ТЕХНОЛОГИЧЕСКОЕ ОБРАЗОВАНИЕ

Чилингарова Софья Александровна

МЕТОДЫ ОПТИМИЗАЦИИ ДЛЯ ДИНАМИЧЕСКИХ (JUST-IN-TIME) КОМПИЛЯТОРОВ

Часть 2. Примеры современных динамических компиляторов

В части 1 настоящей статьи были рассмотрены общие принципы посторения подсистемы оптимизирующей динамической компиляции в виртуальных машинах. В этой части мы покажем на примерах, как данные общие принципы реализуются в конкретных динамических компиляторах современных виртуальных машин. Мы коротко опишем четыре широко известные системы, включающие динамические компиляторы для Java и CIL: Jikes JVM, IBM DK, HotSpot JVM (Sun Microsystems) и StarJit (Intel Research Lab.).

JIKES JVM

Jikes (в первых версиях – Jalapeto) – виртуальная Java-машина с открытым исходным кодом, написанная преимущественно на самом языке Java [4]. Виртуальная машина Jalapeto была разработана в конце 1990-х годов в научно-исследовательской лаборатории IBM. Основной целью проекта первоначально было – выяснить экспериментально, возможно ли написать виртуальную Java-машину собственно на языке Java [3]. Эксперимент увенчался успехом, а сама виртуальная машина впоследствии была переименована в Jikes и передана в открытую разработку. В настоящее время Jikes является популярным исследовательским open-source проектом. Just-in-time компилятор Jikes по производительности во многих случаях не уступает компиляторам коммерческих Java-машин.

АРХИТЕКТУРА

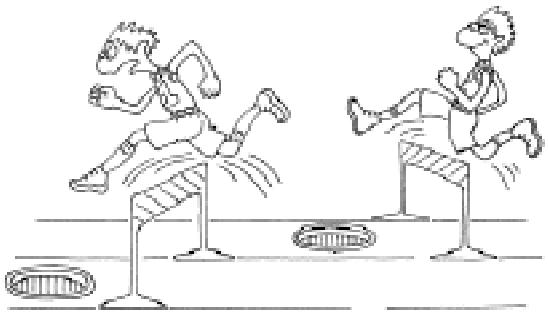
Общая архитектура подсистемы динамической компиляции Jikes показана на рисунке 1. Система включает в себя базовый (неоптимизирующий) компилятор, оптимизирующий компилятор с тремя уровнями оптимизации, подсистему профилирования и контроллер. Все методы при первом вызове компилируются базовым компилятором.

Так как сама Java-машина также, в основном, написана на языке Java, для компиляции ее классов используется та же самая подсистема компиляции. Для ускорения загрузки при первом запуске создается загрузочный образ ядра Java-машины, содержащий скомпилированные классы, который используется при последующих запусках на той же платформе.

Система профилирования встроена в механизм управления Java-потоками. Jikes реализует схему потоков $M \times N$ – M Java-потоков на N потоков операционной системы. Java-потоки, работающие на одном процессоре, переключаются в специальных точках переключения, встроенных в код – *yield points* [3; 4; 8]. В этих точках может

вызываться сборщик мусора, работать профайлер, производиться замещение на стеке и другие служебные действия. Профайлер вызывается в точках переключения через определенные промежутки времени. Управление в точке переключения передается коду, который определяет, какой метод в данный момент выполняется, и наращивает счетчик для этого метода. Также собирается информация о том, кем был вызван метод, и на основе этой информации строится динамический граф вызовов [8] (рисунок 1).

Когда счетчик для некоторого метода превышает определенное пороговое значение, подсистема обработки данных профилирования генерирует событие, которое передается контроллеру, и контроллер ставит метод в очередь на повторную компиляцию [4; 8]. На основе данных профилирования контроллер определяет, на каком уровне оптимизации нужно компилировать конкретный метод и в каких местах делать inline-подстановки. Чтобы принять решение о том, нужно ли компилировать конкретный метод, контроллер подсчитывает предполагаемую выгоду и предполагаемые издержки компиляции. Предполагаемые издержки определяются как приблизительная оценка времени, необходимого для компиляции данного метода на данном уровне оптимизации. Предполагаемая выгода – оценка разности времени выполнения оптимизированного и неоптимизированного кода в будущем, исходя из предположения, что поведение программы будет примерно таким же, как на момент получения профиля, и что в дальнейшем будет сделано в два раза больше вызовов целевого метода, чем было сделано до тех пор. Контроллер оценивает эти две величины и выбирает стратегию, которая минимизирует общее время выполнения [8].



Java-потоки, работающие на одном процессоре, переключаются в специальных точках переключения...

