Методы оптимизации программ на языке JavaScript, основанные на статистике выполнения программы

В.Г. Варданян <vahagvardanyan@gmail.com>
Ереванский государственный университет, 0025, Армения,
г. Ереван, ул. А. Манукяна, дом 1

Аннотация. Язык JavaScript является одним из самых популярных языков для разработки веб-приложений. В связи с ростом производительности персональных компьютеров, мобильных и встраиваемых систем использование JavaScript стало возможным также и в масштабных приложениях. Более того, в настоящее время язык JavaScript активно используется в операционных системах в качестве одного из основных языков для создания пользовательских приложений. Примерами таких систем являются Tizen OS и Firefox OS. С ростом популярности языка многие крупные компании выпустили свои реализации JavaScript, в которых для генерации машинного кода в основном используется многоуровневая динамическая компиляция. В данной работе описываются разработанные методы оптимизации динамических многоуровневых компиляторов с учетом информации о профиле выполнения программы. Метод был реализован в динамическом компиляторе языка JavaScript V8, разработанном компанией Google. Использование профиля выполнения программы позволяет оптимизировать программу для конкретных входных данных. Это особенно актуально в связи с использованием JavaScript в операционных системах. Сценарий использования оптимизации на основе профиля программы в операционных системах следующий: на этапе тестирования программного обеспечения можно организовать сбор информации о профиле программы и использовать его для оптимизации приложений под конкретные случаи выполнения. Одним из новых применений использования информации о профиле программы может быть обеспечение немедленного переключения выполнения часто исполняющихся участков кода на уровень оптимизирующего компилятора. Другое применение – удаление обратных переходов на неоптимизирующие уровни выполнения.

Ключевые слова: JavaScript, V8, оптимизация программ, динамическая компиляция

DOI: 10.15514/ISPRAS-2016-28(1)-1

Для цитирования: Варданян В.Г. Методы оптимизации программ на языке JavaScript, основанные на статистике выполнения программы. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 5-20. DOI: 10.15514/ISPRAS-2016-28(1)-1

1. Введение

В настоящее время широкое распространение получили программы на нетипизированных сценарных языках. Одним из повсеместно используемых языков является JavaScript. С использованием JavaScript написаны многие крупные многофункциональные приложения, такие как Gmail, Google docs и другие. JavaScript также используется в платформе Node.js [1] для разработки веб-приложений на стороне сервера. Более того, уже имеются разработки операционных систем для телефонов, планшетов и ноутбуков, которые подразумевают использование JavaScript как одного из основных языков для создания приложений. Примерами таких систем могут быть Tizen [2] и FirefoxOS [3]. Тем самым, все больше возрастают требования к производительности программ на языке JavaScript, а также к паузам при интерактивном взаимодействии. Многие современные реализации JavaScript используют технологию динамической компиляции (JIT-компиляция), что позволяет применять широкий класс оптимизаций и за счет этого достичь лучшей производительности. При динамической компиляции время, затраченное на компиляцию, добавляется к общему времени выполнения. Поэтому важно соблюдать баланс между сложностью выполняемых оптимизаций и временем задержки запуска программы.

Чтобы достичь такого баланса, используется технология многоуровневой JIT-компиляции. Такое решение обеспечивает быстрый запуск программы, начиная выполнение на неоптимизирующих уровнях компиляции. Далее, наиболее часто исполняющиеся участки кода выполняются на оптимизирующих уровнях компиляции для генерации более качественного машинного кода.

Целью данной работы является разработка и реализация методов оптимизации многоуровневых динамических компиляторов, основанных на профиле выполнения.

Дальнейшее изложение построено следующим образом. В разд. 2 приводится описание архитектуры компилятора V8 и примененные технологии (замена на стеке, спекулятивная компиляция и т.д.) для построения оптимизирующих многоуровневых JIT-компиляторов. В разд. 3 дается обзор работ в предметной области. В разд. 4 описывается схема функционирования предлагаемого решения. В разд. 5 приведены основные результаты.

2. Архитектура V8

Для генерации машинного кода в V8 [4] используются два разных компилятора (рис. 1). Единицей компиляции является функция (метод). Первыми этапами работы V8 являются лексический и синтаксический анализ.

