17. Динамическая компиляция

Интерпретируемые языки и VM

- **Интерпретируемые языки**
  - Основные особенности:
    - Управление памятью (сборка мусора, контроль доступа к объектам, границ при обращении к массивам, и т.п.)
    - Динамические типы (классы могут изменяться, а также создаваться новые во время выполнения)
    - Создание нового кода во время выполнения (*eval*)
  - Эти особенности делают статическую компиляцию затруднительной или невыгодной
  - Примеры: JavaScript, Python, Ruby (и в некоторой степени Java)

Примеры динамических свойств языка

- **Наследование в JS**
  - «Родитель» - объект, а не класс
  - Если поля или метода нет в самом объекте, оно ищется в базовом - «прототипе»:

```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal;

var x = rabbit.eats; // true
```

  - Прототип может быть переопределен во время исполнения!

Примеры динамических свойств языка

- **Наследование в JS**
  - «Родитель» - объект, а не класс
  - Если поля или метода нет в самом объекте, оно ищется в базовом - «прототипе»:

```
var foo = {name: "foo", one: 1, two: 2};
var bar = {two: "two", three: 3};
bar.__proto__ = foo;
console.log("one = " + bar.one);   // from foo
console.log("two = " + bar.two);   // from bar (overridden)
console.log("three = " + bar.three); // from bar (newly added)
```

```
$ d8 ex-proto.js

one = 1
two = two
three = 3
```

  - Прототип может быть переопределен во время исполнения!

Интерпретация vs JIT компиляция

- Стандартный подход к реализации среды выполнения:
  - Программа компилируется в *байт-код* – набор простых инструкций, напоминающих ассемблер
  - Байт-код интерпретируется в виртуальной машине, которая также управляет динамическими объектами и предоставляет доступ к функциям времени выполнения

- JIT (Just-In-Time) Compilation – компиляция «на лету»:
  - Вместо интерпретации, байт-код сначала компилируется в код целевой архитектуры
  - Функции компилируются по мере необходимости, а не все заранее
  - Используется профилирование и спекулятивные оптимизации

JIT компиляция

- Преимущества JIT-компилятора:
  - Выполняет оптимизацию кода
  - Может использовать внутреннее представление более низкого уровня, и более подходящее для оптимизаций, чем байт-код (например, SSA)
  - Может оптимизировать программу под конкретные входные данные, т.к. работая во время выполнения, располагает данными о профиле программы («горячие» места, типы данных, значения переменных, вероятности переходов)
  - Если профиль изменился, может «на лету» перекомпилировать программу

JIT компиляция

- Недостатки:
  - По сравнению с «обычным» (статическим) компилятором, сильно ограничен в сложности выполняемых оптимизаций, т.к. не должен задерживать выполнение программы
- Решение:
  - Оптимизировать только самые «горячие» места
  - Многоуровневый JIT: каждый уровень обрабатывает все более «горячие» места, сложность оптимизаций нарастает
  - Делать сложные оптимизации для следующего уровня параллельно с исполнением неоптимизированного кода на предыдущем уровне