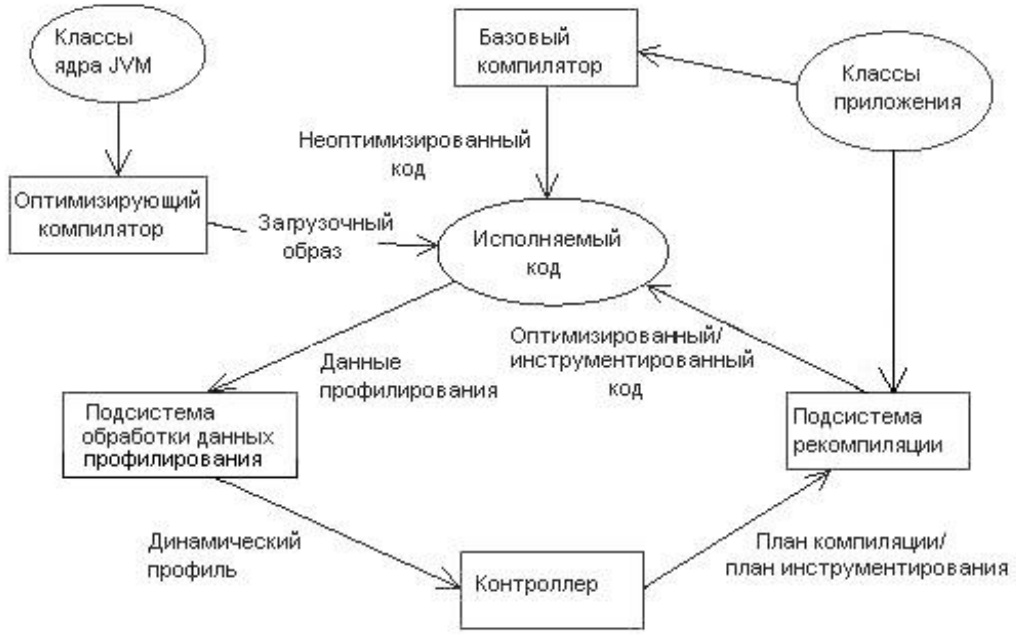
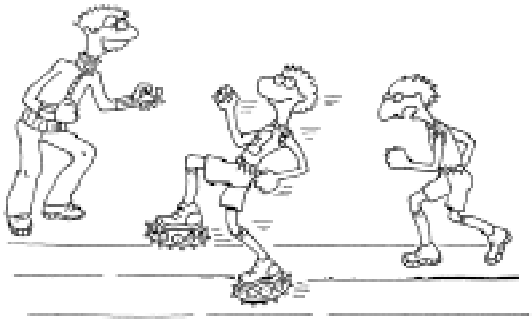


Рисунок 1. Общая архитектура адаптивной системы динамической компиляции Jikes RVM [8, Figure 2.2].



...оценка разности времени выполнения оптимизированного и неоптимизированного кода в будущем...

Оптимизирующий компилятор выбирает методы из очереди и компилирует их в отдельном потоке. Когда компиляция завершена, все ссылки на метод заменяются ссылкой на вновь скомпилированный код. Если метод содержит очень длительный цикл, производится замещение на стеке. Методы, уже скомпилированные с первым или вторым уровнем оптимизации, могут быть поставлены в очередь на перекомпиляцию с более высоким уровнем оптимизации.

Применяется также техника *динамических inline-подстановок*. Если собранный профиль показывает, что некая цепочка вызовов реализуется особенно часто, корневой вызывающий метод перекомпилируется с подстановкой всей цепочки. Если данный метод уже скомпилирован с самым высоким уровнем оптимизации, контроллер исследует возможную выгоду от перекомпиляции его с новой стратегией inline-подстановок.

Более подробная информация о поведении методов, уже выделенных, как «горячие», собирается путем динамической подстановки инструментованного кода [8]. Создаются две версии кода: инструментованная и версия с точками проверки (checking code). Вторая выполняется основную часть времени и не содержит инструментария, а содержит только точки проверки (check): на входе в метод и на входе в итерацию цикла. В точках проверки проверяется условие, определенное конфигурацией системы, и, если оно верно, управление передается инструментованной версии. Инструментованная версия выполняет небольшой участок кода, например, одну итерацию цикла, и передает управление версии с точками проверки. Проверяемым условием может быть число входов в точку проверки с момента последнего выполнения инструментованного кода или истечение определенного промежутка времени [8] (рисунок 2).

ПРОМЕЖУТОЧНОЕ ПРЕДСТАВЛЕНИЕ

На всех уровнях оптимизации используются последовательно три промежуточных представления: высокого уровня (HIR), низкого уровня (LIR) и машинного уровня (MIR) [3; 4]. Представление высокого уровня – это байткод, расширенный инструкциями для действий, которые должны выполняться неявно в определенных ситуаци-

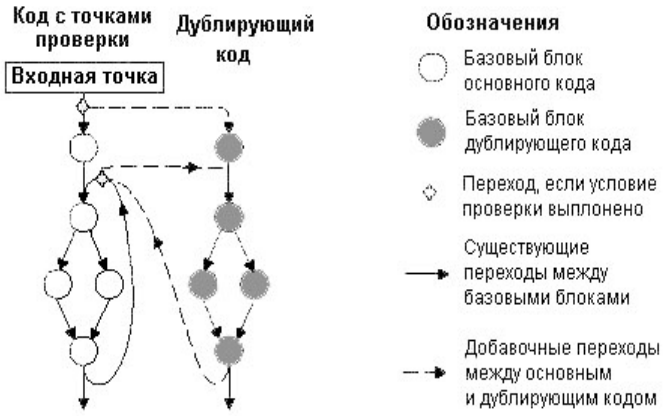


Рисунок 2. Передача управления между кодом с точками проверки и дублирующим кодом [8, Figure 3.3].

ях, например, проверок на null. В нужных местах вставляются точки перехода (yield point). Представление низкого уровня получается из представления высокого уровня добавлением деталей реализации, специфичных для Jikes: например, явно представлена последовательность инструкций для получения ссылки на метод или объект. Промежуточное представление машинного уровня соответствует структуре команд целевой архитектуры и создается из представления низкого уровня парсером типа BURS (Bottom-Up Rewriting System), использующим специальные таблицы для трансляции универсальных инструкций уровня LIR в машинные команды конкретных архитектур.

