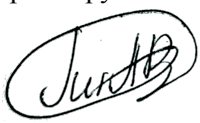
На правах рукописи



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»

Автореферат
диссертации на соискание учёной степени
кандидата технических наук

Санкт-Петербург — 2018

Работа выполнена на кафедре информатики Санкт-Петербургского государственного университета.

Научный руководитель: **Кияев Владимир Ильич**
кандидат физико-математических наук, доцент
Санкт-Петербургский государственный университет, доцент

Официальные оппоненты: **Аветисян Арутюн Ишханович,**
член-корреспондент РАН, доктор физико-математических наук, профессор,
Институт системного программирования РАН, директор

Корзун Дмитрий Жоржевич,
кандидат физико-математических наук, доцент,
Петрозаводский государственный университет, доцент

Ведущая организация: Федеральное государственное автономное образовательное учреждение высшего образования Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики

Защита состоится **«09» октября 2018 г. в 16 часов 00 минут** на заседании диссертационного совета Д 002.199.01, созданного на базе Федерального государственного бюджетного учреждения науки Санкт-Петербургского института информатики и автоматизации Российской академии наук по адресу: 199178, Санкт-Петербург, 14 линия В.О., д. 39.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки Санкт-Петербургского института информатики и автоматизации Российской академии наук, http://www.spiiras.nw.ru.

Отзывы на автореферат в двух экземплярах, заверенные печатью учреждения, просьба направлять по адресу: 199178, Санкт-Петербург, 14 линия В.О., д. 39, ученому секретарю диссертационного совета Д 002.199.01.

Автореферат разослан «_____» **августа 2018 года**.

Ученый секретарь
диссертационного совета
Д 002.199.01,
кандидат технических наук  Зайцева Александра Алексеевна

Общая характеристика работы

Актуальность темы.

Качество программного обеспечения является критически важным аспектом промышленной разработки. 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. 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.

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.

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.

Изначально алгоритмы анализа достижимости методов, удалимости полей и классов были описаны для языка C++ в середине 90-х годов. 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.

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. 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. 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. 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. 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.

Методология и методы исследования определяются практическим фокусом диссертационного 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.

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 .
2. Конференции «Технологии Microsoft в теории и практике программирования 2014 (Новые подходы к разработке ПО на примере технологий Microsoft и EMC)» .
3. Конференции JavaOne San-Francisco 2014 .
4. Международной научно-практической конференции «Интеллектуальные и информационные технологии в формировании цифрового общества» 2017 .

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.

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 [1—5; 7], 3 из которых 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 [2; 3; 6; 7] написаны в соавторстве. В статьях [2; 3; 7] 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 [5; 6] 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.

Содержание работы

Во **введении** обосновывается актуальность исследований, проводимых в рамках диссертационной работы, приводится обзор научной литературы по изучаемой проблеме, формулируется цель, ставятся задачи работы, излагается научная новизна и практическая значимость представляемой работы.

Первая глава посвящена обзору существующих механизмов сокращения размера приложений. Один из рассмотренных механизмов – понижение избыточности путем удаления неиспользуемых методов, полей и классов. Анализ достижимости методов – ключевой анализ при понижении избыточности, так как все последующие анализы удалимости анализируют код методов, выделенных на данном этапе. Для нахождения множества методов, которые могут быть исполнены при выполнении программы, строится транзитивное замыкание методов, достижимых из точек входа программы.

В языке Java есть два механизма динамической диспетчеризации – это виртуальные и интерфейсные вызовы. В обоих случаях связываемый метод определяется на этапе исполнения в зависимости от типа аргумента-получателя. Для анализа таких вызовов вычисляется консервативное приближение множества возможных типов объектов-получателей. В простом случае можно считать, что это множество равно множеству типов, совместимых со статическим типом объекта получателя. Такой подход называется Class Hierarchy Analysis (CHA). Алгоритм Rapid Type Analysis (RTA) уточняет этот подход за счет пересечения

множества совместимых типов с множеством классов, экземпляры которых создаются достижимым кодом.

Для известного набора достижимых методов можно вычислить множество полей, которые используются достижимым кодом, и удалить неиспользуемые. Стандартный критерий используемости поля – наличие операций чтения, позволяет удалять неиспользуемые, но инициализируемые поля. В Java удаление объектного поля, для которого в достижимом коде есть операции записи, но нет операций чтения, нарушает семантику финализации объектов поскольку может нарушить достижимость объектов, удаление которых приложение может отследить. Консервативный критерий – наличие операций чтения или записи, не нарушает семантику финализации, но не позволяет удалять неиспользуемые, но инициализируемые поля.

