ORACLE

Java Memory Model

прагматика модели

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

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: Картинка-детектор

Введение: Абстрактные машины

- Любой язык программирования определяет свою семантику через поведение абстрактной машины, выполняющей программу на этом языке.
- Спецификация языка = спецификация абстрактной машины[1]

Показательный пример:
`Brainfuck`[2] = типичный ассемблер для машины Тьюринга

[1] Java ≠ Java bytecode ⇒ спецификация Java ≠ спецификация JVM

[2] `http://en.wikipedia.org/wiki/Brainfuck`

Введение: Модель памяти

- Часть спецификации абстрактной машины: модель того, как работает хранилище данных = *модель памяти*
- Оказывается, что модели памяти достаточно ответить на один простой вопрос...

Введение: Модель памяти

- Часть спецификации абстрактной машины: модель того, как работает хранилище данных = *модель памяти*
- Оказывается, что модели памяти достаточно ответить на один простой вопрос...

Основной вопрос её существования (ОВЁС):
Какие значения может прочитать
конкретный `read` в программе?

Введение: В последовательных программах...

- Исполняем инструкции языка одну за одной? Тогда модель памяти очевидна:

  «Чтения, следующие за записями в программе, должны видеть ранее записанные значения»[3]

- Часто под «моделью памяти» имеют в виду «модель памяти, покрывающая семантику многопоточных программ»

[3] e.g. для C99: ISO/IEC 9899:1999, «5.1.2.3 Program execution»

Введение: ...тоже не всё просто

- Хрестоматийный пример для C 89/99:

```
int i = 5; (.)
i = (++i + ++i); (.)
assert (13 == i); (.) // FAILS
```

- Отсутствие *sequence point*'ов[4] в выражении приводит к undefined behavior (между `(.)` `(.)` может быть что угодно)
- Модели памяти нужны в том числе для того, чтобы судить о поведении однопоточных программ

[4] ISO/IEC 9899:1999, «Annex C: Sequence Points»

Введение: Coming back to reality

Реализации ЯП делают одну из двух вещей:

1. напрямую эмулируют абстрактную машину на входной программе и существующей машине (интерпретация)
2. специализируют абстрактную машину входной программой, и выполняют результат на существующей машине (компиляция)

Реализациям нужно вложиться в специфицированное поведение абстрактной машины ЯП.

Введение: Be careful what you wish for

Модель памяти = trade-off между
долбанутостью программирования *на языке*,
долбанутостью *быстрой и корректной реализации языка*,
и долбанутостью *хардвара*

- Мечтать не вредно: можно потребовать много удобных штук
- ...вопрос в том, не уйдёт ли на создание подходящей реализации ЯП и железа к нему стотыщмильонов лет?

Введение: логика повествования

Мы попробуем:

1. Показать, что JMM нам нужна
2. Обозначить наши желания
3. Посмотреть, что нам реально доступно
4. Понять, как спецификация балансирует между (2) и (3)
5. Заглянуть, как работают (консервативные) реализации JMM

Формальные требования JMM будут вот в такой рамке

Access atomicity

Access atomicity: Сказка

Хочется:

Атомарность доступа к базовым типам

Т.е. для любого базового типа T:

`T t = V1;`	
`t = V2;`	`T r1 = t;` `assert (r1 ∈ {V1, V2})`

Access atomicity: Реальность

Нужна поддержка со стороны железа, чтобы оно действительно делало атомарные чтения/записи

Засады:

- Отсутствие хардварных операций для крупных чтений: как прочитать 8-байтный `long` на 32-битном x86? А на 32-битном ARM-е?
- Требования подсистемы памяти: в примеру, при пересечении кеш-лайна на x86 атомарность теряется

Access atomicity: Решение (часть 1/2)

Чтения/записи атомарны для всего, кроме `long` и `double`

`volatile long` и `volatile double` атомарны

- Большая часть железа в 2004 умела читать до 32 бит за раз, с 64-битными чтениями пришлось мириться
- Ссылки имеют битность, соответствующую машинной
- Можем вернуть атомарность (подчёркивая возможный performance penalty)

Access atomicity: Решение (часть 2/2)

