Методы предварительной оптимизации программ на языке JavaScript

[1]Роман Жуйков <zhroma@ispras.ru>
[2]Евгений Шарыгин <eugene.sharygin@gmail.com>
[1] Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1., стр. 52, факультет ВМК

Аннотация. Современные виртуальные машины для языка JavaScript используют многоуровневую компиляцию во время выполнения для создания машинного кода. При компиляции во время выполнения нецелесообразно выполнение сложных оптимизаций. Статическая компиляция, наоборот, имеет неограниченные возможности для выполнения сложных оптимизационных преобразований, но не может эффективно применяться к динамическим языкам, таким как JavaScript. В данной работе предлагается общий подход к предварительной компиляции программ на динамических языках, а также применение этого подхода для улучшения двух виртуальных машин — JavaScriptCore и V8. При реализации улучшенной виртуальной машины JavaScriptCore с использованием предварительной компиляции была учтена специфика использования JavaScript-программ в составе локально хранящихся приложений для платформы ARM. Для виртуальной машины V8 для платформы x86-64 в рамках исследования предварительная компиляция была реализована с помощью кэширования в отдельный файл одного из оптимизированных внутренних представлений.

Ключевые слова: оптимизация программ; JavaScript; компиляция во время выполнения; предварительная компиляция; Webkit JavaScriptCore; виртуальная машина V8

DOI: 10.15514/ISPRAS-2015-27(6)-5

Для цитирования: Жуйков Р.А., Шарыгин Е.Ю. Методы предварительной оптимизации программ на языке JavaScript. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 67-86. DOI: 10.15514/ISPRAS-2015-27(6)-5.

1. Введение

В данной работе рассматриваются две виртуальных машины для языка JavaScript. Первая виртуальная машина называется JavaScriptCore (JSC) [1] и

входит в состав браузерного движка WebKit [2] для отображения веб-страниц. Вторая виртуальная машина называется V8 [3] и используется в браузерах Chromium и Chrome. Обе виртуальных машины являются свободным программным обеспечением с открытым исходным кодом и являются многоуровневыми Just-In-Time (JIT) компиляторами, то есть содержат несколько реализаций компиляции программного кода в машинный код во время выполнения программы.

Целью данной работы является разработка подхода к предварительной компиляции программ на динамических языках [4, 5], а также применение этого подхода для разработки схемы улучшения виртуальной машины JavaScriptCore с использованием предварительной компиляции. При реализации улучшенной версии виртуальной машины JavaScriptCore необходимо учесть специфику использования JavaScript-программ в составе локально хранящихся приложений для платформы ARM. Кроме того, необходимо с использованием подхода разработать и реализовать предварительную компиляцию в рамках виртуальной машины V8 для платформы x86-64.

Дальнейшее изложение построено следующим образом. Сначала будет описана общая схема работы многоуровневых JIT-компиляторов, а также особенности реализации этой схемы в виртуальных машинах JavaScriptCore и V8. Потом будут описаны возможные направления улучшения производительности данных виртуальных машин. Далее будет описана общая схема подхода к применению идей предварительной компиляции в рамках многоуровневого JIT, и ее реализация в рамках JavaScriptCore и V8.

2. Многоуровневая JIT-компиляция в JavaScriptCore и V8.

Основная идея многоуровневой JIT-компиляции состоит в том, что время, потраченное на генерацию машинного кода для какого-то участка исходного кода, зависит от горячести этого участка. Для участков кода, которые выполняются один раз, оптимальным вариантом обычно является использование интерпретации. Следующим шагом является базовая JIT-компиляция — генерация неоптимизированного машинного кода, соответствующего заданному участку исходного кода. Для еще более горячих участков кода возможно использование спекулятивных оптимизаций с использованием профиля, собранного на предыдущих уровнях выполнения.

В JavaScriptCore единицей трансляции является функция на языке JavaScript, первыми этапами работы являются лексический и синтаксический анализ. Общая схема работы JavaScriptCore изображена на рис. 1. Исходный код разбивается на токены, методом рекурсивного спуска строится синтаксическое дерево (abstract syntax tree, AST), 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. Необходимо отметить, что байткод для функции создается непосредственно во время выполнения программы при первом вызове данной функции.