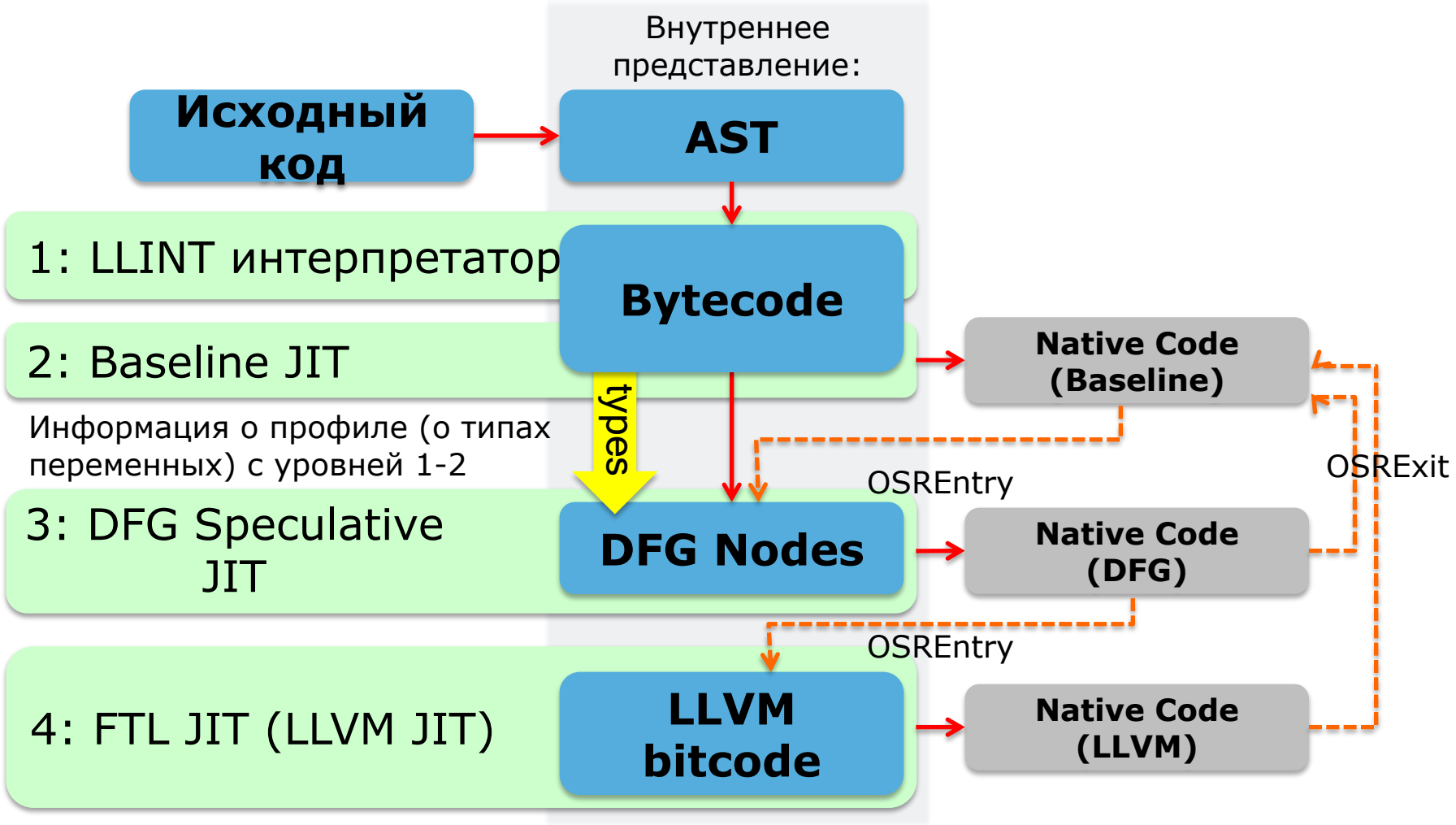
Современные JavaScript-«движки»

- Самые популярные open-source проекты:
  - **SFX** (JavaScriptCore)
    - Используется в Safari и других браузерах на базе WebKit (BlackBerry)
    - Составная часть браузерного движка WebKit, разрабатывается Apple
  - **V8**
    - Используется в Google Chrome, встроенном браузере Android, а также Node.js
    - Основной JavaScript-движок в Blink (изначально был заменой для SFX в WebKit), разрабатывается Google
  - Mozilla SpiderMonkey
    - JS движок в Mozilla FireFox
- Общие особенности SFX и V8
  - Многоуровневый JIT, каждый уровень имеет собственное внутреннее представление и набор оптимизаций
  - Используется профилирование и спекулятивные оптимизации
  - Всего в ~2 раза медленнее C++ кода (SunSpider benchmark)

Особенности JIT

- Многоуровневый JIT
  - 1-й уровень: JIT с быстрой компиляцией (либо даже интерпретатор) – включает только самые простые оптимизации. Сразу же начинает выполнять байткод, и собирает информацию о профиле программы
  - 2-й уровень: оптимизирует только «горячие» места, выполняет более сложные оптимизации с использованием профиля, использует собственное внутреннее представление (например, SSA), может быть спекулятивным
  - Возможно большее число уровней, а также переключение между компиляторами разных уровней при изменении профиля

