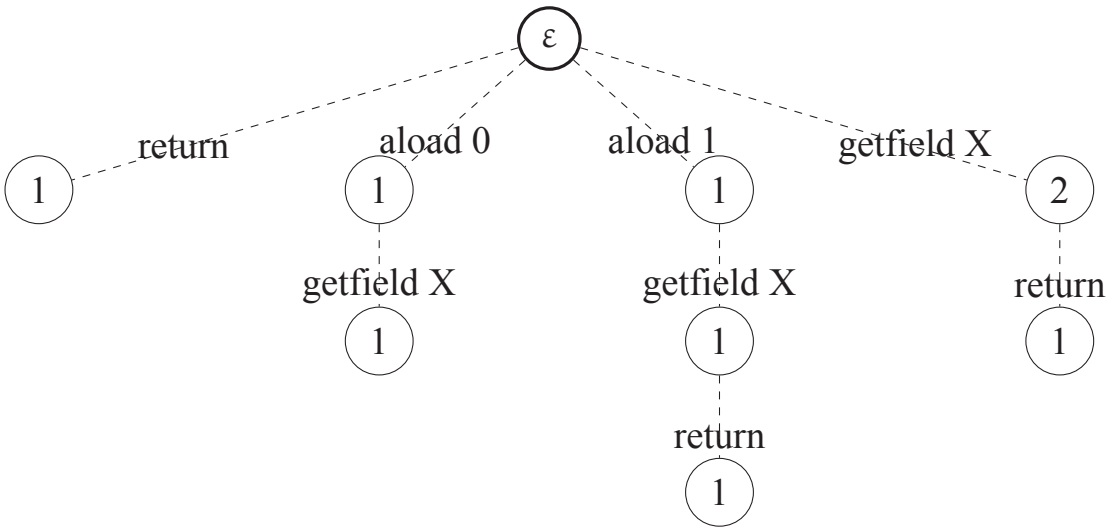
В третьей главе была рассмотрена задача сокращения размера интерпретируемого кода путем специализации набора инструкций. Данный подход является развитием широко используемой идеи словарного сжатия. Был предложен алгоритм специализации набора инструкций Java байт-кода, сокращающий размер кода заданного приложения и интерпретатора, необходимого для его исполнения.

Было отмечено, что избыточные инструкции в стандартном байт-коде могут уменьшить эффективность словарного сжатия. Для того чтобы обойти эту проблему, предложенный алгоритм предварительно упрощает исходный набор инструкций. Другой отличительной особенностью алгоритма является свертка и укорачивание аргументов в процессе построения словаря шаблонов. Это позволяет осуществлять свертку и укорачивание аргументов в контексте последовательностей инструкций.

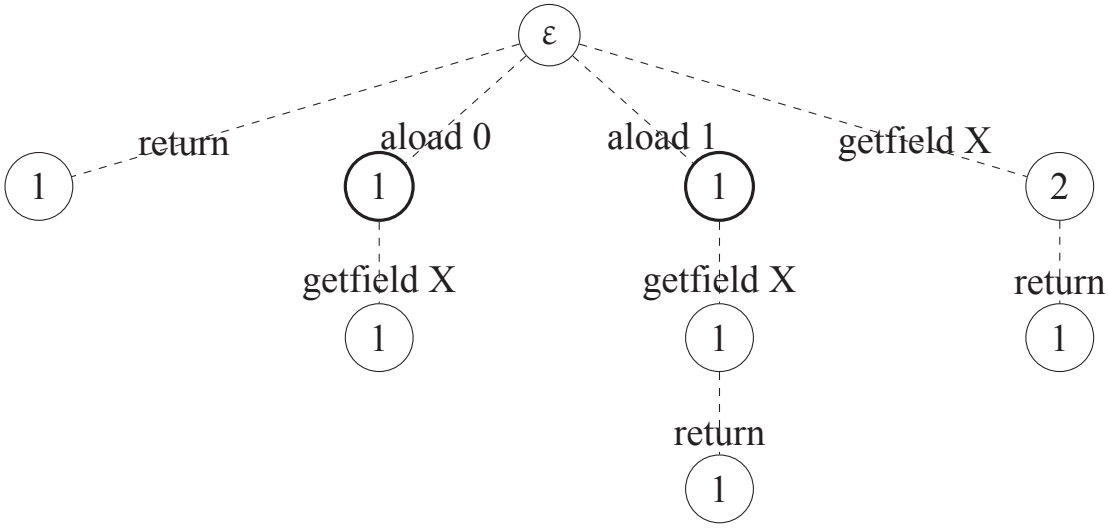
Для перебора шаблонов используется оригинальный алгоритм. Множество шаблонов не хранится в памяти, вместо этого множество вычисляется динамически на основе префиксного дерева, описывающего последовательности

инструкций. Такой подход позволяет существенно сократить потребление памяти.

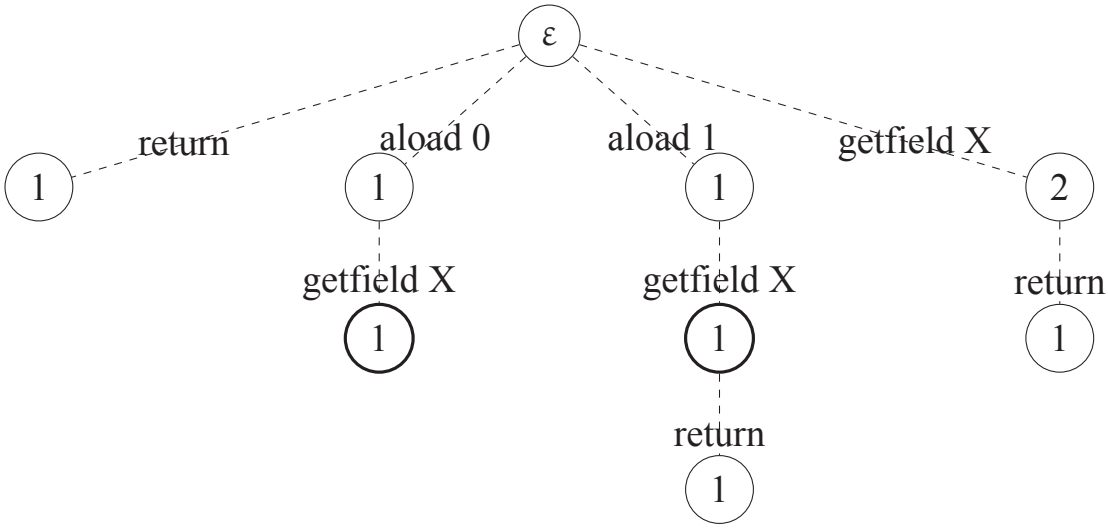
В данной главе решена 4-я задача диссертации: «Разработать алгоритм сжатия Java байт-кода в закрытой модели, применимый для встраиваемых систем с ограниченными ресурсами». Реализация предложенного алгоритма сжатия Java байт-кода при специализации Java-платформы для заданного приложения в закрытой модели обеспечит более компактное кодирование исполняемого кода приложения.



(а) Пустой шаблон



(б) Шаблон `aload *`, количество вхождений = 2



(в) Шаблон `aload * getfield X`, количество вхождений = 2

Жирным выделены узлы, принадлежащие множеству MatchingSequences текущего шаблона.

Рисунок 3.2 — Пример построения шаблона aload * getfield X для последовательностей aload 0 getfield X, aload 1 getfield X return

Глава 4. Программная реализация и экспериментальное исследование

Данная глава посвящена описанию программной реализации предложенных алгоритмов и экспериментальному исследованию данной реализации. Предложенные алгоритмы сравниваются с существующими решениями. Отмечаются недостатки и ограничения предложенных решений, выделяются направления для дальнейшего исследования.

4.1 Реализация предложенных алгоритмов

В рамках работы был реализован инструмент для автоматической специализации Java-платформы Oracle Java ME Embedded для заданного приложения в закрытой модели. Платформа Oracle Java ME Embedded доступна на аппаратных платформах различных производителей, например, в беспроводных модулях Gemalto Cinterion [102—105]. При реализации данного инструмента были использованы алгоритмы, предложенные в главах 2 и 3. Процесс специализации платформы изображен на рисунке 4.1.

Специализация Java-кода платформы для заданного приложения осуществляется в инструменте ромизации виртуальной машины CLDC Hotspot Implementation (romizer). Romizer представляет собой специальную версию виртуальной машины, которая исполняется на хостовом устройстве в процессе построения Java-плафтормы.

Входными данными инструмента являются системные классы и классы приложения. На выходе инструмент генерирует загрузочный образ инициализированного состояния виртуальной машины. В процессе подготовки образа загружаются системные классы и классы приложения, разрешаются символические ссылки, осуществляются оптимизации в памяти виртуальной машины, призванные сократить размер образа и ускорить исполнение на целевом устройстве. Состояние памяти инициализированной виртуальной машины сохраняется в виде исходного файла на языке C++ ROMImage.cpp. Данный файл определяет статические массивы, содержимое которых описывает состояние памяти виртуальной машины.

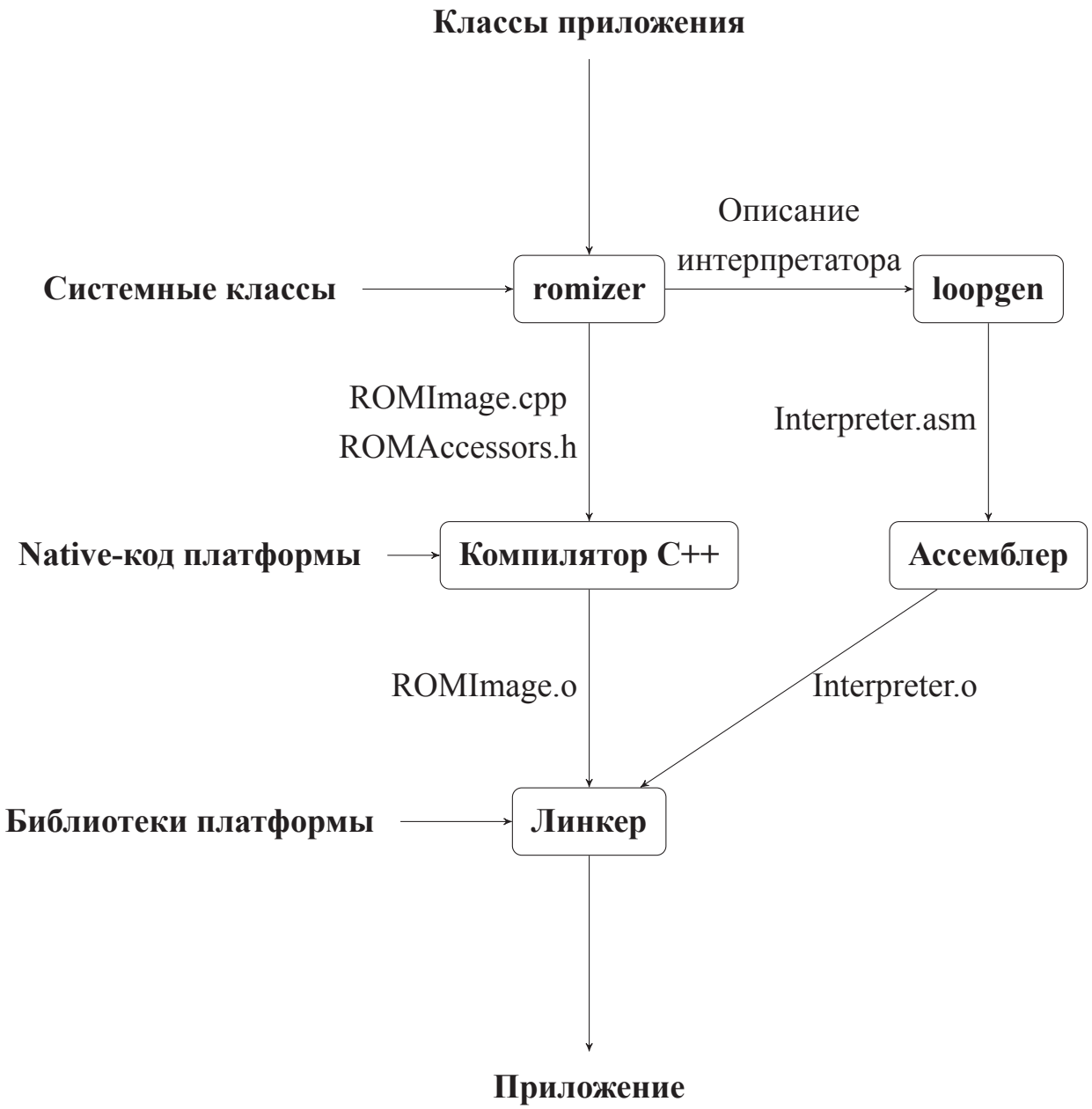


Рисунок 4.1 — Процесс специализации Java-платформы Oracle Java ME Embedded для заданного приложения в закрытой модели

Для специализации Java-кода платформы при подготовке образа в инструменте romizer были реализованы алгоритмы понижения избыточности, представленные в главе 2. Данные алгоритмы были реализованы как часть оптимизаций, осуществляемых после загрузки классов и разрешения символических ссылок. Результатом понижения избыточности с помощью реализованных алгоритмов является:

– Инициализация некоторых из классов, используемых приложением.
– Удаление из памяти виртуальной машины методов, полей и классов, не используемых в загруженном коде.

После понижения избыточности romizer специализирует интерпретатор для компактного кодирования байт-кода оставшихся методов. В результате специализации байт-код методов в памяти виртуальной машины переписывается в новом наборе инструкций. На выходе romizer генерирует файл с описанием специализированного интерпретатора, который затем используется для генерации ассемблерного кода интерпретатора с помощью инструмента loopgen. Впоследствии код интерпретатора компилируется ассемблером целевой платформы.

В текущей реализации romizer использует подмножество оптимизаций, предложенных в главе 3. Полностью предложенный алгоритм специализации набора инструкций был реализован в виде отдельного инструмента. Для заданного набора класс-файлов инструмент порождает код в новом наборе инструкций и описание интерпретатора, необходимого для его исполнения. Для измерений эффективности алгоритма специализации набора инструкций был использован данный инструмент.

Помимо загрузочного образа romizer генерирует файл ROMAccessors.h, содержащий статический интерфейс для доступа к полям Java-объектов. Данный интерфейс используется для взаимодействия с Java-объектами в native-коде виртуальной машины и системных классов. Подробно данный интерфейс был рассмотрен в разделе 2.10.

Файлы ROMImage.cpp, ROMAccessors.h компилируются совместно с native-кодом платформы, а затем линкуются вместе с интерпретатором и библиотеками в один исполняемый файл. В процессе линковки осуществляется специализация native-кода платформы за счет удаления неиспользуемых функций. Итоговый исполняемый файл содержит приложение и реализацию платформы, специализированную для заданного приложения.

4.2 Сравнение алгоритмов понижения избыточности

Сравним систему понижения избыточности, реализованную в рамках данной работы, с существующими системами, описанными в разделе 1.6. В реализованной системе понижение избыточности используется для оптимизации загрузочного образа при раздельной инициализации. При этом при раздельной инициализации для уменьшения размера образа и сокращения времени запуска

приложения используется ранняя инициализация классов. Из всех рассмотренных систем только JITS и ExoVM используют понижение избыточности совместно с ромизацией и ранней инициализацией классов. В JITS для анализа достижимости методов используется алгоритм CHA, что позволяет не анализировать инициализированное состояние виртуальной машины. Алгоритм CHA более консервативен по сравнению с предложенным алгоритмом и не эффективен при анализе универсальных библиотек.

В ExoVM при подготовке образа инициализируются все загруженные классы. Это нарушает спецификацию языка Java и может привести к увеличению размера образа. Алгоритм анализа достижимости методов, предложенный соискателем, осуществляет выборочную инициализацию классов. Из всех классов, которые могут быть инициализированы достижимым кодом, алгоритм выбирает те, что могут быть безопасно инициализированы при подготовке образа. Для ранней инициализации класса может быть использована аннотация `@InitAtBuild`.

Большинство рассмотренных систем понижения избыточности для анализа достижимости методов используют алгоритмы, основанные на RTA. Исключение составляет Jax Application Extractor [18—20; 61], который реализует несколько различных алгоритмов анализа достижимости методов. В число реализуемых алгоритмов входят CHA, RTA, XTA. Анализ достижимости методов, реализованный в данной работе, также построен на базе алгоритма RTA. Более точные алгоритмы анализа достижимости методов могут быть расширены раздельной инициализацией классов аналогичным образом.

