УДК 004.43

Д. Е. Бежецков

Новосибирский государственный университет
ул. Пирогова, 2, Новосибирск, 630090, Россия

dima00782@gmail.com

РАЗРАБОТКА СРЕДЫ ИСПОЛНЕНИЯ ЯЗЫКА JAVASCRIT ДЛЯ АРХИТЕКТУРЫ ЭЛЬБРУС

Рассматривается разработка и реализация среды исполнения языка JavaScript для отечественной платформы Эльбрус. Эльбрус представляет собой архитектуру с длинным командным словом (VLIW). Новая платформа имеет повышенные характеристики безопасности за счет своего аппаратного устройства, а именно защищенный стек для хранения адресов возврата функций и тегирование команд. Это позволяет использовать процессоры Эльбрус для нужд госсектора, обороны и в других специальных областях.

Ключевые слова: Эльбрус, JavaScript, виртуальные машины, V8.

Введение

С момента производства первого компьютера прошло уже больше полувека, и электронные устройства стали частью нашей жизни. Мы читаем новости в Интернете, делаем расчеты на компьютере. Трудно переоценить влияние компьютеров на современный мир. Основной и самой главной частью компьютера является процессор. Фактически он и делает всю полезную работу.

Не так давно на российском рынке появилась линейка процессоров Эльбрус, от компании МЦСТ. Новый процессор призван решить две проблемы: импортозамещение и безопасность.

Все чаще с политической арены мы слышим, что против России вводятся санкции, которые запрещают ввоз той или иной продукции в РФ. Наша страна, к сожалению, до недавнего времени не могла производить процессоры для большинства компьютеров в стране, так что в случае, если какой-нибудь крупный зарубежный производитель откажется поставлять процессоры, страна окажется в довольно затруднительном положении. Поэтому иметь в производстве свой процессор просто необходимо для нашей страны.

С другой стороны, в современном мире все чаще возникает необходимость защищать важную информацию от третьих лиц. Это особенно важно для информации, составляющей государственную тайну. Чтобы обеспечить надежную защиту, необходимо использовать проверенные аппаратные решения.

Чтобы исправить текущее положение дел, компания МЦСТ и выпустила новый процессор Эльбрус, весь цикл производства которого сосредоточен в России и может быть проверен на каждом этапе производства – от составления схемы до выпекания платы. Кроме того, процессор представляет собой новую архитектуру, которая кардинально отличается от устоявшихся архитектур, таких как x86_64, arm64 и др. Новая архитектура предлагает аппаратно решать известные проблемы с безопасностью.

Бежецков Д. Е. Разработка среды исполнения языка JavaScrit для архитектуры Эльбрус // Вестн. НГУ. Серия: Информационные технологии. 2017. Т. 15, № 2. С. 5–12.

ISSN 1818-7900. Вестник НГУ. Серия: Информационные технологии. 2017. Том 15, № 2

В недавнем исследовании рынка используемых языков [1] язык JavaScript в очередной раз занял лидирующее место. Социальные сети, онлайн трансляции спортивных мероприятий – все это было бы сложно представить себе без использования Интернета, а следовательно, и без языка JavaScript.

Разработанный всего 22 года назад язык для описания анимаций на странице, JavaScript стал почти доминирующим языком по количеству программистов, пишущих на нем [2]. Реализация среды исполнения языка JavaScript де-факто является обязательным для всех десктопных архитектур, и для Эльбруса в том числе.

На Эльбрусе в данный момент уже есть среда исполнения JavaScript – интерпретатор из проекта spiderMonkey, браузера Firefox. К сожалению, интерпретатор не удовлетворяет нуждам клиентов, а именно работает довольного медленно и практически никак не учитывает специфику архитектуры Эльбрус. В связи с этим автором была поставлена задача реализовать эффективную среду исполнения языка JavaScript для архитектуры Эльбрус.