В таблице 1 приведены инструкции представлений высокого уровня (HIR), низкого уровня (LIR) и машинного уровня (MIR) для простого метода, а на рисунке 3 показано на примере, как BURS-парсер ис-

Таблица 1. **Промежуточное представление в Jikes** [4].

Код на языке Java	**Java-байткод**

`public static void main()`
`{`
`System.out.println("Hello world");`
`}`
	`Method void main()`
	`0 getstatic #2 <Field java.io.PrintStream out>`
	`3 ldc #3 <String "Hello world">`
	`5 invokevirtual #4 <Method void println(java.lang.String)>`
	`8 return`

Инструкции HIR

```
LABEL0
  EG ir_prologue
  G  yieldpoint_prologue
0    getstatic      t0i(java.io.PrintStream,d) = <mem loc: java.lang.System.out>
5 EG null_check     t1v(GUARD) = t0i(java.io.PrintStream,d)
5 EG call           LR = <unused>, virtual"java.io.PrintStream.println
                        (Ljava/lang/String;)V", t1v(GUARD), t0i(java.io.PrintStream,d),
                         string constant @12944 JTOC
  G  yieldpoint_epilogue
     return
     bbend          BB0 (ENTRY)
```

Инструкции LIR

```
LABEL0
  EG ir_prologue
  G  yieldpoint_prologue
0    int_load               t0i(java.io.PrintStream,d) = JTOC(int), 23156
                                <mem loc: java.lang.System.out>
5 EG null_check             t1v(GUARD) = t0i(java.io.PrintStream,d)
     materialize_constant   t2i(java.lang.String) = JTOC(int), string constant @129445
5    get_obj_tib            t3i([Ljava.lang.Object;) = t0i(java.io.PrintStream,d), t1v(GUARD)
5    int_load               t4i([I) = t3i([Ljava.lang.Object;), 160
5 EG call                   LR = t4i([I), virtual"java.io.PrintStream.println
                                (Ljava/lang/String;)V", t1v(GUARD), t0i(java.io.PrintStream,d),
                                 t2i(java.lang.String) JTOC
G    yieldpoint_epilogue
     return
     bbend                  BB0 (ENTRY)
```

Инструкции MIR для процессора PowerPC

```
LABEL0
      ppc_mfspr    R0(int) = LR(int)
      ppc_lwz      R13(int) = PR(int), -40
      ppc_stwu     FP(int) <-- FP(int), -16
      ppc_lwz      R14(int) = PR(int), -28
      ppc_lwz      R13(int) = R13(int), -52
      ppc_cmpi     C2(int) = R14(int), 0
      ppc_ldi      R14(int) = 5091
      ppc_stw      R0(int), FP(int), 24
      ppc_stw      R14(int), FP(int), 4
  EG  ppc_tw       ppc trap <, FP(int), R13(int), <STACK OVERFLOW>
  EG  ppc_bcl      LR = C2(int), ppc <, LABEL2 JTOC
0     ppc_lwz      R3(java.io.PrintStream,d) = JTOC(int), 23156, <mem loc: java.lang.System.out>
5 EG  ppc_lwz      R4([Ljava.lang.Object;) = R3(java.io.PrintStream,d), -12
5     ppc_lwz      R4([I) = R4([Ljava.lang.Object;), 160
      ppc_addis    R5(int) = JTOC(int), 1
      ppc_lwz      R5(java.lang.String) = R5(int), 51776, <mem loc: JTOC @51776>
5     ppc_mtspr    CTR(int) = R4([I)
.......
```

Обозначения: **E** – может быть брошено исключение; **G** – может быть запущен сборщик мусора

пользует таблицы для преобразования представлений: сначала строится дерево зависимостей, затем к каждому узлу снизу вверх применяются правила из таблицы.

ОПТИМИЗАЦИЯ И ДЕОПТИМИЗАЦИЯ

На первом уровне оптимизации (уровень 0) компилятор Jikes выполняет «на лету» при создании промежуточного представления из байткода распространение и свертку констант, удаление избыточных проверок на null и на выход за границы массивов, избыточных преобразований типов, недостижимого кода. Для всех методов, размер которых меньше длины последовательности инструкций вызова, выполняется inline-подстановка. Над промежуточным представлением производятся такие операции, как локальное (на уровне базового блока) удаление избыточных инструкций загрузки и выгрузки, избыточных проверок на исключения, замена маленьких массивов и агрегатов скалярами, на машинном уровне – некоторые простые машинно-зависимые оптимизации, такие как перестановка инструкций. Производится один глобальный просмотр всего промежуточного представления с удалением избыточных присваиваний и недостижимого кода, распространением и сверткой констант. Для распределения регистров применяется простой линейный алгоритм.

На втором уровне (уровень 1) глобальный анализ производится в несколько итераций. Inline-подстановки более агрессивны: подставляются более крупные методы, используются данные профилирования. Многие из оптимизаций, которые сейчас выполняются на уровне 0, раньше выполнялись на уровне 1 [11].

На третьем, самом высоком уровне оптимизации (уровень 2) выполняется оптимизация циклов, строится SSA-представление (имена присваиваются значениям скалярных данных) для глобального анализа, оптимизируется работа с объектами в «куче» [11].

Для отката в случае несоответствия предположений, сделанных при агрессивной оптимизации, реальной ситуации выполнения, используется замещение на стеке [5]. Замещение производится в точке перехода, вызывается неоптимизированная версия, для которой генерируется специальный пролог, транслирующий локальные данные в нужный формат (как было показано в части 1 в разделе «Деоптимизация»).