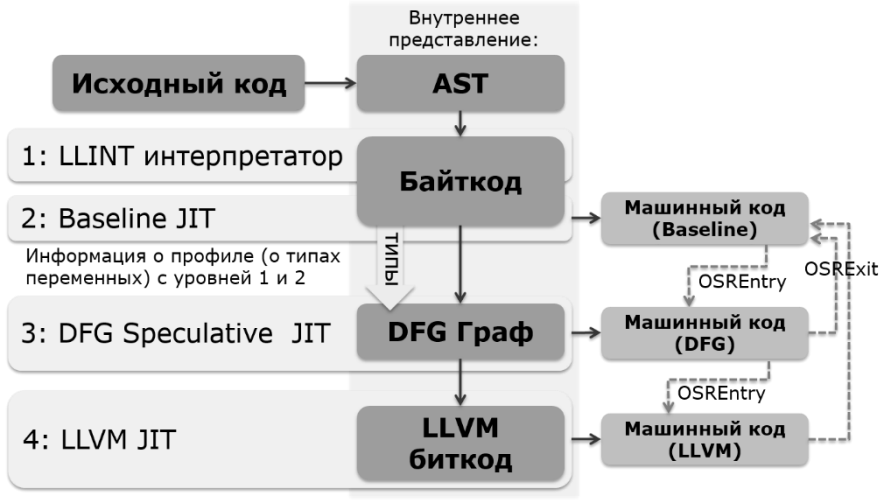

Рис. 1. Устройство виртуальной машины JavaScriptCore

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 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 так же 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, 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, так же 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 качестве внутреннего представления помимо DFG графа используется биткод компилятора LLVM. Перед генерацией машинного кода выполняются оптимизации, уже реализованные в компиляторе LLVM.

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.

Табл. 1. Сравнение производительности уровней JSC.

Тест	v8-richards		Browsermark	
Ускорение	Быстрее интерпрета-тора, раз	Быстрее предыдущего уровня, раз	Быстрее LLInt, раз	Быстрее предыдущего уровня, раз
Интер-претатор	1.00	-	н/д	-
LLInt	2.22	2.22	1.00	-
Baseline JIT	15.36	6.90	2.50	2.5
DFG JIT	61.43	4.00	4.25	1.7
Код на C	107.50	1.75	н/д	

Все уровни выполнения обеспечивают одинаковую семантику выполнения, и единственный эффект переключения между ними — в производительности работы JavaScriptCore. В табл. 1 приведены примеры сравнения скорости выполнения теста v8-richards, а также набора тестов Browsermark. Для теста v8-richards дополнительно приведена скорость выполнения аналогичных вычислений, запрограммированных на языке C. Для набора тестов Browsermark не проводилось измерения производительности обычного интерпретатора в составе JSC, указанное в табл. 1 ускорение взято по среднему геометрическому, и необходимо отметить, что в одном из тестов наблюдается пятикратное преимущество Baseline JIT над LLINT интерпретатором, а также есть пример, где DFG JIT оказывается в 6 раз быстрее чем Baseline JIT.

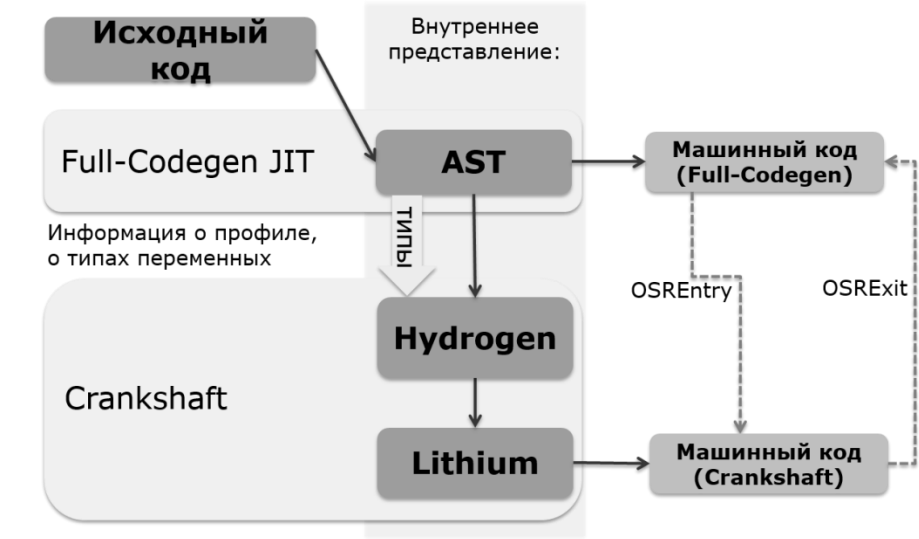

Рис. 2. Устройство виртуальной машины V8

Кратко опишем устройство виртуальной машины V8, изображенное на рис. 2. В V8 используется только два уровня выполнения, причем оба эти уровня осуществляют JIT-компиляцию, а уровень интерпретации отсутствует. Базовый не оптимизирующий JIT-компилятор называется Full-Codegen, а оптимизирующий JIT называется Crankshaft. В Crankshaft в качестве промежуточных представлений используется машинно-независимое представление высокого уровня Hydrogen, на котором выполняется большинство оптимизаций, и машинно-зависимое представление низкого уровня Lithium, на котором происходит распределение регистров и из которого генерируется машинный код. В виртуальной машине V8 так же имеется схема On-Stack Replacement, позволяющая переходить на оптимизированный код, не дожидаясь окончания выполнения функции, и позволяющая в случае ошибочных спекулятивных предположений перейти обратно на неоптимизированную версию машинного кода.