Устройство Webkit JavaScriptCore



Итеративный интерпретатор байтового кода

```
// Итеративный интерпретатор байтового кода на языке Си
// с явным циклом
void Interpreter(void)
{
    const byte *ip;
    // …
    for (;;) {
        switch (*ip++) {
        case iadd: {
            const int a = pop();
            const int b = pop();
            push(a + b) break;
        }
        // …
        }
    }
}
```

Итеративный интерпретатор байтового кода

```
// Итеративный интерпретатор байтового кода на ассемблере
// с явным циклом
InterpreterLoop:
    tmp = *ip++;
    jmp bytecode_table[BytesInWord*tmp];
...
L_iadd:
    pop tmp1;
    pop tmp2;
    tmp1 += tmp2;
    push tmp1;
    jmp InterpreterLoop;
...
Bytecode_table: .word ..., L_iadd, ...
```

Итеративный интерпретатор байтового кода

```
// Итеративный интерпретатор байтового кода на ассемблере
// с неявным циклом
next: macro()
    tmp = *ip++;
    jmp bytecode_table[BytesInWord*tmp];
endm

Interpreter:
    next();
...
L_iadd:
    pop tmp1;
    pop tmp2;
    tmp1 += tmp2;
    push tmp1;
    next();
...
Bytecode_table: .word ..., L_iadd, ...
```

Данная реализация позволяет сократить накладные расходы на выполнение избыточных инструкций диспетчеризации

После исполнения текущей инструкции интерпретатор не возвращается в общую точку (switch),
а переходит сразу на сервисную процедуру следующей инструкции

Представление данных внутри компилятора

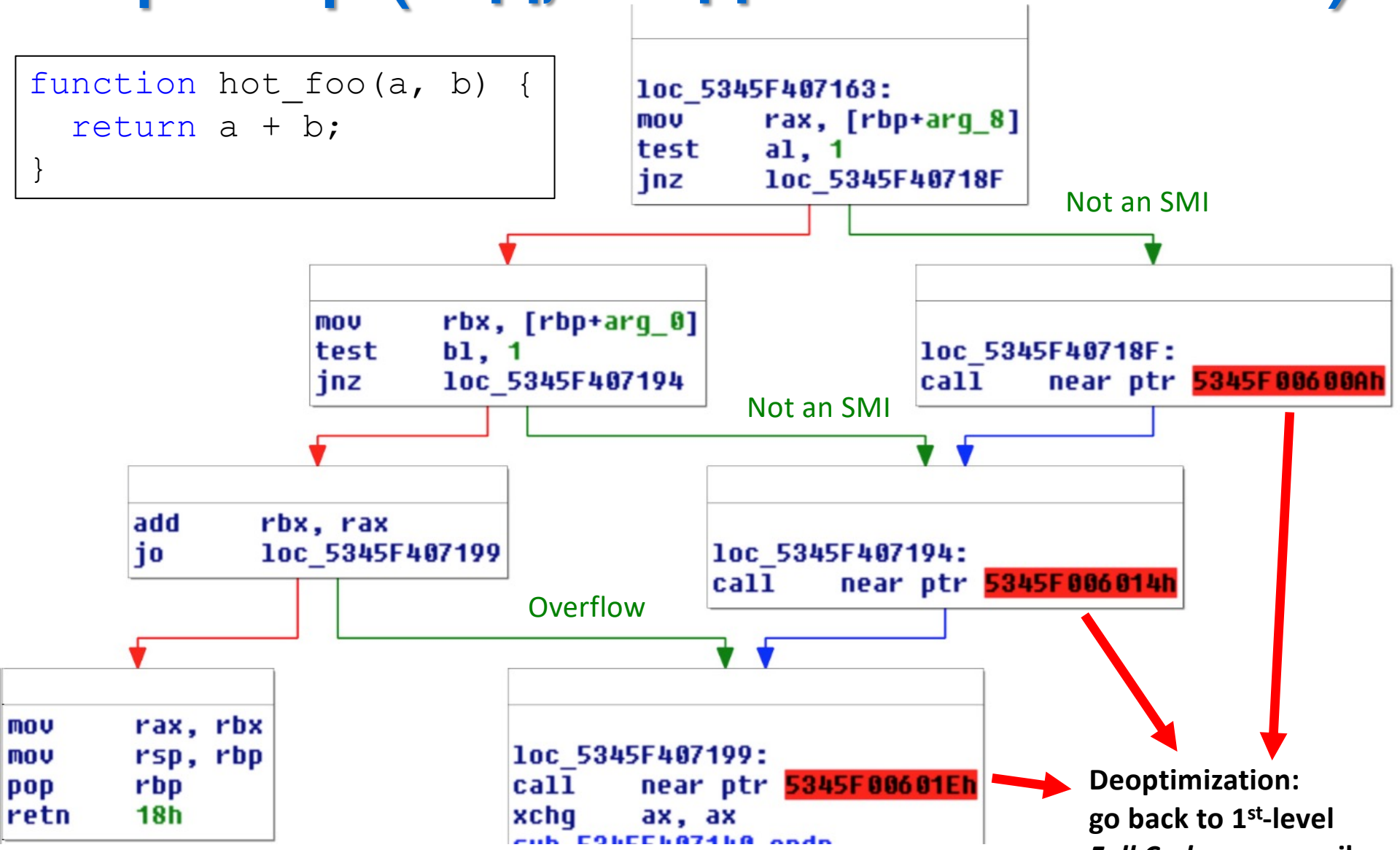
- JavaScriptCore:
  - Все значения - вещественные double
  - 64-битные double: 1 знак + 11 экспонента + 52 бит мантисса
  - Для остальных типов используется специальное значение стандарта IEEE754 QNaN:
    - старшие 12 бит установлены в 1
    - младшие биты могут быть использованы для представления целых и указателей на объекты
  - В целочисленной арифметике можно использовать только младшие 32-битные половины чисел, и хранить их на целочисленных регистрах

