Подход к проведению динамического анализа Java-программ методом модификации виртуальной машины Java[1]

М. К. Ермаков <mermakov@ispras.ru>
С. П. Вартанов <svartanov@ispras.ru>
Факультет вычислительной математики и кибернетики,
Московский государственный университет им. М.В. Ломоносова,
119991, Россия, г. Москва, Ленинские горы, д. 1, с.52

Аннотация. На настоящий момент при разработке программного обеспечения активно используются методы автоматического статического и динамического анализа программ. Так, динамический анализ предоставляет возможности обнаруживать дефекты и ошибки проектирования, проявляющиеся во время выполнения программ, требуя минимального участия эксперта. При проведении динамического анализа распространены два подхода к исследованию выполнения программы — внешний мониторинг, осуществляющийся системными средствами и отладчиками, и инструментация программного кода — модификация и внедрение дополнительной функциональности. При анализе исполняемого кода во внутреннем представлении, обрабатываемого интерпретатором или виртуальной машиной, становиться возможно применение третьего подхода — мониторинга средствами самого интерпретатора или виртуальной машины. В данной статье рассмотрены возможности подобного подхода на примере проведения анализа использования динамической памяти в виртуальной машине Dalvik операционной системы Android.
Работа над представленным методом обусловлена ограничениями виртуальной машины Dalvik, приводящими к невозможности использования большого количества существующих инструментов профилирования памяти. В данной статье рассмотрены непосредственные модификации виртуальной машины Dalvik, включающие расширение поддержки стандартного протокола Java Debug Wire Protocol, для отслеживания событий выделения, освобождения и доступа к памяти. Дополнительно приведён обзор возможностей разработанного отладчика, обеспечивающего регистрацию данных событий в режиме реального времени для последующей обработки. Представленный отладчик, основанный на существующем средстве Dalvik Debug Monitor, позволяет проводить работу одновременно с несколькими активными процессами и обработку результатов множественных запусков. В рамках практических экспериментов были рассмотрен набор стандартных пользовательских приложений

[1] Исследование проводится в рамках научно исследовательских работ Института системного программирования РАН в 2014—2017 годах.

Android, выполняющихся на модифицированной версии виртуальной машины Dalvik под контролем разработанного отладчика. Эксперименты позволили выявить некоторые базовые особенности работы с памятью, включающие активное использование массивов примитивных типов языка Java и неэффективность использования памяти, выделяемой для объектов классов, отвечающих за отображение элементов графического интерфейса приложений.

Ключевые слова: динамический анализ программ; профилирование памяти; Java; Android

DOI: 10.15514/ISPRAS-2015-27(2)-2

Для цитирования: Ермаков М. К., Вартанов С. П. Подход к проведению динамического анализа Java-программ методом модификации виртуальной машины Java. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 23-38. DOI: 10.15514/ISPRAS-2015-27(2)-2.

1. Введение

1.1 Подходы к проведению анализа программ

В настоящее время в процессе разработки и сопровождения программного обеспечения значительную роль играют инструментальные средства для отладки и проведения анализа ключевых аспектов времени выполнения целевого приложения (корректное использование ресурсов, предоставляемых устройствами, и эффективность реализации алгоритмов обработки данных и выполнения заявленной функциональности). Среди данных вспомогательных средств, рассчитанных на нативные приложения (исполняемые файлы которых содержат непосредственно машинные инструкции процессора устройства и управляющую информацию, считываемую загрузчиком операционной системы) можно выделить две большие группы:

- Системные средства отладки и трассировки, предоставляющие стандартные возможности управления выполнением программы и извлечения необходимой информации;
- Средства инструментации кода, производящие модификацию отдельных частей программы для добавления дополнительной функциональности по взаимодействию с некоторым управляющим инструментом во время выполнения или непосредственной генерации данных во время выполнения для последующей обработки.

Средства второй группы являются в высокой степени гибкими и позволяют автоматически производить сложные модификации кода целевого приложения для достижения нужного результата. Непосредственная настройка модификации кода для определённых типов структурных и функциональных единиц (функции, блоки, отдельные инструкции) позволяет эффективно выполнять поставленную задачу. Тем не менее, проведение инструментации неотделимо от проблемы побочных эффектов. Гарантированным побочным

эффектом является непосредственно выполнение действий в рамках анализа программы и увеличение объёма ресурсов, потребляемых программой. При некорректной реализации инструментационного кода побочные эффекты могут вызывать нарушение базовой функциональности программы и появление критических дефектов.

