

Архитектура ARM и WebKit

Дмитрий Мельник

dm@ispras.ru

Архитектура ARM

- ARM: Advanced RISC Machine
 - Область применения: встраиваемые системы
 - Разработчик: компания ARM Holdings, лицензирует дизайн процессора производителям оборудования
- Основные особенности:
 - Энергоэффективность
 - Низкая стоимость
 - Относительно простое ядро
 - Расширяемость

Архитектура ARM

Архитектура	Семейство процессоров	Год	Примеры устройств
ARMv1	ARM1	1985	
ARMv2	ARM2, ARM3		
ARMv3	ARM6, ARM7	1992	
ARMv4	StrongARM, ARM7TDMI, ARM9TDMI	2003	iPaq 4150
ARMv5	ARM7EJ, ARM9E, ARM10E, XScale		
ARMv6	ARM11	2007	iPhone (orig, 3G)
ARMv7	Cortex A8, A9, A15, A7	2008	N900, Galaxy 1-4, iPhone (3GS, 4, 5)
ARMv8	Cortex A57	2011	iPhone 5S

RISC vs CISC

(Reduced vs Complex Instruction Set Computer)

- Простые операции
 - Ограниченный набор простых команд (например, нет деления)
 - Команда выполняется за один такт
 - Фиксированная длина команды (простота декодирования)
- Конвейер
 - Каждая операция разбивается на однотипные простые этапы, которые выполняются параллельно
 - Каждый этап занимает 1 такт, в т.ч. декодирование
- Регистры
 - Много однотипных взаимозаменяемых регистров (могут использоваться и для данных, и для адресации)

RISC vs CISC

(Reduced vs Complex Instruction Set Computer)

- Модель работы с памятью
 - Отдельные команды для загрузки/сохранения в память
 - Команды обработки данных работают только с регистрами
- Сложность оптимизаций перенесена из процессора в компилятор
 - Производительность сильно зависит от компилятора
- Итого: более простое ядро, выше частота процессора

Режимы процессора ARM

- ARM (32 бит)
 - Режим по умолчанию
 - Доступны все команды процессора, работа с сопроцессором
- Thumb-1 (16 бит)
 - Высокая плотность кода, лучшее использование I-Cache
 - На 16-битной памяти команда передается за один такт
 - Ограниченный набор команд и регистров
- Thumb-2 (смешанный 16/32 бит)
 - Объединяет преимущества ARM и Thumb-1
 - Размер кода на 25% меньше при сравнимой производительности
- Jazelle (8 бит)
 - Аппаратно реализовано свыше 60% команд Java-байткода
 - Режим доступен только производителям оборудования

Регистры

- 16 регистров общего назначения
 - Размер: 32 бита, используются в целочисленных командах, полностью взаимозаменяемые
 - Именование: r0 - r15
 - Некоторые регистры имеют специальные имена и назначение:
 - PC (r15) – Program Counter
 - LR (r14) – Link Register
 - SP (r13) – Stack Pointer

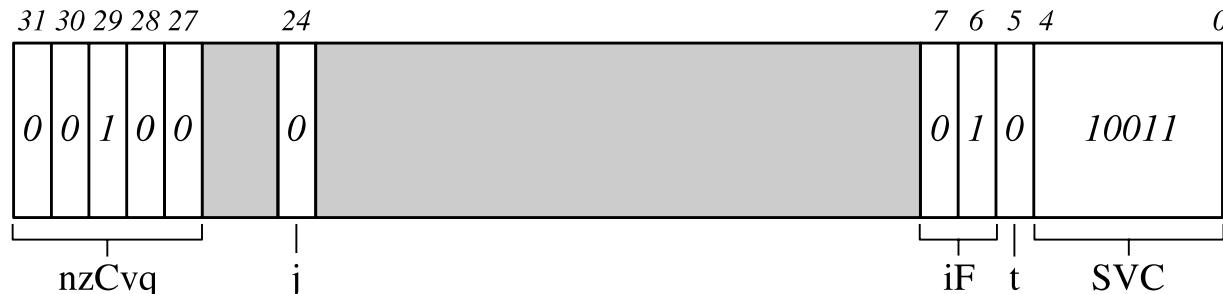
Регистры

- Регистры состояния
 - CPSR (Current Program Status Register): флаги, обработка прерываний
 - SPSR (Saved Program Status Register): хранит копию CPSR при обработке прерывания
 - Доступ: команды MRS/MSR
- Контрольные регистры
 - Изменения параметров кэширования, управления памятью, режимов процессора и т.п.
 - Доступ: команды MRC/MCR
- Вещественные регистры (FPU и векторного сопроцессора)

Условные флаги

Флаг	Описание	Когда устанавливается
Q	Saturation	Насыщение/переполнение в DSP-операциях
V	oVerflow	Переполнение (знаковое)
C	Carry	Перенос бита (беззнаковый)
Z	Zero	Нулевой результат (равенство)
N	Negative	Отрицательное значение (бит 31 установлен)

Флаги расположены в регистре CPSR (current program status register)



Установка флагов

Если команда ALU имеет суффикс 'S', то флаги будут установлены в соответствии с результатом команды.

Примеры:

```
subs r1, r2, #1
```

```
lsls r1, r2, #5
```

```
cmp r2, #1
```

Коды условий

Предикат	Отрицание	Флаги	
EQ	NE	Z	==
CS / HS	CC / LO	C	>= (беззнаковое)
MI	PL	N	< 0
VS	VC	V	переполнение
HI	LS	zC	> (беззнаковое)
GE	LT	NV nv	>= (знаковое)
GT	LE	NzV nzv	> (знаковое)
AL	-	-	Безусловная команда

Условное выполнение

Почти все команды ARM могут быть записаны в *условной форме*. В этом случае условие приписывается после команды, и она будет выполнена, только если условие истинно.