Исходный код разбивается на лексемы, методом рекурсивного спуска строится синтаксическое дерево. После этого начинает работать компилятор первого уровня Full-Codegen. На первом уровне функция переводится в машинный код с выполнением минимального набора оптимизаций, что позволяет быстрее приступить к выполнению кода. При генерации машинного кода для каждой инструкции учитываются все возможные случаи выполнения для данной операции. На этом уровне собирается профиль программы – информация о типах полей объектов. Кроме того, базовый компилятор V8 расставляет счетчики для определения часто выполняемых участков кода (функций и циклов). Когда такой участок кода обнаруживается, компиляция переходит на второй, оптимизирующий уровень – Crankshaft. На этом уровне из абстрактного синтаксического дерева строится граф потока управления в SSA-представлении – Hydrogen. Это внутреннее представление позволяет выполнить ряд машинно-независимых оптимизаций, таких как встраивание функций, удаление мертвого кода, удаление общих подвыражений, оптимизации циклов и т.д.

Crankshaft использует собранную в предыдущем уровне информацию о типах для эффективного хранения целочисленных переменных и операций над ними. На 64-битных архитектурах для целых чисел используется следующее кодирование: старшие 32 бита хранят число, младшие биты нулевые. На 32-битных архитектурах для хранения целых чисел используется 31 бита, младший бит нулевой. Целые числа в таком представлении называются Smi (от английского small integer). В JavaScript отсутствует строгая типизация. Это значит, что в общем случае переменные ссылаются на объекты. Все указатели на объекты в V8 имеют младший бит (определяющий четность числа) установленным в единицу, что позволяет отличить Smi от других объектов. Кодирование использует тот факт, что все адреса выровнены, т.е. объектов с нечетными адресами не существует.

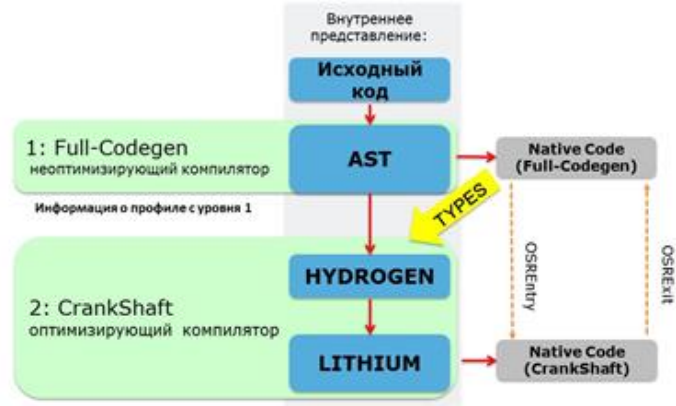

Рис. 1. Многоуровневая архитектура компилятора V8
Fig.1. Multilevel architecture of the V8 compiler

Crankshaft распространяет полученную из уровня Full-Codegen информацию о профиле по всему графу. Это позволяет генерировать машинный код спекулятивным образом, основываясь на предположении, что определенные свойства переменных (тип, значение и т.д.) останутся неизменными при следующих вызовах функции. В код вставляются необходимые проверки. Когда одна из таких проверок не выполняется, происходит переход на первый, неоптимизированный уровень. Этот процесс называется деоптимизацией. Для переключения между разными уровнями компиляции используется технология замены на стеке (OSR) [5]: выполнение функции приостанавливается и текущий стек функции заменяется новым. После выполнения “замены на стеке”, во всех местах вызова этой функции производится перенаправление на новую версию функции. При этом, переключение может произойти как с первого уровня на второй (OSR entry), так и наоборот (OSR exit).

После выполнения всех оптимизаций представление Hydrogen переводится в машинно-зависимое представление – Lithium. В отличие от представления Hydrogen, Lithium использует близкое к машинному коду трехадресное представление. Это представление используется для эффективной организации распределения регистров, а также для кодогенерации.

3. Обзор работ

Существуют два подхода для сбора информации о профиле программы – динамический и статический. Статический метод основан на алгоритмическом анализе и часто используется во время компиляции программы для эффективной реализации той или иной оптимизации. Например, оценка приблизительного количества выполнения функций или циклов позволяет более эффективно реализовать распределение регистров.

При использовании динамического метода, информация собирается во время выполнения программы. По сравнению с статическом профилированием, динамический метод позволяет собирать более точную информацию. Существует два способа получения статистики во время исполнения программы: инструментировать ее, - то есть вставлять счетчики в код при генерации или еще в промежуточное представление программы, - либо собирать выборочным профилировщиком аппаратных прерываний. В первом случае код увеличивается в размерах и сильно замедляется работа программы [6]. Во-втором же случае влияние на время выполнения гораздо меньше.

Профилирование поддерживается во многих индустриальных компиляторах. Например, в компиляторной инфраструктуре LLVM [7] реализованы три метода профилирования: на уровне функций, базовых блоков или ребер в промежуточном представлении. Все методы могут быть использованы независимо друг от друга. Компилятор GCC [8] также поддерживает несколько разных методов профилирования.