В то же время системные инструменты обычно обеспечивают высокую степень изолированности от выполняемого кода и извлечение данных без влияния на ресурсы, отведённые целевому приложению во время исполнения (за исключением ресурса процессорного времени).

1.2 Анализ интерпретируемых языков

Для приложений, файлы которых содержат интерпретируемый код, наличие дополнительного функционального уровня (непосредственно интерпретатора) вносит ряд особенностей. Описанное выше разбиение на две группы инструментов анализа расширяется до трёх групп инструментов по степени абстракции от внутренней организации исследуемых приложений [1]:

- Системные средства могут быть применены только к интерпретатору, что значительно уменьшает возможность подробного анализа целевого кода — полученные результаты в основном описывают поведение интерпретатора, внешнюю статистику по использованию ресурсов и эффективности реализации алгоритмов самого интерпретатора. Тем не менее, при использовании разметки исполняемых файлов и библиотек интерпретатора возможно учитывать внутреннюю структуру и логику кода анализируемых программ [1,2].
- Средства инструментации, направленные на работу с интерпретируемым кодом, позволяют создавать инструменты анализа, применимые (при наличии интерпретатора) на разных машинных архитектурах и операционных системах.
- Третья группа средств анализа основана на прямом взаимодействии с интерпретатором — использовании подключаемых агентов и отладчиков. Также возможно непосредственное изменение механизмов интерпретации или создание нового интерпретатора с встроенным функционалом по извлечению специфической информации при выполнении кода анализируемой программы.

Как и для нативных приложений, системные средства оказывают минимальное влияние на выполнение анализируемой программы, однако не имеют возможности оперировать высокоуровневыми конструкциями целевой программы. Инструментация кода оказывает наибольшее влияние на выполнение программы, однако позволяет проводить анализ на точечном уровне как по выбору исследуемых частей программы, так и по доступу к параметрам инструкций, блоков и функций. В то же время непосредственная

интеграция анализа с интерпретатором позволяет манипулировать данными приложения, предоставляет доступ ко всему контексту выполнения программы, и вносит меньший уровень влияния на внутренние ресурсы, используемые целевым приложением (так, например, для языка Java инструментационный код должен работать с кучей и стеком виртуальной машины, в то время как модификация самой виртуальной машины Java позволит проводить извлечение информации без всякого влияния на данные области памяти). Недостатком подхода анализа на уровне интерпретатора является затратность реализации и настройки инструмента анализа, а также зависимость от особенностей непосредственно интерпретатора.

1.3 Анализ использования памяти в Java-программах

Рассмотрим подробно программы, написанные на языке Java и выполняющиеся на виртуальной машине Java, с точки зрения вопроса эффективности использования памяти. Использование автоматических и полуавтоматических средств анализа, позволяющих обнаруживать ошибки работы с памятью, снижает вероятность наличия критических дефектов в конечной выпускаемой версиях разрабатываемых программ. Целый набор средств (YourKit Java Profiler [3], JProfiler [4], Eclipse MAT [5], hprof [6] и др.) активно используется при разработке приложений Java. Большинство данных инструментов нацелено на официальную реализацию виртуальный машины HotSpot и на стандарты дополнительных функций виртуальных машин Java. Соответственно, для альтернативных версий виртуальных машин Java данные средства либо не могут быть применены напрямую, либо имеют целый ряд ограничений, в то время как проведение анализа использования памяти для данных виртуальных машин остаётся по-прежнему актуальной задачей.

В рамках данной статьи будет рассматриваться задача анализа использования памяти приложениями операционной системы Android, выполняющимися на виртуальной машине Dalvik. В качестве решения данной задачи предлагается проведение модификации виртуальной машины Dalvik с целью добавления возможности сохранения информации о выделении, освобождении и использовании памяти и передачи данной информации с устройства Android на хост-устройство для последующей обработки. В части 2 статьи будут рассмотрены основные положения предметной области (виртуальная машина Dalvik, JDWP — протокол взаимодействия с отладчиком для виртуальных машин Java [7], подходы к проведению анализа использования памяти). В части 3 будет дано описание проведённых в рамках исследования модификаций виртуальной машины Dalvik, обеспечивающих сохранение и передачу необходимой для оценки использования памяти информации. Часть 4 содержит краткое описание клиента, обеспечивающего приём и организацию данных от виртуальной машины Dalvik на хост-устройстве, и оценку тенденций использования памяти для ряда стандартных приложений системы Android. Наконец, в части 5 будут рассмотрены особенности

представленного подхода в контексте результатов тестовых запусков и возможные направления дальнейших исследований.

