Динамическая компиляция программ на языке JavaScript в статически типизированное внутреннее представление LLVM

[1]*В.Г. Варданян <vaag@ispras.ru>*
[1]*В.А. Иванишин <vlad@ispras.ru>*
[2]*С.А. Асрян <seryozha.asryan@ysumail.am>*
[2]*А.А. Хачатрян <aramayis.khachatryan@ysumail.am>*
[2]*Дж.А. Акопян <dzhivan.hakobyan1@ysumail.am>*
[1] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2]*Ереванский государственный университет,*
0025, Армения, г. Ереван, ул. А. Манукяна, дом 1

Аннотация. В статье предлагаются методы, делающие возможной компиляцию программ на языке JavaScript в статически типизированное представление LLVM. В работе рассматривается многоуровневый динамический компилятор языка JavaScript V8, разработанный компанией Google. Основная цель работы – улучшение производительности программ на языке JavaScript. Для этого предлагается способ добавления в компилятор V8 нового уровня оптимизации, который использует инфраструктуру LLVM для генерации машинного кода. Это позволяет применять имеющиеся в LLVM оптимизации и технологии генерации машинного кода для разных архитектур к программам, написанным на JavaScript.

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

DOI: 10.15514/ISPRAS-2015-27(6)-3

Для цитирования: Варданян В.Г., Иванишин В.А., Асрян С.А.,. Хачатрян А.А, Акопян Дж.А. Динамическая компиляция программ на языке JavaScript в статически типизированное внутреннее представление LLVM. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 33-48. DOI: 10.15514/ISPRAS-2015-27(6)-3.

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. Для обеспечения быстрого выполнения скриптов необходимо создание максимально качественного машинного кода, а также снижение затрат на все этапы оптимизации, выполняемые при компиляции программ. Для достижения этой цели многие современные JIT-компиляторы поддерживают разные уровни компиляции горячих участков кода. Целью данной работы является улучшение производительности программ на языке JavaScript методом добавления нового уровня оптимизации с использованием инфраструктуры LLVM [4] в компиляторе V8.

Дальнейшее изложение построено следующим образом. В главе 2 описана архитектура компилятора V8 и примененные технологии (замена на стеке, параллельное выполнение и т.д.) для построения многоуровневых JIT-компиляторов. В главе 3 дается обзор работ в предметной области. В главе 4 описывается схема функционирования предлагаемого решения, обеспечивающего использование биткода LLVM в качестве новой платформы для JIT-компиляции. В главе 5 приведены основные результаты.

2. Архитектура V8

Для генерации машинного кода V8 использует два разных компилятора[1] (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 горячих участков кода (функций и циклов). Когда такой участок кода обнаруживается, компиляция переходит на второй,

[1] На самом деле, разработчиками компилятора V8 активно разрабатывается и другой уровень оптимизации, который использует промежуточное представление “sea of nodes”, но в данной работе этот уровень не представляет интереса.

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.

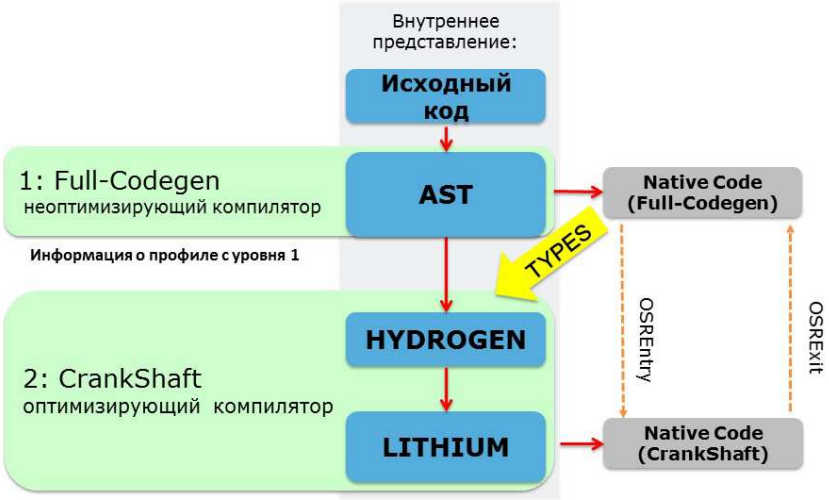

Рис. 1. Многоуровневая архитектура компилятора V8

Согласно спецификации EcmaScript [5], все числа в JavaScript являются вещественными числами двойной точности, удовлетворяющими стандарту IEEE 754. Однако Crankshaft может оптимизировать генерируемый машинный код (используя целочисленность переменных для более оптимального хранения и операций над ними) при помощи собранной на нижнем уровне информации о типах. На 64-битных архитектурах для целых чисел используется следующее кодирование: старшие 32 бита хранят число, младшие биты нулевые. 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.

