

ORACLE®

# Java Concurrency: корректность «Я твой VM конкуренси шатал»

Алексей Шипилёв  
aleksey.shipilev@oracle.com, @shipilev

MAKE THE  
FUTURE  
JAVA



# Дисклеймеры

1. Доклад рассказывает про **тестирование JVM** и **боль**.  
*(Уходите.)*
2. Доклад сложный, быстрый, беспощадный. Серьёзно.  
*(Ещё есть шанс уйти.)*
3. В роли докладчика мизантроп, сноб, и зануда.  
*(Нет, правда, никто не держит.)*
4. Доклад содержит кровь, кишки и расчленёнку JVM и JDK.  
*(Доклады про пони и бабочек в другом месте.)*

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.

# Введение

# Введение: почему мы этим занимаемся?

- Без этого заниматься производительностью толку нет
- Concurrent-баги проявляются **внезапно**
- Concurrent-баги трудно диагностировать и воспроизводить
- Производительность некорректной программы = ?

# Введение: что и где может сломаться

Тысяча и одно место:

- приложение неправильно использует JMM/библиотеку
- библиотека неправильно использует JMM/JVM
- JVM неправильно использует JMM/HW
- HW неправильно следует собственной спеке
- ...или любая комбинация из этих ошибок

# Введение: весь доклад в тезисах

- 1. везде есть ошибки: JDK/JVM/HW не безгрешны**
  - не бывает корректных программ и железок
  - (бывают плохо протестированные)
- 2. надёжная изоляция от хардвара – это мечта**
  - часто вообще шизофреническая иллюзия
  - огромная задача, посильная огромному community
- 3. пуленепробиваемый софт требует экспертизы**
  - «я пишу на Clojure, и меня это не касается» (oh wow)
  - диагностировать баги, фиксить, или обходить

# Подходы



# Подходы: prior art

## 1. Java Compability Kit (JCK)

- разрабатывается Sun/Oracle, JCP
- проверяет утверждения из JLS §17
- ограничена только явной спецификацией

## 2. JSR166 TCK

- разрабатывается Doug Lea и компанией
- проверяет функциональность `java.util.concurrent.*`
- только базовые функциональные тесты

## 3. Litmus/DIY

- кембриджская лаба 'Weak Consistency Models', Peter Sewell<sup>1</sup>
- проверяет формальную/операционную семантику хардвара
- низкоуровневый, уровня железа

---

<sup>1</sup><http://www.cl.cam.ac.uk/~pes20/>

# Подходы: главная проблема тестов

## Отлить в граните

Concurrency-баги – это баги на *гонках*

# Подходы: главная проблема тестов

Для теста требуется обеспечить контролируруемую гонку:

- обширную, чтобы потоки встречались
- ограниченную, чтобы можно было доверять результатам
- лишняя синхронизация маскирует эффекты
- скорость инфраструктуры  $\sim$  надёжность теста

К сожалению, все наивные тесты не тестируют вообще ничего!

# Подходы: попытка №1

Вот это полный бред:

```
volatile int v;  
  
void doTest() {  
    Thread t1 = new Thread(() -> v++);  
    Thread t2 = new Thread(() -> v++);  
    t1.start(); t2.start();  
    t1.join(); t2.join();  
  
    Assert.assertTrue(2, v);  
}
```

«Окошко коллизии» очень маленькое,  
и на практике коллизии не будет никогда.

## Подходы: попытка №2

Уже лучше, но всё равно бредово:

```
volatile int v;  
final CountdownLatch l = new CDL(2);  
  
void doTest() {  
    Thread t1 = new Thread(() -> l.countDown(); l.await(); v++);  
    Thread t2 = new Thread(() -> l.countDown(); l.await(); v++);  
    t1.start(); t2.start();  
    t1.join(); t2.join();  
  
    Assert.assertTrue(2, v);  
}
```

Пока потоки распаркуются, поезд уже давно уйдёт.

# Подходы: наш подход

`http://openjdk.java.net/projects/code-tools/jcstress/`

Очень похож на Litmus, но написан на Java:

- большой обновляющийся массив объектов-состояний
- много актёров, мутируют общее состояние под гонкой, сохраняют свои наблюдения в спецхранилище
- актёры никогда не паркуются, а активно ждут
- инфраструктура склеивает результаты для одного состояния и считает частотность результатов

# Подходы: наш подход

Даёт возможность абстрагировать тесты:

```
class MyTest implements ConcurrencyTest<State, Res> {  
    void actor1(State s, Res r) { r.r1 = s.v++; }  
    void actor2(State s, Res r) { r.r2 = s.v++; }  
  
    class State { volatile int v; }  
    State newState() { new State(); }  
}
```

...и считать частотность (r1, r2):

State	Occurrences	Expectation
[1, 1] (	1,360,407)	ACCEPTABLE_INTERESTING
[1, 2] (	57,137,771)	ACCEPTABLE
[2, 1] (	55,286,472)	ACCEPTABLE

# Подходы: засады

Очень похоже на бенчмаркинг:

- прогрев: несколько итераций
- недетерминизм компиляции: несколько запусков
- профили: форкать каждый тест
- многопоточность: нельзя разбрасываться потоками



# Подходы: главная засада

Быстрая и многопоточная инфраструктура тестов подвержена  
**тем самым** багам, что и ловят сами тесты.

# Подходы: главная засада

Быстрая и многопоточная инфраструктура тестов подвержена **тем самым** багам, что и ловят сами тесты.

## Принцип Шипилёва-Мюнхгаузена

Провал на одном тесте автоматически инвалидирует результаты всех остальных тестов в сюите.

# Контрольные тесты



# Контрольные тесты: мотивация

Перед тем, как бросаться на амбразуру, нужен контроль.

Надо проверить, что:

- инфраструктура может ловить легальные гонки
- инфраструктура не фейлит валидные тесты
- инфраструктура корректно синхронизована

# Контрольные тесты: синглтоны

Можно описать такой классик:

```
class MyTest implements ConcurrencyTest<State, Res> {  
    int get(SingletonFactory f) {  
        Singleton s = f.getInstance();  
        if (s == null) {  
            return 0;  
        }  
        if (s.x == null)  
            return 1;  
        return s.x;  
    }  
  
    void actor1(SingletonFactory f, Res r) { r.r1 = get(f); }  
    void actor2(SingletonFactory f, Res r) { get(f); }  
}
```

Тогда по возвращаемому значению можно будет сказать, как там у нас дела.

