Методы динамической и предварительной оптимизации программ на языке JavaScript

Роман Жуйков <zhroma@ispras.ru>, Дмитрий Мельник <dm@ispras.ru>,Рубен Бучацкий <ruben@ispras.ru>, Вааган Варданян <vaag@ispras.ru>, Владислав Иванишин <vladislav.ivanishin@gmail.com>, Евгений Шарыгин <eugene.sharygin@gmail.com>

Аннотация. Работа посвящена улучшению производительности программ на языке JavaScript. В работе рассматриваются особенности динамических оптимизаций в JIT-компиляторе для языка JavaScript, а также основные способы улучшения производительности для таких оптимизаций. Кроме того, предлагается способ организации предварительной компиляции программ на языке JavaScript с их сохранением в виде байткода, что позволяет сократить время запуска приложений за счет выполнения оптимизаций на этапе предварительной компиляции. Предложенные методы были реализованы в библиотеке с открытым исходным кодом для отображения веб-страниц WebKit. В результате удалось добиться значительного увеличения производительности выбранных тестовых JavaScript-приложений на платформе ARM.

Ключевые слова: оптимизация программ; компиляция во время выполнения; предварительная компиляция; граф потока данных; архитектура ARM.

1. Введение

В настоящее время широкое распространение получили программы на нетипизированных скриптовых языках, одним из повсеместно используемых языков является JavaScript. С ростом производительности персональных компьютеров и встраиваемых систем использование языка JavaScript стало возможно не только для выполнения небольших скриптов на веб-страницах, но и целых веб-приложений. Более того, уже имеются разработки операционных систем для телефонов, планшетов и ноутбуков, которые подразумевают использование JavaScript как одного из главных языков для создания приложений. Примерами таких систем могут быть Tizen[1] и FirefoxOS. Некоторая часть пользовательских приложений на этих системах будет представлять собой набор хранящихся на устройстве веб-страниц со скриптами на языке JavaScript. В связи с этим все больше возрастают требования к производительности программ-скриптов.

Многие современные интерпретаторы поддерживают разнообразные режимы компиляции горячих участков кода во время выполнения скриптов. Для обеспечения быстрого выполнения скриптов необходимо создание максимально качественного машинного кода, а также снижение затрат на все этапы оптимизации, выполняемые при компиляции во время выполнения. В данной работе будет рассматриваться интерпретатор JavaScript, называемый JavaScriptCore (JSC)[2] и входящий в состав браузерного движка WebKit[3] для отображения веб-страниц.

Цель данной работы – адаптировать имеющиеся в JavaScriptCore оптимизации для работы на встраиваемых системах. Также необходимо учесть специфику использования JavaScript-программ в составе локально хранящихся приложений. Планируется добиться генерации более качественного кода при компиляции во время выполнения, а также реализовать систему хранения и предварительной компиляции скриптов для приложений.

Дальнейшее изложение построено следующим образом. Сначала будет описана имеющаяся схема работы JavaScriptCore и перечислены предлагаемые методы оптимизации, а потом каждый из них будет рассмотрен более подробно.

2. Устройство JavaScriptCore

Первыми этапами работы JavaScriptCore являются лексический и синтаксический анализ. Исходный код разбивается на токены, методом рекурсивного спуска строится синтаксическое дерево, из которого в свою очередь строится внутреннее представление, называемое байткод (bytecode). В байткоде инструкции хранятся в виде массива ячеек, разные инструкции могут занимать разное количество ячеек. В первой ячейке хранится тип инструкции, в следующих ячейках хранятся адреса операндов и результата. Адреса операндов могут представлять собой ссылки на константы или номера локальных псевдорегистров. При чтении или записи полей объектов, загрузка адреса поля по имени выглядит как отдельная инструкция, один из операндов которой – константная строка, содержащая имя поля. Для многих инструкций последняя ячейка в байткоде выделена для хранения информации о профиле.

В ранних версиях JavaScriptCore байткод сразу передавался на выполнение интерпретатору. Интерпретатор последовательно читал инструкции байткода и выполнял необходимые действия, переходы и циклы организовывались за счет условных и безусловных операций перехода. Переход указывает, что вместо чтения следующей инструкции в байткоде интерпретатор должен перейти в другое место. В современных версиях JavaScriptCore вместо интерпретатора используется низкоуровневый интерпретатор (LLInt). Он фактически выполняет те же самые действия, однако запрограммирован на специальном мультиплатформенном ассемблере (offlineasm). Этот специальный ассемблер может быть скомпилирован на этапе сборки JavaScriptCore в машинный код для x86, ARM или нескольких других

платформ, а также может быть преобразован в исходный код на языке C. LLInt, как и обычный интерпретатор, позволяет начать выполнение байткода, не выполняя никаких подготовительных этапов, тем самым обеспечивает быстрое начало выполнения. Все другие уровни оптимизации требуют предварительных затрат по созданию машинного кода, соответствующего заданному участку байткода. LLInt поддерживает на уровне вызова функций взаимодействие со всеми уровнями оптимизации. Если какая-то функция уже была скомпилирована в машинный код, то вызов этой функции из низкоуровневого интерпретатора будет выглядеть так же, как и переход на точку входа в общий пролог интерпретатора для любой другой неоптимизированной функции. LLInt использует кэширование на уровне байткода для ускорения доступа к полям объектов по имени.