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. Для этого в код вставляются необходимые проверки. Когда одна из таких проверок не выполняется, происходит переход на первый, неоптимизированный уровень. Для переключения между разными уровнями компиляции используется технология замены на стеке. При этом, как описано выше, переключение может произойти как с первого уровня на второй (OSR entry), так и наоборот (OSR exit, deoptimization).

Помимо технологии замены на стеке (OSR), для некоторых виртуальных машин [6], предусматривающих JIT-компиляцию, характерна еще одна особенность. В ряде случаев возникает необходимость как можно скорее остановить потоки выполнения скомпилированного кода[2]. Наиболее ярким и общим из таких случаев является остановка по причине начала работы сборщика мусора. Многие сборщики мусора не предусматривают параллельной работы вместе с выполнением программы, т.к. это приводило бы к конкуренции потоков за доступ к одним и тем же объектам и порождало бы недетерминированные паузы, кроме того, сборщик мусора может переносить объекты, что приводит к тому, что сам код, ссылающийся на них, должен быть изменен. Помимо приостановки выполнения, необходимо также гарантировать, что все объекты во время паузы будут находиться в целостном, согласованном состоянии.

Для достижения этой цели компилятор V8 использует технологию safepoints. Термин *safepoint* означает согласованное (и известное) состояние программы, а также точку в программе, в которой такое состояние достигается. Для того, чтобы исполняемый код мог быстро достичь safepoint, они, как правило, расставляются на обратных ребрах циклов и местах вызовов функций. В этих точках выставляются проверки, в случае срабатывания которых выполнение остановится в текущем состоянии.

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

С ростом популярности языка JavaScript, в течение последних лет несколько крупных корпораций, таких как Google, Mozilla и Apple, выпустили свои JIT-компиляторы для языка JavaScript. Разработанный компанией Apple компилятор под названием JavaScriptCore [7] также имеет многоуровневую

[2] Для JavaScript, согласно стандарту языка, может быть только один такой поток.

структуру (Рис. 2.). В отличие от V8, в JavaScriptCore реализованы четыре уровня компиляции. На первом уровне работает интерпретатор LLINT. Второй уровень представляет из себя простой JIT-компилятор Baseline JIT. На этом уровне для функций генерируется машинный код с минимальными оптимизациями аналогично уровню Full-Codegen в компиляторе V8. Baseline JIT создает для каждой операции байткода соответствующий машинный код. В этом коде реализуются все возможные случаи для данной операции. Baseline JIT, как и Full-Codegen, используется в качестве базовой версии 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 (OSR exit) и переход к коду Baseline JIT. Следующий уровень компилятора JSC представляет собой спекулятивный компилятор DFG.