3. Оптимизация производительности многоуровневого JIT

Рассмотрим и проанализируем возможные методы улучшения производительности виртуальных машин. Первым направлением является улучшение имеющихся оптимизаций на спекулятивном уровне JIT-компиляции. Как и в случае статической компиляции программ на типизированных языках в данном направлении существует множество возможностей для адаптации оптимизационных преобразований для

особенностей той или иной платформы, либо определенного класса программ. Однако, необходимо учитывать, что, несмотря на поддержку выполнения JIT-компиляции в отдельном потоке, не прерывая выполнения кода на основном ядре процессора, требования ко времени JIT-компиляции предъявляются более жесткие, так как в отличие от статической компиляции возможны сценарии, когда время, затраченное на более сложную оптимизацию, не окупится в терминах времени выполнения кода. Также необходимо отметить, что дополнительный рост сложности оптимизационных преобразований приводит к большим требованиям к размеру оперативной памяти, что может так же оказаться существенным, например, в случае встраиваемых архитектур.

Следующим направлением оптимизации является создание дополнительных уровней JIT-компиляции с использованием спекулятивных оптимизаций, а также тонкая настройка взаимодействия различных уровней выполнения. Дополнительный уровень JIT-компиляции позволяет для самых горячих участков кода выполнить максимально сложный набор оптимизаций, в то время как менее горячие участки так же будут оптимизированы, но с меньшими затратами. Задача выбора эвристик и управления их параметрами для того, чтобы определять момент перехода на следующий уровень выполнения так же является достаточно обширной областью для исследований. Одним из аспектов тонкой настройки может являться не только аспект производительности, но и аспект энергопотребления процессора.

И наконец третьим направлением улучшения виртуальных машин JavaScript является применение идей предварительной компиляции. Данный подход особенно актуален для использования в сценарии выполнения приложений, локально сохраненных на встраиваемом устройстве. Использование предварительной компиляции позволит в такой ситуации совместить преимущества динамической JIT-компиляции и статической компиляции.

Основным вариантом реализации идей предварительной оптимизации является использование кэширования на уровне различных внутренних представлений виртуальной машины, возможно даже использование комбинации из нескольких внутренних представлений. Для некоторых представлений потенциально возможна разработка дополнительных статических оптимизаций, выполняемых в оффлайн фазе. На примере JavaScriptCore рассмотрим каким образом сохранение и последующая загрузка внутреннего представления позволяют увеличить производительность многоуровневого JIT.

Сначала для интересующих нас наборов тестов v8-v6 и SunSpider составим диаграмму профиля, показывающую какой объем времени затрачивается на выполнение тех или иных этапов работы виртуальной машины JavaScriptCore. Диаграмма изображена на рис. 3.

По такой диаграмме можно указать теоретическое максимально возможное увеличение производительности при наличии сохраненных внутренних

представлений. Например, сохранение такого внутреннего представления как байткод позволяет избавиться от двух начальных этапов — построения синтаксического дерева и непосредственно построения байткода. Однако, при этом на саму загрузку внутренних представлений так же необходимо затратить время выполнения, причем важным аспектом является линковка — восстановление ссылок на все внешние для данного представления объекты.