Примеры:

```
addge r1, r1, r1
```

вычисление модуля $|r1 - r2|$:

```
subs r1, r1, r2  
rsblt r1, r1, #0
```

Модуль сдвига

Второй аргумент ALU-команд может быть представлен в виде

$ARG2 = R \ shift_op \ B$, где

R – регистр

B – величина сдвига (0-31)

$shift_op$ – один из LSL , ASL , RSL , ROR или RRX

Пример:

`add r1, r1, r1 lsl #2 //r1 = r1 + r1 *4 = r1 * 5`

Представление констант

Второй аргумент ALU-команд может быть также константой, которая кодируется 12 битами (8 бит значение, 4 бита величина сдвига):

$$\text{CONST_32} = \text{CONST_8} \ll (2 * N), \quad 0 \leq N < 16$$

Примеры правильных и неправильных констант:

and r1, r1, #255

and r1, r1, #510

and r1, r1, #0xff00ff00

and r1, r1, #0xff000000

Представление констант

Второй аргумент ALU-команд может быть также константой, которая кодируется 12 битами (8 бит значение, 4 бита величина сдвига):

$CONST_32 = CONST_8 \ll (2 * N), 0 \leq N < 16$

Примеры правильных и неправильных констант:

and r1, r1, #255

and r1, r1, #510 // $510 = 255 \ll 1$ – нечетный сдвиг

and r1, r1, #0xff00ff00 // значение «шире» 8 бит

and r1, r1, #0xff000000

Кодировка команд

ADDGE r0, r1, r2 asr #31

Действие:

```
if (flags_match (GE))
    r0 = r1 + r2 >> 31;
```

ADD{S}<c> <Rd>, <Rn>, <Rm>{, <shift>}

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

cond	0	0	0	0	1	0	0	S	Rn	Rd	imm5	type	0	Rm
------	---	---	---	---	---	---	---	---	----	----	------	------	---	----

GE ALU ADD 0 1 0 31 ASR 2
(w/reg)

Действие:

```
r0 = r1 & ~0xff000000;
set_flags();
```

BICS r0, r1, #FF000000

ADD{S}<c> <Rd>, <Rn>, #<const>

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

cond	0	0	1	0	1	0	0	S	Rn	Rd	imm12
------	---	---	---	---	---	---	---	---	----	----	-------

Режимы процессора ARM

	ARM	Thumb-1	Thumb-2
Размер команды	32 бит	16 бит	16/32 бит
Кол-во команд	~60	30	~60
Условное выполнение	Почти все команды	Нет	С помощью IT-блоков
Сдвиг аргументов	Во всех командах ALU	Отдельные команды	В 32-битных командах
Доступные регистры	15 общего назначения + pc	8 общего назначения + 7 специальных + pc	15 общего назначения + pc
Примеры команд	add r1, r2, r3 asl #2 (A)	add r1, r2 (B)	Обе формы допустимы; Размер (A) – 16 бит, (B) – 32 бит

Задания

1. Что делают команды

```
adds      r1,  r0,  #42
bicge    r0,  r0,  r0,  asr #31
```

(Эквивалент на С: if ($x \geq -42$) { ... })

2. Вычислить $X = X * 81$ в две команды без использования MUL
($81 = 5 * 16 + 1$ или $(8 + 1)^2$)

3. Записать в одну команду выражение

$$X = (X \geq 0) ? X : X - 1$$

(Использовать ADD, ASR)

Команды работы с памятью

Команды загрузки/сохранения LDR / STR:

```
ldr r1, [r2, #+/-imm12]  
ldr r1, [r2, +/-r3, shift imm5]
```

Примеры:

```
ldr r1, [pc, #256]  
ldr r1, [sp, r2, asl #2]
```

L1:

```
ldr r1, L1 + 248      // по L1 + 248 находится таблица адресов  
ldr pc, [r1, r2, asl #2] // table-jump для switch
```

Команды работы с памятью

Преинкремент / постинкремент адресного регистра:

```
ldr r1, [r2, +/-r3, shift imm5]!  
ldr r1, [r2], +/-r3, shift imm5
```

Множество регистров:

```
ldmia r1, {r1-r16}  
ldmia r1!, {r1-r16} - с обратной записью в адресный регистр
```

Суффиксы:

ia – increment after

db – decrement before

Множество регистров: любое подмножество в порядке возрастания номеров (например, {r0, r1-r10, pc})

Команды работы со стеком

Работа со стеком реализуется через команды

stmdb/ldmia

push {r0-r15}

pop {r0-r15}

stmdb sp!, {r0, r15}

ldmia sp!, {r0, r15}

sp → 0x100	Последнее доступное значение
0x0FC	r0
0x0F8	r1
...	...
0x0C0 ← sp (после)	r15

Вызов функций

Вызов функций выполняется при помощи команды
`bl` – *branch with link*

Текущее значение регистра `pc` сохраняется в
регистре `lr`

Возврат из функции выполняется с помощью
команды `bx lr`

```
some_func:  
    push {lr}  
    ...  
    pop {pc}
```

```
bl some_func  
some_func:  
    bx lr
```

Конвейер современного ARM

- Cortex-A8
 - Конвейер из 13 стадий
 - 2 АЛУ устройства, 1 Load/Store, 1 Multiply
 - Может выполнять до 2-х команд за такт
 - Из них должно быть не более одной Load/Store и Multiply, причем умножение должно идти первым
- Cortex-A9: добавлено динамическое переупорядочение команд

Примеры описания времени выполнения команд на конвейере

- Этапы конвейера обозначаются E1, E2, ..., E5
- Перед началом каждого этапа команда может требовать готовности операндов в регистрах, а после – выдавать готовые операнды
- ADD r1, r2, r3
 - r2, r3: требуется перед E2
 - r1: готов после E2
- MOV r1, r2 asl #const
 - r2: требуется перед E1
 - r1: готов после E1
- Следствие 1: mov r2, r1; add r3, r2, r1 могут начать выполняться одновременно
- Следствие 2: add r1, r2, r3; mov r2, r1 (в обратном порядке) имеют задержку в 2 такта между командами

