ՀՀ ԳԱԱ ԻՆՖՈՐՄԱՏԻԿԱՅԻ ԵՎ ԱՎՏՈՄԱՏԱՑՄԱՆ ՊՐՈԲԼԵՄՆԵՐԻ ԻՆՍՏԻՏՈՒՏ

Վարդանյան Վահագն Գևորգի

ԴԻՆԱՄԻԿ ՏԻՊԵՐՈՎ ԼԵԶՈՒՆԵՐԻ ՍՏԱՏԻԿ ՕՊՏԻՄԱԼԱՑՄԱՆ ՄԵԹՈԴՆԵՐԸ

Ե.13.04 – «Հաշվողական մեքենաների, համալիրների, համակարգերի և ցանցերի մաթեմատիկական և ծրագրային ապահովում» մասնագիտությամբ տեխնիկական գիտությունների թեկնածուի գիտական աստիճանի հայցման ատենախոսության

ՍԵՂՄԱԳԻՐ

Երևան - 2016

ИНСТИТУТ ПРОБЛЕМ ИНФОРМАТИКИ И АВТОМАТИЗАЦИИ НАН РА

Варданян Ваагн Геворгович

МЕТОДЫ СТАТИЧЕСКОЙ ОПТИМИЗАЦИИ ПРОГРАММ ДЛЯ ЯЗЫКОВ С ДИНАМИЧЕСКИМИ ТИПАМИ

АВТОРЕФЕРАТ

диссертации на соискание ученой степени кандидата технических наук по специальности 05.13.04 – «Математическое и программное обеспечение вычислительных машин, комплексов, систем и сетей»

Ереван - 2016

© National Library of Armenia OCR by Portmind

Ատենախոսության թեման հաստատվել է Երևանի պետական համալսարանում

Գիտական ղեկավար՝ ֆիզ.մաթ.գիտ. դոկտոր Վ. Պ. Իվաննիկով

Պաշտոնական ընդդիմախոսներ՝ ֆիզ.մաթ.գիտ. դոկտոր Ս. Կ. Շուքուրյան
տեխ. գիտ. թեկնածու Մ. Ղ. Գյուրջյան

Առաջատար կազմակերպություն՝ Հայ-Ռուսական (Սլավոնական) համալսարան

Պաշտպանությունը կայանալու է 2016թ. հունիսի 13-ին, ժ. 17.00-ին ՀՀ ԳԱԱ Ինֆորմատիկայի և ավտոմատացման պրոբլեմների ինստիտուտում գործող 037 «Ինֆորմատիկա և հաշվողական համակարգեր» մասնագիտական խորհրդի նիստում հետևյալ հասցեով՝ Երևան, 0014, Պ. Սևակի 1:

Ատենախոսությանը կարելի է ծանոթանալ ՀՀ ԳԱԱ ԻԱՊԻ գրադարանում:
Սեղմագիրը առաքված է 2016թ. մայիսի 13-ին:

Մասնագիտական խորհրդի
գիտական քարտուղար, ֆ.մ.գ.դ. Հ. Գ. Սարուխանյան

Тема диссертации утверждена в Ереванском государственном университете

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

Официальные оппоненты: доктор физ.-мат. наук С. К. Шукурян
кандидат тех.наук М. Г. Гюрджян

Ведущая организация: Российско-Армянский (Славянский) университет

Защита состоится 13-ого июня 2016г. в 17.00 на заседании специализированного совета 037 «Информатика и вычислительные системы» Института проблем информатики и автоматизации НАН РА по адресу: 0014, г. Ереван, ул. П. Севака 1.

С диссертацией можно ознакомиться в библиотеке ИПИА НАН РА.

Автореферат разослан 13-ого мая 2016г.

Ученый секретарь
специализированного совета д.ф.м.н. А. Г. Саруханян

© National Library of Armenia

Общая характеристика работы

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

---

[1] M. Arnold, S. J. Fink, D. Grove, M. Hind, P. F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines, IEEE, 2005

[2] J. Lin, T. Chen, W. Hsu, P. Yew, R. Ju, T. Ngai, S. Chan. A compiler framework for speculative analysis and optimizations, SIGPLAN Notices, 2003

[3] F.E. Allen. Control flow analysis. SIGPLAN Notices, 1970

© National Library of Armenia

«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

---

[4] S. Lee, S. Moon, S. Kim, Enhanced hot spot detection heuristics for embedded java just-in-time compilers, SIGPLAN Notices, 2008

[5] Страница компилятора JavaScriptCore - http://trac.webkit.org/wiki/JavaScriptCore

[6] Страница компилятора V8 - https://developers.google.com/v8/

[7] Страница компилятора SpiderMonkey - https://developer.mozilla.org/enUS/docs/Mozilla/Projects/SpiderMonkey

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

[9] W. Jung ; S. M. Moon, Javascript ahead-of-time compilation for embedded web platform, ESTIMedia, 2015.

[10] J. K. Martinsen ; H. Håkan ; A. Isberg, Using speculation to enhance javascript performance in web applications, IEEE, 2013.

[11] I. Jibaja ; P. Jensen ; N. Hu ; M. R. Haghighat and all, Vector Parallelism in JavaScript: Language and Compiler Support for SIMD, PACT, 2015.

[12] P. P. Chang, S.A. Mahlke, W.W. Hwu, Using profile information to assist classic code optimizations, Sowftware-Practice and Experience, 1991

© National Library of Armenia

немедленного переключения выполнения этих участков на уровень оптимизирующего компилятора при последующих запусках программы.

В последние пару лет также активно развиваются платформы asm.js[13] 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[14].

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.

---

[13] Страница платформы asm.js – http://www.asmjs.org

[14] C. Lattner, V. Adve, LLVM: a compilation framework for lifelong program analysis & transformation, CGO, 2004

© National Library of Armenia

- 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 недочетов 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. Список работ приведен в конце автореферата. 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.