Много новых работ посвящены к улучшению производительности программ, написанных на языках с динамическими типами. В работах [9] [10] [11] описывается методы предварительной оптимизации программ на языке JavaScript. В работе [12] описываются методы компиляции программ с динамическими типами в статическое внутренне представление LLVM. Данный метод позволяет применить реализованные в LLVM оптимизации к программам, написанным на языке JavaScript. В работах [13] [14] [15] [16] [17] приведены разные методы оптимизации динамических многоуровневых компиляторов.

В данной работе описываются особенности оптимизации многоуровневых динамических компиляторов с использованием информации о статистке выполнения программы на примере компилятора V8.

4. Особенности использования профиля выполнения программы в многоуровневых динамических компиляторах

В целях улучшения производительности многоуровневых JIT-компиляторов важно обеспечить, чтобы часто выполняемые участки программы как можно больше времени выполнялась на оптимизирующих уровнях компиляции. Для обоснования этого утверждения приведем анализ сравнения времени выполнения разных уровней компилятора V8 на нескольких тестовых наборах языка JavaScript (рис 2а и 2б).

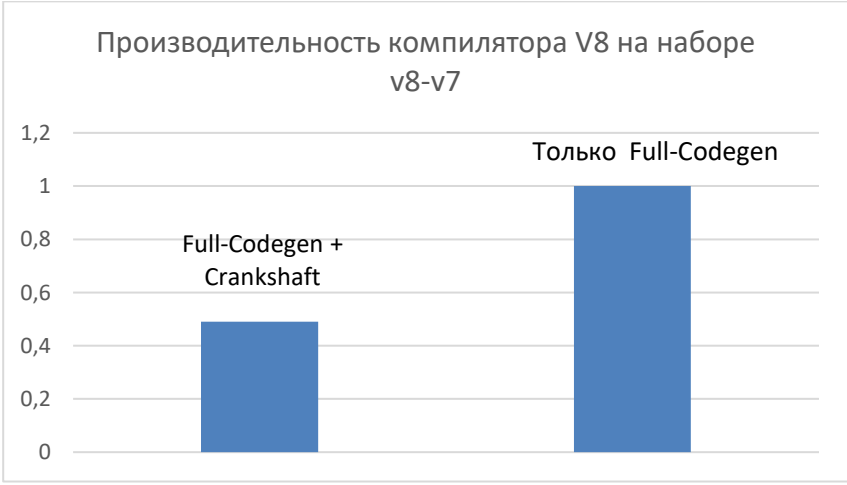

Рис 2а. Сравнение производительности уровней V8 на наборе v8-v7. Единицей времени выступает время выполнения компилятора Full-Codegen
Fig 2a. Comparing performance of the V8 levels on the benchmark v8-v7. The unit of time is duration of run-time of the Full-Codegen compiler

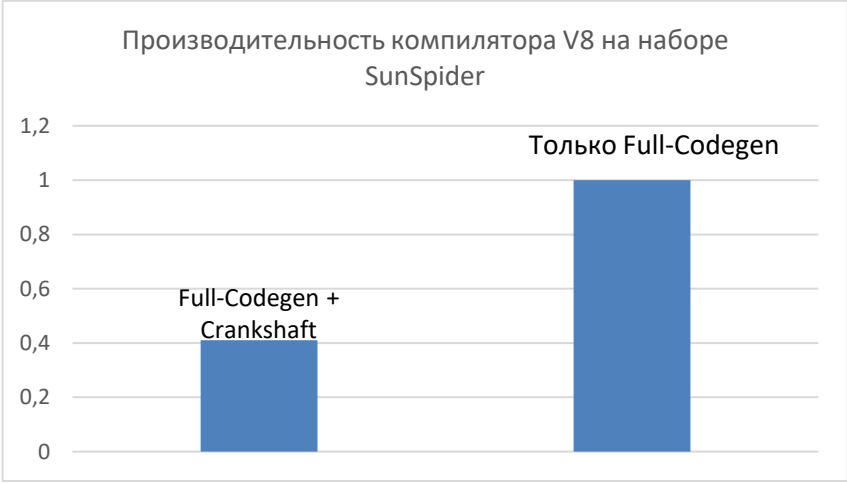

Рис 2б. Сравнение производительности уровней V8 на наборе SunSpider. Единицей времени выступает время выполнения компилятора Full-Codegen.
Fig 2b. Comparing performance of the V8 levels on the benchmark of SunSpider. The unit of time is duration of run-time of the Full-Codegen compiler

Из анализа сравнения видно, что производительность оптимизирующих уровней компиляции в среднем 2-2.5 больше, нежели производительность неоптимизирующих уровней. Поэтому, в целях улучшения производительности компилятора важно добиться, чтобы горячие участки кода как можно больше времени выполнялись именно на оптимизирующих уровнях компиляции. Одним способом достижения этой цели является как можно быстрое переключение выполнения таких участков на оптимизирующие уровни компиляции.