Селективный планировщик

-02

```
.L11:
    ldr      ip, [r3, r7]
    ldr      r4, [r3, r6]
    add    r3, r3, #4
    cmp    r3, r5
    mla    r2, r4, ip, r2
    mla    r1, r2, r1
    bne    .L11

    5 cycles
    2 cycles
    ldr
    ldr
    add
    cmp
    mla
    mla
    bne
```

До:
12 тактов

Source

```
int foo(int N) {
    int i;
    int a=0, b=1;
    for (i=0; i<N; i++)
    {
        a += x[i] * y[i];
        b *= a;
    }
    return a+b;
}
```

Selective Scheduling

```
Prologue
    ldr      r4, [r3, r7]
    mov    ip, #1
    ldr      r5, [r3, r8]
    mov    r2, r3

.L15:
    mla    r2, r5, r4, r2
    cmp    r1, r6
    mov    r3, r1
    ldrne r4, [r3, r7]
    addne r1, r3, #4
    ldrne r5, [r3, r8]
    mul    ip, r2, ip
    bne    .L15
```

После:
8 тактов

Ускорение
8%

Селективный планировщик выполняет конвейеризацию циклов

- «Скрывает» задержки у долгих команд, перенося их на предыдущую итерацию цикла
- Помогает использовать параллелизм на уровне команд
- Среднее ускорение 1-3%

Улучшения кодогенерации для ARM Advanced SIMD (NEON)

- Более полная поддержка команд NEON**

Например, поддержка комбинирования $|a - b|$ в одну команду NEON:

vsub
vabs  vabd

Производительность: **+2.5%** (x264)
Уменьшение размера: **0.1%** (x264)

- Улучшение распределения регистров для векторов констант**

Запрет на размещение векторных констант в обычных ARM регистрах

Производительность:
+3% (evas)

- Поддержка преобразования из float в int в векторизации**

Циклы, в которых используется преобразование типов, теперь могут быть векторизованы с использованием команды vcvt

Производительность: **+9%** (libmp3lame)

Улучшение оптимизации GCSE и комбинирования команд в GCC

Исходный пример

```
if (x)
    a = b + c << 2;
else
    d = e + c << 2;
```

Было в GCC (неверно)

```
mov r1, r2, asl #2
cmp r3, #0
bne L1
add r4, r5, r1
b .L2
.L1:
add r6, r7, r1
```

После исправления

```
cmp r3, #0
bne L1
add r4, r5, r2, asl #2
b .L2
.L1:
add r6, r7, r2, asl #2
```

Проблема:

- Оптимизация GCSE (Global Common Subexpression Elimination) «не знает» о возможности ARM barrel shifter
- Комбинирование команд (которое о нем «знает») работает только в предлах базового блока

Решение:

- Исправить GCSE
- Доработать комбинирование

Результаты:

- Сокращение размера кода: **3.6 Кб** на SPEC 2K INT (**0.1%**)
- Используется меньше регистров

Улучшения обработки условных команд Thumb-2

Проблемы:

- Команда внутреннего представления RTL `if-then-else` раскрывается напрямую в ассемблер слишком поздно
- Слишком ранняя оптимизация коротких Thumb-2 команд мешает последующему преобразованию в условную форму
- IT блоки могут быть разделены в планировщике

Решения:

- Раньше раскрывать `if-then-else` в RTL
- Изменить порядок оптимизаций в GCC
- Отдавать приоритет командам с тем же предикатом в планировщике

RTL псевдо-код

```
a = (x == 0) ? 1 : 2;  
b = (x == 0) ? 3 : 4;
```

До

```
cmp r1, #0  
ite eq  
moveq r2, #1  
movne r2, #2  
ite eq  
moveq r3, #3  
movne r3, #4
```

После

```
cmp r1, #0  
itete eq  
moveq r2, #1  
movne r2, #2  
moveq r3, #3  
movne r3, #4
```

Улучшения обработки условных команд Thumb-2

Другие улучшения:

- Преобразовывать максимум 4 команды, чтобы не увеличивать код
- Не преобразовывать команды в условную форму, если вероятность одной дуги значительно выше другой (напр. 90% к 10%)

Исходный пример

```
cmp r1, #0
bne .L2
mov r1, #1
mov r2, #2
mov r3, #3
mov r4, #4
mov r5, #5
.L2:
...
...
```

Преобразован- ный в условную форму

```
cmp r1, #0
itttt eq
moveq r1, #1
moveq r2, #2
moveq r3, #3
moveq r4, #4
it eq
moveq r5, #5
...
...
```

Условный переход
удаляется,
один IT блок - ОК

Преобразова-
ние более 5
команд
«стоит»
лишней IT-
команды

Улучшения обработки условных команд Thumb-2

Другие улучшения:

- Преобразовывать максимум 4 команды, чтобы не увеличивать код
- **Не преобразовывать команды в условную форму, если вероятность одной дуги значительно выше другой (напр. 90% к 10%)**

Исходный пример

Если вероятность перехода высокая, то лучше полагаться на branch prediction

```
cmp r1, #0
bne .L2
mov r1, #1
mov r2, #2
mov r3, #3
mov r4, #4
mov r5, #5
.L2:
...
...
```