Реализация на основе V8

Для реализации среды исполнения JavaScript на Эльбрусе была выбрана V8. Это среда исполнения JavaScript с открытым исходным кодом, поддерживаемая компанией Google и используемая в браузере Chrome. Выбор основывался на показателях набора промышленных тестов производительности [3]. V8 показывает самую большую производительность на них, и, кроме того, архитектура V8 отличается от традиционной архитектуры виртуальной машины, что способствует более производительной реализации для Эльбруса.

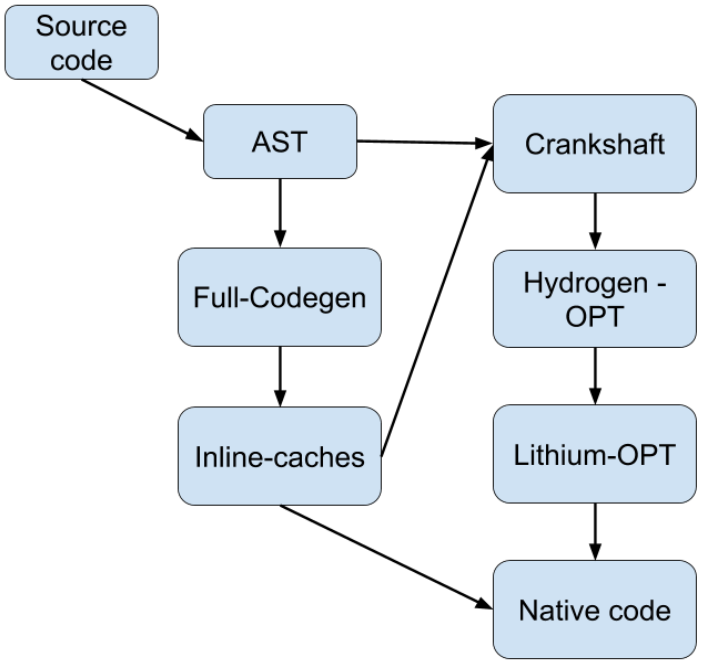

Рис. 1. Архитектура V8

На рис. 1 представлена архитектура V8. Машина не имеет байткода и интерпретатора, вместо этого у нее есть два JIT компилятора – FullCodegen и Crankshaft. Первый компилирует код быстро, не применяя при этом почти никаких оптимизаций. Второй компилирует с применением всех возможных оптимизаций. Первый компилятор вставляет инструментирующий код, который запоминает для каждой операции языка JavaScript типы, участвовавшие в этой операции. Данная техника называется встроенными кэшами. Она позволяет сократить в реальном коде динамичность языка JavaScript, что дает возможность использовать эту информацию о типах для оптимизации.

На уровне исходного кода V8 делится на две части: платформозависимую и платформонезависимую. Сделано это для облегчения портирования на новые архитектуры. Платформозависимая часть представляется в виде интерфейса, который необходимо реализовать. После того как исходный код на языке JavaScript разбирается в абстрактное синтаксическое дерево,

[1] Stackoverflow. URL: https://insights.stackoverflow.com/survey/2016 (дата обращения 17.02.2017).
[2] Ibid.
[3] Octane. URL: https://chromium.github.io/octane (дата обращения 17.02.2017).

почти для каждого узла необходимо написать генератор. Этот генератор должен по заданному узлу генерировать готовый код под конкретную архитектуру. Поскольку узлы в этом дереве представляют собой довольно крупные функциональные блоки языка JavaScript, то и возможностей для применения оптимизаций на таком блоке для Эльбруса больше, что хорошо сказывается на итоговой производительности.

Адаптация стеков

Среда исполнения V8 рассчитана на то, что все поддерживаемые ею архитектуры так или иначе в работе со стеком схожи с x86. Это, например, означает, что адрес возврата из функции лежит сразу после базового указателя фрейма. К сожалению, такое поведение учитывается и в платформонезависимом коде V8, который имеет внушительный объем.

Для того чтобы не переписывать платформонезависимый код в данной работе было решено использовать стек пользовательских данных и два глобальных регистра, эмулирующие rsp и rbp, чтобы не сильно отличаться от x86. Адрес возврата из функции тоже относительно легко определить во время компиляции как адрес следующей за вызовом широкой команды.