Другим способом является устранение обратных переходов на неоптимизирующие уровни выполнения и последующих перекомпиляций функций.

4.1 Использование профиля выполнения программы для обеспечения более быстрого перехода на оптимизирующие уровни компиляции

Более быстрого перехода на оптимизирующие уровни выполнения можно достичь путем внедрения в компилятор механизмов, позволяющих собирать и сохранять информацию о горячих участках кода. Многоуровневая архитектура компилятора V8 уже содержит механизмы для нахождения часто выполняемых участков кода, следовательно, сбор информации не добавляет никаких дополнительных накладных расходов. В компилятор V8 был добавлен модуль для сохранения этой информации. Далее, при последующих запусках программы сохраненная информация о горячих участках кода

позволяет сразу перевести выполнение таких участков на оптимизирующие уровни компиляции. При этом необходимо учитывать тот факт, что на первых уровнях выполнения собирается необходимая для спекулятивной компиляции информация о типах и объектах программы, что ограничивает немедленный переход на оптимизирующие уровни выполнения.

Была реализована новая эвристика для переключения выполнения горячих функций на оптимизирующие уровни компиляции с учетом сохраненной информации. Переход осуществляется, как только будет собран необходимый процент информации о типах (25% по умолчанию) для функции. Этот метод особенно эффективен для программ с небольшим временем выполнения. Для таких программ часто характерна следующая ситуация: функция помечается как кандидат для выполнения на оптимизирующем уровне. Компиляция этой функции на оптимизирующем уровне производится параллельно с выполнением программы на нижних уровнях, и программа заканчивается раньше, чем начинается выполнение оптимизированной версии функции. Реализованный метод позволяет переключать выполнение на оптимизирующий уровень намного раньше, что приводит к значительному росту производительности для программ с небольшим временем выполнения.

4.2 Использование профиля программы для устранения обратных переходов на неоптимизирующие уровни выполнения.

Из-за динамических свойств языка JavaScript в реальных приложениях деоптимизации могут встречается часто. В компиляторе V8 при выполнении деоптимизации для функции собирается новый профиль для организации реоптимизации. Для исключения больших временных затрат на постоянную реоптимизацию кода, при множественных деоптимизаций оптимизация для данной функции запрещается. В компилятор V8 был добавлен модуль для сохранения информации о количестве и причинах деоптимизаций. При последующих запусках программы эта информация используется следующим образом:

- Если при первом запуске программы для какой-либо функции оптимизация была запрещена из-за множественных деоптимизаций, при последующих запусках не будут предприниматься попытки для ее оптимизации.
- Если при первом запуске программы в какой-либо функции происходит деоптимизация из-за недостаточной информации о типах, переключение выполнения этой функции на оптимизирующие уровни откладывается, что позволяет собрать более полную информацию о типах (50% по умолчанию).

4.3 Использование профиля программы для выбора оптимального набора оптимизаций.

На разных приложениях эффект конкретных оптимизаций на сгенерированный машинный код может быть незначительным. При динамической компиляции такие оптимизации могут стать причиной ухудшения производительности для конкретных приложений. Приведем анализ эффективности реализованных в компиляторе V8 оптимизаций на тестовом наборе v8-v7 и на реальном примере использования сайта Gmail. Более подробный анализ и тестирования на других приложениях (Facebook, Wordpress) можно найти в [18]. Для имитации реального использования сайта Gmail была использована платформа Sikuli UI [19], которая позволяет автоматизировать использование графического интерфейса. Сценарий использования сайта Gmail следующий: программа авторизуется в Gmail используя данные тестового пользователя. Затем загружаются все входящие сообщения и производится поиск нескольких сообщений. В конце программа выходит из системы. Запуск тестового набора v8-v7 также был произведен из браузера с помощью Sikuli UI.Для оценки влияния каждой оптимизации на производительность в целом были проведены тестирования с выключением около одиннадцати оптимизаций: нумерация глобальных значений, вынос инвариантного кода за цикл, анализ диапазонов, удаление лишних Ф-функций, встраивание вызовов функций, встраивание полиморфных вызовов функций и с последовательным включением каждой из этих оптимизаций. Результаты тестирований на наборе v8-v7 и сайте Gmail приведены на Рис. 3а и 3б.