Преобразованный в условную форму

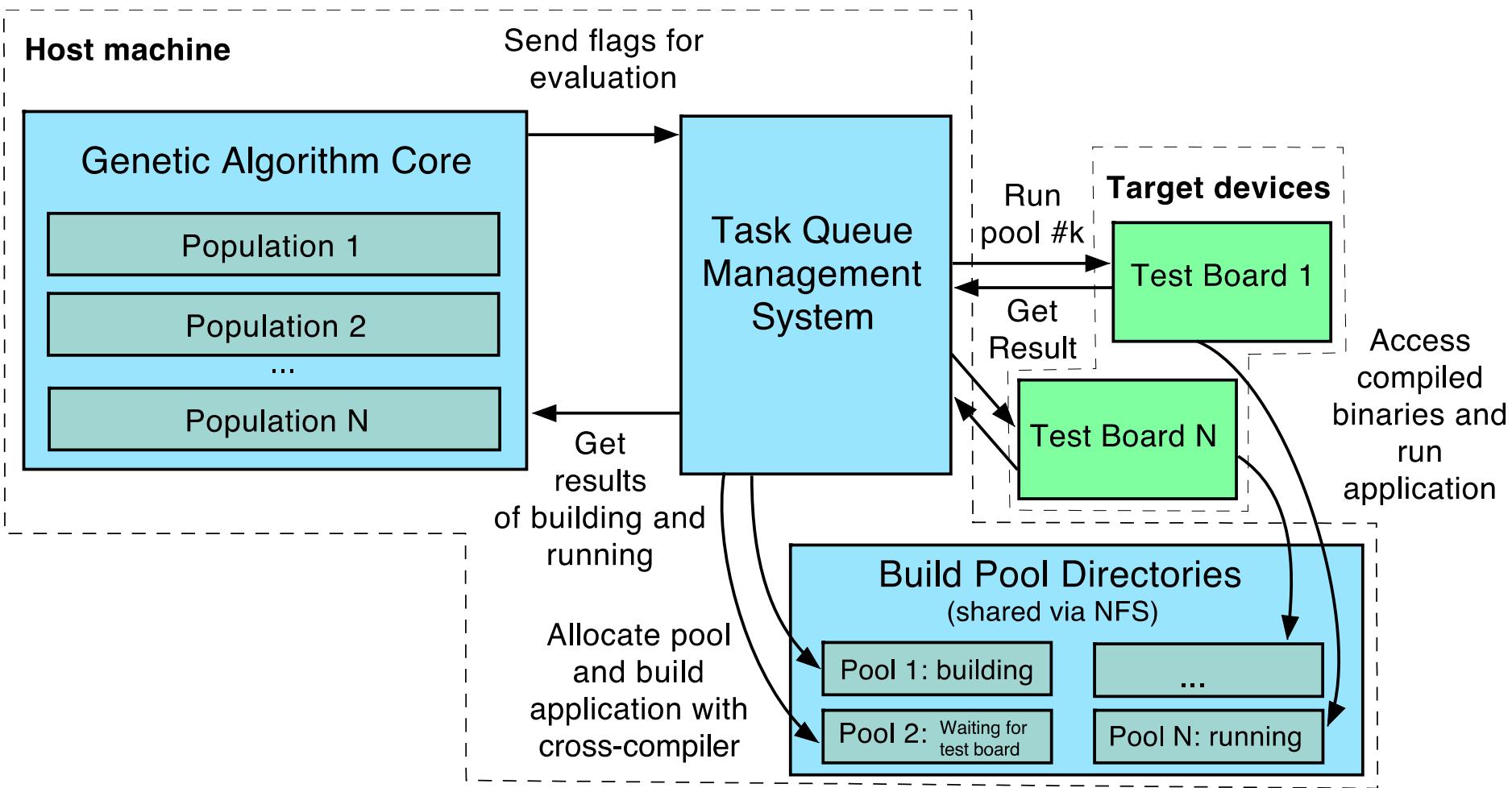
```
cmp r1, #0
itttt eq
moveq r1, #1
moveq r2, #2
moveq r3, #3
moveq r4, #4
it eq
moveq r5, #5
...
...
```

Иначе конвейер может быть заполнен кодом, который никогда не выполняется

TA^CT: Tool for Automatic Compiler Tuning

- Motivation
 - GCC is multiplatform compiler, has 150+ optimizations which are not always tuned well for the specific platform
 - Goal: improve the performance of GCC for ARM
 - Traditional approach: profile application, then find out what needs to be improved in the compiler
 - It's difficult, and takes much time and expertise
- Performance analysis involving TACT:
 - Use Genetic Algorithm to find best-performing compile options
 - Also find default -O2 optimizations that make code worse
 - Find which of the options impact performance the most
 - Compare the assembly
 - Fix relevant optimization in GCC to make what we've found the compiler's default behavior