2. Анализ использования памяти для приложений Android

2.1 Виртуальная машина Dalvik

Виртуальная машина Dalvik является составной частью операционной системы Android, предназначенной для запуска пользовательских и стандартных системных приложений. Dalvik обладает рядом отличий от стандартных виртуальных машин, использующихся для выполнения Java-приложений:

- Формат байт-кода приложений (dex) для виртуальной машины Dalvik отличается от стандартного формата байт-кода Java. Схема разработки приложений для операционной системы Android предполагает написание кода приложения, генерацию стандартного байт-кода и его трансформацию в формат dex средствами, предоставляемыми в рамках набора инструментов разработки программного обеспечения Android (Android SDK).
- Реализация Dalvik включает поддержку набора стандартных функций виртуальных машин Java (протокол JDWP, генерация слепков кучи, подключение нативных библиотек через интерфейс JNI).
- В виртуальной машине Dalvik отсутствует поддержка подключаемых динамических агентов (протокол JVMTI).
- Код виртуальный машины Dalvik находится в открытом доступе (как и все основные составляющие операционной системы Android).

Данные свойства определяют ряд ограничений на возможность создания инструментов анализа:

- Использование внутреннего формата байт-кода не позволяет напрямую применять имеющиеся средства статической инструментации Java байт-кода (полноценных средств для проведения подобных манипуляций для формата dex на данный момент не существует). Инструментация байт-кода возможна косвенно как промежуточный этап создания приложения Android — написание кода на Java, компиляция в байт-код Java, инструментация байт-кода Java, трансформация байт-кода Java в байт-код dex. В этом случае, однако, возрастает степень накладных расходов на выполнение инструментационного кода (этап трансформации в dex ограничивает оптимизацию инструментационного кода).
- Отсутствие поддержки протокола JVMTI не позволяет использовать имеющиеся средства анализа и динамической инструментации кода, реализованные для стандартных виртуальных машин Java.

В то же время, открытость кода виртуальной машины Dalvik предоставляет возможности внесения любых модификаций для проведения необходимого анализа. Указанная выше поддержка стандартного протокола коммуникации с отладчиками JDWP и работы с динамической кучей означает наличие базы, на основе которой возможна эффективная реализация инструмента анализа.

2.2 Протокол JDWP

Протокол Java Debug Wire Protocol (JDWP) представляет собой стандарт низкоуровневого взаимодействия виртуальной машины Java с некоторым средством, производящих отладку целевого приложения. В данный протокол входит описание формата поддерживаемых пакетов, типов данных, отвечающих функциональным элементам, используемым виртуальной машиной (классы, объекты, потоки управления и т.д.). Виртуальная машина Java, предоставляющая поддержку протокола отладки, может быть запущена с соответствующими опциями для инициации прослушивания канала связи, по которому может произойти попытка соединения со стороны средства отладки. В протоколе зафиксирована последовательность пакетов аутентификации и порядок реакции на предусмотренные запросы и команды. Список запросов и команд включает следующие основные группы:

- Базовые команды управления виртуальной машиной — остановка и возобновление потоков управления, инициация сборки мусора и т.д.
- Запросы информации о сущностях, используемых программой, — классах, объектах и т.д.
- Создание точек наблюдения и триггеров событий, автоматически отправляющих данные отладчику при фиксировании некоторой ситуации.
- Команды создания и модификации параметров объектов и классов.

Значительным преимуществом протокола JDWP является возможность расширения пакетов за счёт использования пользовательских кодировок. При реализации соответствующей поддержки на уровне виртуальной машины можно определять фактически произвольные команды и запросы для извлечения нужной информации или влияния на целевое приложение.

Поддержка протокола JDWP в виртуальной машине Dalvik является неполной. В частности, отсутствует возможность создавать триггеры на доступ к полям объектов и ряд других опций протокола.

Протокол JDWP в Dalvik был расширен группой пакетов для работы специального средства Dalvik Debug Monitor Service (DDMS) [8], предоставляющего возможности получения информации по аллокациям памяти, работе потоков управления и содержимого стандартного журнала, используемого большинством приложений операционной системы Android. Данные по аллокациям памяти, получаемые сервисом отладки Android, позволяют проводить анализ использования памяти целевым приложением.

Генерируемые наборы данных, однако, не позволяют проводить полный анализ выделения, освобождения и использования памяти.