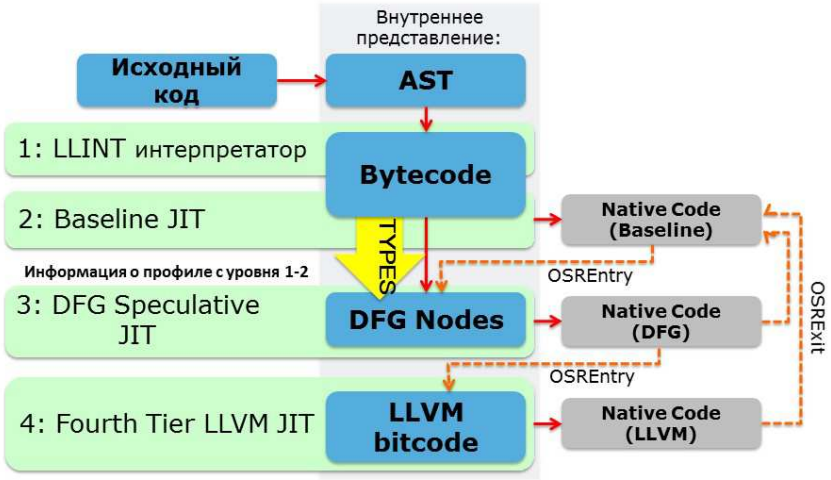

Рис. 2. Многоуровневая архитектура компилятора JavaScriptCore

На этом уровне из промежуточного представления байт-кода строится граф потока управления программы в виде SSA-формы. Это представление, аналогично уровню Crankshaft, делает возможным реализацию нескольких стандартных машинно-независимых оптимизаций, например, удаление общих подвыражений, удаление мертвого кода и т.д. 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 [8]. В мае 2014 года был добавлен новый, четвертый уровень компиляции – FTL JIT, который использует инфраструктуру LLVM для генерации машинного кода. На момент начала данной работы FTL JIT поддерживался только для платформ Mac OS X и iOS, в данной же работе реализация выполняется в первую очередь для операционной системы GNU/Linux и процессоров x86-64. Стоит также отметить, что, несмотря на внешние сходства общей архитектуры, эти два компилятора отличаются по внутренним структурам и реализациям. В данной работе рассматриваются основные подходы, примененные нами при реализации нового уровня оптимизации с использованием инфраструктуры LLVM в динамическом компиляторе V8 языка JavaScript. Освещаются особенности архитектуры V8, существенные с точки зрения реализации нового уровня. Отметим также, что V8 существенно более популярен: JavaScriptCore используется только в веб браузере Safari, тогда как V8 встроен в браузеры Chrome, Opera, Android-браузер, Node.js и в операционную систему ChromeOS.

4. Компиляция программ на языке JavaScript в статически типизированное внутреннее представление LLVM

Промежуточное представление оптимизирующего уровня Hydrogen содержит всю необходимую информацию для построения статически типизированного представления LLVM. Данный метод позволяет применять оптимизации, имеющиеся в статическом компиляторе LLVM, к программам, написанным на языке JavaScript. Инфраструктура LLVM предоставляет средства для динамической компиляции в виде модуля MCJIT. В этом модуле уже задействованы все имеющиеся механизмы LLVM для кодогенерации и машинно-зависимых оптимизаций под различные платформы. Компиляция языка JavaScript в промежуточное представление LLVM была реализована путем добавления дополнительного уровня в компилятор V8 (Рис. 3.).

Несмотря на то, что и Hydrogen, и LLVM-биткод используют форму SSA, есть определенные различия в интерпретации определения SSA-формы. Отличие состоит в том, что LLVM требует, чтобы базовые блоки входов Ф-функций являлись непосредственными предшественниками данного базового блока. На Рис. 4.а изображен граф потока управления, корректный с точки зрения Hydrogen и некорректный с точки зрения LLVM. На Рис. 4.б приведен тот же граф, скорректированный под требования LLVM. Первым этапом при добавлении нового уровня в V8 была разработка и реализация алгоритма для преобразования промежуточного представления Hydrogen в корректную с точки зрения LLVM SSA-форму. При этом, чтобы использовать код, сгенерированный LLVM, в той же манере, что и код, сгенерированный самим компилятором V8, необходимо, чтобы они были совместимы на бинарном уровне. В первую очередь, это означает, что в LLVM должна быть реализована генерация кода в соответствии с соглашением о вызовах,

принятом в V8 (V8 использует собственное соглашение о вызовах). Кроме того, стоит отметить, что для генерации машинного кода для некоторых инструкций необходимо вызывать вспомогательные функции. Эти вызовы, в свою очередь, реализуют собственные соглашения о вызовах. Для достижения совместимости сгенерированных кодов в компиляторе LLVM были реализованы все необходимые соглашения о вызовах.

Одним из важных компонентов для обеспечения быстродействия современных динамических компиляторов является поддержка спекулятивной компиляции.