Представление данных в V8

- Факт: все указатели выровнены – их значения в бинарном виде – *чётные*
- Как это используется в V8:
  - *Нечетные* значения представляют указатели на *boxed* объекты (младший бит обнуляется перед использованием)
  - *Чётные* значения представляют ”small 31-bit integers” (на 32-битной архитектуре)
    - Настоящее значение числа получается сдвигом влево на 1 бит (т.е. x2)
    - Вся арифметика корректна, а переполнения проверяются аппаратно

Пример (код, созданный LLVM JIT)

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



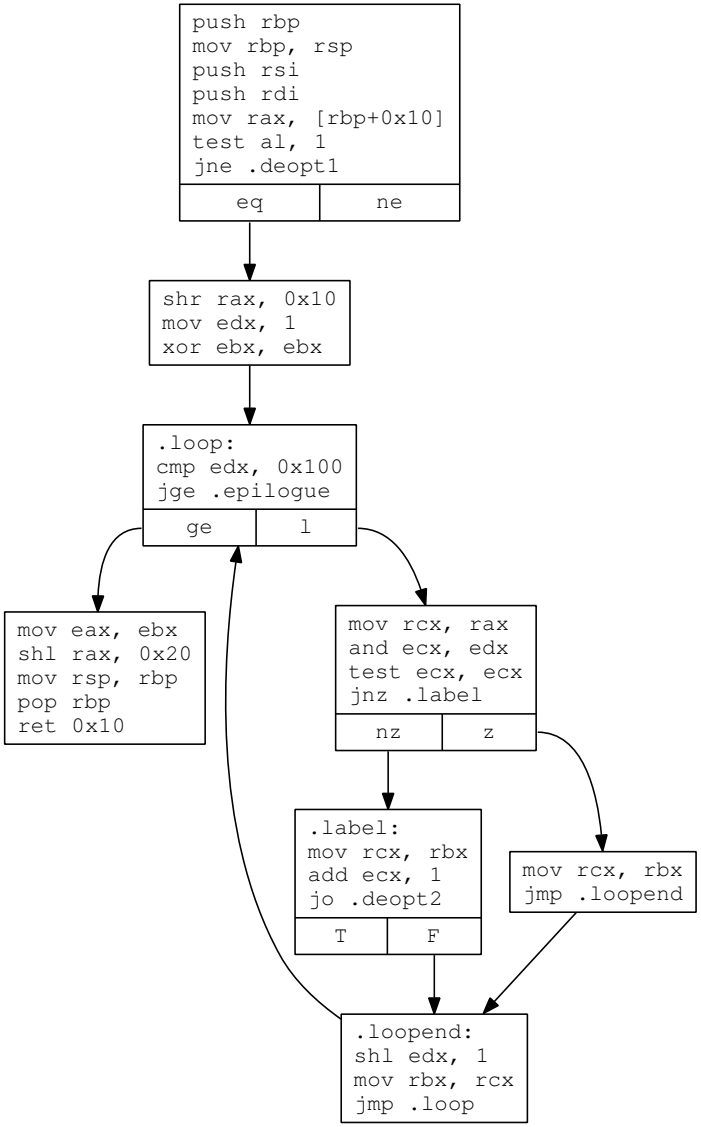
Пример из бенчмарка SunSpider

```
function foo(b) {
    var m = 1, c = 0;
    while(m < 0x100) {
        if(b & m) c++;
        m <<= 1;
    }
    return c;
}
```