How TACT Works



Tuning Results (raw)

```
-O2 -mcpu=cortex-a9 -mtune=cortex-a9 -mfpu=neon -mfloat-abi=softfp -fno-aggressive-loop-optimizations -fno-align-functions -fno-align-jumps -fno-align-labels -falign-loops -fno-argument-alias -fno-argument-noalias -fargument-noalias-anything -fno-argument-noalias-global -fno-associative-math -fno-asynchronous-unwind-tables -fno-auto-inc-dec -fbranch-count-reg -fbranch-probabilities -fno-branch-target-load-optimize -fno-branch-target-load-optimize2 -fno-btr-bb-exclusive -fcaller-saves -fcheck-data-deps -fno-combine-stack-adjustments -fno-common -fcompare-elim -fconserve-stack -fno-cprop-registers -fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fno-data-sections -fdce -fdefer-pop -fno-delayed-branch -fdelete-dead-exceptions -fdelete-null-pointer-checks -fdevirtualize -fno-dse -fearly-inlining -fno-expensive-optimizations -ffast-math -ffinite-math-only -ffloat-store -fno-forward-propagate -ffunction-cse -fno-function-sections -fgcse -fgcse-after-reload -fgcse-las -fgcse-lm -fgcse-sm -fguess-branch-probability -fhoist-adjacent-loads -fif-conversion -fno-if-conversion2 -fno-indirect-inlining -finline -finline-functions -finline-functions-called-once -finline-small-functions -fipa-cp -fipa-cp-clone -fno-ipa-matrix-reorg -fipa-pt -fipa-pure-const -fno-ipa-reference fipa-sra -fipa-struct-reorg -fivopts -fno-jump-tables -fkeep-inline-functions -fkeep-static-consts -floop-nest-optimize -fmerge-all-constants -fmerge-constants -fmodulo-sched -fno-modulo-sched-allow-regmoves -fmove-loop-invariants -fno-optimize-register-move -foptimize-sibling-calls -foptimize-strlen -fno-pack-struct -fno-partial-inlining -fno-peel-loops -fpeephole -fpeephole2 -fpredictive-commoning -fno-prefetch-loop-arrays -fno-reciprocal-math -free -fno-reg-struct-return -fno-regmove -frename-registers -fno-reorder-functions -fno-rerun-cse-after-loop -freschedule-modulo-scheduled-loops -frounding-math -fno-sched-critical-path-heuristic -fno-sched-dep-count-heuristic -fno-sched-group-heuristic -fno-sched-interblock -fno-sched-last-insn-heuristic -fsched-pressure -fno-sched-rank-heuristic -fno-sched-spec -fsched-spec-insn-heuristic -fno-sched-spec-load -fsched-spec-load-dangerous -fsched-stalled-insns -fno-sched-stalled-insns-dep -fsched2-use-superblocks -fsched2-use-traces -fno-schedule-insns -fschedule-insns2 -fshrink-wrap -fno-section-anchors -fno-selective-scheduling -fselective-scheduling2 -fsel-sched-pipelining -fno-sel-sched-pipelining-outer-loops -fno-sel-sched-reschedule-pipelined -fsee -fno-signed-zeros -fno-single-precision-constant -fsplit-ivs-in-unroller -fsplit-wide-types -fno-stack-check -fno-thread-jumps -ftoplevel-reorder -fno-tracer -fno-trapping-math -fno-tree-bit-cpp -fno-tree-builtin-call-dce -fno-tree-cpp -ftree-ch -ftree-coalesce-vars -ftree-copy-prop -ftree-copyrename -ftree-cselim -ftree-dce -fno-tree-dominator-opts -fno-tree-dse -ftree-forwprop -ftree-fre -ftree-loop-distribution -ftree-loop-distribute-patterns -fno-tree-loop-if-convert -ftree-loop-if-convert-stores -fno-tree-loop-im -ftree-loop-ivcanon -fno-tree-loop-optimize -ftree-lrs -ftree-partial-pre -fno-tree-phiprop -fno-tree-pre -ftree-pta -ftree-reassoc -ftree-scev-cprop -ftree-sink -fno-tree-slp-vectorize -ftree-slr -fno-tree-sra -fno-tree-switch-conversion -fno-tree-tail-merge -fno-tree-ter -fno-tree-vec-loop-version -fno-tree-vectorize -fno-tree-vrp -fno-unroll-all-loops -fno-unroll-loops -fno-unsafe-loop-optimizations -fno-unsafe-math-optimizations -fno-unswitch-loops -fno-unwind-tables -fno-variable-expansion-in-unroller -fvect-cost-model -fno-vpt -fno-web -fno-ira-loop-pressure -fira-algorithm=CB -fira-region=all -fira-share-save-slots -fno-ira-share-spill-slots --param ira-max-loops-num=1000 --param inline-unit-growth=50 --param large-function-growth=100 --param large-function-insns=4800 --param large-stack-frame=264 --param large-stack-frame-growth=900 --param large-unit-insns=3000 --param max-delay-slot-live-search=669 --param max-inline-insns-auto=220 --param max-inline-insns-recursive=950 --param max-inline-insns-single=350 --param max-inline-recursive-depth=19 --param max-inline-recursice-depth-auto=17 --param max-variable-expansions-in-unroller=3 --param max-unrolled-insns=330 --param max-average-unrolled-insns=140 --param max-unroll-times=10 --param max-peeled-insns=360 --param max-peel-times=8 --param max-completely-peeled-insns=700 --param max-completely-peel-times=16 --param max-once-peeled-insns=680 --param min-inline-recursive-probability=7 --param simultaneous-prefetches=8 --param l1-cache-line-size=0 --param prefetch-latency=800 --param l1-cache-size=24 --param l2-cache-size=160 --param min-insn-to-prefetch-ratio=10 --param prefetch-min-insn-to-mem-ratio=0
```

Filtering the Results

- If compiling with and without a compiler option produces an identical binary, we throw it out
- Typically, only **15-30%** of the original options left in the resulting set
- Example of the reduced set of options (SPEC2K's 255.vortex benchmark, 41 of 200 left):

```
-O2 -mcpu=cortex-a9 -mtune=cortex-a9 -mfpu=neon -mfloat-abi=softfp -fno-aggressive-loop-optimizations -fno-auto-inc-dec -fbranch-probabilities -fno-common -fconserve-stack -fno-cprop-registers -fno-dse -fno-expensive-optimizations -ffinite-math-only -ffloat-store -fno-forward-propagate -fgcse-after-reload -fgcse-las -fgcse-sm -fno-if-conversion2 -finline-functions -fipa-pta -fno-jump-tables -fkeep-inline-functions -fmodulo-sched -fno-partial-inlining -frename-registers -fno-reorder-functions -fno-rerun-cse-after-loop -fno-sched-interblock -fno-schedule-insns -fno-section-anchors -fselective-scheduling2 -fno-thread-jumps -fno-tree-cpp -fno-tree-dominator-opts -fno-tree-dse -fno-tree-loop-optimize -fno-tree-pre -fno-tree-sra -fno-tree-switch-conversion -fno-tree-ter -fno-tree-vrp -fira-algorithm=CB -fira-region=all -fno-ira-share-spill-slots --param ira-max-loops-num=1000 --param inline-unit-growth=50 --param large-stack-frame=264 --param large-stack-frame-growth=900 --param large-unit-insns=3000 --param max-inline-insns-auto=220 --param l1-cache-line-size=0
```

