ЕРЕВАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Варданян Ваагн Геворгович

Методы статической оптимизации программ для языков с динамическими типами

Специальность 05.13.04 —
«Математическое и программное обеспечение вычислительных машин, комплексов, систем и сетей»

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель:
академик РАН, доктор физ.-мат.наук
Иванников В.П.

Ереван 2016

Содержание

Введение

Актуальность работы.

В настоящее время широкое распространение получили программы на динамических языках программирования. Традиционным способом выполнения программ на таких языках являлась интерпретация. Однако, с ростом применения этих языков и сложности приложений, все больше возрастают требования к производительности программ на языках с динамическими типами. Многие современные реализации этих языков используют технологию многоуровневой (гибридной) динамической компиляции (JIT-компиляция) [1], что позволяет применять широкий класс оптимизаций и тем самым достигать 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. Использование многоуровневого (гибридного) динамического компилятора позволяет достичь этого баланса. Такое решение обеспечивает быстрый запуск программы, начиная выполнение на уровне интерпретации. Далее, наиболее часто исполняющиеся участки кода выполняются на уровне динамического компилятора с применением разных оптимизаций, для генерации более качественного машинного кода. Использование такой архитектуры тем более актуально для современных многоядерных процессоров, позволяя запускать потоки компиляции параллельно с потоком выполнения на разных ядрах, и тем самым минимизировать паузы при интерактивном взаимодействии.

Многоуровневая архитектура также позволяет эффективно реализовывать спекулятивную компиляцию [2], что является одним из важных методов обеспечения быстродействия динамических языков программирования. Эта технология основывается на профиле выполнения программы и позволяет применять известные алгоритмы оптимизации статических языков к языкам с динамическими типами. Такие оптимизации выполняются (и являются корректными) только в предположении, что собранные при профилировании типы данных остаются неизменными. В случае нарушения этого условия выполнение возобновляется на предыдущем уровне компиляции. Такой процесс называется деоптимизацией. При использовании многоуровневой архитектуры на первом уровне выполнения собирается необходимый профиль программы, которая используется на следующих уровнях для реализации спекулятивных оптимизаций.

Производительность многоуровневых JIT-компиляторов может быть существенно увеличена за счет оптимизаций, применяемых на всех уровнях выполнения. Методы статической оптимизации [3] широко известны и используются во многих индустриальных компиляторах, в том числе и в современных многоуровневых (гибридных) компиляторах. Однако сложность

оптимизации многоуровневых компиляторов языков с динамическими типами состоит в том, что реализация оптимизации на одном уровне может привести к негативному эффекту на других уровнях и, в целом, негативно повлиять на производительность. Например, с одной стороны, в целях улучшения производительности следует обеспечивать выполнение максимального количества «горячих» (часто выполняемых) [4] участков кода на оптимизирующих уровнях компиляции. С другой стороны, важно обеспечить минимальное количество деоптимизаций (обратных переходов на неоптимизирующие уровни выполнения). Таким образом, при реализации оптимизаций на каждом уровне необходимо учитывать всю инфраструктуру многоуровневого динамического компилятора в целом.

JavaScript является одним из повсеместно используемых динамических языков. С использованием JavaScript написаны многие крупные многофункциональные приложения, такие как Gmail, Google docs и другие. В связи с ростом применения в последние годы многие крупные компании такие как Apple (компилятор JavaScriptCore [5]), Google (компилятор V8 [6]), Mozilla (компилятор SpiderMonkey [7]) и Microsoft (компилятор ChakraCore [8]), выпустили и активно развивают свои динамические компиляторы для языка JavaScript. Много новых работ посвящены разным методам оптимизации (предварительная компиляция [9], методы спекулятивного выполнения [10], поддержка параллелизма на уровне данных [11] и т.д.) программ с динамическими типами.

Приложения на языках с динамическими типами становятся все более комплексными и сложными. Необходимо постоянно улучшать инфраструктуру многоуровневых JIT-компиляторов, учитывая новые возможности процессоров и возрастающую сложность приложений, написанных на динамических языках. Большинство современных компиляторов динамических языков (в том числе вышеперечисленные) распространяются с открытым исходным кодом, что

позволяет разрабатывать новые оптимизации и адаптировать компиляторы под конкретные классы задач.

В настоящее время широкое распространение получили разработки операционных систем (Tizen OS, Firefox OS), которые используют динамические языки программирования для создания приложений. Это делает возможным реализацию известных методов оптимизации на основе профиля программы [12], для языков с динамическими типами. Можно организовать сбор информации о профиле программы на этапе тестирования программного обеспечения и использовать его в дальнейшем для оптимизации приложений под конкретные случаи исполнения. Более того, использование информации о профиле программы в среде многоуровневых динамических компиляторов дает возможность для разработки новых методов оптимизации. Например, можно использовать сохраненную информацию о часто исполняющихся участках кода для немедленного переключения выполнения этих участков на уровень оптимизирующего компилятора при последующих запусках программы.

В последние пару лет также активно развиваются платформы asm.js и webassembly, которые позволяют эффективно выполнять приложения, написанные на статических языках программирования в динамической среде. Для улучшения производительности этих приложений можно использовать методы, которые применяются для оптимизации программ, написанных на статических языках программирования. Технологии оптимизации таких программ тщательно разработаны и хорошо изучены. Компиляторная инфраструктура LLVM является одной из известных систем для анализа, трансформации и оптимизации программ, написанных на статических языках программирования. Инфраструктура также предоставляет средства для динамической компиляции, в которых уже задействованы все имеющиеся механизмы LLVM для машинно-независимой и машинно-зависимой оптимизации, а также для кодогенерации под различные платформы [13].

Целью диссертационной работы является определение границ применения методов статических оптимизаций к программам с динамическими типами, а также к программам со статическими характеристиками, работающим в динамической среде.

Для достижения поставленной цели были сформулированы и решены следующие задачи:

1. Оптимизация динамических многоуровневых компиляторов с использованием информации о профиле программы.
2. Применение оптимизаций компиляторной инфраструктуры LLVM к веб приложениям со статическими характеристиками.
3. Оценка степени применимости предложенных методов к конкретным компиляторам путем реализации описанных методов в динамических компиляторах JavaScriptCore и V8 и проведения экспериментов - как на тестовых наборах, так и на реальных примерах использования языка JavaScript.

Научная новизна. В работе получены следующие основные результаты, обладающие научной новизной:

- Разработан и реализован новый метод оптимизации динамических многоуровневых компиляторов, который использует информацию о профиле программы для организации немедленного переключения выполнения часто исполняющихся участков кода на уровень оптимизирующего компилятора.
- Разработан и реализован новый метод для динамической компиляции программ на языке JavaScript в статически типизированное внутреннее представление компиляторной инфраструктуры LLVM, позволяющий применять оптимизации LLVM к программам, написанным на языке JavaScript.
- Разработан и реализован эффективный алгоритм рематериализации регистров для динамических многоуровневых компиляторов.

- Разработан и реализован эффективный алгоритм удаления излишних вызовов функций без побочных эффектов для языка JavaScript.
- Экспериментальное подтверждение эффективности разработанных методов путем их реализации в динамических компиляторах JavaScriptCore и V8. Тестирование производительности на известных тестовых наборах SunSpider, Kraken и Octane показало, что использование оптимизаций на основе профиля программы улучшает производительность этих наборов на 11%, 4% и 2% соответственно. При этом улучшение производительности для отдельных тестов достигает 50%. Использование инфраструктуры LLVM в качестве дополнительного уровня выполнения, а также реализованные методы машинно-независимой и машинно-зависимой оптимизации позволяют улучшить те же тестовые наборы в среднем на 8-10%.

Практическая значимость. Все разработанные методы оптимизации используются в рамках совместного научно-исследовательского проекта корпорации Samsung и Института системного программирования РАН. Часть реализованных методов улучшения математических и строковых функций, поддержка новых инструкций в оптимизирующих уровнях компиляции, а также исправление разных недочётов были одобрены сообществом разработчиков компилятора JavaScriptCore и включены в состав этого компилятора.

Апробация работы и публикации. По теме диссертации опубликовано 8 работ в изданиях из перечня рецензируемых научных изданий ВАК [14] [15] [16] [17] [18] [19] [20] [21]. Основные результаты также представлены в докладах на следующих конференциях:

- Конференция, посвященная 80-летию Гюмриского государственного педагогического института, 2014 г. Гюмри, Армения

- 10-я международная конференция по вычислительным наукам и информационным технологиям “Computer Sciences and Information Technologies” (CSIT) 2015 г. Ереван, Армения.
- Открытая конференция по компиляторным технологиям 2015 г. Москва, Россия.

Структура и объем работы. Диссертация состоит из введения, 4 глав и заключения. Работа изложена на 107 страницах. Список источников насчитывает 69 наименований. Диссертация содержит 9 таблиц и 31 рисунок.

Глава 1. Обзор многоуровневых динамических компиляторов языка JavaScript

В настоящее время наиболее популярными компиляторами языка JavaScript являются V8, разработанный компанией Google, и JavaScriptCore, разработанный компанией Apple. Оба этих компилятора распространяются с открытым программным кодом и используют многоуровневую динамическую компиляцию для генерации машинного кода. Общий принцип многоуровневых оптимизирующих JIT-компиляторов – выборочная компиляция наиболее «горячих» участков кода, которая производится в несколько этапов. На каждом новом этапе применяются все более сложные оптимизации.

1.1 Архитектура динамического компилятора JavaScriptCore

Компилятор JavaScriptCore состоит из четырёх уровней компиляции (Рис. 1). Единицей компиляции является функция (метод). На первом этапе работы JavaScriptCore производится лексический [22] и синтаксический анализ [23]. Исходный код разбивается на лексемы, затем строится синтаксическое абстрактное дерево. На последнем этапе разбора из абстрактного дерево строится внутреннее представление (байткод). После построения байткода функция выполняется интерпретатором LLInt (Low Level Interpreter). В ранних версиях JavaScriptCore интерпретатор был реализован на языке C++ и использовал метод “прямого шитого кода” (англ. direct threaded code) [24] для выполнения программы. LLInt же запрограммирован на специальном мультиплатформенном ассемблере (offlineasm), который компилируется в машинный код (для X86, X86_64, ARM и нескольких других архитектур) во время сборки JavaScriptCore. Для компиляции offlineasm используется язык Ruby.

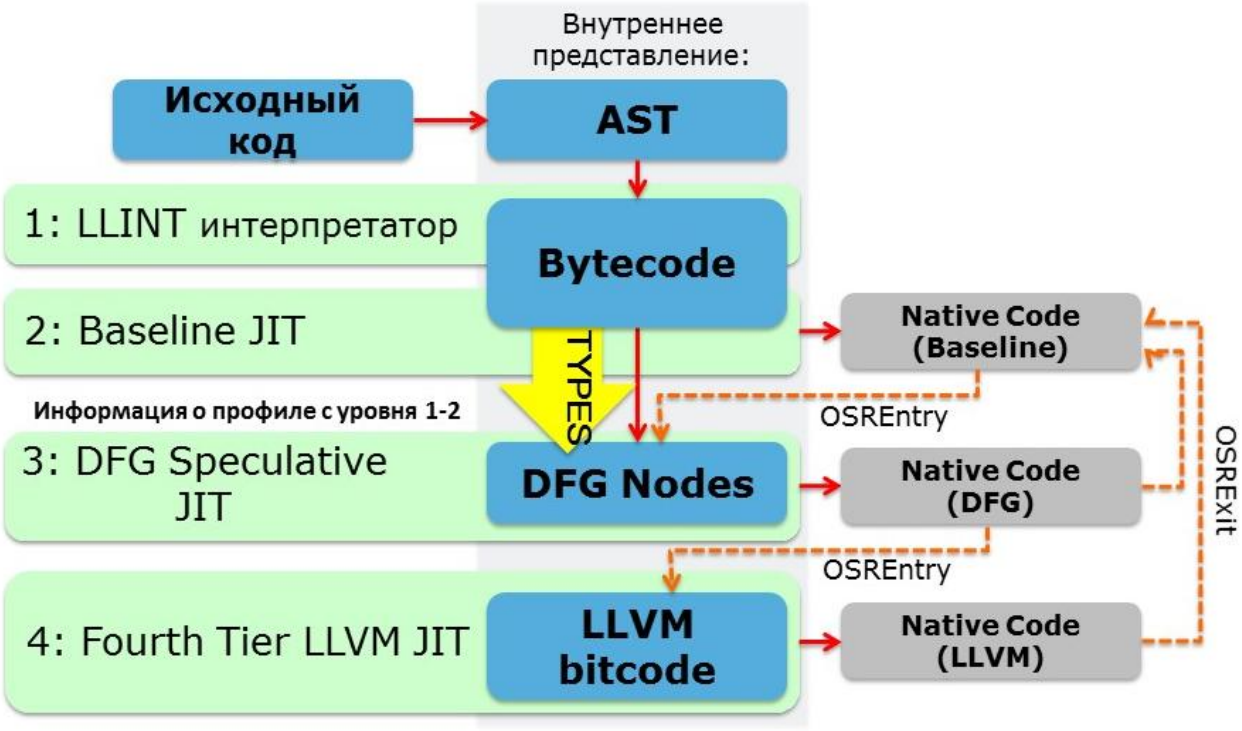


Рисунок 1. Архитектура многоуровневого компилятора JavaScriptCore

По сравнению с классическим интерпретатором, LLInt более компактен и занимает значительно меньше памяти. Более того, использование LLInt позволяет напрямую контролировать стек выполняемой функции, что особенно важно для организации переходов на другие уровни компиляции посредством технологии “замены на стеке” (on stack replacement – технология, позволяющая организовать переход на следующий уровень во время выполнения функции) [25]. При использовании метода “прямого шитого кода” доступен только стек самого интерпретатора. LLInt позволяет начать выполнение байткода без всяких подготовительных расходов, что обеспечивает быстрый запуск программы. Остальные уровни требуют предварительных затрат по созданию и сохранению машинного кода для функций. Интерпретатор LLInt также выполняет сбор информации о профиле функции (типы переменных, полей объектов).

Для принятия решения оптимизации функций проводится оценка частоты выполнения тех или иных участков кода. На всех уровнях компилятора JavaScriptCore оценка происходит с помощью детерминированного профилирования. При таком подходе инструментарий внедряется непосредственно в машинный код либо вызывается интерпретатором каждый раз, когда выполнение достигает определенных точек. В LLInt этими точками являются обратные ребра циклов и возвраты из функций. Для того, чтобы выполнение функции переключилось на следующий уровень оптимизации, необходимо, чтобы функция набрала не менее 100 “очков выполнения”, при этом каждая итерация цикла прибавляет одно “очко”, а за вызов функции - 15 “очков”. Для оценки также применяется дополнительная эвристика, которая зависит от размера функции. Таким образом, небольшая функция, которая была вызвана 7 раз является кандидатом для выполнения на уровне Baseline JIT. Согласно статистике, на уровне LLInt для часто выполняющихся участков кода примерно три четверти времени тратится на выполнение косвенных переходов для нахождения следующей байткод инструкции. Главная цель уровня Baseline

JIT – удаление накладных расходов, связанных с косвенными переходами. На уровне Baseline JIT машинный код функции генерируется с минимальным набором оптимизаций. При генерации машинного кода для каждой операции учитываются все возможные случаи выполнения. Например, операция сложения для строк должна быть выполнена как конкатенация, а для чисел как обычное сложение. Созданный машинный код будет содержатся множество ветвлений для обеспечения выполнения всех возможных вариантов для каждой операции. После генерации машинного кода для функции, необходимо организовать переход с предыдущего уровня на следующий и начать выполнение нового кода. В простом случае, если функция имеет небольшое время выполнения, переход осуществляется путем замены ее адреса на адрес оптимизированного кода во время следующих вызовов функции (это возможно, так как все уровни реализуют одинаковый бинарный интерфейс). Однако в случае, если функция содержит цикл или несколько циклов с большим количеством итераций, появляется необходимость в организации перехода на следующий уровень во время выполнения функции. Такой переход осуществляется путем технологии “замены на стеке” (англ. on-stack replacement entry, OSR entry): 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. На уровне Baseline JIT, так же, как и на LLInt, собирается и сохраняется профиль функции – информация о типах полей объектов и аргументах функций. Информация о профиле, собранная на уровнях Baseline JIT и LLInt, используется для организации спекулятивных оптимизаций на следующем уровне оптимизации.

На уровне BaseLine JIT также производится оптимизация обращений к данным и методам объектов при помощи технологии “полиморфный встроенный кэш вызовов” (англ. polymorphic inline cache) [26]. В динамических языках программирования обращение к данным объектов производится

значительно медленнее, чем в статических языках. Тип и данные объектов могут меняться во время выполнения, и при каждом обращении необходимо заново искать соответствующий метод. Для ускорения этого процесса применяются разные технологии. Самая простая из них — кэширование последнего значения пары “объект, метод”. Эту технологию можно улучшить, если принять во внимание, что по статистике тип самого объекта меняется редко. В этом случае, обращение к методу можно организовать с помощью прямого вызова кэшированной функции. Этот метод известен как “встроенный кэш вызовов”. Тип объекта может измениться при выполнении, поэтому в код необходимо добавить дополнительную проверку. Рис. 2 показывает организацию вызова методов до и после применения технологии встроенного кэша вызовов. Встроенный кэш вызовов работает эффективно, если тип объекта остается неизменным во время выполнения. Однако для полиморфных вызовов этот метод неэффективен и в некоторых случаях может привести к ухудшению производительности. Для оптимизации таких вызовов используется метод полиморфного встроенного кэша. Вместо того, чтобы кэшировать значение только последнего вызова, кэшируются все полиморфные вызовы по мере их поступления (число разных вызовов ограничивается во избежание генерации слишком большого кэша).

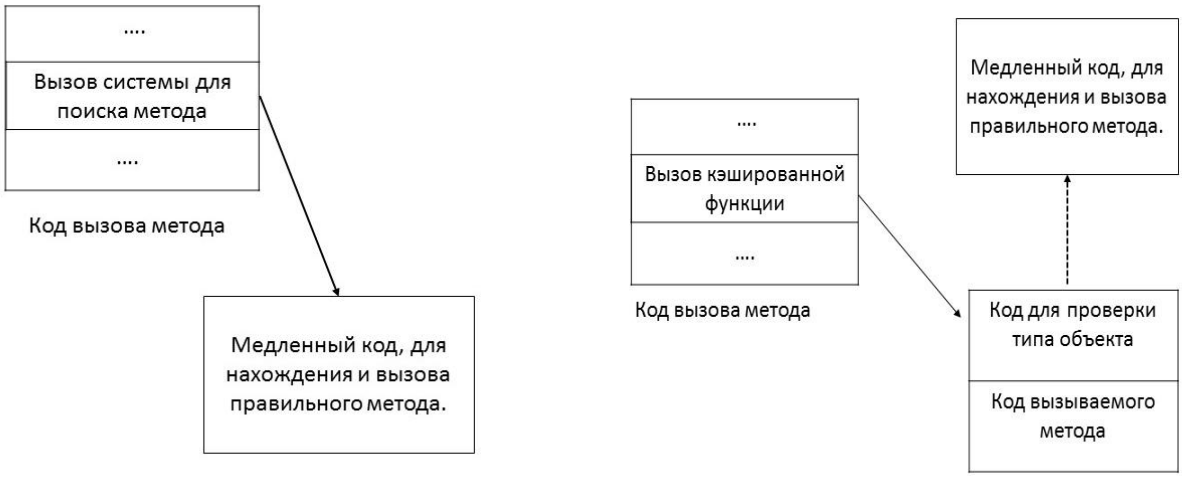


Рисунок 2. Вызов метода до и после использования встроенного кэша

Далее, при вызове метода управление передается специальному обработчику, который ответственен за вызов правильного метода. Проверку типа объекта в коде самого метода можно опустить, так как такая проверка уже была произведена обработчиком. На Рис. 3 приведен пример вызова методов для полиморфных объектов разного типа. Существуют разные методы улучшения технологии полиморфного встроенного кэша вызовов. Например, наиболее часто вызывающиеся функции передвигаются в начало списка для сокращения времени поиска. Функции с маленьким размером кода могут быть встроены непосредственно в код обработчика и т.д. Важным преимуществом использования встроенного кэша является то, что он также позволяет собирать информацию о типах и методах объектов. Эта информация используется на следующих (спекулятивных) уровнях компиляции для организации более эффективного обращения к методам и данным объектов.

Следующий уровень компилятора JSC представляет собой спекулятивный компилятор под названием DFG JIT. DFG JIT компиляция выполняется для функций, которые набрали не менее 1000 “очков выполнения”. На этом уровне из промежуточного представления байткода создается граф потока данных (англ. Data Flow Graph), в котором инструкции описаны в виде SSA-представления [27]. Это представление позволяет

выполнять ряд известных машинно-независимых оптимизаций [28], в том числе:

- встраивание функций,
- удаление мертвого кода,
- локальное/глобальное удаление общих подвыражений,
- распространение констант,
- снижение стоимости операций,
- удаление излишних проверок выхода за границу массива [29].

Кроме того, DFG JIT распространяет полученную информацию о типах переменных и полях объектов по всему графу, что позволяет выполнять спекулятивные (оптимистические) оптимизации. В первую очередь, спекулятивные оптимизации касаются предсказания типов переменных. Согласно спецификации EcmaScript [30], все числа в JavaScript являются вещественными числами двойной точности, удовлетворяющими стандарту IEEE [31]. Однако DFG использует предсказание типов для более эффективного хранения целочисленных переменных и операций над ними. Поскольку типы переменных могут изменяться динамически, в код вставляются необходимые проверки. Baseline JIT код используется как базовая версия кода для функций, которые скомпилированы на спекулятивных уровнях JIT-компилятора. Если оптимизированный код сталкивается со случаем, который в нем не поддерживается (например, тип или значение переменной изменились во время выполнения и более не соответствуют собранному профилю), то происходит переход к коду Baseline JIT. Переход происходит с помощью обратной замены на стеке (on stack replacement exit, OSR exit).

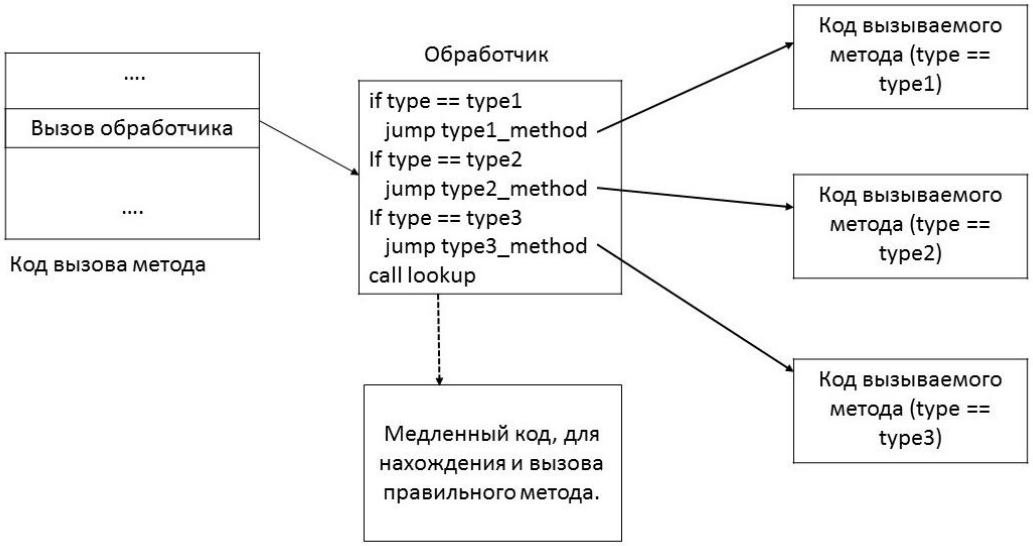


Рисунок 3. Схема работы полиморфного встроенного кэша

Этот процесс называется деоптимизацией. 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 многократных деоптимизаций используется эвристика, оценивающая необходимость дальнейших реоптимизации. Это позволяет исключить возникновение больших временных затрат на постоянную реоптимизацию кода и многократных OSR переходов.