Почти везде невыровненное чтение теряет атомарность
(и уж точно теряет производительность)

- Реализация вынуждена выравнивать типы по их длине:

```
o.o.j.samples.JOLSample_02_Alignment.A[5]
 OFFSET  SIZE  TYPE DESCRIPTION
      0    12       (object header)
     12     4       (alignment/padding gap)
     16     8  long A.f
```

[5] http://openjdk.java.net/projects/code-tools/jol/

Access atomicity: Quiz

Что напечатает?

```
AtomicLong al = new AtomicLong();
```

`al.set(-1L);`	`println(al.get());`

Access atomicity: Quiz

Что напечатает?

```
AtomicLong al = new AtomicLong();
```

`al.set(-1L);`	`println(al.get());`

Почему не 0xFFFFFFFF00000000?

Access atomicity: Quiz

Что напечатает?

```
AtomicLong al = new AtomicLong();
```

`al.set(-1L);`	`println(al.get());`

Почему не 0xFFFFFFFF00000000?

Никакой магии: «`volatile long`» внутри гарантирует.

Access atomicity: Value types

- Многие хотят value types в Java. Кроме всяких бонусов, они вносят лёгкий аромат неадеквата в модель памяти.
- К примеру, C/C++11 требует атомарность для любого POD:

```
typedef struct TT {
    int a, b, c, ..., z; // 104 bytes
} T;
std::atomic<T> atomic();
```

`atomic.set(new T());`	`T t = atomic.get();`

Access atomicity: Value types

- Многие хотят value types в Java. Кроме всяких бонусов, они вносят лёгкий аромат неадеквата в модель памяти.
- К примеру, C/C++11 требует атомарность для любого POD:

```
typedef struct TT {
    int a, b, c, ..., z; // 104 bytes
} T;
std::atomic<T> atomic();
```

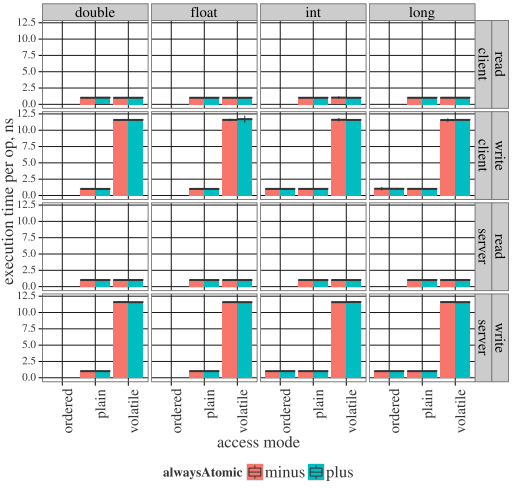
`atomic.set(new T());`	`T t = atomic.get();`

- Реализация **вынуждена** делать lock и на `set()`, и на `get()`

Access atomicity: JMM 9

- Исключения для `long`/`double` вызваны прагматикой 2004 г.
  - Повсеместно распространены 32-битные x86
  - Древние, дремучие ARM'ы и PowerPC'ы
- В 2014 году уже всё гораздо лучше!
  - В серверном мире остались вообще 32-битные машины?
  - Даже на 32-битных давно есть 64-битные (векторные) инструкции
  - На большинстве платформ `long`/`double` де-факто атомарны
  - ...но мы всё равно заставляем писать `volatile`, ибо WORA
- Вопрос: Может, пора выпилить эти исключения из спецификации?

Access atomicity: JMM 9



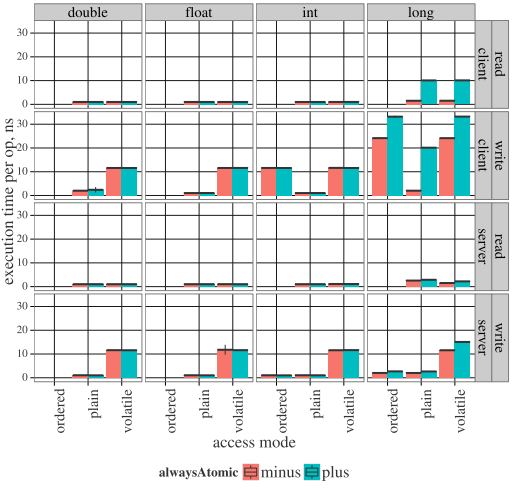
x86, Ivy Bridge, 64-bit:

Никакой разницы:

- `double` уже давно атомарен
- `long` работает на нативной битности

[6] http://shipilev.net/blog/2014/access-atomic/

Access atomicity: JMM 9



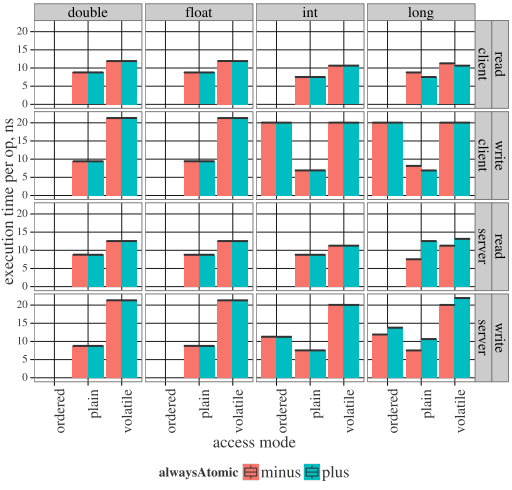
x86, Ivy Bridge, 32-bit:

Разницы чуть-чуть:

- `double` уже давно атомарен
- `long` на живых векторных инструкциях

[7] http://shipilev.net/blog/2014/access-atomic/

Access atomicity: JMM 9



ARMv7, Cortex-A9, 32-bit:

Разницы чуть-чуть:

- `double` уже давно атомарен
- `long` на живых векторных инструкциях

[8] http://shipilev.net/blog/2014/access-atomic/

Word tearing

Word tearing: Сказка

Хочется:

Незалежность операций над независимыми элементами (полями, элементами массивов и т.п.). Например:

```
T[] as = new T[...]; as[1] = as[2] = V0;
```

`as[1] = V1;` <term>	`as[2] = V1;` <term>	<join both> `T r1 = as[1];` `T r2 = as[2];` `assert (r1 == r2)`

Word tearing: Реальность

Нужна поддержка со стороны железа, чтобы оно действительно делало независимые чтения/записи

Засады:

- Отсутствие хардварных операций для мелких чтений/записей: как атомарно записать 1-битный `boolean`, если атомарно можно записать N ($N \geq 8$) бит?

Word tearing: Решение

Word tearing запрещён

- Большая часть железа умеет адресовать от 8 бит за раз
- Если железо умеет адресовать минимум N бит, значит, минимальный размер базового типа *в реализации* тоже разумно сделать N бит
- На большинстве платформ все типы не теряют память (кроме 8-битного `boolean`)

Word tearing: Experimental Proof

Объекты выровнены на 8 байт.
Всё, кроме `boolean`, точного размера под диапазон значений:

```
$ java -jar jol-internals.jar ...
 Running 64-bit HotSpot VM.
 Using compressed references with 3-bit shift.
 Objects are 8 bytes aligned.
 Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
 Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
```


Word tearing: Quiz

Что напечатает?

```
BitSet bs = new BitSet();
```

`bs.set(1);`	`bs.set(2);`	
`<term>`	`<term>`	`<join both>`
		`println(bs.get(1));`
		`println(bs.get(2));`

[9] Есть ли хоть одна реализация, которая напечатает (F, F)?

Word tearing: Quiz

Что напечатает?

```
BitSet bs = new BitSet();
```

`bs.set(1);`	`bs.set(2);`	
`<term>`	`<term>`	`<join both>`
		`println(bs.get(1));`
		`println(bs.get(2));`

Напечатает любую[9] из комбинаций (T, T), (F, T), (T, F).

[9] Есть ли хоть одна реализация, которая напечатает (F, F)?

Word tearing: Bit fields

- Многие хотят `struct`'ы в Java, или вообще способ контролировать layout объектов. Ага, удачи, сэмулируй это:

```
typedef struct TT {
    unsigned a:7;
    unsigned b:3;
} T;
```

T t;	
`t.a = 42;`	`r1 = t.b;`

Word tearing: Bit fields

- Многие хотят `struct`'ы в Java, или вообще способ контролировать layout объектов. Ага, удачи, сэмулируй это:

```
typedef struct TT {
    unsigned a:7;
    unsigned b:3;
} T;
```

T t;	
`t.a = 42;`	`r1 = t.b;`

- Реализация **вынуждена** делать lock и на записи a, и на чтении b.