КРАТКОЕ СОДЕРЖАНИЕ РАБОТЫ

Во введении обсуждаются технологии конструирования многоуровневых динамических компиляторов. Формулируются цели и задачи работы, обосновывается ее актуальность, обсуждаются вопросы практического применения разработанных методов и инструментов, производится краткий обзор работы.

Глава 1 содержит описание инфраструктур современных многоуровневых динамических компиляторов на примере JavaScriptCore и V8. В конце главы приводятся выводы и описание основных задач диссертационной работы.

В разделе 1.1 описывается архитектура многоуровневого динамического компилятора JavaScriptCore. Компилятор JavaScriptCore состоит из четырех уровней компиляции (рис. 1). На первых этапах работы из исходного кода строится синтаксическое дерево, из которого далее строится внутреннее представление - байткод (англ. bytecode). Первый уровень компилятора JavaScriptCore представляет собой интерпретатор под названием LLINT (Low Level Interpreter). Этот уровень выполняется с минимальным набором оптимизаций, что обеспечивает быстрое начало выполнения кода.

© National Library of Armenia

Во время работы LLINT происходит сбор информации: сохраняются типы и последние значения полей объектов. Если участок кода (функция или цикл) выполняется достаточное количество раз, компиляция функции переходит на второй уровень выполнения – Baseline JIT. Переход происходит при помощи технологии замены стека (OSR)[15], что позволяет переходить между уровнями во время выполнения программы. На этом уровне функция компилируется в машинный код, экономя время трансляции при последующих вызовах функции.

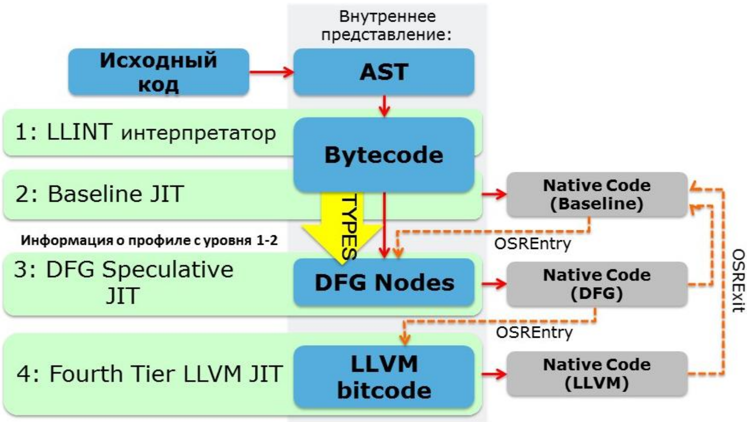

Рис. 1. Многоуровневая архитектура компилятора JavaScriptCore

Следующий уровень компилятора JavaScriptCore - спекулятивный компилятор DFG. На этом уровне из байткода строится граф потока управления программы. Это представление делает возможным реализацию ряда машинно-независимых оптимизаций. DFG JIT распространяет полученную информацию о профиле программы по всему графу и вставляет в код необходимые проверки типов. Если одна из проверок не выполняется, происходит обратная замена на стеке (OSR exit) на код Baseline JIT (деоптимизация). 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 уровень оптимизирующего компилятора. Когда выполняется деоптимизация, происходит переход на неоптимизирующий уровень.

Четвертый уровень компилятора JavaScriptCore – FTL (Fourth Tier LLVM) – использует инфраструктуру LLVM для генерации машинного кода. В нем выполняется большой набор оптимизаций, а в качестве внутреннего представления используется LLVM бит-код.

В разделе 1.2 приведено описание компилятора V8. V8 состоит из двух уровней компиляции. Из исходного кода строится абстрактное синтаксическое дерево. На этом представлении начинает работать компилятор первого уровня – Full-Codegen. В компиляторе Full-codegen трансляция функции в машинный код осуществляется с

[15] S. J. Fink и F. Qian, Design, Implementation, and Evaluation of Adaptive Recompilation, IEEE, 2003

© National Library of Armenia

минимальным набором оптимизаций, что позволяет быстрее перейти к выполнению кода. На этом уровне собирается информация о типах переменных и полях объектов программы, аналогично уровням LLINT и BaseLineJIT компилятора JavaScriptCore. Когда обнаруживается «горячий» участок кода, компиляция переходит на второй уровень – Crankshaft. На этом уровне, как и на третьем уровне JavaScriptCore, строится граф потока управления в виде SSA формы под названием Hydrogen. Далее выполняется ряд машинно-независимых оптимизаций. Как и компилятор DFG JIT, Crankshaft использует собранную на нижнем уровне информацию о профиле программы для спекулятивной компиляции функций. После вышеописанных операций представление Hydrogen переводится в представление Lithium. Это представление является машинно-зависимым и используется для организации распределения регистров и кодогенерации. Компилятор V8, аналогично компилятору JavaScriptCore, использует технологию замены стека (OSR entry, OSR exit) для перехода между уровнями.

Раздел 1.3 содержит описание других известных реализаций языка JavaScript – SpiderMonkey и Chakra.

В разделе 1.4 приводится анализ нескольких известных тестовых наборов языка JavaScript, таких как Octane, BrowserMark, SunSpider и Kraken.

В разделе 1.5 в качестве основной задачи диссертационной работы ставится разработка новых методов оптимизации многоуровневых динамических компиляторов с учетом новых тенденций 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 содержит описание разработанного и реализованного метода оптимизации многоуровневых JIT компиляторов, основанного на использовании информации о профиле программы.

В разделе 2.1 описываются предложенные методы оптимизации. Был разработан и реализован новый метод, позволяющий сохранять и в дальнейшем использовать информацию о часто исполняющихся участках кода для организации немедленного переключения на уровень оптимизирующего компилятора. 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.