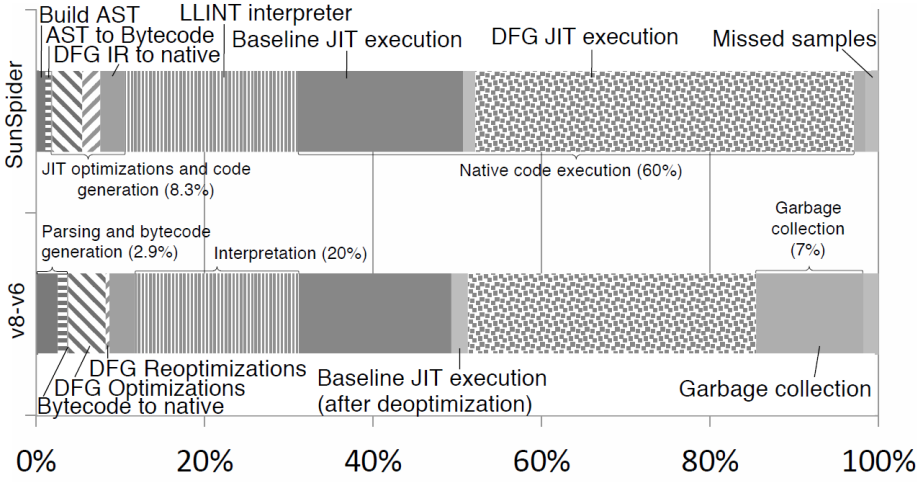

Рис. 3 Результаты профилирования JavaScriptCore

При сохранении оптимизированных внутренних представлений аналогичным образом можно оценить максимально возможное ускорение виртуальной машины для того или иного набора тестов. Для этого, помимо учета той особенности, что можно сэкономить время разбора исходного кода и построения некоторой части промежуточных представлений, необходимо учесть, что успешная загрузка кода позволит выполнять данную единицу трансляции сразу на оптимизированном уровне, не используя более медленные способы выполнения, как это делается в обычном режиме, до тех пор, пока функция не является горячей. Построив для интересующих нас тестов таблицу аналогичную табл. 1, можно получить максимальную оценку дополнительного ускорения, полученного таким образом. Если построить диаграмму как на рис. 3 для тестов из набора Browsermark, по такой диаграмме можно оценить, что потенциально время, обозначенное как “Baseline JIT Execution” может быть уменьшено в 1.7 раза, а время “LLINT interpreter” можно уменьшить в 4.25 раз.

Помимо максимального теоретически возможного ускорения отметим следующие аспекты, которые необходимо принимать во внимание при выборе комбинации внутренних представлений для сохранения. Во-первых, необходимо рассматривать возможность дополнительной статической

оптимизации в оффлайн фазе. Например, для байткода в JavaScriptCore может быть использовано удаление общих подвыражений. Во-вторых, необходимо оценивать сложность линковки во время выполнения. Например, при загрузке машинного кода большое количество времени будет тратиться на замену глобальных адресов, используемых в коде, на актуальные для текущего запуска программы. Третьим аспектом является вопрос поддержки перехода на менее оптимизированный код в случае невыполнения спекулятивных предположений на более высоких уровнях JIT. В обычном режиме работы виртуальной машины все необходимые данные для такого перехода имеются, но при загрузке готового оптимизированного внутреннего представления необходимо так же реализовать создание всей необходимой информации, например, обеспечить готовность создания неоптимизированного варианта JIT-кода. И наконец, в-четвертых, необходимо оценить трудоемкость реализации сохранения и загрузки выбранных внутренних представлений, с учетом реализации предыдущих трех аспектов, обозначенных в данном абзаце.

4. Реализация предварительной оптимизации в JavaScriptCore

В рамках JavaScriptCore была разработана идея инфраструктуры, в которой исходный код заранее преобразовывался в некоторый набор данных, содержащий байткод и другие внутренние представления. Впоследствии, при выполнении программы загружаются готовые оптимизированные внутренние представления, которые корректируются по мере необходимости. Общая схема работы разрабатываемой системы изображена на рис. 4.

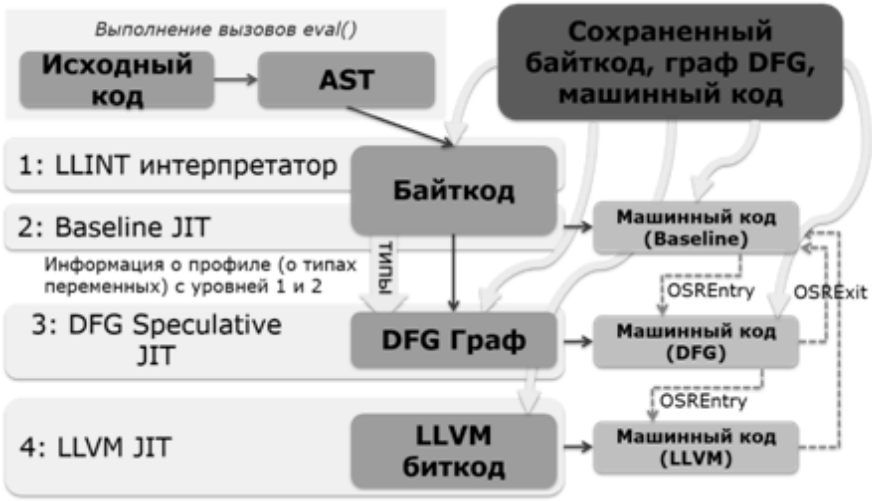

Рис. 4. Устройство системы предварительной компиляции (AOTC).