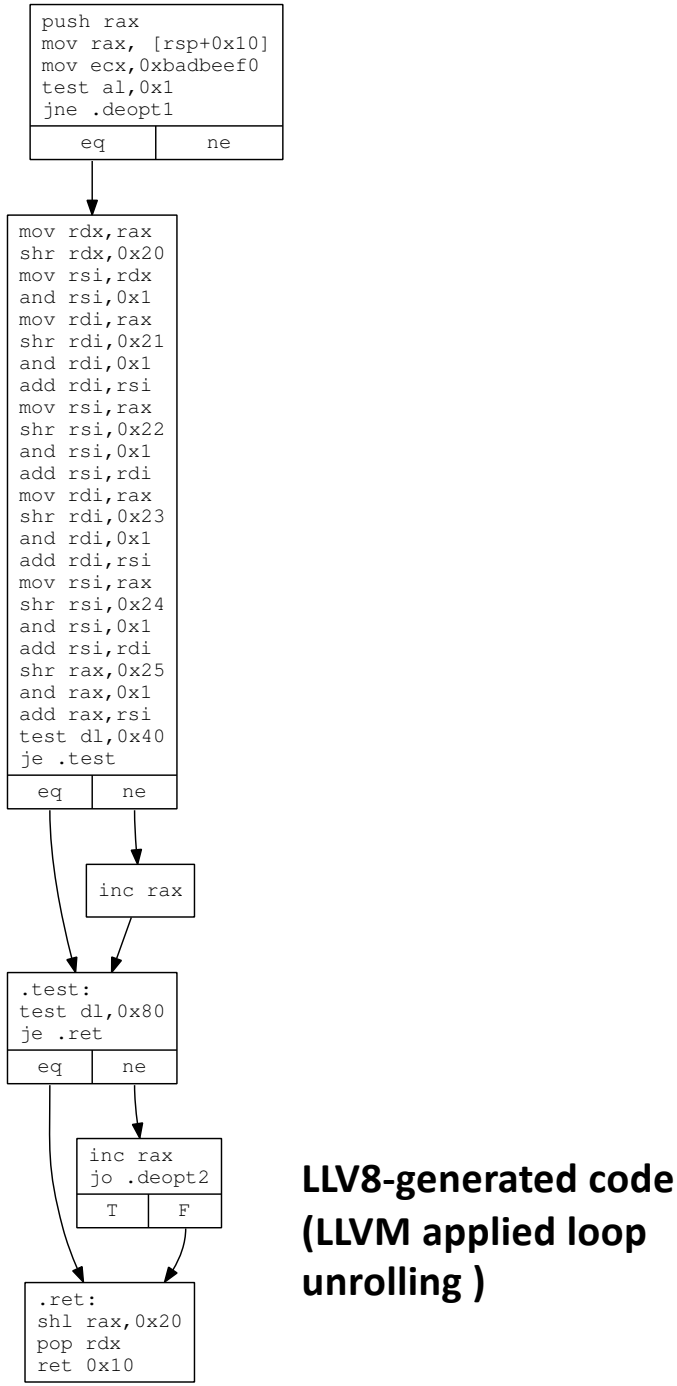
```
function TimeFunc(func) {
    var sum = 0;
    for(var x = 0; x < ITER; x++)
        for(var y = 0; y < 256; y++)
            sum += func(y);
    return sum;
}

result = TimeFunc(foo);
```