# Контрольные тесты: синглтоны

```
public static class UnsafeSingletonFactory implements SingletonFactory {
    private Singleton instance; // intentionally non-volatile

    public Singleton getInstance() {
        if (instance == null) {
            synchronized (this) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}

public static class Singleton {
    public Byte x;
    public Singleton() { x = 42; }
}
```

# Контрольные тесты: синглтоны

```
public static class UnsafeSingletonFactory implements SingletonFactory {
    private Singleton instance; // intentionally non-volatile

    public Singleton getInstance() {
        if (instance == null) {
            synchronized (this) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}

public static class Singleton {
    public Byte x;
    public Singleton() { x = 42; }
}
```

A: (instance == null)  
B: (instance.x == null)  
C: (instance.x == 42)  
D: NullPointerException

# Контрольные тесты: примитивы

Всем известная история про long/double:

```
class MyTest implements ConcurrencyTest<State, Res> {  
    void actor1(State s, Res r) { s.x = -1L; }  
    void actor2(State s, Res r) { r.r1 = s.x; }  
    class State { long x; }  
}
```

Каковы возможные значения r1?



# Контрольные тесты: примитивы

Всем известная история про long/double:

```
class MyTest implements ConcurrencyTest<State, Res> {  
    void actor1(State s, Res r) { s.x = -1L; }  
    void actor2(State s, Res r) { r.r1 = s.x; }  
    class State { long x; }  
}
```

Каковы возможные значения r1?

State	Occurrences	Expectation
[0] (	78,014,883)	ACCEPTABLE
[-1] (	88,013,591)	ACCEPTABLE
[4294967295] (	8,975)	ACCEPTABLE_SPEC
[-4294967296] (	30,801)	ACCEPTABLE_SPEC

JLS: long/double не атомарны.

# Контрольные тесты: примитивы

A с volatile?

```
class MyTest implements ConcurrencyTest<State, Res> {  
    void actor1(State s, Res r) { s.x = -1L; }  
    void actor2(State s, Res r) { r.r1 = s.x; }  
    class State { volatile long x; }  
}
```

# Контрольные тесты: примитивы

A с volatile?

```
class MyTest implements ConcurrencyTest<State, Res> {  
    void actor1(State s, Res r) { s.x = -1L; }  
    void actor2(State s, Res r) { r.r1 = s.x; }  
    class State { volatile long x; }  
}
```

Всё нормально, как JLS и предписывает:

State	Occurrences	Expectation
[0] (	110,284,406)	ACCEPTABLE
[-1] (	32,400,899)	ACCEPTABLE

# Только хардкор: что уже всплыло

# Дело о неправильных метках: формально

```
volatile int x; int y;
```

```
y = 1;
```

```
x = 1;
```

```
int t = y;
```

```
int r1 = x;
```

```
int r2 = y;
```

Какие (r1, r2) возможны?

# Дело о неправильных метках: формально

```
volatile int x; int y;
```

```
y = 1;
```

```
x = 1;
```

```
int t = y;
```

```
int r1 = x;
```

```
int r2 = y;
```

Какие (r1, r2) возможны?

- (1, 1) – Thread1 выполнен полностью раньше Thread2
- (0, 0) – Thread2 выполнен полностью раньше Thread1

# Дело о неправильных метках: формально

```
volatile int x; int y;
```

```
y = 1;
```

```
x = 1;
```

```
int t = y;
```

```
int r1 = x;
```

```
int r2 = y;
```

Какие (r1, r2) возможны?

- (1, 1) – Thread1 выполнен полностью раньше Thread2
- (0, 0) – Thread2 выполнен полностью раньше Thread1
- (0, 1) – легальный interleave, Thread1 успел только «y = 1»



# Дело о неправильных метках: формально

```
volatile int x; int y;
```

```
y = 1;
```

```
x = 1;
```

```
int t = y;
```

```
int r1 = x;
```

```
int r2 = y;
```

Какие (r1, r2) возможны?

- (1, 1) – Thread1 выполнен полностью раньше Thread2
- (0, 0) – Thread2 выполнен полностью раньше Thread1
- (0, 1) – легальный interleave, Thread1 успел только «y = 1»
- (1, 0) – нарушение JMM



# Дело о неправильных метках: тест-кейс

```
class MyTest implements ConcurrencyTest<State, Res> {  
    void actor1(State s, Res r) {  
        s.y = 1;  
        s.x = 1;  
    }  
    void actor2(State s, Res r) {  
        int t = s.y;  
        r.r1 = s.x;  
        r.r2 = s.y;  
    }  
    class State { volatile int x; int y; }  
}
```

# Дело о неправильных метках: тест-кейс

```
class MyTest implements ConcurrencyTest<State, Res> {  
    void actor1(State s, Res r) {  
        s.y = 1;  
        s.x = 1;  
    }  
    void actor2(State s, Res r) {  
        int t = s.y;  
        r.r1 = s.x;  
        r.r2 = s.y;  
    }  
    class State { volatile int x; int y; }  
}
```

Видим  $(r1, r2) = (1, 0)$ :

- воспроизводится не на всех машинах, а на  $\sim 10\%$  машин
- удаление «`int t = y`» устраняет проблему



# Дело о неправильных метках: oops

Ошибка в C1<sup>2</sup> CSE: проигнорировали `volatile read`:

```
t1 = y1;  
r12 = x2;  
r21 = y1;
```

...что дало возможность схлопнуть лод:

```
t1 = y1;  
r12 = x2;  
r21 = t1;
```

---

<sup>2</sup>он же «клиентский компилятор»

# Дело о неправильных метках: oops

Ошибка в C1<sup>2</sup> CSE: проигнорировали `volatile read`:

$$\begin{aligned}t_1 &= y_1; \\r1_2 &= x_2; \\r2_1 &= y_1;\end{aligned}$$

...что дало возможность схлопнуть лод:

$$\begin{aligned}t_1 &= y_1; \\r1_2 &= x_2; \\r2_1 &= t_1;\end{aligned}$$

Происходит только в C1,  
а значит «из коробки» только на Windows.