Такой порядок работы JavaScriptCore позволит получить помимо преимуществ, указанных в предыдущей главе, некоторое подобие шифрования исходного кода — ведь байткод и другие бинарные внутренние представления сложнее прочесть и изменить, чем обычный исходный JavaScript код.

4.1 Сохранение и загрузка байткода

Сначала в рамках работы над предварительной компиляцией (ahead of time compilation, AOTC) был реализован первый этап, который можно назвать AOTB (ahead of time bytecode) [6]. Он подразумевает сохранение исходного кода в виде байткода, для последующей загрузки при выполнении.

В обычном режиме работы JavaScriptCore при выполнении скриптов байткод для каждой функции генерировался только при первом вызове этой функции. Нами была разработана и реализована схема генерации и сохранения байткода без выполнения самого скрипта. К байткоду сохраняется так же вспомогательная информация, такая как таблицы констант, таблицы switch-переходов и исключений, необходимые данные для регулярных выражений. Для сохранения байткода без выполнения потребовалось эмулировать 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, однако 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”.

При выполнении байткода вместо исходного кода стандарт ECMA-262 [7] 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.

4.2 Результаты AOTB

Текущая реализация сохранения и загрузки файла с байткодом успешно проходит регрессионное тестирование на наборах из Webkit JavaScriptCore и V8. За счет уменьшения времени обработки исходного кода, до 2 – 4% ускоряется работа тестов из SunSpider, v8 и kraken на платформе ARM. 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 3x 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 упакованных с помощью Google Closure Compiler. Во втором случае оба файла дополнительно архивировались с помощью утилиты gzip с использованием максимального сжатия. Результаты данных измерений изображены в табл. 2.

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

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

Увеличение размера файла является значительным, но при выполнении файла с байткодом не используется больший объем оперативной памяти, чем при запуске JavaScriptCore в обычном режиме для выполнения аналогичной JavaScript программы. В современных встраиваемых системах объем внешней памяти существенно превышает объем оперативной памяти, и не является критическим параметром при создании программного обеспечения, поэтому инфраструктура AOTB может иметь достаточно широкое применение.

4.3 Сохранение машинного кода Baseline JIT

После реализации AOTB в данную инфраструктуру для платформы x86-64 были добавлены сохранение и загрузка неоптимизированного машинного кода, генерируемого на уровне Baseline JIT. Самой сложной задачей в этой работе была линковка адресов. При загрузке, для всех объектов, обращение к которым производится по абсолютным адресам, необходимо было внести актуальные адреса в загружаемый машинный код. На этапе сохранения генерация машинного кода была так же реализована без выполнения скрипта. При генерации машинного кода, все обращения по абсолютным адресам фиксировались в отдельные таблицы, чтобы в дальнейшем была возможность восстановить необходимые адреса при загрузке. Машинный код и все необходимые для линковки данные записывались рядом с байткодом соответствующей функции в тот же бинарный файл. При этом абсолютные адреса в машинном коде, которые будут заново сформированы при загрузке, при сохранении в файл заменялись нулями для более успешной работы алгоритмов сжатия.

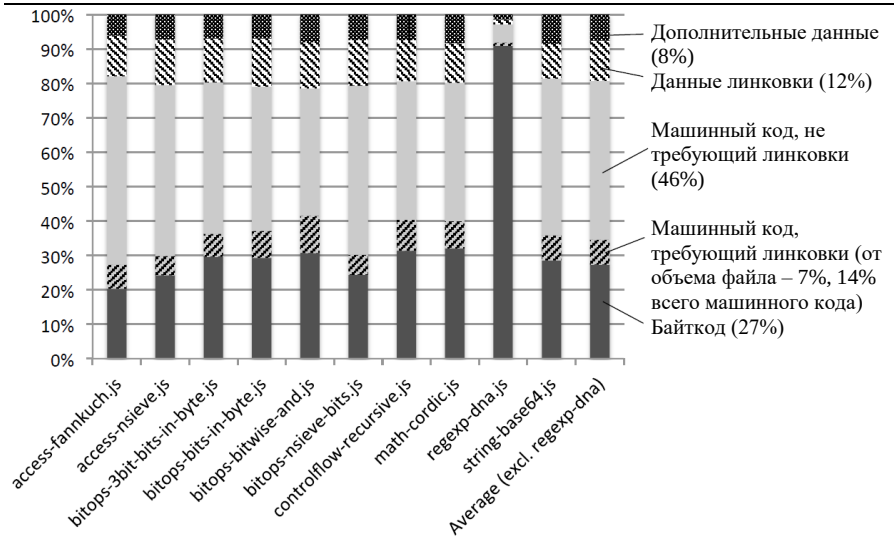

Рис. 5. Структура бинарного файла

В итоге для каждой функции в файле хранились следующие группы данных:

- Байткод и другая информация, необходимая для AOTB
- Машинный код, создаваемый Baseline JIT
- Данные для линковки
- Дополнительные технические данные генерируемые для функции после работы Baseline JIT, их необходимо сохранять из-за пропуска генерации Baseline JIT в варианте запуска с загружаемым готовым машинным кодом

Размер получаемых файлов оказался еще в 2.5 – 5 раз больше, чем размер файлов, генерируемых при работе AOTB, то есть содержащих только байткод. На рис. 5 изображена статистика по объему данных, содержащихся в полученных бинарных, для 10 тестов из набора SunSpider.

Поскольку на x86-64 размер указателя составляет 8 байт, получается, что значительная часть машинного кода состоит из абсолютных адресов. В данном случае, под линковку попадает от 10% до 23%, в среднем 14% машинного кода. Возможно, из-за этого оказывается, что сохранение машинного кода не позволяет ускорить производительность JavaScriptCore на тестах SunSpider и V8. Получается, процесс генерации машинного кода из готового байткода оказывается не таким уж медленным, и чтение значительно более объемного файла с последующей линковкой адресов не дает выигрыша в производительности.

5. Предварительная оптимизация в виртуальной машине V8

Поскольку в виртуальной машине V8 отсутствует самое общее внутреннее представление аналогичное байткоду, было выбрано принципиально иное инфраструктурное решение о реализации предварительной компиляции. Файл с сохраняемой информацией должен лишь дополнять исходный файл, а не заменять его.

От сохранения машинного кода было решено отказаться из-за сложности реализации и плохих практических результатов такого подхода в JavaScriptCore. Сохранение сложного высокоуровневого представления Hydrogen так же было решено не делать, в силу его сложной структуры, необходимой для проведения всех оптимизационных проходов. Машинно-зависимое внутреннее представление Lithium оказалось оптимальным для сохранения – в нем достаточно легко собирать необходимый объем информации для генератора машинного кода, и одновременно не стоит так остро вопрос актуализации ссылок на используемые объекты, как это происходит с адресами в машинном коде.

5.1 Сохранение Lithium

Итак, для виртуальной машины V8 была разработана система кэширования в файл оптимизированного внутреннего представления Lithium уже после оптимизационного прохода распределения регистров, то есть сохраняется ровно тот вариант внутреннего представления, который передается генератору машинного кода. Один из недостатков этого подхода – зависимость от архитектуры системы, так как представление является машинно-зависимым. В рамках данной работы была выполнена исследовательская реализация для процессоров x86-64.

Система встроена в V8 и доступна при запуске через интерфейс командной строки. Если обычный запуск производится командой v8 test.js, то модифицированная нами версия v8 позволяет добавить опцию -save-code=<filename> или -load-code=<filename> соответственно для сохранения и загрузки Lithium представления. Необходимо отметить, что в данной реализации поддерживается работа только с одним исходным файлом на JavaScript. Предположительно, при реализации аналогичного подхода для реальных крупных приложений на JavaScript следует использовать один общий файл кэширующий все Lithium-представления в рамках запускаемого приложения.

Некоторая часть инструкций Lithium не является самодостаточными, и содержит ссылку на «родительскую» инструкцию Hydrogen из которой она была сгенерирована. Генератор машинного кода в Crankshaft использует небольшую часть свойств инструкций Hydrogen, доступных по соответствующим ссылкам из Lithium.

Воссоздание Hydrogen в полном объеме на этапе загрузки предварительно загруженных инструкций Lithium не является целесообразным, а в некоторых случаях это даже невозможно без значительного вмешательства во внутреннюю структуру Hydrogen для отключения или обхода многочисленных проверок целостности.

В ходе данной работы было принято решение о создании дополнительного уровня между Hydrogen и Lithium. Данный дополнительный уровень содержит в точности объем информации необходимый для генерации машинного кода, имеет аналогичный Hydrogen интерфейс доступа к свойствам инструкций и содержит лишь нужную часть этих свойств.

Идентичность интерфейса позволяет минимизировать изменения, вносимые в генератор машинного кода компилятора Crankshaft, требуется лишь заменить обращения к тем же методам другого класса. Классы инструкций дополнительного уровня образуют иерархию, схожую и оригинальной иерархией классов инструкций Hydrogen. Инструкции дополнительного уровня сохраняются вместе с инструкциями Lithium.