При работе низкоуровневого интерпретатора также происходит сбор информации о профиле – сохраняются типы и последние значения полей объектов. Необходимость оптимизации функций определяется с помощью оценки того, сколько раз в ней выполняются те или иные участки кода. Для перехода на первый уровень оптимизации времени выполнения (JIT-оптимизация) необходимо, чтобы функция набрала не менее 100 “очков выполнения”, при этом за каждую пройденную итерацию цикла прибавляется одно “очко”, а за вызов функции - 15 “очков”. Отметим, что эти числа являются примерными, в реальности дополнительно применяется эвристика, результат работы которой зависит от размера рассматриваемой функции. Таким образом, небольшой функции без циклов достаточно быть вызванной около 7 раз, чтобы для нее была выполнена базовая компиляция времени выполнения (Baseline JIT).

Baseline JIT создает для каждой операции байткода соответствующий машинный код. В этом коде реализуются все возможные случаи для данной операции. Например, операция сложения для чисел будет выполнена как сложение, а для операндов-строк – как конкатенация. Генерируемый код будет содержать множество ветвлений для разбора всех таких случаев. После того> как для функции будет создан машинный код, нет необходимости дожидаться окончания функции для запуска выполнения нового кода. Например, если функция выполняет цикл с большим числом итераций, то может быть выполнен немедленный переход на новый код (on-stack-replacement, OSR). Низкоуровневый интерпретатор закончит обработку очередной инструкции байткода и сразу перейдет в машинном коде в то место, которое соответствует началу следующей инструкции. Конечно, во всех местах вызова этой функции будет произведено перенаправление на новую версию функции – в машинном коде.

Baseline JIT код используется, как базовая версия кода для функций, которые скомпилированы с помощью оптимизирующего JIT-компилятора. Если оптимизированный код сталкивается со случаем, который в нем не поддерживается (например, тип или значение переменной не соответствует

собранному профилю), то происходит обратная замена на стеке (on stack replacement exit, OSR exit) к коду Baseline JIT. На уровне Baseline JIT так же, как и на LLInt, сохраняется профиль – информация о типах полей объектов и аргументов функций, и выполняется кэширование для ускорения доступа к полям объектов.

Информация о профиле, собранная на уровнях Baseline JIT и LLInt, используется для организации спекулятивного выполнения на следующем уровне оптимизации – оптимизации с использованием графа потока данных (Data flow graph, DFG JIT, Speculative JIT). Собранная информация содержит последние значения загруженных аргументов, полей объектов, а также результатов выполнения функций. Кэширование доступа к полям объектов на уровнях LLInt и Baseline JIT устроено так, что позволяет DFG быстро получать необходимую информацию. Например, по информации кэширования легко можно узнать, что некоторое обращение к полю объекта иногда, часто или всегда возвращает значение некоторого конкретного типа.

DFG JIT компиляция выполняется для функций, которые набрали не менее 1000 “очков выполнения”. На уровне DFG выполняются разнообразные оптимизации, опирающиеся на информацию о профиле. Из байткода с учетом профиля создается граф потока данных, в котором инструкции описаны в виде SSA-представления. На этом DFG графе выполняются оптимизации, и в конце итоговый набор инструкций преобразуется в машинный код.

DFG JIT распространяет полученную информацию о типах переменных по всему графу и вставляет в код необходимые проверки типов. Иногда DFG даже выполняет спекулятивную оптимизацию по самому значению переменной. Например, если по результатам профилирования поле объекта является конкретной функцией, ее код может быть встроен в вызывающую функцию, с добавлением необходимой проверки. Как было описано выше, когда одна из проверок не выполняется, происходит деоптимизация, то есть обратная замена на стеке (on stack replacement exit, OSR Exit) на код Baseline JIT.

Таким образом, DFG JIT код и Baseline JIT код могут сменять друг друга посредством замены на стеке (OSR). Когда код функции становится “горячим”, происходит переход на DFG JIT. Когда выполняется деоптимизация, происходит обратный переход. В случае многократного OSR exit сохраненная информация о том, почему произошла деоптимизация, также становится своеобразным профилем, который позволяет организовать реоптимизацию DFG, то есть создание нового DFG графа и машинного кода с учетом новой информации о профиле. Эвристика, оценивающая необходимость реоптимизации, использует экспоненциальную задержку в зависимости от количества уже выполненных реоптимизаций. Это позволяет исключить возникновение больших временных затрат на постоянную реоптимизацию кода и выполнение множества OSR переходов.