---

<sup>2</sup>он же «клиентский компилятор»

# Дело о неправильных метках: FIXED

Тривиальная ошибка в компиляторе:

- <https://bugs.openjdk.java.net/browse/JDK-7170145>
- Специфично только для -client

Исправлено:

- В чётных ветках JDK7; JDK8
- Нет тестов, которые это поймали бы раньше
- С этого фейла и начались работы по перетряхиванию concurrency-тестов

# Дело о бессмертном референте: тест-кейс

Найден Кабутцем при построении тест-кейса в дискуссии на c-i@

```
final WeakReference<T> ref = new WR(obj);  
while(ref.get() != null); | ref.clear();
```

# Дело о бессмертном референте: тест-кейс

Найден Кабутцем при построении тест-кейса в дискуссии на c-i@

```
final WeakReference<T> ref = new WR(obj);  
while(ref.get() != null); | ref.clear();
```

Из разумных соображений, этот код должен обрабатывать нормально. На деле, первый поток застревает навсегда.



# Дело о бессмертном референте: копаем

- Другие вводные:
  - воспроизводится уверенно на всех машинах
  - дизасм показывает редуцированный цикл



# Дело о бессмертном референте: копаем

- Другие вводные:
  - воспроизводится уверенно на всех машинах
  - дизасм показывает редуцированный цикл
- Поле *Reference.referent* не *volatile*
  - компилятор редуцирует код в зацикл, не реагирующий на очистки референса

```
T referent = ref.referent;  
if (referent != null)  
    while (true); // burn, MFCKR, burn!
```

# Дело о бессмертном референте: копаем

- Другие вводные:
  - воспроизводится уверенно на всех машинах
  - дизасм показывает редуцированный цикл
- Поле *Reference.referent* не *volatile*
  - компилятор редуцирует код в зацикл, не реагирующий на очистки референса

```
T referent = ref.referent;  
if (referent != null)  
    while (true); // burn, MFCKR, burn!
```

- *volatile* не поможет нативному GC

# Дело о бессмертном референте: FIXED

Продолбанное условие в компиляторе:

- <https://bugs.openjdk.java.net/browse/JDK-7190310>
- Референс может выставить GC
- Уже и так достаточно сложный код ввиду pre/post-барьеров

FIXED:

- В чётных ветках JDK7; JDK8
- Теперь VM не склеивает лоады через safepoint

# Дело о бешеном потоке: прелюдия

Случайно найден одним из студентов Кабутца:

```
Thread t1; Thread t2;  
-----  
while (!T.cT().isInterrupted()); | t1.interrupt();
```

Из разумных соображений, первый поток должен  
останавливаться.

# Дело о бешеном потоке: прелюдия

Случайно найден одним из студентов Кабутца:

```
Thread t1; Thread t2;  
-----  
while (!T.cT().isInterrupted()); | t1.interrupt();
```

Из разумных соображений, первый поток должен  
останавливаться.

**ВНЕЗАПНО:** он действительно корректно останавливается!



# Дело о бешеном потоке: тест-кейс

Сим-салябим, вытаскиваем предикат в отдельный метод:

```
Thread t1; Thread t2;  
def check() = T.cT().isInterrupted();  
-----  
while (!check()); | t1.interrupt();
```

---

<sup>3</sup>он же «серверный компилятор»

# Дело о бешеном потоке: тест-кейс

Сим-салябим, вытаскиваем предикат в отдельный метод:

```
Thread t1; Thread t2;  
def check() = T.cT().isInterrupted();  
-----  
while (!check()); | t1.interrupt();
```

И он «залипает»:

---

<sup>3</sup>он же «серверный компилятор»

# Дело о бешеном потоке: тест-кейс

Сим-салябим, вытаскиваем предикат в отдельный метод:

```
Thread t1; Thread t2;  
def check() = T.cT().isInterrupted();  
-----  
while (!check()); | t1.interrupt();
```

И он «залипает»:

- не фейлится в C1, уверенно фейлится в C2<sup>3</sup>
- break'и, модификаторы методов, etc. работают 50/50

---

<sup>3</sup>он же «серверный компилятор»



# Дело о бешеном потоке: оптимизации

№1: `Thread.isInterrupted()` читает флаг в нативной структуре:

- `volatile` ставить некуда
- Обычно такое чтение требует вызова в VM
- C2 «вклеивает» вместо этого метода код, читающий флаг

# Дело о бешеном потоке: оптимизации

№1: `Thread.isInterrupted()` читает флаг в нативной структуре:

- `volatile` ставить некуда
- Обычно такое чтение требует вызова в VM
- C2 «вклеивает» вместо этого метода код, читающий флаг

№2: Выносим инварианты из цикла:

- ...если они не зависят от переменной индукции
- ...если не дают side effects
- ...если другим способом не нарушают JMM

# Дело о бешеном потоке: интерференция

Главная причина: вклеенный код участвует в оптимизациях!

- выбрасывается из цикла, и привет.
- может выброситься по тысяче разных причин...
- ...но в нашем случае делает это только в OSR-трамплине

# Дело о бешеном потоке: интерференция

Главная причина: вклеенный код участвует в оптимизациях!

- выбрасывается из цикла, и привет.
- может выброситься по тысяче разных причин...
- ...но в нашем случае делает это только в OSR-трамплине

Следует разумный вопрос:

# Дело о бешеном потоке: интерференция

Главная причина: вклеенный код участвует в оптимизациях!

- выбрасывается из цикла, и привет.
- может выброситься по тысяче разных причин...
- ...но в нашем случае делает это только в OSR-трамплине

Следует разумный вопрос:  
Почему не ломается тривиальный пример?

```
Thread t1; Thread t2;  
-----  
while (!T.cT().isInterrupted()); | t1.interrupt();
```



# Дело о бешеном потоке: интерференция

Главная причина: вклеенный код участвует в оптимизациях!

- выбрасывается из цикла, и привет.
- может выброситься по тысяче разных причин...
- ...но в нашем случае делает это только в OSR-трамплине

Следует разумный вопрос:  
Почему не ломается тривиальный пример?

```
Thread t1; Thread t2;  
-----  
while (!T.cT().isInterrupted()); | t1.interrupt();
```

Ответ: Нам просто **ДИКО** повезло!