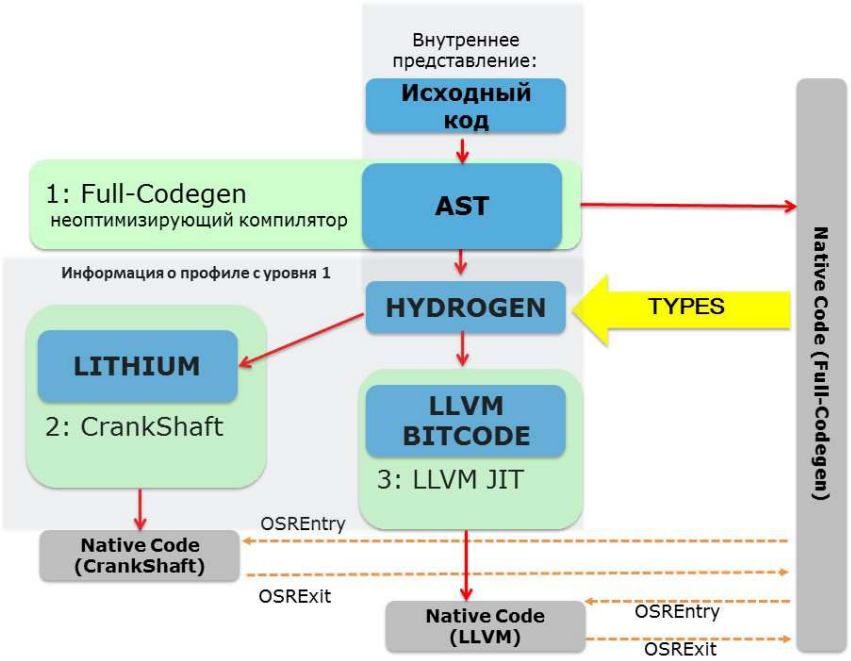

Рис. 3. Архитектура многоуровневого компилятора V8 с добавленным модулем LLVM

Эта технология используется, например, для частичной компиляции кода, что позволяет экономить используемую память, а также уменьшить общее время выполнения программы. Помимо этого, к спекулятивной компиляции можно отнести предсказание типов и технологию “полиморфный встроенный кэш вызовов” (англ. polymorphic inline cache) [9], применяемую для повышения производительности кода. Сложность реализации такой поддержки на уровне LLVM состоит в том, что, перекладывая генерацию кода на сторонний компилятор, мы теряем над ней контроль, что приводит к трудностям при реализации спекулятивной компиляции, а также при поддержке автоматической сборки мусора. Между тем, инфраструктура LLVM

предоставляет инструменты для контроля и модификации сгенерированного компонентом MCJIT кода. Эти инструменты были добавлены в LLVM в рамках реализации уровня FTL в компиляторе JSC. В итоге поддержка деоптимизаций (OSR exit) в новом уровне компилятора V8 была реализована с помощью встроенных в LLVM функций llvm.experimental.stackmap и llvm.experimental.patchpoint, которые позволяют модифицировать сгенерированный LLVM код на лету.

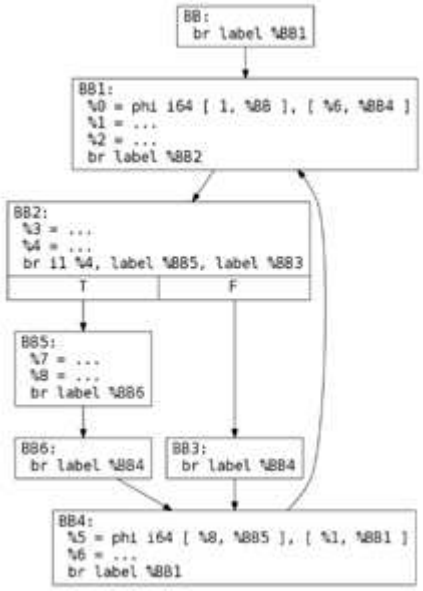

Рис 4.a Граф потока управления, некорректный с точки зрения LLVM

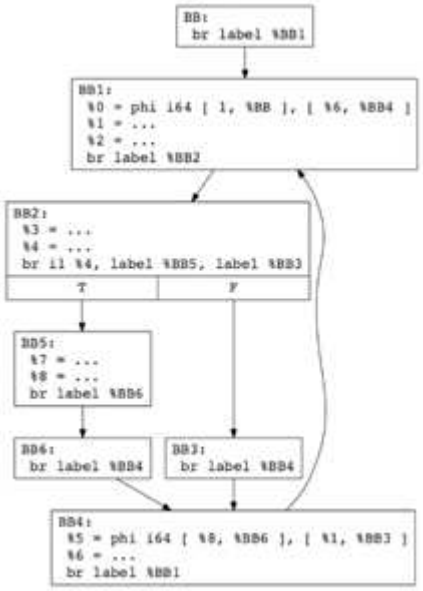

Рис 4.b Граф потока управления, корректный с точки зрения LLVM

Обе эти функции во время компиляции представления LLVM в машинный код инициируют создание специальной секции данных, содержащей структуру Stack Map [10]. В этой структуре сохраняется относительное смещение от начала функции в машинном коде, куда попадает вызов stackmap/patchpoint, а также местонахождение (слот стека, имя регистра, константа и т.п.) всех значений, переданных этим функциям в качестве параметров. Для поддержки деоптимизаций в месте, где происходит передача управления виртуальной машине, вставляется вызов stackmap. В качестве аргументов передаются значения, необходимые для продолжения выполнения на уровне Full-Codegen. Вычисление этих значений происходит с помощью уже имеющегося в Crankshaft механизма симуляции состояния стековой машины (которую реализует Full-Codegen).