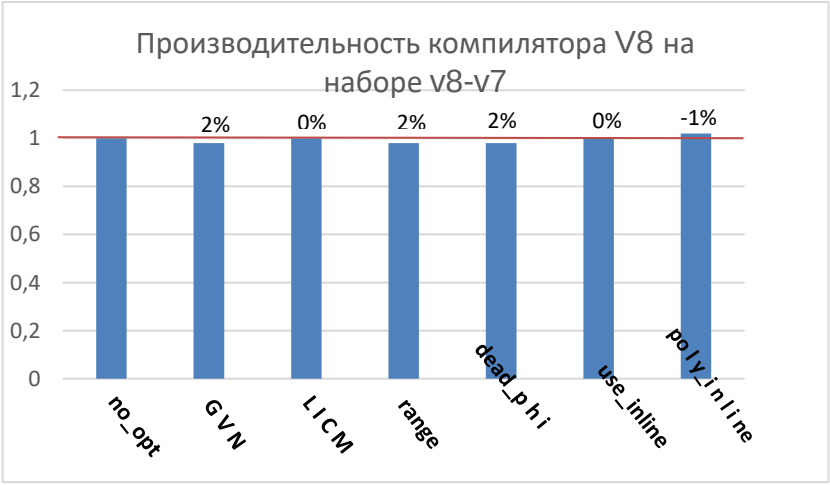

Рис 3а. Относительное время выполнения V8 на наборе v8-v7. Единицей времени выступает время выполнения компилятора с выключенными оптимизациями
Fig 3a. Relative duration of the V8 run-time on the benchmark v8-v7. The unit of time is duration of the compiler run-time with turned-off optimizations

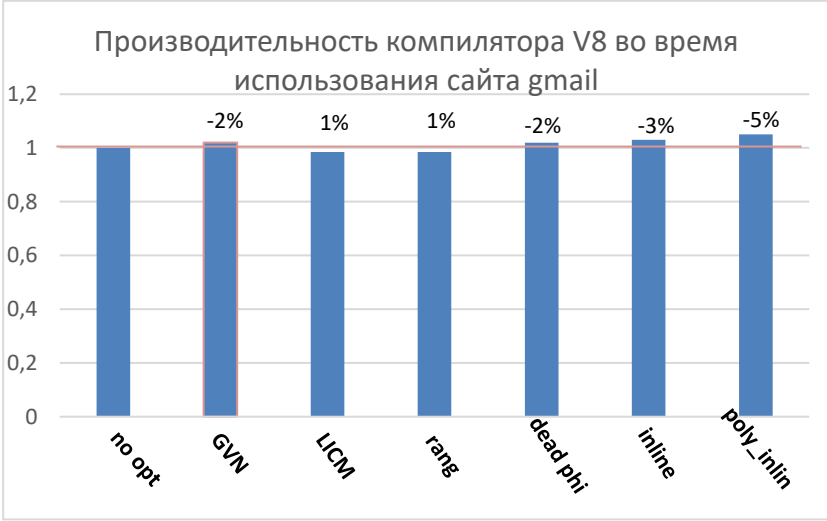

Рис 3б. Относительное время выполнения V8 при использовании сайта gmail. Единицей времени выступает время выполнения компилятора с выключенными оптимизациями
Fig 3b. Relative duration of the V8 run-time with the gmail site. The unit of time is duration of the compiler run-time with turned-off optimizations

Как видно из результатов, на наборе v8-v7 большинство оптимизаций имеет положительный эффект на время выполнения, однако оптимизация “встраивание полиморфных вызовов” в целом ухудшает производительность на 1%. На реальном примере использования сайта Gmail, наоборот, большинство протестированных оптимизаций имеет негативный эффект на производительность. Оптимизация глобальной нумерации значений для удаления общих подвыражений ухудшает производительность на 2%, а встраивание полиморфных вызовов на 5%.