Java™

ORACLE®

# Дело о бешеном потоке: FIXED

- Ошибка в оптимизирующем компиляторе:
  - <http://cs.oswego.edu/pipermail/concurrency-interest/2012-November/010184.html>
  - <https://bugs.openjdk.java.net/browse/JDK-8003135>
- FIXED:
  - В чётных ветках JDK7; JDK8
  - Выставлен эксплицитный барьер в голове интринзика
  - Удивительно, что это не было поймано раньше
  - (может быть и было, но никто не заметил)

# Дело о порванных шортах: увертюра

Всем известно про (не)атомарность long/double-ов.  
А что нам известно про другие типы?

```
short s = 0;  
-----  
s = 0xFFFF; | short r1 = s;
```



# Дело о порванных шортах: увертюра

Всем известно про (не)атомарность long/double-ов.  
А что нам известно про другие типы?

$$\frac{\text{short } s = 0;}{s = 0xFFFF; \quad | \quad \text{short } r1 = s;}$$

JLS требует, что  $r1 \in \{0x0000, 0xFFFF\}$ .

# Дело о порванных шортах: увертюра

Всем известно про (не)атомарность long/double-ов.  
А что нам известно про другие типы?

$$\frac{\text{short } s = 0;}{s = 0xFFFF; \mid \text{short } r1 = s;}$$

JLS требует, что  $r1 \in \{0x0000, 0xFFFF\}$ .

Эксперимент показывает, что это требование выполняется.



# Дело о порванных шортах: идём дальше

```
short s = 0;
-----
s = 0xFFFF; | short t = s;
               | byte r1 = (byte) ((t >> 0) & 0xFF);
               | byte r2 = (byte) ((t >> 8) & 0xFF);
```

# Дело о порванных шортах: идём дальше

```
short s = 0;
-----
s = 0xFFFF; | short t = s;
               | byte r1 = (byte) ((t >> 0) & 0xFF);
               | byte r2 = (byte) ((t >> 8) & 0xFF);
```

Из интуитивных соображений:

$$(r1, r2) \in \{(0x00, 0x00), (0xFF, 0xFF)\}$$

# Дело о порванных шортах: идём дальше

```
short s = 0;
-----
s = 0xFFFF; | short t = s;
               | byte r1 = (byte) ((t >> 0) & 0xFF);
               | byte r2 = (byte) ((t >> 8) & 0xFF);
```

Из интуитивных соображений:

$$(r1, r2) \in \{(0x00, 0x00), (0xFF, 0xFF)\}$$

Эксперимент:  $(r1, r2) \in$   
 $\{(0x00, 0x00), (0xFF, 0xFF), (0x00, 0xFF), (0xFF, 0x00)\}$

# Дело о порванных шортах: копаем...

```
short s = 0;  


---

  
s = 0xFFFF;  
short t = s;  
byte r1 = (byte) ((t >> 0) & 0xFF);  
byte r2 = (byte) ((t >> 8) & 0xFF);
```

# Дело о порванных шортах: копаем...

```
short s = 0;
-----
s = 0xFFFF; | short t = s;
               | byte r1 = (byte) ((t >> 0) & 0xFF);
               | byte r2 = (byte) ((t >> 8) & 0xFF);
```

- C1 не фейлит, C2 фейлит уверенно
- так же фейлятся byte/char/short

# Дело о порванных шортах: копаем...

```
short s = 0;
-----
s = 0xFFFF; | short t = s;
              | byte r1 = (byte) ((t >> 0) & 0xFF);
              | byte r2 = (byte) ((t >> 8) & 0xFF);
```

- C1 не фейлит, C2 фейлит уверенно
- так же фейлятся byte/char/short
- volatile не помогает



# Дело о порванных шортах: оптимизации

```
short t = short_load(s.x);  
r.r1 = byte_store(and(shift(t, 0), 0xFF));  
r.r2 = byte_store(and(shift(t, 8), 0xFF));
```

# Дело о порванных шортах: оптимизации

```
short t = short_load(s.x);  
r.r1 = byte_store(and(shift(t, 0), 0xFF));  
r.r2 = byte_store(and(shift(t, 8), 0xFF));
```

...превращается в:

```
short t = short_load(s.x);  
r.r1 = byte_store(t);  
r.r2 = byte_store(shift(t, 8));
```

# Дело о порванных шортах: оптимизации

```
short t = short_load(s.x);  
r.r1 = byte_store(and(shift(t, 0), 0xFF));  
r.r2 = byte_store(and(shift(t, 8), 0xFF));
```

...превращается в:

```
short t = short_load(s.x);  
r.r1 = byte_store(t);  
r.r2 = byte_store(shift(t, 8));
```

...превращается в:

```
r.r1 = byte_store(unsigned_short_load(s.x));  
r.r2 = byte_store(shift(signed_short_load(s.x), 8));
```

# Дело о порванных шортах: сам баг

```
short t = s.x;
r.r1 = (byte) ((t >> 0) & 0xFF);
r.r2 = (byte) ((t >> 8) & 0xFF);
```

...компилируется в:

```
; reference to $s in %rdx, reference to $r in %rcx
movzwl 0xc(%rdx),%r11d      ; read s.x
mov     %r11b,0xc(%rcx)    ; store r.r1
movswl 0xc(%rdx),%r10d    ; read s.x again!
shr     $0x8,%r10d        ; shift
mov     %r10b,0xd(%rcx)   ; store r.r2
```

# Дело о порванных шортах: сам баг

```
short t = s.x;  
r.r1 = (byte) ((t >> 0) & 0xFF);  
r.r2 = (byte) ((t >> 8) & 0xFF);
```

...компилируется в:

```
; reference to $s in %rdx, reference to $r in %rcx  
movzwl 0xc(%rdx),%r11d      ; read s.x  
mov     %r11b,0xc(%rcx)    ; store r.r1  
movswl 0xc(%rdx),%r10d    ; read s.x again!  
shr     $0x8,%r10d        ; shift  
mov     %r10b,0xd(%rcx)   ; store r.r2
```

Атомарность, давай, до свиданья.



Java™

ORACLE®

# Дело о порванных шортах: FIXED

Ошибка в компиляторе:

- <https://bugs.openjdk.java.net/browse/JDK-8000805>

Исправлено:

- В чётных ветках JDK7; JDK8
- Конкретной оптимизации запрещено рождать лoadы.

# Дело о клэше волатайлов: тест-кейс

Опять компилятор виноват?

```
volatile double d = 0;  
-----  
d = Double.toRawBits(-1L); | double r1 = d;
```

JLS требует, что  $r1 \in \{0x0\dots0, 0xF\dots F\}$ .