Word tearing: Bit fields

- Многие хотят `struct`'ы в Java, или вообще способ контролировать layout объектов. Ага, удачи, сэмулируй это:

```
typedef struct TT {
    unsigned a:7;
    unsigned b:3;
} T;
```

T t;	
`t.a = 42;`	`r1 = t.b;`

- Реализация **вынуждена** делать lock и на записи a, и на чтении b. (C++11 на это феерически забил)

Word tearing: JMM 9

MOVE
ALONG
NOTHING
TO SEE
HERE

ПЕРЕРЫВ

на поесть, попить, покурить

SC-DRF

SC-DRF: Сказка

Хочется:

Простой способ анализировать приложения,
«Sequential Consistency». К примеру:

<action(1, 1)>	<action(2, 1)>
<action(1, 2)>	<action(2, 2)>
<action(1, 3)>	<action(2, 3)>

Удобно думать, что операции исполняются по порядку,
иногда переключаясь на другой поток

SC-DRF: Сказка (формальнее)

Sequential Consistency (SC):

(Лампорт, 1979): «Результат любого исполнения не отличим от случая, когда все операции на всех процессорах исполняются в некотором последовательном порядке, и операции на конкретном процессоре исполняются в порядке, обозначенном программой»

SC-DRF: Сказка (формальнее)

SC – иезуитское определение:

- Программу можно сильно переколбасить, лишь бы нашёлся нужный порядок в оригинальной программе, который приводит к тому же SC-результату

```
int a = 0, b = 0;
```

`a = 1;`	`b = 2;`
`print(b);`	`print(a);`

→

```
...
```

`print(2);`	`print(1);`

SC-DRF: Реальность

- Отношения оптимизаций и модели памяти можно выразить через перестановки чтений/записей
- Можно ли осуществить это преобразование?

```
int a = 0, b = 0;
-----------------
r1 = a;
r2 = b;
```

→

```
int a = 0, b = 0;
-----------------
r2 = b;
r1 = a;
```

SC-DRF: Реальность

int a = 0, b = 0;	
r1 = a;	b = 2;
r2 = b;	a = 1;

int a = 0, b = 0;	
r2 = b;	
	b = 2;
	a = 1;
r1 = a;	

- В исходной программе при SC обязательно либо «`r2 = b`», либо «`a = 1`» должно быть последним, а значит, `(r1, r2)` либо `(*, 2)`, либо `(0, *)`.
- Новая программа приводит к `(r1, r2) = (1, 0)`

SC-DRF: Реальность

Sequential Consistency - очень привлекательная модель.
Даёшь её в массы в XVII пятилетке!

- Очень сложно определить, какие оптимизации можно делать, не нарушая при этом SC
- Глобальный МегаОптимизатор (ГМО) *в теории* может сделать такой анализ
- *На практике* и рантаймы, и железо получаются исключительно без ГМО $\Rightarrow$ большая часть оптимизаций запрещена

SC-DRF: Решение

Нам нужна более слабая модель!
(Вспоминаем про trade-off-ы)

Если подойти к делу основательно, то:

- Разрешим оптимизации в рантаймах и хардваре
- Все ещё сохраним остатки разума девелоперов
- Спецификация будет невменяема, но чуть менее чем наполовину

SC-DRF: Логика построения модели

Формализм JMM пытается получить ответ на ОВЁС
(и делает это неконструктивно!)

- JMM строит все возможные исполнения программы
  - Действия программы образуют program/synchronization order
  - Из них рождаются synchronizes-with, happens-before order
- JMM отбрасывает запрещенные исполнения
  - Для этого порядки имеют структурные ограничения
  - Потом на исполнения налагается ещё немножко ограничений
  - Потом исполнения ещё чуть-чуть валидируются
- Оставшиеся исполнения – разрешены

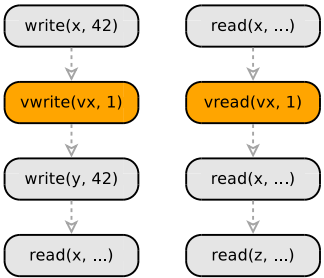
SC-DRF: Логика построения модели

Формализм JMM пытается получить ответ на ОВЁС
(и делает это неконструктивно!)

