C2 The JIT In HotSpot Cliff Click

Слайд 2

1997 год. Sun делает Java, пытаясь сделать его быстрым и хорошим. Тогда было много AoT компиляторов, которые генерировали код медленно, но хорошо. Машины были медленнее, а потому хороший код был действительно нужен. Также было много интерпретируемых языков (Питон). Были языки, которые смешивали интерпретацию и кодогенерацию, но только с генерацией в шаблонном стиле, это было медленнее статического компилятора, но быстрее интерпретатора. Клифф в рамках докторской работы выяснил, как сделать хороший и быстрый компилятор, и с этим он пришел в Sun. Большой упор делается на быструю генерацию кода, а также и на его качество. Сильный упор на скорость, который связан с объемом памяти, означал маленький IR. Потому что IR хотелось бы помещать в l2-кэш. Если он не помещался, ты гораздо медленнее. Маленький не значит простой. И сейчас, подходы, которые применялись тогда, не сильно поменялись, основные принципы еще в ходу.

Слайд 3

Слайд 4

Слева вы видите обычный CFG граф с базовыми блоками, а справа - Sea of Nodes. Как видите, у нас уже есть SSA форма, потому что есть фи-функции. Желтым обозначен граф потока управления, а зеленым - граф потока данных.

Слайд 5

Вот у нас есть граф потока данных.

У нас есть узел Start - начало вычисления, Region - для слияния потоков управления, If - разделяет потоки на 2, Call - сериализованный вызов, на вход и выход один поток.

Заметим, что здесь у нас нет базовых блоков. Их здесь нет, потому что большинство оптимизаций не заботится о границах этих блоков, а потому и нет смысла их держать.

Заметим, что в конце также находится узел Start, мы можем ходить по графу в любом направлении.

Слайд 6

Далее у нас есть граф с данными. Все уже в SSA форме

Тут у нас есть константы, арифметика, фи-узлы, но у нас нет имен переменных. Имена переменных появляются при построении SSA формы из байткода. В начале по байткоду строится SSA форма за один проход, и там появляется мапа имен переменных в соответствующие узлы.

Слайд 7
Узлы принимают и отдают на выход и поток управления, и поток данных.
Call - принимает управление, а выдает поток управления и поток данных через узел проекции (об этом позже). Поток данных здесь - возвращаемое значение от функции, а это значит у нас появляется узел в графе.
If - принимает управление и данные, и выдает 2 управление, на правду и ложь.
Return - принимает управление и данные (возвращаемое значение функции), выдает 1 управление.
Call тоже может принимать значения данных для аргументов
Слайд 8
Отметим также, что значение в графе имеет только порядок ребер! Семантика формируется через ребра, сами узлы как-бы плавают, мы могли бы расположить их по-другому, нам важно только, чтобы связь ребра все связывали также аккуратно.
Слайд 9
Полученный IR используется в нескольких важных проходах, связанных оптимизациями
Пройдемся по основным проходам, которые у нас делаются.
В начале строится Sea of Nodes по байткодам. Там уже работают инлайнинг и оконные оптимизации (трудности перевода).
Далее работают Iter проходы - повторение оконных оптимизаций, пока можешь. Эти оптимизации включаются в Dead Code Elimination, Constant Propogation, Common Sub Expression, оптимизации загрузки и хранения, и многие другие. И эти все оптимизации - это почти самое основное, что отвечает за увеличение производительности. Iter гарантированно линеен по размеру юнита компиляции, iter вызывается после любой оптимизации другого рода, таких как числе после оптимизаций циклов, и он всегда линеен.
Также работают Global Constant Propagation, оптимизации цикла (loop unroll, range extermination)
Далее работает Global Code Motion, что приводит к тому, что граф становится “Хрупким”: появились базовые блоки, и теперь нельзя менять граф, не думая о границах блока. Поэтому все оптимизации делаются ПЕРЕД этим проходом.
После этого уже происходит кодогенерация, аллокация регистров, установка в JVM, запуск.
Слайд 10