2.3 Анализ памяти по слепкам кучи

Виртуальная машина Dalvik поддерживает возможность извлечения полного слепка кучи по запросу от отладчика или анализатора. Данный слепок содержит информацию о всех активных в данный момент времени объектах, а также информацию о связи между ними. Связи между объектами рассчитываются по возможности осуществить доступ к одному объекту при наличии доступа к другому (например, объекты A и B связаны, если ссылка на B записана в одно из полей объекта A). В куче выделяются корневые объекты [9], создаваемые близко к точке входа в приложение и активные в течение всего времени выполнения программы. Если для объекта существует цепочка связей до одного из корневых объектов, то он считается активным. Если такой цепочки нет, то невозможно осуществить доступ к данному объекту, а значит он не может использоваться в программе и память, выделенная для него, может быть освобождена на ближайшем этапе сборки мусора.

Анализаторы, направленные на обработку слепков кучи (Eclipse MAT, hprof), позволяют проводить сравнительную оценку состояния кучи до и после некоторого события в программе, что, в свою очередь, позволяет эксперту выявлять потенциальные проблемы утечек памяти и поддержки излишнего количества долгоживущих объектов, реально не используемых в программе. Подобный анализ, тем не менее, не позволяет получать информацию обо всех созданных объектах (генерация слепка кучи — достаточно дорогая операция, которая требует остановки выполнения виртуальной машины) и не предоставляет информацию о непосредственном использовании выделенной памяти.

2.4 Анализ памяти в режиме реального времени

Большая группа анализаторов (например, указанное выше средство JProfiler) использует протокол JVMTI для подключения и работы агента, обеспечивающего прослушивание ряда событий, включающих создание объектов и освобождение памяти во время проведения сборки мусора. Данные событий, описывающие типы создаваемых и освобождаемых объектов, точки создания и другие параметры, поступают на клиентское приложение для обновления базы данных в реальном времени. Собираемая статистика может быть пополнена и слепками кучи, создаваемыми в ключевые моменты времени.

Отсутствие поддержки протокола JVMTI в виртуальной машине Dalvik делает невозможным применение данных средств. Тем не менее, описанный подход (получение всей информации в режиме реального времени) представляется наиболее мощным подходом анализа использования памяти.

2.5 Анализ памяти в Dalvik в режиме реального времени

Описанные выше особенности реализации виртуальной машины Dalvik и возможные подходы к анализу использованию памяти обуславливают основные составляющие выбранного подхода:

- Проведение анализа в реальном времени — сбор всех событий выделения, освобождения и использования памяти, и пересылка собранных данных на хост-устройство;
- Реализация сбора событий путём проведения модификации непосредственно кода виртуальной машины Dalvik;
- Упаковка собираемых данных в специализированные пакеты и передача пакетов отладчику на основе протокола JDWP;
- Реализация программного средства, осуществляющего получение и обработку данных, генерируемых модифицированной виртуальной машиной Dalvik, на хост-устройстве.

3 Модификации виртуальной машины Dalvik

3.1 Общая структура модификаций

Модификации виртуальной машины Dalvik нацелены на выполнение дополнительных действий при возникновении следующих событий:

- Создание объекта определённого типа;
- Освобождение памяти, выделенной под объект, при проведении сборки мусора;
- Загрузка класса и инициализация его параметров;
- Использование объекта (вызов метода объекта, доступ к полю объекта, синхронизованный доступ к полю объекта, доступ к элементу массива, базовые методы чтения массивов JNI, нативные реализации методов базовых классов, таких как java.lang.String).

Обработка данных событий заключается в извлечении параметров событий и упаковки данных параметров в специализированные пакеты для последующей передачи на хост-устройство. Накапливаемая в пакетах информация буферизуется; размеры внутренних буферов выбраны таким образом, чтобы обеспечивать частые обновления состояния памяти приложения, избегая перегрузки канала связи между устройством Android и хост-устройством.

Полное описание содержимого пакетов данных представлено на следующей схеме, использующей обозначения:

- REquest ALlocations (REAL) — пакет с данными о созданных объектах,
- REquest FReed (REFR) — пакет с данными об освобождённых объектах,

- OBject USage (OBUS) — пакет с данными об использовании объектов,
- CLaSS information (CLSS) — пакет с данными о загруженных классах.