Example: Tuning Results for 255.vortex from SPEC2K INT

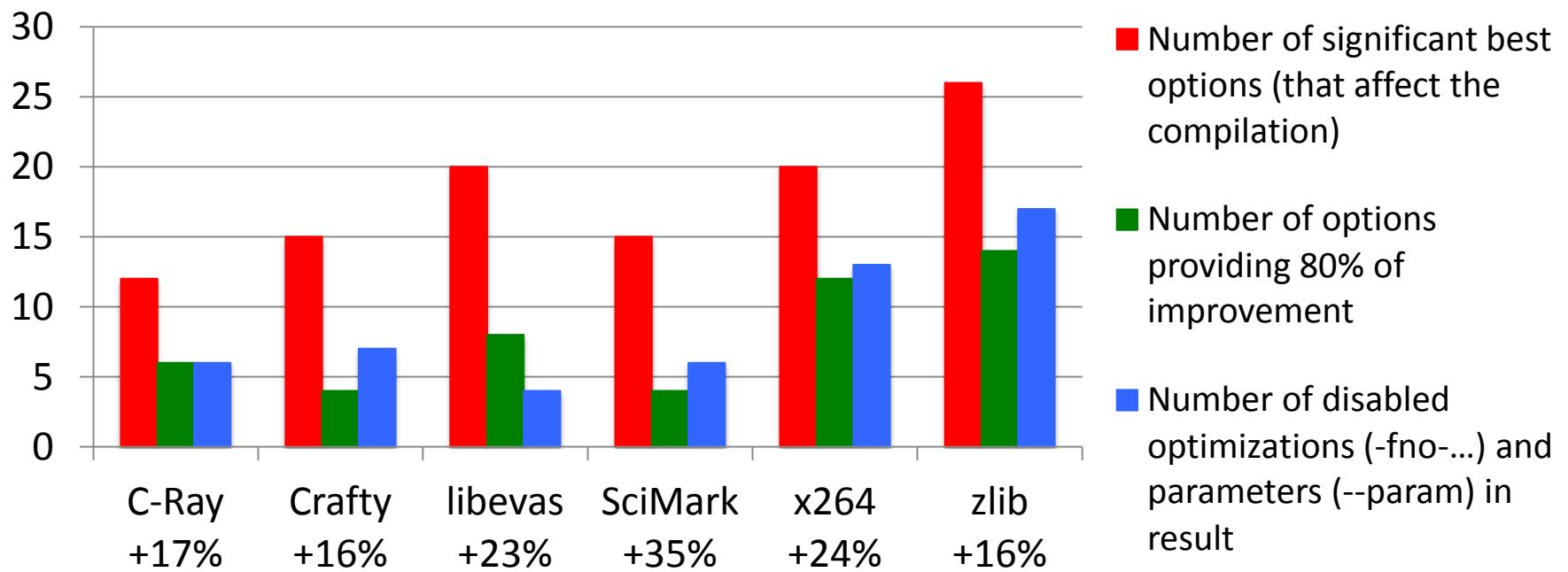
Score	%Prev	%Base	Flags diff
10.720	0.00%	0.00%	-O2 -mcpu=cortex-a9 -mtune=cortex-a9 -mfpu=neon -mfloat-abi=softfp
10.340	3.54%	3.54%	-finline-functions
9.990	3.38%	6.81%	-fno-if-conversion2
10.050	-0.60%	6.25%	-fno-sched-interblock
9.870	1.79%	7.93%	-fselective-scheduling2
9.910	-0.41%	7.56%	-fno-tree-pre
9.730	1.82%	9.24%	--param max-inline-insns-auto=220
9.620	1.13%	10.26%	-fno-tree-dominator-opts
9.530	0.94%	11.10%	-fno-schedule-insns
9.490	0.42%	11.47%	-frename-registers
9.550	-0.63%	10.91%	-fno-forward-propagate
9.360	1.99%	12.69%	-fbranch-probabilities
9.330	0.32%	12.97%	-fno-rerun-cse-after-loop
9.340	-0.11%	12.87%	-fgcse-las -fno-tree-loop-optimize
9.280	0.64%	13.43%	-fno-tree-vrp
9.340	-0.65%	12.87%	-fno-section-anchors
9.290	0.54%	13.34%	-fno-tree-ter
9.260	0.32%	13.62%	--param large-unit-insns=3000
...			

RED: default -O2 optimizations that don't work properly (or parameters that may have incorrect value) with significant impact