Сохраненная информация о эффективности каждой оптимизации позволяет выбрать оптимальный набор оптимизаций для каждого приложения. В компиляторе V8 были добавлены соответствующие счетчики для оценки эффективности каждой оптимизации (вычисляется количество удаленных/измененных оптимизацией инструкций, а также количество итераций циклов функций). Кроме того, был реализован модуль для сохранения собранной информации. Была добавлена поддержка для следующих оптимизаций:

- Нумерация глобальных значений, для удаления общих подвыражений
- Вынос инвариантного кода за цикл
- Удаление мертвого кода
- Анализ диапазонов

- Удаление избыточных проверок
- Удаление избыточных Ф-функций
- Удаление избыточных обращений к памяти (load/store elimination)
- Escape анализ, для оптимизации хранения объектов в куче

Для оценки необходимости использования той или иной оптимизации была реализована метрика, которая зависит от нескольких параметров. Эффективность оптимизации вычисляется следующей формулой:

$$\mathbf{E = \sum_i (FC * LCi),\ i = 0, 1, \dots n}$$

Где n количество удаленных (измененных) оптимизацией инструкций, FC — количество выполнения функции, LCi — количество итераций цикла, в котором находилась удаленная (измененная) инструкция. Вычисляется также примерное количество шагов, необходимых для выполнения каждой из этих оптимизаций (OptSteps). Если для данной оптимизации E < OptSteps, она отключается при последующих запусках программы. Например, оптимизация “удаление мертвого кода”, выполняется за два прохода по всем инструкциям графа управления. При первом проходе отмечаются все живые переменные. При этом для каждой инструкции проверяются несколько условий. При первом проходе выполняются приблизительно 2*m шагов, где m количество инструкций в графе. Во время второго прохода непомеченные инструкции удаляются (не больше чем за m шагов). Сама оптимизация вызывается два раза для каждой функции. Значение OptSteps для этой оптимизации равно m*2*(2+1) = 6*m, где m количество инструкций в графе потока управления. Такая оценка была реализована для каждой из рассматриваемых оптимизаций.

Важно также отметить, что выполнение некоторых оптимизаций может повлиять на эффективность других. Например, анализ диапазонов используется при удалении избыточных проверок переполнения. В таких случаях, для оценки эффективности также учитывается возможный эффект на другие оптимизации.

Количество выполнений циклов и функций используется также для выбора методов распределения регистров. Так для функций с коротким временем выполнения используется обыкновенный алгоритм линейного сканирования [20], а для больших функций с тяжелыми циклами используется жадный алгоритм линейного сканирования [21].

5. Результаты

Использование информации о часто выполняемых участках кода и количестве деоптимизаций не добавляет дополнительных накладных расходов при сборе информации, так как инструментарии уже были реализованы в компиляторе V8. Внедрение инструментарий для вычисления количества итераций циклов

и эффективности оптимизаций замедляет набор SunSpider всего на 5%. Набор Octane замедляется на 23%, а Kraken на 25%.

Использование собранной информации позволяет достичь существенного роста производительности при последующих запусках программы. Организация более быстрого перехода на оптимизирующие уровни выполнения, а также реализация метода выбора оптимального набора оптимизаций для конкретных приложений позволило ускорить множество тестов из наборов SunSpider и Kraken. В среднем тестовый набор SunSpider стал выполняться на 11% быстрее, ускорение конкретных тестов составляет 50% (Таблица 1). Тестовый набор Kraken в среднем ускорился на 4%. На наборе Octane тесты *deltablue* и *crypto* ускорились на 7%, тест *richards* на 5% а *gameboy* на 3%. Пример использования сайта gmail ускорился на 1%.

Таблица 1. Производительности набора SunSpider с использованием оптимизаций на основе профиля программы (меньше - лучше)
Table 1. Performance of the SunSpider benchmark using optimizations based on the profile of the program (less – better)

Тест	Производительность V8, мс	Производительность V8 с реализованными улучшениями, мс	Улучшение, %
3d-cube	48.8	46	5.73
3d-morph	51.7	52.4	-
3d-raytrace	52.4	53	-1.1
access-binary-tree	7.1	7.1	-
access-fannkuch	20.7	20.7	-
access-nbody	13.5	13.1	3
access-nsieve	8.8	7.2	18
bitops-3bit-in-byte	4.0	3.6	10
bitops-bits-in-byte	11.8	11.2	5
bitops-bitwise-and	8.1	8.1	-
bitops-nsieve-bits	11.9	12	-
control-flow-recursive	6.0	6.0	-
crypto-aes	26.2	23.6	9.9
crypto-md5	17.8	17.5	1.7

crypto-sha1	17.8	19.0	-6.7
date-format-tofte	48.7	49.3	-1.2
date-format-xparb	70.1	71.0	-1.2
math-cordic	13.4	13.1	2.2
math-partial-sum	33.0	32.7	-
math-spectral-norm	11	7.2	34
regexp-dna	32.2	32.2	-
string-base64	27.0	26.8	-
string-fasta	70.7	35.4	50
string-tagcloud	101.0	101.0	-
string-unpack-code	91.4	93.1	-1.8
string-validate-input	40.6	30.1	25.8
Суммарное время	680.3	602.2	11.4

6. Заключение

В рамках данной работы был реализован метод оптимизации многоуровневого JIT-компилятора V8, основанный на статистике выполнения программы. При этом по возможности были использованы реализованные в компиляторе инструментарии, что позволило минимизировать накладные расходы при сборе необходимой информации. Собранная статистика выполнения программы позволяет значительно улучшить производительность компилятора V8.