Четвертый уровень оптимизации - LLVM JIT, вызывается для функций, набравших не менее 10000 “очков выполнения”. В нем выполняется более широкий набор оптимизаций, а в качестве внутреннего представления помимо DFG графа используется биткод компилятора LLVM. Перед генерацией машинного кода выполняются оптимизации, уже реализованные в LLVM. Данный уровень JIT-оптимизации находится в состоянии разработки и пока не включается по умолчанию.

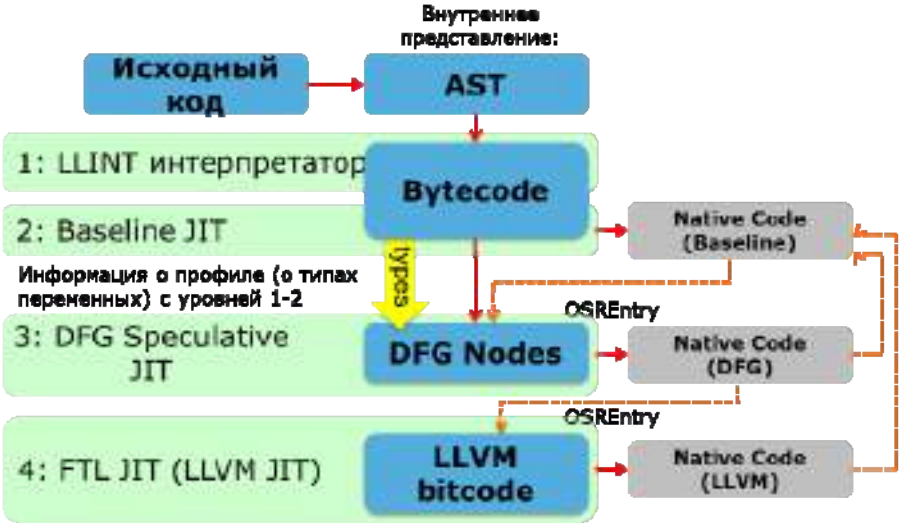

Рис. 1. Устройство JavaScriptCore

Итак, при выполнении скрипта в любой момент времени функции eval-блоки и глобальный код в JSC могут выполняться на любой комбинации LLInt, Baseline JIT и DFG JIT кода. В особом случае при выполнении рекурсивных функций код одной и той же функции может существовать на стеке вызовов в разных вариантах: в одном уровне функция выполняется на LLInt, в другом на Baseline JIT, в третьем на DFG. Возможен еще более сложный случай – допускается выполнение старого варианта DFG кода на одном уровне стека, в то время как на более вложенном уровне рекурсии произошло много деоптимизаций, и была выполнена реоптимизация, после которой был запущен новый вариант DFG кода.

Все уровни выполнения обеспечивают одинаковую семантику выполнения, и единственный эффект переключения между ними – производительность работы JavaScriptCore.

3. Оптимизация производительности

Для обоснования необходимости переключения на более быстрые уровни выполнения приведем два примера: первый – сравнение времени выполнения набора тестов PL benchmark от автора Martin Richard. Этот набор тестов запрограммирован на нескольких языках, поэтому есть возможность взять время выполнения программы на языке C как ориентир.

Таблица 1. Сравнение производительности уровней JSC.

Способ выполнения	Время выполнения, мс
Реализация на языке C	1.2
JavaScript интерпретатор	129
LLInt интерпретатор	58
Baseline JIT	8.4
DFG JIT	2.1

Другой набор тестов – Browsermark. На этом наборе Baseline JIT оказывается в среднем в 2.5 раза быстрее, чем LLInt, причем на некоторых тестах наблюдается различие производительности в 5 раз. Однако оптимизирующий DFG JIT еще в 1.7 раза в среднем быстрее Baseline JIT и позволяет ускорить некоторые тесты до 6 раз.

Как видно из результатов сравнения производительности, в JSC важно добиться, чтобы максимальное количество горячих участков кода выполнялось на уровне DFG JIT. На этом уровне не поддерживается часть операций байткода, поэтому для увеличения эффективности можно рассмотреть возможность реализации поддержки новых операций. Другим направлением может быть изучение причин деоптимизации, то есть обратных переходов на Baseline JIT, и исследование возможности их устранения. Это позволит избежать выполнения более медленной версии кода и последующей перекомпиляции.

Также необходимо улучшать качество генерируемого DFG JIT машинного кода. Здесь важным направлением для исследований является замена в машинном коде реализации некоторых сложных операций, выполняемых с помощью вызова функции. Можно иногда вместо вызова JavaScript-API функции, который требует достаточно больших затрат на подготовку аргументов и стека, выполнять так называемый intrinsic, то есть машинный код, выполняющий необходимые действия. Этот код может также вызывать соответствующую C-функцию, однако временные затраты на такой вызов будут значительно меньше.

Еще одним направлением для рассмотрения является идея предварительной оптимизации. В случае, если программа на языке JavaScript хранится локально, то перед ее выполнением может быть сделана оптимизационная предварительная подготовка, позволяющая ускорить процесс выполнения.