Все это написано на C++. У нас есть v-table, которая представляет собой семантику opcode. Узлы обычно это классы с виртуальными методами. У нас есть примерно 35 виртуальных методов для узла: имена, дозволенные оконные оптимизации, c-prop, дебаг принты и так далее. мы пытаемся сделать узлы как можно меньше в плане количества данных. Запрашивать информацию через виртуальные вызовы довольно дорого, а потому часто используют массивы с флагами внутри. Также есть массивы для входных и выходных ребер.
То есть, узел - это довольно маленькая структура по количеству занимаемой памяти, что важно для скорости.
Use → Def edges - это C указатели от Use узла в Def узел. Важен порядок! Также разрешен NULL.
Также были позже добавлены Def→Use ребра, которые упорядочены. Это просто списки, которые дают информацию об использованиях узла, с которым можно работать: итерироваться, обрабатывать и так далее.

Слайд 11

Поговорим об оконных оптимизациях. У нас есть множество правил переписывания графа (Graph Rewrite Rules). Они представляют из себя оптимизации в локальной области графа. Внутри области мы заменяем кусок графа на что-то получше. И нет никаких изменений вне области. Соседи рядом с областью (те, кто соединены с узлами внутри области) не знают об изменении в области.
Большинство областей “укоренены” в одном узле. То есть, этот узел и порождает собой область. У такого узла есть виртуальный метод Ideal, который делает инспекцию области, порожденной этим узлом и делает изменения, если считает, что оно стоит того, а потом говорит о том, что сделал изменение.
Таких правил очень много. От самых простых до сложных.

Слайд 12

Теперь подробнее про Iter. Как уже говорилось ранее, это один из самых важных проходов при оптимизации. У нас есть рабочий список (worklist) узлов. Мы берем узел из него, оптимизируем, если можем. Также проверяем, constant propagation Global Value Numbering, Dead Code Elimination и другие вещи, которые общие для всех узлов. Если что-то поменялось, то мы пушим соседей в рабочий список. Так повторяем, пока рабочий список не будет пуст.
Оказывается, что вне зависимости от порядка рассмотрения узлов в списке мы в итоге приходим к одному и тому же графическому представлению. Это позволяет избавиться от мусора в коде, сделать его более “чистым”. Оно вызывается почти везде в компиляторе, когда ты

сделал что-то помимо Iter при работе с графом. Также это быстро: все вызовы Iter занимают линейное время от размера программы.

Слайд 13

Теперь подробнее о ребрах. Это простые указатели, что нужно для скорости, а значит у нас не может быть меток на ребрах. Если у нас есть Use→Def ребра, то им и не нужны метки, ведь они упорядочены, и этот порядок несет всю необходимую семантику. Однако есть нюансы, связанные с выводом, аутпутом у узлов. Например, узел Call может выдавать результат что может приводить к изменениям в памяти, исключениям. Так что, если у нас есть указатель на узел Call, нам не ясно, какой из выводов мы намереваемся использовать. То есть, нам хочется все же дать метку ребру.

Мы делаем следующее: мы говорим, что у нас есть кортеж потенциальных результатов вызова метода. И мы делаем проекцию: получить нужный кусок из кортежа. Образуется еще узел, внутри которого хранится номер - какой элемент вырезать из кортежа. То есть, мы пометили ребро там, где нам было нужно. Получается, что в 95% случаев, когда все работает быстро и 5% случаев, когда нам придется смотреть в проекцию, чтобы понять, а какой нужен аутпут.

Слайд 14

Клифф приводит аналогию: граф - это абзац текста, предложения, а типы - это слова и символы. Это то, что нужно, чтобы мы вообще могли хоть что-то делать в плане рассуждений о программе и оптимизаций. Математически, тип - это множество значений. Все целые, флоаты, одно число пи и так далее.

Все узлы в графе имеют тип. Даже узел “Region”, который занимаются слиянием потока управления, имеет тип “Control”.

Типы используются при распространении констант, анализе иерархии классов (CHA), приведения типов и так далее.