Одной из особенностей является то, что Эльбрус поддерживает аппаратно информацию о фрейме. Эти данные хранятся в защищенном стеке связующей информации и недоступны для программиста из непривилегированного режима процессора. При каждом вызове Эльбрус запоминает в этом стеке пару значений – адрес возврата из функции и количество байтов, аллоцированных в стеке пользовательской информации. При выходе из функции аппаратура получает эту информацию, переходит по заданному адресу и откатывает стек пользовательской информации на заданное количество байт.

Получается, с одной стороны, у нас есть фрейм для JavaScript функции, который располагается в стеке пользовательских данных, и, с другой стороны, у нас есть аппаратный фрейм, который поддерживается аппаратурой. В данной работе был выбран вариант отображения 1 – 1, т. е. каждому JavaScript фрейму соответствует ровно один аппаратный фрейм. Это позволяет более просто писать код, но проявляет следующую проблему.

Чтобы положить какое-нибудь значение на стек, необходимо заранее аллоцировать в нем память с помощью инструкции getsp, на x86, например, этого делать не надо. Проблема заключается в том, что мы должны во время компиляции определить размер для фрейма функции на JavaScript. Исходный код может быть сложным, и определить, сколько точно понадобится места, мы не можем, а можем подсчитать лишь максимум места. Соответственно если код функции пошел не по той ветке исполнения, где этот максимум достигается, то во фрейме оказывается дыра между текущим фреймом функции и следующим.

Еще одной особенностью Эльбруса является то, что нельзя аллоцировать память по 1 байту, только по 2. Так что даже если мы идеально определить количество слотов в стеке (и если это количество не кратно двум), то все равно будет зазор. Дыры между фреймами не так страшны, и обычно это всего 8 байт, но когда происходит обращение из текущего фрейма к предыдущему из платформонезависимого кода, могут возникать проблемы. К счастью, V8 обращается к предыдущему фрейму лишь в одном случае: для получения адреса возврата. Поэтому, чтобы положить адрес возврата из функции в текущий фрейм, автором было предложено передавать адрес через регистр, и уже в новом фрейме – на стек пользовательской информации. В итоге получается простое для написания кода и для отладки решение, ведь доступный gdb оперирует аппаратными фреймами.

Обработка исключений

Исключения известны в программировании давно и предоставляют программисту удобный способ обрабатывать исключительные ситуации в коде. Исключения поддерживают почти все современные динамические языки, в том числе и язык JavaScript.

Рассмотрим реализацию механизма обработки исключений в V8 на примере:

```
function foo() { throw 42; }
function bar() {
  try {
    foo();
  } catch (e) {
    print(e);
  }
}
```

На входе в блок *try* V8 создаст новый обработчик исключений и слинкует его в список текущих обработчиков. Также в этот обработчик запишется текущее значение верхушки стека (rsp), указателя на фрейм (frame pointer) и адреса блока *catch*. Когда исполнение дойдет до инструкции *throw 42;*, среда исполнения перейдет из JavaScript кода в C код и выполнит следующий алгоритм:

1) поиск подходящего обработчика исключений;
2) подмена текущего *rsp* и *rbp* на соответствующие значения из обработчика;
3) подмена адреса возврата на адрес блока *catch*;
4) возврат.

Из-за того, что стек, который хранит адреса возвратов из функций, недоступен программисту на Эльбрусе, действия 3 и 4 невозможны без участия операционной системы, так как изменять значения в стеке связующей информации можно только в привилегированном режиме процессора.

Было решено использовать системный вызов, который позволяет менять адреса возврата в защищенном стеке на адрес специального стаба, в который возвратится исполнение V8 после C вызова *throw*. Этот стаб содержит только одну инструкцию *return*, которая отматывает все аппаратные фреймы до заданного.

Решение не обладает большой производительностью, но и исключительные ситуации в коде должны происходить крайне редко.