Процесс инициализации виртуальной машины при каждом старте порождает один и тот же результат. Этот процесс можно ускорить, если сохранить состояние инициализированной виртуальной машины в виде загрузочного образа, и при старте восстанавливать состояние из сохраненного образа. При подготовке образа на хостовом устройстве может осуществляться инициализация всех или некоторых классов программы, что сокращает время инициализации приложения на целевом устройстве и в некоторых случаях сокращает размер образа.

Раздел 1.4 посвящен обсуждению механизма ромизации. В инициализированном состоянии виртуальной машины между методами, полями и объектами существуют зависимости, которые не разрешаются классическими алгоритмами. Впервые задача анализа инициализированного состояния была рассмотрена для языка Virgil, в котором все объекты времени исполнения создаются в процессе инициализации на хостовом устройстве. Reachable Members Analysis (RMA) – расширение алгоритма RTA, которое позволяет анализировать инициализированное состояние виртуальной машины. Алгоритм RMA вычисляет множество полей, читаемых достижимым кодом. Множество возможных типов объектов-получателей косвенных вызовов вычисляется как множество классов объектов, достижимых по ссылкам в читаемых полях.

В **разделе 1.6** приведен обзор систем понижения избыточности для Java-программ. Среди 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. ExoVM использует модификацию алгоритма RMA и инициализирует все загруженные классы до анализа достижимости методов. Такой подход нарушает спецификацию языка Java, может нарушить поведение программы и привести к увеличению размера образа. Все рассмотренные системы понижения избыточности требуют ручного описания зависимостей native-методов, что неэффективно и чревато ошибками.

Другой подход к сокращению размера, рассмотренный в первой главе, – использование интерпретируемого представления. Для заданного приложения

в закрытой модели можно добиться более эффективного кодирования путем специализации набора инструкций интерпретируемого представления. Специализированные наборы инструкций обеспечивают более компактное кодирование за счет следующих преобразований:

- 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.

Использование свертки или укорачивания аргументов одновременно со сверткой последовательностей инструкций позволяет оптимизировать кодирование аргументов инструкций в контексте последовательностей инструкций. Отмечается, что если исходный набор инструкций содержит инструкции, которые могут быть закодированы последовательностями более простых инструкций, упрощение исходного набора инструкций перед словарным сжатием может повысить эффективность сжатия.

Ни один из рассмотренных алгоритмов специализации Java байт-кода не использует все три преобразования одновременно и не упрощает исходный набор инструкций.

В заключение главы были сформулированы следующие подзадачи задач диссертации:

1. Разработать алгоритм понижения избыточности для Java, осуществляющий выборочную инициализацию классов и сохраняющий семантику финализации при удалении неиспользуемых, но инициализируемых полей.
2. Разработать механизм автоматического анализа зависимостей native-методов.
3. Разработать алгоритм специализации Java байт-кода, осуществляющий свертку и укорачивание аргументов совместно со сворачиванием последовательностей инструкций.

Во **второй** главе соискатель предлагает 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 удалении методов, не попавших в замыкание. При таком подходе возникает вопрос: в какой момент инициализировать классы? С одной стороны, инициализация классов влияет на достижимость методов. Методы, используемые исключительно для инициализации классов, становятся недостижимы после инициализации. С другой стороны, до анализа достижимости неизвестно, какие классы могут быть использованы достижимым кодом. Инициализация неиспользуемых классов может нарушить поведение программы и привести к созданию неиспользуемых, но достижимых объектов.

Для того чтобы обойти эти недостатки предложенный алгоритм инициализирует классы в процессе построения замыкания. В процессе построения замыкания вычисляется множество потенциально инициализируемых классов. Классы для ранней инициализации выбираются из данного множества с помощью консервативной эвристики.

Раздел 2.2 посвящен описанию семантики финализации объектов в Java. У приложения есть два способа отследить удаление объекта сборщиком мусора: объявить финализатор в классе объекта или установить на объект специальную ссылку. Объекты, удаление которых приложение может отследить, будем называть неудалимыми. Преждевременное удаление таких объектов может нарушить поведение программы.