SunSpider test: bitops-bits-in-byte.js

Iterations	x100	x1000
Execution time, **Crankshaft**, *ms*	0.19	1.88
Execution time, **LLV8**, *ms*	0.09	0.54
Speedup, times	**x2.1**	**x3.5**



Original V8 CrankShaft's code



LLV8-generated code (LLVM applied loop unrolling)

Компиляция JavaScript в JSC: LLINT и Baseline JIT

JavaScript:

```
function foo() {
  var i, s;
  for (i = 0; i < 10000; i++) {
    s += i;
  }
  return s;
}
```

байткод LLINT / Baseline JIT:

```
[   0] enter
[   1] mov          r0, Int32: 0(@k0)
[   4] jnless           r0, Int32: 10000(@k1), 16(->20)
[   8] loop_hint
[   9] add          r1, r1, r0
[  14] pre_inc          r0
[  16] loop_if_less     r0, Int32: 10000(@k1), -8(->8)
[  20] ret          r1
```

Компиляция JavaScript в JSC: DFG JIT (SSA)

11: < 1:2> GetLocal(@10, JS, r1(E), bc#9) predicting StringIntDoublerealDoublenanOther
0x7f86bc7bc8fe: mov 0x8(%r13), %rcx
13: < 2:3> GetLocal(@12, JS, r0(H<Int32>), bc#9) predicting Int
0x7f86bc7bc902: mov 0x0(%r13), %ebx
14: <!1:2>ValueAdd(@11, @13<Int32>, JS|MustGen|MightClobber|MayOverflow, bc#9)
0x7f86bc7bc906: or %r14, %rbx
0x7f86bc7bc909: mov %rcx, 0x10(%r13)
0x7f86bc7bc90d: mov %rbx, 0x18(%r13)
0x7f86bc7bc911: mov %rcx, %rsi
0x7f86bc7bc914: mov %rbx, %rdx
0x7f86bc7bc917: mov %r13, %rdi
0x7f86bc7bc91a: mov $0x9daff0, %r11
0x7f86bc7bc924: mov $0xbadbeef, (%r11)
0x7f86bc7bc92b: mov $0x9daff4, %r11
0x7f86bc7bc935: mov $0xbadbeef, (%r11)
0x7f86bc7bc93c: mov $0x0, -0x2c(%r13)
0x7f86bc7bc944: mov $0x7f8700738685, %r11
0x7f86bc7bc94e: call %r11
0x7f86bc7bc951: xor %esi, %esi
0x7f86bc7bc953: mov $0x9dc908, %r11
0x7f86bc7bc95d: mov (%r11), %r11
0x7f86bc7bc960: test %r11, %r11
0x7f86bc7bc963: jnz 0x7f86bc7bca05
15: < 2:-> SetLocal(@14, r1(E), bc#9) predicting StringIntDoublerealDoublenanOther
0x7f86bc7bc969: mov %rax, 0x8(%r13)

// r0 <- i (Int32)
// r1 <- s (Unknown)

Компиляция JavaScript в JSC: DFG JIT (SSA)

19: < 1:2>JSConstant(JS, $3 = Int32: 1, bc#14)
20: <!2:2>ArithAdd(@13<Int32>, @19<Int32>, Number|MustGen|CanExit, bc#14) // i++
0x7f86bc7bc96d: mov 0x18(%r13), %rsi
0x7f86bc7bc971: mov %rsi, %rdi
0x7f86bc7bc974: add $0x1, %edi
0x7f86bc7bc977: jo 0x7f86bc7bca17
21: < 1:-> SetLocal(@20<Int32>, r0(H<Int32>), bc#14) predicting Int
0x7f86bc7bc97d: mov %edi, 0x0(%r13)
22: < 1:3>JSConstant(JS, $1 = Int32: 10000, bc#16)
23: <!1:3>CompareLess(@20<Int32>, @22<Int32>, Boolean|MustGen|MightClobber, bc#16)
0x7f86bc7bc981: cmp $0x2710, %edi
0x7f86bc7bc987: jl 0x7f86bc7bc8fe
24: <!0:-> Branch(@23<Boolean>, MustGen|CanExit, T:#1, F:#3, bc#16)