# Дело о клэше волатайлов: тест-кейс

Опять компилятор виноват?

```
volatile double d = 0;  
-----  
d = Double.toRawBits(-1L); | double r1 = d;
```

JLS требует, что  $r1 \in \{0x0\dots0, 0xF\dots F\}$ .

На деле оказывается, что там есть ещё разорванный мусор.  
Воспроизводится **очень** редко, и на таргетированных тестах.





# Дело о клэше волатайлов: копаем

hotspot/src/share/vm/prims/unsafe.cpp<sup>4</sup>

```
#define GET_FIELD_VOLATILE(obj, offset, type_name, v) \  
    oop p = JNIHandles::resolve(obj); \  
    type_name v = \  
        OrderAccess::load_acquire(  
            (volatile type_name*) \  
            index_oop_from_field_offset_long(p, offset));
```

Unsafe\_GetDoubleVolatile() компилируется в<sup>5</sup> to :

```
mov    0x18(%esp),%ebp  
add    %ebp,%eax  
; field offset in %eax  
fldl  (%eax)  
fstpl 0x18(%esp)
```

---

<sup>4</sup>на самом деле, нет

<sup>5</sup>native GCC, targeting i586

# Дело о клэше волатайлов: фейл

```
#define GET_FIELD_VOLATILE(obj, offset, type_name, v) \  
    oop p = JNIHandles::resolve(obj); \  
    volatile type_name v = \  
        OrderAccess::load_acquire( \  
            (volatile type_name*) \  
            index_oop_from_field_offset_long(p, offset));
```

GetDoubleVolatile() на самом деле компилируется в:

```
mov    0x18(%esp),%ebp  
add    %ebp,%eax  
mov    0x4(%eax),%edx  
mov    (%eax),%eax  
mov    %eax,0x20(%esp)  
mov    %edx,0x24(%esp)  
mov    0x28(%esi),%esi  
fldl   0x20(%esp)  
mov    0x8(%esi),%eax  
mov    0x4(%esi),%ebp  
fstpl  0x18(%esp)
```

# Дело о клэше волатайлов: фейл

```
#define GET_FIELD_VOLATILE(obj, offset, type_name, v) \  
    oop p = JNIHandles::resolve(obj); \  
    volatile type_name v = \  
        OrderAccess::load_acquire(  
            (volatile type_name*)  
            index_oop_from_field_offset_long(p, offset));
```

GetDoubleVolatile() на самом деле компилируется в:

```
mov    0x18(%esp),%ebp  
add    %ebp,%eax  
mov    0x4(%eax),%edx  
mov    (%eax),%eax  
mov    %eax,0x20(%esp)  
mov    %edx,0x24(%esp)  
mov    0x28(%esi),%esi  
fldl  0x20(%esp)  
mov    0x8(%esi),%eax  
mov    0x4(%esi),%ebp  
fstpl  0x18(%esp)
```

# Дело о клэше волатайлов: FIXED

Наступили на `volatile` в компиляторе C:

- <https://bugs.openjdk.java.net/browse/JDK-8016538>

Исправлено:

- В чётных ветках JDK7; JDK8
- Идём по пути `volatile long`-ов, который не подвержден искомому багу

# Дело об атомных буферах: интермедия

Java даёт строгие гарантии на атомарность примитивных полей/массивов. Например:

```
byte[] b = new byte[100];  
-----  
b[42] = (byte)0xFF; | byte r1 = b[42];
```

# Дело об атомных буферах: интермедия

Java даёт строгие гарантии на атомарность примитивных полей/массивов. Например:

$$\frac{\text{byte}[] \text{ b} = \text{new byte}[100];}{\text{b}[42] = (\text{byte})0xFF; \quad | \quad \text{byte r1} = \text{b}[42];}$$

JLS гарантирует, что  $r1 \in \{0x00, 0xFF\}$ .

# Дело об атомных буферах: сепульки

В стандартной библиотеке есть буфера, которые «прикидываются» массивами. Что изменится в этом примере?

```
ByteBuffer b = BB.allocate(100);  
-----  
b.put(42, (byte)0xFF); | byte r1 = b.get(42);
```

# Дело об атомных буферах: сепульки

В стандартной библиотеке есть буфера, которые «прикидываются» массивами. Что изменится в этом примере?

```
ByteBuffer b = BB.allocate(100);  
-----  
b.put(42, (byte)0xFF); | byte r1 = b.get(42);
```

Оказывается, что  $r1 \in \{0x00, 0xFF\}$ .





# Дело об атомных буферах: сепуление

А вот это?

```
ByteBuffer b = BB.allocate(100);  
-----  
b.putInt(42, 0xFFFFFFFF); | int r1 = b.getInt(42);
```



# Дело об атомных буферах: сепуление

А вот это?

```
ByteBuffer b = BB.allocate(100);  
-----  
b.putInt(42, 0xFFFFFFFF); | int r1 = b.getInt(42);
```

Ожидалось бы, что  $r1 \in \{0x00000000, 0xFFFFFFFF\}$ .



# Дело об атомных буферах: сепуление

А вот это?

```
ByteBuffer b = BB.allocate(100);  
-----  
b.putInt(42, 0xFFFFFFFF); | int r1 = b.getInt(42);
```

Ожидалось бы, что  $r1 \in \{0x00000000, 0xFFFFFFFF\}$ .

На деле:  $r1 \in G$ , где  $G$  – множество всякого мусора, зависящее от платформы, endianness, и т.п.



# Дело об атомных буферах: оправдания

Считается легальным поведением, ибо работа с несинхронизованным буфером запрещена JavaDoc.

# Дело об атомных буферах: оправдания

Считается легальным поведением, ибо работа с несинхронизованным буфером запрещена JavaDoc.

Реализация:

```
class Bits {  
    ...  
    static void putIntB(ByteBuffer bb, int bi, int x) {  
        bb._put(bi, int3(x));  
        bb._put(bi + 1, int2(x));  
        bb._put(bi + 2, int1(x));  
        bb._put(bi + 3, int0(x));  
    }  
    ...  
}
```

# Дело об атомных буферах: хохма

Из этого наблюдения неявно следует ещё одна хохма.

Все они не атомарны *de jure*, а что *de facto*?