В компиляторе JavaScriptCore используется сборщик мусора на основе поколений объектов (англ. generational garbage collection), который реализован с помощью алгоритма Джоела Бартлета [32].

Четвертый уровень оптимизации компилятора JavaScriptCore — FTL JIT, вызывается для функций, набравших не менее 10000 “очков выполнения”. Этот уровень использует инфраструктуру LLVM [13] для генерации машинного кода. Из представления DFG строится статическое внутреннее представление LLVM — LLVM биткод [33]. Это представление позволяет применять уже имеющиеся в LLVM оптимизации для программ, написанных на JavaScript. В настоящий момент уровень FTL JIT поддерживается только для операционных систем Mac OS X и iOS.

1.2 Архитектура динамического компилятора V8

Компилятор V8 состоит из двух уровней компиляции (Рис. 4). Первыми этапами работы V8 являются лексический и синтаксический анализ. Исходный код разбивается на лексемы, методом рекурсивного спуска строится абстрактное синтаксическое дерево (АСД). После этого начинает работать компилятор первого уровня Full-Codegen, который из АСД создает машинный код для целевой архитектуры. Единицей компиляции является функция (метод).

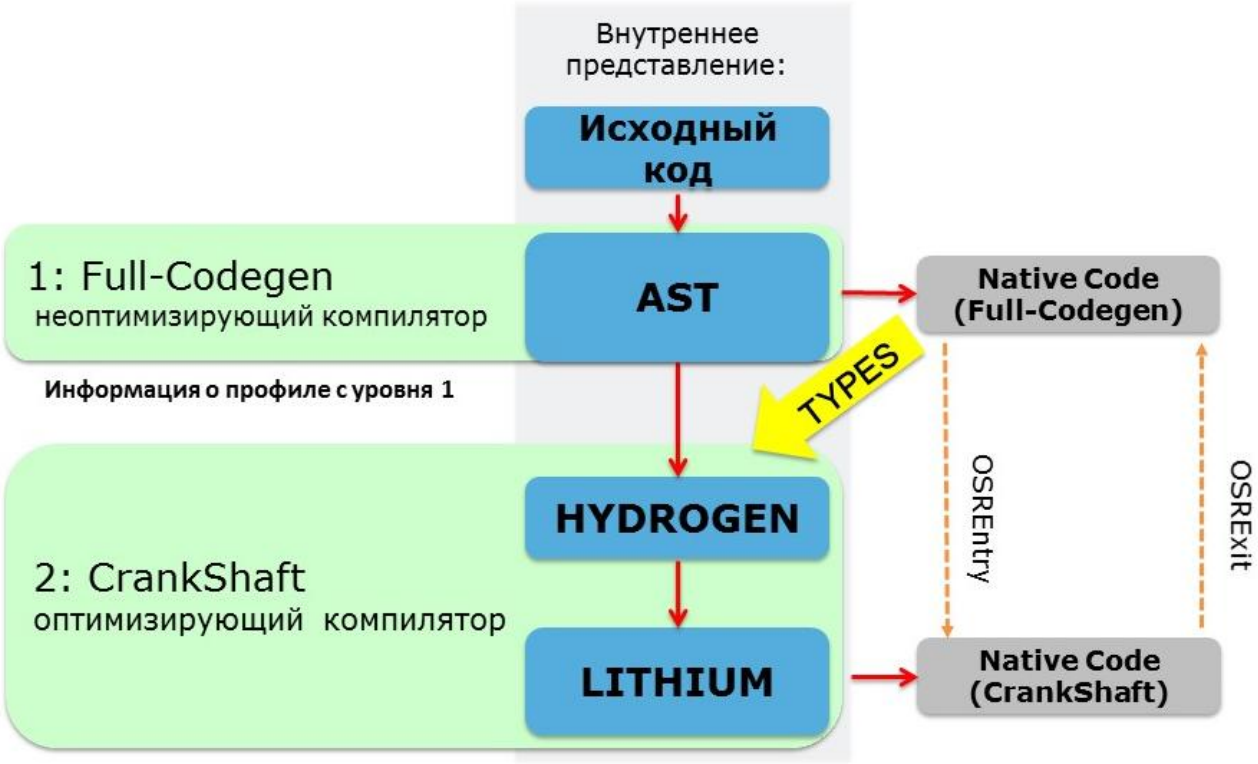


Рисунок 4. Многоуровневая архитектура компилятора V8

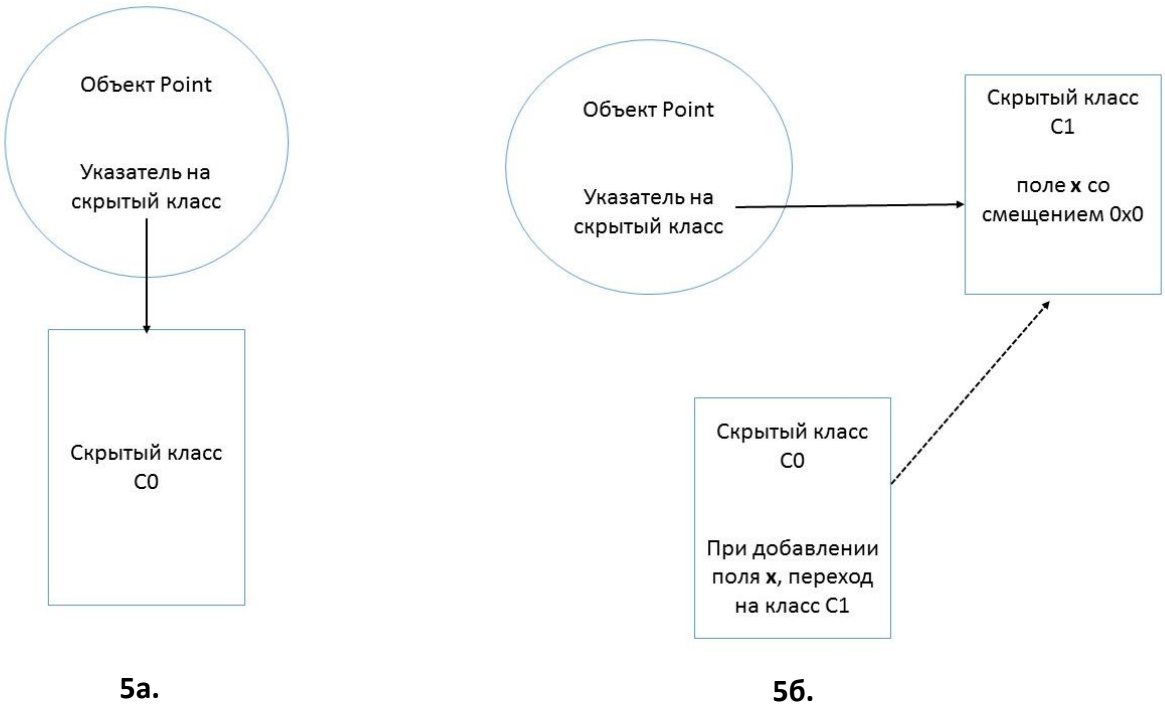
Реализация Full-Codegen представляет собой простую стековую машину (все локальные переменные хранятся на стеке или в куче) [34]. Во время генерации машинного кода применяются несколько простых оптимизаций, например, кэширование значения вершины стека для быстрого доступа.

Выполнение минимального набора оптимизаций обеспечивает быстрый запуск программы. При генерации машинного кода для каждой инструкции учитываются все возможные случаи выполнения для данной операции. На этом уровне также производится оптимизация обращений к полям и методам объектов. Для этой цели в Full-Codegen используется технология “скрытых классов” (англ. hidden classes) и встроенного кэша. Встроенный кэш также позволяет собирать информацию о типах и полях объектов. Эта информация используется на следующем уровне компиляции для организации спекулятивной компиляции. Во время выполнения программы Full-Codegen создает скрытые классы для каждого объекта. Эти классы во многом похожи на классы/структуры статических языков программирования и позволяют организовать быстрый доступ к полям объектов. Скрытый класс описывает структуру объектов JavaScript в текущий момент выполнения и меняется каждый раз при добавлении/удалении нового свойства или метода.

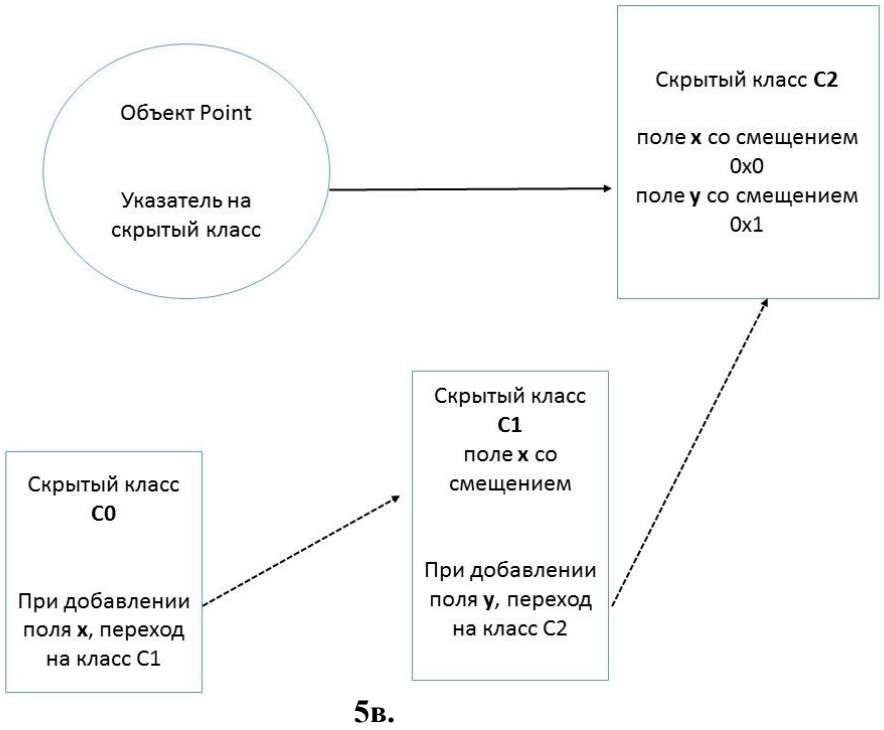
Рассмотрим следующую программу.

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}
var p1 = new Point(1, 3);
var p2 = new Point(4, 6);
```

Во время создания p1 при срабатывании *new Point(x, y)* для объекта *p1* создается скрытый класс — C0 (Рис. 5а). Выполнение первого выражения *this.x = x* приводит к созданию нового скрытого класса — C1. Класс C1 “наследуется” из класса C0 и содержит информацию о поле **x** объекта Point (например, поле **x** находится со смещением 0x0 от начала объекта). Класс C0 дополняется информацией о том, что, если в объект, описанный классом C0 добавится поле **x**, для него необходимо использовать класс C1 (Рис. 5б).



5а. **5б.**



5в.

Рисунок 5. Схема генерации скрытых классов

После выполнения *this.y = y* создается класс C2, унаследованный из класса C1. В C2 добавляется информация о поле **y**. Класс C1 обновляется аналогично

классу C0 (Рис. 5в). Во время создания объекта p2 новые скрытые классы не создаются, вместо этого p1 и p2 используют один и тот же скрытый класс C2. Скрытые классы позволяют применять одинаковую стратегию оптимизации обращений к полям и методам для объектов с общим скрытым классом. Оптимизация производится с помощью технологии встроенного кэша.

При первом обращении к полю объекта Full-Codegen определяет тип его текущего скрытого класса. Машинный код модифицируется и при последующих обращениях используется информация скрытого класса для загрузки значений непосредственно по смещению. В код также вставляются проверки для валидации текущего скрытого класса. Использование технологии “скрытых классов” и встроенного кэша позволяет оптимизировать случаи, когда в программе периодически создаются и используются объекты одного и того же типа.

Ниже приведен пример сгенерированного оптимизированного машинного кода для выражения *p1.x* для архитектуры x86_64.