При использовании ранней инициализации классов анализ достижимости методов должен учитывать объекты, созданные при инициализации классов, так как такие объекты могут быть использованы в качестве получателей для косвенных вызовов. В ExoVM для этого используется расширение алгоритма RTA под названием RMA. В дополнение к множеству инстанциируемых классов RMA вычисляет множество классов объектов, достижимых для приложения. Классы из объединения этих множеств используются для разрешения косвенных вызовов. Алгоритм, реализованный соискателем, использует аналогичный подход, но уточняет его для финализируемых объектов.

Удаление неиспользуемых полей сокращает статическое и динамическое потребление памяти за счет удаления ассоциированных с полями метаданных и сокращения размера объектов. Наиболее консервативный подход к удалению

неиспользуемых полей состоит в удалении таких полей, для которых в достижимом коде нет никаких операций. Такой подход используется в системе, описанной в [21].

Более точный подход состоит в удалении полей, для которых в достижимом коде нет операций чтения [17]. Такое уточнение позволяет удалять поля, которые инициализируются достижимым кодом, но не используются после инициализации. Этот подход используется в системах Jax Application Extractor и ExoVM.

Как было показано ранее в Главе 2, в Java удаление инициализированных, но неиспользуемых объектных полей не всегда безопасно. Удаление объектных полей может привести к преждевременной финализации объектов и нарушению видимого поведения. В работах, посвященных Jax Application Extractor и ExoVM, данная проблема не рассматривается. Инструмент Jamaica Builder виртуальной машины JamaicaVM не удаляет объектные поля, для которых в достижимом коде есть операции записи, даже если для них нет операций чтения. Такой подход сохраняет семантику финализации.

Алгоритм удаления неиспользуемых полей, предложенный в данной работе, позволяет удалять инициализируемые, но неиспользуемые поля, сохраняя при этом семантику финализации. Предложенный алгоритм не удаляет поля, для которых не удалось доказать, что через ссылку в поле не может быть достижим неудалимый объект.

Любая практическая реализация системы понижения избыточности должна иметь механизм для описания дополнительных зависимостей. Такой механизм используется для ручного описания зависимостей native методов и кода, использующего рефлексию. Стандартный подход к описанию дополнительных зависимостей состоит в перечислении методов, полей и классов, которые не могут быть удалены. Такой подход используется в Jax Application Extractor и в системе, описанной в статье [21]. Недостатком такого подхода является то, что перечисленные методы, поля и классы будут сохранены, даже если использующие их методы окажутся недостижимы.

Более точно дополнительные зависимости обрабатываются в ExoVM, где реализация анализа достижимости содержит описание зависимостей известных native методов стандартной библиотеки классов. Дополнительные зависимости метода применяются, только если метод оказался достижимым. Более точное

описание дополнительных зависимостей позволяет повысить эффективность понижения избыточности. Недостатком такой реализации является то, что описание дополнительных зависимостей зафиксировано в реализации алгоритма.

Система понижения избыточности, реализованная в рамках данной работы, позволяет описывать дополнительные зависимости индивидуальных методов. Зависимости метода применяются, только если метод оказался достижимым. При этом дополнительные зависимости не зафиксированы в алгоритме, а описываются в виде конфигурационных файлов или аннотаций в Java-коде. Кроме того, в работе был предложен и реализован механизм автоматического анализа зависимостей native методов.

4.3 Экспериментальная оценка алгоритмов понижения избыточности

Для оценки эффективности реализованных алгоритмов понижения избыточности и сравнения их с существующими решениями был проведен ряд численных экспериментов. Сформулируем задачи для экспериментов.

1. **Определить сокращение размера, обеспечиваемое реализованными алгоритмами.**
2. **Определить, как использование более точного анализа влияет на размер образа.** Реализованный алгоритм анализа достижимости методов является адаптацией алгоритма RTA для анализа инициализированного состояния виртуальной машины. Альтернативный подход реализован в виртуальной машине JITS, где используется более консервативный анализ достижимости методов CHA. Данный анализ не зависит от инициализированного состояния виртуальной машины.
3. **Продемонстрировать сохранение поведения при использовании выборочной инициализации. Определить, на сколько выборочная инициализация классов сокращает размер образа по сравнению с безусловной инициализацией.** Реализованный алгоритм анализа достижимости методов выборочно инициализирует классы, которые могут быть инициализированы в процессе работы приложения. Альтернативный подход реализован в виртуальной машине ExoVM, где все

загруженные классы инициализируются безусловно до анализа достижимости. Безусловная инициализация классов может нарушить поведение программы и увеличить размер образа.

4. **Продемонстрировать сохранение семантики финализации при использовании реализованного алгоритма. Определить, как удаление инициализируемых, но неиспользуемых полей влияет на размер образа.** Реализованный алгоритм анализа удалимости полей позволяет удалять инициализируемые, но неиспользуемые поля, сохраняя при этом семантику финализации объектов. Большинство рассмотренных систем понижения избыточности нарушают семантику финализации, удаляя инициализируемые, но неиспользуемые объектные поля. В виртуальной машине JamaicaVM объектные поля, для которых есть операции записи не удаляются.
5. **Определить, какое количество native-методов проанализировано автоматически, какое количество зависимостей сгенерировано автоматически.** Система понижения избыточности, реализованная в рамках данной работы, автоматически анализирует зависимости native-методов. Ни одна из рассмотренных систем понижения избыточности не анализирует межъязыковые зависимости автоматически.

Отметим, что во всех рассмотренных системах понижения избыточности удаление неиспользуемых методов, полей и классов используется совместно с другими оптимизациями. Эти оптимизации могут оказывать влияние на итоговый размер приложения. Кроме того, в разных системах используются различные конфигурации и реализации стандартных библиотек. Таким образом, даже для одного и того же исходного приложения, набор анализируемых классов может существенно отличаться. Данные факторы не позволяют непосредственно сравнивать размеры приложений, сгенерированных разными системами. Распространенной практикой является реализация сравниваемых алгоритмов в одной виртуальной машине и сравнение размеров приложений, сгенерированных при применении различных алгоритмов [15; 30; 50—52]. Для решения задач экспериментов в виртуальной машине CLDC HI были реализованы модификации предложенного алгоритма.

Для измерений были использованы 8 приложений для следующих конфигураций:

– CLDC – реализация стандарта Connected Limited Device Configuration 8 (JSR360) [106].
– MEEP – реализация стандарта Java ME Embedded Profile (JSR361) [107].

Таблица 1 — Набор приложений

	Приложение	Конфигурация	Описание
1	Hello World	CLDC	приложение, печатающее ”Hello World”
2	Chess	CLDC	игра шахматы
3	Crypto	CLDC	криптографический пакет
4	Parallel	CLDC	тест на многопоточное исполнение
5	kXML	CLDC	XML парсер
6	PNG	CLDC	декодер PNG изображений
7	RegExp	CLDC	пакет вычисления регулярных выражений
8	AMS	MEEP	Application Management System

Информация об использованных приложения приведена в таблице 1. Приложение 1 является примером минимального приложения для платформы CLDC, приложения 2-7 используются в наборе тестов EEMBC GrinderBench [108]. Данный набор тестов часто используется для оценки производительности реализаций Java-платформы для встраиваемых устройств. В частности, данные приложения были использованы для анализа эффективности алгоритмов понижения избыточности виртуальной машины ExoVM [29]. Приложение 8 является примером приложения для конфигурации с большим количеством классов.

Эксперимент 1. Определить сокращение размера, обеспечиваемое реализованными алгоритмами.

Для данного эксперимента были измерены размеры образов:

– none – без понижения избыточности;
– base – с понижением избыточности с помощью реализованных алгоритмов.

Информация о размере образов и относительном сокращении размера приведена в таблице 2 и рисунке 4.2. Применение реализованных алгоритмов понижения избыточности в закрытой модели сокращает размер образа на 57,62–95,18 % (среднее значение 74,57 %).

Эксперимент 2. Как использование более точного анализа влияет на размер образа?

Таблица 2 — Эксперимент 1. Размер образа в байтах и относительное сокращение размера

	Приложение	none	base (vs none)
1	Hello World	255968	12340 (-95,18 %)
2	Chess	288516	76404 (-73,52 %)
3	Crypto	291476	86684 (-70,26 %)
4	Parallel	308284	84944 (-72,45 %)
5	kXML	311224	80472 (-74,14 %)
6	PNG	289196	63652 (-77,99 %)
7	RegExp	301652	74288 (-75,37 %)
8	AMS	1613388	683832 (-57,62 %)

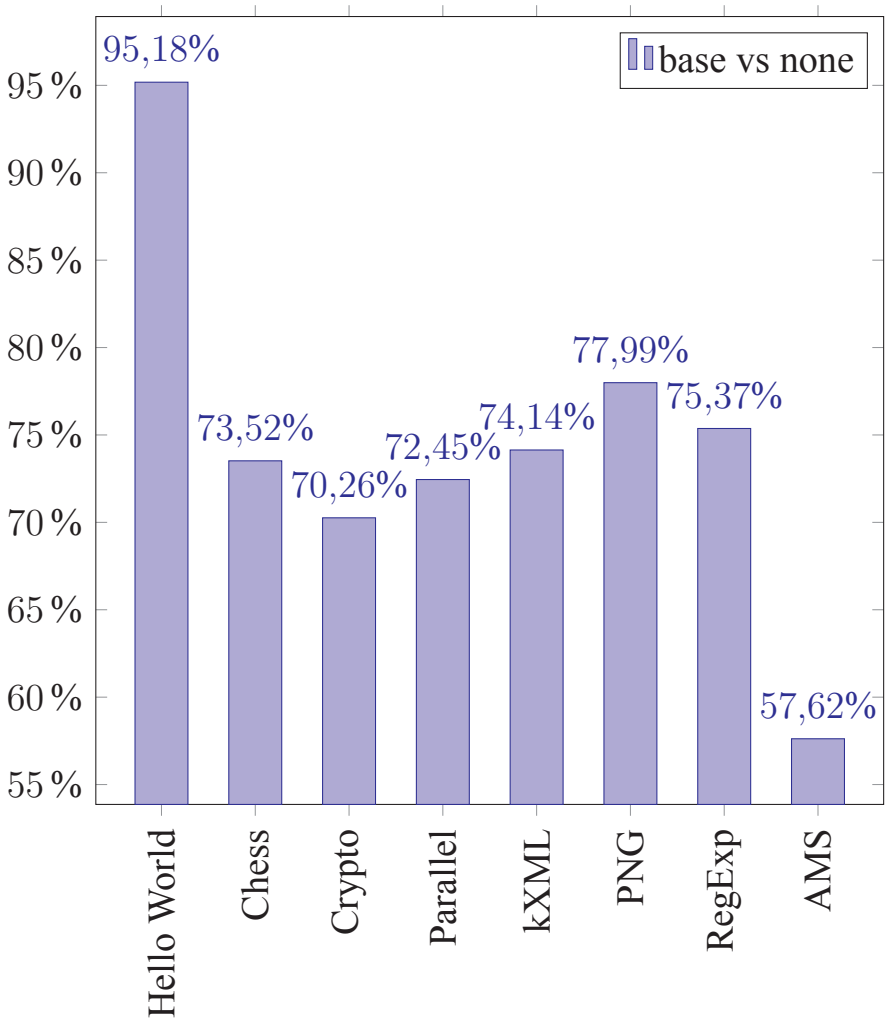


Рисунок 4.2 — Эксперимент 1. Относительное сокращение размера образа

Для данного эксперимента была реализована модификация предложенного алгоритма, которая использует анализ CHA для разрешения косвенных вызовов (cha). Измерен размер образов, полученных при использовании исходного и модифицированного алгоритмов. Информация о размере образов и относительном сокращении размера приведена в таблице 3 и рисунке 4.3. Применение предложенного алгоритма анализа достижимости методов сокращает размер образа на 29,57–50,86 % (среднее значение 40,88 %) по сравнению с алгоритмом CHA.

Эксперимент 3. Продемонстрировать сохранение поведения при использовании выборочной инициализации. Определить, на сколько выборочная инициализация классов сокращает размер образа по сравнению с безусловной инициализацией.

Для данного эксперимента была реализована модификация предложенного алгоритма, которая безусловно инициализирует загруженные классы до анализа достижимости (init-all).

Следующий тест демонстрирует нарушение поведения при безусловной инициализации классов.

```
class A {
  static {
    Test.a_is_initialized = true;
  }
}
class Test {
  public static boolean a_is_initialized;
  public static void main(String args[]) throws Exception {
    if (a_is_initialized)
      throw new Exception("A could not have been initialized");
  }
}
```

Ромизация с использованием модифицированного алгоритма приводит к инициализации класса `A` и бросанию исключения при исполнении программы. Ромизация с использованием предложенного алгоритма не инициализирует класс `A` и исполнение программы завершается корректно.

Для оценки влияния выборочной инициализации на размер образа был измерен размер образов, полученных при использовании исходного и модифицированного алгоритмов. Информация о размере образов и относительном

Таблица 3 — Эксперимент 2. Размер образа в байтах и относительное сокращение размера

	Приложение	cha	base (vs cha)
1	Hello World	19040	12340 (-35,19 %)
2	Chess	125052	76404 (-38,90 %)
3	Crypto	131556	86684 (-34,11 %)
4	Parallel	149896	84944 (-43,33 %)
5	kXML	152268	80472 (-47,15 %)
6	PNG	129532	63652 (-50,86 %)
7	RegExp	142732	74288 (-47,95 %)
8	AMS	970948	683832 (-29,57 %)

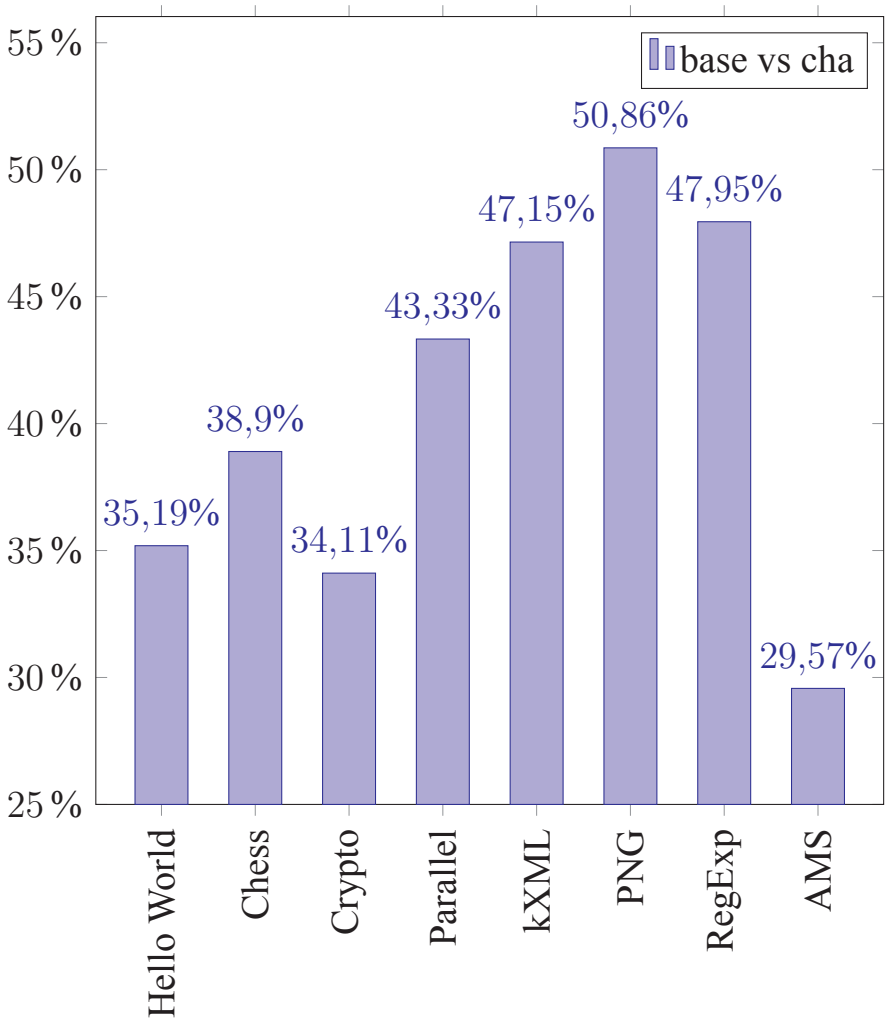


Рисунок 4.3 — Эксперимент 2. Относительное сокращение размера образа

сокращении размера приведена в таблице 4 и рисунке 4.4. Применение выборочной инициализации во время анализа достижимости методов сокращает размер образа на 0,3–11,71 % (среднее значение 4,27 %) по сравнению с безусловной инициализацией всех классов.