- `ByteBuffer.allocate().putInt()` **не атомарен**

# Дело об атомных буферах: хохма

Из этого наблюдения неявно следует ещё одна хохма.

Все они не атомарны *de jure*, а что *de facto*?

- `ByteBuffer.allocate().putInt()` **не атомарен**
- `ByteBuffer.allocateDirect().putInt()`

# Дело об атомных буферах: хохма

Из этого наблюдения неявно следует ещё одна хохма.

Все они не атомарны *de jure*, а что *de facto*?

- `ByteBuffer.allocate().putInt()` **не атомарен**
- `ByteBuffer.allocateDirect().putInt()` атомарен



# Дело об атомных буферах: хохма

Из этого наблюдения неявно следует ещё одна хохма.

Все они не атомарны *de jure*, а что *de facto*?

- `ByteBuffer.allocate().putInt()` **не атомарен**
- `ByteBuffer.allocateDirect().putInt()` атомарен
- `IntBuffer.allocate().putInt()`

# Дело об атомных буферах: хохма

Из этого наблюдения неявно следует ещё одна хохма.

Все они не атомарны *de jure*, а что *de facto*?

- `ByteBuffer.allocate().putInt()` **не атомарен**
- `ByteBuffer.allocateDirect().putInt()` атомарен
- `IntBuffer.allocate().putInt()` атомарен

# Дело об атомных буферах: хохма

Из этого наблюдения неявно следует ещё одна хохма.

Все они не атомарны *de jure*, а что *de facto*?

- `ByteBuffer.allocate().putInt()` **не атомарен**
- `ByteBuffer.allocateDirect().putInt()` атомарен
- `IntBuffer.allocate().putInt()` атомарен
- `IntBuffer.allocateDirect().putInt()`

# Дело об атомных буферах: хохма

Из этого наблюдения неявно следует ещё одна хохма.

Все они не атомарны *de jure*, а что *de facto*?

- `ByteBuffer.allocate().putInt()` **не атомарен**
- `ByteBuffer.allocateDirect().putInt()` атомарен
- `IntBuffer.allocate().putInt()` атомарен
- `IntBuffer.allocateDirect().putInt()` атомарен

# Дело об атомных буферах: хохма

Из этого наблюдения неявно следует ещё одна хохма.

Все они не атомарны *de jure*, а что *de facto*?

- `ByteBuffer.allocate().putInt()` **не атомарен**
- `ByteBuffer.allocateDirect().putInt()` атомарен
- `IntBuffer.allocate().putInt()` атомарен
- `IntBuffer.allocateDirect().putInt()` атомарен
- `ByteBuffer.allocate().asIntBuffer().putInt()`

# Дело об атомных буферах: хохма

Из этого наблюдения неявно следует ещё одна хохма.

Все они не атомарны *de jure*, а что *de facto*?

- `ByteBuffer.allocate().putInt()` **не атомарен**
- `ByteBuffer.allocateDirect().putInt()` атомарен
- `IntBuffer.allocate().putInt()` атомарен
- `IntBuffer.allocateDirect().putInt()` атомарен
- `ByteBuffer.allocate().asIntBuffer().putInt()` **не атомарен**

# Дело об атомных буферах: на самом деле

- «Нельзя просто так взять, и сделать полное чтение»
- Атомарность невыровненных чтений не гарантируется на целевых платформах!

# Дело об атомных буферах: на самом деле

- «Нельзя просто так взять, и сделать полное чтение»
- Атомарность невыровненных чтений не гарантируется на целевых платформах!
- Приходится выбирать из двух зол:
  1. Текущее, сразу видимое зло:
    - атомарность не гарантируется вообще никогда
    - невыровненное чтение никогда не фейлится
  2. Возможное, скрытое зло:
    - атомарность гарантируется, пока мы не пересекаем кеш-лайн
    - невыровненное чтение может внезапно зафейлиться



# Дело об атомных буферах: ?

Не очень ясно, что делать.

All Hail the Holy War:

- <http://cs.oswego.edu/pipermail/concurrency-interest/2012-December/010390.html>
- <http://mail.openjdk.java.net/pipermail/core-libs-dev/2012-December/013133.html>

# Дело о беге с барьерами: увертюра

Классический тест, «Dekker idiom» (кусочек Dekker Lock):

```
volatile int x, y;  
-----  
x = 1;      |      y = 1;  
int r1 = y; | int r2 = x;
```

# Дело о беге с барьерами: увертюра

Классический тест, «Dekker idiom» (кусочек Dekker Lock):

volatile int x, y;	
-----	
x = 1;	y = 1;
int r1 = y;	int r2 = x;

При sequentially-consistent исполнении  $(r1, r2) \notin \{(0, 0)\}$ .



# Дело о беге с барьерами: увертюра

Классический тест, «Dekker idiom» (кусочек Dekker Lock):

volatile int x, y;	
-----	
x = 1;	y = 1;
int r1 = y;	int r2 = x;

При sequentially-consistent исполнении  $(r1, r2) \notin \{(0, 0)\}$ .

Эксперимент показывает, что это требование выполняется на всех известных нам реализациях.



# Дело о беге с барьерами: предыстория

- У маленькой такой компании есть маленький такой проц:
  - в проце куча регистров
  - это побуждает тащить больше данных в регистрах
  - можно поправить register allocation (только часть проблемы)
  - надо шедулить лоады как можно раньше (больше live range)

# Дело о беге с барьерами: предыстория

- У маленькой такой компании есть маленький такой проц:
  - в проце куча регистров
  - это побуждает тащить больше данных в регистрах
  - можно поправить register allocation (только часть проблемы)
  - надо шедулить лоады как можно раньше (больше live range)
- Маленькая компания имеет форк HotSpot'а
  - пилит HotSpot под себя
  - ранний шедулинг лоадов – тривиальная оптимизация.

# Дело о беге с барьерами: предыстория, 2

- C2 обходится с кодом очень жёстко:
  - Перемалывает программу в data dependency graph, оптимизирует его, и упаковывает обратно в CFG.
  - Консистентность memory-эффектов там достигается специальными проекциями и барьерными нодами.
  - Пример:

```
volatile int x, y;  
x = 1;  
r1 = y;
```

MB → store(x, 1) → MB → load(r1, y) → MB

# Дело о беге с барьерами: оптимизация

Одна из оптимизаций клеит лишние барьеры:

MB → store(x, 1) → MB → load(r1, y) → MB



# Дело о беге с барьерами: оптимизация

Одна из оптимизаций клеит лишние барьеры:

MB  $\rightarrow$  store(x, 1)  $\rightarrow$  MB  $\rightarrow$  load(r1, y)  $\rightarrow$  MB

В этой оптимизации есть ошибка.