```
# ebx объект p1.x
cmp [ebx,<hidden class offset>],<cached hidden class>; проверка скрытого класса
jne <inline cache miss>; переход на медленный код
mov eax, [ebx, <cached x offset>]; быстрый доступ к полю x
```

Построение встроенного кэша для объекта происходит в четыре этапа.

- Неинициализированный этап: обработка нового объекта. Вызывается общий (медленный) код для нахождения искомого метода (поля). Состояние меняется на пре-мономорфное.
- Пре-мономорфное состояние: в этом состоянии обращение к методам происходит так же, как и в предыдущем — вызывается общий код для нахождения метода (поля). Состояние меняется на мономорфное. Смысл этого этапа заключается в том, чтобы предотвратить генерацию

оптимизированного кода для объектов, которые используются только один раз.

- Мономорфное состояние: на этом этапе записывается информация о скрытом классе объекта и генерируется оптимизированный код для обращений к его полям и методам.
- Мегаморфное состояние: состояния для объектов, тип которых меняется часто. Для таких объектов обращение происходит через общий (медленный) код.

На уровне Full-Codegen также происходит профилирование для выявления часто выполнявшихся участков кода. Каждой JavaScript функции присваивается восьмибайтное значение, где хранится количество “срабатываний” функции. В Full-Codegen используется профилирование на основе (базе) счетчика (англ. counter-based profiling). Каждой функции, скомпилированной с Full-Codegen, присваивается счетчик. Значение счетчика уменьшается каждый раз при достижении обратных ребер циклов или при возврате из функции. Значение, на которое уменьшается счетчик, зависит от размера функции или цикла. Когда значение равняется нулю, управление передается коду профилировщика, где сканируется стек программы и выявляется текущая активная функция. Число срабатываний этой функции увеличивается на единицу. Далее применяются разные эвристики для принятия решения о переводе компиляции функции на оптимизированный уровень. Выполнение функции переводится на следующий уровень, если:

- Размер машинного кода (сгенерированного Full-Codegen) функции маленький, число срабатываний равно единице, и для него собран достаточный процент профильной информации: оценка размера функции зависит от целевой архитектуры. Например, для платформы X86 это число составляет 525 байт, а для ARM — 745. Процент пороговой профильной информации по умолчанию равен 25%.
- Число срабатываний функции равно двум, и для него собран

достаточный процент профильной информации (25% по умолчанию).

- Процент собранной профильной информации меньше порогового, но число срабатываний функции больше ста.

Переход на оптимизирующий уровень осуществляется с помощью технологии замены на стеке (OSR entry). Crankshaft — оптимизирующий уровень компилятора V8, из исходного кода заново строит абстрактное синтаксическое дерево (V8 не сохраняет АСД для экономии памяти). Из АСД строится граф потока управления (англ. control flow graph, CFG) в SSA-представлении – Hydrogen. Внутреннее представление Hydrogen позволяет выполнить ряд машинно-независимых оптимизаций [28], в том числе:

- встраивание функций;
- удаление мертвого кода;
- оптимизацию хранения объектов в куче (англ. escape analysis) [35];
- глобальную нумерацию значений [36];
- вынос инвариантного кода за цикл;
- анализ диапазонов (англ. range analysis);
- удаление излишних проверок переполнения;
- удаление излишних проверок выхода за границу массива.

Crankshaft использует собранную на предыдущем уровне информацию о типах для эффективного хранения целочисленных переменных и операций над ними. На 64-битных платформах целочисленные значения хранятся в старших 32 битах, а младшие биты заполняются нулями. На 32-битных платформах целые числа хранятся в старших 31 битах, а младший бит вставляется равным нулю. Целые числа в такой кодировке называются Smi (от английского small integer). В общем случаи в JavaScript все переменные ссылаются на объекты. Для того, чтобы возможно было отличить указатели от Smi, в V8 все указатели кодируются с младшим битом (определяющий четность числа), установленным в единицу. Такое кодирование возможно, так как все адреса выровнены, т.е. объектов с нечетными адресами не существует.

Использование сохраненной информации позволяет организовать спекулятивные оптимизации. Эти оптимизации основаны на предположении, что собранная на нижнем уровне информация (например, тип или значение переменных) останется неизменной во время выполнения функции. Для этого в код вставляются необходимые проверки. Когда одна из таких проверок не выполняется, происходит переключение на уровень Full-Codegen — деоптимизация. Для переключения между разными уровнями компиляции используется технология замены на стеке. 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 этом представлении инструкции являются трехадресными. Каждая Lithium инструкция описывается соответствующей операцией и ее выходными, входными и промежуточными операндами. Эти операнды описывают особенности целевой архитектуры. Например, для архитектуры X86, операнд операции деления в представлении Lithium будет содержать информацию о том, что значение делимой должно находиться в регистре ax (dx:ax). Эта информация далее используется при распределении регистров, которое реализовано с помощью алгоритма линейного сканирования [37]. После распределения регистров все Lithium операнды отображаются в соответствующие регистры или слоты памяти. После этого происходит кодогенерация.

В настоящее время разработчиками компилятора V8 активно разрабатывается альтернативный уровень оптимизации — TurboFan. TurboFan использует промежуточное представление “sea of nodes” [38], что позволяет выполнять больше динамических оптимизаций. Это представление также позволяет реализовать оптимизации специально для asm.js кода.

V8 использует сборщик мусора на основе поколений (англ. generational garbage collection) [39].

Для организации эффективной сборки мусора, куча в V8 разделена на несколько частей:

- Новое поколение. Большинство объектов создаются в этой части кучи.
- Старое поколение указателей. Здесь хранятся объекты-указатели, которые остались “в живых” после нескольких итераций сборки мусора в новом поколении.
- Старое поколение данных. Здесь хранятся объекты, содержащие данные, которые остались “в живых” после нескольких итераций сборки мусора в новом поколении.
- Пространство для больших объектов. Здесь хранятся объекты больших размеров.
- Пространство кода. Здесь хранятся созданные компилятором исполняемые коды функций.

Каждая часть состоит из последовательности страниц. Каждая такая страница занимает 1МБ памяти.

По статистике, в большинстве программ объекты имеют короткий цикл жизни. Учитывая это, в V8 реализованы три алгоритма сборки мусора. Для объектов “нового поколения” реализован алгоритм Чейни [40], который обеспечивает быструю сборку мусора для небольшого объема памяти, что позволяет минимизировать паузы при сборке мусора. Объекты в “новом поколении”, которые остаются “в живых” в течение двух сборок мусора, переводятся в часть “старого поколения”. На этом уровне объем занимаемой объектами памяти может достичь нескольких сотен мегабайт. Для управления памятью в “старом поколении” в V8 реализованы алгоритмы Mark-sweep и Mark-compact [41]. Mark-sweep используется для удаления мертвых объектов, Mark-compact — для удаления фрагментации кучи.

Сборщик мусора V8 не предусматривает параллельной работы вместе с выполнением программы, т.к. это может приводить к конкуренции потоков во время доступа к одним и тем же объектам и порождать недетерминированные паузы. Кроме того, сборщик мусора может переносить объекты, что приводит к тому, что сам код, ссылающийся на них, должен быть изменен. Для некоторых виртуальных машин [42], предусматривающих JIT-компиляцию, в том числе и для V8, в ряде случаев возникает необходимость как можно скорее остановить потоки выполнения скомпилированного кода. Необходимость приостановления выполнения всех потоков возникает, например, при сборке мусора. Для того, чтобы информация о всех объектах была доступна во время сборки мусора, в V8 используется технология *safepoints*. В точке *safepoint* программа находится в согласованном состоянии, где доступна вся информация об объектах программы. Чтобы обеспечить быстрый доступ к точкам *safepoint* они, как правило, расставляются на обратных ребрах циклов и в местах вызовов функций. В этих точках также генерируются различные проверки состояния программы. Если эти проверки срабатывают, выполнение останавливается и управление передается сборщику мусора.

1.3 Другие реализации языка JavaScript

Другими популярными компиляторами языка JavaScript являются SpiderMonkey [7] (используется в браузере Firefox и в операционной системе FirefoxOS), разработанный компанией Mozilla, и ChakraCore [8] (используется в браузерах IE и Edge), разработанный компанией Microsoft. Эти компиляторы также реализуют многоуровневую архитектуру. Компилятор SpiderMonkey распространятся с открытым исходным кодом и состоит из двух уровней компиляции. Первый уровень представляет собой простой интерпретатор, где происходит сбор информации о профиле программы. На втором уровне работает оптимизирующий JIT-компилятор IonMonkey, который использует собранную информацию о профиле для спекулятивных оптимизаций. В

отличие от компиляторов JavaScriptCore и V8, в SpiderMonkey реализован специальный модуль для оптимизации asm.js кода — OdinMonkey. OdinMonkey не является отдельным уровнем оптимизации и основан на уровне IonMonkey.

ChakraCore, в отличие от других компиляторов, до января 2016 года поставлялся только с закрытым исходным кодом. В этом компиляторе реализованы три уровня компиляции. Из исходного кода строится промежуточное представление байткода, после чего функция начинает выполняться на уровне интерпретатора. Второй уровень оптимизации – Simple JIT – генерирует машинный код с применением минимального набора оптимизаций. На первых двух уровнях компиляции собирается профиль программы. Третий уровень – Full JIT – использует собранный профиль для спекулятивных оптимизаций. Компилятор Chakra, как и SpiderMonkey, поддерживает модуль для оптимизации asm.js кода.

1.4 Обзор тестовых наборов JavaScript

Наиболее популярными тестовыми наборами для измерения производительности компиляторов JavaScript являются SunSpider (LongSpider) [43], Octane [44], Browsermark [45] и Kraken [46]. Набор SunSpider разработан компанией Apple и содержит 26 тестов, что обеспечивает покрытие почти всех конструкций языка JavaScript. Ниже приведено описание тестов из набора SunSpider:

- 3d{cube, raytrace, morph} — тесты для графических вычислений. Покрывают такие конструкции языка, как массивы (создание, чтение/запись элементов) и вещественные математические операции.
- Access{binary-trees, nbody, nsieve} — покрывают часть работы над объектами. Создание объектов, доступ к полям и методам.
- Bitops{3bit-bits-in-byte, bits-in-byte, bitwise-and, nsieve-bits} — покрывают часть бытовых операций.

- Controlflow-recursive — тест для управляющих конструкций языка (циклы, условные переходы), а также рекурсивные вызовы функций.
- Crypto{aes, md5, sha1} — криптографические тесты. Покрывают часть бытовых операций, а также операций над строками.
- Date{format-tofte, format-xparb} — тесты производительности разных операций над объектом Date языка JavaScript.
- Math{cordic, partial-sum, spectral-norm} — покрывают часть математических операций. Вызовы функций из библиотеки Math, операции с вещественными числами.
- Regexp-dna — тест производительности регулярных выражений.
- String{base64, fasta, tagcloud} — тестирование производительности строковых операций, в том числе операций над JSON объектами, сжатия/извлечения текстов.

Набор LongSpider содержит те же тесты с удлиненным временем выполнения.

Тестовые наборы Octane и v8-v7 разработаны компанией Google. Octane является обновлением набора v8-v7 и содержит тесты для проверки пропускной способности интерактивных веб приложений и сборки мусора. Ниже приведено описание некоторых тестов:

- Richards — симуляция ядра операционной системы. Содержит функционалы, отвечающие за загрузку объектов, вызов функции, оптимизацию кода и удаление повторного кода.
- Deltablue ориентирован на полиморфизм и ООП.
- Raytrace — работа с объектами и аргументами функции.
- Regexp — тестирование производительности операций с регулярными выражениями, полученными на основе анализа более 50 популярных веб-страниц.
- NavierStokes, Pdf, Box2DWeb — создание, чтение и запись массивов, а также операции с числами с плавающей точкой.

- Crypto — криптографические вычисления, содержащие битовые операции.
- Splay, EarleyBoyer — создание и удаление объектов.
- SplayLatency — измерение задержки сборщика мусора.
- Zlib — библиотека для сжатия данных, представляет собой asm.js код, полученный компилятором Emscripten из языка C. Используется для измерения производительности компиляции JavaScript кода.
- CodeLoad — загрузка кода: измеряет время разбора и компиляции JavaScript кода.
- TypeScript — компилятор для языка TypeScript компании Microsoft. Измеряет производительность компиляторов JavaScript для приложений больших размеров.

Тестовый набор Kraken, разработан компанией Mozilla и содержит тесты производительности для следующих операций:

- обработка аудио (звука);
- обработка изображений;
- разбор данных в формате JSON;
- обработка криптографических данных

Browsermark — тестовый набор, нацеленный на измерение производительности браузеров мобильных и встраиваемых систем. Содержит тесты для визуализации (англ. rendering), графической библиотеки WebGl, видео-тесты HTML5, тесты для манипуляции с объектами DOM, а также тесты для разбора текстов.

1.5 Выводы и постановка задачи

В настоящее время динамические языки программирования получили широкое распространение. С ростом производительности персональных компьютеров и встраиваемых систем использование динамических языков стало возможно не только для выполнения небольших скриптов на веб-страницах, но и для разработки целых веб-приложений. Более того, уже

имеются разработки операционных систем для телефонов, планшетов и ноутбуков, которые используют динамические языки для создания пользовательских приложений. Примерами таких систем являются Tizen и FirefoxOS. Некоторая часть пользовательских приложений на этих системах представляет собой набор хранящихся на устройстве веб-страниц со скриптами на языке JavaScript.

С другой стороны, в настоящее время активно развиваются платформы asm.js и webassembly, которые позволяют эффективно выполнять приложения со статическими характеристиками в динамической среде.

Приложения на языках с динамическими типами становятся все более комплексными и сложными. В связи с этим все больше возрастают требования к производительности динамических компиляторов. Использование динамических языков в операционных системах и в приложениях со статическими характеристиками открывает возможности для разработки новых методов оптимизации современных динамических компиляторов. Для оптимизации приложений в операционных системах можно использовать информацию о профиле программы. Для улучшения производительности приложений со статическими характеристиками можно использовать методы, которые применяются для оптимизации программ, написанных на статических языках программирования. Технологии оптимизации таких программ тщательно разработаны и хорошо изучены. Одной из известных систем для анализа, трансформации и оптимизации программ, написанных на статических языках программирования, является компиляторная инфраструктура LLVM.

Инфраструктуру LLVM можно также использовать для выполнения более сложных оптимизаций для наиболее часто исполняющихся участков кода, и генерации более оптимального машинного кода для таких участков.

Основной задачей диссертационной работы является разработка новых методов оптимизации многоуровневых динамических компиляторов, учитывая новые тенденции применения динамических языков программирования в

качестве основных языков для разработки приложений в операционных системах, а также для разработки веб приложений со статическими характеристиками.

Глава 2. Методы оптимизации многоуровневых динамических компиляторов, основанные на статистике выполнения программы

Применение динамических языков программирования в качестве основного языка для разработки приложений в операционных системах делает возможным использование информации о профиле программы для реализации как известных оптимизаций, так [12] и разработки новых. Сбор информации о профиле программы можно организовать на этапе тестирования программного обеспечения и использовать его для оптимизации приложений под конкретные случаи исполнения.

2.1 Особенности использования профиля выполнения программы в многоуровневых динамических компиляторах

Одним из новых применений использования информации о профиле программы может быть обеспечение немедленного перехода выполнения «горячих» участков кода на уровень оптимизирующего компилятора. Для обоснования необходимости скорейшего переключения на оптимизирующие уровни выполнения приведем примеры сравнения времени выполнения разных уровней компиляторов JavaScriptCore на нескольких тестовых наборах.

В таблице 1 приведен результат тестирования набора PL benchmark от автора Martin Richard. Из таблицы видно, что на оптимизирующем уровне DFG JIT код выполняется около 4 раз быстрее чем на уровне Baseline JIT. Тестирование на другом наборе Browsermark показало, что для этого набора

Baseline JIT оказывается в среднем в 2,5 раза быстрее, чем LLInt. Уровень DFG JIT, в свою очередь, в среднем еще 1.7 раза быстрее Baseline JIT.

Таблица 1 Сравнение производительности уровней компилятора JSC на наборе PL Benchmark

Способ выполнения	**Время выполнения, мс**
LLInt интерпретатор	129
Baseline JIT	8.4
DFG JIT	2.1

На рис. 6 и рис. 7 приведено относительное время выполнения разных уровней компилятора JSC на тестовых наборах SunSpider и V8_V7. На этих наборах также уровень DFG JIT оказывается в несколько раз быстрее, чем LLINT и Baseline JIT.

Из анализа сравнения производительности разных уровней выполнения многоуровневых динамических компиляторов видно, что производительность кода, генерируемого на оптимизирующих уровней компиляции в среднем в 2 – 3 раза больше, нежели производительность неоптимизирующих уровней. Поэтому важно добиться, чтобы «горячие» участки кода как можно больше времени выполнялись именно на оптимизирующих уровнях компиляции.

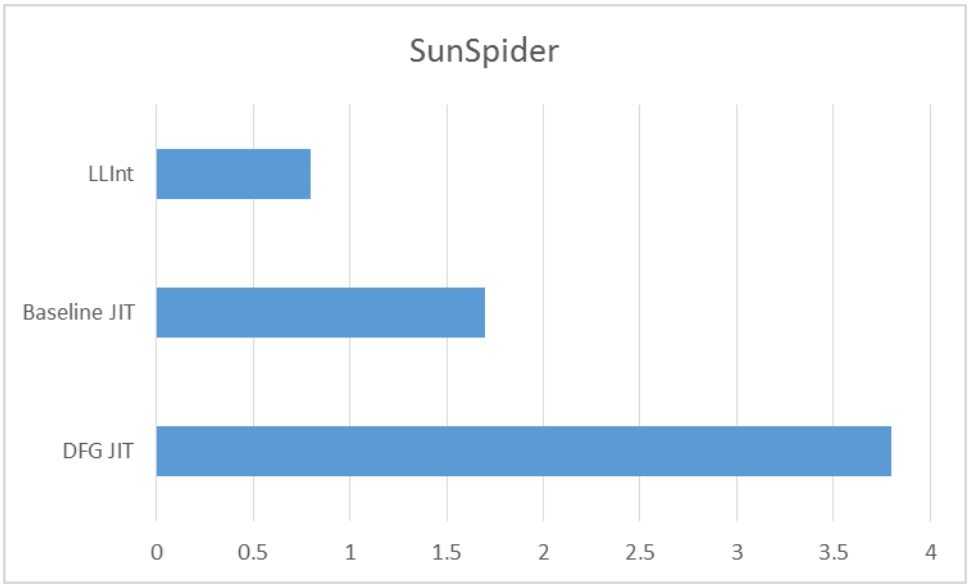


Рисунок 6.Относительная производительность разных уровней компилятора JSC на тестовом наборе SunSpider (больше - лучше)

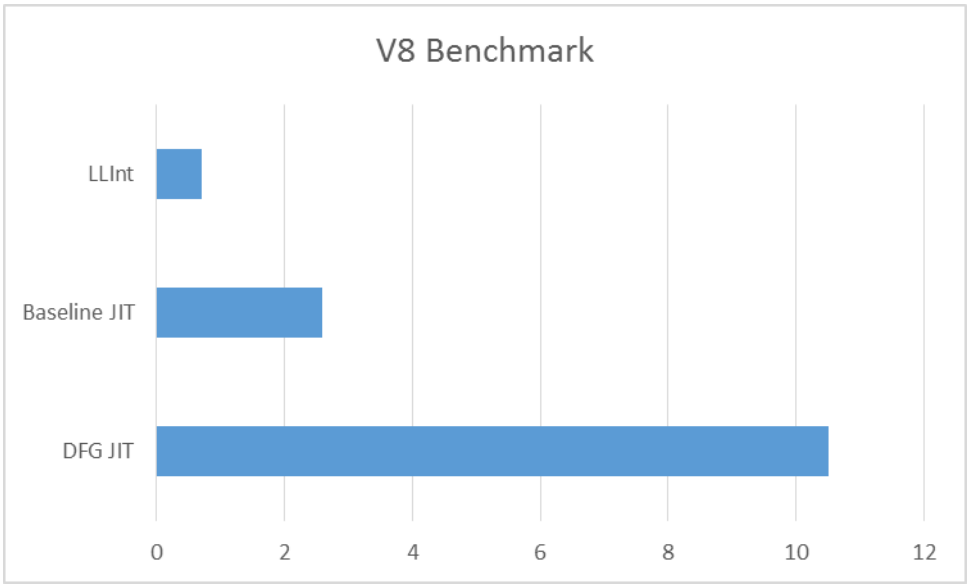


Рисунок 7. Относительная производительность разных уровней компилятора JSC на тестовом наборе V8 (больше - лучше)

Один из способов достижения этой цели — обеспечение немедленного (или как можно более быстрого) переключения выполнения «горячих» участков кода на оптимизирующие уровни компиляции.

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

2.1.1 Использование профиля выполнения программы для обеспечения немедленного переключения на оптимизирующие уровни компиляции

Немедленного переключения на оптимизирующие уровни выполнения можно достичь путем внедрения в компилятор механизмов, позволяющих собирать и сохранять информацию о «горячих» участках кода. Многоуровневая архитектура динамических компиляторов уже содержит механизмы для нахождения часто выполняемых участков кода. В компилятор V8 был добавлен модуль для сохранения информации о «горячих» участках кода. Далее, при последующих запусках программы, сохраненная информация позволяет сразу перевести выполнение таких участков на оптимизирующие уровни

компиляции. Кроме информации о «горячих» участках кода, была также добавлена поддержка сохранения информации о типах переменных и полях объектов, поскольку при организации немедленного переключения, необходимо учитывать, что выполнение спекулятивных оптимизаций требует наличия профиля функции

Этот метод особо эффективен для программ с небольшим временем выполнения. При выполнении таких программ часто встречается следующая ситуация: некий участок кода помечается как кандидат для оптимизации на следующем уровне компиляции. Параллельно с выполнением программы начинается компиляция этого участка на оптимизирующем уровне, но программа заканчивается раньше, чем начинается выполнение оптимизированной версии кода. Использование информации о профиле программы позволяет переключать выполнение «горячих» участков кода на оптимизирующий уровень немедленно, что приводит к значительному росту производительности для программ с небольшим временем выполнения [21].

2.1.2 Использование профиля программы для устранения обратных переходов на неоптимизирующие уровни выполнения

В реальных приложениях, написанных на динамических языках, деоптимизации могут встречаться очень часто. В компиляторе V8 информация о количестве и причинах деоптимизаций используется для исключения больших временных затрат на постоянную реоптимизацию. 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 способом:

- Если при первом запуске программы для какой-либо функции оптимизация запрещается, при последующих вызовах программы не будут предприниматься попытки ее оптимизации.

- Если при первом запуске программы в какой-либо функции происходит деоптимизация из-за недостаточной информации о типах, при последующих вызовах переключение выполнения этой функции на оптимизирующие уровни откладывается, что позволяет собрать более точную информацию о типах (50% по умолчанию).

2.2 Использование профиля программы для выбора набора оптимизаций

Влияние конкретных оптимизаций на созданный бинарный код для разных приложений может быть незначительным Использование таких оптимизаций во время динамической компиляции может стать причиной ухудшения производительности для конкретных приложений. Приведем анализ эффективности разных оптимизаций компилятора V8 на тестовом наборе v8-v7 и на реальном примере использования сайта Gmail. Более подробный анализ других приложений (Facebook, Wordpress) можно найти в [47].

Для имитации реального использования сайта Gmail была использована платформа Sikuli UI [48], которая позволяет автоматизировать использование графического интерфейса. Сценарии использования сайта Gmail следующая: производится авторизация, затем запрашиваются все входящие сообщения и производится поиск сообщений по нескольким критериям.

Для оценки эффективности реализованных оптимизаций был проведен сравнительный анализ времени выполнения компилятора V8 при запусках с тремя разными конфигурациями. При первом запуске используется конфигурация V8 по умолчанию. Второй запуск был произведен с выключением около одиннадцати оптимизаций, в том числе, нумерации глобальных значений, выноса инвариантного кода за цикл, анализа диапазонов, удаления лишних Ф-функций, встраивания вызовов функций, встраивания полиморфных вызовов функций. При третьем запуске оптимизирующий уровень компилятора был полностью отключен (только Full-Codegen).

На рис. 8 и 9 приведены результаты сравнения производительности на

наборе v8-v7 и при использовании сайта Gmail. Как видно из результатов на тестовом наборе v8-v7, отключение оптимизирующего уровня приводит к значительному ухудшению производительности (около 51%). Отключение разных оптимизаций на уровне Crankshaft также приводит к ухудшению производительности. Однако на примере использования сайта Gmail выключение некоторых оптимизаций, наоборот, приводит к улучшению производительности.

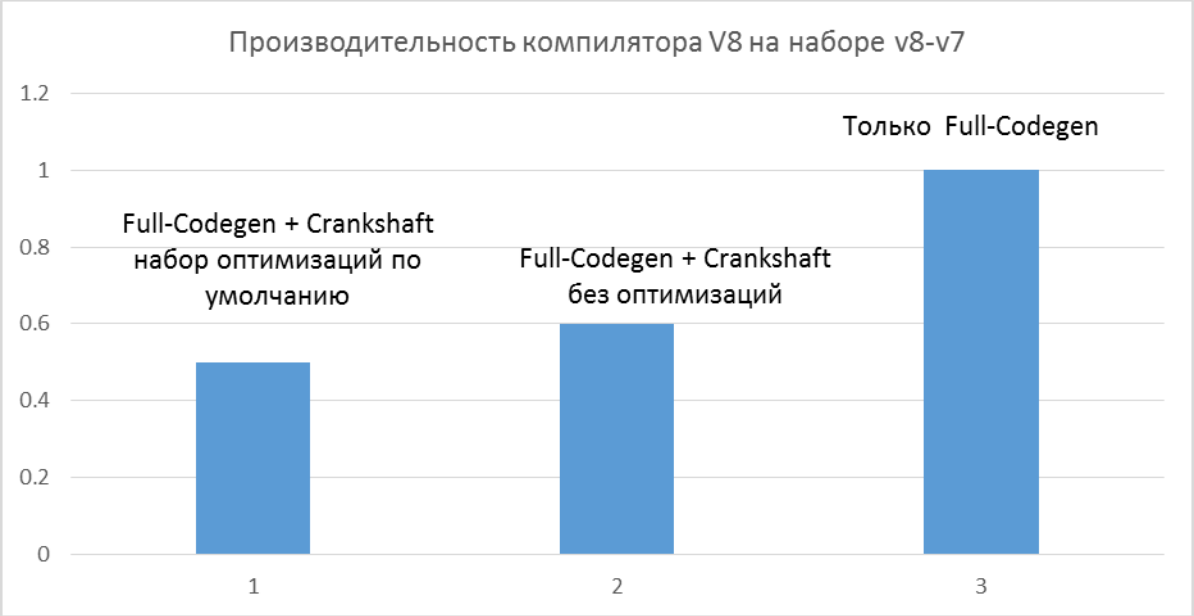


Рисунок 8. Относительное время выполнения V8 при разных конфигурациях на наборе v8-v7. Единицей времени выступает время выполнения Full-Codegen

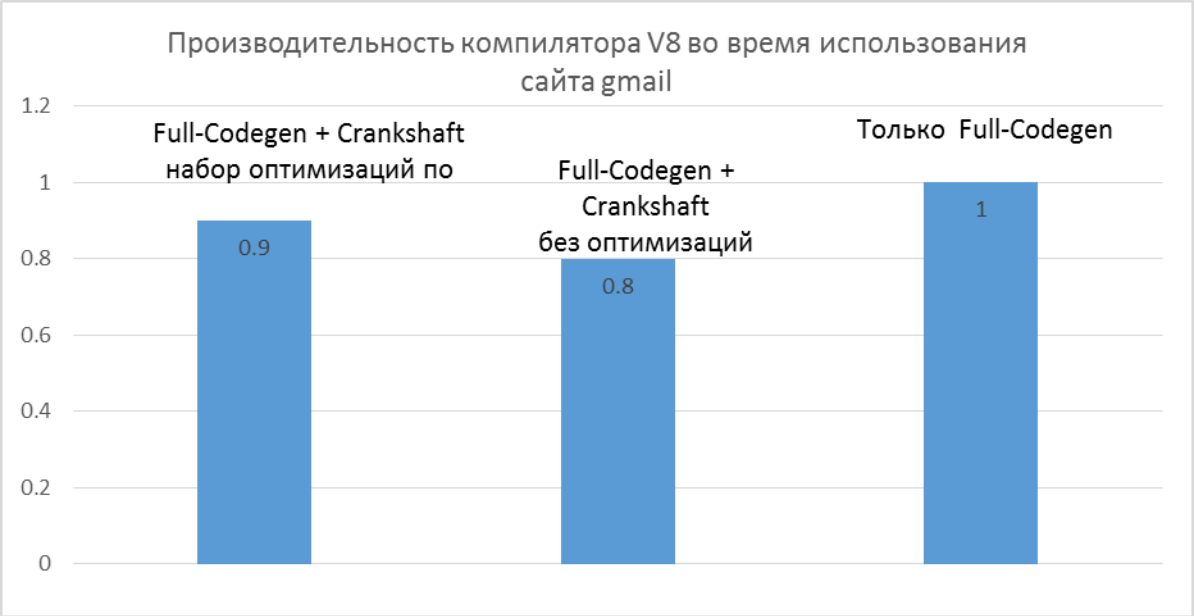


Рисунок 9. Относительное время выполнения V8 при разных конфигурациях на сайте Gmail. Единицей времени выступает время выполнения Full-Codegen

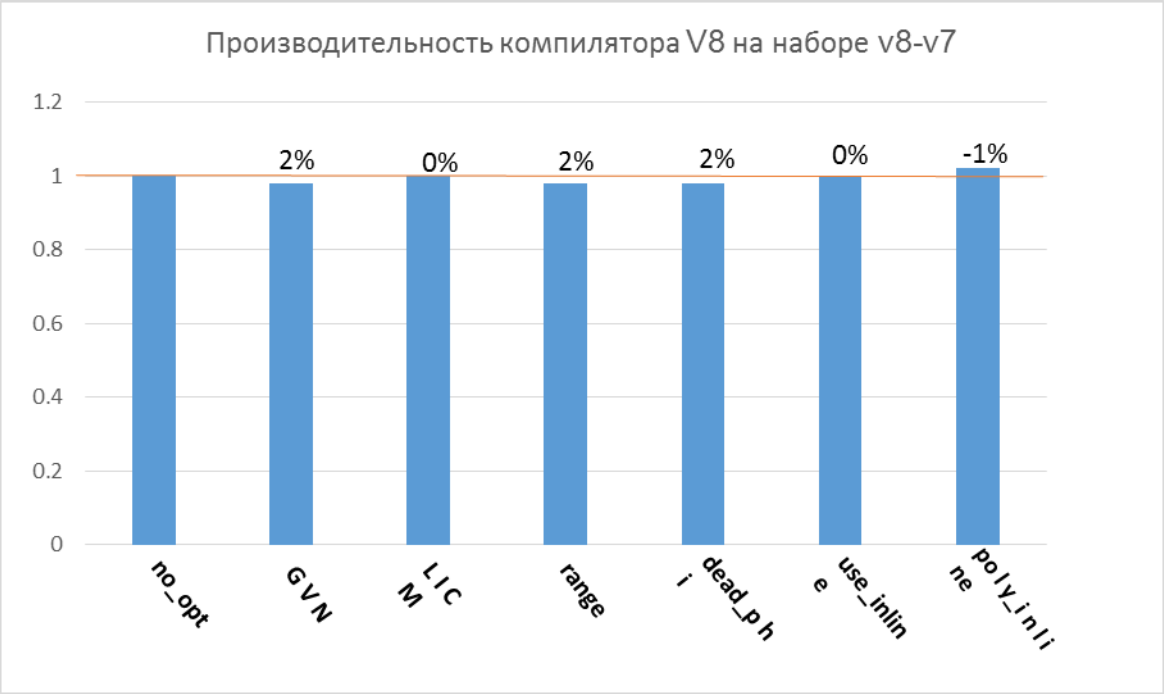


Рисунок 10. Относительное время выполнения V8 на наборе v8-v7. Единицей времени выступает время выполнения компилятора с выключенными оптимизациями

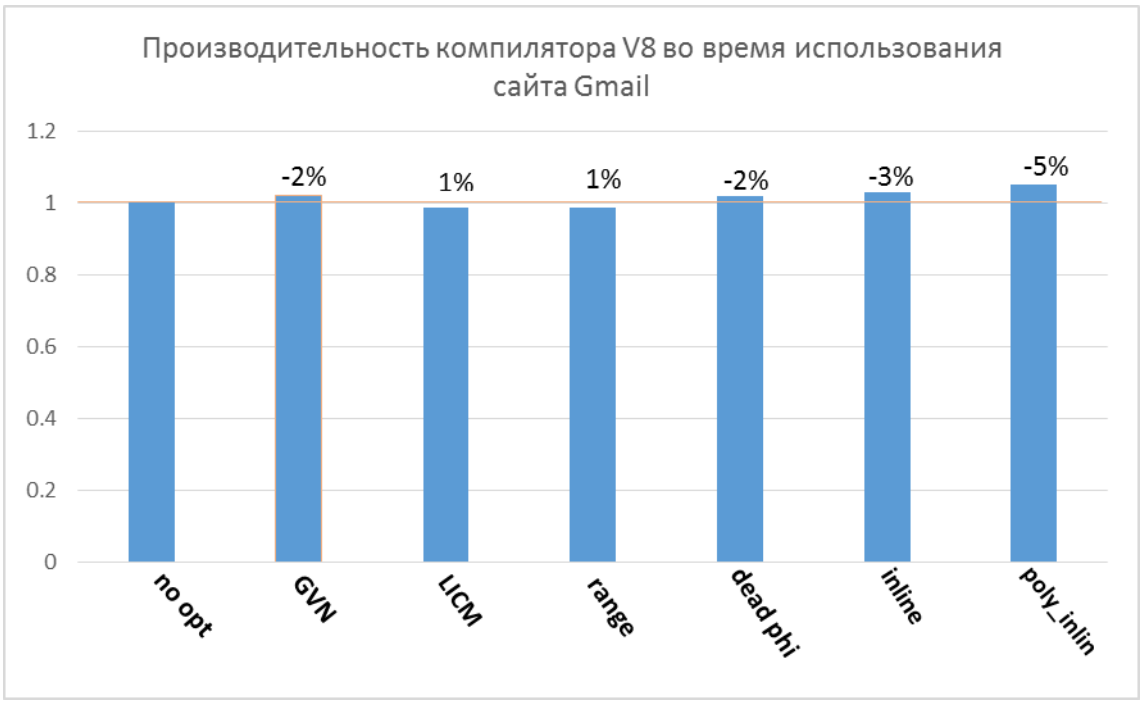


Рисунок 11. Относительное время выполнения V8 при использовании сайта Gmail. Единицей времени выступает время выполнения компилятора с выключенными оптимизациями

Для оценки влияния каждой оптимизации на производительность в целом, были проведены тестирования с последовательным включением каждой из оптимизаций. Результаты тестирований на наборе v8-v7 и на сайте Gmail приведены на рис. 10 и 11.

Как видно из результатов, на наборе v8-v7 большинство оптимизаций имеет положительный эффект, 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 об эффективности каждой оптимизации позволяет выбрать оптимальный набор оптимизаций для каждого приложения. Оценка необходимости использования той или иной оптимизации вычисляется по следующей формуле:

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

Где **n** — количество удаленных (измененных) оптимизаций инструкций, FC — количество выполнений функции, LC_i — 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 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.

Поддержка оценки и сохранения эффективности была добавлена 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;

Важно также отметить, что выполнение некоторых оптимизаций может повлиять на эффективность других. Например, анализ диапазонов используется при удалении избыточных проверок. В таких случаях, для оценки эффективности также учитывается возможное влияние 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 [37], а для больших функций с тяжелыми циклами используется жадный алгоритм линейного сканирования [49].

2.3 Выводы и результаты тестирований

Был разработан и реализован новый метод оптимизации многоуровневых динамических компиляторов с использованием информации о профиле программы. Метод позволяет улучшить производительность компиляторов, путем

- Организации немедленного переключения выполнения «горячих» участков кода на уровень оптимизирующего компилятора.

- Удаления деоптимизаций и обратных переходов на неоптимизирующий уровень выполнения.
- Выбора набора оптимизаций для конкретных приложений.

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 инструментария для вычисления количества итераций циклов и эффективности оптимизаций при первом запуске программы замедляет набор SunSpider всего на 5%. Набор Octane замедляется на 23%, а Kraken — на 25%.

Использование информации о профиле программы для организации более быстрого перехода на оптимизирующие уровни выполнения, а также для реализации метода выбора оптимального набора оптимизаций для конкретных приложений позволили ускорить множество тестов из наборов SunSpider и Kraken. В среднем, тестовый набор SunSpider стал выполняться на 11% быстрее, ускорение конкретных тестов составляет 50% (таблица 2). 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 2. Производительность набора SunSpider с использованием оптимизаций на основе профиля программы (меньше - лучше)

Тест	Производительность V8, мс	Производительность V8 с реализованными улучшениями, мс	Улучшение, %
3d-cube	48.8	46	5.73
3d-morph	51.7	52.4	-1.3
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	2.7

Таблица 2. Продолжение			
access-nsieve	8.8	7.2	18.2
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	0.9
math-spectral-norm	11	7.2	34.5
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

Глава 3. Компиляция программ на языке JavaScript в статически типизированное внутреннее представление LLVM

В настоящее время развиваются платформы asm.js [50] и webassembly [51], которые позволяют эффективно выполнять приложения, написанные на статических языках программирования в динамической среде. Asm.js представляет собой подмножество языка JavaScript. Основная его идея – ограничить динамические свойства языка JavaScript для достижения большей производительности. Например, в asm.js переменные строго типизированы. Рассмотрим следующий пример кода на JavaScript:

```
var first = 5
var second = first;
```

Синтаксис соответствующего asm.js кода для этого примера имеет следующий вид:

```
var first = 5
var second = first | 0;
```

Единственная разница между этими примерами — операция ИЛИ над переменной *first*. Эта операция гарантирует, что переменная *second* всегда будет иметь тип *int32*. Ограничение динамических свойств позволяет эффективно выполнять приложения, написанные на статических языках в веб-

среде. Для генерации asm.js кода используются разные трансляторы. Наиболее популярным из них является Emscripten [52], который генерирует asm.js код из внутреннего представления инфраструктуры LLVM и позволяет выполнять приложения на языках C/C++[1] в веб-среде. Webassembly [51] является еще одной развивающейся платформой для поддержки выполнения приложений, написанных на статических языках в веб-среде.

Для улучшения производительности этих приложений можно использовать методы, которые применяются для оптимизации программ, написанных на статических языках программирования, таких как C/C++. Технологии оптимизации таких языков тщательно разработаны и хорошо изучены. Компиляторная инфраструктура LLVM является одной из известных систем для анализа, трансформации и оптимизации статически типизированных языков программирования. Система также предоставляет средства для динамической компиляции в виде модуля под названием MCJIT [53], в котором уже задействованы все имеющиеся механизмы LLVM для машинно-независимых и машинно-зависимых оптимизаций, а также для кодогенерации под различные платформы. Более того, приложения на динамических языках программирования становятся все более комплексными и сложными. Инфраструктуру LLVM можно также использовать для оптимизации наиболее часто исполняющихся участков кода.

В данной главе приводится описание метода динамической компиляции программ на языке JavaScript в статически типизированное внутреннее представление инфраструктуры LLVM. Компиляция была осуществлена путем добавления дополнительного уровня выполнения (LLV8[2]) в компиляторе V8 (рис. 12). Добавление нового уровня оптимизации с использованием инфраструктуры LLVM позволит применить оптимизации, имеющиеся в LLVM, к программам, написанным на языке JavaScript. Однако для компиляции языков с динамическими типами в статическое представление требуется

[1] Инфраструктура LLVM также поддерживается для языков ADA, D, Delphi, Fortran, Objective-C, Swift
[2] Читается “эл-эл-ви-эйт”, производное от названий LLVM и V8.

решить много новых задач: например, обеспечить поддержку спекулятивной компиляции и деоптимизаций, а также обеспечить бинарную совместимость и переход между разными уровнями оптимизации, поддержку сборщика мусора и перемещений объектов.

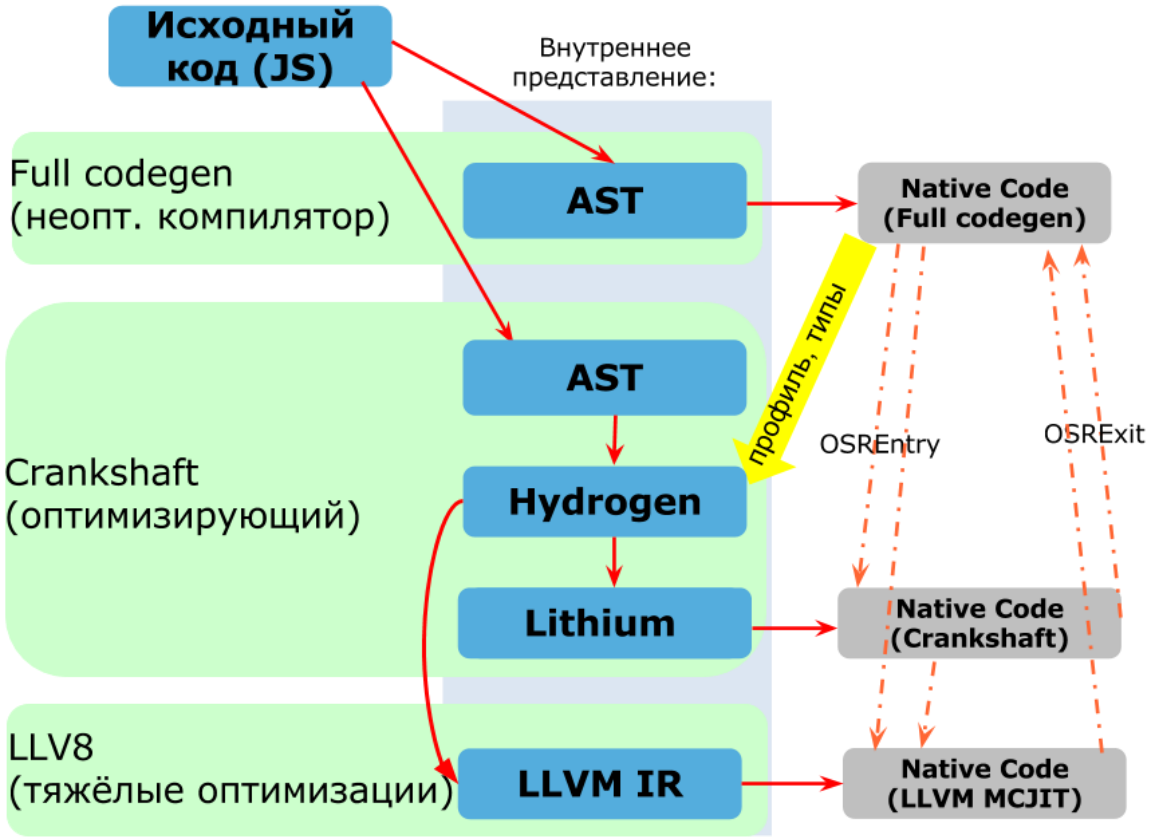


Рисунок 12. Архитектура многоуровневого компилятора V8 с добавленным модулем LLVM

Инфраструктура LLVM используется в компиляторе JavaScriptCore на четвертом уровне оптимизации (FTL JIT) [5]. На момент написания данной работы FTL JIT поддерживался только для платформ Mac OS X и iOS, в данной же работе предлагается добавить уровень оптимизации LLVM в компилятор V8, и в первую очередь для операционной системы GNU/Linux и процессоров x86-64. Важно отметить, что компилятор V8 существенно более популярен: JavaScriptCore используется только в веб браузере Safari, тогда как V8 встроен в браузеры Chrome, Opera, Android-браузер, в платформу Node.js и в

операционную систему ChromeOS. Стоит также отметить, что, несмотря на внешние сходства общей архитектуры, эти два компилятора отличаются по внутренней структуре и реализации.

3.1 Трансляция конструкций JavaScript в промежуточное представление LLVM.

В качестве исходного представления для преобразования в LLVM биткод было выбрано промежуточное представление Hydrogen. Hydrogen уже содержит информацию о типах переменных и полях объектов, таким образом, в этом представлении уже имеется вся информация, необходимая для получения статически типизированного представления. Несмотря на то, что и Hydrogen, и LLVM биткод используют форму SSA [27], существуют 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 (в котором находится Ф-функция). Вход Ф-функции в LLVM — это пара из названия переменной и базового блока, в котором эта переменная “жива”, то есть уже определена. Таких базовых блоков может быть несколько, и в LLVM это не обязательно блок, в котором переменная определяется. На рис. 13 изображен граф потока управления (в обозначениях LLVM), корректный с точки зрения Hydrogen и некорректный с точки зрения LLVM. На рис. 14 приведен такой же граф, скорректированный под требования LLVM. Все отличия находятся в базовом блоке *BB4*. LLVM требует, чтобы базовые блоки инструкций *%8* и *%1* являлись непосредственными предшественниками базового блока *BB4*.

3.1.1 Алгоритм преобразования Ф-функций в корректную с точки зрения LLVM форму.

Был разработан и реализован алгоритм преобразования Ф-функций представления Hydrogen в корректную с точки зрения LLVM форму. Приведем описание алгоритма. Для этого прежде дадим определение доминирования в графе потока управления. Узел *d* доминирует над узлом *n* (записывается как *d*

dom n), если любой путь из начального узла графа потока управления в узел *n* проходит через *n*. При таком определении каждый узел доминирует над самим собой.

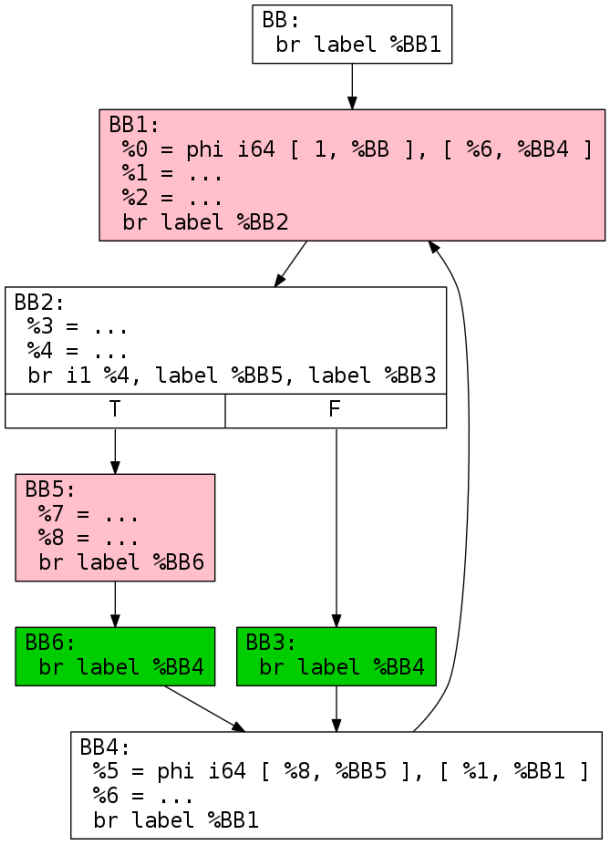


Рисунок 13. Граф потока управления, некорректный с точки зрения LLVM

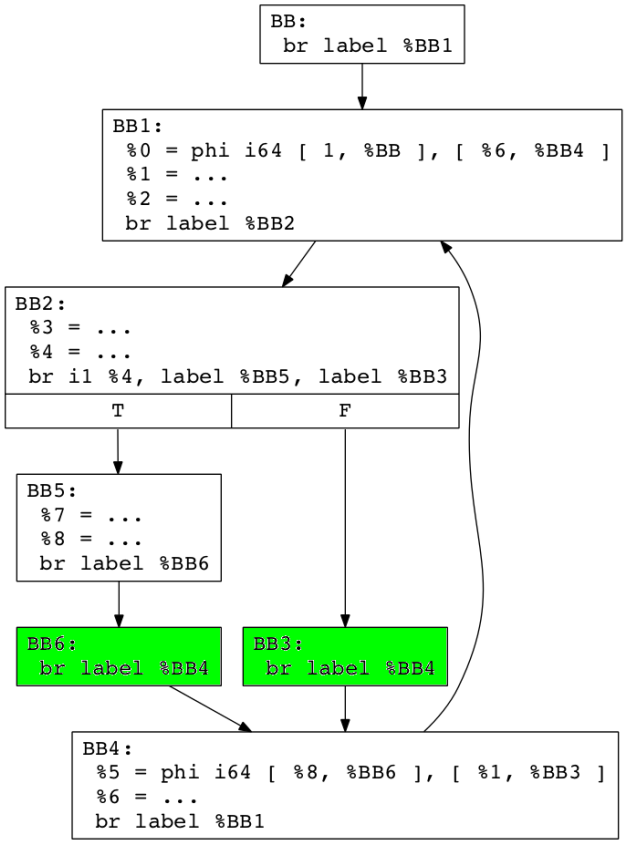


Рисунок 14. Граф потока управления, корректный с точки зрения LLVM

Каждый узел *n*, кроме начального, имеет единственный непосредственный доминатор — последний доминатор на любом пути от начального узла к *n* (не совпадающий с *n*). В терминах отношения *dom* непосредственный доминатор *m* обладает следующим свойством: если $d \neq n$ и *d dom n*, то *d dom m*. Из этого следует, что для графа потока можно построить дерево доминаторов — дерево, в котором входной узел является корнем, а каждый узел *d* доминирует только над своими потомками в дереве. На рис. 15 показано дерево доминаторов для графа потока на рис 13. Во внутреннем представлении LLVM число входов Ф-функций всегда совпадает с числом блоков-предшественников. Таким образом, задача состоит в том, чтобы

определить однозначное соответствие из множества входных базовых блоков, определенных для Ф-функции во множество предшественников базовых блоков.

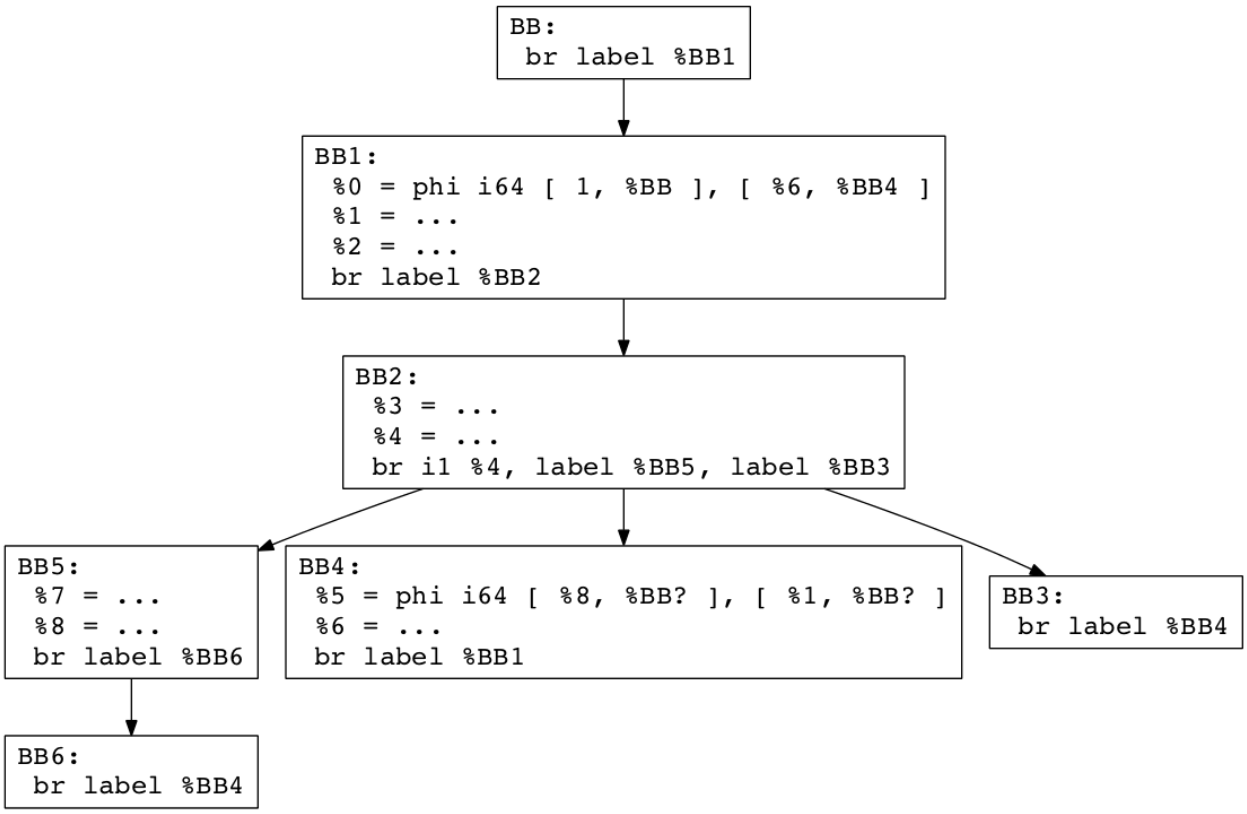


Рисунок 15. Дерево доминаторов для графа потока на рисунке 13

Псевдокод реализованного алгоритма приведен на рис. 16. Если входной базовый блок для Ф-функции является предшественником базового блока, в котором она находится, то очевидно, что он уже определен правильно (как, например, для Ф-функции в блоке BB1). В блоке BB4 оба входа Ф-функции определены неверно. Рассмотрим все отношения доминирования для пар блоков из множества неверно определенных блоков *Wrongs* = {BB1, BB5} над блоками из множества предшественников BB4 – *Preds = {BB6, BB3}*. Поскольку блок BB5 доминирует над единственным блоком из множества Preds (BB6), в искомом отображении именно BB6 будет соответствовать блоку BB5. После исключения этого блока из рассматриваемых, остается лишь один блок, над которым доминирует блок BB1, и в отображении блоку BB1 будет

соответствовать блок BB3. Таким образом, ответ построен. Отметим, что структура графа Hydrogen обеспечивает правильность алгоритма.

```
for all BB ∈ G do
    for all ϕ ∈ BB.phis do
        Preds ← BB.preds
        Wrongs ← Ø
        for all i ∈ Φ.incoming do
            if i ∈ Preds then
                Preds ← Preds \ {i}
            else
                Wrongs ← Wrongs ∪ {i}
            end if
    end for
    while |Wrongs| > 0 do
        if ∃ w ∈ Wrongs : |p ∈ Preds : w dom p}| = 1 then
            ϕ.incoming[j] ← p, где ϕ.incoming[j] = w, w dom p
            Wrongs ← Wrongs \ {w}
            Preds ← Preds \ {p}
        else
            Ошибка: Граф Hydrogen построен неправильно
        end if
    end while
end for
```

Рисунок 16. Псевдокод алгоритма нормализации Ф-функций

Приведем оценку асимптотической сложности алгоритма. Можно считать, что после линейного по числу вершин дерева доминаторов предподсчета, отношение *dom* для двух узлов вычисляется за константное время *O(1)*. Учитывая тот факт, что количество входов Ф-функции на практике мало и можно считать его константой, сложность всего алгоритма составит *O(/I/)*, где *I* — число инструкций в графе.

Алгоритм был реализован в виде дополнительного прохода в инфраструктуре LLVM. LLVM предоставляет удобные средства для объявления анализирующих и трансформирующих проходов над внутренним представлением функций, модулей и отдельных циклов. Также возможно объявить зависимости между проходами. Реализованный проход использует дерево доминаторов, которое может быть вычислено с помощью прохода

DominatorTreePass, уже реализованного в LLVM. Поскольку проход используется для нескольких других анализов, дополнительного времени на создание дерева доминаторов потрачено не будет.

3.1.2 Обеспечение совместимости на уровне бинарных интерфейсов

Для того чтобы возможно было использовать код, скомпилированный с помощью LLVM, в той же манере, что и код, скомпилированный V8, необходимо, чтобы сгенерированные машинные коды двух этих компиляторов были совместимы на бинарном уровне. В первую очередь, это означает, что LLVM должен реализовать такое же соглашение о вызовах, как и V8 (V8 использует собственное соглашение о вызовах). Опишем реализованное в компиляторе V8 соглашение о вызовах для платформы x86_64. Регистр RSI используется для передачи указателя на контекст JavaScript. В регистре RDI передается указатель на вызываемую JavaScript-функцию. Остальные параметры передаются через стек в прямом порядке. Очистка стека и сохранение значений регистров возлагаются на вызываемую функцию. Регистр RBP используется в качестве указателя фрейма. Данное соглашение о вызовах было добавлено в инфраструктуру LLVM в рамках реализации уровня LLV8. В результате была достигнута совместимость кода, созданного LLVM, с кодом компиляторов Full-Codegen и Crankshaft. Стал возможен вызов любого из этих типов кода из любого.

Помимо взаимодействия функций друг с другом, необходимо также учесть взаимодействие сгенерированного кода для JavaScript-функций и виртуальной машины. При некоторых событиях V8 совершает обход стека вызовов JavaScript-функций. Например, обход совершается профилировщиком для выявления горячих функций или при встраивании функций (Глава 1). В компиляторе V8 кадры стека имеют специальную структуру (рис. 17): после адреса возврата сохраняется указатель фрейма вызывающей функции, затем указатель на JavaScript-контекст (содержимое регистра RSI) и указатель на вызываемую функцию (содержимое регистра RDI). Для генерации стековых

кадров такого вида в LLVM были внесены следующие изменения: регистры RSI и RDI были исключены из множества регистров, на которые разрешается распределять значения (подобно регистру указателя стека RSP).

Рисунок 17. Стековый кадр компилятора V8

В LLVM была отключена оптимизация "Frame pointer elimination", которая позволяла адресовать память стека только с помощью указателя стека (RSP), делая регистр RBP свободным для распределения на него значений, что в компиляторе V8 могло привести к ошибке при обходе стека.

В бинарном интерфейсе V8 имеется ещё одна особенность: закрепление за определёнными регистрами значений, которые используются особенно часто (эта технология известна как register pinning). В V8 всего один такой регистр — R13. Этот регистр содержит адрес некоторого объекта в памяти, хранящего различные свойства по известным смещениям, которые могут понадобиться функциям во время выполнения. Регистр R13 также был исключен из множества распределяемых регистров (как и регистры RSI, RDI).

3.1.3 Преобразование представления Hydrogen в LLVM биткод

К разработке уровня *LLV8* был применён инкрементальный подход: количество допустимых операций наращивалось постепенно, делая возможной компиляцию всё более широких подмножеств языка JavaScript. Для

преобразования графа Hydrogen совершается последовательный обход базовых блоков (узлов графа). Для каждого базового блока последовательно посещаются инструкции, составляющие его. При этом каждому базовому блоку ставятся в соответствие базовые блоки представления LLVM. В общем случае их может быть больше одного, так как узлы графа Hydrogen могут содержать разные проверки и ветвления (например, инструкция add может содержать проверку на переполнение). Каждой Hydrogen-инструкции также ставится в соответствие LLVM-значение, что необходимо, поскольку значения (они же инструкции в SSA) могут иметь неограниченное число использований.

Рассмотрим пример преобразования для функции add_int из листинга 3.1.

Листинг 3.1. Функция сложения для двух аргументов

```
function add_int (a, b) {
      return a + b;
}
```

Представление Hydrogen для этой функции приведено на рис. 18. Как видно из рисунка, граф функции содержит только два базовых блока. На рис. 19 представлен граф внутреннего представления LLVM для функции add_int, полученный с помощью реализованного уровня LLV8. LLVM не предоставляет средства для извлечения значений флагов процессора, поскольку уровень абстракции, который предоставляет LLVM, выше процессорных флагов. Для реализации проверки на переполнение во время целочисленных операций используются специальные функции, возвращающие структуру из двух полей: однобитного значения флага и результата операции. В приведенном примере для сложения двух чисел вызывается функция llvm.sadd.with.overflow[3] [54], которая, кроме значения сложения, возвращает значение флага переполнения.

[3] Вызов этой функции всегда встраивается в машинный код.

