ORACLE

Сжимай меня полностью
java -jar jar.jar

Aleksey Shipilëv
aleksey.shipilev@oracle.com, @shipilev

Об ожиданиях

Доклад помогает понять:

1. Какой ад и вакханалия творится в кишках VM, JDK, хардвара
2. Как низкоуровневое знание рождает высокоуровневое понимание
3. Что благими намерениями вымощена дорога в перформанс ад
4. Как клёво и спокойно жить в кровавом энторпрайзе

Доклад будет:

1. Быстрым, сложным, беспощадным
2. Не прощающим невнимания и отвлечений
3. Заставляющим думать по ходу повествования

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Intro


Укладка полей

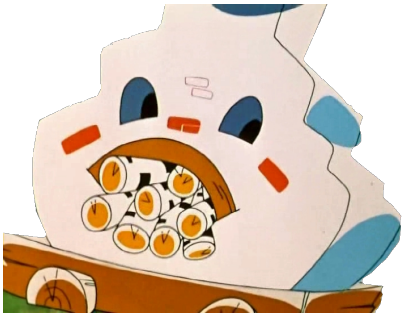
Укладка полей: пререквизиты

```
public class C {
  boolean b;
  char    c;
  long    l;
}
```

Как уложить в памяти?
Положим все поля друг за дружкой,
и всё будет OK!

Укладка полей: почему не работает

Выравнивание не забыл, милок?



```
// [head] is aligned at 8
public class C {
  boolean b; // [head + 0]
  char    c; // [head + 1]
  long    i; // [head + 3]
}
```

Укладка полей: бдыдыщь

```
private int doCasWith(long addr) {
  int v;
  do {
    v = U.getIntVolatile(null, addr);
  } while(!U.compareAndSwapInt(null, addr, v, v + 1));
  return v;
}
```

```
# A fatal error has been detected by the JRE
#  SIGBUS (0x7) at pc=0xb3f00630, pid=32042, tid=2443179104
# Problematic frame:
# v  ~StubRoutines::atomic_cmpxchg
```

Укладка полей: да ну ладно

Но это на ARM'е, и если атомарные инструкции...

- Да на простых чтениях и записях всё равно!
- На x86-то всё точно окей!
- Да вообще, что это за пудрёж мозгов!
- Хардвар приде, порядок наведе!

Укладка полей: x86, read

```
@CompilerControl(CompilerControl.Mode.DONT_INLINE)
private int doReadWith(long addr) {
  return U.getInt(addr);
}
```

x86_64, i7-4790K (Haswell, 2014) @ 4.0 GHz, 8u40, Linux x86_64:

Benchmark	crossCL[1]	size	Score, us/op	
aligned	false	16M	39468.7	± 79.1
aligned	true	16M	39427.0	± 219.6

[1] crosses cache line

Укладка полей: x86, read

```
@CompilerControl(CompilerControl.Mode.DONT_INLINE)
private int doReadWith(long addr) {
  return U.getInt(addr);
}
```

x86_64, i7-4790K (Haswell, 2014) @ 4.0 GHz, 8u40, Linux x86_64:

Benchmark	crossCL[1]	size	Score, us/op	
aligned	false	16M	39468.7	± 79.1
aligned	true	16M	39427.0	± 219.6
misaligned	false	16M	39374.4	± 53.7
misaligned	true	16M	60347.1	± 59.3

[1] crosses cache line

Укладка полей: x86, read

Возьмём в руки -prof perfnorm:

Benchmark	crossCL	size	Score, #/op	
misaligned	false	16M	39563	± 324
misaligned:·CPI	false	16M	0.448	± 0.012
misaligned:·L1-dcache-load-misses	false	16M	15558141	± 413808
misaligned:·cycles	false	16M	162977257	± 5936931
misaligned:·instructions	false	16M	363813283	± 5040277
misaligned	true	16M	59599	± 268
misaligned:·CPI	true	16M	0.675	± 0.019
misaligned:·L1-dcache-load-misses	true	16M	30934993	± 308608
misaligned:·cycles	true	16M	245238217	± 10922256
misaligned:·instructions	true	16M	363323233	± 6487361

Укладка полей: x86, read

Возьмём в руки -prof perfnorm:

Benchmark	crossCL	size	Score, #/op	
misaligned	false	16M	39563	± 324
misaligned:·CPI	false	16M	0.448	± 0.012
misaligned:·L1-dcache-load-misses	false	16M	15558141	± 413808
misaligned:·cycles	false	16M	162977257	± 5936931
misaligned:·instructions	false	16M	363813283	± 5040277
misaligned	true	16M	59599	± 268
misaligned:·CPI	true	16M	0.675	± 0.019
misaligned:·L1-dcache-load-misses	true	16M	30934993	± 308608
misaligned:·cycles	true	16M	245238217	± 10922256
misaligned:·instructions	true	16M	363323233	± 6487361

Невыровненное чтение хватает два кешлайна сразу, упс.

Укладка полей: x86, compare-and-set

```
@CompilerControl(CompilerControl.Mode.DONT_INLINE)
private int doCasWith(long addr) {
  return U.getAndAddInt(null, addr, 1);
}
```

x86_64, i7-4790K (Haswell, 2014) @ 4.0 GHz, 8u40, Linux x86_64:

Benchmark	crossCL	size	Score, us/op
aligned	false	4096	27.9 ± 0.1
aligned	true	4096	27.8 ± 0.1

Укладка полей: x86, compare-and-set

```
@CompilerControl(CompilerControl.Mode.DONT_INLINE)
private int doCasWith(long addr) {
  return U.getAndAddInt(null, addr, 1);
}
```

x86_64, i7-4790K (Haswell, 2014) @ 4.0 GHz, 8u40, Linux x86_64:

Benchmark	crossCL	size	Score, us/op
aligned	false	4096	27.9 ± 0.1
aligned	true	4096	27.8 ± 0.1
misaligned	false	4096	

Укладка полей: x86, compare-and-set

```
@CompilerControl(CompilerControl.Mode.DONT_INLINE)
private int doCasWith(long addr) {
  return U.getAndAddInt(null, addr, 1);
}
```

x86_64, i7-4790K (Haswell, 2014) @ 4.0 GHz, 8u40, Linux x86_64:

Benchmark	crossCL	size	Score, us/op
aligned	false	4096	27.9 ± 0.1
aligned	true	4096	27.8 ± 0.1
misaligned	false	4096	28.0 ± 0.2
misaligned	true	4096	

Укладка полей: x86, compare-and-set

```
@CompilerControl(CompilerControl.Mode.DONT_INLINE)
private int doCasWith(long addr) {
  return U.getAndAddInt(null, addr, 1);
}
```

x86_64, i7-4790K (Haswell, 2014) @ 4.0 GHz, 8u40, Linux x86_64:

Benchmark	crossCL	size	Score, us/op	
aligned	false	4096	27.9	± 0.1
aligned	true	4096	27.8	± 0.1
misaligned	false	4096	28.0	± 0.2
misaligned	true	4096	3004.2	± 21.4

Укладка полей: x86, CAS

Возьмём в руки -prof perfnorm:

Benchmark	crossCL	size	Score, us/op	
misaligned	false	4096	27.9	± 0.2
misaligned:·CPI	false	4096	1.1	± 0.1
misaligned:·bus-cycles	false	4096	2859.3	± 46.6
misaligned:·cycles	false	4096	114577.6	± 2508.3
misaligned:·instructions	false	4096	103208.6	± 480.0
misaligned	true	4096	2949.3	± 22.8
misaligned:·CPI	true	4096	95.1	± 67.3
misaligned:·bus-cycles	true	4096	303256.0	± 14919.7
misaligned:·cycles	true	4096	12119340.7	± 359775.6
misaligned:·instructions	true	4096	131116.5	± 96913.2

Атомарная операция через кешлайн – это БОЛЬ.

Укладка полей: сжимай меня полностью

```
org.openjdk.AligmentSample.C object internals:
 OFFSET  SIZE     TYPE DESCRIPTION                               VALUE
      0    12          (object header)                           N/A
     12     2     char C.c                                       N/A
     14     1  boolean C.b                                       N/A
     15     1          (alignment/padding gap)                   N/A
     16     8     long C.l                                       N/A
Instance size: 24 bytes
```

Выравниваем объект на 8. Укладываем поля по группами по размеру, подтыкаем дырки. Всё разложили по выравниваниям, да ещё и место под один `boolean/byte` осталось!

Java


Сжатые ссылки

Сжатые ссылки: загадка

```
@Param({"1024", "1048576"})
int size;

Integer[] o = ...; // setup int[size] and populate

@Benchmark
public void test() {
  int s = 0;
  for (Integer i : o)
     s += i;
  return s;
}
```

Зависит ли производительность от количества памяти в JVM?

Сжатые ссылки: битов на тип

Тип	Модель			
	32 бит		64 бит	
	Мин.	Факт.	Мин.	Факт.
boolean	1	?	1	?
byte	8	?	8	?
short	16	?	16	?
char	16	?	16	?
int	32	?	32	?
float	32	?	32	?
long	64	?	64	?
double	64	?	64	?

Сжатые ссылки: битов на тип

Тип	Модель			
	32 бит		64 бит	
	Мин.	Факт.	Мин.	Факт.
boolean (удобно)	1	8	1	8
byte (удобно)	8	8	8	8
short (удобно)	16	16	16	16
char (удобно)	16	16	16	16
int (удобно)	32	32	32	32
float (удобно)	32	32	32	32
long (удобно)	64	64	64	64
double (удобно)	64	64	64	64

Сжатые ссылки: битов на тип

Тип	Модель			
	32 бит		64 бит	
	Мин.	Факт.	Мин.	Факт.
boolean (удобно)	1	8	1	8
byte (удобно)	8	8	8	8
short (удобно)	16	16	16	16
char (удобно)	16	16	16	16
int (удобно)	32	32	32	32
float (удобно)	32	32	32	32
long (удобно)	64	64	64	64
double (удобно)	64	64	64	64
reference	?	?	?	?

Сжатые ссылки: битов на тип

Тип	Модель			
	32 бит		64 бит	
	Мин.	Факт.	Мин.	Факт.
boolean (удобно)	1	8	1	8
byte (удобно)	8	8	8	8
short (удобно)	16	16	16	16
char (удобно)	16	16	16	16
int (удобно)	32	32	32	32
float (удобно)	32	32	32	32
long (удобно)	64	64	64	64
double (удобно)	64	64	64	64
reference (удобно)		32		64

Сжатые ссылки: проблема

«Лезут все в эти кэши, будто они резиновые»

О чём надо беспокоиться:

- Кеши: сэкономишь на размерах полей, сэкономишь на занимаемой памяти, лучше ляжешь в кеши, лучше будет всем
- Доступ: если требуется нетривиальный набор команд, чтобы добраться до поля, то это хуже, чем «плохо» лежать в кеше

Идеально:

```
mov [refField], %reg1
mov [%reg1 + $field], %reg2
```


Сжатые ссылки: трюк №1

Но если все объекты помещаются в кучу размером < 4 Гб...

Java

Сжатые ссылки: трюк №1

Но если все объекты помещаются в кучу размером < 4 Гб...
Можно взять младшие 32 бита адреса!

```
mov [refField], %reg1  // extend 32 -> 64
mov [%reg1 + $field], %reg2
```

Java

Сжатые ссылки: трюк №2

Объекты выравнены на 8 байт!
Сколько младших битов в адресе всегда равны нулю?

Сжатые ссылки: трюк №2

Объекты выравнены на 8 байт!
Сколько младших битов в адресе всегда равны нулю?

Сжатые ссылки: трюк №2

Объекты выравнены на 8 байт!
Сколько младших битов в адресе всегда равны нулю?
Можно использовать 32-битные ссылки и в куче <32 Гб, если сдвинуть!

```
mov [refField], %reg1 // extend 32 -> 64
shl %reg1, 3
mov [%reg1 + $field], %reg2
```

Не хватит «3», то можно
попросить выравнивание объектов побольше.

Сжатые ссылки: трюк №3

Но если куча размером 4 Гб, но виртуальные адреса не начинаются с нуля...