4. Динамические оптимизации

Опишем алгоритм работы с предсказаниями типов на уровне DFG JIT. Определение типов [4] достигается с помощью профилирования и последующего предположения о результатах операций, базирующегося на информации о профиле.

В код вставляются необходимые проверки типов, а далее производится попытка продвижения информации о типах результатов операций. Рассмотрим следующий пример выражения на языке JavaScript: *p.x * p.x + p.y * p.y*

Допустим, что в контексте нашего кода, объект *p* описывает точку на плоскости в декартовой системе координат, соответственно у него два поля *x* и *y*, которые хранят значение типа *double*. Эти значения иногда могут быть целыми, и, несмотря на то, что стандарт языка JavaScript не подразумевает хранения целых чисел, в целях эффективности выполнения следует по возможности хранить целые числа в виде *int32*, а не в виде *double*. Чтобы оценить проблему определения типов и способы ее решения в JavaScriptCore, необходимо в первую очередь оценить, какой объем работы должен выполнить интерпретатор языка JavaScript для вычисления выражения, приведенного выше, если он не имеет никакой информации об объекте *p* и его полях.

- В выражении *p.x* в первую очередь необходимо понять, не имеет ли объект *p* какой-то специальной обработки обращения к полю *x*. *p* может быть, например, DOM-объектом, который нетривиальным образом перехватывает обращения к своим полям. Если нет никакой специальной обработки, нужно проверить для заданного объекта *p* существование поля *x*, где “*x*” - строка из одного символа. Объекты хранятся в виде таблиц, где символьным строкам соответствует значение поля или метод доступа. Если это метод доступа, он должен быть вызван. Если в таблице конкретное значение - оно и должно быть возвращено. Если в объекте *p* нет поля *x*, то необходимо повторить весь процесс поиска в прототипе объекта. Ускорение доступа к полям объектов с помощью профилирования или кэширования не рассматривается более подробно в данной работе.

- Бинарная операция умножения должна в первую очередь проверить типы операндов. Если операнд является объектом — то необходимо вызвать метод *valueOf* для данного объекта. Если

операнд является строкой — должна быть сделана попытка преобразовать строку в число. Когда получены операнды-числа, необходимо проверить, являются ли они целыми. Если да, то выполняется умножение целых чисел. Оно может вызвать переполнение, и тогда будет выполнено преобразование и умножение в типе *double*. Также оно будет выполнено сразу, если один из операндов-чисел не был целым числом. Получается, что результатом выполнения умножения может быть как целое число (*int32*), так и вещественное (*double*). И нет никакого способа в общем случае определить, какое это будет число и как оно будет представлено по результатам умножения.

- Бинарная операция сложения в выражении *p.x * p.x + p.y * p.y* сталкивается почти с теми же сложностями, что и операция умножения. Помимо перечисленных случаев для умножения, при операции сложения дополнительно должен быть рассмотрен вариант, когда оба операнда являются строками — для сложения строк должна быть выполнена их конкатенация. В нашем примере можно доказать, что такой вариант невозможен, поскольку результатом умножения строка быть не может, как и не может быть другой сложный объект. Однако по-прежнему необходимы проверки для *int32* или *double*, поскольку неизвестно, каков будет результат умножения. В итоге, результатом сложения также может быть как целое, так и вещественное число.

Идея определения типов в JSC основана на том, что мы с большой вероятностью можем предсказать типы, которые возвращают арифметические операции, если у нас есть предположение о типах операндов. Таким образом, возникает что-то вроде шага математической индукции — для операций, у которых мы можем предсказать типы операндов, мы можем предсказать и результат. Но для индукции нужна база, и базой становятся все операции, которые загружают внешние для заданной функции значения: например, загрузка полей объектов, использование аргумента функции, или использование возвращаемого значения. Типы значений для этих операций берутся из результатов профилирования значений на уровне LLInt и Baseline JIT. Каждой операции загрузки нелокального значения соответствует ровно одна ячейка информации о профиле, и там хранится последнее значение.

В самом простом виде алгоритм определения типов можно описать так: для каждого из хранящихся последних значений можно узнать тип, а дальше применить индукцию для распространения информации о типах на все операции внутри функции. Это дает нам предсказания типов для всех операций и переменных внутри функции.

В реальности, JavaScriptCore помимо последнего значения, хранит еще одно поле, которое описывает спекулятивный тип *SpecType*, в который вмещается

случайное подмножество значений, виденных ранее. Сначала этот тип заполняется как *SpecNone* — тип, которому не соответствует ни одно значение, аналог пустого множества. Когда выполнение программы проходит через эту точку, иногда включается логика профилирования, и поле типа заполняется новым значением. Новый тип должен включать в себя старый и одновременно допускать хранение записанного последнего значения.

Продвижение информации о типах *SpecType* по всем операциям выполняется с помощью стандартного итеративного алгоритма анализа потоков данных, реализованного как поиск неподвижной точки. На этапе DFG-компиляции этот алгоритм выполняется в одном из первых проходов, который обрабатывает построенный DFG граф.