# Дело о беге с барьерами: оптимизация

Одна из оптимизаций клеит лишние барьеры:

MB  $\rightarrow$  store(x, 1)  $\rightarrow$  MB  $\rightarrow$  load(r1, y)  $\rightarrow$  MB

В этой оптимизации есть ошибка. Она считает, что перед `volatile read` всё равно есть leading-барьер, и стирает актуальный:

MB  $\rightarrow$  store(x, 1)  $\rightarrow$  load(r1, y)  $\rightarrow$  MB

# Дело о беге с барьерами: оптимизация

Внезапно срабатывает **та самая** маленькая оптимизация маленькой компании:

```
MB → store(x, 1) → load(r1, y) → MB  
MB → load(r1, y) → store(x, 1) → MB
```

# Дело о беге с барьерами: оптимизация

Внезапно срабатывает **та самая** маленькая оптимизация маленькой компании:

```
MB → store(x, 1) → load(r1, y) → MB  
MB → load(r1, y) → store(x, 1) → MB
```

Что вкупе со вторым потоком:

```
MB → load(r1, y) → store(x, 1) → MB  
MB → load(r2, x) → store(y, 1) → MB
```

...даёт нам  $(r1, r2) = (0, 0)$ . Упс.

# Дело о беге с барьерами: in progress

Проблема понята, думаем, как правильно исправить:

- <http://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2013-February/009604.html>
- <https://bugs.openjdk.java.net/browse/JDK-8007898>

# Дело о беге с барьерами: in progress

Проблема понята, думаем, как правильно исправить:

- <http://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2013-February/009604.html>
- <https://bugs.openjdk.java.net/browse/JDK-8007898>

Кроме того:

- Нет релевантных падений на других тестах
- Написали внутрь VM instruction scheduling fuzzer, падение только на этом тесте

# Дело о б. перестановках: сыр-бор

Те же парни оптимизируют свою платформу дальше:

```
AtomicInteger ai;  
-----  
ai = new AtomicInteger(42); | r1 = ai.get();
```

# Дело о б. перестановках: сыр-бор

Те же парни оптимизируют свою платформу дальше:

$$\frac{\text{AtomicInteger } ai;}{ai = \text{new AtomicInteger}(42); \mid r1 = ai.get();}$$

Из здравого смысла, в отсутствие NPE:  $r1 \in \{42\}$ .



# Дело о б. перестановках: упростим

```
class A {  
    volatile int f;  
    A(int v) { f = v; }  
}
```

```
A a;
```

---

```
a = new A(42); | r1 = a.f;
```

# Дело о б. перестановках: упростим

```
class A {  
    volatile int f;  
    A(int v) { f = v; }  
}
```

```
A a;
```

---

```
a = new A(42); | r1 = a.f;
```

Опять же, из здравого смысла, в отсутствие NPE:  $r1 \in \{42\}$ .

# Дело о б. перестановках: ещё проще

```
class A {  
    volatile int f;  
    A(int v) { f = v; }  
}
```

```
A a;
```

```
A ta = <CreateObject>;  
ta.f = 42;  
a = ta;
```

...

```
A ta = a;  
r1 = ta.f;
```

# Дело о б. перестановках: ещё проще

```
class A {  
    volatile int f;  
    A(int v) { f = v; }  
}
```

```
A a;
```

```
A ta = <CreateObject>;
```

```
ta.f = 42;
```

```
a = ta;
```

...

```
A ta = a;
```

```
r1 = ta.f;
```

reorder!

# Дело о б. перестановках: реордеринг?

```
class A {  
    volatile int f;  
    A(int v) { f = v; }  
}
```

```
A a;
```

<pre>A ta = &lt;CreateObject&gt;; a = ta; ta.f = 42;</pre>	<pre>A ta = a; r1 = ta.f;</pre>
--	-------------------------------------

АТЬ!  $r1 \in \{0, 42\}$ ?!

# Дело о б. перестановках: cookbook

Получается, что нет возможности безопасно сделать недоконструированный объект?

Идиома «Поменяй свои `final` на `volatile` и ничего не сломается» – **BUSTED!**

# Дело о б. перестановках: cookbook

Получается, что нет возможности безопасно сделать недоконструированный объект?

Идиома «Поменяй свои `final` на `volatile` и ничего не сломается» – **BUSTED!**

Или JSR 133 Cookbook нам **врёт?**

# Дело о б. перестановках: BUSTED

- JMM разрешает подобное «неожиданное» поведение
  - На x86 замена `final` на `volatile` действительно работает
  - На других архитектурах фейлит, чем оставляет юзеров в шокирующем недоумении
  - Некоторые реализации JVM проглотили пилюлю и запрещают такое поведение даже для `volatile`
- <http://cs.oswego.edu/pipermail/concurrency-interest/2013-November/011951.html>



# Дело о фантомном ППЦэ: интермедия

Просто массив, просто default values:

```
int[] arr;  
-----  
arr = new int[1]; | r1 = arr[0];
```

# Дело о фантомном ППЦэ: интермедия

Просто массив, просто default values:

$$\frac{\text{int [] arr;}}{\text{arr = new int [1];} \mid \text{r1 = arr [0];}$$

Из здравого смысла:  $r1 \in \{0\}$ .

# Дело о фантомном ППЦэ: интермедия

Просто массив, просто default values:

```
int[] arr;  
-----  
arr = new int[1]; | r1 = arr[0];
```

Из здравого смысла:  $r1 \in \{0\}$ .

А вот на некотором серваке:  $r1 \in G$ ,  
где  $G$  – множество всякого мусора.

# Дело о фантомном ППЦэ: копаем

- Обычные тесты:
  - воспроизводится только на конкретной машине, 6-core PPC
  - воспроизводится только на массивах
  - объявление массива `volatile` «чинит» проблему

# Дело о фантомном ППЦэ: копаем

- Обычные тесты:
  - воспроизводится только на конкретной машине, 6-core PPC
  - воспроизводится только на массивах
  - объявление массива `volatile` «чинит» проблему
- Дальше – больше:
  - барьеры выставлены правильно, в соответствии со спекой
  - перерываем спецификации...