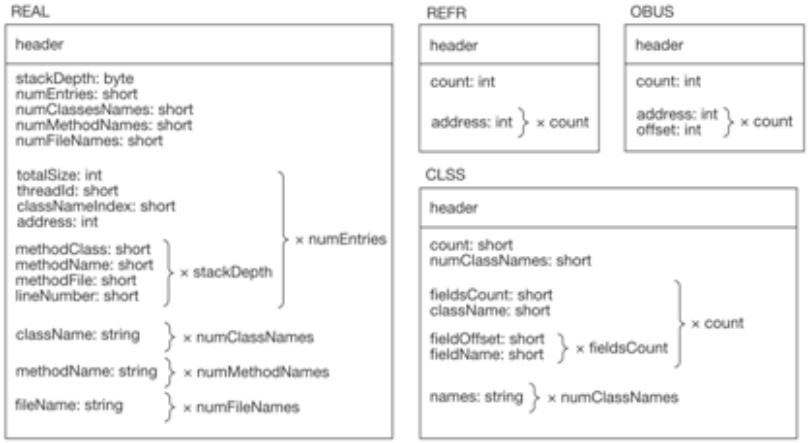

Рис. 1: Форматы пакетов данных, генерируемых модифицированной виртуальной машиной Dalvik

3.2 События создания объекта

Создание объекта и выделение памяти в виртуальной машине Dalvik происходит в рамках единственной процедуры, которая в свою очередь вызывается несколькими модулями (модуль создания стандартных объектов по обрабатываемому байт-коду, модуль загрузки классов и создания объектов-контейнеров для данных классов, управляющий модуль обработки запросов JNI и т.д.), что позволяет производить точечную обработку данных событий. Для работы средства DDMS протокол JDWP уже имеет расширение, описывающее пакеты с информацией о выделенных объектах. Данные пакеты имеют сложную структуру, описывающую основные параметры создаваемых объектов:

- количество объектов пакета;
- классы объектов (индексы множества строк);
- размеры выделенной памяти;
- идентификаторы потоков;
- реальные адреса выделенной памяти (для идентификации объектов);
- счётчики количества элементов во множестве строк;
- стеки вызовов (определяющие точки кода, в которых произошло создание объектов);
- множество строк, разделяемое между всеми остальными полями для сокращения размера пакета.

3.3 События освобождения памяти

Сборка мусора в виртуальной машине Dalvik осуществляется на основе алгоритма MarkSweep [9]. Работа алгоритма состоит из нескольких этапов, наиболее важными из которых является разметка объектов по степени активности (корневые объекты кучи помечаются как активные, все объекты, связанные по ссылке с активными, также считаются активными) и обход меток с целью освобождения памяти, занимаемой неактивными объектами. Во время этапа обхода работа с объектами осуществляется по реальным адресам памяти. Именно данные адреса сохраняются во внутренние списки с целью пересылки на хост-устройство. Пакеты данных формируются из двух групп адресов — использованных объектов и неиспользованных объектов (по биту структуры объекта при проведении частичного анализа использования памяти). При проведении полного анализа использования памяти разделение на две группы не используется.

3.4 События использования памяти (неполный анализ)

Виртуальная машина Dalvik содержит значительное количество модулей, осуществляющих прямой доступ к содержимому объектов приложения. Реализованные в рамках исследования модификации виртуальной машины Dalvik направлены на перехваты событий доступа к объектам в следующих модулях:

- Внутренние модули интерпретации операций байт-кода dex: доступ к полю объекта заданного типа, доступ к элементу массива по заданному индексу, вызов метода объекта.
- Частные реализации методов базовых классов (java.lang.String, java.lang.System и т. д.).
- Реализации основных операций интерфейса JNI: работа с массивами, трансформация внутренних типов JNI в объекты Java, выделяемые на куче.

В рамках проведения неполного анализа использования памяти обработка этих событий приводит к взведению флага использования во внутренней структуре виртуальной машины Dalvik, определяющей объект Java. Данные подход не приводит к необходимости записи информации во внутренний список и передачи на хост-устройство, однако позволяет определять объекты приложения только как использованные/неиспользованные без указания о том, какая часть выделенной памяти была прочитана за время существования объекта.

3.5 События использования памяти (полный анализ)

При проведении полного анализа использования памяти в качестве событий рассматриваются все вышеуказанные группы. Однако вместо взведения флага,

информация о том, какое поле объекта или какой элемент массива был использован, записывается во внутренний список для последующей передачи на хост-устройство. Это значительно повышает накладные расхода на проведение анализа по сравнению с неполным подходом, однако позволяет получать более точные данные об эффективности использования созданных объектов.