Применение свертки условных выражений и спекулятивности

Так как Эльбрус поддерживает исполнение инструкций под предикатами и имеет 32 предикатных регистра, то при реализации среды исполнения V8 можно применить свертку условных выражений.

Свертка условных выражений для Эльбруса позволяет удалить дорогостоящие переходы и сократить количество широких команд. Рассмотрим простой пример с циклом:

```
for (var i = 0; i < N; ++i) {
  if (cond)
    a = x;
  else
    a = y;
  b = a + 2;
}
```

Используя предикатный регистр, можно трансформировать цикл в следующий эквивалентный:

```
for (I = 0; I < N; ++i) {
  pred1 = cond;
  a = x ? pred1
  a = y ? ~pred1
  b = a_2 + 2;
}
```

В данном примере присваивание переменной *a* исполняется под предикатом 1 и его отрицанием. Из примера видно, что теперь тело цикла состоит только из двух широких команд и не содержит переходов, что, во-первых, увеличивает количество инструкций в одной широкой команде, а во-вторых, позволяет применять дальнейшие оптимизации по раскрутке цикла.

Следует заметить, что инструкции на Эльбрусе могут исполняться под предикатом или его отрицанием либо вообще без предиката, и, следовательно, мы можем вычислять одновременно 65 потоков управления программы.

Рассмотрим другой пример:

```
function foo(a, b) {
  let c;
  if (a !== null) {
    c = a.getValue();
  } else {
    c = b.getValue();
}
```

Наивная реализация предполагает использование 4 широких команд и как минимум один переход по условию. Так как Эльбрус поддерживает явную спекулятивность на уровне команд, то мы можем сделать вызов и загрузку из памяти еще до проверки того, что *a* не является *null*:

```
function foo(a, b) {
  let c;
  let d = a.getValue(); | speculative
  let e = b.getValue(); | speculative
  if (a !== null) {
    c = a.getValue();
  } else {
    c = b.getValue();
}
```

Дальше можно применить свертку условных выражений с помощью предикатных регистров и получить следующий код:

```
function foo(a, b) {
  let c = a.getValue(); | speculative
  let d = b.getValue(); | speculative
  p1 = (a === null);

  c = d ? p1
}
```

В данном коде всего две широких команды в отличие от 4-х в наивной реализации. Эти две оптимизации применялись при компилировании среды исполнения V8, бóльшая часть которой написана на JavaScript. Следует отметить что, эти оптимизации не учитывались в реализованном компиляторе Full-Codegen, потому что он должен работать быстро и не проводить практически никаких оптимизаций (кроме встроенных КЭШей), которые тормозили бы загрузку скриптов.

Сравнение производительности

Для сравнения производительности использовались тесты с открытым исходным кодом octane [4]. Тестирование проводилось на машине с 4-ядерным процессором Эльбрус E2S monokub, с тактовой частотой 750 МГц и 24 Гб оперативной памяти DDR2, работающей на частоте 667 МГц. Сравниваться будут интерпретатор из проекта spiderMonkey, который в настоящее время является единственным способом исполнения JavaScript на Эльбрусе, и базовый не оптимизирующий компилятор из V8 Full-Codegen, реализованный автором.