Необходимо отметить, что инициализаторы классов нередко используют метод `System.getPropery` и аналогичные платформо-зависимые методы, результат исполнения которых при подготовке образа и на целевом устройстве может отличаться. Ранняя инициализация таких классов также может привести к нарушению поведения программы. В наборе классов конфигурации CLDC данным свойством обладает 1 класс, в конфигурации MEEP – 47 классов.

Эксперимент 4. Продемонстрировать сохранение семантики финализации при использовании реализованного алгоритма. Определить, как удаление инициализируемых, но неиспользуемых полей влияет на размер образа.

Для данного эксперимента были реализованы две модификации предложенного алгоритма:

– dont-remove – наиболее консервативная модификация, не удаляет инициализируемые, но неиспользуемые объектные поля;
– dont-preserve – модификация не сохраняет семантику финализации и удаляет все инициализируемые, но неиспользуемые объектные поля.

Следующий тест демонстрирует нарушение семантики финализации при удалении инициализируемых, но неиспользуемых полей.

```
class A {}
class B {}
public class Test {
  static A removable = new A();
  static B unremovable = new B();
  static WeakReference<B> weak = new WeakReference<>(unremovable);
  public static void main(String args[]) throws Exception {
    if (weak.get() == null)
      throw new Exception("Object should be reachable!");
  }
}
```

При ромизации теста с использованием модификации dont-preserve удаляются поля `removable` и `unremovable`. Удаление поля `unremovable` приводит к нарушению достижимости экземпляра класса `B`, на который ссылалось поле.

Таблица 4 — Эксперимент 3. Размер образа в байтах и относительное сокращение размера

	Приложение	init-all	base (vs init-all)
1	Hello World	13976	12340 (-11,71 %)
2	Chess	80332	76404 (-4,89 %)
3	Crypto	88560	86684 (-2,12 %)
4	Parallel	86972	84944 (-2,33 %)
5	kXML	83944	80472 (-4,14 %)
6	PNG	67108	63652 (-5,15 %)
7	RegExp	76980	74288 (-3,50 %)
8	AMS	685884	683832 (-0,30 %)

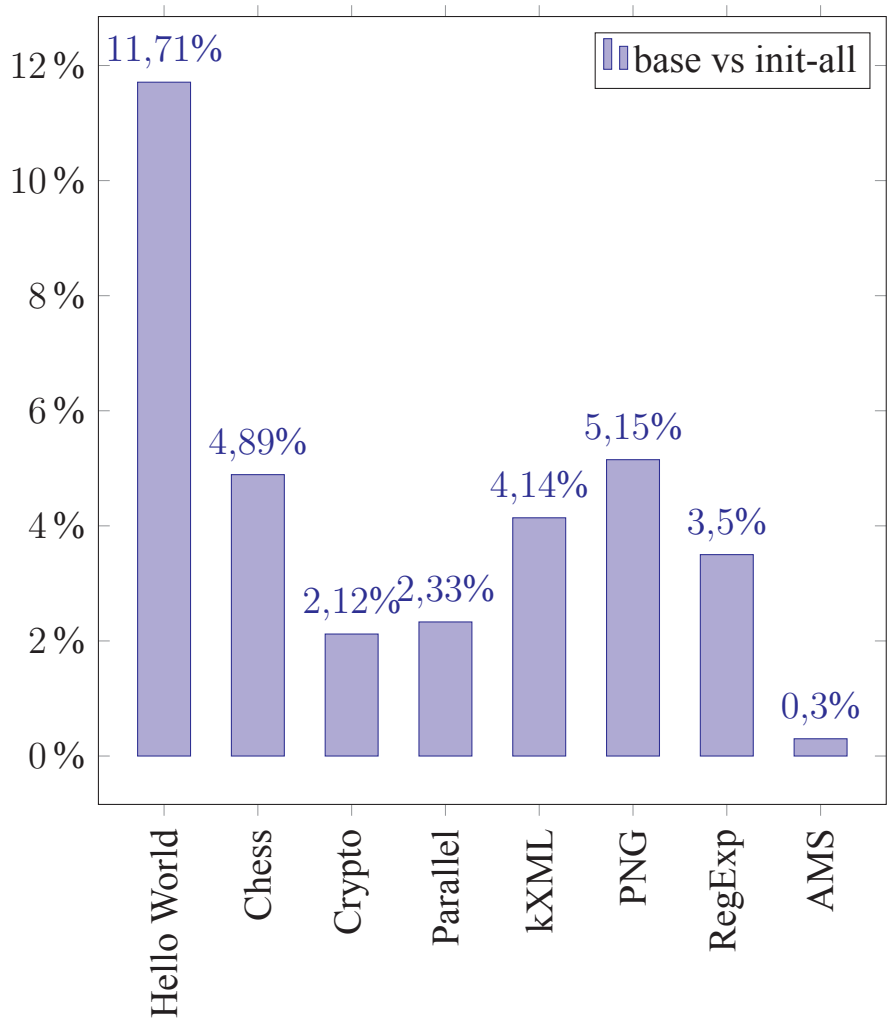


Рисунок 4.4 — Эксперимент 3. Относительное сокращение размера образа

Объект удаляется сборщиком мусора и слабая ссылка на данный объект обнуляется. В процессе исполнения теста бросается исключение, поскольку слабая ссылка не содержит ссылки на объект.

Ромизация с использованием предложенного алгоритма удаляет поле `removable`, но в то же время не удаляет поле `unremovable`. Таким образом, сохраняется достижимость объекта, доступного посредством специальной ссылки, и исполнение программы завершается корректно.

Приложение, сгенерированное с помощью инструмента Jamaica Builder, завершает исполнение корректно. Jamaica Builder не удаляет поля `removable` и `unremovable`, так как для них есть операции записи. Аналогичное поведение демонстрирует модификация алгоритма dont-remove.

Для оценки влияния различных критериев удалимости полей на размер образа был измерен размер образов, полученных при использовании исходного и модифицированных алгоритмов. Информация о размере образов и относительном сокращении размера приведена в таблице 5 и рисунке 4.5. Кроме того, для каждого образа было измерено количество включенных в него полей. Данная информация приведена в таблице 6 и рисунке 4.6.

Удаление инициализируемых, но неиспользуемых полей с помощью предложенного критерия сокращает размер образа на 0,64–4,37 % (среднее значение 1,32 %) и сокращает количество полей в образе на 5,55–13,08 % (среднее значение 10,09 %). Применение предложенного алгоритма анализа удалимости полей не приводит к существенному сокращению размера образа, но сокращает количество полей в образе, что может сократить динамическое потребление памяти.

Удаление всех инициализируемых, но неиспользуемых полей без учета семантики финализации приводит к незначительному сокращению размера образа (среднее значение 0,02 %) и количества полей (среднее значение 1,55 %) по сравнению с предложенным алгоритмом.

Эксперимент 5. Какое количество native-методов проанализировано автоматически? Какое количество зависимостей сгенерировано автоматически?

В рассматриваемых конфигурациях (CLDC, MEEP) классы приложений не могут содержать native-методы, такие методы могут встречаться только в классах системных библиотек. Таким образом, набор native-методов зависит только от конфигурации и не зависит от приложения. Информация о количестве native-методов и количестве зависимостей, сгенерированных для данных методов с помощью реализованного механизма анализа, приведена в таблице 7.

Таблица 5 — Эксперимент 4. Размер образа в байтах и относительное сокращение размера

	Приложение	dont-remove	base (vs dont-remove)	dont-preserve (vs base)
1	Hello World	12904	12340 (-4,37 %)	12340 (-0,00 %)
2	Chess	77064	76404 (-0,86 %)	76384 (-0,03 %)
3	Crypto	87240	86684 (-0,64 %)	86664 (-0,02 %)
4	Parallel	85580	84944 (-0,74 %)	84924 (-0,02 %)
5	kXML	81100	80472 (-0,77 %)	80436 (-0,04 %)
6	PNG	64352	63652 (-1,09 %)	63632 (-0,03 %)
7	RegExp	75156	74288 (-1,15 %)	74268 (-0,03 %)
8	AMS	690152	683832 (-0,92 %)	683812 (-0,00 %)

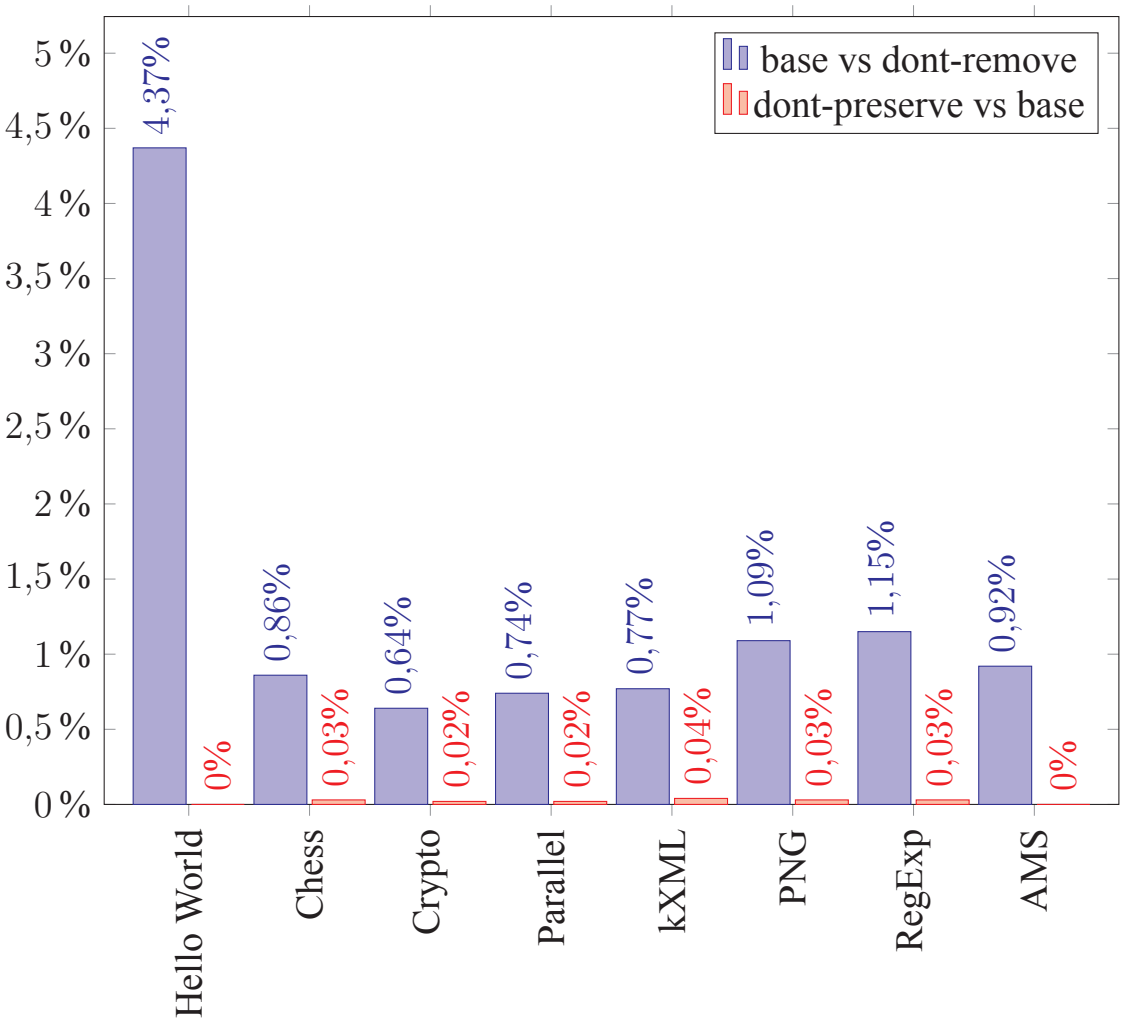


Рисунок 4.5 — Эксперимент 4. Относительное сокращение размера образа

Таблица 6 — Эксперимент 4. Количество полей в образе и относительное сокращение количества полей

	Приложение	dont-remove	base (vs dont-remove)	dont-preserve (vs base)
1	Hello World	52	46 (-11,54 %)	46 (-0,00 %)
2	Chess	233	210 (-9,87 %)	208 (-0,95 %)
3	Crypto	168	154 (-8,33 %)	152 (-1,30 %)
4	Parallel	192	171 (-10,94 %)	169 (-1,17 %)
5	kXML	128	116 (-9,38 %)	109 (-6,03 %)
6	PNG	130	113 (-13,08 %)	111 (-1,77 %)
7	RegExp	200	176 (-12,00 %)	174 (-1,14 %)
8	AMS	4303	4064 (-5,55 %)	4062 (-0,05 %)

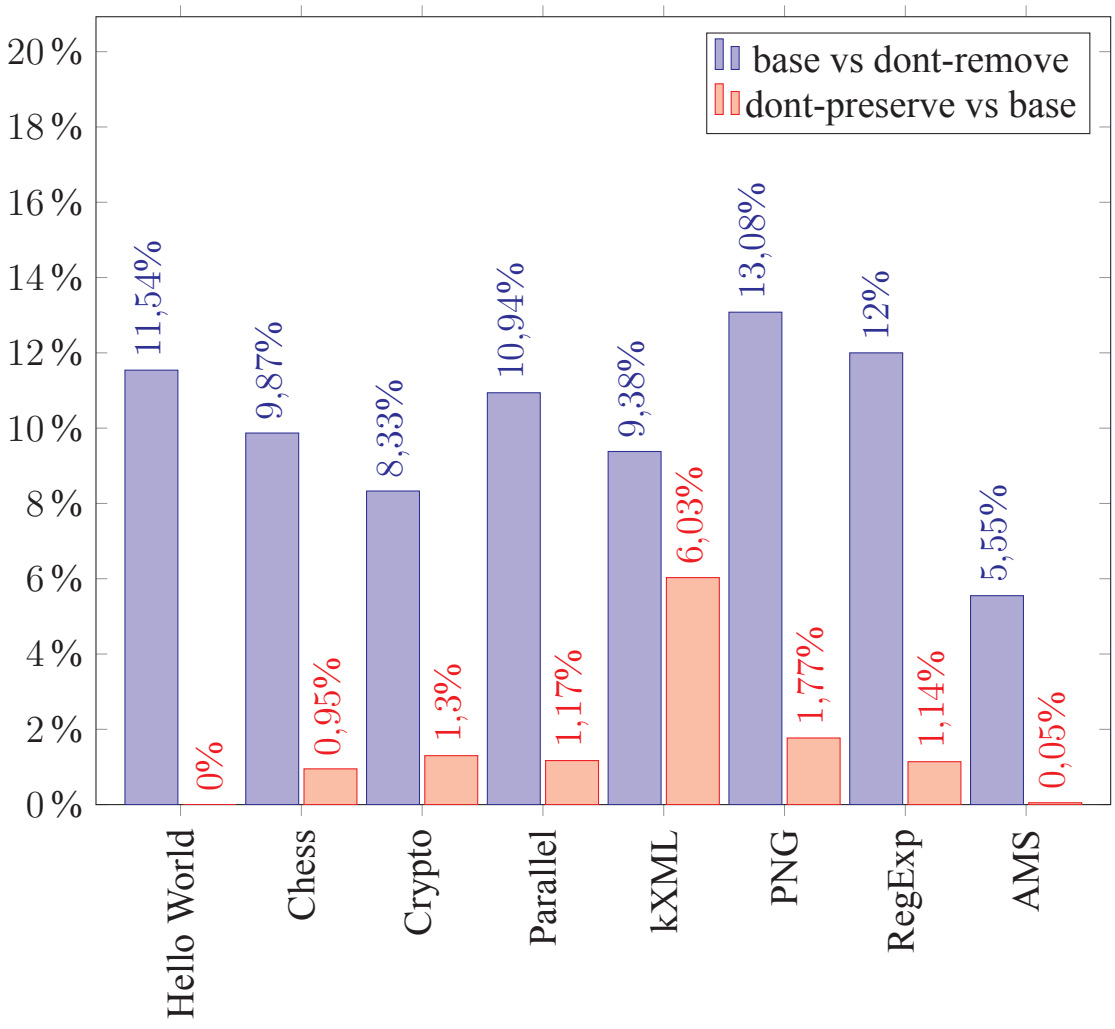


Рисунок 4.6 — Эксперимент 4. Относительное сокращение количества полей в образе

Таблица 7 — Эксперимент 5. Количество native-методов и их зависимостей

	Конфигурация	Методов	Методов с зависимостями	Зависимостей
1	CLDC	93	5	5
2	MEEP	933	309	689

По результатам проведенных экспериментов можно сделать следующие выводы:

1. Применение предложенных алгоритмов понижения избыточности в закрытой модели сокращает размер образа на 57,62–95,18 % (среднее значение 74,57 %).
2. Применение предложенного алгоритма анализа достижимости методов сокращает размер образа на 29,57–50,86 % (среднее значение 40,88 %) по сравнению с алгоритмом CHA.
3. Применение выборочной инициализации в процессе анализа достижимости методов сокращает размер образа на 0,3–11,71 % (среднее значение 4,27 %) по сравнению с безусловной инициализацией всех классов, применяемой в системе ExoVM.
4. Удаление инициализируемых, но неиспользуемых полей с помощью предложенного алгоритма сокращает количество полей в образе на 5,55–13,08 % (среднее значение 10,09 %) и сокращает размер образа на 0,64–4,37 % (среднее значение 1,32 %). Удаление всех инициализируемых, но неиспользуемых полей без учета семантики финализации приводит к незначительному сокращению количества полей (среднее значение 1,55 %) и размера образа (среднее значение 0,02 %) по сравнению с предложенным алгоритмом.
5. При большом количестве native-методов применение автоматического анализа межъязыковых зависимостей существенно упрощает описание дополнительных зависимостей.