Список литературы

[1]. Страница платфомы Node.js: nodejs.org.
[2]. Страница платформы Tizen: tizen.org.
[3]. Страница платофомы FirefoxOS: www.mozilla.org.
[4]. Страница компилятора V8: https://developers.google.com/v8/.
[5]. S. J. Fink и F. Qian, «Design, Implementation, and Evaluation of Adaptive Recompilation with on-stack replacement» *Proceedings of the IEEE,* pp. 241-252, 2003.
[6]. T. Ball и J. R. Larus, «Thomas Ball and James R. Larus. Optimally profiling and tracing programs» *In POPL* '92: Proceedings of the 19th ACM SIGPLAN-SIGACT

symposium on Principles, pp. 59-70, 1992.
[7]. Страница инфраструктуры LLVM: LLVM.org.
[8]. Страница компилятора GCC: https://gcc.gnu.org/.
[9]. R. Zhuykov, V. Vardanyan, D. Melnik, R. Buchatskiy и E. Sharygin, «Augmenting JavaScript JIT with Ahead-of-Time Compilation,» In Proceedings of IEEE,Computer Science and Information Technologies, pp. 116-120, 2015.
[10]. R. Zhuykov, V. Vardanyan, D. Melnik, R. Buchatskiy и E. Sharygin, «Augmenting JavaScript JIT with Ahead-of-Time Compilation,» 10th International Conference on Computer Science and Information Technologies, pp. 236-240, 2015.
[11]. S. Jeon и J. Choi, «Reuse of JIT compiled code based on binary code patching in JavaScript engine,» J. Web Eng, pp. 337-349, 2012.
[12]. В. Варданян, В. Иванишин, С. Асрян, А. Хачатрян и Д. Акопян, «Динамическая компиляция программ на языке JavaScript в статически типизированное внутреннее представление LLVM». Труды Института системного программирования РАН, том 27 (выпуск 6), 2015 г., стр. 33-48. DOI: 10.15514/ISPRAS-2015-27(6)-3
[13]. V. Vardanyan, «Optimizations of JavaScript programs,» GSPI's scientific journal 2014, pp. 122-128.
[14]. Р. Жуйков, Д. Мельник, Р. Бучацкий, В. Варданян, В. Иванишин и 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
[15]. S.-W. Lee и S.-M. Moon, «Selective just-in-time compilation for client-side mobile javascript engine,» Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems, pp. 5-14, 2011.
[16]. S.-W. Lee, S.-M. Moon, W.-K. Jung, J.-S. Oh и H.-S. Oh, «Code size and performance optimization for mobile JavaScript just-in-time compiler,» Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture, 2010.
[17]. V. Vardanyan, S. Asryan и R. Buchatskiy, «Integrated register rematerialization in JavaScript V8 JIT compiler,» 10th International Conference on Computer Science and Information Technologies, pp. 240-244, 2015.
[18]. Анализ эффективности оптимизаций в компиляторе V8: http://www.cs.cmu.edu/~ishafer/compilers/.
[19]. Страница платформы Sikuli UI: http://www.sikuli.org/.
[20]. M. Poletto и V. Sarkar, «Linear scan register allocaton» ACM Transactions on Programming Languages and Systems, pp. 895-913 , 1999.
[21]. Описаные жадного алгоритма линейного сканирования: http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html.