После того, как для каждой операция в заданной функции был вычислен предсказанный тип значения, вставляются спекулятивные проверки типов. Например, для умножения делается проверка, что операнды являются числами. Если во время выполнения проверка получит отрицательный результат — будет выполнена деоптимизация, и выполнение перейдет на неоптимизированный код Baseline JIT. Это позволяет при выполнении DFG JIT кода использовать информацию о типах в последующих операциях. Например, пусть выполняется сложение *a+b*, и предсказаны целые типы операндов *SpecInt32*. До сложения будет вставлена проверка что *a* и *b* целые, иначе запускается механизм деоптимизации. После сложения будет вставлена проверка на переполнение, при переполнении также будет выполнен OSR exit. После завершения операции сложения можно считать известным, что операнды *a* и *b* и результат их сложения являются целыми числами, помещающимися в *int32*. Это позволяет при выполнении последующих операций не вставлять проверки для этих переменных. Удаление избыточных проверок реализовано с помощью второго алгоритма анализа потоков данных, использующего анализ потока управления на графе DFG. Анализ потока управления также выполняет условное продвижение констант, которое иногда позволяет аналогично информации о типах получить информацию о том, что какое-то значение является постоянным.

Вернемся к рассмотрению нашего примера, выражения *p.x * p.x + p.y * p.y*. Здесь потребуются только проверки загружаемых значений *p.x* и *p.y*. После проверки, что *p.x* и *p.y* являются числами, мы можем для хранения всех промежуточных результатов использовать тип *double*, и остается только выполнить два умножения и одно сложение чисел с плавающей точкой. Можно сказать, что почти всегда после удаления избыточных проверок DFG JIT код будет выполнять проверку типа не более одного раза для каждой загрузки внешнего значения.

4.1. Проверки на отрицательный ноль

Одним из аспектов, который необходимо учитывать при оптимизации арифметических операций, является отрицательный ноль. В стандарте языка

JavaScript числа не делятся на целые и вещественные, и подразумевается поведение всех чисел как вещественных, поэтому следует различать положительный и отрицательный ноль. Например, значениями следующих операций с целыми числами является минус бесконечность (-Infinity). *1 / (-0) ; 1 / (0 / -3) ; 1 / (-4 % 4)*. В случае, если для оптимизации выполнения мы заменяем выполнение операций с числами double операциями с целыми числами *int32*, необходимо, помимо проверок переполнения, учитывать также различие положительного и отрицательного нуля, чтобы не получить неверный с точки зрения стандарта результат. Необходимо отметить, что в некоторых ситуациях такие проверки можно опустить без ущерба для корректности выполнения. Например, при вычислении выражения *5/(a%b+3)*, если результатом выражения *a%b* является ноль, нет необходимости различать положительный ноль и отрицательный ноль.

В JavaScriptCore имеется реализация проверок арифметических операций на отрицательный ноль, однако она содержит несколько ошибок и другие недочеты. Исправление всех проблем, связанных с обработкой случая отрицательного ноля, значительно ускорило выполнение тестов.

Итак, первое исправление в логику было внесено как раз в части алгоритма, отвечающей за необходимость проверок. Для каждой операции в графе DFG выставляется флаг *NodeNeedsNegZero*, который установлен, когда для результата данной операции необходимо различать положительный и отрицательный ноль. Флаг в некоторых случаях расставлялся неправильно, правильный алгоритм таков: если результат вычисления выражения x в дальнейшем участвует в вычислении суммы *x+C*, *C+x* или разности *C−x*, где *C*—константа, не являющаяся отрицательным нолем (*C!=-0*), то для вычисления *x* не нужны проверки на отрицательный ноль. Аналогично для разности *x−C*, где *C!=+0*. Для всех других операций с числами флаг необходимо копировать. То есть операндам, если они сами получены как результат другой операции, он нужен тогда и только тогда, когда нужен для результата данной операции. Для операции унарного минуса флаг ошибочно стирался, это было исправлено.

Необходимо понять, в каких случаях целочисленные операции деления дают в результате отрицательный ноль, не представимый в виде *int32* значения. Операция деления *x/y* дает в результате -0, тогда и только тогда, когда *x* равен нолю и *y* меньше 0. Одна из ошибок реализации была в том, что на платформе ARM при равенстве делимого нулю сразу происходил OSR exit, без проведения проверки, что делитель отрицателен. Это вызывало множество ненужных возвратов на более медленный машинный код Baseline JIT.

Операция взятия остатка *x%y* дает в результате отрицательный ноль тогда и только тогда, когда *x* делится на *y* нацело, и x — отрицателен. Здесь также была реализована правильная проверка, а в имеющейся реализации после проверки результата на равенство нолю сразу выполнялся OSR Exit. Теперь он

выполняется только после дополнительной проверки, что делимое является отрицательным числом.