5.2 Результаты

На искусственных тестовых примерах ускорение работы виртуальной машины составило до 20 раз. Для создания такого примера, необходимо выбирать такое количество итераций цикла внутри функции, которое минимально подходит для запуска Crankshaft-компиляции данной функции. Тогда в режиме запуска с загружаемым готовым представлением Lithium получается, что такая функция все итерации выполняет на оптимизированной версии кода, в то время как исходном режиме работы виртуальной машины V8 такая функция почти все время выполнялась на неоптимизированной версии машинного кода. А соотношение до 20 раз может быть достигнуто при добавлении в цикл нескольких присваиваний в неиспользуемые в дальнейшем переменные, поскольку не оптимизирующий компилятор Full-Codegen не реализует удаления мертвого кода.

Необходимо отметить, что реализация сохранения и загрузки Lithium была сделана лишь в рамках исследовательской работы, поэтому не была реализована совместимость механизма сохранения и загрузки внутреннего представления с обычно используемым на современных многоядерных процессорах режимом работы виртуальной машины V8, при котором компиляция оптимизированного кода происходит в фоновом режиме в отдельном потоке параллельно выполнению самого скрипта. В качестве базового времени для сравнения производительности использовались результаты работы оригинальной версии V8 с отключенной компиляцией в фоновом режиме.

Тестирование производительности производилось только при запуске тестов в режиме загрузки с теми же параметрами и входными данными, что и при сохранении. Данное ограничение связано с реализацией сохранения и загрузки

только для одного исходного файла в рамках одного запуска, а также с отсутствием промышленных тестов для языка JavaScript, позволяющих передавать входные данные посредством командной строки.

К сожалению, на тестах из наборов SunSpider, Octane и Kraken не было получено ускорения. Возможно, это связано с тем, что в данных тестах незначительное время тратится на работу оптимизирующего компилятора, а также нет функций, выполнение которых существенное время происходит на неоптимизированном машинном коде с последующей оптимизацией.

Поскольку в виртуальной машине V8 отсутствует поддержка статической генерации функций, реализованных на языке asm.js [8], являющемся подмножеством языка JavaScript, было решено протестировать сравнительно объемные тесты на asm.js.

Ускорение было получено на трех asm.js приложениях с сайта “Are we fast yet?” [9], занимающегося сравнением производительности различных JavaScript движков. Тесты Box2d и Bullet данного сайта ускоряются до 15%, а тест Zlib — до 33%. Данные тесты можно запускать с указанием выполняемого объема вычислений, передавая в командной строке число от 1 до 5. Важно отметить, что такое ускорение получено при выполнении с параметром 1. Однако, в абсолютном выражении данное ускорение сохраняется и при использовании большего объема вычислений, тесты ускоряются на 0.1 – 0.2 секунды. Таким образом, можно предположить, что при реализации данной исследовательской работы в виде реально используемого программного продукта в составе веб-движка для отображения веб-страниц, пользовательское JavaScript-приложение, основанное на box2d, будет загружаться на 0.1 секунду быстрее при использовании сохранения и загрузки Lithium.

Дополнительно был создан еще один тест на asm.js. Для статических компиляторов языка C++ существует тест tramp3d [10], предназначенный для оценок времени работы самих компиляторов. То есть обычно данный достаточно объемный исходный код на C++ используется для сравнения скорости компиляции. В рамках исследования, данный тест был скомпилирован с помощью emscripten в исходный файл на языке asm.js и полученный скрипт был протестирован. Поскольку в V8 компиляция происходит во время выполнения, и подход с загрузкой готового ранее сохраненного Lithium кода позволил уменьшить время компиляции, на таком тесте было получено ускорение в 30%.

6. Заключение

В рамках данной работы разработан общий метод применения предварительной компиляции в виртуальных машинах с использованием многоуровневой JIT-компиляции. С помощью этого метода разработан и реализован для платформы ARM оптимизирующий JavaScript компилятор на базе виртуальной машины JavaScriptCore, позволяющий добиться более

высокой производительности для локально хранимых программ. Данная реализация позволяет избавиться от хранения JavaScript кода в виде открытых исходных файлов, сохраняя код в файл в виде байткода JavaScriptCore, и позволяет сократить до 3х раз время, затрачиваемое во время выполнения для получения байткода, поскольку компилятору не нужно делать лексический и синтаксический анализ. Дополнительно к байткоду в файл может быть также сохранен неоптимизированный машинный код. На некоторых тестах данная реализация позволяет получить ускорение до 2 – 4%, при этом размер файла с байткодом может быть до 4.5 раз больше исходных файлов на JavaScript, а при сохранении машинного кода объем файла увеличивается еще в 2.5 – 5 раз без существенных улучшений производительности. В дальнейшем планируется продолжать разработку системы, добавив в нее предварительные оптимизации на уровне байткода.