4.4 Сравнение алгоритмов сжатия

Сравним алгоритм сжатия байт-кода, предложенный в рамках данной работы, с алгоритмами, рассмотренными в главе 1. Алгоритмы, описанные в [36; 37],

используют специализированные инструкции для компактного кодирования последовательностей исходных инструкций. Предложенный алгоритм использует специализированные инструкции для кодирования шаблонов последовательностей оригинальных инструкций. Шаблоны в предложенном алгоритме могут быть параметризованы значением аргументов некоторых из входящих в шаблон инструкций. Параметризованные шаблоны позволяют добиться более эффективного сжатия.

В отличие от алгоритма, описанного в [36], предложенный алгоритм использует код интерпретатора для реализации новых инструкций. С одной стороны, это позволяет избежать накладных расходов при исполнении специализированных инструкций. С другой стороны, такой подход увеличивает размер интерпретатора.

Впервые свертка и укорачивание аргументов для Java байт-кода были описаны в [27]. Эти оптимизации было предложено использовать для более компактного кодирования индивидуальных инструкций. Авторы не рассматривали взаимодействия между оптимизациями кодирования индивидуальных инструкций и сверткой последовательностей инструкций. Специализированные версии индивидуальных инструкций сокращают набор свободных опкодов, доступных для кодирования последовательностей инструкций. Алгоритм, предложенный соискателем, осуществляет свертку и укорачивание аргументов в процессе построения словаря шаблонов, что позволяет осуществлять эти оптимизации в контексте других инструкций.

Алгоритм, описанный в [38], использует специализированные инструкции для кодирования шаблонов последовательностей исходных инструкций. Параметрами шаблонов могут быть как аргументы инструкций, так и сами инструкции. Аналогично предложенному алгоритму, шаблоны могут фиксировать произвольный набор аргументов, что позволяет сворачивать аргументы. Сворачивание аргументов осуществляется в процессе построения словаря шаблонов.

Алгоритм, предложенный в рамках данной работы, не позволяет параметризовать опкоды инструкций в шаблонах, но может быть расширен такой поддержкой. Использование более сложных шаблонов позволяет добиться более высокой степени сжатия, однако исполнение специализированных инструкций, кодирующих такие шаблоны, сопряжено с дополнительными накладными расходами. Использование более простых шаблонов в предложенном алгоритме позволяет упростить исполнение специализированных инструкций и ускоряет

процесс построения словаря шаблонов. В отличие от предложенного алгоритма, алгоритм, описанный в [38], не укорачивает аргументы.

Упрощение исходного набора инструкций было предложено в [40] для RISC набора инструкций виртуальной машины OmniVM. В статье делается вывод, что удаление избыточных инструкций из исходного набора инструкций увеличивает эффективность словарного сжатия. Набор инструкций Java байт-кода не минимален и содержит инструкции, которые могут быть выражены через другие инструкции. Ни один из рассмотренных алгоритмов сжатия Java байт-кода не упрощает исходный набор инструкций перед сжатием. Алгоритм, реализованный в данной работе, упрощает исходный набор инструкций путем замены избыточных инструкций в коде программы эквивалентными последовательностями базовых инструкций. Экспериментальные данные подтверждают эффективность такого преобразования для Java байт-кода.

В отличие от многих рассмотренных алгоритмов, предложенный алгоритм не хранит множество шаблонов в памяти. Вместо этого в памяти хранится множество последовательностей. Множество шаблонов восстанавливается по множеству последовательностей. Такая организация словаря шаблонов позволяет сократить потребление памяти.

4.5 Экспериментальная оценка алгоритма сжатия

Для оценки эффективности реализованного алгоритма сжатия был проведен ряд численных экспериментов. В качестве характеристики эффективности сжатия использовалась степень сжатия, вычисляемая по формуле

$$(1 - \frac{compressed\_size}{original\_size}) * 100\ \%,$$

где $original\_size$ – суммарный размер оригинального байт-кода и интерпретатора, $compressed\_size$ – суммарный размер нового байт-кода и интерпретатора, необходимого для его исполнения.

Сформулируем задачи для экспериментов.

1. **Определить, какую степень сжатия обеспечивает реализованный алгоритм в зависимости от** $max\_pattern\_length$**.**

2. **Определить, как свертка и укорачивание аргументов в контексте последовательностей инструкций влияют на степень сжатия.** В отличие от существующих алгоритмов сжатия Java байт-кода, реализованный алгоритм осуществляет свертку и укорачивание аргументов в контексте последовательностей инструкций.
3. **Определить, как упрощение исходного набора инструкций влияет на степень сжатия.** Реализованный алгоритм впервые использует упрощение исходного набора инструкций Java байт-кода перед основным алгоритмом сжатия.
4. **Определить, какую степень сжатия обеспечивает реализованный алгоритм в открытой модели.**

Для измерений были использованы следующие наборы классов:

– java.lang.*, java.util.* – реализации стандартных пакетов Java из OpenJDK версии 7u17 [109].
– CLDC – Java-классы реализации стандарта Connected Limited Device Configuration 8 (JSR360).
– MEEP – Java-классы реализации стандарта Java ME Embedded Profile (JSR361).
– EEMBC – EEMBC GrinderBench – набор тестов для измерения производительности Java ME платформ.
– Scala Library – стандартная библиотека языка Scala версии 2.10.3 [110].
– Kotlin Runtime – подмножество стандартной библиотеки языка Kotlin версии 0.6.13, написанное на языке Kotlin [111].

Репрезентативность данного набора классов обеспечивается следующими фактами:

– Классы в пакетах `java.lang.*` и `java.util.*` являются подмножеством стандартной библиотеки классов Java SE. Классы CLDC и MEEP являются классами стандартной библиотеки Java-платформы для встраиваемых устройств. Виртуальная машина, специализированная для заданного приложения с использованием вышеописанных алгоритмов понижения избыточности, будет включать в себя подмножество классов стандартной библиотеки.
– Набор классов EEMBC включает в себя приложения Chess, Crypto, Parallel, kXML, PNG, RegExp из набора приложений использованных для

анализа эффективности алгоритмов понижения избыточности. Информация о данных приложениях представлена в таблице 1.

– Наборы классов Scala Library и Kotlin Runtime отличаются тем, что они скомпилированы не компилятором javac, а компиляторами соответствующих языков. Такие компиляторы могут генерировать шаблоны инструкций, отличающиеся от шаблонов, порождаемых компилятором javac.

Информация об исходном размере байт-кода и интерпретатора, необходимого для его выполнения, для использованных наборов классов приведена в таблице 8. Для вычисления степени сжатия в экспериментах измерялся размер байт-кода и соответствующего интерпретатора после специализации набора инструкций.

Эксперимент 1. Какую степень сжатия обеспечивает реализованный алгоритм в зависимости от $max\_pattern\_length$?

В таблице 9 приведены значения степени сжатия в зависимости от максимальной длины шаблона $max\_pattern\_length$. Стоит отметить, что при значении $max\_pattern\_length = 1$ осуществляются только свертка и укорачивание аргументов. Средняя степень сжатия при этом равна 18,98 %. В практических целях может быть оправдано использование только этих оптимизаций. Дорогостоящего построения словаря последовательностей для них не требуется.

Добавление к этим оптимизациям сворачивания последовательностей инструкций увеличивает степень сжатия почти в два раза. Средняя степень сжатия при $max\_pattern\_length = 2$ равна 36,18 %. Дальнейшее увеличение максимальной длины шаблона незначительно увеличивает степень сжатия. При значениях

Таблица 8 — Размер байт-кода и интерпретатора, необходимого для его выполнения, в байтах

Набор классов	Байт-код	Интерпретатор	Суммарный размер
java.lang.*	180419	5998	186417
java.util.*	308665	5820	314485
CLDC	29742	6043	35785
MEEP	105124	6353	111477
EEMBC	110320	3423	113743
Scala Library	747332	6576	753908
Kotlin Runtime	56158	3812	59970

больших 5 эффективность сжатия практически не растет. При использовании данного алгоритма на практике можно рекомендовать ограничивать максимальную длину шаблона пятью инструкциями. Для дальнейших измерений использовалось значение $max\_pattern\_length = 5$.

Эксперимент 2. Как свертка и укорачивание аргументов в контексте последовательностей инструкций влияют на степень сжатия?

В таблицах 10, 11 и 12 приведены значения степени сжатия в зависимости от того, какие варианты инструкций шаблона порождаются для инструкции с аргументом. Сворачивание аргументов на рассматриваемых примерах увеличивает степень сжатия в среднем на 22,28 %. Укорачивание аргументов в шаблонах увеличивает степень сжатия еще на 22,33 %. Сворачивание и укорачивание аргументов в совокупности значительно увеличивают степень сжатия (в среднем на 49,56 %). График на рисунке 4.7 демонстрирует относительное увеличение степени сжатия при добавлении свертки и укорачивания аргументов к свертке последовательностей инструкций.

Эксперимент 3. Как упрощение исходного набора инструкций влияет на степень сжатия?

В таблицах 13 и 14 приведены значения степени сжатия в зависимости от того, был ли упрощен исходный набор инструкций. Относительное увеличение степени сжатия при упрощении исходного набора инструкций показано на графике на рисунке 4.8. Относительное увеличение степени сжатия при упрощении исходного набора инструкций варьируется в диапазоне от 1,33 до 14,4 %. Из этого можно сделать вывод о целесообразности упрощения набора инструкций стандартного Java байт-кода перед применением словарного сжатия.

Эксперимент 4. Какую степень сжатия обеспечивает реализованный алгоритм в открытой модели?

В таблицах 15 и 16 приведены значения степени сжатия в зависимости от того, совместим ли генерируемый байт-код со стандартным набором инструкций. При генерации совместимого набора инструкций для новых инструкций доступно меньшее число кодов, в результате чего эффективность сжатия падает в среднем на 18,14 %. Достаточно высокая степень сжатия при генерации совместимого набора инструкций (в среднем 31,54 %) делает оправданным использование алгоритма для оптимизации размера стандартных библиотек в открытой модели. График на рисунке 4.9 демонстрирует относительное увеличение степени сжатия при генерации несовместимого набора инструкций.

По результатам проведенных экспериментов можно сделать следующие выводы:

1. Среднее значение степени сжатия реализованного алгоритма варьируется в диапазоне от 18,98 % ($max\_pattern\_length = 1$) до 38,76 % ($max\_pattern\_length = 6$).
2. Сворачивание аргументов в контексте последовательностей инструкций увеличивает степень сжатия в среднем на 22,28 %. Контекстное укорачивание аргументов в шаблонах увеличивает степень сжатия еще на 22,33 %. Контекстное сворачивание и укорачивание аргументов в совокупности увеличивают степень сжатия в среднем на 49,56 % ($max\_pattern\_length = 5$).
3. Относительное увеличение степени сжатия при упрощении исходного набора инструкций варьируется в диапазоне от 1,33 до 14,4 % ($max\_pattern\_length = 5$).
4. Среднее значение степени сжатия реализованного алгоритма в открытой модели в составляет 31,54 % ($max\_pattern\_length = 5$).

4.6 Ограничения

Можно выделить следующие ограничения предложенных алгоритмов:

– Алгоритм анализа достижимости методов не поддерживает анализ инструкции `invokedynamic`.
– Для выбора классов, которые можно инициализировать в процессе подготовки образа, используется очень консервативный анализ. Текущий анализ не позволяет автоматически инициализировать классы, которые создают новые объекты. Для инициализации таких классов необходимо использовать аннотацию `@InitAtBuild`.
– Реализованный алгоритм понижения избыточности требует ручного описания зависимостей рефлексивного кода.
– Эффективность алгоритмов понижения избыточности в закрытой модели была измерена на ограниченном наборе приложений. Ромизация в закрытой модели была реализована только для платформы CLDC, что существенно ограничило набор доступных приложений.

4.7 Направления дальнейшего исследования

Отметим следующие направления для дальнейшего исследования.

– Реализация поддержки инструкции `invokedynamic` в алгоритме анализа достижимости методов.
– Реализация менее консервативного анализа безопасности инициализации класса. Существующий анализ может быть расширен анализом вызовов в инициализаторах классов. Для определения порядка инициализации классов возможно применение анализа из статьи [66].
– Использование межпроцедурного анализа указателей для анализа достижимости методов. Алгоритм, описанный в разделе 2.5, построен на базе алгоритма RTA и расширяет его выборочной инициализацией классов. В статье [42] соискателем предложен оптимистичный алгоритм анализа достижимости методов, использующий межпроцедурный анализ указателей. Данный алгоритм может быть расширен выборочной инициализацией классов аналогичным образом. Использование более точного анализа достижимости методов увеличит эффективность предложенных алгоритмов понижения избыточности.
– Автоматический анализ зависимостей рефлексивного кода. Использование межпроцедурного анализа указателей позволит реализовать автоматический анализ зависимостей для рефлексивного кода аналогично тому, как предложено в статьях [70; 72].
– Измерение влияния предложенного критерия удалимости полей на динамическое потребление памяти.
– Текущая реализация алгоритма сжатия использует однобайтовое кодирование инструкций, что позволяет закодировать 256 инструкций. Реализация многобайтового кодирования переменной длины позволит увеличить количество доступных опкодов, что увеличит размер словаря, используемого для сжатия.
– Алгоритм перебора шаблонов может быть дополнен поддержкой шаблонов, параметризованных инструкциями. Интересно сравнение такого алгоритма сжатия с алгоритмом, описанным в работе Saougkos [38].

4.8 Выводы

В четвертой главе приведено описание программной реализации алгоритмов, предложенных в главах 2 и 3. Проведено сравнение предложенных алгоритмов с существующими решениями. На основании данного сравнения сформулированы задачи для экспериментального исследования реализации. Подробные выводы по результатам экспериментов представлены в разделах 4.3, 4.5. В частности, экспериментальным путем продемонстрировано:

1. Выборочная инициализация в реализованном анализе достижимости методов сохраняет семантику инициализации и сокращает размер образа на 0,3–11,71 % (среднее значение 4,27 %) по сравнению с безусловной инициализацией.
2. Реализованный алгоритм анализа удалимости полей сокращает количество полей в образе на 5,55–13,08 % (среднее значение 10,09 %) по сравнению с консервативной реализацией, которая не удаляет инициализируемые, но неиспользуемые поля. Сокращение размера образа при этом составляет 0,64–4,37 % (среднее значение 1,32 %).
3. Сворачивание аргументов в контексте последовательностей инструкций увеличивает степень сжатия в среднем на 22,28 %. Контекстное укорачивание аргументов в шаблонах увеличивает степень сжатия еще на 22,33 %. Контекстное сворачивание и укорачивание аргументов в совокупности увеличивают степень сжатия в среднем на 49,56 %.

В данной главе решена 5-я задача диссертации: «Реализовать предложенные алгоритмы и экспериментальным путем измерить их эффективность». На основании экспериментов можно сделать вывод, что применение предложенных алгоритмов при специализации Java-платформы для заданного приложения в закрытой модели позволяет сократить аппаратные требования платформы, не нарушая поведения данного приложения.