Сжатые ссылки: трюк №3

Но если куча размером 4 Гб, но виртуальные адреса не начинаются с нуля...
Можно прибавить волшебный адрес начала кучи!

```
mov [refField], %reg1 // extend 32 -> 64
shl %reg1, 3
add $base, %reg1
mov [%reg1 + $field], %reg2
```


Сжатые ссылки: простенький бенчмарк

```
@Param({"1024", "1048576"})
int size;

Integer[] o = ...; // setup int[size] and populate

@Benchmark
public void test() {
  int s = 0;
  for (Integer i : o) {
     s += i;
  }
  return s;
}
```


Сжатые ссылки: посмотрим на результаты

Когда доминируют cache miss'ы, сжатые ссылки дают дикие приросты:

Mode	Size	Score, us/op	
-Xmx1g	1M	1069.6	± 4.9
-Xmx8g	1M	1074.0	± 7.9
-Xmx40g	1M	1725.6	± 5.8
-Xmx40g -XX:ObjAlign=16	1M	1089.4	± 35.7

- На кучах до 32 Гб включились сжатые ссылки
- На куче больше 32 Гб сжатые ссылки выключились
- Добавление `-XX:ObjectAlignmentInBytes=16` включило сжатые ссылки обратно

Сжатые ссылки: посмотрим на результаты

Когда всё в кешах, режимы кодирования имеют значение:

Mode	Size	Score, us/op	
-Xmx1g	1024	0.333	± 0.001
-Xmx8g	1024	0.370	± 0.006
-Xmx40g	1024	0.362	± 0.003

- На куче $<$ 4Гб включились 32-битные ссылки, без всяких сдвигов
- На куче $>$ 32Гб хоть ссылки и 64-битные, всё в кешах и так

Сжатые ссылки: посмотрим на результаты

Когда всё в кешах, режимы кодирования имеют значение:

Mode	Size	Score, us/op	
-Xmx1g	1024	0.333	± 0.001
-Xmx8g	1024	0.370	± 0.006
-Xmx40g	1024	0.362	± 0.003

- На куче < 4Гб включились 32-битные ссылки, без всяких сдвигов
- На куче > 32Гб хоть ссылки и 64-битные, всё в кешах и так

Т.е. *зажимание хипа* может дать
БОЛЬШЕ ПЕРФОРМАНСА
ДЛЯ БОГОВ ПЕРФОРМАНСА


Remembered Sets

Remembered Sets: загадка

```
Object src = new Object(); long longSrc = 42;

Object   sink1, sink2;
long longSink1, longSink2;

@Benchmark @Group("ref") void r1() { sink1 = src; }
@Benchmark @Group("ref") void r2() { sink2 = src; }

@Benchmark @Group("long") void l1() { longSink1 = longSrc; }
@Benchmark @Group("long") void l2() { longSink2 = longSrc; }
```

Кто быстрее: `ref` или `long` – и почему?

Remembered Sets: загадка

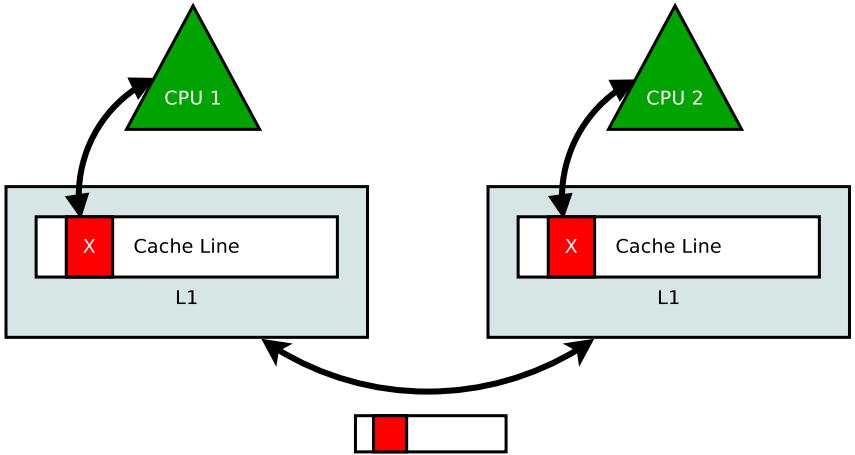
Benchmark	Score, ns/op	
long	2.479	± 0.461
long:·CPI	1.347	± 0.514
long:·L1-dcache-load-misses	0.067	± 0.068
long:·L1-dcache-loads	3.096	± 0.138
ref	19.462	± 10.797
ref:·CPI	4.445	± 19.590
ref:·L1-dcache-load-misses	0.553	± 3.068
ref:·L1-dcache-loads	3.022	± 0.246

Remembered Sets: загадка

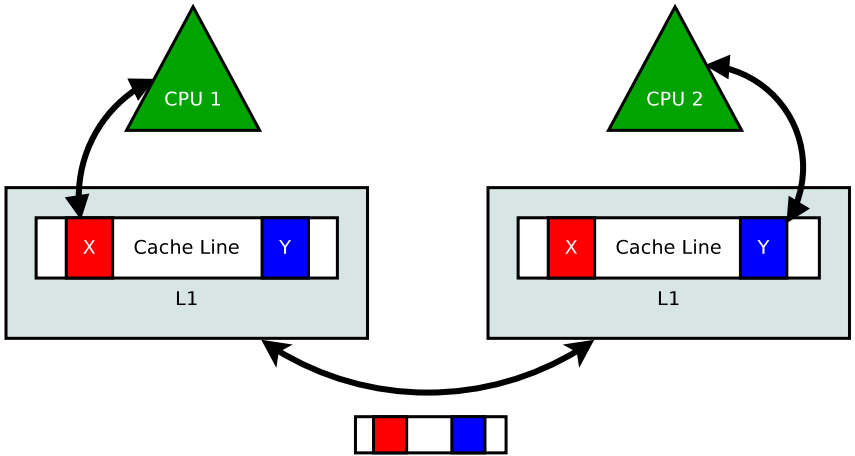
Benchmark	Score, ns/op	
long	2.479	± 0.461
long:·CPI	1.347	± 0.514
long:·L1-dcache-load-misses	0.067	± 0.068
long:·L1-dcache-loads	3.096	± 0.138
ref	19.462	± 10.797
ref:·CPI	4.445	± 19.590
ref:·L1-dcache-load-misses	0.553	± 3.068
ref:·L1-dcache-loads	3.022	± 0.246