Слайд 15

Какие же бывают типы? Все интовые: байт, шорт, инт, лонг, также типы для интов с малым диапазоном (0-10). Флоты, 32, 64, float константы. Кортежи, массивы. Объекты (instances), содержит в себе информацию о полях и их значениях. Есть указатели: сырые (raw) - которые небезопасны, есть ООП указатели, это instance, array, klass (Java Class объект). Klass могут быть точными или неточными. Как на примере в иерархии. Класс HashMap имеет родителей и детей. Мы можем сказать: это HashMap и что-то внизу в иерархии, или это просто HashMap и все?

Слайд 16

Типы определяют такую алгебраическую структуру, как решетка. Более того, они определяют булеву алгебру. В решетке есть такие понятия, как Meet (Supremum) и Join (Inf). У нас есть пример решетки - целые константы.
У нас есть понятия “Bot” - мы не знаю значения, но это какое-то целое число, может быть любым из них, я должен реализовать правильно семантику, которую имел в виду программист, чтобы получить все же верное значение. Есть понятие “Top” - противоположность “Bot” - это все константы, мы можем выбрать в данный момент любую из этих констант. Это идея оптимистичных оптимизаций. Сначала мы начинаем с Top, и если все плохо, падаем в Bot. И вся эта применима не только к числам, но и к классам, объектам, и любым другим типам. Top и Bottom - это как 1 и 0 в обычной
Если говорить о свойствах булевой алгебры, то имеем следующие: Complete - у каждого подмножества можно взять sup и inf. Complement - есть отрицание, и оно работает как отрицание - обратная сторона для всего. Дистрибутивность - можем делать meet и join друг с другом в любом порядке. Более того, порядок не важен, мы получим один и тот же ответ, и он оптимален. Как было в примере с worklist: я могу делать pull узлов и мне не важно, в каком порядке я буду оптимизировать узлы в worklist-е. Я получу один и тот-же оптимальный ответ.

Слайд 17

Типы могут быть большими, с большим описанием. также они иммутабельные. Используется такая технология, как hash consing (интернирование). Суть в том, чтобы шарить значения, которые структурно похожи. Тип может быть сложным, но когда мы сделаем инстанс, мы хотим, чтобы он был один. После интернирования лучше сравнивать через “==”, потому что equals() медленнее (мы же сравниваем все поля сложного типа), а если у нас есть цикл в типах, то это бесконечная рекурсия, креш.
Все же, большинство типов попадают в таблицу hash-cons. Это позволяет переиспользовать память и часто попадать в L1 кэш.

Слайд 18

Пессимистичность против оптимистичности в оптимизации. Итер и оконные оптимизации все пессимистичны. Это значит, что программа корректна до и после оптимизации. То есть, оконные оптимизации в любом случае сохраняют корректность программы в своей локальной области. Типы с начинают с какого-то значения, а потом подинмаются вверх по решетке от Bot, могут дойти до констант до Top. Но в любой итерации мы получаем корректную программу.

Global Constant Propagation - оптимистичная. Мы инициализируем все типы в Top. Программа теперь не корректна, ведь тип теперь - неоднозначное состояние. Типы постепенно падают по решетке, пока не разрешатся конфликты, не спадет оптимизация, и пока программа не станет коректной.
В целом, GCP лучше чем Iter, он находит каждую константу, что находит Iter, но и больше, особенно с объектами и циклами. Он находит разные null, находит больше типов, которые Iter не находит.

Слайд 19

Iter и GVN используют вызов Value(), который предназначен для математики типов. В виртуальный вызов. Он смотрит на Use→Def ребра, получает типы от input типов, и вычисляет тип у данного узла основываясь на input типах. Например, делаем 2 + 3, смотрим на тип 2, на тип 3, и понимаем, что у нас целое. Если у нас 2 и Bot, например, то у нас Bot. Этот вызов используется и в Iter, и в GCP, в одном типы строго поднимаются, в другом падают.
Value() вызов должен быть монотонным, что значит, что аутпут падает токда и только тогда, когда инпут падает. То есть, если все интпут типы isa новые инпут типы, то старые аутпут типы isa новый аутпут тип.