# Дело о фантомном ППЦэ: копаем

- Обычные тесты:
  - воспроизводится только на конкретной машине, 6-core PPC
  - воспроизводится только на массивах
  - объявление массива `volatile` «чинит» проблему
- Дальше – больше:
  - барьеры выставлены правильно, в соответствии со спекой
  - перерываем спецификации... роем...

# Дело о фантомном ППЦэ: копаем

- Обычные тесты:
  - воспроизводится только на конкретной машине, 6-core PPC
  - воспроизводится только на массивах
  - объявление массива `volatile` «чинит» проблему
- Дальше – больше:
  - барьеры выставлены правильно, в соответствии со спекой
  - перерываем спецификации... роем... роем...

# Дело о фантомном ППЦэ: копаем

- Обычные тесты:
  - воспроизводится только на конкретной машине, 6-core PPC
  - воспроизводится только на массивах
  - объявление массива `volatile` «чинит» проблему
- Дальше – больше:
  - барьеры выставлены правильно, в соответствии со спекой
  - перерываем спецификации... роем... роем... находим!
  - « Yet, hpcx exhibits non-SC behaviours for some A-cumulativity tests, [...]. **We understand that this is due to an erratum in the Power 5 implementation.** IBM is providing a workaround, replacing the sync barrier by a short code sequence [...]»



# Дело о фантомном ППЦэ: WTF

- Хардварный баг!
  - что делать, не очень ясно
  - к результатам на конкретной машине теперь относятся подозрительно
  - (потенциально отправляется на свалку)
- <https://bugs.openjdk.java.net/browse/JDK-8007283>
- <http://www0.cs.ucl.ac.uk/staff/j.alglave/papers/cav10.pdf>

# Дело о тоталитарных порядках: формально

Есть классический тест, IRIW:

```
volatile int x, y;  
-----  
x = 1; | y = 1; | int r1 = y; | int r3 = x;  
       |       | int r2 = x; | int r4 = y;
```

# Дело о тоталитарных порядках: формально

Есть классический тест, IRIW:

```
volatile int x, y;  
-----  
x = 1; | y = 1; | int r1 = y; | int r3 = x;  
       |       | int r2 = x; | int r4 = y;
```

Состояние  $(r1, r2, r3, r4) = (1, 0, 1, 0)$  запрещено:  
прочитали  $x$  и  $y$  в разном порядке.

# Дело о тоталитарных порядках: Power Again

- Сравнительно новое наблюдение:
  - практически все платформы обеспечивают «write atomicity»
  - апдейт в конкретное место виден сразу всем
  - единственная платформа, где это не выполняется – PowerPC
  - (они это даже считают фичей)

# Дело о тоталитарных порядках: Power Again

- Сравнительно новое наблюдение:
  - практически все платформы обеспечивают «write atomicity»
  - апдейт в конкретное место виден сразу всем
  - единственная платформа, где это не выполняется – PowerPC
  - (они это даже считают фичей)
- Это означает, что простого acquire-барьера недостаточно:

```
Load Acquire: ld; cmp; bc; isync
```

```
Load Seq Cst: hwsync; ld; cmp; bc; isync
```

# Дело о тоталитарных порядках: Power Again

- Сравнительно новое наблюдение:
  - практически все платформы обеспечивают «write atomicity»
  - апдейт в конкретное место виден сразу всем
  - единственная платформа, где это не выполняется – PowerPC
  - (они это даже считают фичей)

- Это означает, что простого acquire-барьера недостаточно:

```
Load Acquire: ld; cmp; bc; isync  
Load Seq Cst: hwsync; ld; cmp; bc; isync
```

- Требуется **два** барьера на volatile read'е, чтобы обеспечить sequential consistency!



# Дело о тоталитарных порядках: BUSTED

- Та же беда с JSR133 cookbook'ом:
  - проблема с PPC найдена в ходе стандартизации C++11 MM
  - но она до сих пор не отражена в JSR133 Cookbook!
  - разработчики портов на PPC негодуэ
- <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2177.html>
- <http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html>
- <http://cs.oswego.edu/pipermail/concurrency-interest/2013-January/010608.html>

# Выводы





# Выводы: JDK/JVM/HW не безгрешны

- Bugs, bugs everywhere:
  - В приложениях куча ошибок.
  - В JDK куча ошибок.
  - В JVM куча ошибок.
  - В хардваре куча ошибок.
  - В документации куча ошибок.

# Выводы: JDK/JVM/HW не безгрешны

- Bugs, bugs everywhere:
  - В приложениях куча ошибок.
  - В JDK куча ошибок.
  - В JVM куча ошибок.
  - В хардваре куча ошибок.
  - В документации куча ошибок.
  
- Мы работаем над этим, но там поле непаханное.

# Выводы: JDK/JVM/HW не безгрешны

- Bugs, bugs everywhere:
  - В приложениях куча ошибок.
  - В JDK куча ошибок.
  - В JVM куча ошибок.
  - В хардваре куча ошибок.
  - В документации куча ошибок.
- Мы работаем над этим, но там поле непаханное.
- Баги исправляются со временем  
⇒ **планируйте обновления софта и железа!**

# Выводы: нам нужна помощь!

- Не проходи мимо:
  - Увидел подозрительное поведение, потыкай в него палочкой.
  - Потыкал и ничего не понял, спроси.
  - Потыкал и понял, что оно сломано, рапортуй!
  - Потыкал, понял, знаешь как исправить – тем более рапортуй!

# Выводы: нам нужна помощь!

- Не проходи мимо:
  - Увидел подозрительное поведение, потыкай в него палочкой.
  - Потыкал и ничего не понял, спроси.
  - Потыкал и понял, что оно сломано, рапортуй!
  - Потыкал, понял, знаешь как исправить – тем более рапортуй!
- Пишите в Спортлото!
  - `concurrency-interest@cs.oswego.edu`
  - или лично, `aleksey.shipilev@oracle.com`
  - мы с радостью потыкаем в это вместе с вами.



# Выводы: Personal Appeal



Дяденьки и тётеньки, вы вместо очередного %bullshit%.js, лучше попишите тестов для вашей платформы, пожалуйста, а?