Таблица 9 — Эксперимент 1. Степень сжатия в зависимости от $max\_pattern\_length$

	1	2	3	4	5	6
java.lang.*						
Размер байт-кода	144536	113893	106529	104406	102825	101353
Размер интерпретатора	8036	10103	11050	11596	11849	12118
Суммарный размер	152572	123996	117579	116002	114674	113471
Степень сжатия	18,16	33,48	36,93	37,77	38,49	39,13
java.util.*						
Байт-код	253855	202514	200380	200681	200426	200252
Интерпретатор	7799	8734	9012	9250	9391	9502
Суммарный размер	261654	211248	209392	209931	209817	209754
Степень сжатия	16,80	32,83	33,42	33,25	33,28	33,30
CLDC						
Байт-код	22784	19692	19516	19388	19388	19388
Интерпретатор	7727	7625	7734	7724	7724	7724
Суммарный размер	30511	27317	27250	27112	27112	27112
Степень сжатия	14,74	23,66	23,85	24,24	24,24	24,24
MEEP						
Байт-код	81476	64110	63255	62816	62736	62736
Интерпретатор	8131	8718	8776	8872	8923	8923
Суммарный размер	89607	72828	72031	71688	71659	71659
Степень сжатия	19,62	34,67	35,38	35,69	35,72	35,72
EEMBC						
Байт-код	81726	59147	57728	56717	54196	54186
Интерпретатор	6221	8376	8470	8793	9303	9083
Суммарный размер	87947	67523	66198	65510	63499	63269
Степень сжатия	22,68	40,64	41,80	42,41	44,17	44,38
Scala Library						
Байт-код	597303	407115	373200	370389	370776	368637
Интерпретатор	8387	10074	11714	12112	12582	13046
Суммарный размер	605690	417189	384914	382501	383358	381683
Степень сжатия	19,66	44,66	48,94	49,26	49,15	49,37
Kotlin Runtime						
Байт-код	40812	27003	25402	25258	25052	25237
Интерпретатор	6459	7007	7660	7707	7714	7625
Суммарный размер	47271	34010	33062	32965	32766	32862
Степень сжатия	21,18	43,29	44,87	45,03	45,36	45,20
Средняя степень сжатия	18,98	36,18	37,88	38,24	38,63	38,76

Таблица 10 — Эксперимент 2. Степень сжатия при использовании параметризованных шаблонов ($p_{max} = 1$)

Набор классов	Байт-код	Интерпретатор	Суммарный размер	Степень сжатия
java.lang.*	115548	13076	128624	31,00
java.util.*	216990	11340	228330	27,40
CLDC	26297	6406	32703	8,61
MEEP	77383	9569	86952	22,00
EEMBC	69504	7562	77066	32,25
Scala Library	520588	13760	534348	29,12
Kotlin Runtime	36594	5147	41741	30,40
Средняя степень сжатия				25,83

Таблица 11 — Эксперимент 2. Степень сжатия при использовании параметризованных шаблонов со сворачиванием аргументов ($p_{max} = 2$)

Набор классов	Байт-код	Интерпретатор	Суммарный размер	Степень сжатия
java.lang.*	109245	12266	121511	34,82 (+12,31 %)
java.util.*	215511	9646	225157	28,40 (+3,68 %)
CLDC	22360	7854	30214	15,57 (+80,76 %)
MEEP	74680	9098	83778	24,85 (+12,94 %)
EEMBC	61127	9376	70503	38,02 (+17,89 %)
Scala Library	448089	12857	460946	38,86 (+33,43 %)
Kotlin Runtime	27874	7782	35656	40,54 (+33,38 %)
Средняя степень сжатия				31,58 (+22,28 %)

Таблица 12 — Эксперимент 2. Степень сжатия при использовании параметризованных шаблонов со сворачиванием и укорачиванием аргументов ($p_{max} = 3$)

Набор классов	Байт-код	Интерпретатор	Суммарный размер	Степень сжатия
java.lang.*	102825	11849	114674	38,49 (+10,53 %)
java.util.*	200426	9391	209817	33,28 (+17,17 %)
CLDC	19388	7724	27112	24,24 (+55,68 %)
MEEP	62736	8923	71659	35,72 (+43,75 %)
EEMBC	54196	9303	63499	44,17 (+16,20 %)
Scala Library	370776	12582	383358	49,15 (+26,48 %)
Kotlin Runtime	25052	7714	32766	45,36 (+11,89 %)
Средняя степень сжатия				38,63 (+22,33 %)

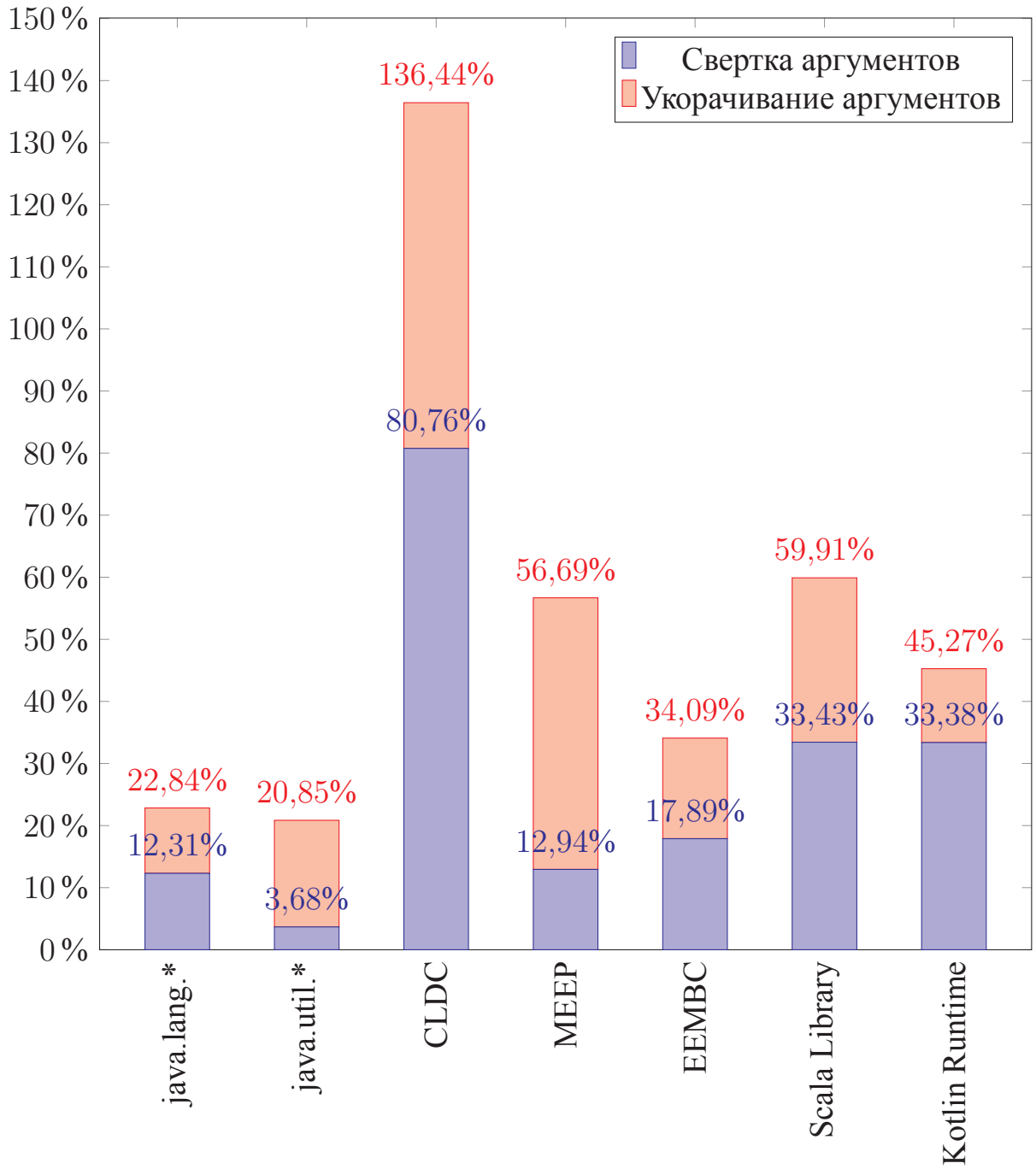


Рисунок 4.7 — Эксперимент 2. Относительное увеличение степени сжатия при использовании свертки и укорачивания аргументов

Таблица 13 — Эксперимент 3. Степень сжатия при использовании стандартного набора инструкций

Набор классов	Байт-код	Интерпретатор	Суммарный размер	Степень сжатия
java.lang.*	106445	11395	117840	36,79
java.util.*	214087	8907	222994	29,09
CLDC	19349	8283	27632	22,78
MEEP	66787	8744	75531	32,25
EEMBC	54599	9561	64160	43,59
Scala Library	398165	11522	409687	45,66
Kotlin Runtime	25524	8153	33677	43,84
Средняя степень сжатия				36,29

Таблица 14 — Эксперимент 3. Степень сжатия при использовании упрощенного набора инструкций

Набор классов	Байт-код	Интерпретатор	Суммарный размер	Степень сжатия
java.lang.*	102825	11849	114674	38,49 (+4,62 %)
java.util.*	200426	9391	209817	33,28 (+14,40 %)
CLDC	19388	7724	27112	24,24 (+6,38 %)
MEEP	62736	8923	71659	35,72 (+10,77 %)
EEMBC	54196	9303	63499	44,17 (+1,33 %)
Scala Library	370776	12582	383358	49,15 (+7,65 %)
Kotlin Runtime	25052	7714	32766	45,36 (+3,46 %)
Средняя степень сжатия				38,63 (+6,46 %)

Таблица 15 — Эксперимент 4. Степень сжатия при генерации совместимого набора инструкций

Набор классов	Байт-код	Интерпретатор	Суммарный размер	Степень сжатия
java.lang.*	124936	8164	133100	28,60
java.util.*	232642	6427	239069	23,98
CLDC	21112	6590	27702	22,59
MEEP	72252	6967	79219	28,94
EEMBC	63275	5771	69046	39,30
Scala Library	444631	8771	453402	39,86
Kotlin Runtime	32463	5008	37471	37,52
Средняя степень сжатия				31,54

Таблица 16 — Эксперимент 4. Степень сжатия при генерации несовместимого набора инструкций

Набор классов	Байт-код	Интерпретатор	Суммарный размер	Степень сжатия
java.lang.*	102825	11849	114674	38,49 (+34,56 %)
java.util.*	200426	9391	209817	33,28 (+38,79 %)
CLDC	19388	7724	27112	24,24 (+7,30 %)
MEEP	62736	8923	71659	35,72 (+23,44 %)
EEMBC	54196	9303	63499	44,17 (+12,41 %)
Scala Library	370776	12582	383358	49,15 (+23,31 %)
Kotlin Runtime	25052	7714	32766	45,36 (+20,91 %)
Средняя степень сжатия				38,63 (+22,48 %)

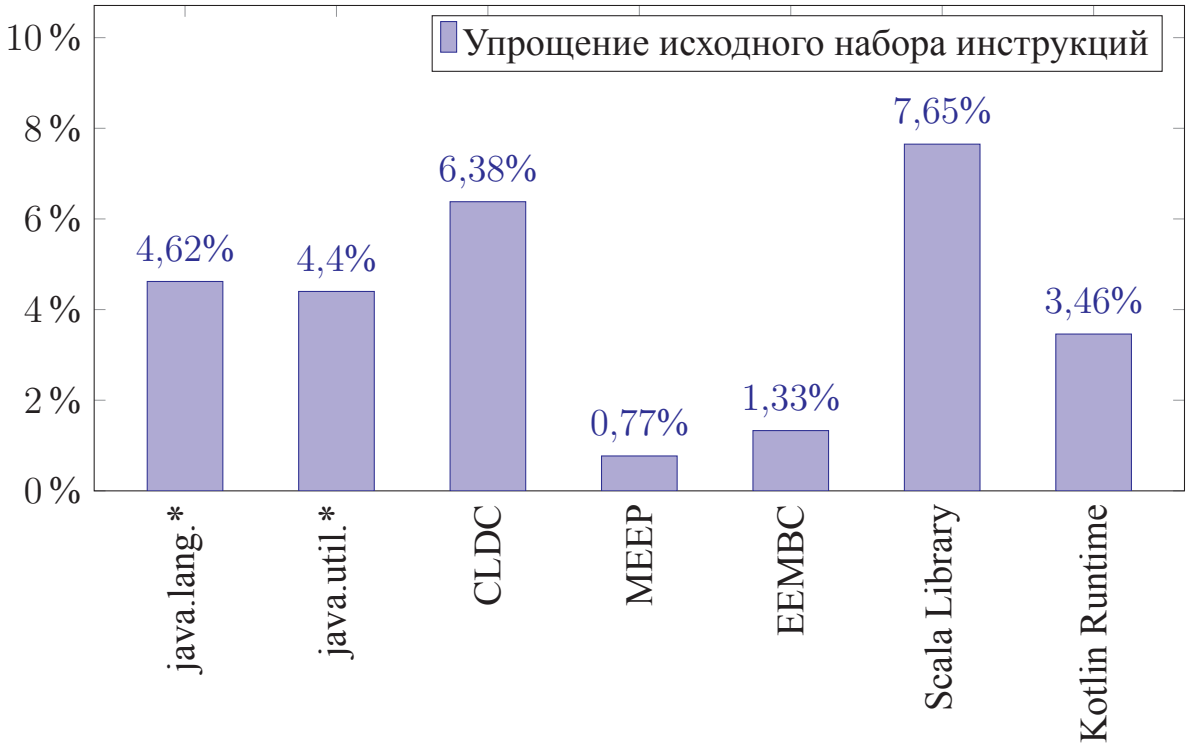


Рисунок 4.8 — Эксперимент 3. Относительное увеличение степени сжатия при упрощении исходного набора инструкций

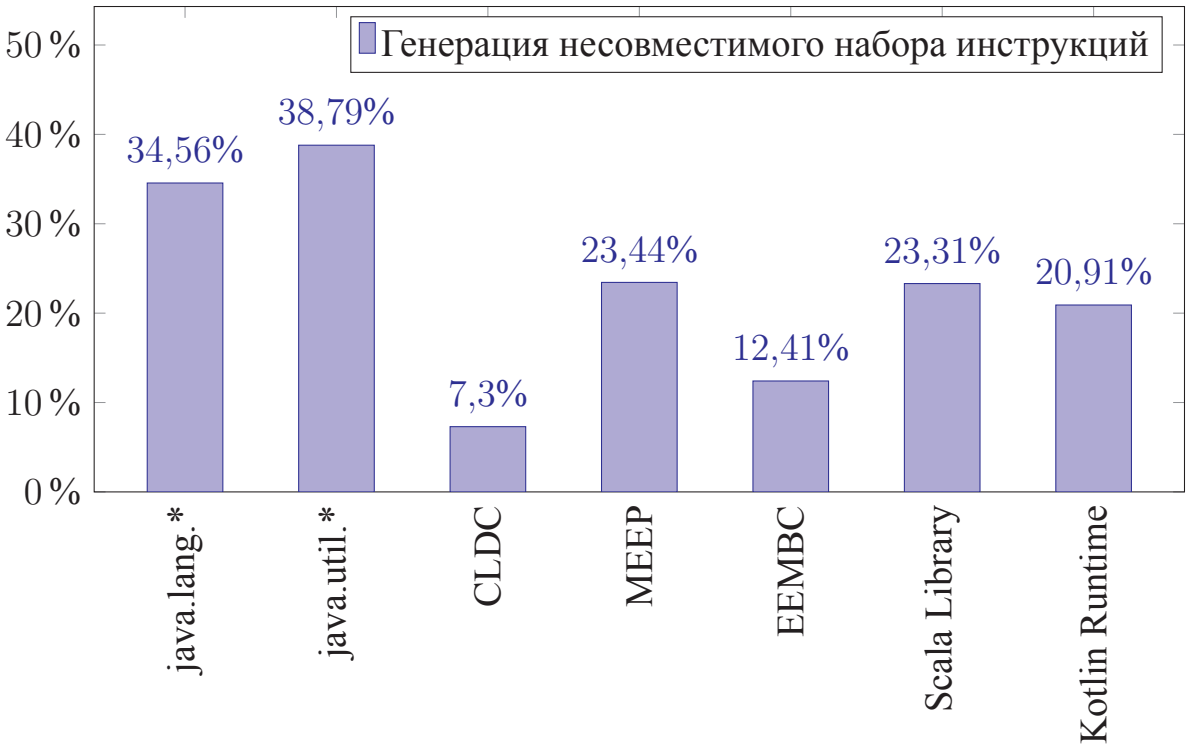


Рисунок 4.9 — Эксперимент 4. Относительное увеличение степени сжатия при генерации несовместимого набора инструкций

Заключение

В работе решена важная задача по сокращению аппаратных требований Java-плафтормы при исполнении заданного приложения в закрытой модели:

1. Существующие алгоритмы понижения избыточности программ изучены с точки зрения их применимости при использовании раздельной инициализации. Обозначены ограничения, которые не позволяют применять их при использовании раздельной инициализации. Отмечено, что существующие алгоритмы понижения избыточности требуют ручного описания межъязыковых зависимостей между кодом на Java и C++.
2. Исследованы существующие алгоритмы сжатия бинарного кода. Изучена их применимость для сжатия Java байт-кода в закрытой модели. Обозначены оптимизации, применяемые при сжатии с помощью специализации набора инструкций. Отмечено, что существующие механизмы сжатия Java байт-кода не используют некоторые из описанных оптимизаций. Например, ни один из ранее описанных механизмов не осуществляет упрощение исходного набора инструкций перед сжатием.
3. Разработаны алгоритмы понижения избыточности Java-программ, применимые при раздельной инициализации:
    а) алгоритм анализа Java-программ, определяющий достижимость методов Java-классов и осуществляющий выборочную инициализацию используемых классов;
    б) алгоритм анализа Java-программ, определяющий удалимость полей Java-классов и позволяющий удалять инициализируемые, но неиспользуемые поля без нарушения семантики финализации;
    в) алгоритм анализа Java-программ, определяющий удалимость Java-классов.