Для операции взятия остатка в DFG рассмотрен отдельно случай, когда делитель является константой и степенью двойки. В этом случае операция взятия остатка для оптимизации может быть заменена на операцию побитовой конъюнкции с числом на единицу меньшим. В случае отрицательного делимого также возможна такая реализация – необходимо предварительно поменять знак операнда, а в конце поменять знак результата. Однако в имеющейся реализации в этом случае пропускалась проверка на отрицательный ноль. Но при выставленном флаге *NodeNeedsNegZero* ее необходимо выполнять, чтобы не потерять знак в выражениях, таких как *z%4*, при значении *z* равном -4.

4.2. Поддержка новых операций байткода и внедренного машинного кода

Одной из неподдерживаемых операций на уровне DFG JIT является встроенная в JavaScript функция определения типа переменной – typeof. Поддерживался только часто используемый вариант, когда результат вызова typeof сравнивался со строковой константой, равной “number”, “string”, “object”, “function” или “undefined”. Причем, поддерживалось только сравнение на равенство, например, для конструкции отрицания сравнения !(typeof (x) == “string”) выполнялась DFG JIT компиляция. Для сравнения на неравенство (typeof (y) != “object”) компиляция не происходила, и функция, содержащая такую конструкцию, всегда работала на уровнях оптимизации не выше Baseline JIT. Нами было реализована полная поддержка операции typeof на уровне DFG JIT. Теперь поддерживается любой вариант использования, даже без последующего сравнения со строковой константой.

Другим примером неподдерживаемой операции являются циклы, организованные как перечисление всех полей объекта (for-in циклы). Для обработки таких циклов в граф DFG были добавлены все необходимые типы операций, и теперь функции, содержащие такой цикл, также могут эффективно выполняться на DFG JIT.

Посредством внедрения машинного кода вместо вызова с использованием JavaScript-API было реализовано ускоренное выполнение таких функций JavaScript, как *Math.power*, *Math.floor* и *String.fromCharCode*. Вместо того, чтобы организовывать сложный вызов функции на уровне DFG с большими затратами на создание пролога и эпилога для корректной работы со стеком обращения к таким функциям заменялись на более легковесные обращения к функциям в машинном коде, скомпилированном из языка C++ на этапе сборки JavaScriptCore. Для функции *Math.power* упрощенный вызов создавался только для случая, когда степень является целым числом.

Была также сделана попытка реализации *Math.floor* для платформы ARM вообще без вызова функции, с помощью использования операций с

плавающей точкой ARM NEON. Однако тестирование показало, что вариант с вызовом обычной C-функции floor показывает такую же производительность и создает меньший объем машинного кода.

5. Предварительные оптимизации

Одной из идей оптимизации JavaScriptCore является использование компиляции до выполнения (ahead of time compilation, AOTC)[5, 6], то есть добавление в систему предварительных оптимизаций. Изначально JavaScript используется для скриптов на веб-страницах, при загрузке страницы необходимо выполнить весь возникающий на ней код, и заранее про этот код ничего не известно. Однако теперь на языке JavaScript будут разрабатываться и более статичные приложения, хранящиеся на самом устройстве. Получается, что не обязательно использовать только подход интерпретатора, допускается выполнение некоторой подготовки кода. В данной работе разрабатывается идея, что исходный код заранее преобразуется в некоторый набор данных, содержащий байткод и другие внутренние представления, на которых можно провести какие-то предварительные оптимизации. Возможно также добавление сохранения машинного кода. Впоследствии, при выполнении программы загружаются готовые оптимизированные внутренние представления, которые корректируются по мере необходимости.

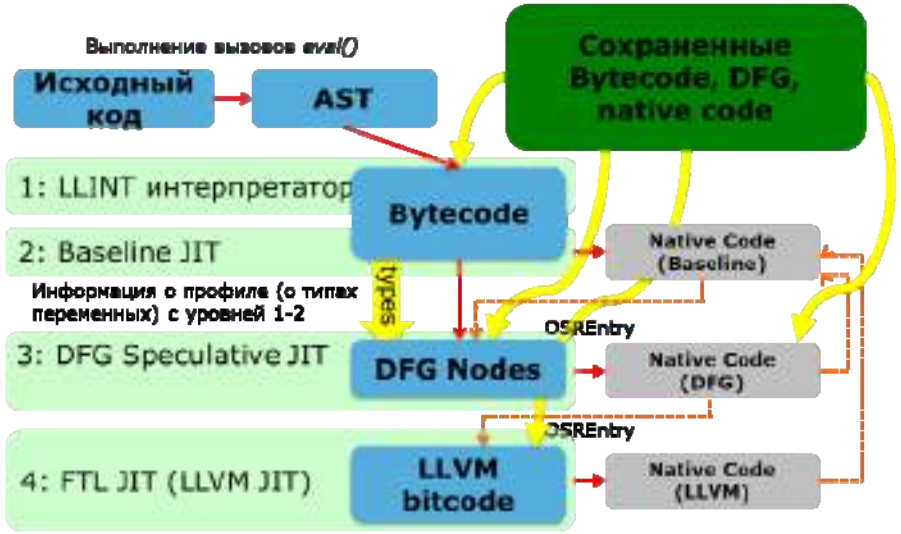

Рис. 2. Устройство системы предварительной компиляции (AOTC).