GREEN: candidate optimizations to enable by default for -O2 on this target

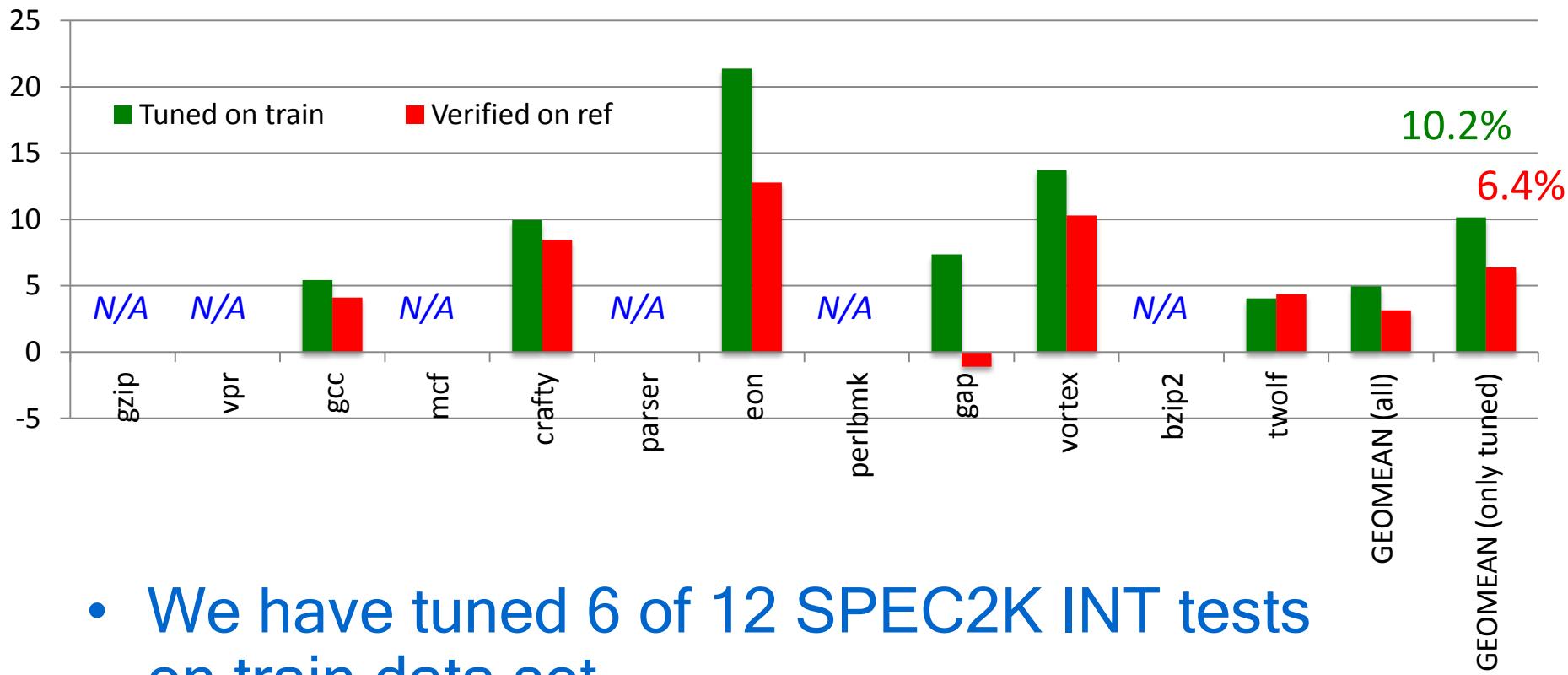
Tuning of Selected Open-Source Apps

- 200 original GCC options, 30 generations



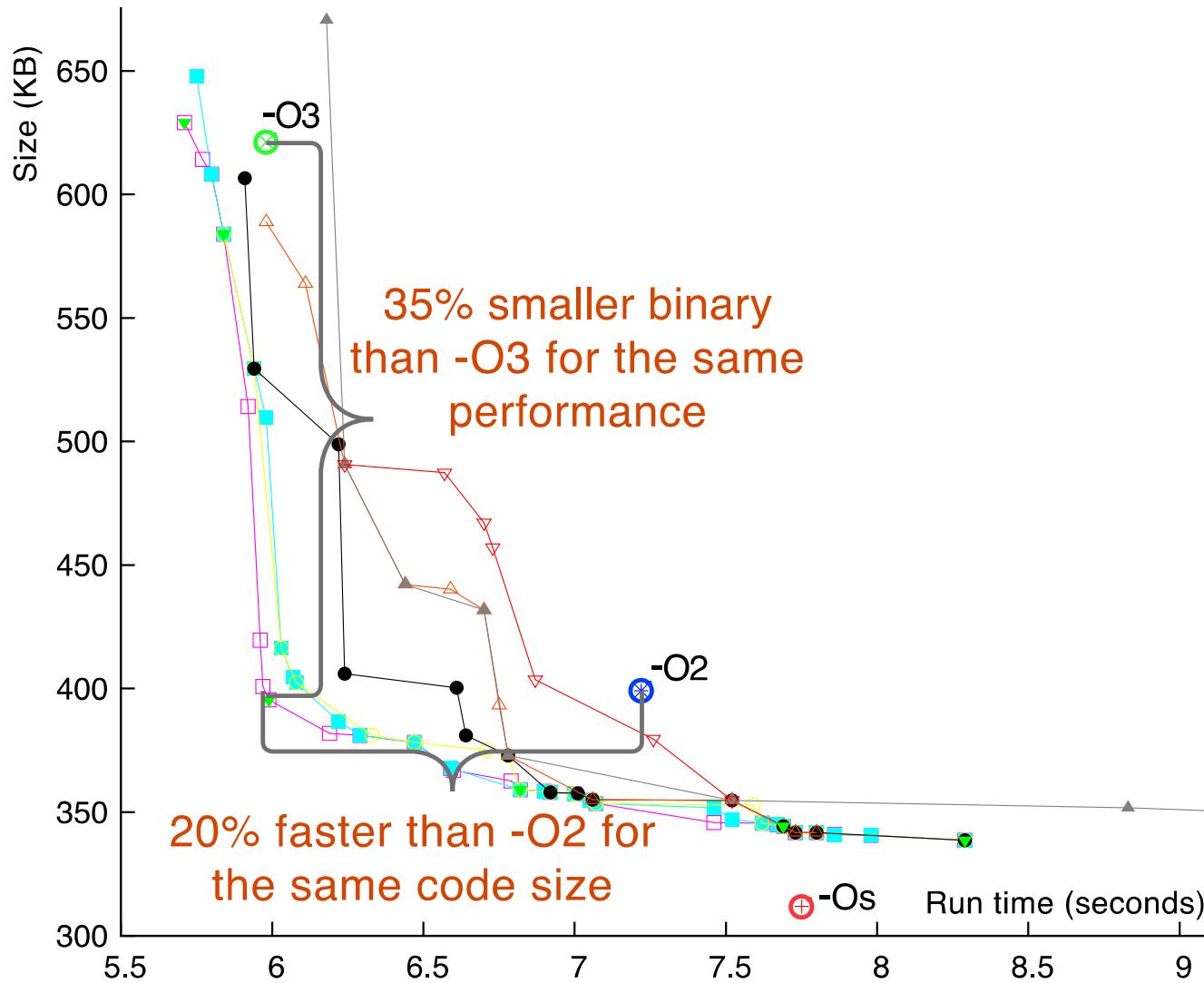
- There are few compiler options left for further manual analysis