    Описанные алгоритмы понижения избыточности используют оригинальный метод анализа программ, автоматически выявляющий межъязыковые зависимости между кодом на языках Java и C++.

4. Разработан алгоритм сжатия Java байт-кода в закрытой модели, применимый для встраиваемых систем с ограниченными ресурсами. Предложенный алгоритм специализирует набор инструкций и порождает компактное исполняемое представление.
5. Предложенные алгоритмы реализованы на практике. Экспериментальным путем продемонстрировано:
    а) Выборочная инициализация в реализованном анализе достижимости методов сохраняет семантику инициализации и сокращает размер образа на 0,3–11,71 % (среднее значение 4,27 %) по сравнению с безусловной инициализацией.
    б) Реализованный алгоритм анализа удалимости полей сокращает количество полей в образе на 5,55–13,08 % (среднее значение 10,09 %) по сравнению с консервативной реализацией, которая не удаляет инициализируемые, но неиспользуемые поля. Сокращение размера образа при этом составляет 0,64–4,37 % (среднее значение 1,32 %).
    в) Сворачивание аргументов в контексте последовательностей инструкций увеличивает степень сжатия в среднем на 22,28 %. Контекстное укорачивание аргументов в шаблонах увеличивает степень сжатия еще на 22,33 %. Контекстное сворачивание и укорачивание аргументов в совокупности увеличивают степень сжатия в среднем на 49,56 %.

Применение разработанных и реализованных алгоритмов при специализации Java-платформы для заданного приложения в закрытой модели позволяет сократить аппаратные требования платформы на нарушая поведения приложения. Сокращение аппаратных требований позволяет расширить область применения языка Java на более ограниченные в ресурсах устройства, что позволяет повысить эффективность процесса разработки и надежность программного обеспечения для таких устройств.

Рекомендации по применению результатов работы:

1. При разработке языков программирования для встраиваемых применений необходимо учитывать, что ленивое связывание и инициализация затрудняют оптимизации при раздельной инициализации. Строго специфицированная семантика достижимости объектов совместно с

возможностью программно отследить потерю достижимости объектов затрудняют удаление неиспользуемых полей.

2. Использование динамического связывания затрудняет статический анализ зависимостей для понижения избыточности. При разработке интерфейсов межъязыкового взаимодействия рекомендуется использовать статическое связывание.
3. Возможность отследить удаление объектов произвольных типов с помощью специальных ссылок представляет собой еще один пример динамического поведения, статический анализ которого затруднителен. Анализ удалимости полей в присутствии специальных ссылок требует реализации межпроцедурного анализа указателей. Данный анализ может быть существенно упрощен, если приложения не используют специальные ссылки.
4. Разработанные алгоритмы могут быть использованы в открытой модели для сокращения размера классов системных библиотек.

Перспективы дальнейшей разработки темы:

1. Использование межпроцедурного анализа указателей для анализа достижимости методов и анализа кода, использующего рефлексию.
2. Исследование возможности переноса большего количества операций по инициализации приложения на этап подготовки образа.
3. Реализация многобайтового кодирования переменной длины в специализированном интерпретаторе.
4. Использование более сложных шаблонов для кодирования специализированными инструкциями.

Список литературы

1. *Кияев*, *В. И.* Стандартизация, метрология и качество разработки программного обеспечения и информационных технологий / В. И. Кияев. — СПб : Изд-во Санкт-Петербургского гос. экономического ун-та, 2016. — 475 с.
2. *Korzun*, *D. G.* Deployment of Smart Spaces in Internet of Things: Overview of the Design Challenges / D. G. Korzun, S. I. Balandin, A. V. Gurtov // Internet of Things, Smart Spaces, and Next Generation Networking / под ред. S. Balandin, S. Andreev, Y. Koucheryavy. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. — С. 48—59.
3. *Сафронов*, *А. Ю.* Среда разработки программного обеспечения для встраиваемых систем на основе JME / А. Ю. Сафронов, Д. Е. Намиот // International Journal of Open Information Technologies. — 2013. — Т. 1, № 2. — С. 17—24.
4. *Higuera-Toledano*, *M. T.* Java Embedded Real-Time Systems: An Overview of Existing Solutions / M. T. Higuera-Toledano, V. Issarny // Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. — Washington, DC, USA : IEEE Computer Society, 2000. — С. 392—. — (ISORC ’00).
5. Providing an Embedded Software Environment for Wireless PDAs / V. Issarny [и др.] // Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System. — Kolding, Denmark : ACM, 2000. — С. 49—54. — (EW 9).
6. *Holgado-Terriza*, *J. A.* A Flexible Java Framework for Embedded Systems / J. A. Holgado-Terriza, J. Viúdez-Aivar // Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems. — Madrid, Spain : ACM, 2009. — С. 21—30. — (JTRES ’09).
7. Making Object Oriented Efficient for Embedded System Applications / J. C. B. Mattos [и др.] // Proceedings of the 18th Annual Symposium on Integrated Circuits and System Design. — Florianolpolis, Brazil : ACM, 2005. — С. 104—109. — (SBCCI ’05).

8. An Approach for a Dependable Java Embedded Environment / G. Cabillic [и др.] // Proceedings of the 10th Workshop on ACM SIGOPS European Workshop. — Saint-Emilion, France : ACM, 2002. — С. 157—160. — (EW 10).

9. *Barry*, *B.* Embedded Java (Embedded Tutorial) (Abstract Only): Techniques and Applications / B. Barry, J. Duimovich // Proceedings of the 1999 IEEE/ACM International Conference on Computer-aided Design. — San Jose, California, USA : IEEE Press, 1999. — С. 613—. — (ICCAD '99). — Chairman-Bergamaschi, Reinaldo.

10. *Griffin*, *P.* An Energy Efficient Garbage Collector for Java Embedded Devices / P. Griffin, W. Srisa-an, J. M. Chang // Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. — Chicago, Illinois, USA : ACM, 2005. — С. 230—238. — (LCTES '05).

11. An Optimized Java Interpreter for Connected Devices and Embedded Systems / A. Beatty [и др.] // Proceedings of the 2003 ACM Symposium on Applied Computing. — Melbourne, Florida : ACM, 2003. — С. 692—697. — (SAC '03).

12. Exploiting Static Application Knowledge in a Java Compiler for Embedded Systems: A Case Study / C. Erhardt [и др.] // Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems. — York, United Kingdom : ACM, 2011. — С. 96—105. — (JTRES '11).

13. *Srivastava*, *A.* Unreachable Procedures in Object-oriented Programming / A. Srivastava // ACM Lett. Program. Lang. Syst. — New York, NY, USA, 1992. — Дек. — Т. 1, № 4. — С. 355—364.

14. Slicing Class Hierarchies in C++ / F. Tip [и др.] // SIGPLAN Not. — New York, NY, USA, 1996. — Окт. — Т. 31, № 10. — С. 179—197.

15. *Bacon*, *D. F.* Fast Static Analysis of C++ Virtual Function Calls / D. F. Bacon, P. F. Sweeney // SIGPLAN Not. — New York, NY, USA, 1996. — Окт. — Т. 31, № 10. — С. 324—341.

16. *Tip*, *F.* Class Hierarchy Specialization / F. Tip, P. F. Sweeney // SIGPLAN Not. — New York, NY, USA, 1997. — Окт. — Т. 32, № 10. — С. 271—285.

17. *Sweeney*, *P. F.* A Study of Dead Data Members in C++ Applications / P. F. Sweeney, F. Tip // SIGPLAN Not. — New York, NY, USA, 1998. — Май. — Т. 33, № 5. — С. 324—332.

18. *Tip*, *F.* Extracting Library-based Java Applications / F. Tip, P. F. Sweeney, C. Laffra // Commun. ACM. — New York, NY, USA, 2003. — Авг. — Т. 46, № 8. — С. 35—40.

19. *Sweeney*, *P. F.* Extracting Library-based Object-oriented Applications / P. F. Sweeney, F. Tip // SIGSOFT Softw. Eng. Notes. — New York, NY, USA, 2000. — Нояб. — Т. 25, № 6. — С. 98—107.

20. Practical Experience with an Application Extractor for Java / F. Tip [и др.] // SIGPLAN Not. — New York, NY, USA, 1999. — Окт. — Т. 34, № 10. — С. 292—305.

21. *Rayside*, *D.* Extracting Java Library Subsets for Deployment on Embedded Systems. / D. Rayside, K. Kontogiannis // CSMR. — IEEE Computer Society, 1999. — С. 102—110.

22. Tailor-made JVMs for statically configured embedded systems / M. Stilkerich [и др.] // Concurrency and Computation: Practice and Experience. — 2012. — Т. 24, № 8. — С. 789—812.

23. Application-Driven Customization of an Embedded Java Virtual Machine / A. Courbot [и др.] // Proceedings of the 2005 International Conference on Embedded and Ubiquitous Computing. — Nagasaki, Japan : Springer-Verlag, 2005. — С. 81—90. — (EUC'05).

24. *Wagner*, *G.* Slim VM: Optimistic Partial Program Loading for Connected Embedded Java Virtual Machines / G. Wagner, A. Gal, M. Franz // Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java. — Modena, Italy : ACM, 2008. — С. 117—126. — (PPPJ '08).

25. *Wagner*, *G.* "Slimming" a Java Virtual Machine by Way of Cold Code Removal and Optimistic Partial Program Loading / G. Wagner, A. Gal, M. Franz // Sci. Comput. Program. — Amsterdam, The Netherlands, The Netherlands, 2011. — Нояб. — Т. 76, № 11. — С. 1037—1053.

26. *Marquet*, *K.* Ahead of Time Deployment in ROM of a Java-OS / K. Marquet, A. Courbot, G. Grimaud // Proceedings of the Second International Conference on Embedded Software and Systems. — Xi'an, China : Springer-Verlag, 2005. — С. 63—70. — (ICESS'05).

27. Optimized Java Binary and Virtual Machine for Tiny Motes / F. Aslam [и др.] // Proceedings of the 6th IEEE International Conference on Distributed Computing in Sensor Systems. — Santa Barbara, CA : Springer-Verlag, 2010. — С. 15—30. — (DCOSS'10).

28. *Titzer, B. L.* Virgil: Objects on the Head of a Pin / B. L. Titzer // SIGPLAN Not. — New York, NY, USA, 2006. — Окт. — Т. 41, № 10. — С. 191—208.

29. The ExoVM System for Automatic VM and Application Reduction / B. L. Titzer [и др.] // SIGPLAN Not. — New York, NY, USA, 2007. — Июнь. — Т. 42, № 6. — С. 352—362.

30. Romization: Early Deployment and Customization of Java Systems for Constrained Devices / A. Courbot [и др.] // In Proceedings of Second International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. — 2005.

31. Compiler Techniques for Code Compaction / S. K. Debray [и др.] // ACM Trans. Program. Lang. Syst. — New York, NY, USA, 2000. — Март. — Т. 22, № 2. — С. 378—415.

32. *Chen, W.-K.* Code Compaction of Matching Single-entry Multiple-exit Regions / W.-K. Chen, B. Li, R. Gupta // Proceedings of the 10th International Conference on Static Analysis. — San Diego, CA, USA : Springer-Verlag, 2003. — С. 401—417. — (SAS'03).

33. *Debray, S.* Cold Code Decompression at Runtime / S. Debray, W. S. Evans // Commun. ACM. — New York, NY, USA, 2003. — Авг. — Т. 46, № 8. — С. 54—60.

34. *Thuresson, M.* Evaluation of Extended Dictionary-based Static Code Compression Schemes / M. Thuresson, P. Stenstrom // Proceedings of the 2Nd Conference on Computing Frontiers. — Ischia, Italy : ACM, 2005. — С. 77—86. — (CF '05).

35. *Lucco, S.* Split-stream Dictionary Program Compression / S. Lucco // Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. — Vancouver, British Columbia, Canada : ACM, 2000. — С. 27—34. — (PLDI '00).

36. Java Bytecode Compression for Low-end Embedded Systems / L. R. Clausen [и др.] // ACM Trans. Program. Lang. Syst. — New York, NY, USA, 2000. — Май. — Т. 22, № 3. — С. 471—489.

37. *Stefanov, E.* On the Implementation of Bytecode Compression for Interpreted Languages / E. Stefanov, A. M. Sloane // Softw. Pract. Exper. — New York, NY, USA, 2009. — Февр. — Т. 39, № 2. — С. 111—135.

38. Revisiting Java Bytecode Compression for Embedded and Mobile Computing Environments / D. Saougkos [и др.] // IEEE Trans. Softw. Eng. — Piscataway, NJ, USA, 2007. — Июль. — Т. 33, № 7. — С. 478—495.

39. *Fraser, C. W.* Custom Instruction Sets for Code Compression / C. W. Fraser, T. A. Proebsting. — 1995.

40. Code Compression / J. Ernst [и др.] // SIGPLAN Not. — New York, NY, USA, 1997. — Май. — Т. 32, № 5. — С. 358—365.

41. *Пилипенко, А. В.* Оптимизация представления байт-кода JVM для встраиваемых систем / А. В. Пилипенко // Материалы научной конференции по проблемам информатики СПИСОК-2013. — 2013. — С. 104—106.

42. *Пилипенко, А. В.* Анализ достижимости методов с помощью межпроцедурного анализа потока данных / А. В. Пилипенко, В. О. Сафонов // Материалы учебно-практической конференции студентов, аспирантов и молодых учёных Северо-Западного федерального округа ”Технологии Microsoft в теории и практике программирования (Новые подходы к разработке программного обеспечения на примере технологий Microsoft и EMC)”. — 2014. — С. 95—96.

43. Extremely Small Yet Powerful [Электронный ресурс]. — 2014. — URL: https://www.youtube.com/watch?v=Guikp8olXrk (дата обр. 23.02.2018).

44. *Пилипенко, А. В.* Использование межпроцедурного анализа потока данных для понижения избыточности Java-программ / А. В. Пилипенко, В. И. Кияев // Сборник научных статей международной научно-практической конференции ”Интеллектуальные и информационные технологии в формировании цифрового общества”. — 2017. — С. 58—60.

45. *Пилипенко, А. В.* Обзор интерпретации и компиляции в виртуальных машинах / А. В. Пилипенко // Компьютерные инструменты в образовании. — 2012. — № 3. — С. 3—15.

46. *Пилипенко*, *А. В.* Словарное сжатие байт-кода JVM с помощью специализации набора инструкций / А. В. Пилипенко, О. А. Плисс // Системное программирование. — 2013. — Т. 8. — С. 97—114.

47. *Пилипенко*, *А. В.* Анализ достижимости методов при выборочной инициализации классов в программах на языке Java / А. В. Пилипенко, О. А. Плисс // Программная инженерия. — 2014. — № 8. — С. 3—8.

48. *Пилипенко*, *А. В.* Понижение избыточности Java-программ при выборочной инициализации классов / А. В. Пилипенко, О. А. Плисс // Программная инженерия. — 2016. — Т. 7, № 8. — С. 339—350.

49. *Dean*, *J.* Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis / J. Dean, D. Grove, C. Chambers // Proceedings of the 9th European Conference on Object-Oriented Programming. — London, UK, UK : Springer-Verlag, 1995. — С. 77—101. — (ECOOP '95).