Такой порядок работы JavaScriptCore позволит получить следующие преимущества.

- В предварительной фазе могут быть выполнены значительно более сложные оптимизации, так как нет жестких ограничений по времени выполнения, имеющихся в JIT-компиляции. Однако в то же время необходимо отметить, что до выполнения программ-скриптов, нет никакой информации о профиле и значениях переменных.
- На этапе работы программы не тратится время на разбор и построение синтаксического дерева, кроме того, возможна экономия времени на генерацию машинного кода.
- Появляется шифрование исходного кода — байткод, другие внутренние представления и машинный код сложнее прочесть, чем исходный кода скрипта.

5.1. Сохранение и загрузка байткода

В данный момент в рамках работы над AOTC был реализован первый этап, который можно назвать AOTB (ahead of time bytecode). Он подразумевает сохранение исходного кода в виде байткода, для последующей загрузки при выполнении.

В обычном режиме работы JavaScriptCore при выполнении скрипта байткод генерировался только при первом вызове каждой функции. Нами была разработана и реализована схема генерации и сохранения байткода без выполнения самого скрипта. Вместе с байткодом сохраняется также вспомогательная информация, такая как таблицы констант, таблицы switch-переходов и исключений, необходимые данные для регулярных выражений. Для сохранения байткода без выполнения потребовалось сэмулировать работу стека пространств имен.

Байткод JavaScriptCore не был задуман как промежуточное внутреннее представление для сохранения, основной его целью является эффективное выполнение и генерация машинного кода на уровне Baseline JIT. Байткод, в отличие от исходной программы на JavaScript, отражает семантику программы только в определенном контексте. Например, в зависимости от свойств объектов, созданных к моменту начала выполнения программы, для нее может быть сгенерирован различный байткод. В основном, эта разница в байткоде относится к дополнительным подсказкам, например, позволяющим быстрее организовать обращение к полям объектов. Однако в некоторых случаях байткод, сохраненный вне того контекста, в котором программа будет исполняться, может приводить к некорректным результатам с точки зрения стандарта JavaScript. Эти особенности были учтены при сохранении байткода без выполнения. Кроме того, обращения к глобальным объектам содержат абсолютные адреса, и необходимо организовать сохранение так, чтобы можно было при загрузке байткода поменять адреса на новые, соответствующие адресам объектов во время выполнения.

Изначально планировалось хранение всей информации в виде базы данных SQLite [7], однако от такого способа хранения пришлось отказаться из соображений эффективной загрузки. Теперь все данные, относящиеся к одной функции, хранятся в виде последовательного набора байтов внутри файла. В начале файла сохраняется карта адресов (смещений), по которым можно найти информацию для каждой из функций. Соответственно, при выполнении из файла читается эта карта смещений и байткод для глобального JavaScript кода, то есть всего кода, описанного вне функций. В дальнейшей работе, при первом вызове функции вместо обычного разбора исходного кода байткод и все необходимые данные подгружается из файла по заданному смещению.

Необходимо отметить один из моментов, который позволил уменьшить размер сохраняемого файла — отказ от хранения двух вариантов байткода для каждой функции. При обычном выполнении JavaScript программ для функций, вызываемых как конструктор с помощью вызова new (“var z = new f()”), создается отдельный байткод. В нашей реализации хранится только байткод для случая обычного вызова функции, который при необходимости преобразуется в вариант “для конструктора”.

Таким образом, первый этап по решению задачи предварительной компиляции javascript программ выполнен. Реализовано сохранение и загрузка байткода. Выполняется статическая компиляция исходного кода в байткод без выполнения скрипта. При выполнении байткода вместо исходного кода стандарт ECMA-262[8] поддерживается полностью, вызовы eval поддерживаются, для них исходный код компилируется обычным образом в процессе работы JavaScriptCore. Исключением является только работа операций, явным образом требующих наличия исходного кода. Примерами таких операций могут служить вызовы function.toString(), либо использование поля line у объекта исключения. В этом поле должен храниться номер строки в исходном коде, которая создала исключение.

6. Результаты тестирования

Все найденные недостатки, связанные с реализацией проверок на отрицательный ноль в DFG коде, были устранены, что позволило исключить генерацию некорректного с точки зрения стандарта машинного кода. Исправление процесса удаления избыточных проверок позволило ускорить на платформе ARM несколько тестов из набора SunSpider. В среднем тестовый набор стал выполняться на 7% быстрее, ускорение конкретных тестов составляет до 35%.

Добавление ускоренного выполнения Math.power для случая целой степени ускоряет на 5% тест math-partial-sums из набора sunspider. Реализация for-in циклов ускоряет тест string-fasta на 2.5%.

Поддержка функции определения типа (typeof) ускоряет на 15% тест ArrayBlur из набора Browsermark. Что касается реализации fromCharCode и Math.floor,

они не дают видимого прироста производительности на исследуемых наборах тестов, однако увеличивают производительность искусственных тестов, проверяющих соответствующую функциональность.