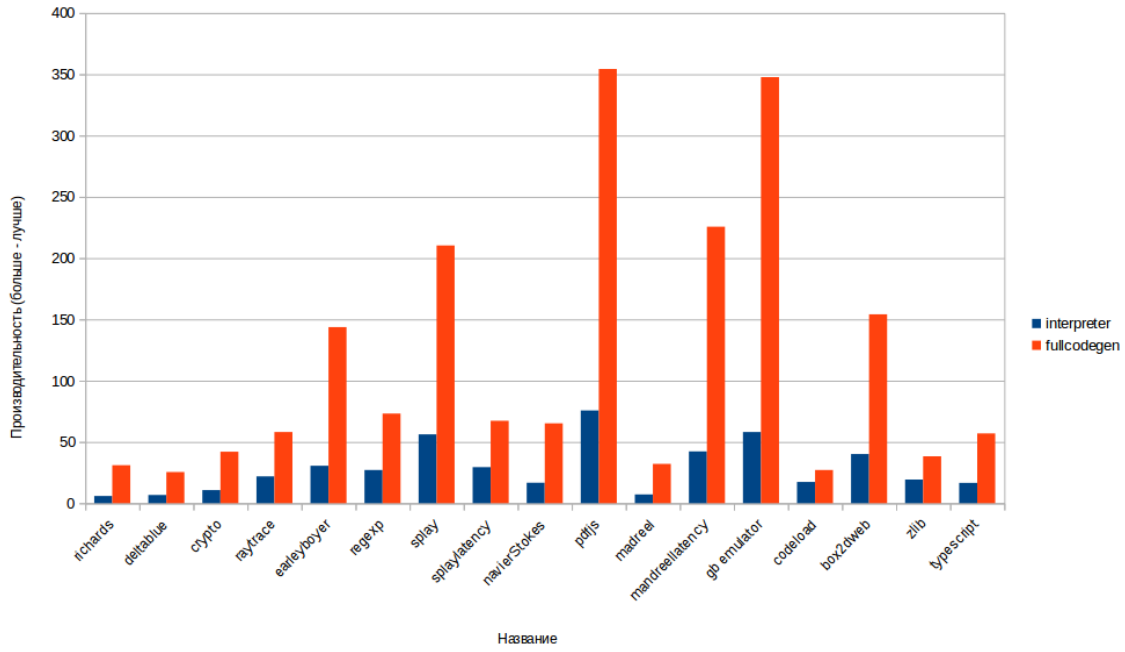

Рис. 2. Сравнение интерпретатора и базового компилятора

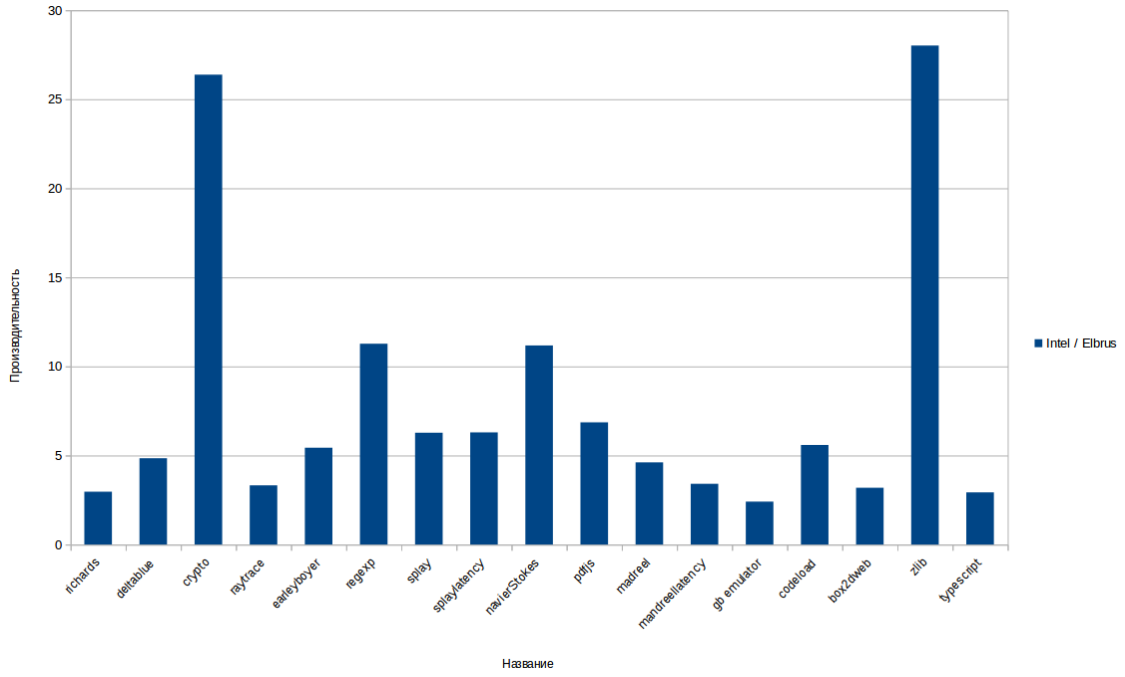

Рис. 3. Сравнение с Intel

[4] URL: https://chromium.github.io/octane

На рис. 2 видно, что компилятор значительно выигрывает за счет компиляции в бинарный код и примененных оптимизаций к самой среде исполнения. Отметим, что наибольший вклад в этот прирост производительности дают встроенные кэши.

Также интерес представляет сравнение с x86_64 архитектурой. Для теста использовался Intel Core i7-4770 с 3.4 ГГц и 16 Гб оперативной памяти DDR3 с частотой 1 600 МГц.

На рис. 3 представлено отношение результатов octane компилятора Full-Codegen на intel core i7 и на E2S. Видно, что Эльбрус проигрывает в разы, но стоит учитывать то, что техпроцесс и тактовая частота core i7 и E2S тоже сильно различаются. Конечно, тактовая частота – это не оправдание для автора, поэтому предложенная реализация будет дальше оптимизироваться. Мы только начинаем понимать, как нужно писать код для VLIW архитектуры. Следует также отметить, что и техпроцесс Эльбруса не стоит на месте, и компания МЦСТ выпустила процессор Эльбрус с тактовой частотой 1400 МГц.

Также на рис. 3 видно сильное замедление на бенчмарках crypto и zlib, связано это с тем, что intel поддерживает аппаратно команды для работы с AES и битами (BMI set), которых пока еще нет на Эльбрусе.

Заключение

В ходе работы над статьей, автором были достигнуты следующие результаты.

- Портирован базовый компилятор Full-Codegen для среды исполнения V8. Исследованы различия работы со стеком, исключениями, поддержкой сборщика мусора для архитектур x64 и Эльбрус.
- Реализация полностью соответствует стандарту ECMAScipt [5], проходит сертификационные тесты [6] и соответственно может называться JavaScript виртуальной машиной.