Для виртуальной машины V8 с помощью применения метода разработан механизм кэширования внутренних представлений и реализована возможность сохранения и последующей загрузки промежуточного оптимизированного машинно-зависимого внутреннего представления, что позволяет ускорить повторный запуск программы. Данная реализация позволяет значительно ускорить специальный образом подобранные искусственные тесты, а также ряд тестов на языке asm.js. В данный момент — это лишь исследовательская разработка и в дальнейшем планируется расширить ее для возможности реального использования в составе систем, аналогичных движкам веб-браузеров.

Список литературы

[1]. Веб-страница описания реализации JavaScriptCore на веб-сайте разработчиков WebKit. http://trac.webkit.org/wiki/JavaScriptCore
[2]. Веб-сайт Webkit. http://www.webkit.org
[3]. Веб-сайт V8. https://code.google.com/p/v8/
[4]. 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
[5]. 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
[6]. Р. Жуйков, Д. Мельник, Р. Бучацкий, В. Варданян, В. Иванишин, Е. Шарыгин. Методы динамической и предварительной оптимизации программ на языке JavaScript. // Труды Института системного программирования РАН Том 26. Выпуск 1. 2014 г. Стр. 297-314. DOI: 10.15514/ISPRAS-2014-26(1)-10
[7]. Описание стандарта ECMA-262. http://www.ecma-international.org/publications/standards/Ecma-262.htm
[8]. Веб-сайт языка asm.js. http://asmjs.org/
[9]. Веб-сайт “Are we fast yet?”. https://arewefastyet.com/
[10]. Веб-страница тестирования tramp3d. http://gcc.opensuse.org/c++bench/tramp3d/

Ahead of time optimization for JavaScript programs

[1]*Roman Zhuykov <zhroma@ispras.ru>*
[2]*Eugene Sharygin <eugene.sharygin@gmail.com>*
[1]*Institute for System Programming of the RAS,*
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. Modern JavaScript engines use just-in-time (JIT) compilation to produce binary code. JIT compilers are limited in a complexity of optimizations they can perform at a runtime without delaying an execution. Static ahead-of-time (AOT) compilers do not have such limitations, but they are not well suited for compiling dynamic languages such as JavaScript. In the paper, we discuss methods for augmenting multi-tiered JIT compiler with a capability for AOT compilation, and implement some of ahead-of-time compilation ideas in two JavaScript engines — JavaScriptCore and V8. In JavaScriptCore (JSC), which is a part of open-source WebKit library, we have developed and implemented a framework, which allows saving of JavaScript programs as a binary package containing bytecode and a native code. The framework consists of two components: command-line compiler, which compiles source JavaScript program into compressed binary package, consisting of JSC internal representation called bytecode and native code produced by JSC's non-optimized JIT compiler. The second component is patched JSC engine with a capability for loading and executing binary packages produced by the compiler. In addition, we have implemented saving of optimized internal representation in another JavaScript engine, which is called V8 and is used in Chrome and many other browsers. When running the same JavaScript program, cached internal representation can be loaded to reduce JIT-compilation time and decrease percentage of running unoptimized code before it became hot.

Keywords: program optimizations; JavaScript; just-in-time optimization (JIT-optimization); ahead of time optimization; Webkit JavaScriptCore; V8.

DOI: 10.15514/ISPRAS-2015-27(6)-5

For citation: Zhuykov Roman, Sharygin Eugene. Ahead of Time Optimization for JavaScript Programs. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 67-86 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-5

References

[1]. JavaScriptCore webpage on WebKit website. http://trac.webkit.org/wiki/JavaScriptCore
[2]. WebKit Browser Engine website. http://www.webkit.org
[3]. V8 Browser Engine webpage. https://code.google.com/p/v8/
[4]. 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

[5]. 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

[6]. R. Zhuykov, D. Melnik, R. Buchatskiy, V. Vardanyan, V. Ivanishin, E. Sharygin. Metody dinamicheskoj i predvaritel'noj optimizacii programm na jazyke JavaScript [Dynamic and ahead of time optimization for JavaScript programs]. *Trudy ISP RAN* [*The Proceedings of ISP RAS*], 2014, vol. 26, no. 1, pp. 297-314. DOI: 10.15514/ISPRAS-2014-26(1)-10 (in Russian)

[7]. ECMA-262 Standard description. http://www.ecma-international.org/publications/standards/Ecma-262.htm

[8]. Asm.js language website. http://asmjs.org/

[9]. "Are we fast yet?" website. https://arewefastyet.com/

[10]. Tramp3d testing webpage. http://gcc.opensuse.org/c++bench/tramp3d/