```
 B0
 v0 BlockEntry Tagged
 t1 Context Tagged
 t2 Parameter 0 Tagged
 t3 Parameter 1 Tagged
 t4 Parameter 2 Tagged
 t5 ArgumentsObject t2 t3 t4 Tagged
 v7 Simulate id=2 var[3] = t1, var[2] = t4, var[1] = t3, var[0] = t2 Tagged
 v8 Goto B1 Tagged

 B1
 v9 BlockEntry Tagged
v10 Simulate id=3 Tagged
v11 StackCheck t1 changes[NewSpacePromotion] Tagged
s18 Change t3 t to s -0? allow-undefined-as-nan TaggedNumber
s19 Change t4 t to s allow-undefined-as-nan TaggedNumber
s13 Add s18 s19 ! TaggedNumber
t20 Change s13 s to t
s21 Constant 2  Smi
v16 Return t20 (pop s21 values) Tagged
```

Рисунок 18. Представление Hydrogen для функции из листинга 3.1

3.1.4 Разработка тестовой системы для обеспечения корректности трансляции

Для обеспечения корректности трансляции конструкций JavaScript в LLVM биткод была разработана система регрессионного тестирования. Система запускает каждый тест с модифицированной версией V8 и сравнивает результат с результатом, полученным с помощью немодифицированной версии. Важно отметить, что компилятор V8 содержит порядка десятка разных опций (флагов), которые контролируют разные свойства. Для обеспечения корректности запуски производятся с разными комбинациями флагов.

На первом этапе разработки писались тесты с функциями, содержащими лишь малое подмножество JavaScript. Далее класс поддерживаемых конструкций постепенно наращивался, и в систему добавлялись новые тесты.

3.2 Обеспечение перехода на уровень компиляции LLV8 во время выполнения программы

В данном разделе описывается предложенный метод для обеспечения перехода выполняемого кода на уровень оптимизации LLVM. В простом случае, когда время выполнения функции небольшое, переключение на уровень оптимизирующего компилятора происходит с помощью замены адреса функции адресом оптимизированного кода. Это возможно, так как сгенерированные коды реализуют одинаковый бинарный интерфейс. Однако, в случае, если в функции содержится цикл или несколько циклов с большим количеством итераций, переключение на уровень оптимизирующего компилятора необходимо организовать во время выполнения функции (OSR entry).

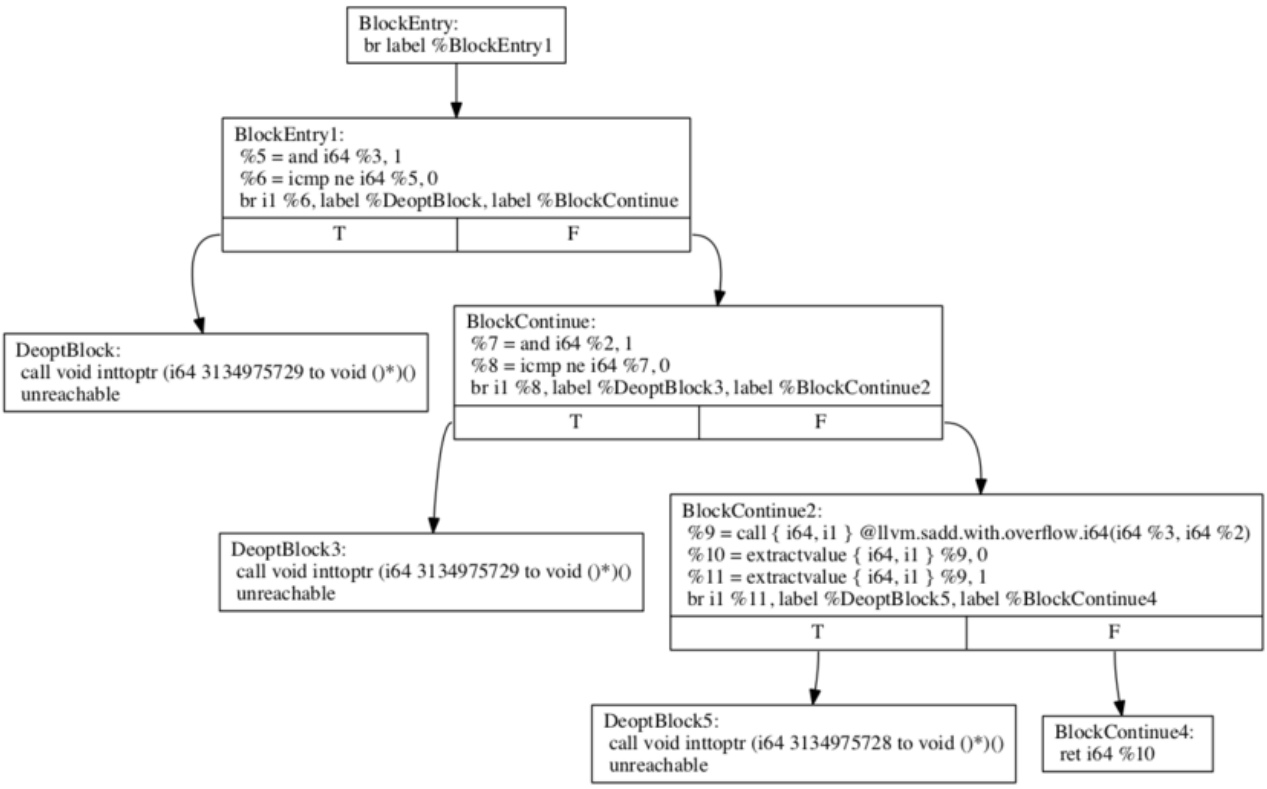


Рисунок 19. LLVM биткод для графа, приведенного на рисунке 18

При генерации машинного кода в начале каждой функции вставляется специальный код, так называемый пролог функции. В архитектуре X86 [55] в прологе функции производятся следующие действия: сохраняется значение регистра EBP (для последующего использования в вызывающей функции).

Значение регистра EBP вставляется равным значению регистра ESP. Регистр EBP сохраняет свое значение на протяжении всей функции и далее используется для доступа к локальным переменным и аргументам функции. Значение регистра RSP уменьшается на количество локальных переменных функции, которые были определены в память во время распределения регистров. Также производится сохранение значений разных регистров в соответствии с принятым соглашением о вызове. В конце кода функции вставляется эпилог, в котором восстанавливаются значения регистров RBP и RSP (рис. 20).

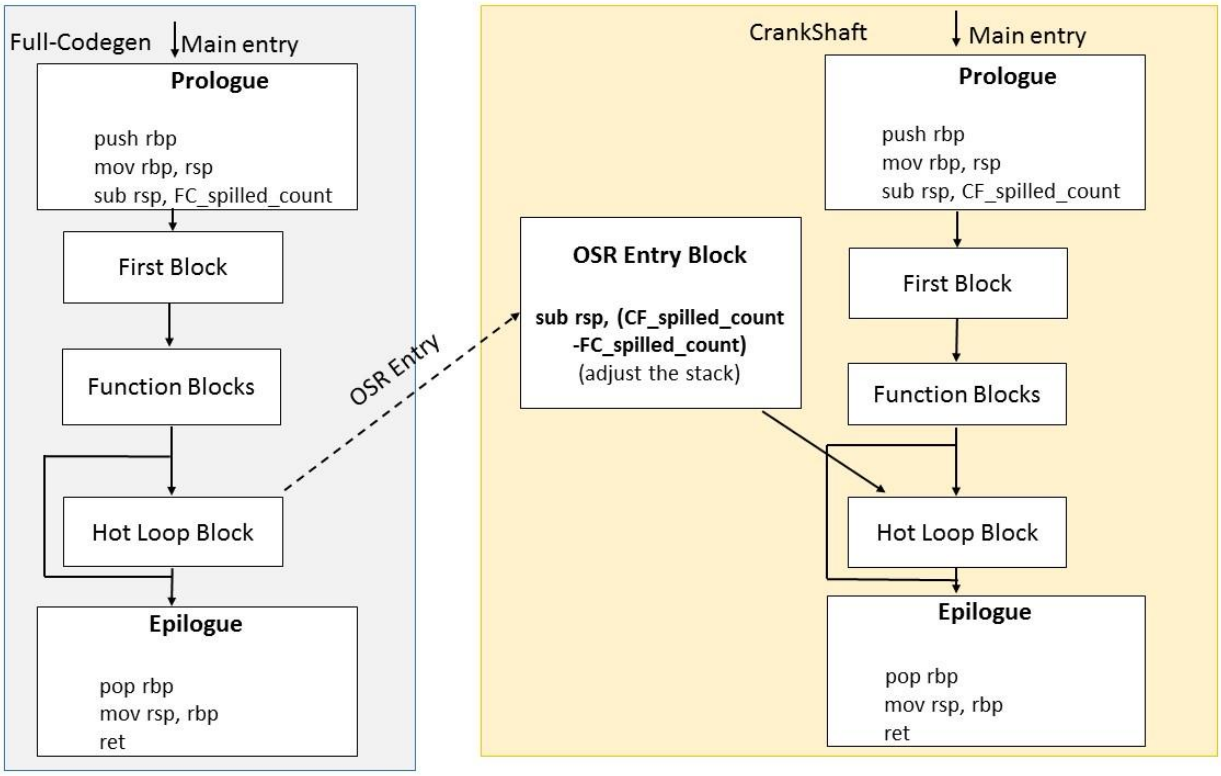


Рисунок 20. Схема перехода между компиляторами Full-Codegen и Crankshaft

Для организации переключения между уровнями Full-Codegen и Crankshaft используется простой безусловный переход на начало соответствующего цикла в сгенерированном Crankshaft-ом коде (рис. 20 Поскольку управление передается сразу в тело функции, то при переключении на оптимизированный код необходимо также выполнить адаптацию стека. Например, значение указателя стека необходимо уменьшить на количество

локальных переменных, которые были распределены в слоты стека во время распределения регистров в оптимизирующем уровне. При этом надо учитывать, что прежний уровень, в свою очередь, уже уменьшил это значение на количество своих локальных переменных. Использование такого подхода для переключения на LLVM код нецелесообразно, так как в LLVM предполагается, что каждая функция может содержать только одну точку входа.

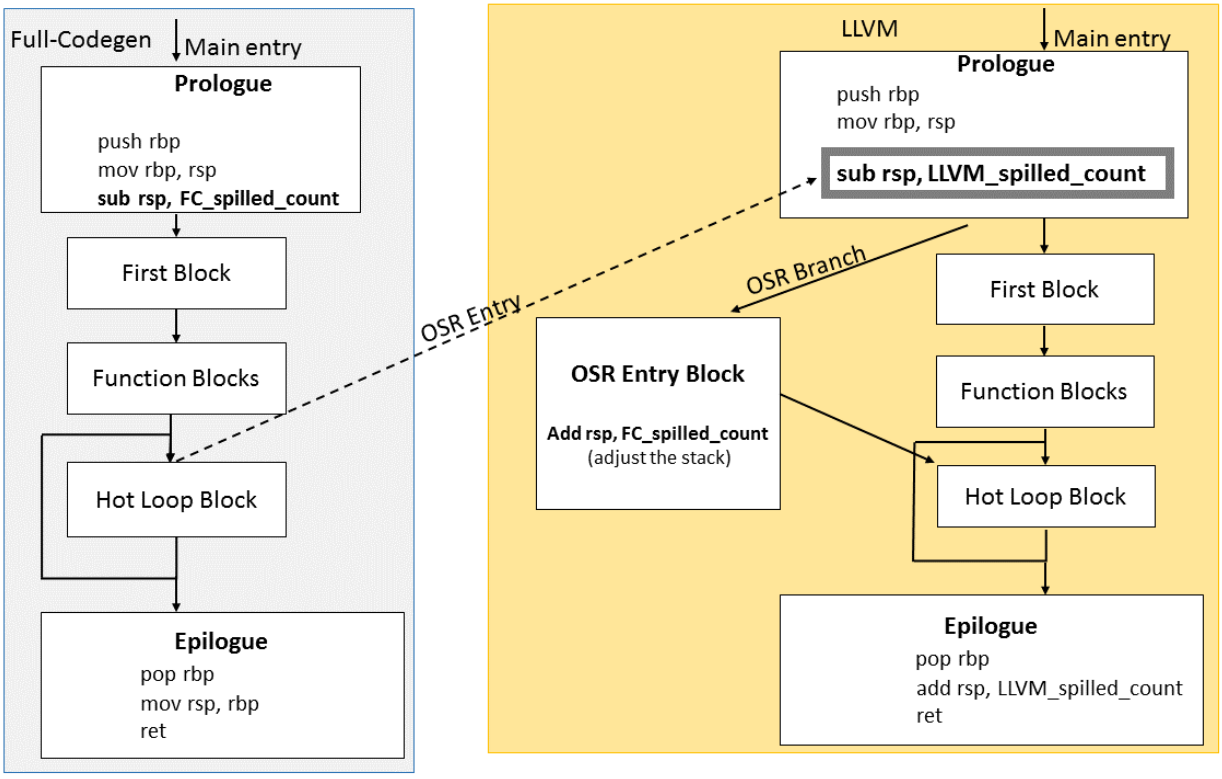


Рисунок 21. Схема перехода между компиляторами Full-Codegen и LLVM

Нарушение этого условия может стать причиной ограничения выполняемых в LLVM оптимизаций. В компиляторе JavaScriptCore для организации переключения на уровень LLVM создаются копии функций для каждого цикла (возможного входа) [56]. Все аргументы и локальные переменные передаются через глобальный буфер, а для переключения на уровень LLVM во время выполнения вызывается соответствующая копия функции. Недостатком такого решения являются большие затраты при создании всех копий функции. Нами был разработан и реализован другой метод, который позволяет осуществить переход во время выполнения на LLVM код без создания дополнительных

копий функции. Основная проблема при переключении заключается во том, что при использовании внешнего компилятора LLVM информация о количестве локальных переменных, которые были распределены в слоты стека, недоступна (эта информация становится доступна гораздо позже, при распределении регистров LLVM). Для решения проблемы при переходе через цикл точка входа была перемещена в пролог кода функции, где значение указателя стека было уменьшено на количество локальных переменных, хранящихся в стеке. После этого с помощью дополнительного аргумента решается, откуда именно будет выполняться функция. Если управление перешло на LLVM код через цикл, то необходимо выполнить дополнительные действия для адаптации стека. В частности, надо учитывать тот факт, что прежний уровень мог, в свою очередь, уменьшить значение указателя стека (рис. 21). Предложенный подход может привести к генерации двух точек входа для LLVM функции, однако эти точки будут находиться в прологе кода и не будут являться ограничением для выполнения оптимизаций LLVM [16].

3.3 Поддержка динамических свойств языка JavaScript на уровне выполнения LLV8

3.3.1 Поддержка спекулятивной компиляции на уровне выполнения LLV8

Одним из важных компонентов для обеспечения быстродействия современных динамических компиляторов является поддержка спекулятивной компиляции [2] [57] [58]. 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 относится предсказание типов и оптимизация обращений к полям и методам объектов. Поддержка спекулятивной компиляции, в первую очередь, означает реализацию поддержки деоптимизаций (обратных переходов на неоптимизирующий уровень). Для правильного возобновления состояния виртуальной машины Full-Codegen, в Crankshaft используется специальная инструкция simulate. Эта инструкция не

переводится в машинный код, а только симулирует и сохраняет текущее состояние стековой машины Full-Codegen в данной точке программы. При деоптимизации используется информация ближайшей доминирующей инструкции simulate для возобновления работы Full-Codegen. Для организации деоптимизации необходимо иметь информацию о том, куда именно отображены локальные переменные функции (конкретная позиция в стеке, адрес в куче или конкретный регистр), для передачи этих значений на неоптимизирующий уровень выполнения. Перекладывая генерацию машинного кода на сторонний компилятор LLVM, мы теряем контроль над ней, и информация об отображении локальных переменных в физическую память машины становится недоступной. 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. Для модификации сгенерированного кода на лету используются функции *llvm.experimental.stackmap* и *llvm.experimental.patchpoint* [59]. Для возможности использования информации необходимой для организации деоптимизаций используется структура *Stack Map* [59]. Эта структура создается при компиляции функций *llvm.experimental.stackmap* и *llvm.experimental.patchpoint* и используется для хранения местонахождений (слот стека, имя регистра, константа и т.п.) всех значений, переданных этим функциям в качестве аргументов. Для поддержки деоптимизаций в точках проверки условий спекулятивных оптимизаций 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 (инструкции simulate). При кодогенерации LLVM помещает информацию обо всех *stackmap* структурах в специальный сегмент кода (*stackmap section*). Далее, при деоптимизации всю необходимую информацию о живых переменных

можно извлечь из сегмента *stackmap*. Формат сегмента *stackmap* приведен на рис. 22.

3.3.2 Поддержка сборщика мусора и перемещений объектов на уровне выполнения LLV8

Объекты в машинном коде могут иметь абсолютные адреса. После перемещении объектов необходимо заменить их старые адреса. Кроме того, компилятор V8 хранит созданный машинный код в куче в виде обыкновенного объекта. Следовательно, во время сборки мусора сгенерированный код также может быть перемещен. Однако в коде могут содержатся вызовы по абсолютным адресам, которые также необходимо заменить. Для реализации такой поддержки используется функция *llvm.experimental.patchpoint*. Вызов функции *llvm.experimental.patchpoint*, помимо тех же параметров, что и *llvm.experimental.stackmap*, принимает также вызываемую функцию. Кроме создания *stackmap*, при компиляции *llvm.experimental.patchpoint* в генерируемый код вставляется вызов этой функции в соответствии с заданным соглашением о вызовах. В структуре *stackmap* сохраняется смещение инструкции вызова (instruction offset) относительно начала кода, и при необходимости адрес вызываемой функции может быть заменен новым.

Для обеспечения правильности работы сборщика мусора необходимо реализовать также поддержку механизма *safepoint* (Глава 1). Для этого, кроме информации о местонахождении переменных, необходимо также обладать информацией о всех указателях, которые будут живы в точке *safepoint*. Для получения информации об интервалах жизни переменных (LLVM-значений), накрывающих определенную точку, были задействованы анализирующие проходы, уже реализованные в LLVM [60]. Кроме того, соответствующие преобразующие проходы LLVM были использованы для автоматической расстановки точек safepoints. Для генерации структуры stackmap для точек safepoint была применена функция llvm.experimental.gc.statepoint [61].

```
Header {
  uint8  : Stack Map Version
  uint8  : Reserved (expected to be 0)
  uint16 : Reserved (expected to be 0)
}
uint32 : NumFunctions
uint32 : NumConstants
uint32 : NumRecords
StkSizeRecord[NumFunctions] {
  uint64 : Function Address
  uint64 : Stack Size
}
Constants[NumConstants] {
  uint64 : LargeConstant
}
StkMapRecord[NumRecords] {
  uint64 : PatchPoint ID
  uint32 : Instruction Offset
  uint16 : Reserved (record flags)
  uint16 : NumLocations
  Location[NumLocations] {
    uint8  : Register | Direct | Indirect | Constant | ConstantIndex
    uint8  : Reserved (location flags)
    uint16 : Dwarf RegNum
    int32  : Offset or SmallConstant
  }
  uint16 : Padding
  uint16 : NumLiveOuts
  LiveOuts[NumLiveOuts]
    uint16 : Dwarf RegNum
    uint8  : Reserved
    uint8  : Size in Bytes
  }
  uint32 : Padding (only if required to align to 8 byte)
}
```

Рисунок 22. Формат сегмента stackmap

3.4 Выводы и результаты

Был разработан и реализован метод динамической компиляции программ на языке JavaScript в статически типизированное внутреннее представление компиляторной инфраструктуры LLVM. При этом была добавлена поддержка компиляции всех конструкций и динамических свойств языка JavaScript:

- сборщика мусора и перемещений объектов;

- бинарной совместимости и переключении между разными уровнями компиляции;
- спекулятивной компиляции и деоптимизации.

Уровень LLV8 позволяет применять уже имеющиеся в LLVM оптимизации к программам, написанным на языке JavaScript. Приведем сравнение сгенерированного уровнем LLV8 кода для теста bitops-bits-in-byte (листинг 3.2), с кодом, сгенерированным уровнем Crankshaft. Как видно из рис. 23, код, сгенерированный уровнем LLV8, не имеет цикла, тогда как код, сгенерированный Crankshaft, содержит цикл (рис. 24). Причиной этому служит реализованная в LLVM оптимизация “разворачивание циклов” [62], которая отсутствует в компиляторе Crankshaft. На данном тесте было достигнуто существенное ускорение за счет компиляции функции “foo” компилятором LLVM.

Листинг 3.2. Исходный код теста bitops-bits-in-byte из набора SunSpider