- Полученная реализация показывает увеличение производительности примерно в 4,3 раза относительно стандартного платформенно-независимого C++ интерпретатора JavaScript [7] на Эльбрус.
- Исследованы и применены алгоритмы оптимизаций для VLIW архитектур.

В заключение можно сказать следующее. Несмотря на то, что Эльбрус пока сильно проигрывает в производительности аналогичным по цене процессорам, комфортно работать на нем в браузере уже можно. Реализация очень динамического языка JavaScript для такой экзотической архитектуры, как Эльбрус, представляет академический интерес, а также тестирует V8 с интересной стороны.

Материал поступил в редколлегию 15.03.2017

[5] ECMAScript 2015 Language Specification. URL: http://www.ecma-international.org/ecma-262/6.0 (дата обращения 17.02.2017).

[6] Test262. URL: https://github.com/tc39/test262 (дата обращения 17.02.2017).

[7] SpiderMonkey. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey (дата обращения 17.02.2017).

D. E. Bezhetskov

Novosibirsk State University
2 Pirogov Str., Novosibirsk, 630090, Russian Federation

dima00782@gmail.com

IMPLEMENTATION OF JAVASCRIPT FOR ELBRUS ARCHITECTURE

The JavaScript (JS) language is the most popular language for web development in the world that is used by many modern web application such as Gmail, Google search engine, social networks etc. Elbrus is a new Russian CPU created for fast and secure computation. Implementation of JS virtual machine is usually obligatory for a modern hardware platform such as Elbrus. The solution is to write new a JS engine based on V8 Google engine. In that case we can significantly reduce time of development by reusing existing parts of V8. To sum up, this approach allows us to learn internals of V8 engine and introduce the first implementation of the JS language for the first modern Russian processor.

Keywords: Elbrus, JavaScript, virtual machines, V8.