Отдельное внимание следует уделить реализации механизма safepoint (см. главу 2). По сути, реализация safepoint во многом аналогична реализации

деоптимизации с помощью llvm.experimental.stackmap. Однако определение значений, информация о местоположении которых после генерации кода для нас важна, отличается. Для точки safepoint это все живые в этой точке значения, являющиеся указателями на объекты в динамической памяти. Когда сборщик мусора V8 совершает обход указателей, он учитывает объекты, ассоциированные с safepoint, на котором приостановлено выполнение, как живые. Без правильно сформированных сущностей safepoint, занимаемая этими объектами память могла бы быть освобождена, что привело бы к некорректному поведению. Для получения информации об интервалах жизни переменных (LLVM-значений), накрывающих определенную точку, были задействованы анализирующие проходы, уже реализованные в LLVM [11]. Кроме того, соответствующие преобразующие проходы LLVM были использованы для автоматической расстановки safepoints. А для генерации Stack Map для точек safepoint применена функция llvm.experimental.gc.statepoint [12].

Функции llvm.experimental.stackmap/patchpoint также используются для поддержки возможности перемещения объектов сборщиком мусора V8. Сгенерированный код может содержать абсолютные адреса объектов. При перемещении объектов необходимо заменить их старые адреса на новые. Кроме того, сгенерированный машинный код находится в куче и интерпретируется компилятором V8 как обыкновенный объект. Следовательно, сборщик мусора может передвигать сгенерированный код в процессе своей работы. Этот перемещаемый код вызывается по смещению из генерируемого LLVM кода. Для реализации такой поддержки используется функция llvm.experimental.patchpoint, которая помимо тех же параметров, что и llvm.experimental.stackmap, принимает также вызываемую функцию. Помимо создания Stack Map, при компиляции llvm.experimental.patchpoint в генерируемый код вставляется вызов этой функции в соответствии с заданным соглашением о вызовах. Благодаря Stack Map, позже сам вызываемый объект можно будет подменить.

Другим важным аспектом при проектировании многоуровневых JIT-компиляторов является поддержка замены исполняемого кода на оптимизированный код для горячих функций. В простом случае, когда функция имеет короткое время выполнения, переход осуществляется путем замены ее адреса на адрес оптимизированного кода во время следующих вызовов функции. (Это возможно, так как сгенерированные коды реализуют одинаковый бинарный интерфейс.) Однако в случае, если функция содержит цикл или несколько циклов с большим количеством итераций, появляется необходимость в организации перехода на оптимизирующий уровень во время выполнения функции. Такой переход между уровнями Full-Codegen и Crankshaft осуществляется путем простого безусловного перехода на начало соответствующего цикла в оптимизированном коде (Рис. 5). Так как управление передается не на начало функции, при переходе на новый код

также выполняются определенные действия для приспособления стека. Например, необходимо уменьшить значение указателя стека на количество локальных переменных, которые будут храниться в стеке в оптимизированном коде. При этом необходимо учитывать, что прежний уровень, в свою очередь, уже уменьшил это значение на количество своих локальных переменных. Основная проблема такого подхода при переходе на LLVM код заключается в том, что LLVM подразумевает, что каждая функция может иметь только одну точку входа.