```
function foo(b) {
    var m = 1, c = 0;
    while(m<0x100) {
    if (b & m) c++;
      m <<= 1;
    }
  return c;
}
function TimeFunc(func) {
      var x, y, t;
      var sum = 0;
      for (var x=0; x<350; x++)
      for (var y=0; y<256; y++) sum += func(y);
      return sum;
}
result = TimeFunc(foo);
```

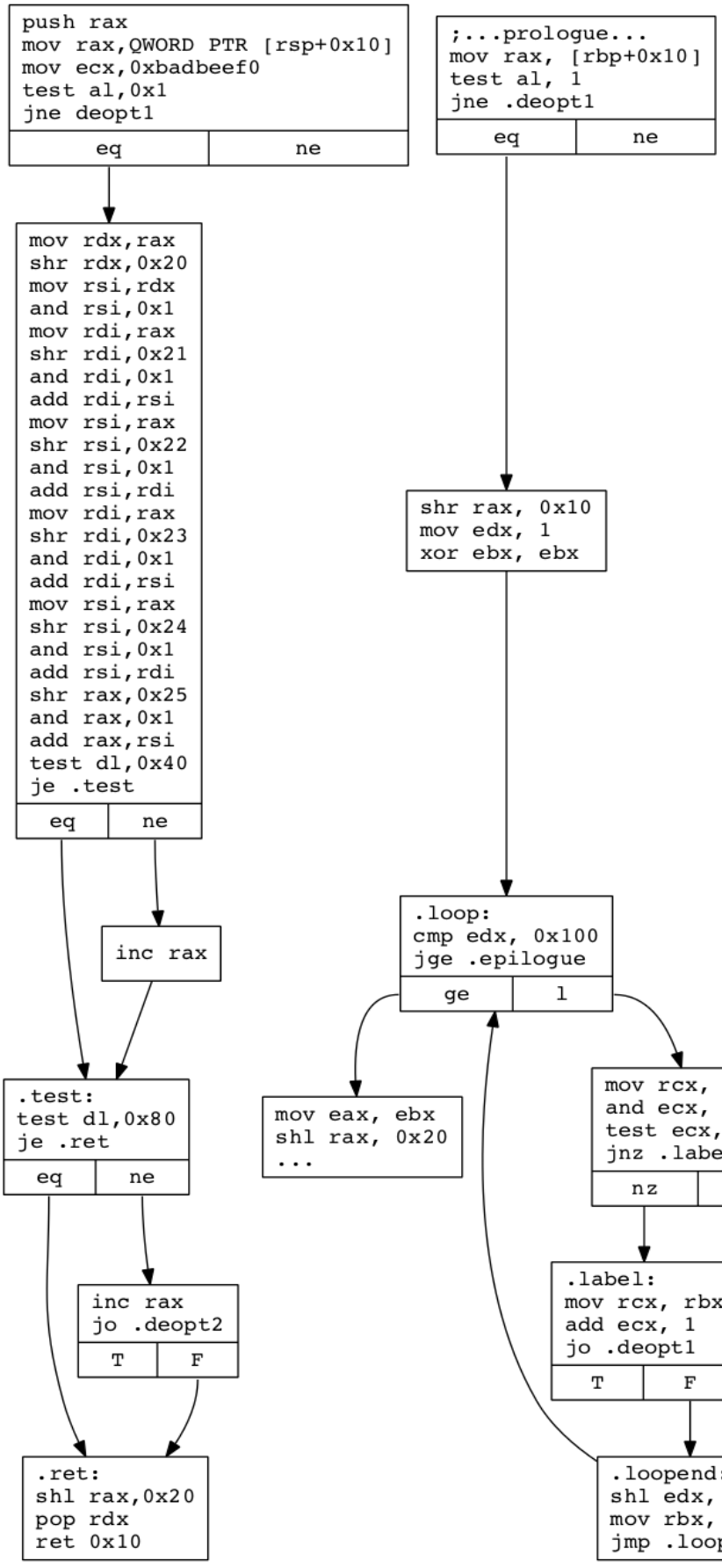


Рисунок 23. Код, генерируемый уровнем LLV8 для функции foo из листинга 3.2

Рисунок 24. Код, генерируемый уровнем Crankshaft для функции foo из листинга 3.2

В таблице 3 приведен результат сравнения производительности уровней Cranksaft и LLV8 для теста bitops-bits-in-bytes. Как видно из таблицы, при увеличении числа итераций (времени выполнения машинного кода) увеличивается и выигрыш в производительности, обеспечиваемый более оптимальным машинным кодом уровня LLV8 по сравнению с Crankshaft.

Таблица 3. Сравнение производительности уровней Crankshaft и LLV8 на тесте 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. LLV8, мс.	27	74	538
Ускорение, число раз	1.37	2.92	3.81

В таблице 4 приведены результаты сравнения производительности на тестовом наборе LongSpider. Выполнение теста 3d-cube на уровне LLV8 позволяет ускорить этот тест на 62%, тест access-nsieve ускорился на 17.3%.

Таблица 4. Сравнение производительности уровней Crankshaft и LLV8 на наборе LongSpider (на остальных тестах из набора изменений в производительности замечено не было)

Тест	**Производительность Crankshaft, мс**	**Производительность LLV8, мс**	**Улучшение, %**
3d-cube	740	280	62.1
access-binary-tree	1580	1400	11.3
access-nsieve	750	620	17.3
bitops-bits-in-byte	37	27	27
math-partial-sum	1004	990	1.4
math-spectral-norm	790	720	8.8

В дальнейшем планируется продолжать улучшение уровня LLV8, в частности, планируется разработка оптимизаций специально для модуля asm.js, улучшение генерируемого на уровне LLV8 биткода LLVM. Также планируется разработка оптимизационных проходов в инфраструктуре в LLVM,

учитывающих особенности биткода, получаемого для программ на языке JavaScript, разработка оптимизаций для векторизации циклов, разработка предсказания переходов и т.д.

Глава 4. Оптимизации

В данной главе описываются разработанные и реализованные методы машинно-зависимой и машинно-независимой оптимизации в многоуровневых динамических компиляторах JavaScriptCore и V8.

4.1 Машинно-зависимые оптимизации

Язык JavaScript является мультиплатформенным. При реализации мультиплатформенных систем разработчиками часто не учитываются особенности конкретных архитектур. С другой стороны, в связи с ростом популярности мобильных и встраиваемых систем, в настоящее время особенно остро стоит задача оптимизации программ для архитектур, используемых в этих системах. Это позволяет не только улучшить производительность программ, но и характеристики энергопотребления. Одной из самых известных архитектур для мобильных и встраиваемых систем является платформа ARM.

4.1.1 Оптимизация доступа к глобальным переменным для платформы ARM

Данная оптимизация позволяет исключить излишнее обращение к памяти для каждой пары чтения/записи глобальных переменных. По итогам исследования сгенерированного машинного кода компилятора V8 для архитектуры ARM было обнаружено, что каждая операция чтения/записи глобальных переменных раскрывается на две машинные инструкции чтения/записи из памяти. Более того, так как все глобальные переменные хранятся в одном и том же глобальном объекте, то первое обращение к памяти совпадает для каждой пары чтения и записи глобальных переменных. Если такая операция присутствует в цикле, то первая ее часть будет инвариантом цикла (рис. 25).

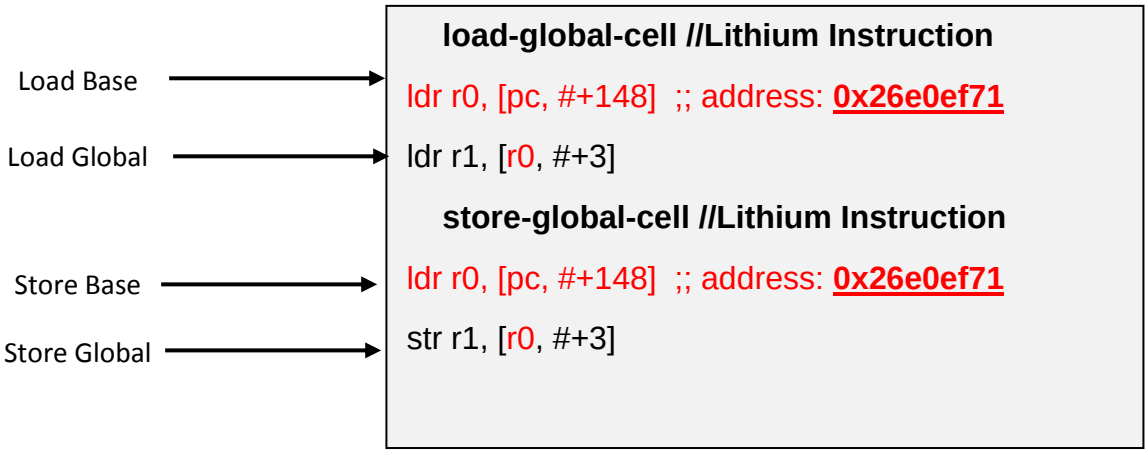


Рисунок 25. Машинный код для инструкций обращения к глобальным переменным для платформы ARM

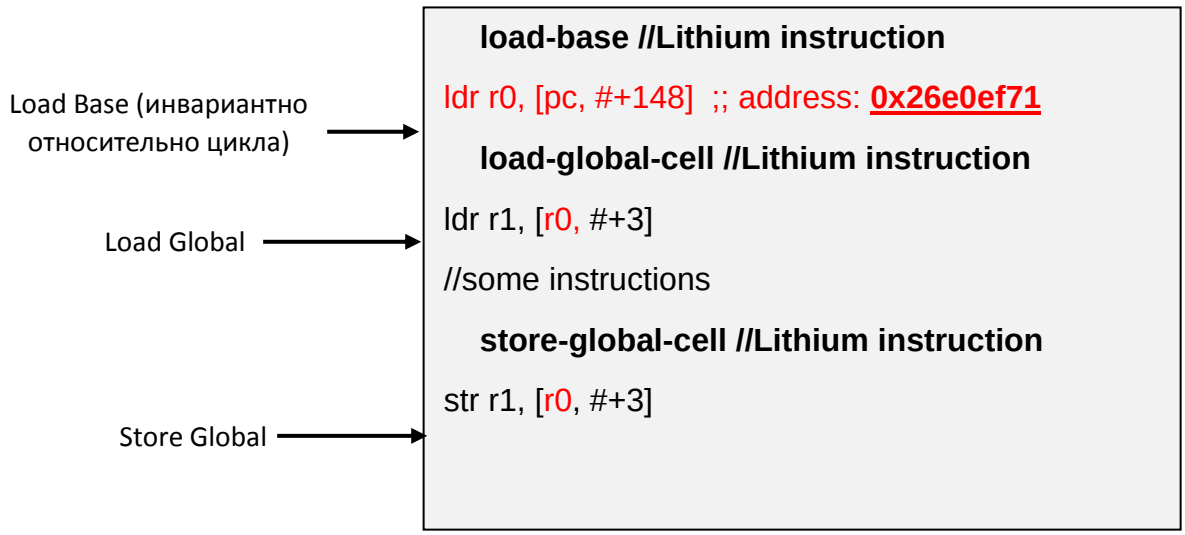


Рисунок 26. Модифицированный машинный код для инструкций обращения к глобальным переменным для платформы ARM

Был реализован метод, который позволяет разделить реализацию обращения к глобальным переменным на две разные инструкции в графе управления (рис. 26). Данный метод позволяет организовать вынос инвариантного обращения к памяти за пределы циклов, а также сократить количество таких обращений для каждой пары чтения/записи глобальных переменных. При реализации такого подхода важно учесть, что при выносе части инструкции за цикл также увеличивается давление регистров (количество одновременно живых переменных) в данном участке кода, что может привести к генерации дополнительных обращений к памяти в цикле. Был реализован анализ, позволяющий оценивать возможное давление регистров. Такая оценка

позволяет ограничивать число вынесенных за цикл инструкций и избегать излишних обращений к памяти.

4.1.2 Эффективная реализация некоторых операций с учетом специфики архитектуры ARM

В компиляторе V8 при реализации некоторых операций создавалось множество инструкций обращения к последовательным адресам памяти (*LDR/STR*). Однако архитектура ARM поддерживает команды, которые более эффективно осуществляют множественную загрузку/сохранение по последовательным адресам - *LDM/STM* [63]. С использованием инструкций LDM/STM была добавлена более эффективная реализация для таких операций.

Кроме того, операция сравнения (*cmp*) в условных предикатах вида *a&b* была заменена на более быструю инструкцию *test*, а умножение вещественного числа на -1 было реализовано с помощью более эффективной инструкции *vneg*.

Архитектура ARM также поддерживает команды для эффективного вычисления последовательных операций умножения и сложения/вычитания – *MLA/MLS* [63] (англ. *multiply-accumulate/ multiply-subtract*). Нами была добавлена поддержка этих команд на оптимизирующем уровне DFG JIT компилятора JavaScriptCore. При реализации поддержки инструкций *MLA* (*MLS*) были учтены возможные проблемы, связанные с переходами между уровнями оптимизаций. Например, между инструкциями умножения и сложения может произойти деоптимизация, которая в дальнейшем может привести к потере значения умножения на неоптимизирующих уровнях.

4.2 Машинно-независимые оптимизации

4.2.1 Улучшение оптимизации удаления мертвого кода

На оптимизирующем уровне DFG JIT компилятора JavaScriptCore реализована оптимизация удаления мёртвого кода, однако её эффективность невысока. В частности, в процессе проведения данной оптимизации ни один из вызовов функций не будет удалён, даже если удаление таких вызовов возможно

без ущерба для семантики программного кода. Удаление рассматриваемого оператора безопасно, если для него выполняются следующие два условия:

- результат не используется ни в одном из участков кода;
- оператор не имеет побочных эффектов (не изменяет состояние программы и не обращается к другим функциям с побочными эффектами и к системным вызовам).

Проверка первого условия может быть сведена к применению достаточно простого графового алгоритма, и уже была реализована в компиляторе DFG JIT. Проверка же второго условия требует привлечения семантики используемой системы команд, и здесь возникает некоторая проблема с операторами вызова функций. В компиляторе DFG JIT нет реализации межпроцедурного анализа, поэтому все вызовы функций рассматриваются как обладающие побочным эффектом и не удаляются во время оптимизации удаления мертвого кода. Однако для некоторого класса функций всё же возможно доказать отсутствие побочных эффектов при выполнении дополнительных условий.

В стандарте JavaScript не поддерживаются атрибуты функций (например, в компиляторе gcc [64] можно объявить функцию без побочных эффектов с атрибутом __pure__), однако в стандарте содержится описание всех библиотечных методов строк, чисел и прочих встроенных объектов, по которым можно восстановить информацию об их побочных эффектах. Среди этих стандартных функций можно выделить только функции без побочных эффектов, но такой подход довольно груб. Однозначно “чистых” функций немного: большинство функций требуют, чтобы передаваемые аргументы удовлетворяли определённым налагаемым условиям. Рассмотрим пример из листинга 4.1. Из стандарта языка JavaScript следует, что функции replace, match и search могут изменить свойство lastIndex, передаваемого им регулярного выражения. Следовательно, для удаления этих вызовов достаточно только убедиться, что результат самой функции и переданное ей регулярное

выражение больше нигде не используются. Так, вызовы функций replace, match и search могут быть эффективно удалены из функции *foo1*. Удаление вызовов из функции *foo2* приведет к ошибке, так как переменная *re* используется при дальнейших вызовах.

Листинг 4.1. Вызов функций с побочным эффектом

```
function foo1 (string_arg) {
    string_arg.replace(/fo+/, “bar”);
    string_arg.match(/fo+/, “bar”);
    string_arg.search(/fo+/, “bar”)
}
function foo2 (string_arg) {
    var re = /fo+/;
    string_arg.replace(re, “bar”);
    string_arg.match(re, “bar”);
    string_arg.search(re, “bar”)
}
```

Для разных функций может понадобиться проверка разных утверждений, поэтому наиболее универсальным и надёжным механизмом является оформление процедур проверки в виде функций-обработчиков и оперирование указателями на эти функции. Для этого во время компиляции JavaScriptCore из специальным образом отформатированных комментариев генерируется хеш-таблица, ключом которой является адрес функции, а значением – указатель на соответствующую функцию-обработчик. Все 152 обработанные функции были распределены на 8 классов по предъявленным ими условиям (таблица 5). Процесс удаления вызовов стартует в фазе свертки констант (constant folding). Рассматривается каждая инструкция Call, и если с ней ассоциирован адрес статической функции, то в хеш-таблице по соответствующему ключу отыскивается нужный обработчик. Если обработчик найден и вернул истинное

значение, вызов помечается как не имущий побочных эффектов. Далее инструкция удаляется, если ее результат не используется ни в одном из участков кода. Полный список функций и соответствующих обработчиков приведен в приложении А.

Таблица 5. Классы обработчиков для библиотечных функций языка JavaScript

#	Описание	Обработчик	Число функций
1	Функция не имеет побочных эффектов — заменять всегда	looseHandler	61
2	Функция всегда имеет побочные эффекты — никогда не заменять	prohibitionHandler	46
3	Аргументы должны иметь примитивные типы	primitiveHandler	36
4	Аргументы имеют примитивные типы или являются одноразовыми регулярными выражениями	primitiveOrOneTimeRegexpHandler	3
5	Второй аргумент должен иметь примитивный тип	primitiveSecondHandler	2
6	Все элементы массива должны иметь примитивные типы	arrayOfPrimitivesHandler	2
7	И элементы массива, и аргументы должны иметь примитивный тип	arrayPrimitiveArgumentsHandler	1
8	Аргументов быть не должно	noArgumentsHandler	1

Нами была реализована полная поддержка удаления вызовов библиотечных функций без побочных эффектов, что привело к значительному росту производительности на тестовых наборах языка JavaScript.

4.2.2 Улучшение математических и строковых функций.

На уровне DFG JIT компилятора JavaScriptCore математические и строковые функции были реализованы с помощью вызовов JavaScript-API. Однако некоторые из этих функций можно реализовать непосредственно с помощью внедрения машинного кода. В итоге, посредством внедрения машинного кода вместо вызова с использованием JavaScript-API было

реализовано ускоренное выполнение математических функций Math.floor, Math.log, Math.exp и выполнение строковых операций String.fromCharcode, String.CharCodeAt для архитектур ARM и X86. Кроме того, были реализованы разные улучшения библиотечных функций. Был реализован метод, кэширующий значения функций String.replace, Math.sin и Math.cos. В компиляторе V8 был реализован улучшенный алгоритм сортировки массивов в библиотечной функции Array.Sort, учитывающий размер данных при сортировке. Для массивов больших размеров (больше 350k элементов) был реализован быстрый алгоритм сортировки “dual pivot quicksort” [65].

4.2.3 Улучшение встраивания функций

В реализации компилятора V8 функция является кандидатом встраивания, если соответствует некоторым критериям. Одним из таких критериев являлся размер исходного кода функции. Однако исходный код может содержать множество комментариев, что не учитывалось при оценке размера функции. Нами была разработана другая метрика, основанная на размере сгенерированного машинного кода. Также была предложена метрика, учитывающая такие факторы, как количество локальных переменных и аргументов вызываемой и вызывающей функций. Использование данной метрики позволило избежать многократных обращений к памяти из-за высокого давления регистров и добиться улучшения производительности компилятора.

Тип инструкции на уровне Crankshaft определяется, исходя из типов ее аргументов и из типов инструкций, в которых она используется. При встраивании функций такой подход может привести к неправильному определению типов и к генерации множественных инструкций конвертации типов. Рассмотрим пример программы на рис. 27.

```
1.  function g(z) {
2.    return z/6;
3.  }

4.  function f(arg) {
5.   var x =10.5;
6.   var y = 0;
7.   for (var i = 0; i < 5; ++i )
8.     g(x)
9.   for (var i = 0; i < 5; ++i )
10.    g(y)
11.  y = y + arg;
12.  return y << 5;
13. }
14. f(10);
```

27.a