Пакеты данных о использовании объектов содержат непосредственно адреса использованных объектов и смещения в памяти, по которым произошёл доступ на чтение.

3.6 События загрузки классов Java

Для корректной работы полного анализа использования памяти в рамках модификаций, внесённых в виртуальную машину Dalvik, была добавлена обработка событий загрузки классов (как системных, так и пользовательских). Данные о полях класса и внутренних смещениях, используемых виртуальной машиной Dalvik, формируются в список и в пакеты, передаваемые на хост-устройство. Впоследствии, они используются для отображения смещений, передаваемых в пакетах использования объектов, на имена полей класса.

4 Практические результаты

4.1 Средство сборки и обработки данных анализа

На основе библиотеки ddmlib [10] был разработан специализированный инструмент, предназначенный для работы на рабочей станции, к которой подключено устройство Android, и обработки данных, получаемых от анализируемого приложения. Работа с инструментом возможна как в автоматическом режиме для трассировки длительной сессии выполнения анализируемого приложения, так и в интерактивном режиме с целью более строгого контроля за выполнением анализируемого приложения.

В число функций разработанного инструмента входят следующие:

- Установление соединения с процессом, в котором выполняется виртуальная машина, и инициализация параметров передачи пакетов от устройства и команд на устройство.
- Активное прослушивание канала связи, приём пакетов данных и обновление общего набора информации, полученного во время сессии работы с анализируемым приложением.
- Командный пользовательский интерфейс для отображения общего набора информации по работе анализируемого приложения с предоставленной виртуальной памятью. Данный интерфейс предоставляет набор настраиваемых фильтров для отображения данных, возможность сохранения и загрузки данных сессии для

анализа в автономном режиме (без активного взаимодействия с устройством Android).

- Работа с несколькими наборами данных, полученными в разные моменты времени в рамках одного запуска приложения или в рамках нескольких запусков приложения.
- Отсылка контрольных запросов и команд виртуальной машине Dalvik с целью выполнения задач отладки (временная остановка и возобновление выполнения, запроса запуска сборки мусора, запрос создания слепка кучи для использования возможностей существующих инструментов профилировщиков Java).

4.2 Анализ стандартных приложений Android

Модифицированная версия виртуальной машины Dalvik была использована для исследования работы с памятью ряда стандартных приложений операционной системы Android:

- Browser – браузер ресурсов сети Интернет;
- Calendar – приложение, предоставляющее возможности органайзера;
- Contacts – менеджер списка контактов и групп;
- Mms — приложение для обмена текстовыми и мультимедийными сообщениями;
- Deskclock – экранные часы и менеджер будильников.

Для указанных приложений был проведён ряд запусков на выполнение некоторой последовательности действий после подключения средства приёма пакетов. Запуски производились на устройстве PandaBoard [11] с версией операционной системы Android 4.2.2, подсоединённой к хост-устройству через подключение USB. Замедление работы приложений по сравнению с работой без проведения обработки событий, рассмотренных ранее, не превышало двухкратного. Потеря производительности была обусловлена в большей степени значительным количеством передаваемых данных при ограниченной (2-3 Мб/с) пропускной скорости соединения (внутренняя организация и передача данных через утилиту adb, предоставляемую в рамках набора инструментов разработки программного обеспечения Android (Android SDK).

Проведённые запуски выявили ряд закономерностей использования памяти в приложениях.

Для всех рассмотренных приложений основная доля выделенной памяти (от 50% для приложения Calendar до 80% для приложения Browser) приходится на объекты типов char[], byte[] и int[].

- Объекты типа byte[] являются долгоживущими и используются для хранения изображений и контейнеров, отображаемых на экране устройства. Классами, осуществляющими создание наибольшей части

объектов данного типа являются java.nio.ByteBuffer, android.graphics.BitmapFactory и android.graphics.Bitmap.

- Объекты типа char[] создаются в значительном количестве при работе со строками классами java.lang.String и java.lang.StringBuilder.
- Объекты типа int[] являются короткоживущими и создаются в основном при работе системы исключений классом java.lang.Throwable. Обработка ситуаций возникновения исключений, проводимая виртуальной машиной Dalvik, включает в себя набор действий, выполнение которых является более затратным (по времени и ресурсам), чем реализация в программном коде развёрнутой системы проверок ограничений. Большое количество исключений, генерируемых при выполнении приложений, позволяет говорить о целесообразности проведения оптимизации методов, приводящих к появлению данных исключений.