2% и 20% (!) L1-кэш-миссов

Remembered Sets: True Sharing



Remembered Sets: False Sharing

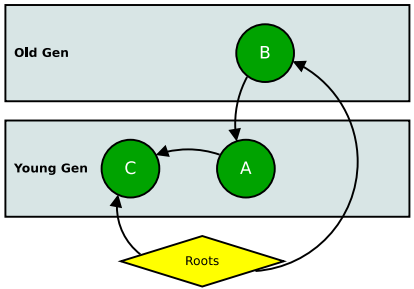


Remembered Sets: добавим на поля `@Contended`

Benchmark	Score, ns/op	
long	0.454	± 0.114
long:·CPI	0.241	± 0.072
long:·L1-dcache-load-misses	$\approx 10^{-6}$	
long:·L1-dcache-loads	3.074	± 0.115
ref	3.799	± 1.528
ref:·CPI	1.262	± 2.652
ref:·L1-dcache-load-misses	0.074	± 0.259
ref:·L1-dcache-loads	3.085	± 0.197

`long` сильно полегчало;
`ref` тоже сильно полегчало, но всё равно проигрывает 8x.
Откуда ещё кэш-миссы?

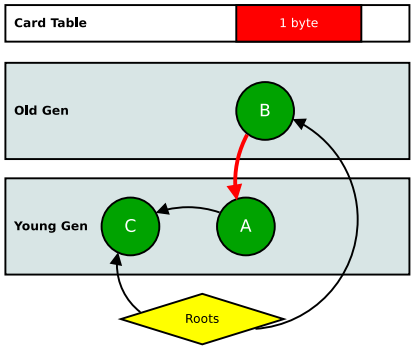
Remembered Sets: копаем



Самоподтверждающееся пророчество, GC style:

- Молодая сборка посчитает A мусором и переиспользует его место
- После сборки пойдём по ссылке B → A... и там **действительно** мусор
- Mission Accomplished!

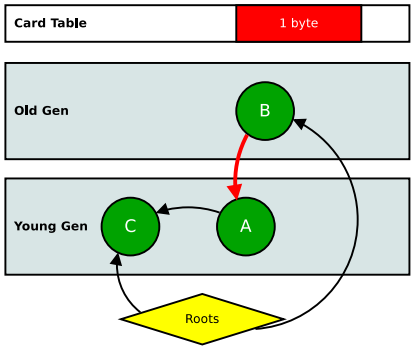
Remembered Sets: Card Tables



Нужно отслеживать ссылки между поколениями:

- При записи ссылки на A в поле объекта B нужно записать, что при молодой сборке надо посетить и B
- Записывать конкретные ссылки – замучаешься!
- Имеет смысл пометить часть **поколения** как интересную для сборки, и просматривать только её

Remembered Sets: Write Barriers

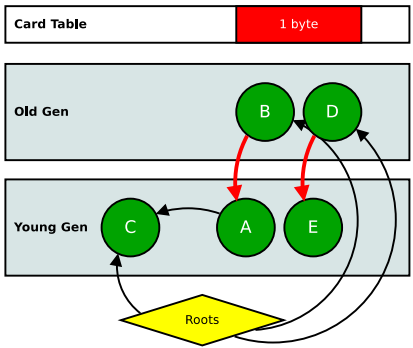



```
; put field at offset obj+0x2c
mov     %r9d,0x2c(%rcx)

; calculate the card address
mov     %rcx,%r10
shr     $0x9,%r10

; put 0 to cardTable[obj >> 9]
mov     %r12b,(%r8,%r10,1)
```

Remembered Sets: жирные такие кард-марки



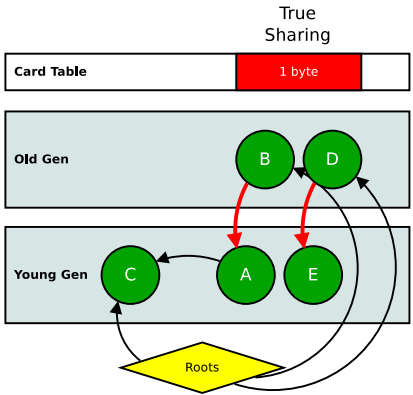
...по 512 байт хипа на 1 байт на карту:

```
; put field at offset obj+0x2c
mov     %r9d,0x2c(%rcx)

; calculate the card address
mov     %rcx,%r10
shr     $0x9,%r10

; put 0 to cardTable[obj >> 9]
mov     %r12b,(%r8,%r10,1)
```


Remembered Sets: ну и понеслось...



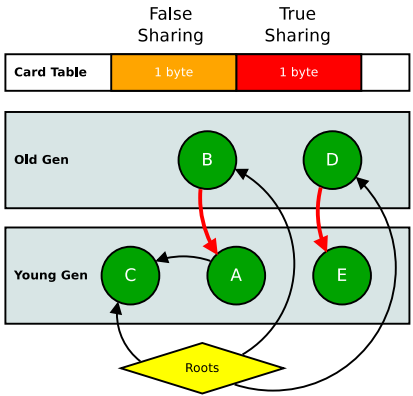
Жирненькие такие кард-марки, по 512 байт хипа на 1 байт на карту:

```
; put field at offset obj+0x2c
mov     %r9d,0x2c(%rcx)

; calculate the card address
mov     %rcx,%r10
shr     $0x9,%r10

; put 0 to cardTable[obj >> 9]
mov     %r12b,(%r8,%r10,1)
```


Remembered Sets: понеслось...