```
4. function f(arg) {
5.   var x =10.5;
6.   var y = 0;
7.   for (var i = 0; i < 5; ++i )
8.     x  = x/6;
9.   for (var i = 0; i < 5; ++i )
10.    y = y/6;
11.  y = y + arg;
12.  return y << 5;
}
```

27.б

Рисунок 27. Пример программы с неправильно определенными типами переменных

Функция *g* вызывается с аргументом типа double в инструкции *8*, следовательно, тип инструкции *3* (z/6) определяется как double. После встраивания тела функции g в функцию f, тип переменной *y* будет вычисляться из инструкций *6, 10, 11* и *12* (рис. 27. б). Однако, если тип инструкции *10* (z/6) был определен как double, тип переменной *y* ошибочно будет определяться как *double* (хотя *y* всегда имеет целое значение). Это приведет к генерации конвертации типов (*double -> int*) каждый раз во время выполнения операции *12*. Был реализован новый метод, который учитывает встраивание функций при распространении информации о типах: если функция вызывается с разными типами аргументов, то после встраивания этой функции информация о типах ее переменных не учитывается при распространении типов.

4.2.4 Алгоритм глобального распределения регистров для оптимизирующего уровня компилятора JavaScriptCore

Одной из важнейших технологий при конструировании оптимизирующих компиляторов является распределение регистров. Эффективное распределение регистров в оптимизирующем компиляторе позволяет минимизировать количество создаваемых инструкций обращений к памяти, тем самым улучшая качество сгенерированного кода. В результате исследований компилятора JSC было выяснено, что в оптимизирующем компиляторе DFG JIT реализовано только локальное распределение регистров. Это приводило к тому, что все локальные переменные записывались в память при выходе из базовых блоков и читались из памяти при входе в новый блок. Такой подход обеспечивал сохранение правильных значений локальных переменных в памяти при переходе между разными уровнями оптимизации, однако приводил к множественным обращениям к памяти. Существует множество алгоритмов для глобального распределения регистров [66] [67] [37]. Классическим методом является распределение регистров с помощью раскраски графа [67]. Этот подход требует создания и поддержки графа несовместимости переменных, имеет большую вычислительную сложность и в основном используется в статических компиляторах (например, в gcc). В итоге в компиляторе DFG JIT был реализован алгоритм линейного сканирования [37] для глобального распределения регистров. Алгоритм позволяет распределять регистры за линейный проход по количеству живых переменных и широко используется в динамических компиляторах. Реализация глобального алгоритма распределения регистров позволяет избежать излишних обращений к памяти в пределах базовых блоков.

При реализации глобального алгоритма были учтены особенности многоуровневых компиляторов. В частности, выполнение программы может переключаться на оптимизирующий уровень во время выполнения циклов (OSR entry). В таком случае, алгоритм распределения регистров должен загружать все локальные переменные до начала цикла. Для решения проблемы в промежуточное представление был добавлен новый базовый блок перед

началом каждого цикла, в котором записываются все необходимые инструкции для загрузки локальных переменных, а точка входа в компилятор DFG во время перехода через цикл была перемещена в новый базовый блок (рис. 28).

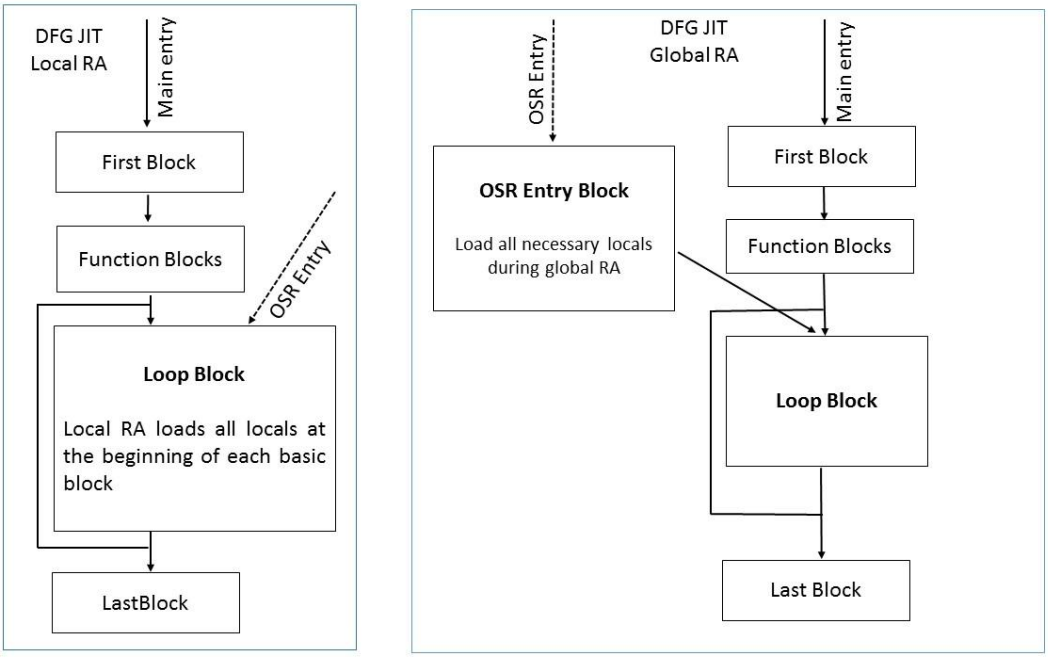


Рисунок 28. Модификация промежуточного представления DFG JIT для обеспечения глобального распределения регистров

Другой особенностью компилятора DFG JIT является то, что для 32-битных архитектур тип и значение переменных хранятся в двух разных регистрах. Алгоритм линейного сканирования был модифицирован так, чтобы возможно было присвоение двух разных регистров одной и той же переменной.

4.2.5 Жадный алгоритм линейного сканирования на оптимизирующем уровне компилятора V8

На оптимизирующем уровне компилятора V8 уже был реализован алгоритм линейного сканирования для глобального распределения регистров. Для улучшения качества создаваемого на этом уровне машинного кода нами был реализован жадный алгоритм линейного сканирования (англ. greedy backtracking register allocation, GBRA) [49], который также используется в компиляторах LLVM и SpiderMonkey.

Жадный алгоритм распределения регистров почти всегда генерирует более эффективный машинный код, нежели обычный алгоритм линейного сканирования, но использование такой дорогостоящей и сложной оптимизации иногда приводит к увеличению времени работы компилятора, что может отрицательно сказаться на общем времени выполнения небольших программ. Нами был предложен метод, который позволяет выбрать между двумя алгоритмами распределения регистров, основываясь на размере функции или собранной статической информации о программе (Глава 2).

4.2.6 Алгоритм рематериализации регистров

Во время распределения регистров, если одновременно живых переменных больше, чем доступных регистров, алгоритм должен распределить некоторые значения в память, а затем загружать их в регистры по мере использования. Но есть и второй способ: значения могут быть вычислены заново, если все нужные операнды доступны в данной точке программы. Данная технология называется “прямой рематериализацией регистров”. Более того, значение некоторой переменной V может быть получено из других переменных $\{w_i\}$, которые были вычислены с использованием переменной V. Эта технология известна как “обратная рематериализация регистров”. Рассмотрим пример рематериализации регистров (рис. 29). Предположим, что имеются четыре регистра (r0-r3). Регистр r4 используется для хранения загруженных из памяти значений.

Значение переменной **c** можно вычислить из переменных **a** и **b**, так как эти переменные живы в точках использования **c**. Как видно из рисунков 29б и 29в, при помощи рематериализации регистров два обращения к памяти можно заменить на одну операцию сложения. Технология рематериализации регистров используется в некоторых крупных компиляторах, таких как GCC и LLVM. Тем не менее, статические компиляторы не ограничены по времени компиляции, тогда как JIT компиляторы должны идти на компромисс между генерацией качественного машинного кода и временем компиляции. Многие известные алгоритмы рематериализации регистров требуют наличия структур для

определения повторно используемых регистров (англ. register resue chain) для нахождения пригодных для рематериализации значений [68] [69].

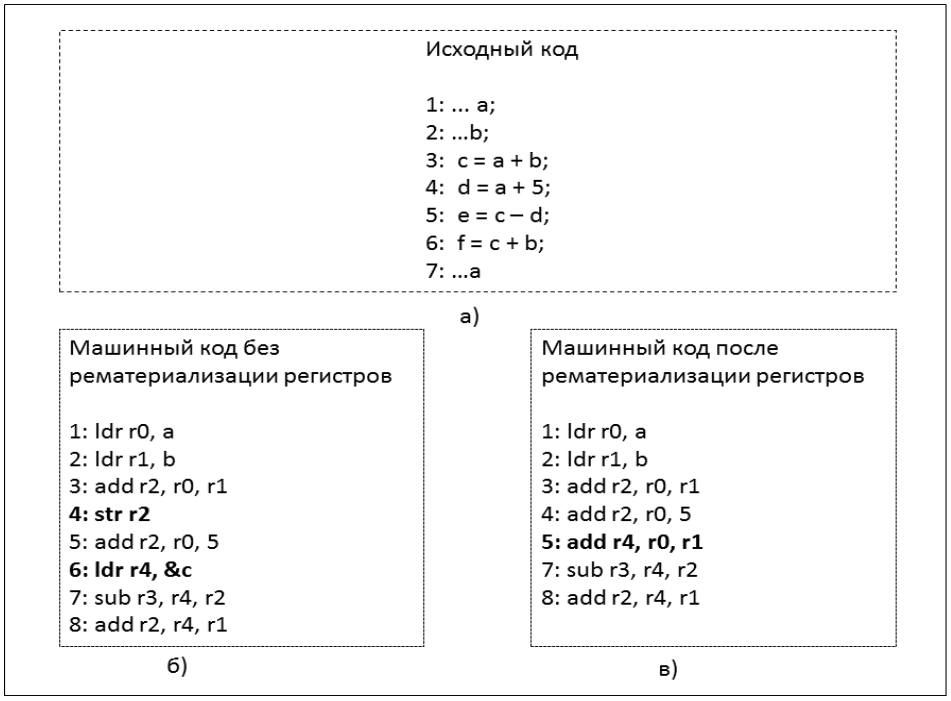


Рисунок 29. Пример применения метода рематериализации регистров

Однако оптимизирующий уровень Crankshaft не поддерживает такие структуры. Конструирование таких структур могло бы привести к большим задержкам времени компиляции, что привело бы к ухудшению производительности компилятора в целом. Чтобы реализовать технологию рематериализации без негативного влияния на время компиляции, нами был предложен метод, который использует имеющиеся в V8 структуры и интегрирован в алгоритм линейного сканирования распределения регистров.

При принятии решения о рематериализации некоторого значения важно учесть, будет ли вычисление этого значения выполнятся быстрее чем операция загрузки из памяти. Например, для некоторых платформ ARM более выгодно загрузить значение из памяти, чем вычислить его заново с помощью операции умножения (если это значение находится в кэше L1). В текущей реализации рематериализация поддерживается для операций сложения, вычитания, битовых операций и операций сдвига. Операции умножения и деления

поддерживаются частично, в случаях, когда они могут быть представлены с помощью операций сдвига. Операнд инструкций сложения и вычитания может быть вычислен с использованием обратной рематериализации, если доступны значения результата и второго операнда. Однако это правило не распространяется на все бинарные операции. Например, для операции умножения, результат которой равен нулю, необходимо сохранять дополнительную информацию, для того чтобы вычислить, какой из операндов был равен нулю. В текущей реализации обратная рематериализация поддерживается только для операций сложения и вычитания. Реализованный алгоритм рематериализации состоит из пяти этапов.

На первом этапе выявляются все переменные, которые могут быть вычислены с помощью рематериализации. В представлении Lithium отсутствует зависимость по данным между инструкциями. Нами был модифицирован модуль трансляции представления Hydrogen в представление Lithium для присваивания каждой Lithium-инструкции информации о рематериализации. Например, переменная **c** (рис. 29a) в представлении Lithium представляется как выходной операнд инструкции 3 (*OutputC3*) и как входной операнд инструкций 5 (*InputC5*) и 6 (*InputC6*) (все эти операнды отображаются в один и тот же регистр или слот памяти). Операнд *OutputC3* может быть вычислен с помощью прямой рематериализации из переменных **a** и **b.** Операнд *InputC5* может быть вычислен из переменных **e** и **d,** а *InputC6* из переменных **f и b** с помощью обратной рематериализации. На первом этапе такая информация присваивается всем Lithium-операндам.

Второй этап алгоритма — распространение полученной информации о рематериализации по созданным во время распределения регистров интервалам жизни (live interval) переменных. Предположим, что инструкция 6 является последним использованием переменной **c**. Интервалом жизни переменной **c** является промежуток [1, 7), где **c** используется в точках 5 и 6. При создании интервалов жизни для переменных информация о рематериализации передается

от Lithium-операндов к интервалам жизни. В итоге, интервал жизни переменной **c** будет содержать информацию об операндах *OutputC3 InputC5* и *InputC6*. Следовательно, значение **c** можно будет вычислить тремя разными способами.

На третьем этапе алгоритма модуль распределения регистров модифицируется таким образом, чтобы интервалы жизни, содержащие информацию о рематериализации распределялись в слоты памяти, а интервалы, которые могут участвовать в рематериализации значений, наоборот, распределялись в регистры.

Четвертый этап алгоритма выполняется после распределения регистров. Последовательно сканируются все интервалы жизни, содержащие информацию о рематериализации, которые были распределены в слоты памяти. Для каждой точки использования (англ. use position) таких интервалов проверяется, является ли условие рематериализации валидным: то есть операнды, используемые для вычисления значения, доступны и находятся в регистрах в данной точке.

Последний шаг алгоритма — модификация модуля кодогенерации. На этом шаге генерируется код для вычисления всех значений с валидной информацией о рематериализации, вместо их загрузки из памяти.

Для улучшения качества распределения регистров алгоритм линейного сканирования может разбивать интервалы жизни переменных на несколько частей. Например, алгоритм может разбить интервал в пределах цикла так, чтобы часть интервала в цикле можно было распределить в регистр. Следовательно, части одного и того же интервала могут быть распределены в разные регистры или слоты памяти. После распределения алгоритм вставляет операции пересылки на границах таких интервалов (пересылка из регистра в регистр, из регистра в память, из памяти в регистр или из памяти в память). Такие пересылки вставляются также для разрешения конфликтов Ф-функций (декомпозиция SSA формы). В реализованном алгоритме учитываются все

такие случаи, и значения могут быть заново вычислены также на границах интервалов жизни.

4.3 Реализация поддержки новых операций на оптимизирующем уровне компилятора JavaScriptCore

Как видно из результатов сравнения производительности разных уровней оптимизаций компилятора JavaScriptCore (таблица 1, рис. 6, рис. 7), важно обеспечить выполнение как можно большего количество «горячих» участков кода на уровне оптимизирующего компилятора.

На оптимизирующем уровне компилятора JSC не поддерживается часть операций языка JavaScript, поэтому для увеличения эффективности можно рассмотреть возможность реализации поддержки новых операций на уровне DFG JIT.

4.3.1 Добавление поддержки итератора по ключам объектов JavaScript на оптимизирующем уровне DFG JIT

Примером конструкции, которая не поддерживалась на оптимизирующих уровнях компиляции, является структура “for…in” – итератор по ключам объектов JavaScript. Функция, содержащая такую структуру, выполнялась только на неоптимизирующих уровнях LLInt и BaseLine JIT, даже если эта функция вызывалась много раз (рис. 30).

```
functon HotFunction () {
   var array = new Array(100);
   for (var i in array) {
      DoSomthing(i)
   }
}
functon HotFunction () {
    for (var j = 0; j < 100000; j++) {
        HotFunction ();
     }
}
```

Рисунок 30. Пример функции, работающей только на неоптимизирующих уровнях компиляции

Внутреннее представление уровня DFG JIT имеет типизированную структуру (тип переменных вычисляется из профиля, собранного на нижних уровнях компиляции). Для эффективной реализации структуры “for…in” на уровне DFG JIT, в первую очередь, необходимо вычислить тип возвращаемых значений итератора. Для этого были внесены соответствующие изменения на неоптимизирующих уровнях LLInt и Baseline JIT для пополнения профиля функции информацией о возвращаемых значениях итератора “for…in”.

В промежуточном представлении уровней LLInt и Baseline JIT итератор по ключам раскрывается в три байткод инструкции:

- get_pnames: создает итератор, сохраняет его размер, инициализирует текущий индекс. При реализации этой инструкции также производятся разные проверки, например, проверяется, является ли значение объекта *Null* или *Undefined*.
- next_pnames: возвращает текущий итератор, если текущий индекс меньше размера ключей объекта, иначе завершает цикл.
- get_by_pname: возвращает значение по текущему ключу итератора.

Добавление поддержки итератора по ключам предполагает реализацию трансляции байткод инструкций get_pnames, get_by_pname и next_pname во внутреннее представление уровня DFG JIT. При этом надо учесть, что в представлении DFG JIT каждый узел может возвращать только одно значение. Так инструкцию get_pnames на уровне DFG JIT необходимо разделить на три узла в графе управления. Первый узел возвращает созданный итератор, второй — количество ключей в объекте, третий — текущий индекс. Инструкция next_pnames была разделена на два узла в графе управления DFG. В первом узле производится проверка условия цикла, второй возвращает текущий итератор. Также были внесены изменения во все оптимизирующие проходы компилятора DFG JIT для продвижения разной информации об операцию “for…in” по всему графу управления (например, модуль распространения типов

был изменен для продвижения информации о типах возвращаемых значений итератора).

Поддержка итератора по ключам была добавлена для платформ x86 (x86_64) и ARM. Таким образом, функции, содержащие такой цикл, также могут эффективно выполняться на уровне DFG JIT.

4.3.2 Добавление поддержки оператора switch на оптимизирующем уровне DFG JIT

Одной из неподдерживаемых операций на уровне DFG JIT является оператор switch. Выполнение оператора switch на уровне байткода компилятора JavaScriptCore в самом общем случае происходит с помощью реализации последовательных сравнений и условных переходов. Однако, при возможности оптимизации, когда во всех рассматриваемых “case” вариантах используется одинаковый тип значения для сравнения, например, числа, символы или строки, то допускается создание специальной операции байткода, организующей переход по табличным значениям. Есть также дополнительные ограничения на создание табличного перехода, например, для целочисленных значений проверяется, чтобы не менее 10% строк таблицы оказывались заполнены переходами. Исследования показали, что оптимизирующий уровень DFG не поддерживает ни один из оптимизированных табличных вариантов оператора switch. Есть несколько подходов реализации switch на уровне DFG JIT. В самом простом случае, в представлении DFG оператор switch можно реализовать с помощью последовательных условных переходов (if-else блоков). Такая реализация имеет несколько недостатков. Во-первых, структура внутреннего представления DFG предполагает, что каждой байткод инструкции соответствует всего один базовый блок в графе управления, и для добавления поддержки последовательных условных переходов для одной байткод инструкции необходимо внести изменения в саму структуру промежуточного представления DFG JIT. Во-вторых, для больших switch операций (с множественными ветвлениями) такая реализация неэффективна и может

привести к ухудшению производительности (из-за генерации множественных базовых блоков и условных переходов).

Второй подход — реализация поддержки операции switch на уровне внутреннего представления DFG JIT. Сложность такой реализации состоит в том, что в представлении DFG JIT каждый базовый блок может иметь максимум два исходящих ребра в другие базовые блоки (базовый блок операции switch может иметь множество исходящих ребер). Для решения проблемы были внесены изменения в структуру графа управления для поддержки базовых блоков с множественными преемниками. В итоге, поддержка была реализована следующим способом: операции switch с небольшим количеством ветвлений (разных case случаев) реализованы с помощью последовательных условных переходов (if-else блоков). В остальных случаях switch реализован на уровне внутреннего представления DFG JIT. Также были внесены изменения во все оптимизирующие проходы компилятора DFG JIT для продвижения разной информации об операцию switch по всему графу управления. Поддержка была реализована для архитектур X86 и ARM.

4.3.3 Добавление поддержи оператора typeof на оптимизирующем уровне DFG JIT

Еще одним примером операции, которая не поддерживалась на уровне оптимизирующего компилятора JavaScriptCore является функция typeof. Для этой функции поддерживался только вариант, когда результат вызова typeof сравнивался со строковой константой, равной “number”, “string”, “object”, “function” или “undefined” (например, *typeof (x) == “string”*). При этом, поддержки сравнения на неравенство реализована не было. Так, выражение *!(typeof (x) == “string”)* проходила компиляцию на уровне DFG JIT, а выражение *(typeof (y) != “object”)* могла только выполнятся на уровнях LLINT и Baseline JIT. В промежуточном представлении уровней LLInt и Baseline JIT операция typeof с последующим сравнением с константой реализована с помощью отдельных байткод инструкций: *op_is_undefined, op_is_boolean, op_is_number, op_is_string, op_is_object, op_is_function*. На первом этапе работы

была добавлена поддержка операции typeof с последующим сравнением на неравенство. Для этого при генерации байткода операции типа *typeof(A)!=B* были заменены на *!typeof(A)==B*. Это позволило создавать байткод инструкций типа *!op_is_B*, которые уже поддерживались на уровне DFG JIT.

Добавление полной поддержки оператора typeof (без последующих сравнений) на уровне DFG JIT было реализовано следующим образом:

- Если для операции *typeof(A)* можно статически доказать, что операнд *A* имеет тип *T*, в DFG JIT операция *typeof* заменятся константной строкой, содержащей имя типа *T*. Пример такой ситуации приведен на рис. 31. Можно статически доказать, что при срабатывании выражения *console.log(typeof(A))* переменная *A* имеет тип "*object*".
- Если тип переменной A нельзя вычислить статически, используется собранная информация о типах, и операция typeof спекулятивным образом заменяется предугаданным типом. В код также вставляется необходимая проверка условия спекуляции. Если результат проверки оказывается ложным, вызывается *C* функция для вычисления типа переменной (эта же функция используется и на уровнях LLInt и Baseline JIT).