IBM DK

Виртуальная Java-машина IBM DK – одна из первых Java-машин и первая, где использовался динамический компилятор. Первые подробные описания многоуровне-

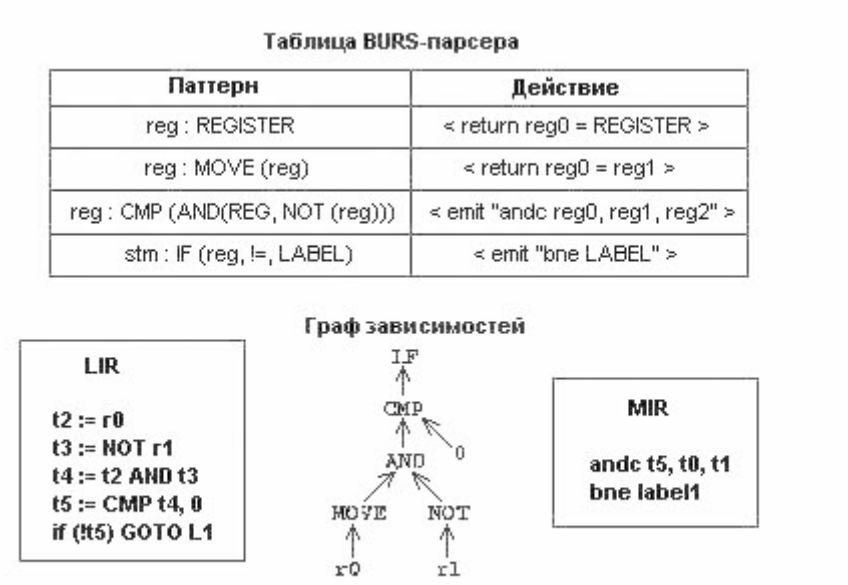


Рисунок 3. Преобразование кода низкого уровня в код машинного уровня в Jikes [4].

вого динамического компилятора IBM DK появились в конце 1990-х – 2000 году. Компилятор создан в научно-исследовательской лаборатории IBM в Токио и развивается как исследовательский проект до настоящего времени. Нововведения, показавшие хорошие результаты, включаются в коммерческий продукт.

...оптимизируется работа с объектами в «куче».

АРХИТЕКТУРА

Общая архитектура системы многоуровневой динамической компиляции IBM DK показана на рисунке 4.

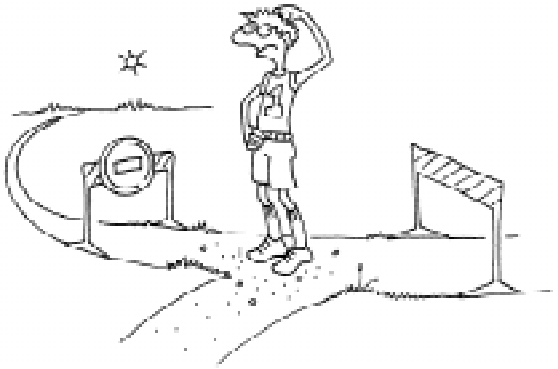
Система включает интерпретатор, написанный на ассемблере, трехуровневый оптимизирующий компилятор, выборочный профайлер, собирающий данные о частоте исполнения методов, инструментирующий профайлер, собирающий более подробную информацию о методах, выделенных для перекомпиляции, и контроллер, управляющий работой всей системы.

Интерпретатор (MMI – Mixed Mode Interpreter) позволяет интерпретируемому коду использовать один и тот же стек и один и тот же механизм обработки исключений с компилируемым кодом [1]. MMI вставляет в интерпретируемый код инструкции, записывающие количество вызовов метода, итераций циклов и выполнений ветвей условных переходов. Когда счетчик превышает некоторый порог, интерпретатор посылает запрос на компиляцию.

Данные об уже скомпилированных методах собирает выборочный профайлер, который периодически делает снимки состояния стека и обновляет информацию. Приостановка всех потоков во время получения снимка рассматривается авторами, как чересчур дорогая стратегия, поэтому потоки не останавливаются. Вместо этого, механизм управления потоками приостанавливает профайлер всякий раз, когда поток удаляется из очереди. Благодаря этому приему фактическая ошибка в данных профилирования оказывается по результатам экспериментов, пренебрежимо малой [2]. Интерпретируемые методы выборочный профайлер игнорирует, так как интерпретатор использует собственную встроенную систему сбора данных.

Выборочный профайлер поставляет контроллеру список «горячих» методов – кандидатов на перекомпиляцию. Контроллер составляет план профилирования для инст-

Рисунок 4. Архитектура системы динамической компиляции IBM DK [1].



Ссылка в таблице методов заменяется ссылкой на вновь скомпилированный код, а по адресу входа в старую версию метода подставляется заглушка...

рументирующего профайлера и сообщает его инструментирующему профайлеру. Инструментирующий профайлер встраивает в целевые методы инструментарий, позволяющий собрать более точные данные о поведении методов во время выполнения. Данные поступают к контроллеру, который составляет план компиляции и ставит метод в очередь на компиляцию более высокого уровня оптимизации.

Оптимизирующий компилятор выбирает методы из очереди и компилирует их на указанном уровне оптимизации, в соответствии с планом для данного метода. Ссылка в таблице методов заменяется ссылкой на вновь скомпилированный код, а по адресу входа в старую версию метода подставляется заглушка, которая заменяет код вызова в вызывающем методе. Это позволяет эффективно заменить старую версию метода на новую в тех случаях, когда вызовы не динамические, например, в результате предыдущей оптимизации [1].