В **разделе 2.5** предложен алгоритм анализа достижимости методов осуществляющий выборочную инициализацию классов. В процессе работы алгоритм вычисляет следующие множества:

- *ReachableMethods*, *NewReachableMethods* – множество достижимых методов и его рабочее подмножество;
- *IndirectInvocations*, *NewIndirectInvocations* – множество методов, используемых для косвенных вызовов, и его рабочее подмножество;
- *InstantiableClasses*, *NewInstantiableClasses* – множество классов, экземпляры которых доступны для косвенных вызовов, и его рабочее подмножество;
- *InitializableClasses*, *NewInitializableClasses* – множество классов, потенциально инициализируемых достижимым кодом, и его рабочее подмножество;
- *ReadFields* – множество читаемых полей.

При добавлении нового элемента в множество, у которого есть рабочее подмножество, элемент также добавляется в 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* анализируются его зависимости. Вначале просматривается код метода:
    а) методы, вызываемые инструкциями прямого вызова, добавляются в множество *ReachableMethods*;
    б) методы, используемые инструкциями `invokevirtual` и `invokeinterface`, запоминаются в множестве *IndirectInvocations*;
    в) классы, используемые для создания объектов с помощью инструкции `new`, запоминаются в множестве *InstantiableClasses*;
    г) классы, на которые ссылаются инструкции `new`, `invokestatic`, `getstatic`, `putstatic`, добавляются в множество *InitializableClasses*;

д) поля, используемые инструкциями `getstatic`, `getfield`, добавляются в множество *ReadFields*.

Проанализированные методы удаляются из множества *NewReachableMethods*.

2. Обрабатываются классы из множества *NewInitializableClasses*. Инициализируются классы, которые могут быть инициализированы в процессе создания образа. Методы `<clinit>` остальных классов добавляются в множество *ReachableMethods*. Обработанные классы удаляются из множества *NewInitializableClasses*.
3. Выполняется сборка мусора, финализируются удаленные объекты.
4. Обходятся объекты, достижимые для приложения. Это объекты, для которых существует путь от какой-либо корневой ссылки, не содержащий ссылок в нечитаемых полях. Классы посещенных объектов добавляются в множество *InstantiableClasses*.
5. В множество *ReachableMethods* добавляются методы, доступные через косвенные вызовы:
    а) добавляются методы классов из множества *InstantiableClasses*, достижимые через косвенные вызовы методов из множества *NewIndirectInvocations*;
    б) добавляются методы классов из множества *NewInstantiableClasses*, достижимые через косвенные вызовы методов из множества *IndirectInvocations*.
6. В множество *ReachableMethods* добавляются финализаторы классов из *NewInstantiableClasses*.
7. Обнуляются множества *NewInstantiableClasses*, *NewIndirectInvocations*.

В **разделе 2.5** предложен алгоритм анализа удалимости полей. Обозначены следующие критерии неудалимости поля:

– В достижимом коде есть операции чтения поля.
– Существует доступ к полю, который может привести к инициализации класса.
– Экземпляр объектного поля может содержать ссылку, которая препятствует уничтожению неудалимого объекта.

Как определить, что экземпляр объектного поля может содержать ссылку, которая препятствует уничтожению неудалимого объекта? Предложено считать, что поле нельзя удалять, если поле может прямо или косвенно ссылаться на неудалимый объект. Зная типы неудалимых объектов, можно вычислить множество полей, которые могут прямо или косвенно ссылаться на объекты данных типов:

– Объявленный тип поля определяет типы объектов, на которые может ссылаться экземпляр поля.
– Множество объявленных типов полей класса определяет типы объектов, на которые может ссылаться экземпляр класса.

Неудалимыми считаются финализируемые объекты и объекты, доступные посредством специальных ссылок. Типы финализируемых объектов известны статически, это классы с непустым методом `finalize`. Для вычисления типов объектов, доступных посредством специальных ссылок, используется межпроцедурный анализ указателей.

В **разделе 2.10** обсуждается метод для автоматического вычисления зависимостей native-методов. Для упрощения автоматического анализа зависимостей предлагается генерировать статический нативный интерфейс, который моделирует Java-сущности с помощью сущностей языка C++. При использовании такого статического интерфейса, анализ Java-зависимостей native-методов сводится к анализу достижимости сущностей языка C++.

В **третьей** главе предлагается алгоритм специализации набора инструкций Java байт-кода 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. Шаблоном называется последовательность инструкций, параметризованная 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. 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.