```
if (typeof(A) == "object") {
   console.log(typeof(A));
   return;
}
```

Рисунок 31. Пример использования операции Typeof

В итоге, была реализована полная поддержка операции typeof на уровне DFG JIT для архитектур X86 и ARM.

4.3.4 Удаление ложных деоптимизаций и обратных замен на стеке

Другим направлением для обеспечения выполнения большего количества кода может быть устранение причин излишних деоптимизации. В ряде случаев часть таких возвратов можно избежать за счет добавления дополнительного анализа, либо за счет улучшения реализации операций, вызывающих деоптимизацию, на уровне оптимизирующего компилятора.

Удаление излишних проверок на отрицательный ноль

Одним из аспектов, который необходимо учитывать при оптимизации арифметических операций, является отрицательный ноль. В стандарте языка JavaScript все числа являются вещественными числами двойной точности, удовлетворяющими стандарту IEEE [31], поэтому следует различать положительный и отрицательный ноль. Например, значением следующих операций с целыми числами является минус бесконечность (англ. *minus infinity*). 2 / (-0) ; 3/ (0 / -4) ; 7 / (-8 % 8). Необходимость реализации проверок на отрицательный ноль возникает, когда во время выполнения оптимизаций операции с числами *double* заменяются операциями с целыми числами *int32*. Если результат такой операции – отрицательный ноль, необходимо организовать деоптимизацию, чтобы не получить неверный с точки зрения стандарта результат (отрицательный ноль не представляется в 32 битах).

В JavaScriptCore имеется реализация проверок арифметических операций на отрицательный ноль, однако эту реализацию можно улучшить, учитывая, что в некоторых ситуациях проверки на отрицательный ноль можно опустить без ущерба для корректности выполнения. Например, при вычислении выражения *5/(a%b+3)*, если результатом выражения a%b является ноль, нет необходимости различать положительный ноль и отрицательный ноль.

Нами был предложен улучшенный алгоритм для реализации проверок арифметических операций на отрицательный ноль. На первом шаге алгоритма происходит обход по всем вершинам графа DFG. Если значение результата некой операции может быть нулем, для этой операции в графе потока

управления выставляется флаг *NodeNeedsNegZero*. Если некое выражение y в дальнейшем участвует в вычислении суммы или в разности ($y+A$, $A+y$, A−y, где A – константа, которая не является отрицательным нулем, A !=- 0), то при вычислении значения выражения y проверка отрицательного нуля удаляется. Аналогично для разности y−A, где $A!=+0$. Для всех других операций с числами флаг необходимо копировать.

Другие улучшения касаются операций деления и взятия остатка. Целочисленные операции деления x%y дают в результате отрицательный ноль тогда и только тогда, когда x равен нулю, и y отрицательное число. Один из недочетов, имеющихся в DFG JIT реализации состоял в том, что при равенстве делимого нулю сразу происходила деоптимизация, без проведения проверки, что делитель отрицателен. Это вызывало множество ненужных возвратов на более медленный уровень Baseline JIT.

Результат операции взятия остатка x%y может быть отрицательным нулем тогда и только тогда, когда x делится на y нацело, и x меньше 0. Здесь также в имеющейся реализации после проверки результата на равенство нолю сразу выполнялась деоптимизация. Нами был реализован улучшенный алгоритм, и теперь деоптимизация выполняется только если значение делимой является отрицательным числом.

В имеющейся реализации проверок на отрицательный ноль также было выявлено несколько ошибок. Для операции взятия остатка в DFG отдельно рассмотрен случай, когда делитель является константой и степенью двойки. В этом случае операция взятия остатка для оптимизации может быть заменена битовой операцией. В случае отрицательного делимого также возможна такая реализация: необходимо предварительно изменить знак операнда, а в конце изменить знак результата. Однако в имеющейся реализации в этом случае пропускается проверка на отрицательный ноль. Но при выставленном флаге *NodeNeedsNegZero* ее необходимо выполнять, чтобы не потерять знак в выражениях, таких как x%8, при значении x, равном -8. Другая ошибка

касалась операции унарного минуса, для которого флаг *NodeNeedsNegZero* ошибочно стирался. Все найденные ошибки также были исправлены.

Эффективная реализация некоторых операций на оптимизирующих уровнях компилятора JSC и V8.

Оптимизирующий уровень компилятора JavaScriptCore использует информацию, собранную на нижних уровнях выполнения для эффективной организации обращений к методам и полям объектов. Использование профиля позволяет обращаться к полям объектов непосредственно по вычисленному смещению. Однако такое обращение приводило к деоптимизации при изменении или перемещении (например, сборщиком мусора) объектов. Нами был предложен другой подход, который позволяет вместо деоптимизации реализовать “неоптимизированное” обращение к полям объектов на уровне DFG JIT, избегая тем самым деоптимизаций и перехода на неоптимизированный уровень компиляции.

При генерации машинного кода для функции *String.fromCharCode* на оптимизирующем уровне Crankshaft компилятора V8 требовалось, чтобы аргументы этой функции являлись целыми числами, в противном случае происходила деоптимизация. Однако деоптимизаций можно избежать в случаях, когда тип аргумента является вещественным числом. Для этого реализация функции *String.fromCharCode* была модифицирована следующим образом:

- Для целочисленных аргументов используется имеющаяся в Crankshaft реализация.
- Если аргумент (аргументы) функции — вещественное число, которое помещается в целочисленный регистр без потери точности (например, 5.0), генерируется машинный код для конвертации вещественного числа в целочисленный тип (для x86 это инструкция *CVTTSD2SI*, для ARM — *VCVT*). Далее целое число используется для вычисления значения функции.

- Если аргумент функции – вещественное число, которое не помещается в целочисленный регистр, то генерируется код для округления аргумента (согласно стандарту Ecmascript [30]). Далее полученное целое число используется для вычисления значения функции.
- В остальных случаях происходит деоптимизация.

4.3 Выводы и результаты

Были разработаны и реализованы новые алгоритмы машинно-независимой оптимизации для динамических многоуровневых компиляторов:

- эффективный алгоритм удаления излишних вызовов функций без побочных эффектов;
- эффективный алгоритм рематериализации регистров.

В компиляторах JavaScriptCore и V8 также были реализованы следующие оптимизации:

- жадный алгоритм линейного сканирования для глобального распределения регистров;
- алгоритм для эффективной организации встраивания функций;
- алгоритм для эффективной организации проверок на отрицательный ноль;
- Оптимизация доступа к глобальным переменным для платформы ARM.

После выполнения оптимизации обращений к глобальным переменным ускорилось выполнение всех тестов, использующих обращение к глобальным переменным. Тест bitops-bitwise-and из набора SunSpider ускорился на 10%. Синтетические тесты, использующие глобальные переменные, ускорились до 20%.

Улучшение оптимизации через удаление избыточных вызовов функций позволило ускорить множество тестов из наборов SunSpider, v8-v7 и BrowserMark. В таблице 6 приведены результаты сравнения

производительности на наборе v8-v7; удаление вызовов функций без побочных эффектов позволило ускорить тест v8-regexp на 18%.

Таблица 6. Результаты улучшения производительности на наборе v8-v7 после удаления вызовов функций без побочных эффектов

Имя теста	Производительность V8, мс	Производительность V8 с улучшенным удалением мертвого кода, мс	Ускорение, %
V8-crypto	2004	1989	0.75
V8-deltablue	4182	4172	-
V8-early-boyer	6619	6618	-
V8-raytrace	7710	7680	-
V8-regexp	2254	1849	18
V8-richards	1379	1369	0.7
V8-splay	3455	3460	-
Суммарное время	27603	27137	1.68

В таблице 7 приведены результаты сравнения производительности на наборе BrowserMark. Удаление излишних вызовов позволило ускорить тест *ArrayWeighted* на 18%, а тест *StringWeighted* ускорился 3.5 раз. В целом, набор ускорился примерно на 12%. Тестовый набор SunSpider улучшился в среднем на 0,7%.

Таблица 7. Результаты улучшения производительности на наборе BrowesrMark после удаления вызовов функций без побочных эффектов (больше-лучше)

Имя теста	Производительность V8	Производительность V8 с улучшенным удалением мертвого кода	Ускорение, число раз
ArrayBlur	5.1485	5.2093	1.012
ArrayWeghted	5629.6	6652.4	1.18
MathDijkstra	34791.4	35525.2	1.02
MathPrimes	109929.6	109343.8	0.95

Таблица 7. Продолжение

StringChat	10567.2	10532	-
StringFilter	2902.8	2921.4	-
StringSHA1	4816.6	4916	1.02
StringUADetect	20626.2	20940	1.015
StringValidate	35672.0	35774.6	-
StringWeighted	5477.6	19457.4	3.5

Добавление ускоренного выполнения математических и строковых функций также позволило ускорить несколько тестов из тестовых наборов JavaScript. Например, улучшение функции *Math.power* для случая целой степени ускоряет на 5% тест *math-partial-sums* из набора *SunSpider*.

Улучшение алгоритма распространения типов позволило ускорить тесты *math-cordic* и *math-spectral-norm* из набора SunSpider на 7% и 4% соответственно.

Реализация жадного алгоритма линейного сканирования для распределения регистров в компиляторе Crankshaft ускорила производительность тестового набора *Octane* в среднем на 3%. Набор *SunSpider* улучшился на 0,3%, а *LongSpider* в среднем ускорился на 3%. В таблице 8 приведены результаты тестирования на наборе *LongSpider*.

Таблица 8. Производительность компилятора V8 на наборе LongSpider после реализации жадного алгоритма линейного сканирования.

	Время выполнения, мс		
	LongSpider		
	LSRA	**GBRA**	**улучшение, %**
3d-cube	2845.0	2854.1	**-**
3d-morph	6908.5	6850.3	0.85
3d-raytrace	3239.6	3066.2	5.3
access-binary-trees	3373.0	3382.2	-

Таблица 8. Продолжение

access-fannkuch	1169.4	1108.7	5.2
access-nbody	2325.9	2332.6	-
access-nsieve	1638.8	1653.2	-0.88
bitops-3bit-bits-in-byte	108.5	108.6	-
bitops-bits-in-byte	780.7	781.2	-
bitops-nsieve-bits	2905.5	2873.1	1.11
controlflow-recursive	2156.8	2164.4	-
crypto-aes	2821.9	2795.1	0.96
crypto-md5	6494.1	5569.7	14.20
crypto-sha1	22782.3	21601.0	5.17
date-format-tofte	4220.5	4078.4	3.38
date-format-xparb	9671.8	9650.4	-
math-cordic	4020.8	4009.1	-
math-partial-sums	5199.9	5166.0	0.66
math-spectral-norm	2381.7	2381.0	-
string-base64	2610.0	2610.3	-
string-fasta	4398.7	4119.1	6.3
string-tagcloud	2032.2	2013.7	0.92
Суммарное время	94085.6	91168.4	3.1

Для оценки эффективности реализованного алгоритма рематериализации регистров были проведены тестирования на тестовых наборах SunSpider, Kraken, Octane.

В наборе SunSpider около 30 инструкций обращений к памяти удалось заменить на более быстрые арифметические операции. На некоторых тестах удалось заменить до восьми инструкций во вложенных циклах на более быстрые операции. К сожалению, эти изменения не повлияли на общую производительность тестов, так как тесты в наборе SunSpider имеют короткое время выполнения и не содержат тяжелых циклов.

На тесте *audio-beat-detection* из набора Kraken удалось заменить около 8 инструкций обращения к памяти во вложенных циклах на более быстрые арифметические операции, что привело к почти 5% улучшению производительности на этом тесте. Такая же замена на тесте *audio-fft* из того же набора, позволила ускорить тест на 7%.

В наборе Octane удалось заменить около 200 инструкций обращения к памяти на более быстрые операции. 5 – 10 инструкций были заменены в тестах *CodeLoad*, *Zlib* и *TypeScript*. 20 – 30 в тестах *Crypto*, *Pdf* и *NavierStokes*. 45 инструкций в тесте *GameBoy* и более 70 инструкций в тесте *Mandreel*. Такие замены позволили ускорить тест *GameBoy* на 2%, а тест *Mandreel* ускорился на 4%. Тестирование также показало, что реализованный алгоритм не имеет негативного влияния на время компиляции. Ухудшение производительности в тестовых наборах не было выявлено.

Реализация поддержки оператора ‘switch’ на оптимизирующем уровне DFG JIT компилятора JSC привела к 39%-му росту производительности на тесте StringSHA1, а реализация оператора ‘typeof’ – к 17%-му росту производительности на тесте ArrayBlur. Реализация поддержки оператора “for…in” улучшила производительность теста string-fasta на 4%, а теста string-tagcloud – на 2%.

Удаление ложных срабатываний деоптимизаций в компиляторе DFG JIT позволило ускорить множество тестов из набора SunSpider. В среднем тестовый набор стал выполняться на 5% быстрее, ускорение отдельных тестов составляет до 35%. Результат тестирования набора SunSpider приведен в таблице 9.

Таблица 9. Сравнение производительности набора SunSpider после удаления ложных деоптимизаций (меньше - лучше)

Тест	**Производительность JSC, мс**	**Производительность JSC с реализованными улучшениями, мс**	**Улучшение, %**
3d-cube	96.4	96.3	-
3d-morph	44.6	44.6	-
3d-raytrace	126.3	106.4	15.7
access-binary-tree	27.8	27.7	-
access-fannkuch	50.3	50.3	-
access-nbody	47.2	47	-
access-nsieve	24	24	-

Таблица 9. Продолжение

bitops-3bit-in-byte	16.7	16.5	1.2
bitops-bits-in-byte	23.5	23.5	-
bitops-bitwise-and	23	22.7	1.3
bitops-nsieve-bits	27.9	28.1	-
control-flow-recursive	25	24.7	1.2
crypto-aes	80.5	68.3	15.1
crypto-md5	56.8	48.7	14.2
crypto-sha1	37.9	33.3	12.1
date-format-tofte	87.5	87.2	-
date-format-xparb	83.3	78.6	5.6
math-cordic	33.2	29.7	10.5
math-partial-sum	51.9	52	-
math-spectral-norm	36	23.6	34.5
regexp-dna	69.7	69.9	-
string-base64	38	38	-
string-fasta	68	67.3	1
string-tagcloud	94.7	94	-
string-unpack-code	148	145.2	1.9
string-validate-input	54.4	48.1	11.58
Суммарное время	1472.6	1395.7	5.2

Заключение

Разработанные и реализованные методы оптимизации позволили достичь значительного улучшения производительности рассматриваемых компиляторов. Реализованные оптимизации на основе профиля программы улучшают производительность тестовых наборов SunSpider, Kraken и Octane на 11%, 4% и 2%, соответственно. При этом, улучшение производительности для отдельных тестов достигает 50%. Использование инфраструктуры LLVM в качестве дополнительного уровня выполнения, а также реализованные методы машинно-независимой и машинно-зависимой оптимизации позволяют улучшить те же тестовые наборы в среднем на 8-10%. Более подробный анализ полученных результатов приведен в тексте диссертации.

Все разработанные методы оптимизации используются в рамках совместного научно-исследовательского проекта корпорации Samsung и Института системного программирования РАН. Некоторые из них были одобрены сообществом разработчиков компилятора JavaScriptCore и включены в состав этого компилятора.

В настоящее время ведутся работы по интеграции разработанных методов оптимизации в компиляторы SpiderMonkey и ChakraCore.

Основные результаты диссертационной работы.

1. Разработан и реализован новый метод оптимизации многоуровневых динамических компиляторов, основанный на информации о профиле программы. Использование информации о профиле позволяет улучшить производительность компиляторов, путем
   - организации немедленного переключения выполнения «горячих» участков кода на уровень оптимизирующего компилятора;
   - удаления деоптимизаций и обратных переходов на неоптимизирующий уровень выполнения;
   - выбора набора оптимизаций для конкретных приложений.

2. Разработан и реализован новый метод компиляции программ с динамическими типами в статически типизированное внутреннее представление LLVM, позволяющий применять оптимизации LLVM, к программам, написанным на языке JavaScript. При этом была добавлена поддержка компиляции всех конструкций и динамических свойств языка JavaScript:
   - сборщика мусора и перемещений объектов;
   - бинарной совместимости и переключении между разными уровнями компиляции;
   - спекулятивной компиляций и деоптимизаций.
3. Разработан и реализован эффективный алгоритм рематериализации регистров для динамических компиляторов.
4. Разработан и реализован эффективный алгоритм удаления излишних вызовов функций без побочных эффектов для языка JavaScript.
5. Реализованы жадный алгоритм линейного сканирования и алгоритм для эффективной организации встраивания функций в компиляторе V8 и алгоритм для эффективной организации проверок на отрицательный ноль в компиляторе JavaScriptCore.

В будущем планируется расширить класс оптимизаций для динамических многоуровневых компиляторов, а также улучшить технологию компиляции программ с динамическими типами в статическое представление для генерации более качественного биткода LLVM. Планируется разработка специального модуля для оптимизации asm.js кода в рамках разработки уровня LLV8. Также планируется разработка оптимизационных проходов в инфраструктуре LLVM, учитывающих особенности структуры биткода LLVM, получаемого из JavaScript.

Приложение А. Разметка библиотечных функций JavaScript

Array.prototype.toString	arrayOfPrimitiveItemsHandler
Array.prototype.toLocaleString	arrayOfPrimitiveItemsHandler
Array.prototype.concat	looseHandler
Array.prototype.join	arrayPrimitiveArgumentsAndItemsHandler
Array.prototype.pop	prohibitionHandler
Array.prototype.push	prohibitionHandler
Array.prototype.reverse	prohibitionHandler
Array.prototype.shift	prohibitionHandler
Array.prototype.slice	primitiveHandler
Array.prototype.sort	prohibitionHandler
Array.prototype.splice	prohibitionHandler
Array.prototype.unShift	prohibitionHandler
Array.prototype.every	prohibitionHandler
Array.prototype.forEach	prohibitionHandler
Array.prototype.some	prohibitionHandler
Array.prototype.indexOf	primitiveSecondArgumentHandler
Array.prototype.lastIndexOf	primitiveSecondArgumentHandler
Array.prototype._lter	prohibitionHandler
Array.prototype.reduce	prohibitionHandler
Array.prototype.reduceRight	prohibitionHandler
Array.prototype.map	prohibitionHandler
Boolean.prototype.toString	looseHandler
Boolean.prototype.valueOf	looseHandler
Date.prototype.toString	looseHandler
Date.prototype.toISOString	looseHandler
Date.prototype.toUTCString	looseHandler
Date.prototype.toDateString	looseHandler
Date.prototype.toTimeString	looseHandler
Date.prototype.toLocaleString	looseHandler
Date.prototype.toLocaleDateString	looseHandler
Date.prototype.toLocaleTimeString	looseHandler
Date.prototype.getTime	looseHandler
Date.prototype.getTime	looseHandler
Date.prototype.getFullYear	looseHandler
Date.prototype.getUTCFullYear	looseHandler
Date.prototype.toGMTString	looseHandler
Date.prototype.getMonth	looseHandler
Date.prototype.getUTCMonth	looseHandler
Date.prototype.getDate	looseHandler

Date.prototype.getUTCDate	looseHandler
Date.prototype.getDay	looseHandler
Date.prototype.getUTCDay	looseHandler
Date.prototype.getHours	looseHandler
Date.prototype.getUTCHours	looseHandler
Date.prototype.getMinutes	looseHandler
Date.prototype.getUTCMinutes	looseHandler
Date.prototype.getSeconds	looseHandler
Date.prototype.getUTCSeconds	looseHandler
Date.prototype.getMilliSeconds	looseHandler
Date.prototype.getUTCMilliseconds	looseHandler
Date.prototype.getTimezoneO_set	looseHandler
Date.prototype.setTime	prohibitionHandler
Date.prototype.setMilliSeconds	prohibitionHandler
Date.prototype.setUTCMilliseconds	prohibitionHandler
Date.prototype.setSeconds	prohibitionHandler
Date.prototype.setUTCSeconds	prohibitionHandler
Date.prototype.setMinutes	prohibitionHandler
Date.prototype.setUTCMinutes	prohibitionHandler
Date.prototype.setHours	prohibitionHandler
Date.prototype.setUTCHours	prohibitionHandler
Date.prototype.setDate	prohibitionHandler
Date.prototype.setUTCDate	prohibitionHandler
Date.prototype.setMonth	prohibitionHandler
Date.prototype.setUTCMonth	prohibitionHandler
Date.prototype.setFullYear	prohibitionHandler
Date.prototype.setUTCFullYear	prohibitionHandler
Date.prototype.setYear	prohibitionHandler
Date.prototype.getYear	looseHandler
Date.prototype.toJSON	noArgumentsHandler
parseInt	primitiveHandler
parseFloat	primitiveHandler
isNaN	primitiveHandler
isFinite	primitiveHandler
escape	primitiveHandler
unescape	primitiveHandler
decodeURI	primitiveHandler
decodeURIComponent	primitiveHandler
encodeURI	primitiveHandler
encodeURIComponent	primitiveHandler
Math.abs	primitiveHandler
Math.aCos	primitiveHandler
Math.aSin	primitiveHandler
Math.aTan	primitiveHandler
Math.aTan2	primitiveHandler
Math.ceil	primitiveHandler
Math.cos	primitiveHandler
Math.exp	primitiveHandler
Math._oor	primitiveHandler

Math.log	primitiveHandler
Math.max	primitiveHandler
Math.min	primitiveHandler
Math.pow	primitiveHandler
Math.random	prohibitionHandler
Math.round	primitiveHandler
Math.sin	primitiveHandler
Math.sqrt	primitiveHandler
Math.tan	primitiveHandler
Number.prototype.toString	looseHandler
Number.prototype.toLocaleString	looseHandler
Number.prototype.valueOf	looseHandler
Number.prototype.toFixed	looseHandler
Number.prototype.toExponential	looseHandler
Number.prototype.toPrecision	looseHandler
Object.prototype.toString	looseHandler
Object.prototype.toLocaleString	looseHandler
Object.prototype.valueOf	looseHandler
Object.prototype.hasOwnProperty	primitiveHandler
Object.prototype.propertyIsEnumerable	primitiveHandler
Object.prototype.isPrototypeOf	looseHandler
Object.prototype.de_neGetter	prohibitionHandler
Object.prototype.de_neSetter	prohibitionHandler
Object.prototype.lookupGetter	primitiveHandler
Object.prototype.lookupSetter	primitiveHandler
RegExp.prototype.compile	prohibitionHandler
RegExp.prototype.exec	prohibitionHandler
RegExp.prototype.test	prohibitionHandler
RegExp.prototype.toString	looseHandler
String.prototype.toString	looseHandler
String.prototype.toString	looseHandler
String.prototype.charAt	primitiveHandler
String.prototype.charCodeAt	primitiveHandler
String.prototype.concat	primitiveHandler
String.prototype.indexOf	primitiveHandler
String.prototype.lastIndexOf	primitiveHandler
String.prototype.match	primitiveOrOnetimeRegexpHandler
String.prototype.replace	primitiveOrOnetimeRegexpHandler
String.prototype.search	primitiveOrOnetimeRegexpHandler
String.prototype.slice	primitiveHandler
String.prototype.split	primitiveHandler
String.prototype.substr	primitiveHandler
String.prototype.substring	primitiveHandler
String.prototype.toLowerCase	looseHandler
String.prototype.toUpperCase	looseHandler
String.prototype.localeCompare	primitiveHandler
String.prototype.toLowerCase	looseHandler
String.prototype.toUpperCase	looseHandler
String.prototype.big	looseHandler

String.prototype.small	looseHandler
String.prototype.blink	looseHandler
String.prototype.bold	looseHandler
String.prototype._xed	looseHandler
String.prototype.italics	looseHandler
String.prototype.strike	looseHandler
String.prototype.sub	looseHandler
String.prototype.sup	looseHandler
String.prototype.fontcolor	primitiveHandler
String.prototype.fontsize	primitiveHandler
String.prototype.anchor	primitiveHandler
String.prototype.link	primitiveHandler
String.prototype.trim	looseHandler
String.prototype.trimLeft	looseHandler
String.prototype.trimRight	looseHandler

Список рисунков

Список таблиц

Список литературы

[1] M. Arnold, S. J. Fink, D. Grove, M. Hind и P. F. Sweeney., «A Survey of Adaptive Optimization in Virtual Machines» *IEEE,* с. 449 - 466, 2005

[2] J. Lin, T. Chen, W. Hsu, P. Yew, R. Ju, T. Ngai и S. Chan, «A compiler framework for speculative analysis and optimizations,» *SIGPLAN Notices,* с. 289-299, 2003.

[3] F. Allen, «Control flow analysis,» *SIGPLAN Notices,* с. 1 - 19, 1970.

[4] S. Lee, S. Moon и S. Kim, «Enhanced hot spot detection heuristics for embedded java just-in-time compilers,» *SIGPLAN Notices,* с. 13-22 , 2008.

[5] «Сайт компилятора JavaScriptCore» http://www.webkit.org.

[6] «Сайт компилятора V8» https://developers.google.com/v8/.

[7] «Страница компилятора SpiderMonkey»https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[8] «Страница компилятора ChakraCore» https://github.com/Microsoft/ChakraCore.

[9] W. Jung и S. M. Moon, «Javascript ahead-of-time compilation for embedded web platform,» *ESTIMedia,* с. 1 - 9, 2015.

[10] J. K. Martinsen, H. Håkan и A. Isberg, «Using speculation to enhance javascript performance in web applications,» IEEE Internet Computing, с. 10-19, 2013.

[11] I. Jibaja, P. Jensen, N. Hu и M. R. H. a. all, « Vector Parallelism in JavaScript: Language and Compiler Support for SIMD,» *PACT,* с. 1-12, 2015.

[12] P. P. Chang, S. Mahlke и W. Hwu, «Using profile information to assist classic

code optimizations,» *Sowftware-Practice and Experience,* с. 1301 - 1321 , 1991.

[13] C. Lattner и V. Adve, «LLVM: a compilation framework for lifelong program analysis & transformation,» *CGO,* с. 75-87, 2004.

[14] V. Vardanyan, «Optimizations of JavaScript programs,» *GSPI's scientific journal 2014,* с. 122-128.

[15] 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,* pp. Том 26, Выпуск 1, с. 297-314, 2014.

[16] 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,* с. 33-48, 2015.

[17] 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,* с. 236-240, 2015.

[18] 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,* с. 116-120, 2015.

[19] V. Vardanyan, S. Asryan и R. Buchatskiy, «Integrated register rematerialization in JavaScript V8 JIT compiler,» *10th International Conference on Computer Science and Information Technologies,* с. 240-244, 2015.

[20] S. Sargsyan, S. Kurmangaleev, V. Vardanyan и V. Zakaryan, «Code Clones Detection Based on Semantic Analysis for JavaScript Language,» *10th International Conference on Computer Science and Information Technologies,* с.

182-185, 2015.

[21] В. Варданян, «Методы оптимизации программ на языке JavaScript, основанные на статистике выполнения программы,» *Труды Института системного программирования РАН, Том 28. Выпуск 1,* с. 5-20, 2016.

[22] A. Aho, M. Lam, R. Sethi и J. Ullman, в *Compilers: Principles, Techniques, and Tools*, 2008, с. 155-247.

[23] A. Aho, M. Lam, R. Sethi и J. Ullman, в *Compilers: Principles, Techniques, and Tools*, 2010, с. 251-378.

[24] Ian Piumarta и Fabio Riccardi , «Optimizing direct threaded code by selective inlining,» *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation,* с. 291-300, 5 May 1998.

[25] S. J. Fink и F. Qian, «Design, Implementation, and Evaluation of Adaptive Recompilation,» *Proceedings of the IEEE,* с. 241-252, 2003.

[26] C. Chambers D. Ungar, U. Hölzle, «Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches,» *91 Proceedings of the European Conference on Object-Oriented Programming,* с. 21-38, 1991.

[27] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck «Efficiently computing static single assignment form and the control dependence graph,» *ACM Transactions on Programming Languages and Systems,* с. 451-490, 1991.

[28] A. Aho, M. Lam, R. Sethi и J. Ullman, «Compiler Principles, Techniques and Tools,» 2008, с. 705-838.

[29] T. Würthinger, Ch. Wimmer, H. Mössenböck «Array bounds check elimination for the Java HotSpot™ client compiler,» *Proceedings of the 5th international symposium on Principles and practice of programming in Java,* с.

125-133, 2007.

[30] «Стандарт Ecma-262,» http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[31] «Стандард IEEE 754,» http://grouper.ieee.org/groups/754/.

[32] J. F. Bartlett, «Compacting Garbage Collection with Ambiguous Roots,» *ACM SIGPLAN Lisp Pointers,* с. 3-12, 1988.

[33] «Сайт инфраструктуры LLVM,» http://www.LLVM.org.

[34] M. Schoeberl «Design and Implementation of an Efficient Stack Machine,» *IEEE proceedings,* с. 159-160, 2005.

[35] B. Blanchet, «Escape analysis for object-oriented languages: application to Java,» *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications,* с. 20-34, 1999.

[36] C. Click, «Global code motion/global value numbering,» *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation,* с. 246-257, 1995.

[37] M. Poletto и V. Sarkar, «Linear scan register allocaton,» *ACM Transactions on Programming Languages and Systems,* с. 895-913 , 1999.

[38] C. N. Click «Combining analyses, combining optimizations,» *ACM Transactions on Programming Languages and Systems ,* с. 181-196, 1995.

[39] A. W. Appel, «Simple generational garbage collection and fast allocation,» *Software: Practice and Experience,* с. 171-183, 1989.

[40] C. J. Cheney, «A nonrecursive list compacting algorithm,» *Communications of the ACM,* с. 677-678, 1970.

[41] T. HYPERLINK "http://dl.acm.org/author_page.cfm?id=81100229688&coll=DL&dl=ACM&trk=0&cfid=599338109&cftoken=33692660" \t "_self" \o "Author Profile Page" Endo , K. HYPERLINK "http://dl.acm.org/author_page.cfm?id=81100136693&coll=DL&dl=ACM&trk=0&cfid=599338109&cftoken=33692660" \t "_self" \o "Author Profile Page" Taura и A. Yonezawa, «A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines,» *IEEE proceedings,* с. 48-49, 1997.

[42] «Страница документации динамического компилятора HotSpot для языка Java,» http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html.

[43] «Страница тестового набора SunSpider,» https://webkit.org/perf/sunspider/sunspider.html.

[44] «Страница тестового набора Octane,» https://developers.google.com/octane/.

[45] «Страница тестового набора Browsermark,» http://web.basemark.com/.

[46] «Страница тестового набора Kraken,» http://krakenbenchmark.mozilla.org/.

[47] «Анализ эффективности оптимизаций в компиляторе V8,» http://www.cs.cmu.edu/~ishafer/compilers/.

[48] «Страница платформы Sikuli UI,» http://www.sikuli.org/.

[49] «Описаные жадного алгоритма линейного сканирования,» http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html.

[50] «Страница платформы asm.js,» asmjs.org.

[51] «Страница документации платформы Webassembly,»

https://github.com/webassembly.

[52] «Страница компилятора Emscripten,» https://github.com/kripken/emscripten.

[53] «Страница документации динамического модуля MCJIT,» http://llvm.org/docs/MCJITDesignAndImplementation.html.

[54] «Страница документации инфраструктуры LLVM,» http://llvm.org/docs/LangRef.html.

[55] «Страница платформы x86,» https://docs.oracle.com/cd/E26502_01/html/E28388/index.html

[56] «Страница документации уровня FTL компилятора JavaScriptCore,» https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/.

[57] C. Zilles и N. J, «Explicit software speculation for dynamic language runtimes,» *Doctoral Dissertation,* 2009.

[58] L. Su и M. H. Lipasti, «Speculative optimization using hardware-monitored guarded regions for java virtual machines,» *Proceedings of the 3rd international conference on Virtual execution environments,* с. 22-32, 2007.

[59] «Страница документации структуры StackMaps,» http://llvm.org/docs/StackMaps.html.

[60] «Страница обзора изменений LLVM в ревизии rL229945,» http://reviews.llvm.org/rL229945.

[61] «Страница документации структуры StatePoint,» http://llvm.org/docs/Statepoints.html.

[62] M. Booshehri, A. Malekpour и P. Luksch, «An Improving Method for Loop

Unrolling,» *International Journal of Computer Science and Information Security,* 2013.

[63] «Страница платформы ARM» https://www.arm.com/.

[64] «Страница компилятора GCC,» https://gcc.gnu.org/.

[65] V. ILIOPOULOS и D. B. PENMAN, «DUAL PIVOT QUICKSORT,» *Discrete Mathematics, Algorithms and Applications,* p. Volume 04 Issue 03, 2012.

[66] G. Chaitin, «Register Allocation via Coloring,» *Computer Languages,* c. 47–57, 1981.

[67] G. J. Chaitin, «Register Allocation and Spilling via Graph Coloring,» *SIGPLAN Notice,* c. 201-207, 1982.

[68] M. Punjani, «Register Rematerialization In GCC».*GCC developers' Summit, 2004.*

[69] M. Bahi и C. Eisenbeis, «Register Reverse Rematerialization,» *HAL-Inria resaerch center,* 2011.