ПРОМЕЖУТОЧНОЕ ПРЕДСТАВЛЕНИЕ

На настоящий момент компилятор использует три внутренних представления, которые последовательно получаются друг из друга: расширенный байткод, «четверки» (quadruples) – представление в виде инструкций с тремя операндами (результат и два входных операнда) и ациклический ориентированный граф. Последний используется только на третьем уровне оптимизации и для генерации кода преобразуется опять к «четверкам» [2]. На рисунке 5 показаны примеры расширенного байткода и «четверок», а также пример удаления избыточного копирования на «четверках».

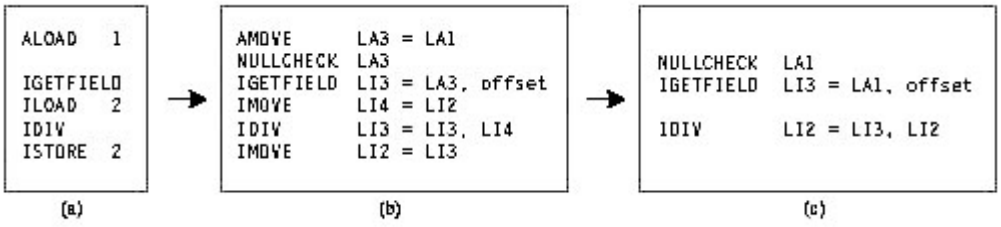


Рисунок 5. Пример преобразования расширенного байткода в «четверки»: (a) расширенный байткод, (b)»четверки», (c) после удаления избыточного копирования [2, Figure 2].

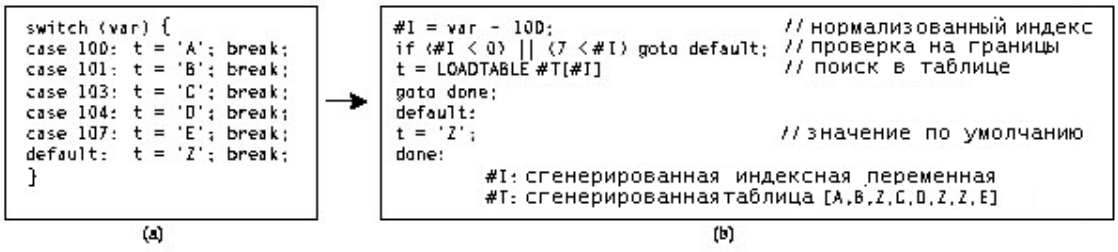


Рисунок 6. Пример преобразования оператора switch в простую операцию загрузки значения из таблицы, сгенерированной компилятором:
(a) оператор switch, (b) псевдокод – после преобразования, обращение к таблице [2, Figure 6].

ОПТИМИЗАЦИЯ И ДЕОПТИМИЗАЦИЯ

На первом уровне производятся простые оптимизации промежуточных представлений. На расширенном байткоде – это inline-подстановка «очень маленьких» методов, девиртуализация на основе анализа иерархии классов, оптимизация операторов switch. На «четверках» – удаление избыточного копирования и недостижимого кода в самых простых случаях, удаление избыточных операций, появившихся в результате транслирования расширенного байткода в «четверки» (см. рисунок 5). На рисунке 6 представлен пример преобразования оператора switch.

На втором уровне производятся inline-подстановки более крупных методов на основе статического эвристического анализа, выполняется несколько итераций анализа потока данных на «четверках» (пример анализа потока данных приведен на рисунке 7), анализ типов с удалением ненужных проверок, удаление избыточной синхронизации. Производятся inline-подстановки методов на основе исключений. Последняя техника состоит в следующем: если какое-то исключение особенно часто пробрасывается по определенной цепочке методов, то может быть принято решение об inline-подстановке всей цепочки и замене бросания и исключения условным переходом.

На последнем, третьем уровне добавляются более сложные оптимизации: escape-анализ (анализ, определяющий, будет ли объект доступен после завершения вызова метода) и выполняемые на его основе замена объектов скалярами и удаление избыточной синхронизации, оптимизации циклов на представлении в виде графа, перестановка инструкций для оптимального выполнения. Оптимизации циклов включают создание версий и упрощение (развертку циклов). Версии циклов применяются в случае, когда внутри цикла может быть брошено исключение (например, в результате проверки на границы массива), однако на практике это исключение бросается только в некоторых редких случаях. Тогда создаются две версии цикла – одна оптимизированная для общего случая, другая специальная – для тех ситуаций, в которых возможны исключения. Развертка цикла уменьшает число итераций, а если число итераций неве-