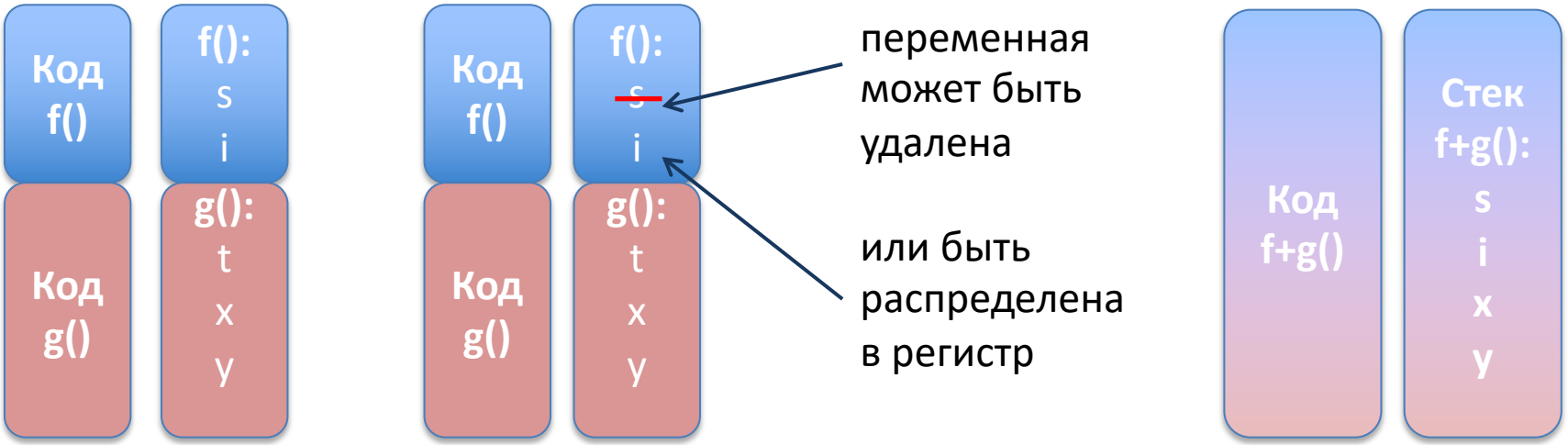
Замена на стеке (On-Stack Replacement)

```
function g(t) {
  var x, y;
  …
}

function f() {
  for (var i = 0; i < 10000; i++) {
    s = g(i);
  }
}
```

i=2000 в DFG JIT
принято решение
о встраивании g() в
f():

i=1000: переход на
DFG JIT:

Baseline JIT:

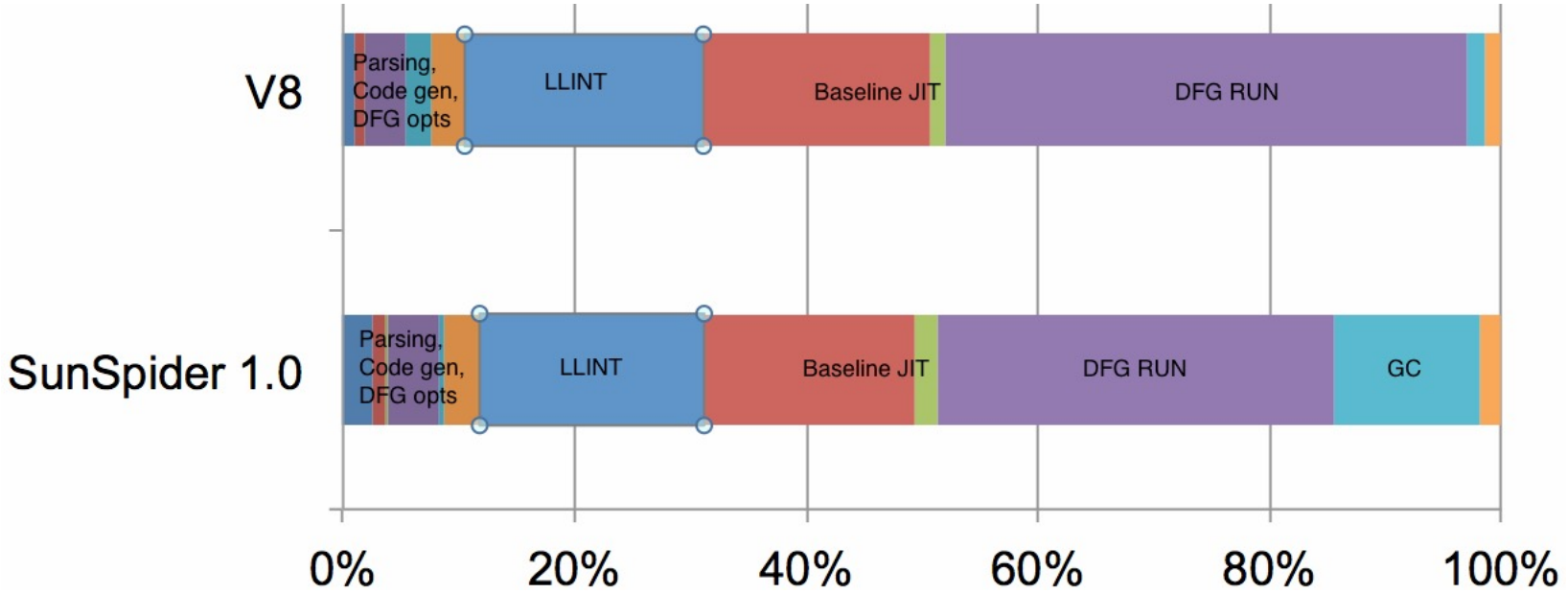


По-разному оптимизированный код работает с разной раскладкой стека

Особенности JIT

- Спекулятивный JIT
  - Специализирует код для данных, полученных при профилировании (прежде всего типы объектов). Для обработки остальных случаев предусмотрены проверки, возвращающие выполнение на предыдущий уровень JIT
  - Например, если профилирование показывает, что код работает только с целыми числами, можно использовать целочисленную арифметику и «обычные» регистры процессора, а результаты операций проверять на переполнение, и если оно произошло, продолжить выполнение на «базовом» JIT, код которого работает для любых типов объектов

Анализ работы JavaScriptCore



- Построение AST
- Создание Bytecode
- Генерация BaselineJIT кода
- Компиляция DFG первый раз
- Перекомпиляция DFG
- SpeculativeJIT
- [Run] LL Interpreter
- [Run] BaselineJIT
- [Run] BaselineJIT after DFG
- [Run] DFG JIT
- Garbage Collection
- Miss

Компоненты JSC: сравнение

Время выполнения мульти-язычного теста PL benchmark от автора Martin Richards

Тест	Время, мс
Язык C	1.2
JavaScript обычный интерпретатор	129
LLINT интерпретатор	58
Baseline JIT	8.4
DFG JIT	2.1

Сравнение производительности уровней JIT у JavaScriptCore

Test	V8-richards speedup, times		Browsermark speedup, times	
	Relative to interpreter	Relative to prev. tier	Relative to LLINT	Relative to prev. tier
JSC interpreter	1.00	-	n/m	-
LLINT	2.22	2.22	1.00	-
Baseline JIT	15.36	6.90	2.50	2.5
DFG JIT	61.43	4.00	4.25	1.7
Same code in C	107.50	1.75	n/m	-

Особенности оптимизации JavaScript

0 / -1 = ?

Особенности оптимизации JavaScript

0 / -1 = ?

1 / (0 / -1) = ?

Особенности оптимизации JavaScript

0 / -1 = -0

1 / (0 / -1) = -INF

Отрицательный ноль

```
function foo() {
  var x = 1;
  var y = -1;
  var s;
  var i;
  for (i = 0; i < 2001; i++) {
     if (i == 2000) x = 0;
     s = 1/(x/y);
  }
  return s;
}
var a = (1/(0/-1)).toString();
var b = foo().toString();
```

В выражении 1/(x/-1) переменная x меняет значение с 1 на 0.

Аналогичные примеры строятся для выражений

1/(-x) и 1/(x/-1+(-0)), а так же для выражения 1/(x % 4), в последнем x меняет значение с 1 на -4

Проверка на отрицательный ноль

Результатом должно быть –Infinity
1 / (-0) ; 1 / (0 / -3) ; 1 / (-4 % 4)

Проверки при целочисленном делении
x % y равно -0, когда result == 0 и x < 0
x / y равно -0, когда x == 0 и y < 0

Проверка не всегда необходима:
5 / (A%B + 3)