50. *Tip*, *F.* Scalable Propagation-based Call Graph Construction Algorithms / F. Tip, J. Palsberg // SIGPLAN Not. — New York, NY, USA, 2000. — Окт. — Т. 35, № 10. — С. 281—293.

51. Practical Virtual Method Call Resolution for Java / V. Sundaresan [и др.] // SIGPLAN Not. — New York, NY, USA, 2000. — Окт. — Т. 35, № 10. — С. 264—280.

52. Call Graph Construction in Object-oriented Languages / D. Grove [и др.] // SIGPLAN Not. — New York, NY, USA, 1997. — Окт. — Т. 32, № 10. — С. 108—124.

53. *Andersen*, *L. O.* Program Analysis and Specialization for the C Programming Language: тех. отч. / L. O. Andersen ; — 1994.

54. *Steensgaard*, *B.* Points-to Analysis in Almost Linear Time / B. Steensgaard // Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages. — 1996. — С. 32—41.

55. *Liang*, *D.* Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for Java / D. Liang, M. Pennings, M. J. Harrold // Proceedings of the ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering. — 2001. — С. 73—79.

56. *Rountev*, *A.* Points-To Analysis for Java using Annotated Constraints / A. Rountev, A. Milanova, B. Ryder // Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. — 2001. — С. 43—55.

57. General flow-sensitive pointer analysis and call graph construction / E. Horváth [и др.] // Proceedings of the Ninth Symposium on Programming Languages and Software Tools. — 2005. — С. 49—58.

58. *Zhu*, *J.* Symbolic Pointer Analysis Revisited / J. Zhu, S. Calman // SIGPLAN Not. — New York, NY, USA, 2004. — Июнь. — Т. 39, № 6. — С. 145—157.

59. *Milanova*, *A.* Precise call graph construction in the presence of function pointers / A. Milanova, A. Rountev, B. G. Ryder // Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on. — IEEE. 2002. — С. 155—162.

60. *Whaley*, *J.* Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams / J. Whaley, M. S. Lam // SIGPLAN Not. — New York, NY, USA, 2004. — Июнь. — Т. 39, № 6. — С. 131—144.

61. Practical Extraction Techniques for Java / F. Tip [и др.] // ACM Trans. Program. Lang. Syst. — New York, NY, USA, 2002. — Нояб. — Т. 24, № 6. — С. 625—666.

62. *Qian*, *F.* A Study of Type Analysis for Speculative Method Inlining in a JIT Environment / F. Qian, L. Hendren // Proceedings of the 14th International Conference on Compiler Construction. — Edinburgh, UK : Springer-Verlag, 2005. — С. 255—270. — (CC'05).

63. *Ananian*, *C. S.* Data Size Optimizations for Java Programs / C. S. Ananian, M. Rinard // SIGPLAN Not. — New York, NY, USA, 2003. — Июнь. — Т. 38, № 7. — С. 59—68.

64. *Rippert*, *C.* A Low-Footprint Class Loading Mechanism for Embedded Java Virtual Machines / C. Rippert, A. Courbot, G. Grimaud // 3rd ACM International Conference on the Principles and Practice of Programming in Java. — Las Vegas, USA, 2004.

65. *Rippert*, *C.* On-the-fly metadata stripping for embedded Java operating systems / C. Rippert, D. Deville // Smart Card Research and Advanced Applications VI. — 2004. — С. 17—32.

66. *Kozen*, *D.* Eager Class Initialization for Java / D. Kozen, M. Stillerman // Formal Techniques in Real-Time and Fault-Tolerant Systems: 7th International Symposium, FTRTFT 2002 Co-sponsored by IFIP WG 2.2 Oldenburg, Germany, September 9–12, 2002 Proceedings / под ред. W. Damm, E. .-.-R. Olderog. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2002. — С. 71—80.

67. Measuring and improving application launching performance on android devices / K. Nagata [и др.] // Computing and Networking (CANDAR), 2013 First International Symposium on. — IEEE. 2013. — С. 636—638.

68. aicas GmbH JamaicaVM 8.1 — User Manual [Электронный ресурс] / aicas GmbH. — 2017. — URL: https://www.aicas.com/cms/sites/default/files/JamaicaVM-8.1-Manual-A5.pdf (дата обр. 01.01.2018).

69. SlimVM: A Small Footprint Java Virtual Machine for Connected Embedded Systems / C. Kerschbaumer [и др.] // Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. — Calgary, Alberta, Canada : ACM, 2009. — С. 133—142. — (PPPJ '09).

70. *Livshits*, *B.* Reflection Analysis for Java / B. Livshits, J. Whaley, M. S. Lam // Proceedings of the Third Asian Conference on Programming Languages and Systems. — Tsukuba, Japan : Springer-Verlag, 2005. — С. 139—160. — (APLAS'05).

71. *Christensen*, *A. S.* Precise Analysis of String Expressions / A. S. Christensen, A. Møller, M. I. Schwartzbach // Proceedings of the 10th International Conference on Static Analysis. — San Diego, CA, USA : Springer-Verlag, 2003. — С. 1—18. — (SAS'03).

72. Self-inferencing Reflection Resolution for Java / Y. Li [и др.] // Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586. — New York, NY, USA : Springer-Verlag New York, Inc., 2014. — С. 27—53.

73. More Sound Static Handling of Java Reflection / Y. Smaragdakis [и др.] // Programming Languages and Systems: 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings / под ред. X. Feng, S. Park. — Cham : Springer International Publishing, 2015. — С. 485—503.

74. *Li*, *Y.* Effective soundness-guided reflection analysis / Y. Li, T. Tan, J. Xue // International On Static Analysis. — Springer. 2015. — С. 162—180.

75. *Li*, *Y.* Understanding and Analyzing Java Reflection / Y. Li, T. Tan, J. Xue // CoRR. — 2017. — Т. abs/1706.04567. — arXiv: 1706.04567.

76. Fast Online Pointer Analysis / M. Hirzel [и др.] // ACM Trans. Program. Lang. Syst. — New York, NY, USA, 2007. — Апр. — Т. 29, № 2.

77. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders / E. Bodden [и др.] // Proceedings of the 33rd International Conference on Software Engineering. — Waikiki, Honolulu, HI, USA : ACM, 2011. — С. 241—250. — (ICSE '11).

78. *Landman*, *D.* Challenges for static analysis of Java reflection: literature review and empirical study / D. Landman, A. Serebrenik, J. J. Vinju // Proceedings of the 39th International Conference on Software Engineering. — IEEE Press. 2017. — С. 507—518.

79. *Brown*, *P.* Macros without tears / P. Brown // Software: Practice and Experience. — 1979. — Т. 9, № 6. — С. 433—437.

80. The Java Virtual Machine Specification, Java SE 8 Edition / T. Lindholm [и др.]. — 1st. — Addison-Wesley Professional, 2014.

81. *Welch*, *T. A.* A Technique for High-Performance Data Compression / T. A. Welch // Computer. — Los Alamitos, CA, USA, 1984. — Июнь. — Т. 17, № 6. — С. 8—19.

82. On the Dictionary Compression for Java Card Environment / M. Zilli [и др.] // Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems. — St. Goar, Germany : ACM, 2013. — С. 68—76. — (M-SCOPES '13).

83. *Meilă*, *M.* An Experimental Comparison of Model-Based Clustering Methods / M. Meilă, D. Heckerman // Machine Learning. — 2001. — Янв. — Т. 42, № 1. — С. 9—29.

84. *Blekas*, *K.* Incremental Mixture Learning for Clustering Discrete Data / K. Blekas, A. Likas // Methods and Applications of Artificial Intelligence: Third Hellenic Conference on AI, SETN 2004, Samos, Greece, May 5-8, 2004. Proceedings / под ред. G. A. Vouros, T. Panayiotopoulos. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2004. — С. 210—219.

85. *Fraser*, *C. W.* A Retargetable C Compiler: Design and Implementation / C. W. Fraser, D. R. Hanson. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995.

86. *Rayside*, *D.* Compact Java Binaries for Embedded Systems / D. Rayside, E. Mamas, E. Hons // Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. — Mississauga, Ontario, Canada : IBM Press, 1999. — С. 9—. — (CASCON '99).

87. *Huffman*, *D. A.* A Method for the Construction of Minimum-Redundancy Codes / D. A. Huffman // Proceedings of the IRE. — 1952. — Сент. — Т. 40, № 9. — С. 1098—1101.

88. *Latendresse*, *M.* Generation of Fast Interpreters for Huffman Compressed Bytecode / M. Latendresse, M. Feeley // Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators. — San Diego, California : ACM, 2003. — С. 32—40. — (IVME '03).

89. A Locally Adaptive Data Compression Scheme / J. L. Bentley [и др.] // Commun. ACM. — New York, NY, USA, 1986. — Апр. — Т. 29, № 4. — С. 320—330.

90. *Latendresse*, *M.* Automatic Generation of Compact Programs and Virtual Machines for Scheme / M. Latendresse // Proceedings of the Workshop on Scheme and Functional Programming. — 2000. — С. 45—52.

91. *Pugh*, *W.* Compressing Java Class Files / W. Pugh // SIGPLAN Not. — New York, NY, USA, 1999. — Май. — Т. 34, № 5. — С. 247—258.

92. *Nigel Horspool*, *R.* Tailored Compression of Java Class Files. / R. Nigel Horspool, J. Corless. — 1998. — Окт.

93. *Bradley*, *Q.* JAZZ: An Efficient Compressed Format for Java Archive Files / Q. Bradley, R. N. Horspool, J. Vitek // Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research. — Toronto, Ontario, Canada : IBM Press, 1998. — С. 7—. — (CASCON '98).

94. *Kim*, *D.-W.* A Study on the Optimization of Class File for Java Card Platform / D.-W. Kim, M.-S. Jung // Revised Papers from the International Conference on Information Networking, Wireless Communications Technologies and Network Applications-Part I. — London, UK, UK : Springer-Verlag, 2002. — С. 563—570. — (ICOIN '02).

95. *Shaylor*, *N.* A Java Virtual Machine Architecture for Very Small Devices / N. Shaylor, D. N. Simon, W. R. Bush // SIGPLAN Not. — New York, NY, USA, 2003. — Июнь. — Т. 38, № 7. — С. 34—41.

96. *Hovemeyer*, *D.* More Efficient Network Class Loading Through Bundling / D. Hovemeyer, W. Pugh // Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1. — Monterey, California : USENIX Association, 2001. — С. 17—17. — (JVM'01).

97. The Java Language Specification, Java SE 8 Edition / J. Gosling [и др.]. — 1st. — Addison-Wesley Professional, 2014.

98. *Bloch*, *J.* Effective Java (2Nd Edition) (The Java Series) / J. Bloch. — 2-е изд. — Upper Saddle River, NJ, USA : Prentice Hall PTR, 2008.

99. *Liang*, *S.* Java Native Interface: Programmer's Guide and Reference / S. Liang. — 1st. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999.

100. libclang: C Interface to Clang [Электронный ресурс]. — 2018. — URL: http://clang.llvm.org/doxygen/group__CINDEX.html (дата обр. 11.01.2018).

101. *Garey*, *M. R.* Computers and Intractability: A Guide to the Theory of NP-Completeness / M. R. Garey, D. S. Johnson. — New York, NY, USA : W. H. Freeman & Co., 1979.

102. Gemalto Cinterion® EHS5 Wireless Module Datasheet [Электронный ресурс]. — 2015. — URL: https://www.gemalto.com/brochures-site/download-site/Documents/M2M_EHS5_datasheet.pdf (дата обр. 24.02.2018).

103. Gemalto Cinterion® EHS6 Wireless Module Datasheet [Электронный ресурс]. — 2015. — URL: https://www.gemalto.com/brochures-site/download-site/Documents/M2M_EHS6_datasheet.pdf (дата обр. 24.02.2018).

104. Gemalto Cinterion® EHS8 Wireless Module Datasheet [Электронный ресурс]. — 2015. — URL: https://www.gemalto.com/brochures-site/download-site/Documents/M2M_EHS8_datasheet.pdf (дата обр. 24.02.2018).

105. *Коротких*, *Н.* Передовые JAVA-модули Cinterion. Особенности программирования / Н. Коротких // Беспроводные Технологии. — 2013. — Т. 3, № 32. — С. 24—26.

106. JSR 360: Connected Limited Device Configuration 8 [Электронный ресурс] / JCP. — URL: https://jcp.org/en/jsr/detail?id=360 (дата обр. 07.01.2018).

107. JSR 361: JavaTM ME Embedded Profile [Электронный ресурс] / JCP. — URL: https://jcp.org/en/jsr/detail?id=360 (дата обр. 07.01.2018).

108. GrinderBench™ [Электронный ресурс] / EEMBC. — URL: https://www.eembc.org/techlit/datasheets/grinder_db.pdf (дата обр. 04.01.2018).

109. OpenJDK [Электронный ресурс]. — 2017. — URL: http://openjdk.java.net/ (дата обр. 11.01.2018).

110. The Scala Programming Language [Электронный ресурс]. — 2018. — URL: https://www.scala-lang.org/ (дата обр. 11.01.2018).

111. Kotlin Programming Language [Электронный ресурс]. — 2017. — URL: https://kotlinlang.org/ (дата обр. 11.01.2018).

Список рисунков

1.1 Результат анализов CHA и RTA 17
1.2 Алгоритмы анализа достижимости методов 18
1.3 Результат анализа XTA 19
1.4 Результат анализов VTA и DTA 20

2.1 Зависимости между инициализацией классов и анализом достижимости методов 47
2.2 Пример анализа достижимости объектов в куче 52
2.3 Пример анализа дополнительных зависимостей 55
2.4 Пример анализа достижимости неудалимых объектов 63
2.5 Пример модельного графа 67

3.1 Пример префиксного дерева словаря последовательностей 78
3.2 Пример построения шаблона 86

4.1 Процесс специализации Java-платформы Oracle Java ME Embedded для заданного приложения в закрытой модели 88
4.2 Эксперимент 1. Относительное сокращение размера образа 95
4.3 Эксперимент 2. Относительное сокращение размера образа 97
4.4 Эксперимент 3. Относительное сокращение размера образа 99
4.5 Эксперимент 4. Относительное сокращение размера образа 101
4.6 Эксперимент 4. Относительное сокращение количества полей в образе 102
4.7 Эксперимент 2. Относительное увеличение степени сжатия при использовании свертки и укорачивания аргументов 114
4.8 Эксперимент 3. Относительное увеличение степени сжатия при упрощении исходного набора инструкций 117
4.9 Эксперимент 4. Относительное увеличение степени сжатия при генерации несовместимого набора инструкций 117

Список таблиц

1 Набор приложений . . . 94
2 Эксперимент 1. Размер образа в байтах и относительное сокращение размера . . . 95
3 Эксперимент 2. Размер образа в байтах и относительное сокращение размера . . . 97
4 Эксперимент 3. Размер образа в байтах и относительное сокращение размера . . . 99
5 Эксперимент 4. Размер образа в байтах и относительное сокращение размера . . . 101
6 Эксперимент 4. Количество полей в образе и относительное сокращение количества полей . . . 102
7 Эксперимент 5. Количество native-методов и их зависимостей . . . 103
8 Размер байт-кода и интерпретатора, необходимого для его выполнения, в байтах . . . 107
9 Эксперимент 1. Степень сжатия в зависимости от $max\_pattern\_length$ 112
10 Эксперимент 2. Степень сжатия при использовании параметризованных шаблонов ($p_{max} = 1$) . . . 113
11 Эксперимент 2. Степень сжатия при использовании параметризованных шаблонов со сворачиванием аргументов ($p_{max} = 2$) 113
12 Эксперимент 2. Степень сжатия при использовании параметризованных шаблонов со сворачиванием и укорачиванием аргументов ($p_{max} = 3$) . . . 113
13 Эксперимент 3. Степень сжатия при использовании стандартного набора инструкций . . . 115
14 Эксперимент 3. Степень сжатия при использовании упрощенного набора инструкций . . . 115
15 Эксперимент 4. Степень сжатия при генерации совместимого набора инструкций . . . 116
16 Эксперимент 4. Степень сжатия при генерации несовместимого набора инструкций . . . 116