Учитывая особенности многоуровневых динамических компиляторов, было также реализовано сохранение следующей информации о статистике выполняемой программы:

- количество и причины деоптимизаций для каждой функции;
- влияние на созданный машинный код каждой реализованной оптимизации;
- количество итераций циклов;

Были добавлены соответствующие счетчики для вычисления вышеуказанных значений.

Информация о количестве деоптимизаций используются для принятия решения о переключении функции на уровень оптимизирующего компилятора. Например, если во время первого выполнения программы в какой-то функции произошел ряд деоптимизаций, при последующих запусках переключение выполнения этой функции на уровень оптимизирующего компилятора можно отложить, что позволит собрать более точную информацию о типах и объектах и избежать деоптимизаций.

© National Library of Armenia

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. Был разработан и реализован новый метод для сохранения информации об эффективности каждой оптимизации, которая позволяет выбрать оптимальный набор оптимизаций для конкретных программ.

Информация о количестве итераций циклов дает возможность эффективно реализовать распределения регистров.

Раздел 2.2 содержит выводы и основные результаты тестирования на тестовых наборах языка JavaScript.

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.

Результаты тестирований подтверждают эффективность разработанного метода. Тестовый набор *SunSpider* в среднем стал выполняться на 11% быстрее, ускорение отдельных тестов составляет 50%. Тестовые наборы *Kraken* и *Octane* в среднем ускорились на 4% и 2%. Улучшение на отдельных тестах из набора Octane составило 10%.

Глава 3 содержит описание метода динамической компиляции программ на языке JavaScript в статически типизированное внутреннее представление инфраструктуры LLVM. Компиляция была осуществлена путем добавления нового уровня выполнения в компиляторе V8 (рис. 2). Данный метод позволяет применять оптимизации LLVM, к программам, написанным на языке JavaScript. Такой подход особенно актуален в связи с активным развитием таких платформ, как *asm.js* и *webassembly*, которые позволяют эффективно выполнять программы, написанные на статических языках программирования в динамической среде. Инфраструктура LLVM используется в компиляторе JavaScriptCore на четвертом уровне оптимизации (FTL JIT). Однако поддержка уровня FTL реализована только для операционных систем Mac OS X и iOS. В данной же работе предлагается добавить уровень оптимизации LLVM в компилятор V8, и в первую очередь для платформы GNU/Linux. Стоит отметить, что внутренние структуры этих компиляторов сильно отличаются, и добавление нового уровня оптимизации в компиляторе V8 с использованием инфраструктуры LLVM требует решения многих новых задач: поддержка спекулятивной компиляции и деоптимизации, бинарная совместимость и переключение между разными уровнями оптимизации, поддержка сборщика мусора.

В разделе 3.1 описывается схема функционирования разработанного метода, которая обеспечивает использование биткода LLVM в качестве новой платформы для динамической компиляции. Инфраструктура LLVM содержит модуль для динамической компиляции (MCJIT [16]), в котором уже задействованы все механизмы LLVM для

[16] Страница документации динамического модуля MCJIT - http://llvm.org/docs/MCJITDesignAndImplementation.html

© National Library of Armenia

машинно-независимой и машинно-зависимой оптимизации, а также для генерации кода под различные платформы. 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 есть вся информация, необходимая для получения статически типизированного представления. Более того, представление Hydrogen, как и LLVM биткод использует SSA-форму. Несмотря на это, в этих двух представлениях есть некие различия в трактовке определения SSA-формы. Например, в представлении LLVM аргументы Ф-функций обязательно должны быть определены в непосредственных предшественниках базового блока, где находится Ф-функция.

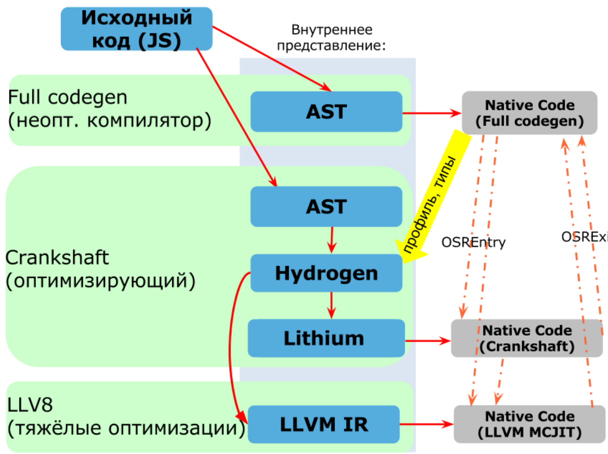

Рис. 2. Архитектура многоуровневого компилятора V8 с добавленным модулем LLVM

Был разработан и реализован алгоритм, который позволяет преобразовать промежуточное представление Hydrogen в корректную, с точки зрения LLVM SSA-форму.

Кроме того, для того чтобы можно было использовать код, созданный уровнем LLVM в среде компилятора V8, необходимо обеспечить бинарную совместимость создаваемых на разных уровнях кодов. Это означает, что LLVM должен поддерживать такое же соглашение о вызовах, как и V8. Чтобы достичь бинарной совместимости, в компиляторе LLVM были внесены соответствующие изменения для добавления поддержки всех необходимых соглашений о вызовах.

Для обеспечения корректности трансляции конструкций JavaScript в LLVM биткод была разработана система регрессионного тестирования, которая запускает каждый тест с модифицированной версией V8 и сравнивает результат с результатом, полученным с помощью немодифицированной версии.

В разделе 3.2 описывается разработанный метод для обеспечения переключения выполняемого кода на уровень оптимизации LLVM. В простом случае, когда время

© National Library of Armenia

выполнения функции небольшое, переключение на уровень оптимизирующего компилятора происходит с помощью замены адреса функции адресом оптимизированного кода при последующих вызовах. 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).