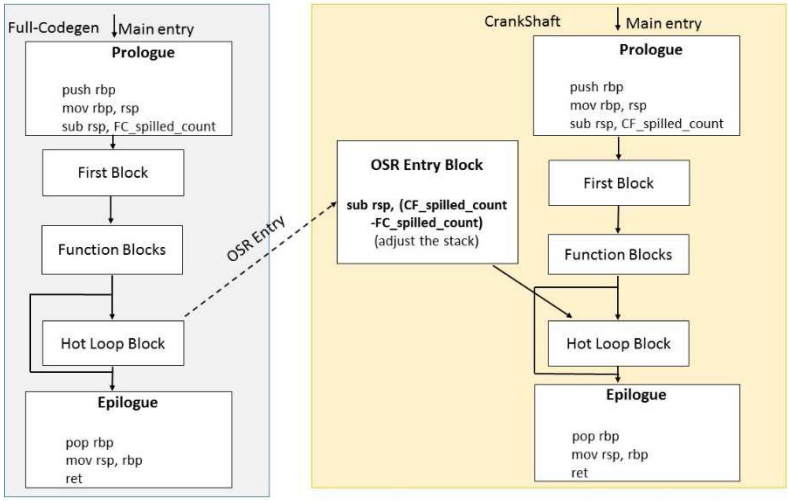

Рис. 5. Схема перехода между компиляторами Full-Codegen и CrankShaft

Нарушение этого инварианта может привести к сильному ограничению выполняемых в LLVM оптимизаций или даже к неправильной работе программы. Одно решение этой проблемы, использующееся в компиляторе JSC – это создание копий функции для каждого возможного входа. При этом все аргументы и локальные переменные сохраняются в глобальном буфере, а переход на уровень LLVM осуществляется путем вызова соответствующей функции. Явным недостатком такого подхода является более длительное время компиляции из-за создания копий функции. Нами был разработан и реализован другой метод, который позволяет осуществить переход во время выполнения на LLVM-код без создания дополнительных копий функции. При реализации такого подхода основная проблема состоит в приспособлении стека, так как при использовании стороннего компилятора LLVM нет возможности выяснить, какое количество локальных переменных будет храниться на стеке (эта информация становится доступна гораздо позже, при распределении регистров LLVM). Для решения проблемы при переходе на

оптимизирующий код во время выполнения цикла точка входа была перемещена в пролог кода функции, где значение указателя стека уменьшается на количество локальных переменных, хранящихся в стеке. После этого с помощью дополнительного аргумента решается, откуда именно будет выполняться функция.

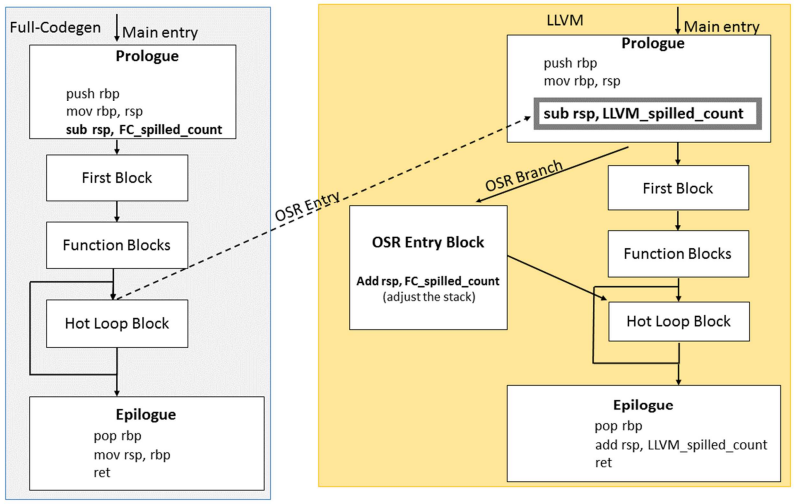

Рис. 6. Схема перехода между компиляторами Full-Codegen и LLVM

Если управление перешло на LLVM код из цикла, то необходимо выполнить дополнительное приспособление стека. В частности, надо учитывать тот факт, что прежний уровень мог, в свою очередь, уменьшить значение указателя стека (Рис. 6.). При таком подходе LLVM функция также может содержать две точки входа, но, в отличие от уровня Crankshaft, эти точки находятся в прологе функции и не являются препятствием для реализации оптимизаций в LLVM. Однако при реализации такого подхода появляются определенные проблемы при передаче локальных значений из уровня Full-Codegen уровню LLVM. Эти значения хранятся в начале стека, начиная с адреса, записанного в специальный регистр rbp (stack base pointer), и загружаются в регистры после соответствующей проверки для нахождения точки входа функции (Рис. 6.). Однако модуль распределения регистров LLVM не подразумевает наличия зарезервированных слотов стека. Таким образом, LLVM, в свою очередь, может использовать адреса начиная с регистра rbp для распределения своих локальных переменных. В итоге может получиться, что локальные переменные Full-Codegen могут быть переписаны компилятором LLVM. Одним решением этой проблемы может быть внесение изменений в модуль распределения регистров LLVM для резервирования адресов в стеке. Однако

подобные глобальные изменения в коде LLVM не универсальны и осложняют применение обновлений к кодовой базе из основного репозитория. По этой причине был избран другой подход. Желаемый эффект был достигнут посредством добавления фиктивных volatile значений. Так, при генерации биткода LLVM с помощью volatile значений были зарезервированы соответствующие слоты стека для локальных значений Full-Codegen.