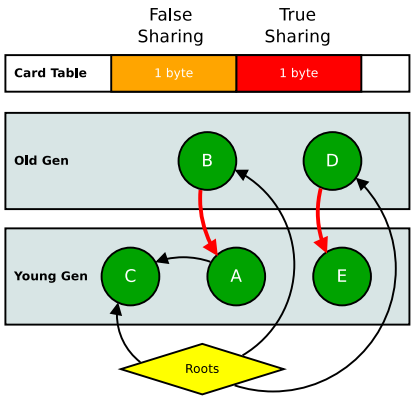
Жирненькие такие кард-марки, по 512 байт хипа на 1 байт на карту:

```
; put field at offset obj+0x2c
mov     %r9d,0x2c(%rcx)

; calculate the card address
mov     %rcx,%r10
shr     $0x9,%r10

; put 0 to cardTable[obj >> 9]
mov     %r12b,(%r8,%r10,1)
```


Remembered Sets: -XX:+UseCondCardMark




```
; put field at offset obj+0x2c
mov     %r10d,0x2c(%r9)

; calculate the card address
mov     %r9,%r10
shr     $0x9,%r10
mov     $0x7f71e9c9d000,%r11
add     %r10,%r11

; read-test card
movsbl  (%r11),%r10d
test    %r10d,%r10d
jne     GO_DIRTY
```

Remembered Sets: `-XX:+UseCondCardMark`

Benchmark	Score, ns/op	
long	0.466	± 0.109
long:·CPI	0.248	± 0.168
long:·L1-dcache-load-misses	$\approx 10^{-6}$	
long:·L1-dcache-loads	3.085	± 0.068
ref	1.027	± 0.260
ref:·CPI	0.259	± 0.123
ref:·L1-dcache-load-misses	$\approx 10^{-6}$	
ref:·L1-dcache-loads	4.085	± 0.117

- И никаких кэш-миссов! (их порядка 10^{-6} на операцию)
- CPI близок к идеальному: четыре инструкции за цикл

Remembered Sets: -XX:+UseG1GC

Benchmark	Score, ns/op	
long	0.392	± 0.044
long:·CPI	0.220	± 0.024
long:·L1-dcache-load-misses	$\approx 10^{-6}$	
long:·L1-dcache-loads	3.085	± 0.043
ref	1.108	± 0.344
ref:·CPI	0.245	± 0.271
ref:·L1-dcache-load-misses	$\approx 10^{-6}$	
ref:·L1-dcache-loads	4.105	± 0.203

- G1 барьеры сложнее, но зато и не подвержены таким проблемам
- CPI близок к идеальному: четыре инструкции за цикл

Remembered Sets: сжимай меня, если сможешь

Рантаймы будут вмешиваться в работу,
чтобы обеспечить корректность программы

- Даже если вы не мусорите и всё «предусмотрели»
- Выбор GC – это не только выбор политики сборки, но и политики аллокации, и рантайм-оверхеда
- Часто эти оверхеды нетривиальны, потому что VM пытается всячески сэкономить

Java


Скрытый код

Скрытый код: фокус

```
class MyObject {
  int x;
}

int doMe(MyObject o) {
  return o.x;
}
```


Скрытый код: фокус

```
class MyObject {
  int x;
}

int doMe(MyObject o) {
  return o.x;
}
```

```
mov     %eax,-0x14000(%rsp)
push    %rbp
sub     $0x30,%rsp
mov 0xc(%rdx),%eax
add     $0x30,%rsp
pop     %rbp
test    %eax,0x18c86a66(%rip)
retq
```

Скрытый код: фокус

```
class MyObject {
  int x;
}

int doMe(MyObject o) {
  return o.x;
}
```

```
mov     %eax,-0x14000(%rsp)
push    %rbp
sub     $0x30,%rsp
mov 0xc(%rdx),%eax
add     $0x30,%rsp
pop     %rbp
test    %eax,0x18c86a66(%rip)
retq
```

Не-не-не-не! Куда ты скукожил null check?
Быстро раскукожь его обратно!

Скрытый код: фокус

```
class MyObject {
  int x;
}

int doMe(MyObject o) {
  if (o == null) {
    throwNPE("BPMH!");
  }
  return o.x;
}
```

```
mov     %eax,-0x14000(%rsp)
push    %rbp
sub     $0x30,%rsp
mov 0xc(%rdx),%eax
add     $0x30,%rsp
pop     %rbp
test    %eax,0x18c86a66(%rip)
retq
```

В рот мне ноги! Дооптимизировались наотличьненько!

Скрытый код: покус

```
 ;*getfield x
 ; implicit exception: dispatches to HANDLER
 mov    0xc(%rdx),%eax

HANDLER:
 ;*getfield x
 ;   {runtime_call}
 callq  0x00007fbcc9105740
```

- При нулевом %rdx хардвар кидает Segmentation Fault (SEGV)
- VM-ный обработчик хватает SEGV и передаёт управление в нужное место нашего кода

Скрытый код: неудобный вопрос №1

Что если у меня есть нативный код,
который переопределит SEGV handler?

- SEGV рождается, но пролетает мимо VM-ного обработчика, прилетает в ОС ⇒ кровь, кишки, ад и родина Баруха (Java process crash)
- Крэш-дамп будет показывать на место в *сгенерированном коде*, т.е. выглядеть, как ошибка в компиляторе!
- JDK Docs: «Signal Chaining»: `https://docs.oracle.com/javase/8/docs/technotes/guides/vm/signal-chaining.html`

Скрытый код: неудобный вопрос №2

Стоимость любой «выстрелившей» проверки на null равна стоимости кидания сигнала и его обработки?

- Это сумасшедшая цена: тысячи циклов
- Много trap-ов $\Rightarrow$ «trap storm»

Скрытый код: неудобный вопрос №2

Стоимость любой «выстрелившей» проверки на null равна стоимости кидания сигнала и его обработки?