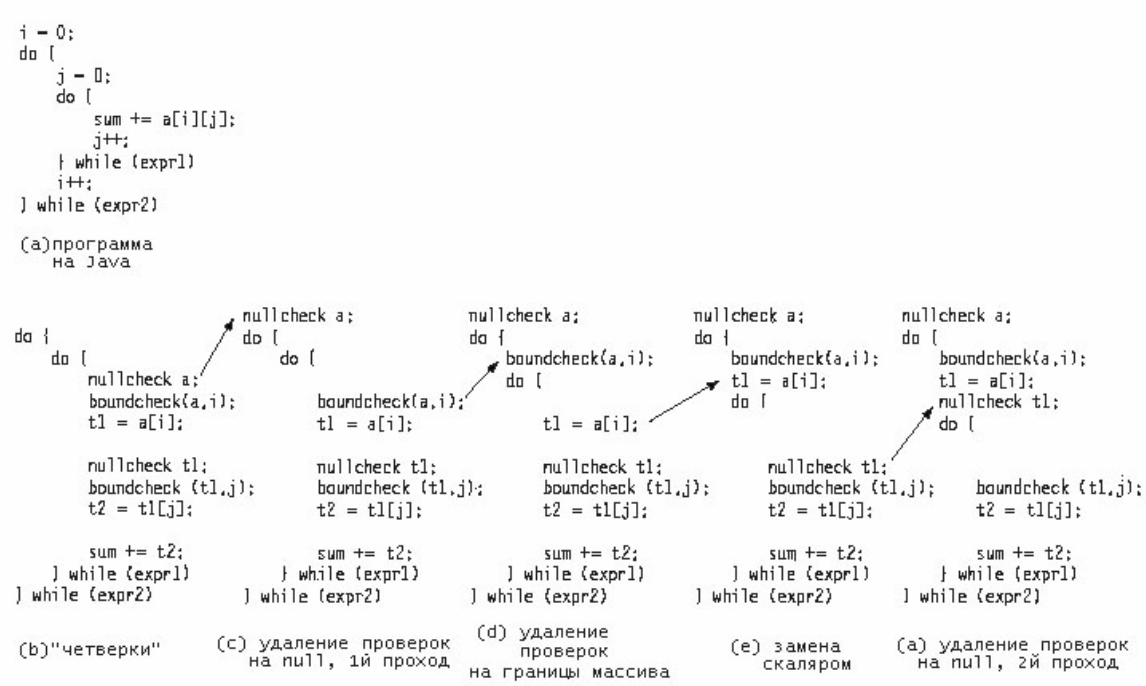


Рисунок 7. Пример успешной оптимизации путем нескольких итераций преобразований потока данных [2, Figure 8].

лико и точно известно, цикл вообще может быть заменен линейной последовательностью инструкций [2].

Машинный код генерируется из «четверок», и для различных целевых платформ применяются различные стратегии. В настоящий момент в компиляторе IBM DK используеюся три разных механизма распределения регистров: менеджер регистров для платформы IA-32, линейный алгоритм – для PPC и раскраска графа – для IA-64. Выбор механизма обусловлен особенностями целевой платформы: у процессора IA-64 много равноправных регистров общего назначения, что позволяет достичь значительной выгоды только с помощью распределения регистров, у процессора PPC регистров меньше, но все они равноправны, у процессора IA-32 небольшой набор регистров, часть из которых всегда или в некоторых конкретных случаях выполняет специфические функции, которые надо учитывать. Кроме того, применяются две общих стратегии распределения регистров: стратегия, нацеленная на увеличение числа инструкций, выполняемых параллельно, и стратегия, нацеленная на уменьшение числа одновременно используемых регистров [2].

Замещение на стеке не используется. Деоптимизация реализуется с помощью защищенных участков кода (просто выбирается другая ветка) и с помощью техники динамического исправления кода (code patching), которая уже рассматривалась выше. Часть кода метода заменяется в момент вызова, если какие-то из условий, обеспечивающих возможность оптимизации, оказываются неверными.

HOTSPOT JVM

АРХИТЕКТУРА

Виртуальная Java-машина HotSpot JVM от Sun Microsystems в версиях, начиная с 1.4, включает два компилятора: серверный и клиентский, использующие один и тот же интерфейс с виртуальной машиной [9]. При первом вызове все методы выполняются интерпретатором. Клиентский компилятор настроен на специфические нужды клиентского приложения, которое должно быстро запускаться и работает, как правило, относительно небольшое время. Поэтому клиентский компилятор использует только один уровень оптимизации и производит только ограниченный набор простых преобразований. Серверный компилятор предназначен для работы на серверном приложении, где постепенная оптимизация в течение длительного времени может быть выгодна, а время старта менее существенно. Серверный компилятор использует два уровня оптимизации: первый – быстрая оптимизация, второй – полная.

Интерпретируемые и компилируемые методы используют один и тот же стек. Интерпретатор для серверной и клиентской версии Java-машины используется один и тот же. Когда вызывается компилятор, то, в зависимости от контекста выполнения, происходит переключение на серверную или клиентскую версию.

Информацию о поведении программы собирает профайлер, который запускается периодически размещаемым на стеке специальным объектом – задачей профилирования. Задача профилирования вызывает профайлер, профайлер приостанавливает на небольшое время все потоки (если в данный момент это возможно), делает снимок состояния стека и обновляет информацию. Основная информация, которую собирает профайлер – это число вызовов различных методов, число выполняемых итераций циклов и динамические цепочки вызовов.[1]

Клиентский компилятор производит лишь ограниченное число самых простых и быстрых оптимизаций: удаление избыточного кода, избыточных проверок условий и проверок на null, а также набор машинно-зависимых оптимизаций.

[1] Детали реализации профайлера и компилятора HotSpot JVM получены путем исследования исходного кода HotSpot JVM версии 1.5, свободно доступного по академической лицензии на сайте www.java.sun.com

ПРОМЕЖУТОЧНОЕ ПРЕДСТАВЛЕНИЕ, ОПТИМИЗАЦИЯ И ДЕОПТИМИЗАЦИЯ

Серверный компилятор HotSpot JVM использует два уровня оптимизации: первый – быстрая компиляция, и второй – полная оптимизация. Промежуточное представление – SSA-граф [10], на котором строятся различные вспомогательные графы, нужные для оптимизации, например, def-use-цепочки. На первом уровне производится inline-подстановка только очень маленьких методов, отключены оптимизации циклов и оптимизации при генерации последовательности инструкций (scheduling).