Слайд 20

Некоторое исполнение способно триггерить компиляцию, например когда оно достигает 10к раз. Далее мы идем по стеку вызовов и смотрим, какой метод компилировать. Мы так делаем, потому что первыми 10к достигают геттеры. Нам не хочется компилировать просто геттер, хочется скомпилировать контекст, в котором этот геттер вызывался.
Далее парсим байткод, строим SSA. Идем по стеку, выбираем, в какой момент надо инлайнить, пока билдим IR. Также используем CHA чтобы уточнить типы (и решать, нужно инлайнить или нет), который полагается на оконные оптимизация (которые всегда корректны и упрощают код). То есть пока мы еще парсим метод, мы уже можем применять локальные оптимизации. После этого получаем обычно маленький и горячий метод, который можно инлайнить. Далее эвристика: мы всегда инлайним триггеры, встроенные функции, простые геттеры и сеттеры, небезопасные функции. При этом исключаем вещи, которые уже скомпилированы или слишком большие.

Слайд 21

Выбор инлайнить или нет делается в моменте парсинга байткода, что бывает довольно рано. Мы обычно имеем ограниченные знания об оптимизаторе, есть peepholes, но нет GCP, например. Не везде доступны результаты анализа частот использования методов. Мы не достигаем той

точности, которой хотим. Например, пользуемся эвристикой “постоянные аргументы”. Она значит, что мы можем встроить этот метод, потому что у него (почти) всегда одни и те же аргументы. Однако может быть промах при встраивании, что может быть скомпенсировано over-inlining.

Слайд 22

Поговорим теперь о Global Code Motion. Мы строим настоящий CFG, распутывает Sea of Nodes, помещаем узлы в блоки. Это глобальный планировщик, смотрящий на задержку и частоту. Приводит к перемещению кода из цикла в низкочастотные ветви, обычно пути деоптимизации. Требует анти-зависимости, которые используют точной информации о зависимостях. Теперь IR становится хрупким, имеет место местоположение кода, узлов. Поэтому лучше не делать оптимизаций после GCM.

Слайд 23

Это, вероятно, самая медленная и сложная фаза компиляции, но при этом ответственен за наибольшее ускорение во всех кодах. Основная причина - устойчив к овер-инлайнингу. Затраты на промах кода почти всегда меньше чем при отсутствии инлайнинга (мы платим за пролог и эпилог вызова). Единственная проблема овер-инлайнинга - потенциальный выброс кеша, что может привести к падению производительности. Но данный аллокатор к такому устойчив. Что было не так для многих статических компиляторов

Слайд 24

Также мы можем проводить оптимизации, специфичные для Java. У нас есть много проверок байткода. Есть проверки на null. Нет никакой разницы между пользовательским нулчеком и байткодовым нулчеком, реализация остается той же.

Это все позволяет избавляться от 90% нулчеков в моменте парсинга байткода через пипхолзы. Более того, такие оптимизации лучше делать при парсинге, а не при анализе IR, потому что IR получается гораздо больше, и там уже потребуется больше времени на оптимизацию. Если не удалось избавиться она нулчека, есть 2 стратегии. Первая (5%) - как проверка через операции с памятью. Загружаем память, смотрим на нул, если нул, делаем segv, а потом в райнтайме оно ловится, бросается NPE, который конвертируется в Java NPE. А вторая (5%) - через отдельную ветку, jump not equal, стандартно.

Слайд 25

Также мы можем делать анзиппинг нулчеков. Тут такой случай: проверяем на нул, далее идет разветление на да и нет, потом слияние управления, а потом опять проверка этого на нул. И так снова и снова. В

середине мы сделали анзип, и мы получили красивые 2 ветки, которые мы можем также оптимизировать. Может быть такое, что проверка на нул почти никогда не проваливается, то мы можем убрать проверку и в худшем случае деоптимизироваться.

Слайд 26

Оказывается, CHA превращает 90 % вызовов в статические на момент парсинга, что позволяет их инлайнить. Оставшиеся 10 % используют инлайн-кэш. 90% (из 10%) всегда попадание. Инлайн кэш - это кеш, который встроен в код. Это структура с ключами и значениями. Ключи - это классы, значение - статический метод, его нужная версия. В итоге при попадании нам надо только сравнить класс и сделать нужный вызов. 2 такта. Если нет попададния (10 % из 10 %) - делаем обычный виртуальный вызов, примерно 30 тактов на многих процессорах.