Еще одним важным аспектом при компиляции программ на языке JavaScript является снижение пауз при интерактивном взаимодействии. Для достижения этой цели в многоуровневых компиляторах реализована поддержка параллельного выполнения функций. Так, например, код, сгенерированный уровнем Full-Codegen, будет выполняться параллельно, пока на уровне Crankshaft выполняются оптимизации и кодогенерация. Поддержка параллелизма на уровне LLVM полностью осуществлена и в нашем решении.

После осуществления поддержки всех технологий, обеспечивающих быстродействие современных многоуровневых JIT-компиляторов (спекулятивная компиляция, технология “замена на стеке”, параллельное выполнение и т. д.) следующим шагом была трансляция представления Hydrogen в промежуточное представление LLVM. При реализации преобразования Hydrogen в биткод LLVM был применен инкрементальный подход: количество доступных операций наращивалось постепенно, делая возможным компиляцию все более широких подмножеств языка JavaScript. В представлении Hydrogen насчитывается около 120 различных операций, на данный момент реализована трансляция более 70 вершин в промежуточное представление LLVM.

5. Результаты

На данный момент компиляция в LLVM биткод поддерживается не для всех структур языка JavaScript. Это накладывает ограничение на полное тестирование бенчмарков языка JavaScript. По этой причине в данном разделе приведены результаты тестирования на нескольких тестах из набора SunSpider. Оценка производительности для этих тестов была проведена для разного количества итераций, что позволяет оценивать реальную пользу компиляции “очень горячих” участков кода с помощью инфраструктуры LLVM.

Табл. 1. Сравнение производительности на тесте access-nsieve

access-nsieve			
Число итераций	ориг. кол-во итераций	*10	*100
Время выполнения с исп. Crankshaft, мс	59	590	2980
Время выполнения с исп. LLVM, мс	63	580	2465
Ускорение, число раз	0.94	1.02	1.21

Табл. 2. Сравнение производительности на тесте bitops-bits-in-byte

bitops-bits-in-byte			
Число итераций	ориг. кол-во итераций	*10	*100
Время выполнения с исп. Crankshaft, мс	37	216	2050
Время выполнения с исп. LLVM, мс	27	74	538
Ускорение, число раз	1.37	2.92	3.81

Табл. 3. Сравнение производительности на тесте 3d-cube

3d-cube			
Число итераций	ориг. кол-во итераций	*5	*10
Время выполнения с исп. Crankshaft, мс	100	391	842
Время выполнения с исп. LLVM, мс	85	177	325
Ускорение, число раз	1.18	2.21	2.59

Как видно из представленных выше таблиц, при увеличении числа итераций (времени выполнения машинного кода) увеличивается и выигрыш в производительности, обеспечиваемый более оптимальным машинным кодом уровня LLV8 в сравнении с Crankshaft. Для ответа на вопрос, чем обоснована более высокая производительность уровня LLV8, нами был проведен сравнительный анализ машинных кодов. Анализ теста bitops-bits-in-byte, например, показывает, что ускорение достигается в первую очередь благодаря наличию оптимизации “разворачивание цикла” (англ. loop unrolling) в списке оптимизационных проходов, выполняемых над внутренним представлением LLVM (в Crankshaft такая оптимизация отсутствует). Код, сгенерированный LLV8 для данного теста, не содержит цикла, тогда как код, сгенерированный Crankshaft, имеет цикл. Отметим также, что, время компиляции на уровне LLVM почти всегда больше чем на уровне Crankshaft. Например, для теста bitop-bits-in-byte, время, затраченное на компиляцию, составило примерно 0,2 мс в случае Crankshaft и 6,5 мс в случае LLVM. Однако даже такое увеличение времени компиляции приемлемо для функций, время выполнения которых существенно превосходит время компиляции. (Кроме того, при компиляции в параллельном потоке это имеет еще меньшее значение.) Именно для оптимизации таких функций изначально было принято решение реализации уровня LLVM в компиляторе V8.