Приложение А

Формальное описание алгоритмов

A.1 Reachable Member Analysis

```
// Функция post() добавляет элемент в рабочее множество если
// соответствующий элемент не присутствует в рабочем множестве
// и не был ранее обработан.

function analyze(Program p) {
 for each method in p.entrypoints
   post(method)
 while worklist ≠ ∅
   analyze(dequeue(worklist))
}

function analyze(Method m) {
 methods.add(m)
 for each expression in m.body
   if (expression = read(C.f)) post(C, f)
   if (expression = read(e.f)) post(type(e), f)
   if (expression = call(C.m)) post(C, m)
   if (expression = call(e.m)) post(type(e), m)
}

function analyze(Type t) {
 subtypes(t).add(t);
 for each type in parents(t))
   subtypes(type).add(t)
 let pm = members(parent(t))
 for each member in pm
   post(t, member)
}

function analyze(Type t, Field f) {
 members(t).add(f)
 for each object in instances(t)
   post(value(object.f))
```

```
 for each type in subtypes(t)
   post(type, f)
}

function analyze(Type t, Method m) {
 members(t).add(m)
 for each type in subtypes(t)
   post(resolve(type, m))
}

function analyze(Object o) {
 post(type(o))
 instances(type(o)).add(o)
 for each field f in members(t)
   post(value(o.field))
}
```

A.2 Алгоритмы понижения избыточности

Анализ достижимости методов

```
function find_reachable_methods {
  reachable_methods, new_reachable_methods = entry_points;
  indirect_inocations = ∅;
  instantiable_classes = ∅;
  initializable_classes = ∅;

  while new_reachable_methods ≠ ∅ ∨
        new_initializable_classes ≠ ∅ {
    while new_reachable_methods ≠ ∅ {
      method = @new_reachable_methods; // @ - операция извлечения
                                       // и удаления произвольного
                                       // элемента из множества
      for each inst in method {
        switch inst {
          case direct_call:
            add(reachable_methods, new_reachable_methods,
```

```
            inst.callee);
        case indirect_call:
          add(indirect_invocations, new_indirect_invocations,
            inst.callee);
        case new:
          add(instantiable_classes, new_instantiable_classes,
            inst.class);
        case class_init:
          add(initializable_classes, new_initializable_classes,
            inst.class);
        case field_read:
          read_fields ∪= {inst.field};
        case ldc:
          add(instantiable_classes, new_instantiable_classes,
            inst.class);
          add(initializable_classes, new_initializable_classes,
            inst.class);
      }
      for each dependency in method.dependencies {
      switch dependency {
        case MethodCall:
          add(reachable_methods, new_reachable_methods,
            dependency.method);
        case ClassNew:
          add(instantiable_classes, new_instantiable_classes,
            dependency.class);
          add(initializable_classes, new_initializable_classes,
            dependency.class);
        case ClassAccess:
          add(initializable_classes, new_initializable_classes,
            dependency.class);
        case FieldRead:
          read_fields ∪= {dependency.field};
      }
      for each exception_class in method.exception_table {
        referenced_classes ∪= {exception_class};
      }
    }
  }

  for each class in new_initializable_classes {
    if can_initialize(class) {
```

```
        class.initialize();
      } else {
        add(reachable_methods, new_reachable_methods,
            class.clinit);
      }
    }

    gc(); visit_objects();

    add_all_matching_methods(new_indirect_invocations,
                             instantiable_classes);
    add_all_matching_methods(indirect_invocations,
                             new_instantiable_classes);

    for each class in new_instantaible_classes {
      add(reachable_methods, new_reachable_methods,
          class.finalize);
    }

    new_indirect_invocations = ∅;
    new_instantiable_classes = ∅;
  }
}

function add(set, working_set, item) {
  if item ∉ set {
    set, working_set ∪= {item};
  }
}

function visit_objects() {
  for each finalizable object {
    add(instantiable_classes, new_instantiable_classes,
        object.class);
  }

  reachable_objects, new_reachable_objects = rootset;
  while new_reachable_objects ≠ ∅ {
    object = @new_reachable_objects;
    add(instantiable_classes, new_instantiable_classes,
        object.class);
    if is_instance_of(object, java.lang.Class) {
```

```
      add(instantiable_classes, new_instantiable_classes,
          class);
      add(initializable_classes, new_initializable_classes,
          class);
    }
    if is_special_reference(object) {
      add(reachable_objects, new_reachable_objects,
          object.referent);
    }
    for each reference in object {
      if field(reference) ∈ read_fields {
        add(reachable_objects, new_reachable_objects,
            object.field);
      }
    }
  }
}

function add_all_matching_methods(methods, classes) {
  for each indirect_invocation ∈ methods {
    for each subtype of indirect_inocation.holder {
      if subtype ∈ classes {
        method = find(indirect_inocation, subtype);
        add(reachable_methods, new_reachable_methods,
            method);
      }
    }
  }
}
```

Анализ удалимости полей

```
function find_unremovable_fields {
  compute_putstatic_fields();
  compute_written_object_fields();

  unremovable_fields = read_fields;
  for each field in putstatic_fields {
    if !is_initialized(field.holder) {
```

```
    unremovable_fields ∪= {field};
  }
}

if special_reference_classes ∩ instantiable_classes ≠ ∅ {
  for each object in heap {
    for each field in object {
      if object.field is reference {
        unremovable_fields ∪= {field};
      }
    }
  }
  unremovable_fields ∪= written_fields;
} else {
  initialize_may_refer_to();
  initialize_refer_to();

  ref_to_finalizable_classes, new_ref_to_finalizable_classes =
    finalizable_classes ∩ instantiable_classes;
  while new_ref_to_finalizable_classes ≠ ∅ {
    while new_ref_to_finalizable_classes ≠ ∅ {
      class = @new_ref_to_finalizable_classes;
      for each field in may_refer_to(class) ∩
            (refer_to(class) ∪ written_fields) {
        add(ref_to_finalizable_fields,
            new_ref_to_finalizable_fields, field);
      }
    }
    while new_ref_to_finalizable_fields ≠ ∅ {
      field = @new_ref_to_finalizable_fields;
      if field is non-static {
        for each subclass of field.holder {
          if subclass in instantiable_classes {
            add(ref_to_finalizable_classes,
                new_ref_to_finalizable_classes, subclass);
          }
        }
      }
    }
  }
  unremovable_fields ∪= ref_to_finalizable_fields;
}
```

```
}

function compute_putstatic_fields {
  for each method in reachable_methods {
    for each inst in method {
      if inst is putstatic {
        putstatic_fields ∪= {inst.field};
      }
    }
  }
}

function compute_written_object_fields {
  for each method in reachable_methods {
    for each inst in method {
      if inst is object_field_write {
        written_object_fields ∪= {inst.field};
      }
    }
    for each dependency in method.dependencies {
      if dependency is FieldWrite {
        written_object_fields ∪= {dependency.field};
      }
    }
  }
}

function initialize_may_refer_to {
  for each class {
    for each field in class {
      for each subtype of field.type {
        may_refer_to(subtype) ∪= {field};
      }
    }
  }
}

function initialize_refer_to {
  for each object in heap {
    for each field in object {
      if object.field is reference {
        referent = object.field;
```

```
        refer_to(referent.class) ∪= {field};
      }
    }
  }
}
```

Анализ удалимости классов

```
function find_unremovable_classes {
  compute_referenced_classes()
  unremovable_classes = referenced_classes ∪ instantiable_classes;
  for each field in unremovable_fields {
    unremovable_classes ∪= {field.holder};
  }
}

function compute_referenced_classes {
  for each method in reachable_methods {
    for each inst in method {
      if inst is class_reference {
        referenced_classes ∪= {inst.class};
      }
    }
  }
}
```

A.3 Алгоритм сжатия Java байт-кода

Построение словаря последовательностей

```
function build_pattern_tree {
  for each basic_block in program {
    sequences[0] = {ε};
    for i from 1 to basic_block.length {
```

```
      inst = basic_block[i - 1];
      sequences[i] = {ε};
      for each sequence in sequences[i - 1] {
        if sequence.length < max_pattern_length {
          sequences[i] ∪= {extend(sequence, inst)};
        }
      }
    }
  }
}

function extend(sequence, instruction) {
  if instruction ∉ sequence.egdes {
    sequence.egdes[instruction] = sequence(0, sequence.length + 1);
  }
  extended = sequence.egdes[instruction];
  extended.count++;
  return extended;
}

// Создает экземпляр вершины префиксного дерева последовательностей.
function sequence(count, length);
```

Перебор словаря шаблонов

```
// visit_function -- функция, которая вызывается при посещении
// каждого шаблона
function visit_patterns(pattern, matching_sequences, visit_function) {
  visit_function(pattern, sum_counts(matching_sequences));
  pattern_continuations = ∅;
  for each sequence in matching_sequences {
    for instr in sequence.edges {
      for each pattern_instr in pattern_instructions(instr) {
        pattern_continuations ∪= {pattern_instr};
      }
    }
  }
  for each pattern_inst in pattern_continuations {
```

```
    new_pattern = pattern + pattern_inst;
    new_matching_sequences = ∅;
    for each sequence in matching_sequences {
      for instr in sequence.edges {
        if matches(pattern_inst, inst) {
          new_matching_sequences ∪= sequence.edges[instr];
        }
      }
    }
    visit_patterns(new_pattern, new_matching_sequences,
                   visit_function);
  }
}

function sum_counts(matching_sequences) {
  count = 0;
  for each sequence in matching_sequences {
    count += sequence.count;
  }
  return count;
}

// Функция возвращает множество инструкций шаблона,
// которыми можно покрыть заданную инструкцию
// последовательности instr.
function pattern_instructions(instr) {
  if !instr.has_argument {
    return {pattern_instruction(instr.opcode)};
  }
  // Инструкция шаблона с фиксированным аргументом
  patterns = {pattern_instruction(instr.opcode, instr.argument)};
  // Инструкции шаблона с различной длиной аргумента
  // inst.argument_width - ширина аргумента инструкции
  // width(inst.argument) - количество байт, необходимое
  //                        для кодирования значения аргумента
  arg_width = inst.argument_width;
  while width(inst.argument) <= arg_width {
    patterns ∪= {pattern_instruction(instr.opcode, arg_width)};
    arg_width--;
  }
  return patterns;
}
```

```
// Проверяет, соответствует ли заданная инструкция
// последовательности inst инструкции шаблона pattern_inst.
function matches(pattern_inst, inst) {
  if pattern_inst.opcode ≠ inst.opcode {
    return false;
  }
  if pattern_inst.has_argument {
    return pattern_inst.argument = inst.argument;
  }
  return width(inst.argument) <= pattern_inst.argument_width
}

// Создает экземпляр инструкции шаблона без аргумента.
function pattern_instruction(opcode)
// Создает экземпляр инструкции шаблона с фиксированным аргументом.
function pattern_instruction(opcode, argument)
// Создает экземпляр инструкции шаблона с параметризованным аргументом
// заданной длины.
function pattern_instruction(opcode, arg_width)
```

Выбор набора инструкций

```
function compress {
  while opcode = find_unused_opcode ≠ None &&
        best_pattern = find_best_pattern ≠ ε {
    add_instruction(best_pattern, opcode);
    replace_pattern_with_instruction(best_pattern, opcode);
  }
}

function weight(pattern, count) {
  interpreter_size = 0;
  for each instr in pattern {
    interpreter_size += instr.interpreter_size;
  }
  for each instr in pattern {
    bytecode_gain += 1 + instr.original_argument_width
```

```
                        - instr.argument_width;
  }
  return count * bytecode_gain - interpreter_size;
}

function find_unused_opcode {
  for each instr in program {
    used[instr.opcode] = true;
  }
  for i from 0 to 256 {
    if !used[i] {
      return i;
    }
  }
  return None;
}

function find_best_pattern {
  build_pattern_tree;

  best_pattern_weight = 0;
  best_pattern = ε;
  function visit_pattern(pattern, count) {
    if weight(pattern) > best_pattern_weight {
      best_pattern = pattern;
      best_pattern_weight = weight(pattern, count);
    }
  }
  visit_patterns(ε, ∅, visit_pattern);
  return best_pattern;
}

function matches_pattern(basic_block, start, pattern) {
  for i from 0 to pattern.length {
    if !matches(pattern[i], basic_block[start + i]) {
      return false;
    }
  }
  return true;
}

function replace_pattern(basic_block, i, pattern, opcode) {
```

```
  for i from 0 to pattern.length {
    pattern_inst = pattern[i];
    inst = basic_block[start + i];
    if pattern_inst.argument_width ≠ 0 {
      arguments += inst.argument truncate to
                   pattern_inst.argument_width
    }
  }
  replace(basic_block, i, pattern.length,
          instruction(opcode, arguments));
}

function replace_pattern_with_instruction(pattern, opcode) {
  for each basic_block in program {
    for i from 0 to basic_block.length {
      if matches_pattern(basic_block, i, pattern) {
        replace_pattern(basic_block, i, pattern, opcode);
      }
    }
  }
}

// Заменяет последовательность инструкций новой инструкцией.
// basic_block - базовый блок,
// start - позиция исходной последовательности,
// count - количество инструкций последовательности,
// instruction - новая инструкция.
function replace(basic_block, start, count, instruction)

// Добавляет инструкцию для кодирования шаблона pattern в набор
// инструкций.
// opcode - опкод для кодирования новой инструкции.
function add_instruction(pattern, opcode)

// Создает экземпляр инструкции с заданным опкодом и аргументами.
function instruction(opcode, arguments)
```

Приложение Б

Акты о внедрении результатов диссертационного исследования

Ниже приводятся копии актов о внедрении результатов диссертационного исследования.

Акт

о внедрении результатов

кандидатской диссертационной работы Пилипенко А. В. "Разработка и реализация механизмов сокращения размера Java-приложений для встраиваемых систем в закрытой модели", представленной на соискание ученой степени кандидата технических наук.

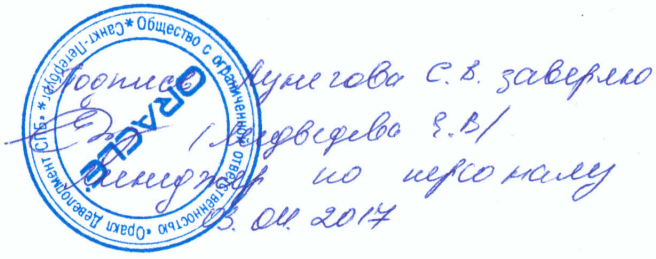
Результаты, полученные в работе, были использованы на практике в виртуальной машине CLDC HotSpot Implementation в продукте Java ME Embedded.

Реализация предложенных алгоритмов понижения избыточности в виртуальной машине CLDC HotSpot Implementation позволила сократить статическое и динамическое потребление памяти при исполнении фиксированного набора приложений. Это, в свою очередь, позволило сократить минимальные аппаратные требования платформы Java ME Embedded в закрытой модели.

Особое внимание в работе уделено сохранению семантики языка Java, описанной в его спецификации. Набор тестов на соответствие спецификации Technology Compatibility Kit (TCK) не выявил нарушения поведения при использовании предложенных алгоритмов.

Тестирование производительности виртуальной машины не выявило негативного влияния реализованных алгоритмов на скорость исполнения кода.

Директор отдела разработки
программного обеспечения
ООО «Оракл Девелопмент СПБ» Лунегов С.В.
3 апреля 2017 года



ООО «ИТ центр СПбГУ»

Общество с ограниченной ответственностью «Научно-исследовательский центр информационных технологий СПбГУ»

198504, город Санкт-Петербург, город Петергоф,
улица Ульяновская, дом 1, литер А,
тел. (812) 4287109, факс (812) 4287409

Р/С 40702810855240001553 в Северо-Западном банке Публичного акционерного общества «СБЕРБАНК РОССИИ», БИК 044030653, ОКПО 38037767, ОГРН 1117847705975, ИНН/КПП 7819315060/781901001

Акт
о внедрении результатов

кандидатской диссертационной работы "Разработка и реализация механизмов сокращения размера Java-приложений для встраиваемых систем в закрытой модели", представленной на соискание ученой степени кандидата технических наук.

Настоящим актом подтверждается, что результаты диссертационной работы Пилипенко А. В., а именно:

- алгоритмы понижения избыточности, применимые при использовании раздельной инициализации;
- метод анализа межъязыковых зависимостей между кодом на Java и C++;
- алгоритм специализации Java байт-кода минимизирующий суммарный размер программы и интерпретатора, необходимого для ее выполнения

внедрены в научную и проектную деятельность научно-исследовательского центра информационных технологий СПбГУ.

Полученные результаты обладают актуальностью, представляют практический интерес и могут быть использованы для сокращения аппаратных требования целого ряда используемых программных платформ, включая Java, Microsoft .NET.

Генеральный директор
ООО «ИТ центр СПбГУ» А.Н. Терехов

ОБЩЕСТВО С ОГРАНИЧЕННОЙ ОТВЕТСТВЕННОСТЬЮ "Научно-исследовательский центр информационных технологий СПбГУ" САНКТ-ПЕТЕРБУРГ