- Это сумасшедшая цена: тысячи циклов
- Много trap-ов $\Rightarrow$ «trap storm»
- Адаптивные рантаймы могут позволить себе *деоптимизировать* и скомпилировать код *в новых условиях*, если обнаруживают частые trap-ы в конкретном месте

Скрытый код: побенчмаркаемЪ

```
Object[] objs;

@Benchmark
public long test() {
  long s = 0;
  for (MyObj o : objs) {
    if (o == null)
      throw NPE;
    s += o.x;
  }
  return s;
}
```

Все объекты в objs не нулевые:

```
LOOP:
 mov     0x10(%r10,%r11,4),%ecx
 movslq  0xc(%r12,%rcx,8),%r9
 add     %r9,%rdx
 inc     %r11d
 cmp     %r8d,%r11d
 jl      LOOP
```


Скрытый код: побенчмаркаемЪ

```
Object[] objs;

@Benchmark
public long test() {
  long s = 0;
  for (MyObj o : objs) {
    if (o == null)
      throw NPE;
    s += o.x;
  }
  return s;
}
```

«Отравим», вызвав `test()` несколько раз с массивом `null`-ов, а потом снова бенчмаркаем с ненулевыми `objs`:

```
LOOP:
  mov     0x10(%r11,%r10,4),%r9d
  test    %r9d,%r9d
  je      THROWING
  movslq  0xc(%r12,%r9,8),%r8
  add     %r8,%rdx
  inc     %r10d
  cmp     %ebp,%r10d
  jl      LOOP
```


Скрытый код: perfnorm с нами согласен

Benchmark	count	poison	Score		
test	10000	false	1.096	± 0.170	ns/op
test:·branch-misses	10000	false	$\approx 10^{-4}$		#/op
test:·branches	10000	false	1.081	± 0.211	#/op
test:·cycles	10000	false	1.988	± 1.711	#/op
test:·instructions	10000	false	6.488	± 1.554	#/op
test	10000	true	1.251	± 0.295	ns/op
test:·branch-misses	10000	true	$\approx 10^{-4}$		#/op
test:·branches	10000	true	2.204	± 0.627	#/op
test:·cycles	10000	true	2.272	± 6.069	#/op
test:·instructions	10000	true	8.821	± 2.676	#/op

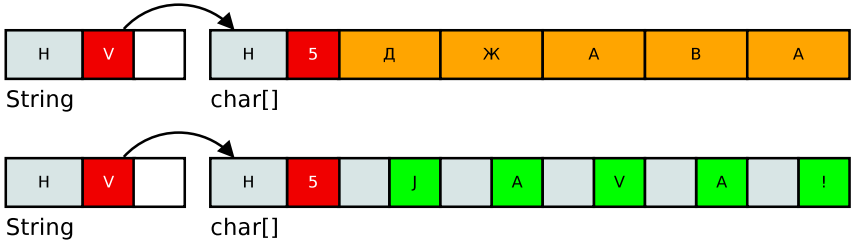
Скрытый код: сжимай меня, если сможешь

Рантаймы будут пытаться сэкономить
на сгенерированом коде, где смогут

- Часто будут «халявить» и не компилировать прямо всё
- Код и результаты в (плохих) бенчмарках будет выглядеть совсем не так, как в реальном проде
- Если не получается воспроизвести условия прода, важно делать комбинаторные эксперименты!

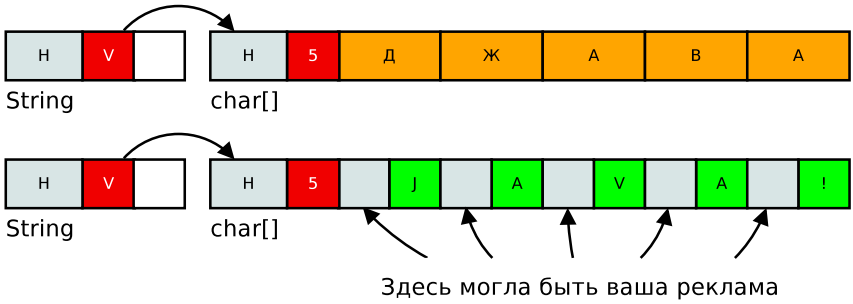

Сжатые стринги

Сжатые стринги: как устроен String



- Два объекта: `String` и `char[]`
- Нигде ничего лишнего нет?

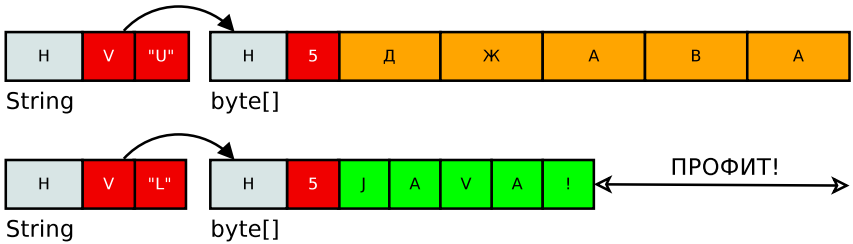
Сжатые стринги: как устроен String



- Большая часть байтов в `char[]` – нули
- Потому что большая часть стрингов укладывается в Latin1

Сжатые стринги: было круто, если...

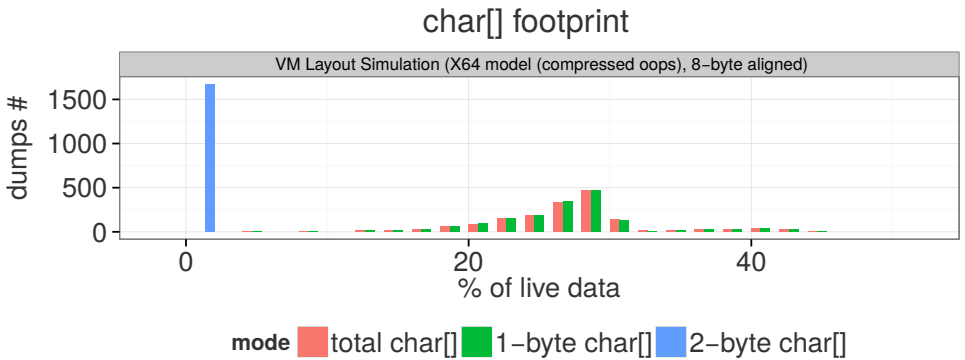
...String был устроен вот так:



- Храним не `char[]`, а плотный `byte[]`
- Сам `String` знает, какой «кодер» использовать для `byte[]`