Машинно-независимые оптимизации на обоих уровнях включают удаление недостижимого кода, распространение констант, удаление избыточных проверок типов и проверок на null. Все эти действия производятся глобально (на уровне метода) на SSA-представлении. На втором уровне оптимизации добавляется фиксированное число итераций оптимизации циклов.

Для генерации машинного кода из промежуточного представления используется BURS-парсер. Распределение регистров выполняется методом раскраски графа с некоторыми улучшениями, введенными для ускорения работы алгоритма. В число таких улучшений входят определение «связанных» значений, которые могут помещаться только в некоторых фиксированных регистрах (эти регистры вместе со связанными с ними значениями исключаются из общей раскраски), а также разделение последовательности инструкций на участки большого и небольшого дефицита регистров. Если регистров не хватает, копии вставляются в тех местах, где область жизни значения пересекает границу участка большого дефицита регистров [10]. Другие машинно-зависимые оптимизации включают локальные оптимизации (peephole optimizations) и перестановку базовых блоков.

Там, где может возникнуть потребность в деоптимизации, вставляются так называемые «ловушки для нестандартных ситуаций» (uncommon traps): проверяется условие, которое при оптимизации было принято как верное, и, если оно не выполняется, производится замещение на стеке и откат к состоянию интерпретации.

Замещение на стеке (OSR) очень активно используется в серверном варианте HotSpot JVM как для перехода от исполнения скомпилированного кода назад к интерпретации (в случае, когда срабатывает «ловушка»), так и для перехода от интерпретации к исполнению скомпилированного кода в ситуации долго выполняющегося цикла. В последнем случае компилируется специальный вариант метода с точкой входа в начале итерации цикла. Для корректной передачи состояния исполняемого метода от интерпретатора к компилятору или наоборот генерируются адаптеры.

STARJIT

Компилятор StarJIT, созданный в исследовательской лаборатории Intel[12], принимает на входе Java-байткод и CIL и генерирует машинный код для процессоров архитектуры IA-32 и Itanium. Компилятор использует общее промежуточное представление и общую схему оптимизации для обоих входных языков.

ПРОМЕЖУТОЧНОЕ ПРЕДСТАВЛЕНИЕ

Промежуточное представление StarJIT (STIR) – это граф потока управления, состоящий из базовых блоков, каждый из которых содержит последовательность инструкций. Инструкции STIR объединяют в себе особенности инструкций Java-байткода и CIL (в частности, в STIR тип есть и у переменных, как в CIL, и у операций, как в Java). Генерируются явные инструкции для

Там, где может возникнуть потребность в деоптимизации, вставляются так называемые «ловушки для нестандартных ситуаций»...

всех действий, которые должны производиться неявно, согласно стандартам языка, таких как преобразование типов или проверки на null. Кроме того, вместе с промежуточным представлением генерируются и сохраняются def-use-цепочки[1], которые используются во время оптимизации. Для проведения таких оптимизаций как глобальное распространение и свертка констант, строится SSA-представление.

ОПТИМИЗАЦИЯ И ДЕОПТИМИЗАЦИЯ

StarJIT использует два уровня оптимизации. Компиляция производится в два прохода. При первом проходе производится набор простых и быстрых оптимизаций: распространение и свертка констант, удаление избыточного кода, девиртуализация и inline-подстановки на основе данных статического анализа, а также удаление избыточных операций с памятью и замена объектов скалярами там, где для этого достаточно статических данных и не требуется сложного анализа. Затем компилятор проверяет, доступны ли данные профилирования по компилируемому методу. Если нет и это значит, что метод компилируется первый раз, оптимизированное промежуточное представление дополняется профилирующим инструментарием и передается кодогенератору, который производит ряд машинно-зависимых оптимизаций [12].

Если данные профилирования доступны (то есть метод был выбран для повторной компиляции), то после первого прохода промежуточное представление аннотируется профилем и делается еще один проход. Во время второго прохода выполняется, в целом, та же последовательность действий, что и во время первого прохода, но оптимизации более агрессивны, активно используются данные динамического профилирования (например, при девиртуализации), производятся более сложные оптимизации циклов.

Кодогенератор также использует данные динамического профилирования для генерации последовательности инструкций (scheduling) и оптимального распределения регистров (при повторной компиляции методов). Последовательность действий та же, что и при оптимизации промежуточного представления: проверка, доступны ли данные профилирования; если да, выполняются дополнительные оптимизации сгенерированной последовательности инструкций, и данные профилирования используются при принятии решений; если нет – производится статическая оценка, и на ее основе – быстрая генерация [12; 13].

В целом, стратегия оптимизации в StarJIT является наиболее «плоской», с точки зрения распределения оптимизаций по уровням среди всех рассмотренных примеров. Уровня интерпретации или базовой неоптимизирующей компиляции нет, и уже при первой компиляции применяется достаточно большое количество оптимизаций. Тем не менее, эта стратегия также дает положительный результат, и немалый вклад в него вносит активное использование машинно-зависимых оптимизаций [13].