Для организации такого переключения между уровнями Full-Codegen и Crankshaft используется простой безусловный переход на начало соответствующего цикла в сгенерированном Crankshaft-ом коде (рис. 3а). 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. Использование такого подхода для переключения на LLVM код нецелесообразно, так как в LLVM предполагается, что каждая функция может содержать только одну точку входа. Нарушение этого условия может стать причиной ограничения выполняемых в LLVM оптимизаций. В компиляторе JavaScriptCore переключение на уровень LLVM происходит следующим образом: для функции создаются новые копии для каждого возможного входа (цикла). Далее, 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 предложен новый метод, который позволяет организовать переключение на уровень LLVM без накладных расходов на создание 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. В разработанном методе при переключении на оптимизированный код точка входа была помещена в пролог кода функции, где выполняются все необходимые действия для адаптации стека. Также учитывается тот факт, что уровень Full-Codegen мог, в свою очередь, уменьшить значение указателя стека на количество своих локальных переменных (рис. 3.б). 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.

В разделе 3.3 описываются особенности компиляции программ с динамическими типами в статическое представление, в частности, поддержка спекулятивной компиляции и деоптимизаций, а также проблемы, возникающие при автоматической сборке мусора.

Спекулятивные оптимизации являются одним из важных способов для обеспечения высокой производительности современных динамических компиляторов. С их помощью можно организовать частичную компиляцию, которая позволяет уменьшить размер кода, а также сократить общее время выполнения приложений. Кроме того, спекулятивная компиляция позволяет реализовать предсказание типов и технологию “полиморфный встроенный кэш вызовов” (англ. polymorphic inline cache)[17]. Поддержка спекулятивных оптимизаций на уровне LLVM затрудняется тем, что при использовании стороннего

[17] U. Hölzle, C. Chambers, D. Ungar, Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches, *European Conference on Object-Oriented Programming, 1991*

© National Library of Armenia

компилятора LLVM теряется контроль над процессом генерации машинного кода (например, информация о том в какой именно регистр была распределена та или иная переменная становится недоступной).

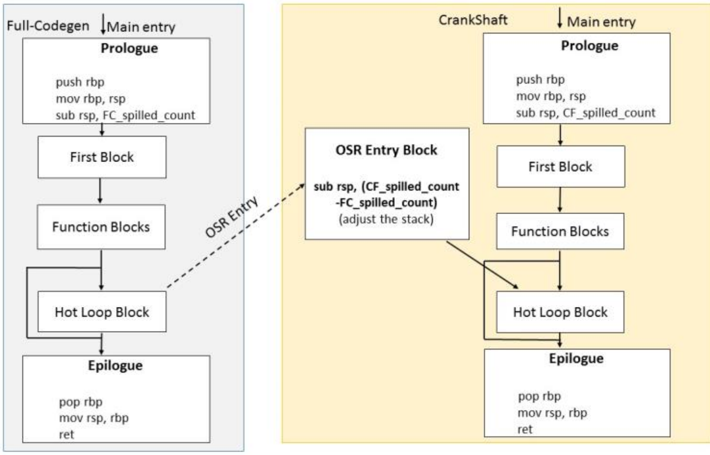

Рис. 3а. Схема переключения между уровнями Full-Codegen и Crankshaft

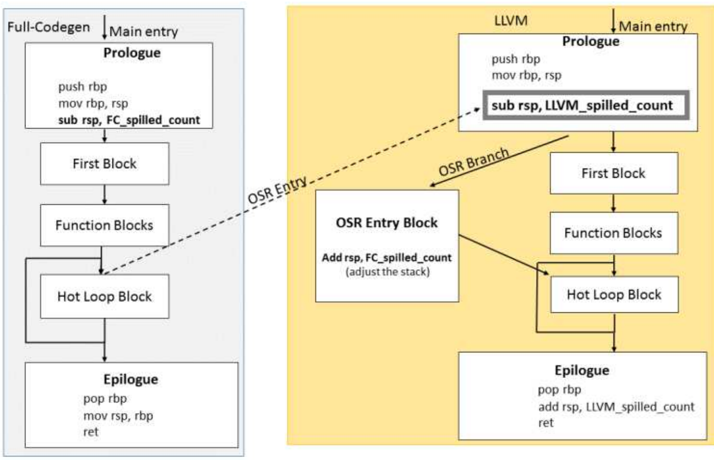

Рис. 3б. Схема переключения между уровнями Full-Codegen и LLVM

Между тем, в LLVM реализованы средства для контроля и модификации сгенерированного динамическим модулем кода. Поддержка деоптимизаций на уровне LLVM была реализована с помощью функций *llvm.stackmap* и *llvm.patchpoint*[18], которые позволяют преобразовать созданный компилятором LLVM код на лету. Обе эти функции

---

[18] Страница документации LLVM: http://llvm.org/docs/StackMaps.html

© National Library of Armenia

во время компиляции представления LLVM в машинный код инициируют создание специальной секции данных, содержащей структуру *Stack Map*. В этой структуре сохраняется вся необходимая информация для организации деоптимизаций, например, местонахождение (слот стека, имя регистра, константа и т.п.) всех значений в точке где произошла деоптимизация.

Эти функции также используются для обеспечения правильности работы программы при автоматической сборке мусора. 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. Поддержка работы сборщика мусора и перемещений объектов также была достигнута с помощью встроенной в LLVM функции llvm.patchpoint.

В разделе 3.4 приводятся выводы и основные 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 переключений между разными уровнями компиляции;
- спекулятивной компиляции и деоптимизаций.

Для оценки эффективности реализованного метода были проведены тестирования на тестовых наборах языка JavaScript. Производительность теста 3d-cube из набора LongSpider улучшилась на 62%, тесты bitops-bits-in-byte, access-nsieve и math-spectral-norm из того же набора ускорились на 27%, 17% и 8.8% соответственно.

В главе 4 рассматриваются разработанные и реализованные оптимизации многоуровневых динамических компиляторов языка JavaScript.