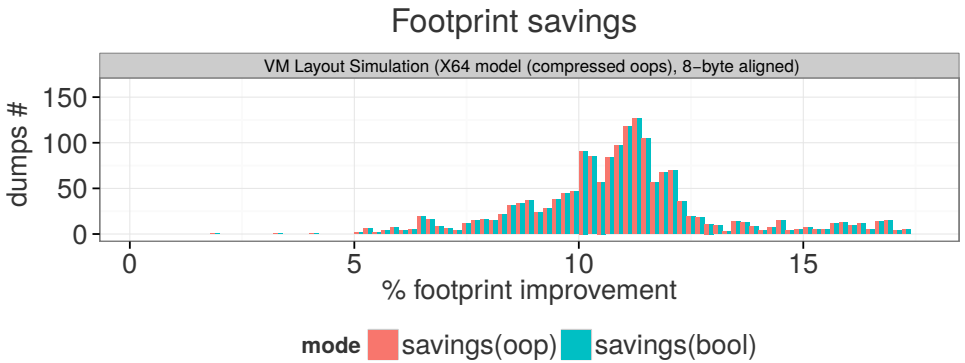
Сжатые стринги: пререквизиты

В обычных приложениях куча строчек:



Сжатые стринги: пререквизиты

В обычных приложениях большие приросты:
(и это только live data set!)



Сжатые стринги: цели

Главная цель – footprint – ужать больше данных в одной VM.
Вторичная цель – не сломать производительность.

1. Размер Latin1 `String`-ов должен снизиться
2. Размер UTF16 `String`-ов может чуть регрессировать
3. Операции над Latin1 `String`-ами не должны регрессировать
4. Операции над UTF16 `String`-ами могут чуть регрессировать

Сжатые стринги: побенчмаркаемЪ

```
String cmp1... = ...; // 1-byte char String
String cmp2... = ...; // 2-byte char String

@Benchmark
boolean cmpX_cmpY() {
  return cmpX_1.equals(cmpY_2);
}
```


Сжатые стринги: побенчмаркаемЪ

Benchmark	Size	JDK 9		JDK 9 Density	
cmp1_cmp1	1	4.8	± 0.1	5.7	± 0.1
cmp1_cmp2	1	4.8	± 0.1	3.3	± 0.1
cmp2_cmp1	1	4.9	± 0.1	3.3	± 0.1
cmp2_cmp2	1	4.7	± 0.1	5.3	± 0.1
cmp1_cmp1	4096	185.9	± 1.3	106.8	± 3.4
cmp1_cmp2	4096	186.2	± 4.6	3.3	± 0.1
cmp2_cmp1	4096	188.3	± 3.4	3.3	± 0.1
cmp2_cmp2	4096	183.1	± 3.8	186.4	± 9.1

(нс/оп, меньше – лучше)

Java

Сжатые стринги: побенчмаркаем ещё

```
int x = 42; String msg = "Msg";

@Benchmark String cyrillic()
 { return "[" + x + "] Ой, всё пропало, шеф: " + msg; }

@Benchmark String latin()
 { return "[" + x + "] Everything's FUBARed: " + msg; }
```

Сжатые стринги: побенчмаркаем ещё

```
int x = 42; String msg = "Msg";

@Benchmark String cyrillic()
 { return "[" + x + "] Ой, всё пропало, шеф: " + msg; }

@Benchmark String latin()
 { return "[" + x + "] Everything's FUBARed: " + msg; }
```

Benchmark	JDK 9		JDK 9 Density		Units
cyrillic	83.7	± 0.8	82.5	± 1.8	ns/op
cyrillic:·gc.churn.norm	103.1	± 1.2	102.2	± 1.1	B/op
latin	83.2	± 0.7	51.5	± 1.7	ns/op
latin:·gc.churn.norm	103.1	± 1.4	71.5	± 0.8	B/op

Сжатые стринги: засады

Штуки, которые не дают нам легко жить:

- Для N версий `String`-ов нужно N^2 интринзиков
- Выбор кодера $\rightarrow$ ветвление $\rightarrow$ потенциальный mispredict
- Переупаковка на `String::new`, `String::substring` и проч.
- Интринзики на не-x86 платформах – без векторных операций плохо
- Отдельные fast-path-ы для UTF-8 кодеров

Сжатые стринги: засады

Штуки, которые не дают нам легко жить:

- Для N версий `String`-ов нужно N^2 интринзиков
- Выбор кодера $\rightarrow$ ветвление $\rightarrow$ потенциальный mispredict
- Переупаковка на `String::new`, `String::substring` и проч.
- Интринзики на не-x86 платформах – без векторных операций плохо
- Отдельные fast-path-ы для UTF-8 кодеров

Но всё это пока выглядит решаемым.
Следите за анонсами!
`https://bugs.openjdk.java.net/browse/JDK-8054307`


Q/A

Q/A: Ресурсы

- Бенчмарки:
  `https://github.com/shipilev/article-compress-me`
- Блог:
  `http://shipilev.net/`
- Твиттер:
  `http://twitter.com/shipilev`

Автобоксинг

Автобоксинг: спецификация

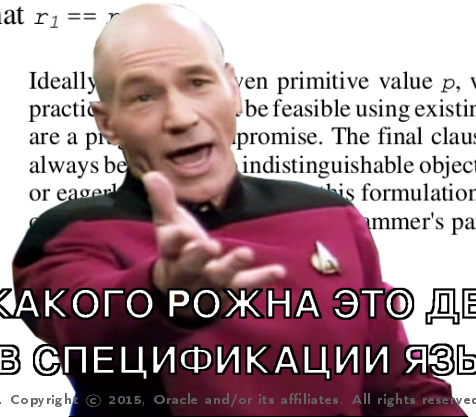
If the value *p* being boxed is `true`, `false`, a `byte`, or a `char` in the range `\u0000` to `\u007f`, or an `int` or `short` number between `-128` and `127` (inclusive), then let r_1 and r_2 be the results of any two boxing conversions of *p*. It is always the case that $r_1 == r_2$.

> Ideally, boxing a given primitive value *p*, would always yield an identical reference. In practice, this may not be feasible using existing implementation techniques. The rules above are a pragmatic compromise. The final clause above requires that certain common values always be boxed into indistinguishable objects. The implementation may cache these, lazily or eagerly. For other values, this formulation disallows any assumptions about the identity of the boxed values on the programmer's part. This would allow (but not require) sharing of some or all of these references.