Слайд 27

Range Check Elimination. Тоже специфично для Java.

Сначала мы обнаруживаем ограниченные циклы и переменные индукции. Мы можем это сделать это в IR, там понятный паттерн внутри графа.

Далее делаем очистку (peel). Делаем первую итерацию до основных, потому что проверки, которые инвариантны для всего цикла, будут выполняться на первой итерации, и так как они уже были запущены в первой итерации, то во второй и так далее они уже будут true, а значит от них можно избавляться. Так можно избавиться от некоторых нулчеков.

Теперь основное тело не имеет никаких нулчеков, но есть проверки на range. Поэтому мы вставляем тело наверху и внизу основного, которые называются пред-цикл (pre-loop) и пост-цикл (post-loop). Первый рассматривает случай на нижней границе, когда вышли за нее, а второй - на верхней границе, где обработка случая на переполнение. Для них мы не проводим оптимизаций.

Далее можно избавляться от проверок внутри основного цикла. Далее делаем Iter на основном цикле. Если после этого тело стало совсем маленьким, делаем развертку цикла по степени двойки. Потом по развернувшемуся циклу опять делаем Iter, и так далее. Также надо разрешать проблемы, когда количество итераций нечетное, и тогда оставшуюся итерацию кладем в post-loop.

В итоге мы имеем хороший буст. Удвоение цикла не будет бесконечным, потому что могут быть проблемы с I-кэшем.

Слайд 28

Checkcast,instanceof, arraystore. Проверки на тип очень быстрые. Если a instanceof B, мы смотрим на иерархию классов для B, смотрим на таблицу для a, и если есть совпадение, то instanceof, а иначе нет.

Слайд 29

Многие проверки (safety-checks) в коде довольно редко падают (нулчеки, ренджчеки и так далее). Поэтому мы героически предполагаем, что раз она еще не упала, она никогда не упадет, и мы работаем вы этом предположении. Но все равно готовимся к тому, что придется упасть в торт и делать деоптимизацию.

Слайд 30

Обычно такие оптимизации делают ускоряют код, но проверка может провалиться. Тогда делаем деоптимизацию. Мы меняем фрейм на интерпретаторский. Также поднимает флаг того, что у нас случился фейл, делать профиляцию заново. Далее опять делаем компиляцию, но с учетом флага, не делаем героических оптимизаций. Трудная часть - правильно отслеживать флаги, потому что есть шанс попасть в бесконечный цикл деоптимизации.

Слайд 31

Деоптимизация! Это можно сказать главный бэкапный план в случае всех неудач. Вернемся к интерпретатору. У нас принцип - оптимизируй горячие методы, и что можно откомпилировать, если провал - возвращайся к интерпретатору.

Мы отслеживаем состояние JVM, и у нас появляются сейфпоинты, если происходит исключение, мы идем на сейфпоинт с состоянием.

Слайд 32

Со стороны компилятора, сейпоинт - это просто инструкция Call, которая считывает все состояние JVM вместе с эффектами памяти, и к нему обращение происходит довольно редко. Это просто узел в нашем IR. У нас есть Use→Def для получения состояния. Далее оптимизатор делает свою работу. Здесь задействована ветвь с очень низкой частотой выполнения! То есть, вещи, которые нужные для интерпретатора, но не для компилятора, отправляются на стек и перемещаются ветви с низкой частотой выполнения. И мы не будем задействовать эту информацию, пока не будем делать деоптимизацию.

Слайд 33

Дебаггинг C2. Тут проблемы те же, что и с Heroic opts. У нас есть обычный быстрый случай, и редкий, который сложно найти и отдебажить. Как его найти. Можно включать огромное количество флагов, однако это может привести к тому, что редкий случай станет обычным, и это займет много времени. Ты можешь обнаружить свою ошибку, есть бинарный

поиск при помощи флагов CIStart, CiStop, и много других опций для поиска ошибок.