Доля использованной памяти для объектов классов с большим количество полей, например, стандартных классов контейнеров и графических элементов операционной системы Android, в значительной степени варьируется.

В качества примера можно рассмотреть использование класса графических объектов android.widget.TextView, активно применяемого для отображения данных в приложении. Следующий график показывает распределение доли использованных байтов на момент освобождения памяти объектов TextView в рамках сессии работы с приложением.

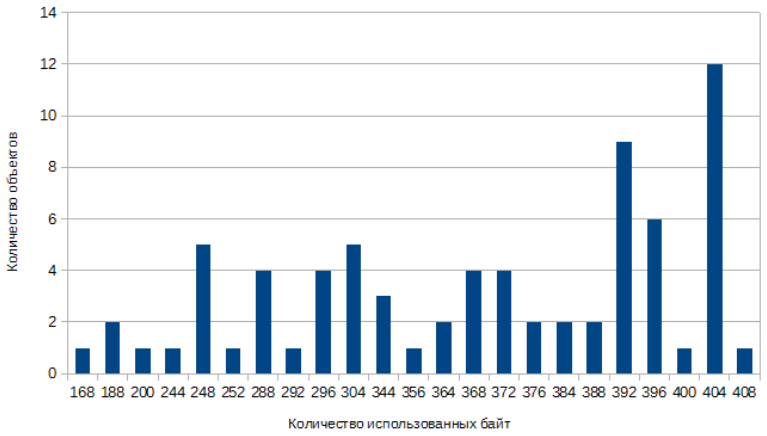

Рис. 2: Распределение доли использованной памяти объектов android.widget.TextView

В то время как полный размер аллокации каждого объекта android.widget.TextView составляет 688 байт, менее, чем 412 байт было

использовано для каждого созданного объекта данного типа за время его существования. В данные расчёты входят как поля непосредственно самого класса android.widget.TextView, так и классов, от которых данных класс наследуется. Подобные результаты свидетельствуют о возможности оптимизации классов, используемых для отображения пользовательского интерфейса.

5 Заключение

Предложенный подход, основанный на получении и обработке данных от модифицированной виртуальной машины Dalvik, предоставляет возможности определения критических и других проблем работы с динамической памятью целевого приложения, выполняющегося в рамках виртуальной машины. Получаемая статистика позволяет осуществлять профилирование как кода пользовательских приложений, так и компонентов системных классов Android. Возможность совмещения и сравнительной оценки наборов данных, соответствующих независимым сессиям выполнения приложения, позволяет проводить анализ стабильности целевого приложения при различных сценариях его использования. Возможность одновременного анализа нескольких процессов, в контексте которых выполняются виртуальные машины, позволяет исследовать взаимодействие приложений.

Тем не менее, существует ряд вопросов и возможностей получения некорректных результатов при активном использовании нативного кода JNI. На данный момент покрыт ряд операций интерфейса JNI по доступу к данным объектов (прежде всего, массивов), однако непосредственная обработка содержимого данных объектов проводится внутри модулей, реализованных в машинных кодах целевой архитектуры, и недоступна виртуальной машине Dalvik. При активном использовании подобных модулей появляется вероятность наличия некорректных данных в итоговой статистике из-за невозможности перехвата событий доступа к содержимому объектов.

В качестве потенциального решения данной проблемы рассматривается возможность проведения совмещённого анализа на основе модификаций виртуальной машины Dalvik и дополнительных методов, осуществляющих контроль за выполнением внешнего для виртуальной машины кода. В качестве подобных методов может быть использована статическая бинарная инструментация библиотек, реализованных в машинных кодах целевой архитектуры, или перехват механизмов интерфейса JNI для запуска инструмента динамического анализа.

Список литературы

[1]. Stephen Kell, Danio Ansaloni, Walter Binder, Lukáš Marek. The JVM is not observable enough (and what to do about it). Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages, 2012. pp. 33-38. doi: 10.1145/2414740.2414747