6. Заключение и будущая работа

В рамках данной работы был разработан метод динамической компиляции программ на языке JavaScript в статически типизированное представление LLVM. Предложенный метод был реализован путем добавления дополнительного уровня оптимизации в компиляторе с открытым исходным кодом V8. При этом была реализована поддержка всех технологий,

обеспечивающих быстродействие современных многоуровневых JIT-компиляторов (спекулятивная компиляция, технология “замена на стеке”, параллельное выполнение и т. д.). В дальнейшем планируется добавление поддержки компиляции в LLVM биткод для всех конструкций языка JavaScript и тестирование системы на популярных тестовых наборах.

Список литературы

[1]. Страница платформы Node.js — https://nodejs.org
[2]. Страница платформы Tizen — http://www.tizen.org
[3]. Веб-сайт Mozilla — https://www.mozilla.org
[4]. Страница платформы LLVM – http://www.llvm.org/
[5]. Описание стандарта ECMA-262 http://www.ecma-international.org/publications/standards/Ecma-262.html
[6]. Страница документации динамического компилятора HotSpot для языка Java – http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html
[7]. Веб-сайт Webkit – http://www.webkit.org
[8]. 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
[9]. U. Hölzle, C. Chambers, D. Ungar “Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches” ECOOP '91 Proceedings of the European Conference on Object-Oriented Programming, 21-38, 1991
[10]. Страница документации структуры StackMaps – http://llvm.org/docs/StackMaps.html
[11]. Страница обзора изменений LLVM в ревизии rL229945 – http://reviews.llvm.org/rL229945
[12]. Страница документации структуры StatePoints – http://llvm.org/docs/Statepoints.html

Dynamic Compilation of JavaScript Programs to the Statically Typed LLVM Intermediate Representation

[1]V. Vardanyan <vaag@ispras.ru>
[1]V. Ivanishin <vlad@ispras.ru>
[2]S. Asryan <seryozha.asryan@ysumail.am>
[2]A. Khachatryan <aramayis.khachatryan@ysumail.am>
[2]J. Hakobyan <dzhivan.hakobyan1@ysumail.am>
[1]I Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[2] Yerevan State University, 1 Alex Manoogian Str., Yerevan, 0025, Republic of Armenia

Abstract. Since its inception in the middle of the 90's, JavaScript has become one of the most popular web development languages. Although initially developed as a browser-agnostic scripting language, in recent years JavaScript continues its evolution beyond the desktop to areas such as mobile and server-side web applications. 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. Modern JavaScript engines use just-in-time (JIT) compilation to produce binary code. JIT compilers are limited in complexity of optimizations they can perform at runtime without delaying the execution. To maintain a trade-off between quick startup and doing sophisticated optimizations, JavaScript engines usually use multiple tiers for compiling hot functions. Our work is dedicated to performance improvement of JavaScript programs by adding a new optimizing level to the JavaScript V8 compiler. This level uses the LLVM infrastructure to optimize JavaScript functions and generate machine code. The main challenge of adding a new optimizing level is to support all the technologies (speculative compilation, on-stack replacement, concurrent compilation etc.) that are used in the modern multi-tier JIT compilers for increasing the performance and minimizing pauses during the interactive communication. All these technologies are fully supported in our solution. This has resulted in significant performance gains on the JavaScript benchmark suites when compiling hot functions.

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

DOI: 10.15514/ISPRAS-2015-27(6)-3

For citation: Vardanyan V., Ivanishin V., Asryan S., Khachatryan A., Hakobyan J. Dynamic Compilation of JavaScript Programs to the Statically Typed LLVM Intermediate Representation. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 33-48 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-3

References

[1]. Node.js website – https://nodejs.org
[2]. Tizen platform website – http://www.tizen.org
[3]. Mozilla website – https://www.mozilla.org
[4]. LLVM website – http://www.llvm.org/
[5]. ECMA-262 Standard http://www.ecma-international.org/publications/standards/Ecma-262.htm
[6]. HotSpot dynamic compiler documentation website – http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html
[7]. Webkit website – http://www.webkit.org
[8]. 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], Volume 26 (Issue 1). 2014, pp. 297-314. DOI: 10.15514/ISPRAS-2014-26(1)-10 (in Russian)
[9]. U. Hölzle, C. Chambers, D. Ungar "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches" ECOOP '91 Proceedings of the European Conference on Object-Oriented Programming, 21-38, 1991
[10]. StackMaps structure documentation webpage — http://llvm.org/docs/StackMaps.html

[11]. LLVM rL229945 revision review page — http://reviews.llvm.org/rL229945
[12]. StatePoints structure documentation webpage — http://llvm.org/docs/Statepoints.html