Profile-based optimizations for JavaScript programs

V. Vardanyan <vaag@ispras.ru>,
Yerevan State University
Alex Manoogian, 1, 0025, Yerevan, Republic of Armenia

Abstract. In recent years, JavaScript has become one of the most popular programming languages on the web. Many massive applications are written using JavaScript, such as Gmail, Google docs, etc. JavaScript is also used in the Node.js — server-side web application developing platform. Moreover, JavaScript is the main language for developing applications on some operating systems for mobile and media devices. Examples of such systems are Tizen and FirefoxOS. Due to increasing popularity of JavaScript many big companies produced and continue to develop their own dynamic compilers for this language. JIT compilation makes it possible to implement many well-known classic optimizations to improve programs performance. To maintain a trade-off between quick startup and doing sophisticated optimizations, JavaScript engines usually use multiple tiers for compiling hot functions: lower tier JITs generate less efficient code, but can start almost immediately (e.g., even with interpretation), while higher tier JITs aim at generating very effective code for hot places, but at the cost of long compilation time. So even highly optimized JavaScript execution engines require some time to "warm-up" before reaching their peak performance. This work is dedicated to the performance improvement of modern dynamic multitier JIT compilers by designing and implementing profile-based optimizations in JavaScript V8 JIT compiler.

Keywords: JavaScript, V8, program optimization, dynamic compilation.

DOI: 10.15514/ISPRAS-2016-28(1)-1

For citation: V. Vardanyan. Profile-based optimizations for JavaScript programs. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 1, 2016, pp. 5-20 (in Russian). DOI: 10.15514/ISPRAS-2016-28(1)-1

References

[1]. Node.js website: nodejs.org.
[2]. Tizen website: tizen.org.
[3]. FirefoxOS website: www.mozilla.org.
[4]. V8 compiler website: https://developers.google.com/v8/.
[5]. S. J. Fink и F. Qian, «Design, Implementation, and Evaluation of Adaptive Recompilation with on-stack replacement» Proceedings of the IEEE, pp. 241-252, 2003.
[6]. T. Ball и J. R. Larus, «Thomas Ball and James R. Larus. Optimally profiling and tracing programs» In POPL ’92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles, pp. 59-70, 1992.
[7]. LLVM website: http://LLVM.org.
[8]. GCC website: https://gcc.gnu.org/
[9]. R. Zhuykov, V. Vardanyan, D. Melnik, R. Buchatskiy и E. Sharygin, «Augmenting JavaScript JIT with Ahead-of-Time Compilation,» In Proceedings of IEEE,Computer Science and Information Technologies, pp. 116-120, 2015.
[10]. R. Zhuykov, V. Vardanyan, D. Melnik, R. Buchatskiy и E. Sharygin, «Augmenting JavaScript JIT with Ahead-of-Time Compilation,» 10th International Conference on

Computer Science and Information Technologies, pp. 236-240, 2015.
[11]. S. Jeon и J. Choi, «Reuse of JIT compiled code based on binary code patching in JavaScript engine,» J. Web Eng, pp. 337-349, 2012.
[12]. V. Vardanyan, V. Ivanishin, S. Asryan, A. Khachatryan, J. Hakobyan, «Dinamicheskaja kompiljacija programm na jazyke JavaScript v staticheski tipizirovannoe vnutrennee predstavlenie LLVM» («Dynamic Compilation of JavaScript Programs to the Statically Typed LLVM Intermediate Representation»). Trudy ISP RAN [Proceedings of ISP RAS], Volume 27 (Issue 6), 2015. pp. 33-48 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-3
[13]. V. Vardanyan, «Optimizations of JavaScript programs,» GSPI's scientific journal 2014, pp. 122-128.
[14]. R. Zhuykov, D. Melnik, R. Buchatskiy, V. Vardanyan, V. Ivanishin и E. Sharygin, «Metody dinamicheskoj i predvaritel'"noj optimizacii programm na jazyke JavaScript» («Dynamic and ahead of time optimization for JavaScript programs») Trudy ISP RAN [Proceedings of ISP RAS], Volume 26, (Issue 1), 2014. pp. 297-314 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-10
[15]. S.-W. Lee и S.-M. Moon, «Selective just-in-time compilation for client-side mobile javascript engine,» Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems, pp. 5-14, 2011.
[16]. S.-W. Lee, S.-M. Moon, W.-K. Jung, J.-S. Oh и H.-S. Oh, «Code size and performance optimization for mobile JavaScript just-in-time compiler,» Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture, 2010.
[17]. V. Vardanyan, S. Asryan и R. Buchatskiy, «Integrated register rematerialization in JavaScript V8 JIT compiler,» 10th International Conference on Computer Science and Information Technologies, pp. 240-244, 2015.
[18]. Quantifying Optimization Efficacy in V8 JIT compiler: http://www.cs.cmu.edu/~ishafer/compilers/
[19]. Sikuli UI website: http://www.sikuli.org/.
[20]. M. Poletto и V. Sarkar, «Linear scan register allocaton» ACM Transactions on Programming Languages and Systems, pp. 895-913 , 1999.
[21]. Description of greedy linear scan algorithm: http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html