Раздел 4.1 содержит описание машинно-зависимой оптимизации для платформы ARM: оптимизация доступа к глобальным переменным, более эффективная реализация некоторых операций, учитывая специфику архитектуры ARM.

В разделе 4.2 описываются предложенные методы машинно-независимой оптимизации: разработка и реализация алгоритма удаления излишних вызовов функций, улучшение технологии встраивания функций, реализация алгоритма глобального распределения регистров на оптимизирующем уровне компилятора JavaScriptCore, реализация жадного алгоритма линейного сканирования на оптимизирующем уровне компилятора V8, разработка и реализация алгоритма рематериализации регистров, улучшение математических и строковых функций, поддержка новых операций на уровне оптимизирующего компилятора.

Оптимизация доступа к глобальным переменным.

После исследования сгенерированного машинного кода компилятора V8 для архитектуры ARM было обнаружено, что обращение к глобальным переменным в машинном коде раскрывается на две 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

© National Library of Armenia

переменных. Кроме того, если такая операция присутствует в цикле, то первая ее часть будет инвариантом цикла. Был разработан и реализован метод, который позволяет разделить реализацию обращения к глобальным переменным на две разные инструкции в графе управления. Данный метод позволяет организовать вынос инвариантной части обращения к памяти за пределы цикла, а также сократить количество таких обращений для каждой пары чтения/записи глобальных переменных.

Эффективная реализация некоторых операций, учитывая специфику архитектуры ARM.

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

В компиляторе V8 при реализации некоторых операций создавалось множество инструкций обращения к последовательным адресам памяти. Однако, архитектура ARM поддерживает команды, осуществляющие множественную загрузку/сохранение по последовательным адресам – LDM/STM, с помощью которых была добавлена более эффективная реализация таких операций.

Улучшение оптимизации удаления мертвого кода.

На оптимизирующем уровне DFG компилятора JavaScriptCore реализована оптимизация удаления мертвого кода, однако ее эффективность невысока. В частности, в процессе проведения данной оптимизации ни один из вызовов функций не будет удален, даже если удаление таких вызовов возможно без ущерба семантике программного кода. При этом удаление рассматриваемого оператора безопасно, если для него выполнены 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 системным вызовам, например, исключение системных вызовов “sleep” из криптографической библиотеки могло бы сделать ее уязвимой к timing-атакам).

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 организации межпроцедурного анализа и такая проверка в компиляторе DFG JIT не была реализована. Однако, в стандарте ECMA-262 JavaScript содержится описание всех стандартных методов строк, чисел и прочих встроенных объектов, по которым можно восстановить информацию об их побочных эффектах. Среди этих стандартных функций можно выделить функции без побочных эффектов, но такой подход довольно груб. Однозначно “чистых” функций немного: большинство функций требует, чтобы передаваемые аргументы удовлетворяли определенным налагаемым условиям. Для разных функций может понадобиться проверять различные утверждения. Был разработан и реализован новый алгоритм удаления вызовов библиотечных функций без побочных

© National Library of Armenia

эффектов, что привело к значительному росту производительности на тестовых наборах языка JavaScript.

Улучшение алгоритма 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. Еще одной проблемой при встраивании функций является неверное распространение информации о типах, приводящее к генерации множественных инструкций конвертации типов на некоторых тестах. Был реализован новый метод, который учитывает фактор встраивания функций при распространении информации о типах.

Алгоритм глобального распределения регистров для оптимизирующего уровня компилятора JavaScriptCore.

В результате исследований компилятора JavaScriptCore 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. Нами был реализован алгоритм линейного сканирования для глобального распределения регистров в компиляторе DFG JIT. Алгоритм позволяет избежать излишних обращений к памяти в пределах базовых блоков. При реализации алгоритма были учтены особенности многоуровневых компиляторов. В частности, выполнение программы может переключиться на оптимизирующий уровень во время выполнения функции (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.

Жадный алгоритм линейного сканирования на оптимизирующем уровне компилятора V8.

В компиляторе V8 используется алгоритм линейного сканирования для глобального распределения регистров. Для улучшения качества создаваемого машинного кода, нами был реализован жадный алгоритм линейного сканирования на уровне оптимизирующего компилятора Crankshaft. 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,

© National Library of Armenia

основываясь на размере функции или собранной статической информации о программе (Глава 2).

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 по мере использования. Но есть и второй способ: значения могут быть вычислены заново, если все нужные операнды доступны в конкретной точке программы. Данная технология называется “рематериализацией регистров”. Рассмотрим пример такой оптимизации (рис. 4).

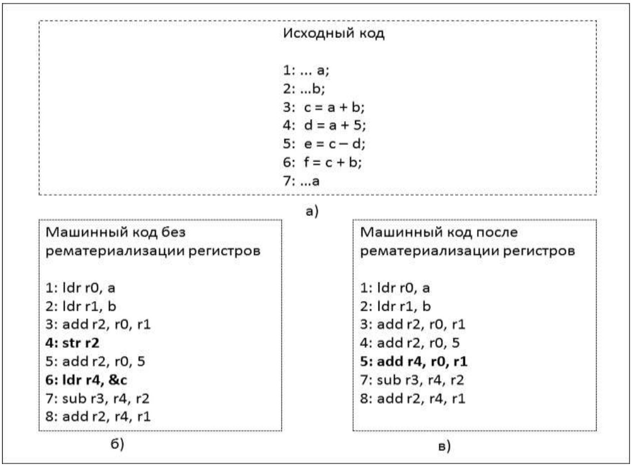

Рис. 4. Пример применения метода рематериализации регистров

Предположим, что имеются четыре регистра (r0-r3). Регистр r4 используется для загрузки значений из памяти. Значение переменной **c** можно вычислить из переменных **a** и **b**, так как эти переменные живы в точках использования **c**. Как видно из рис. 4, при помощи рематериализации регистров два обращения к памяти можно заменить на одно вычисление.