Перед специализацией набора инструкций осуществляется упрощение исходного набора инструкций. Инструкции, которые могут быть представлены в виде последовательностей других инструкций, в коде методов заменяются на эквивалентные последовательности.

В **разделе 3.4** приводится описание алгоритма построения словаря последовательностей, удовлетворяющих следующим ограничениям:

1. Длина последовательности не превышает max_len.
2. Последовательность не пересекает границы расширенных базовых блоков с одной точкой входа и множественными выходами.

Словарь последовательностей реализуется с помощью префиксного дерева, ребра которого помечаются инструкциями. Каждая вершина такого дерева описывает последовательность инструкций, которую можно восстановить, выписав инструкции всех ребер на пути от корня до данной вершины. Корень

дерева соответствует пустой последовательности ϵ. При построении словаря для каждой последовательности вычисляется число вхождений, которое записывается в соответствующей вершине.

Для построения словаря последовательностей тело программы разбивается на расширенные базовые блоки, код каждого блока анализируется отдельно. Для каждой позиции внутри блока с помощью динамического программирования вычисляется множество $Sequences(i)$ – множество последовательностей, которые оканчиваются в i-ой позиции.

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

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

- 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_len из множества предыдущей позиции может быть продолжена текущей инструкцией, поэтому в множество добавляется расширенная последовательность. Для каждой расширенной последовательности счетчик вхождений увеличивается на единицу.

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

где $instr(i)$ – инструкция в i-ой позиции блока, $extend(s, i)$ – функция, которая 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.

В **разделе 3.5** описывается алгоритм перебора шаблонов, покрывающих последовательности словаря. Перебор шаблонов осуществляется рекурсивно, начиная с пустого шаблона. На каждом шаге вычисляется множество всех возможных продолжений текущего шаблона. Далее приводится описание процесса вычисления всех возможных продолжений заданного шаблона.

Для каждого шаблона вычисляется множество $MatchingSequences(p)$[1] – множество всех последовательностей, покрываемых шаблоном. Последовательности в этом множестве представлены вершинами префиксного дерева.

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

Множество $MatchingSequences$ для пустого шаблона содержит один элемент – пустую последовательность.

Для данного шаблона p вычислим множество $PIContinuations(p)$ – множество инструкций шаблона, которыми может быть продолжен данный шаблон.

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

где $PI(si)$ – множество инструкций шаблона, которыми можно покрыть инструкцию последовательности si, $SIContinuations(p)$ – множество инструкций программы, которыми может быть продолжен шаблон p.

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

где $Edges(s)$ – множество исходящих ребер для вершины префиксного дерева s.

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.

Для выбора шаблонов, которые будут закодированы с помощью специализированных инструкций, используется жадный итеративный алгоритм. На каждом шаге алгоритм выбирает шаблон, кодирование которого с помощью специализированной инструкции обеспечивает наибольшее сокращение размера. Вхождения шаблона в теле методов заменяются на новую специализированную инструкцию. Сокращение размера, обеспечиваемое заданным шаблоном, считается следующим образом:

$$weight(p) = count(p) * (\sum_{i=1}^{length} bytecode\_gain_i - 1) - interpreter\_size(p),$$

где $count(p)$ – сколько раз данный шаблон встречается в коде программы;
$length$ – длина шаблона;
$bytecode\_gain_i$ – сокращение размера от инструкции в шаблоне с индексом i, равное 1 + (размер оригинального аргумента инструкции минус размер параметра инструкции в шаблоне);

$interpreter\_size(p)$ – размер кода интерпретатора для данного шаблона, считается равным сумме размеров кода интерпретатора для каждой составляющей его инструкции.

Для заданного шаблона $count(p)$ вычисляется как сумма вхождений последовательностей, покрываемых шаблоном. Вершины префиксного дерева хранят количество вхождений соответствующих последовательностей, таким образом, количество вхождений шаблона равно сумме значений в узлах $MatchingSequences(p)$.

В **четвертой** главе предложенные алгоритмы сравниваются с существующими решениями, отмечаются недостатки и ограничения предложенных решений. 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. Данные об относительном сокращении размера для различных приложений приведены на рисунке 2.
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 %). Статистика для различных приложений показана на графике на рисунке 1.
3. Сворачивание аргументов в контексте последовательностей инструкций увеличивает степень сжатия в среднем на 20 %. Контекстное укорачивание аргументов в шаблонах увеличивает степень сжатия еще на 26,52 %. Контекстное сворачивание и укорачивание аргументов в совокупности увеличивают степень сжатия в среднем на 51,85 %. График на рисунке 3 показывает относительное увеличение степени сжатия для разных наборов классов.