Tuning SPEC2K INT (Thumb-2)



- We have tuned 6 of 12 SPEC2K INT tests on train data set
- The results were verified with ref dataset
- Half of the speedup persists across different data sets

Pareto Tuning for x264



Движки JavaScript (SFX, V8)

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)

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

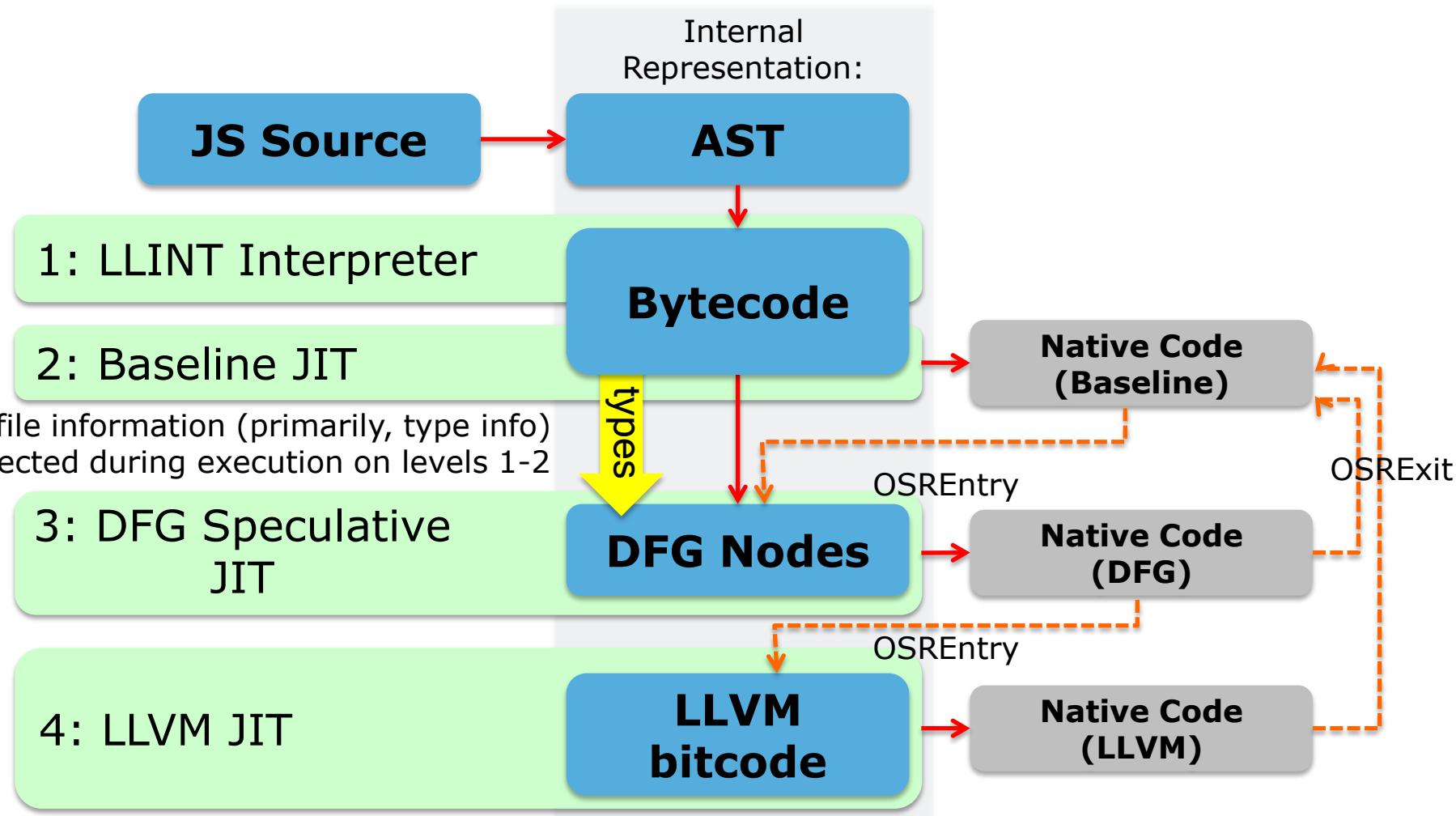
➤ Решение:

- 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

Modern JavaScript Engines

- Major Open-Source Engines:
 - **SFX (JavaScriptCore)**
 - Used in Safari and other WebKit-based browsers (Tizen, BlackBerry)
 - Part of WebKit browser engine, maintained by Apple
 - **V8**
 - Used in Google Chrome, and Android built-in browser
 - Default JS engine for Blink browser engine (initially was an option to SFX in WebKit), mainly developed by Google
 - **Mozilla SpiderMonkey**
 - JS engine in Mozilla FireFox
- SFX and V8 common features
 - Multi-level JIT, each level have different IRs and complexity of optimizations
 - Rely on profiling and speculation to do effective optimizations
 - Just about 2x slower than native code (SunSpider benchmark)

SFX Multi-Tier JIT Architecture



When the executed code becomes “hot”, SFX switches **Baseline JIT → DFG → LLVM** using *On Stack Replacement* technique

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

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

SFX Overhead Analysis