Технология рематериализации регистров используется в некоторых крупных компиляторах, таких как GCC и LLVM. Однако 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 графа зависимостей по данным и структуры для определения повторно используемых регистров (англ. register resue chain) для нахождения пригодных для рематериализации значений. Однако оптимизирующие уровни компиляторов JavaScript не поддерживают такие структуры. Конструирование таких структур для реализации известных алгоритмов рематериализации могло бы привести к большим задержкам времени компиляции, что привело бы к ухудшению производительности компилятора.

Был разработан и реализован новый алгоритм рематериализации регистров, который полностью интегрирован в процесс распределения регистров и не добавляет дополнительных расходов во время компиляции.

© National Library of Armenia

Улучшение математических и строковых функций.

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. Был реализован метод, кэширующий значения библиотечных функций String.replace, Math.sin и Math.cos. Кроме того, были предложены улучшения для алгоритма сортировки массивов в библиотечной функции Array.Sort, учитывающие размер данных при сортировке.

Добавление поддержки конструкции “for…in”, операторов Switch и Typeof на оптимизирующем уровне компилятора JavaScriptCore.

На оптимизирующем уровне компилятора JavaScriptCore не поддерживается часть операций языка JavaScript. Один из методов для увеличения эффективности многоуровневых динамических компиляторов может быть реализация поддержки новых операций на уровне оптимизирующего компилятора. Примером конструкции, которая не поддерживалась на уровне DFG JIT, является операция “for..in”. Функция, содержащая такую структуру, выполнялась только на неоптимизирующих уровнях, даже если эта функция вызывалась много раз. Внутреннее представление оптимизирующего компилятора имеет типизированную структуру (тип переменных вычисляется из профиля, собранного на нижних уровнях компиляции). Для эффективной реализации структуры “for…in” на уровне оптимизирующего компилятора, в первую очередь необходимо вычислить тип возвращаемых значений итератора. В неоптимизирующих уровнях компиляции были внесены соответствующие изменения для пополнения профиля функции информацией о возвращаемых значениях итератора “for…in”. После этого, на оптимизирующем уровне компиляции были добавлены все необходимые операции для обработки конструкции “for…in”, и теперь функции, в которых содержится такой цикл, также могут выполняться на уровне компилятора DFG JIT.

Другими примерами операций языка JavaScript, которые не поддерживались на уровне оптимизирующего компилятора JavaScriptCore, являются операторы *switch* и *typeof*, поддержка которых также была добавлена на уровне оптимизирующего компилятора DFG JIT.

Удаление излишних проверок на отрицательный ноль.

В JavaScript числа не делятся на вещественные и целые, и подразумевается поведение всех чисел как вещественных, поэтому следует различать положительный и отрицательный ноль. Например, результат выполнения следующих операций равен минус бесконечности (англ. *minus infinity*): **2 / (-0); 2 / (0 / -5); 1 / (-8 % 8)**. В случае, если во время оптимизации операции с вещественными числами заменяются операциями с целыми числами, кроме проверок переполнения, необходимо также учитывать различие отрицательного и положительного нуля, чтобы не получить неправильный результат. На оптимизирующем уровне компилятора JavaScriptCore имеется реализация проверки результата арифметических операций на отрицательный ноль. Однако в существующей реализации не был учтен тот факт, что в некоторых ситуациях проверка значения результата на отрицательный ноль может быть опущена. Например, при исполнении выражения *8/(c%d+5)*, если результатом выражения *c%d* является ноль, нет надобности различать положительный и отрицательный ноль. Нами был предложен улучшенный алгоритм для реализации проверок арифметических операций на отрицательный ноль, который позволяет избежать излишних проверок и деоптимизаций.

© National Library of Armenia

В разделе 4.3 приводятся выводы и основные результаты тестирования реализованных методов.

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;

Результаты тестирований подтверждают эффективность реализованных оптимизаций. Оптимизация доступа к глобальным переменным для архитектуры ARM ускорила производительность теста *bitops-bitwise-and* из набора *SunSpider* на 10%. Удаление избыточных вызовов функций позволило ускорить производительность теста *v8-regexp* из набора v8-v7 на 18%, а производительность теста *StringWeighted* из набора *BrowserMark* улучшилась в 3.5 раза. В среднем набор v8-v7 улучшился на 1.7%, а набор *BrowserMark* – на *12%.* Улучшение оптимизации "встраивание функции" на 7% ускорило тест math-cordic из набора SunSpider.

Реализация жадного алгоритма линейного сканирования для распределения регистров ускорила производительность тестовых наборов Octane и Kraken в среднем на 3%. Реализация алгоритма рематериализации регистров в компиляторе Crankshaft ускорила производительность тестов *audio-beat-detection* и *audio-fft* из набора Kraken на 5% и на 7% соответственно. В наборе Octane в целом удалось заменить около 200 инструкций обращения к памяти на более быстрые операции. Это позволило ускорить тест GameBoy на 2%, а тест Mandreel ускорился на 4%.

Реализация поддержки оператора '*switch'* на оптимизирующем уровне DFG компилятора JavaScriptCore привела к 39% росту производительности на тесте *StringSHA1,* а реализация оператора '*typeof'* к 17% росту производительности на тесте *ArrayBlur* из тестового набора *BrowserMark*. Реализация поддержки оператора "*for…in*" улучшила производительность теста *string-fasta* из набора *SunSpider* на 4%. 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 конкретных тестов составляет 35%. Тестирования производились на платформе ARM.

Заключение содержит выводы и направления дальнейших исследований задач, рассмотренных в работе. 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%.

© National Library of Armenia

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 [3,7,8]. 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 [5,6]. При этом была добавлена поддержка компиляции всех конструкций и динамических свойств языка JavaScript:
   - сборщика мусора и перемещений объектов;
   - бинарной совместимости и переключений между разными уровнями компиляции;
   - спекулятивной компиляции и деоптимизаций.
3. Разработан и реализован эффективный алгоритм рематериализации регистров для динамических компиляторов [4].
4. Разработан и реализован эффективный алгоритм удаления излишних вызовов функций без побочных эффектов для языка JavaScript [1,2].
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 [1,2].