Заключение.

В работе решена важная задача по сокращению аппаратных требований Java-плафтормы при исполнении заданного приложения в закрытой модели, в том числе получены следующие научные результаты, составляющие итоги исследования:

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.

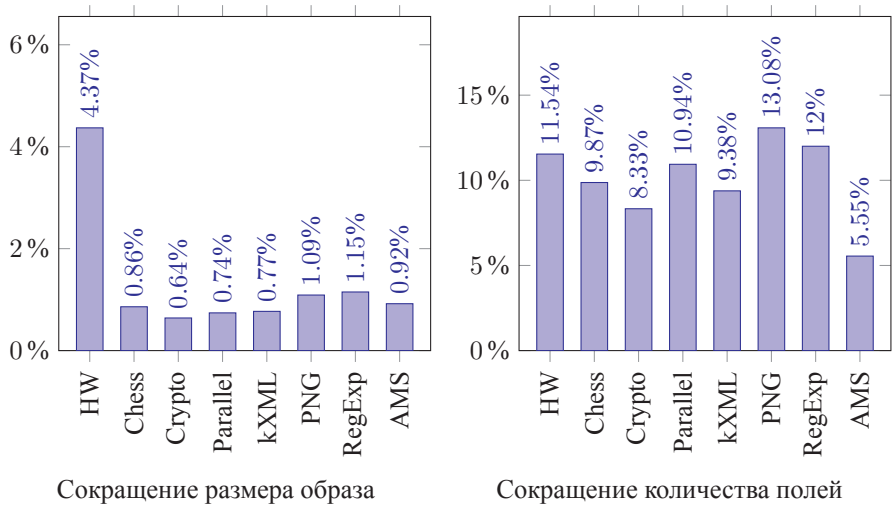


Рис. 1 — Влияние удаления неиспользуемых полей

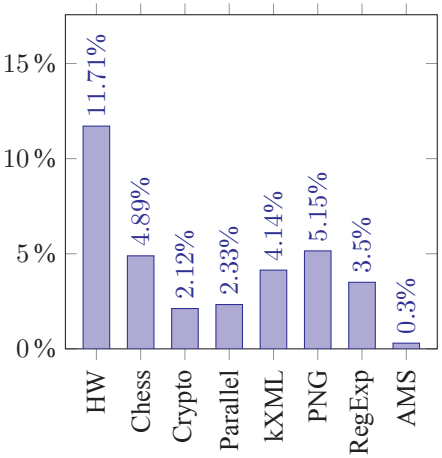


Рис. 2 — Сокращение размера образа

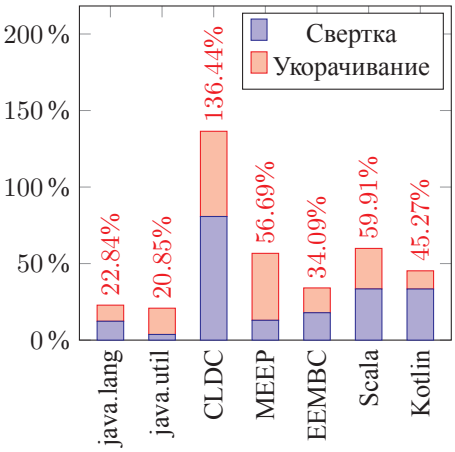


Рис. 3 — Увеличение степени сжатия

Обозначены оптимизации, применяемые при сжатии с помощью специализации набора инструкций. Отмечено, что существующие механизмы сжатия Java байт-кода не используют некоторые из описанных оптимизаций.

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.
5. Предложенные алгоритмы реализованы на практике. Экспериментальным путем измерена их эффективность.

Полученные результаты 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».

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.

Публикации автора по теме диссертации

В журналах из перечня рецензируемых научных изданий, в которых должны быть опубликованы основные научные результаты диссертаций на соискание ученой степени кандидата наук, доктора наук

1. *Пилипенко*, *А. В.* Обзор интерпретации и компиляции в виртуальных машинах / А. В. Пилипенко // Компьютерные инструменты в образовании. — 2012. — № 3. — С. 3—15.
2. *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.
3. *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.

В прочих изданиях

4. *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.
5. *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.
6. *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.
7. *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.