Текущая реализация сохранения и загрузки файла с байткодом (AOTB) успешно проходит регрессионное тестирование на наборах из Webkit JavaScriptCore и V8. За счет уменьшения времени обработки исходного кода на 2-4% ускоряется работа тестов из SunSpider, v8 и kraken. Крупные data-файлы для тестов из kraken обрабатываются значительно быстрее, время их обработки не учитывается в результатах теста. По результатам профилирования работы JavaScriptCore было выявлено, что на больших исходных текстах время, затрачиваемое на загрузку файла с байткодом, может быть до 3-х раз меньше времени, необходимого на обычную обработку исходного кода.

Для тестов из наборов SunSpider, v8, kraken было измерено соотношение размера бинарного файла с сохраненным байткодом и размера исходного JavaScript-файла. Причем был взят пример как использования оригинальных файлов, так и файлов, упакованных с помощью Google Closure Compiler. Во втором случае оба файла дополнительно архивировались с помощью утилиты gzip с использованием максимального сжатия.

Таблица 2. Результаты сравнения объема JS-файлов и файлов с байткодом.

Тестовый набор	Соотношение размеров файла с байткодом и исходного файла	
	Оригинальный JavaScript	Google Closure Compiler + gzip
SunSpider	1.19	1.25
V8-v6	2.3	4.41
Kraken	1.97	1.31

7. Заключение

В рамках данной работы проведен анализ имеющихся в JavaScriptCore оптимизаций. По итогам исследования производительности на наборах тестов были выявлены недостатки в оптимизационных алгоритмах компиляции во время выполнения. Была добавлена поддержка компиляции новых операций, благодаря чему расширен класс функций, которые могут быть скомпилированы в более эффективный машинный код. Также были устранены недочеты, выявленные в системе предсказания типов переменных.

Изменения, внесенные в динамические оптимизации JavaScriptCore, позволили значительно ускорить выполнение тестовых наборов SunSpider, v8, kraken и Browsermark.

Для приложений на языке JavaScript, хранящихся локально на устройстве, была разработана система, позволяющая производить предварительную оптимизацию программ с последующей загрузкой и выполнением оптимизированного кода. Была реализована часть системы, позволяющая сохранять и загружать внутреннее представление – байткод. Она позволяет сократить до 3х раз время, затрачиваемое во время выполнения для получения готового байткода, поскольку компилятору не нужно делать лексический и синтаксический анализ.

В дальнейшем планируется продолжать разработку системы, добавив в нее предварительные оптимизации на уровне байткода. Необходимо также рассмотреть возможность сохранения других промежуточных представлений и оптимизированного машинного кода для последующей загрузки при выполнении.

Список литературы

[1] Веб-сайт платформы Tizen. http://www.tizen.org

[2] Описание реализации JavaScriptCore на веб-сайте разработчиков WebKit http://trac.webkit.org/wiki/JavaScriptCore

[3] Веб-сайт Webkit. http://www.webkit.org

[4] S. Li, B. Cheng, X. Li “TypeCastor: demystify dynamic typing of JavaScript applications”, Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, 2011, pp. 55-65

[5] S. Hong, J. Kim, J. W. Shin, S. Moon, H. Oh, J. Lee, H. Choi “Java client ahead-of-time compiler for embedded systems”, Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, 2007, pp. 63-72

[6] S. Hong, S. Moon “Client-Ahead-Of-Time Compilation for Digital TV Software Platform” 3rd workshop on Dynamic Compilation Everywhere preprint, 2013. http://sites.google.com/site/dynamiccompilationeverywhere/home/dce-2014/DCE-2014-Sunghyun-Hong-article.pdf

[7] Веб-сайт SQLite. http://www.sqlite.org/about.html

[8] Описание стандарта ECMA-262 http://www.ecma-international.org/publications/standards/Ecma-262.htm

Dynamic and ahead of time optimization for JavaScript programs

Roman Zhuykov <zhroma@ispras.ru>
Dmitry Melnik <dm@ispras.ru>,
Ruben Buchatskiy <ruben@ispras.ru>
Vahagn Vardanyan <vaag@ispras.ru>
Mamikon Vardanyan <mamikon@ispras.ru>,
Vladislav Ivanishin <vladislav.ivanishin@gmail.com>,
Eugene Sharygin <eugene.sharygin@gmail.com>

Abstract. The paper is dedicated to performance improvement of JavaScript programs. In this work we examine the specifics of dynamic optimizations in JIT-compiler for JavaScript, and how the performance of such optimizations can be improved. Also we propose a method for ahead-of-time (AOT) compilation of JavaScript programs, and for saving them as a bytecode. This method allows reducing startup time of applications by moving the optimizations to AOT phase. The proposed methods were implemented in open-source WebKit library, and resulted in significant performance gain for popular JavaScript benchmarks on ARM platform.

Keywords: program optimizations; JIT optimization; ahead of time optimization; data flow graph; ARM.