Список опубликованных статей по теме диссертации:

1. Р. Жуйков, Д. Мельник, Р. Бучацкий, В. Варданян, В. Иванишин, Е. Шарыгин. Методы динамической и предварительной оптимизации программ на языке JavaScript // Труды Института системного программирования РАН, Том 26. Выпуск 1. 2014 г. С. 297- 314
2. Vardanyan V. Optimizations of JavaScript programs // GSPI's scientific journal 2014. С. 122-128
3. Zhuykov R., Vardanyan V., Melnik D., Buchatskiy R., Sharygin E. Augmenting JavaScript JIT with Ahead-of-Time Compilation // 10th International Conference on Computer Science and Information Technologies. 2015. С. 195-198.
4. Vardanyan V., Asryan S., Buchatskiy R. Integrated register rematerialization in JavaScript V8 JIT compiler // 10th International Conference on Computer Science and Information Technologies. 2015. С. 186-189.

© National Library of Armenia

5. Sargsyan S., Kurmangaleev S., Vardanyan V., Zakaryan V. Code Clones Detection Based on Semantic Analysis for JavaScript Language // 10th International Conference on Computer Science and Information Technologies. 2015. C. 182-185.
6. Варданян В., Иванишин В., Асрян С., Хачатрян А., Акопян Дж., Динамическая компиляция программ на языке JavaScript в статически типизированное внутреннее представление LLVM // Труды Института системного программирования РАН, Том 27. Выпуск 6. 2015 г. Стр. 33-48.
7. Zhuykov R., Vardanyan V., Melnik D., Buchatskiy R., Sharygin E. Augmenting JavaScript JIT with Ahead-of-Time Compilation // In Proceedings of IEEE,Computer Science and Information Technologies (CSIT), 2015, pages 116 –120.
8. Варданян В. Методы оптимизации программ на языке JavaScript, основанные на статистике выполнения программы // Труды Института системного программирования РАН, Том 28. Выпуск 1. 2016 г. Стр. 5-20.

Resume

Vahagn Vardanyan

Static optimizations for programs with dynamically typed languages

Dynamically typed languages have gain popularity during the last decade. One of the examples of such languages is JavaScript. In recent years, JavaScript has become one of the most popular web development languages. Its usage evolved beyond the small web-applications to areas such as operating systems and server-side applications. JavaScript is used in many modern and massive applications, such as Node.js, Gmail, Google docs, etc. Moreover, JavaScript is the main language for developing user applications in some operation systems (e.g. Tizen OS, Firefox OS). Hence, many modern engines for dynamically typed languages use just-in-time (JIT) compilation to produce optimized binary code. Due to increasing popularity many companies such as Apple (JavaScriptCore compiler), Google (V8 compiler), Mozilla (SpiderMonkey compiler) and Microsoft (ChakraCore compiler) have produced and are continuing to develop their own JIT-compilers for dynamically typed languages.

JIT compilers are limited in complexity of optimizations they can perform at runtime without delaying the execution. To maintain a trade-off between quick startup and doing sophisticated optimizations, compiler engines usually use multiple tiers: lower tier JITs generate less efficient code, but can start almost immediately (e.g. even with interpretation), while higher tier JITs aim at generating effective code for hot places, but at the cost of long compilation time. One of the well-known and effective methods of optimizing dynamically typed languages is speculative compilation. Design of multitier architecture allows effective implementation of speculative optimizations: during the execution on the lower tiers, profile for program is collected. This profile is used then in the higher levels of execution for organizing speculative optimizations. These optimizations are based on the assumption that the state of the program will not change during execution and usually are built on specialization of types and values. If the optimized code encounters a case it cannot handle (for example, when type of the variable does not correspond to profile information), it bails to lower tiers. Such transition is called *deoptimization*.

The performance of the multitier JIT compilers can be improved by applying optimizations on each level of execution. However, optimization implementation on one level of execution can bring negative effect on the other levels. For example, on the one hand to improve performance it is necessary to ensure execution of maximal count of code on

© National Library of Armenia

optimizing levels of compilation. On the other hand, it is important to minimize number of deoptimizations. Hence, the optimizations must be designed to consider all the levels of the multitier infrastructure entirely.

The applications written on dynamically typed langugaes are becoming massive and more complex. It is important to constantly improve multitier JIT architectures to meet requirements of the increasing complexity of applications and support new features of processors.

This work is dedicated to analysis, design and development of static optimizations for modern multi-tier dynamic compilers.

Main results of the work

- A new method for improving performance of multitier JIT compiler based on program profile information that allows execution of "hot" code immediately on the optimizing compiler level.
- A new method for dynamic compilation of JavaScript programs to the statically typed LLVM intermediate representation. Method is implemented by adding new level of compilation in V8 compiler and allows applying optimizations already implemented in LLVM to JavaScript programs.
- An effective register rematerialization algorithm for dynamically typed languages.
- An effective algorithm for eliminating redundant functions calls without side effects for JavaScript language.