[2]. Страница документации механизма Dtrace для виртуальной машины HotSpot (http://docs.oracle.com/javase/6/docs/technotes/guides/vm/dtrace.html) [HTML]. Дата обращения: 24.06.2015.
[3]. Страница продукта YourKit Java Profiler (https://www.yourkit.com/features/) [HTML]. Дата обращения: 30.06.2015.
[4]. Страница продукта Jprofiler (https://www.ej technologies.com/products/jprofiler/overview.html) [HTML]. Дата обращения: 30.06.2015.
[5]. Страница проекта Eclipse MAT (https://eclipse.org/mat/) [HTML]. Дата обращения: 30.06.2015.
[6]. Страница документации инструмента hprof (http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html) [HTML]. Дата обращения: 30.06.2015.
[7]. Страница документации протокола JDWP (https://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html) [HTML]. Дата обращения: 23.06.2015.
[8]. Страница описания средства DDMS (http://developer.android.com/tools/debugging/ddms.html) [HTML]. Дата обращения: 30.06.2015.
[9]. Paul R. Wilson. Uniprocessor garbage collection techniques. Proceedings of the International Workshop on Memory Management, 1992. pp. 1-42.
[10]. Репозиторий библиотеки ddmlib (https://android.googlesource.com/platform/tools/base/+/master/ddmlib/) [HTML]. Дата обращения: 30.06.2015.
[11]. Сайт платформы PandaBoard (http://pandaboard.org/) [HTML]. Дата обращения 30.06.2015.

Dynamic Java Program Analysis Using Virtual Machine Modification

M. Ermakov <mermakov@ispras.ru>
S. Vartanov <svartanov@ispras.ru>
Department of Computational Mathematics and Cybernethics, Moscow State University, 1/52, Leninskie Gory, Moscow, Russia, 119991

Abstract. At the present time automatic static and dynamic program analysis methods are extensively used during software development. Dynamic methods in particular allow cost-efficient detection of various runtime issues with minimal expert interaction, thus saving time and resources. Dynamic analysis methods for native applications employ external monitoring instruments and code modification in order to evaluate execution, while programs running under control of an interpreter or a virtual machine it offers a balanced observation layer – the interpreter or virtual machine itself. In this paper we focus on automatic memory profiling methods for Java applications running on Dalvik virtual machine – a part of rapidly growing Android operating system.

Current Dalvik VM limitations make it impossible to use complex Java profilers designed for desktop applications; Dalvik-compatible profiling tools provide insufficient data to perform deep memory usage analysis. In this paper we describe a set of extensions for Dalvik implementation of standard Java Debug Wire Protocol which allow tracking of in-depth program memory events: object allocation, garbage collection, memory read and write access. We present a debugger-level tool based on existing Dalvik Debug Monitor utility, that is compatible with the extended Dalvik VM and allows online tracking of aforementioned events along with base debugging features and gprof memory dump support. As the original DDM utility, our design supports concurrent tracking of multiple applications to identify memory usage issues related to interprocess communication.
We have used the developed system to analyze a set of standard Android applications identifying several memory usage trends, including prevalence of primitive array allocations and inefficient use of memory among standard Android GUI Java classes.

Keywords: dynamic program analysis; memory profiling; Java; Android

DOI: 10.15514/ISPRAS-2015-27(2)-2

For citation: Ermakov M.K., Vartanov S.P. Dynamic Java Program Analysis Using Virtual Machine Modification. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 23-38 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-2.

References

[1]. Stephen Kell, Danio Ansaloni, Walter Binder, Lukáš Marek. The JVM is not observable enough (and what to do about it). Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages, 2012. pp. 33-38. doi: 10.1145/2414740.2414747
[2]. Dtrace for HotSpot VM (http://docs.oracle.com/javase/6/docs/technotes/guides/vm/dtrace.html) [HTML]. Retrieved: 24.06.2015.
[3]. YourKit Java Profiler tool overview (https://www.yourkit.com/features/) [HTML]. Retrieved: 30.06.2015.
[4]. JProfiler tool overview (https://www.ej technologies.com/products/jprofiler/overview.html) [HTML]. Retrieved: 30.06.2015.
[5]. Eclipse MAT project (https://eclipse.org/mat/) [HTML]. Retrieved: 30.06.2015.
[6]. hprof tool documentation (http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html) [HTML]. Retrieved: 30.06.2015.
[7]. JDWP protocol overview (https://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html) [HTML]. Retrieved: 23.06.2015.
[8]. DDMS tool overview (http://developer.android.com/tools/debugging/ddms.html) [HTML]. Retrieved: 30.06.2015.
[9]. Paul R. Wilson. Uniprocessor garbage collection techniques. Proceedings of the International Workshop on Memory Management, 1992. pp. 1-42.
[10]. ddmlib project online repository (https://android.googlesource.com/platform/tools/base/+/master/ddmlib/) [HTML]. Retrieved: 30.06.2015.
[11]. PandaBoard project overview (http://pandaboard.org/) [HTML]. Retrieved: 30.06.2015.