Автобоксинг: спецификация

If the value *p* being boxed is `true`, `false`, a `byte`, or a `char` in the range `\u0000` to `\u007f`, or an `int` or `short` number between `-128` and `127` (inclusive), then let r_1 and r_2 be the results of any two boxing conversions of *p*. It is always the case that r_1 == r_2

> Ideally [boxing], given a primitive value *p*, would always yield an identical reference. In practice, this may not be feasible using existing implementation techniques. The rules above are a pragmatic compromise. The final clause above requires that certain common values always be boxed into indistinguishable objects. The implementation may cache these, lazily or eagerly. For other values, this formulation disallows any assumptions about the identity of boxed values on the programmer's part. This would allow (but not require) sharing



КАКОГО РОЖНА ЭТО ДЕЛАЕТ
В СПЕЦИФИКАЦИИ ЯЗЫКА?!

Java

Автобоксинг: реализация

Каждый боксинг `int` → `Integer` идёт через:

```
public class Integer {
  public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
      return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
  }
}
```


Автобоксинг: побенчмаркаемЪ

```
@Param({"42", "146"}) int value;

@Setup void setup() {
  c  = new HashSet<Integer>();
  nc = new HashSet<Integer>();c
  .add(value);
}

@Benchmark boolean contains_valueOf()
  { return c.contains(Integer.valueOf(value)); }

@Benchmark boolean contains_new()
  { return c.contains(new Integer(value)); }

// repeat the same for "nc"
```


Автобоксинг: notContains

Benchmark	value	new		valueOf		Units
notContains	42	16.2	± 0.4	6.5	± 0.3	ns/op
gc.alloc.rate.norm	42	16.0	± 0.1	$\approx 10^{-2}$		B/op
notContains	146	16.9	± 2.1	16.8	± 0.2	ns/op
gc.alloc.rate.norm	146	16.0	± 0.1	16.0	± 0.1	B/op

- Всё классно: кэш автобоксинга работает для мелких int
- Во всех остальных случаях аллоцируем 16 байт (`java.lang.Integer`)
- Вроде даже и проверка кеша почти ничего не стоит

Автобоксинг: contains

Benchmark	value	new	valueOf	Units
contains	42	9.6 ± 0.4	8.8 ± 0.1	ns/op
gc.alloc.rate.norm	42	$\approx 10^{-5}$	$\approx 10^{-5}$	B/op
contains	146	9.8 ± 0.4	17.8 ± 0.4	ns/op
gc.alloc.rate.norm	146	$\approx 10^{-4}$	16.0 ± 0.1	B/op

Автобоксинг: contains

Benchmark	value	new		valueOf		Units
contains	42	9.6	± 0.4	8.8	± 0.1	ns/op
gc.alloc.rate.norm	42	$\approx 10^{-5}$		$\approx 10^{-5}$		B/op
contains	146	9.8	± 0.4	17.8	± 0.4	ns/op
gc.alloc.rate.norm	146	$\approx 10^{-4}$		16.0	± 0.1	B/op

- Но чу! Ты слышишь – там scalar replacement!
  Взорвал объект он; стало как с кэшом...

Автобоксинг: contains

Benchmark	value	new	valueOf	Units
contains	42	9.6 ± 0.4	8.8 ± 0.1	ns/op
gc.alloc.rate.norm	42	$\approx 10^{-5}$	$\approx 10^{-5}$	B/op
contains	146	9.8 ± 0.4	17.8 ± 0.4	ns/op
gc.alloc.rate.norm	146	$\approx 10^{-4}$	16.0 ± 0.1	B/op

- Но чу! Ты слышишь – там scalar replacement!
  Взорвал объект он; стало как с кэшом...
- С насильным кешем опять же всё сломалось!

Автобоксинг: contains

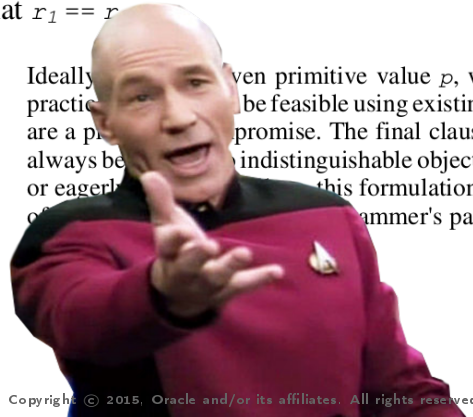
Benchmark	value	new	valueOf	Units
contains	42	9.6 ± 0.4	8.8 ± 0.1	ns/op
gc.alloc.rate.norm	42	$\approx 10^{-5}$	$\approx 10^{-5}$	B/op
contains	146	9.8 ± 0.4	17.8 ± 0.4	ns/op
gc.alloc.rate.norm	146	$\approx 10^{-4}$	16.0 ± 0.1	B/op

- Но чу! Ты слышишь – там scalar replacement!
  Взорвал объект он; стало как с кэшом...
- С насильным кешем опять же всё сломалось!
- И было бы радостно выкинуть кэш вообще и отдать всё на откуп реализациям, но...

Автобоксинг: FAIL

If the value `p` being boxed is `true`, `false`, a `byte`, or a `char` in the range `\u0000` to `\u007f`, or an `int` or `short` number between `-128` and `127` (inclusive), then let r_1 and r_2 be the results of any two boxing conversions of `p`. It is always the case that r_1 == r

> Ideally ven primitive value `p`, would always yield an identical reference. In practic be feasible using existing implementation techniques. The rules above are a pr promise. The final clause above requires that certain common values always be indistinguishable objects. The implementation may cache these, lazily or eagerl this formulation disallows any assumptions about the identity o ammer's part. This would allow (but not require) sharing



Автобоксинг: сжимай меня, если сможешь

Дизайнер нового языка, помни:
пока ты поёшь дифирамбы монадам,
где-то в твоей недоделанной спеке прячется грабля,
которая через 10 лет будет бить тебя в темечко каждый день!