Ամփոփում

Վարդանյան Վահագն Գևորգի

ԴԻՆԱՄԻԿ ՏԻՊԵՐՈՎ ԼԵԶՈՒՆԵՐԻ ՍՏԱՏԻԿ ՕՊՏԻՄԱԼԱՑՄԱՆ ՄԵԹՈԴՆԵՐԸ

Ներկայումս լայն տարածում են ստացել ոչ տիպականացված սկրիպտային լեզուներով գրված ծրագրերը: Ծրագրավորման այդտեսակ լեզուներից լայն տարածում ունի JavaScript-ը: Համակարգիչների և ներդրված համակարգերի արդյունավետության բարձրացման հետ մեկտեղ JavaScript լեզվի գործածումը հնարավոր դարձավ ոչ միայն վեբ կայքերում օգտագործվող փոքր ծավալով սկրիպտային ծրագրերում, այլ նաև այնպիսի ծավալուն ծրագրերում ինչպիսին են Gmail-ը և Google docs-ը: JavaScript լեզուն նաև օգտագործվում է Node.js սերվարային ծրագրավորման համակարգում: Ավելին, որոշ օպերացիոն համակարգեր JavaScript լեզուն օգտագործում են որպես հիմնական ծրագրավորման լեզու: Այդպիսի համակարգերի օրինակներ են Tizen և FirefoxOS օպերացիոն համակարգերը: Որպես հետևանք՝ առաջանում է սկրիպտային ծրագրերի արդյունավետության բարձրացման պահանջարկ:

Դինամիկ տիպերով լեզուների ժամանակակից իրականացումների մեծամասնությունը մեքենայական կոդի գեներացման համար օգտագործում է դինամիկ կոմպիլյացիաի մեթոդը: Դինամիկ կոմպիլյացիայի ընթացքում կարևոր է ապահովել հավասարակշռություն օգտագործվող օպտիմալացումների բարդության և ծրագրի մեկնարկի հետաձգման միջև: Հավասարակշռություն ապահովելու նպատակով օգտագործվում է բազմամակարդակ դինամիկ կոմպիլյացիաի մեթոդը, որը թույլ է տալիս առանց հետաձգման սկսել ծրագրի կատարումը կոմպիլյացիայի

© National Library of Armenia

առաջին մակարդակում (այս մակարդակում կատարվում են միան պարզ օպտիմալացումներ), իսկ հաճախ օգտագործվող կոդի հատվածների թարգմանությունը կատարվում է օպտիմալացնող մակարդակներում ավելի որակյալ մեքենայական կոդ ստանալու նպատակով:

Բազմամակարդակ ճարտարապետության օգտագործումը թույլ է տալիս նաև արդյունավետ իրականացել սպեկուլյատիվ օպտիմալացումներ, ինչը դինամիկ տիպերով ծրագրերի արագագործության բարելավման կարևոր մեթոդներից մեկն է: Սպեկուլյատիվ օպտիմալացումները հիմնվում են ծրագրի կատարման ժամանակ ձեռք բերված փոփոխականների տիպերի և արժեքների վերաբերյալ տեղեկատվության վրա: Բազմամակարդակ ճարտարապետությունը թույլ է տալիս հավաքել ծրագրի փոփոխականների մասին տեղեկատվությունը կատարման առաջին մակարդակներում և օգտագործել այդ տեղեկատվությունը կատարման բարձր մակարդակներում սպեկուլյատիվ օպտիմալացումներ իրականացնելու համար: Այդ օպտիմալացումների կատարումը հիմնված է այն ենթադրության վրա, որ տվյալների տիպերը ծրագրի կատարման ընթացքում փոփոխության չեն ենթարկվի, հակառակ դեպքում ծրագրի կատարումը տեղափոխվում է կատարման առաջին մակարդակ: Այս պրոցեսը կոչվում է դեօպտիմալացում (deoptimization).

Ժամանակակից բազմամակարդակ կոմպիլյատորների օրինակներ են Google ընկերության կողմից ստեղծված V8 կոմպիլյատորը, և Apple ընկերության կողմից ստեղծված JavaScriptCore-ը: Չնայած բազմաթիվ տարբերություններին իրականացման և մակարդակների քանակի մեջ, այս կոմպիլյատորները ընդհանուր առմամբ օգտագործում են նույնանման բազմամակարդակ կառուցվածք, ինչը թույլ է տալիս մշակել ընդհանուր, նմանատիպ մեթոդներ արդյունավետության բարելավման համար:

Դինամիկ տիպերով լեզուներով գրված ծրագրերը դառնում են ավելի ծավալուն և բարդ: Անհրաժեշտ է անընդհատ բարելավել բազմամակարդակ կոմպիլյատորների ենթակառուցվածքները՝ հաշվի առնելով պրոցեսորների նոր հնարավորությունները և ծրագրերի աճող բարդությունը:

Ատենախոսության նպատակն է գնահատել դինամիկ տիպերով ծրագրավորման լեզուներով գրված ծրագրերում ստատիկ օպտիմալացումների կիրառման սահմանները:

Ատենախոսության հիմնական արդյունքներն են՝

- Մշակվել և իրականացվել է բազմամակարդակ դինամիկ կոմպիլյատորների օպտիմալացման նոր մեթոդ, հիմնված ծրագրի կատարման մասին տեղեկատվության պահպանման և օգտագործման վրա, որը թույլ է տալիս սկսել հաճախ կատարվող կոդի հատվածների կատարումը օպտիմալացնող կոմպիլյատորի մակարդակում:
- Մշակվել և իրականացվել է նոր մեթոդ, որը թույլ է տալիս JavaScript լեզվով գրված ծրագրերը թարգմանել LLVM կոմպիլյատորի ստատիկ ներքին ներկայացման: Մեթոդը իրականացվել է V8 կոմպիլյատորում թարգմանության նոր մակարդակ ավելացնելու միջոցով և թույլ է տալիս կիրառել LLVM

© National Library of Armenia

կոմպիլյատորում իրականացված բոլոր օպտիմալացումները JavaScript լեզվով գրված ծրագրերի արտադրողականության բարելավման համար:

- Մշակվել և իրականցվել է առանց կողմնակի ազդեցության ֆունկցիաների ավելորդ կանչերի հեռացման ալգորիթմ JavaScript լեզվի համար:
- Մշակվել և իրականացվել է ռեգիստրների ռեմատերիալիզացիայի արդյունավետ ալգորիթմ դինամիկ բազմամակարդակ կոմպիլյատորների համար:

© National Library of Armenia