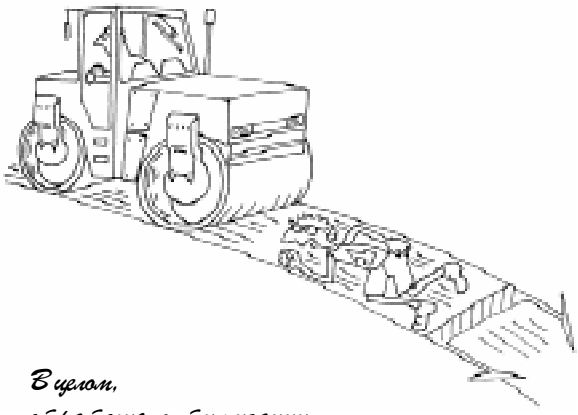
ЗАКЛЮЧЕНИЕ

Итак, мы рассмотрели несколько примеров успешной реализации подсистемы динамической компиляции для виртуальных машин. Некоторые из рассмотренных нами компиляторов входят в состав коммерческих продуктов, используемых разработчиками на протяжении уже нескольких лет. Существуют также и другие реализации динамических компиляторов в составе виртуальных машин других производителей, не вошедшие в данный обзор, использующие, тем не менее, те же общие принципы, что и рассмотренные здесь компиляторы.

Применение принципов многоуровневой компиляции, выборочной компиляции «го-

[1] Def-use-цепочки (цепочки «определение-использование») – это граф, узлами которого являются аргументы и результаты выражений, а дуги соединяют узлы-результаты с теми узлами в последующих выражениях, где эти результаты являются аргументами. Def-use-цепочки используются для удаления избыточного копирования, неиспользуемого кода и недостижимого кода в масштабе всей процедуры.

рячих» методов, различных техник инструментирующего и выборочного профилирования, агрессивные, основанные на данных профилирования оптимизации (для «горячих» методов) в сочетании с техниками деоптимизации, применяемыми «по требованию» (что было бы невозможно для статически компилируемого кода), позволяет получить выигрыш в скорости выполнения в 5–7 раз, по сравнению с интерпретируемым кодом или кодом, не выборочно компилируемым неоптимизирующим компилятором. Наибольшую выгоду динамическая компиляция приносит в серверных приложениях, работающих в течение длительного периода времени.



В целом, стратегия оптимизации в StarJIT является наиболее "плоской"...

Литература

[1] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 2001.

[2] T. Suganuma , T. Ogasawara , K. Kawachiya , M. Takeuchi , K. Ishizaki , A. Koseki , T. Inagaki, T. Yasue , M. Kawahito , T. Onodera , H. Komatsu , T. Nakatani. Evolution of a java just-in-time compiler for IA-32 platforms, IBM Journal of Research and Development, v.48 n.5/6, p.767-795, September/November 2004

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, J. Whaley. The Jalapeno Virtual Machine. IBM Systems Journal, vol. 39, No 1, 2000.

[4] David Grove and Michael Hind. The Design and Implementation of the Jikes RVM Optimizing Compiler. OOPSLA '02 Tutorial, Nov 5, 2002

[5] Stephen J.Fink and Feng Qjan. Design, Implementation, and Evaluation of Adaptive Recompilation with On-Stack Replacement, IBM T.J. Watson Research Center, March 2003.

[6] J. Whaley. A portable sampling-based profiler for Java virtual machines. In ACM 2000 Java Grande Conference, June 2000.

[7] K. Ishizaki, M. Kawahito, T. Yasue, and H. K. and T. Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, Oct. 2000.

[8] M. Arnold. Online Profiling and Feedback-Directed Optimization of Java. PhD thesis, Rutgers University, October 2002.

[9] The Java HotSpot Virtual Machine, v1.4.1, d2, A Technical White Paper. Sun Microsystems, September 2002.

[10] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In USENIX Java Virtual Machine Research and Technology Symposium, pages 1–12, 2001.

[11] Stephen Fink, David Grove, and Michael Fink. Dynamic Compilation and Adaptive Optimization in Virtual Machines. IBM T.J. Watson Research Center, 2004.

[12] Ali-Reza Adl-Tabatabai, Jay Bharadwaj, Dong-Yuan Chen, Anwar Ghuloum, Vijay Menon, Brian Murphy, Mauricio Serrano, Tatiana Shpeisman. The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments. Intel Technology Journal, vol. 07, Issue 01, February, 2003.

[13] Ali-Reza Adl-Tabatabai, Jay Bharadwaj, Dong-Yuan Chen, Vijay Menon, Brian R. Murphy, Tatiana Shpeisman. The StarJIT Dynamic Compiler – A Performance Study on the Itanium Architecture. 2nd Workshop on Managed Runtime Environments, 2004 (MRE'04).

[14] Todd Anderson, Marsha Eng, Neal Glew, Brian Lewis, Vijay Menon, and James Stichnoth. Experience Integrating a New Compiler and a New Garbage Collector into Rotor. Journal of Object Technology, Vol. 3, No. 9, 2004.

[15] Kapil Vaswani, Y.N. Srikant. Dynamic Recompilation and Profile-Guided Optimizations for a .NET JIT Compiler. IEE Software 2004.

[16] Kang Su Galtin. Power Your App with the Programming Model and Compiler Optimizations of Visual C++. MSDN Magazine, January 2005.

[17] Emmanuel Schanzer. Performance Considerations for Run-Time Technologies in the .NET Framework. MSDN Library, August 2001.

[18] Gregor Noriskin. Writing High-Performance Managed Applications: A Primer. MSDN Magazine, June 2003

[19] David Stutz, Ted Neward, Geoff Shilling. Shared Source CLI Essentials. O'Reilly, 2003.

Чилингарова Софья Александровна,
аспирант кафедры «Информатика»
математико-механического
факультета СПбГУ.