- JMM строит все возможные исполнения программы
  - Действия программы образуют program/synchronization order
  - Из них рождаются synchronizes-with, happens-before order
- JMM отбрасывает запрещенные исполнения
  - Для этого порядки имеют структурные ограничения
  - Потом на исполнения налагается ещё немножко ограничений
  - Потом исполнения ещё чуть-чуть валидируются
- Оставшиеся исполнения – разрешены

SC-DRF: Program Order

Program Order (PO):
порядок действий внутри одного потока



- PO связывает действия только для конкретного потока
- JMM работает только с действиями над полями и элементами массивов
- PO не хватает, чтобы судить о многопоточных программах: надо связать PO со всей остальной реальностью

SC-DRF: Лирическое отступление

Порядок *действий* $\neq$ порядок *операторов*
(хотя эти два порядка обманчиво схожи)

```
opA();
if (predicate()) {
  opB();
} else {
  opC();
}
opD();
```

- Операторы полный порядок могут не образовывать, а
  Program Order – полный порядок
- Программа *генерирует* действия по ходу исполнения

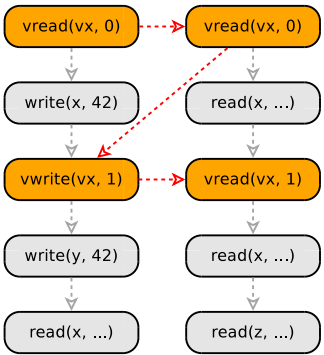
SC-DRF: Synchronization Actions

Определим специальные операции!

Synchronization Actions (SA):

- volatile read, volatile write
- lock monitor, unlock monitor
- первое и последнее действие в потоке
- операции, обнаруживающие завершение потока (Thread.join(), Thread.isInterrupted() и т.п.)

SC-DRF: Synchronization Order

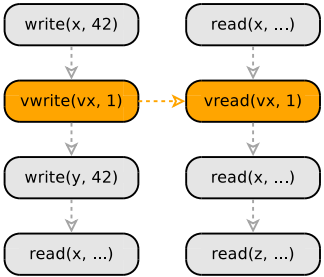


Synchronization Actions
образуют Synchronization Order (SO):

- SO – полный порядок (total order)
  = все видят один порядок SA

- SO согласован с порядком SA в PO
  = сохраняются, например, свойства парности lock/unlock

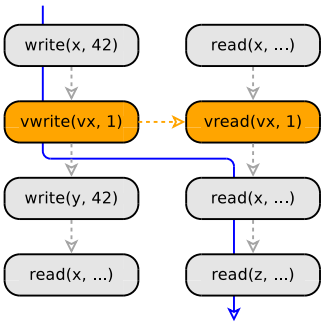
SC-DRF: Synchronizes-With Order (SW)



Synchronizes-With Order (SW):
подпорядок SO, ограниченный для конкретных пар чтений/записей, lock/unlock и т.п

- SO неоправданно ограничивает оптимизации: зачем требовать порядок volatile read'ов в двух разных потоках?

SC-DRF: Happens-before (HB)



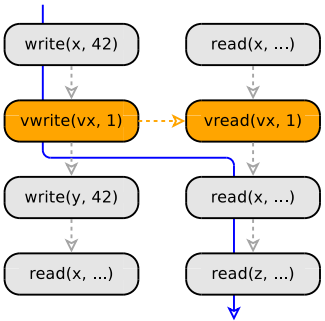
Happens-Before Order (HB):

транзитивное замыкание объединения

$$\{SW \cup PO\}_+$$

- PO даёт семантику внутри потока
- SW даёт «мостик» между потоками
- Результат из обоих порядков: HB!

SC-DRF: Happens-before (HB)



HB даёт понимание,
какую запись может увидеть
конкретное чтение (ОВЁС!)

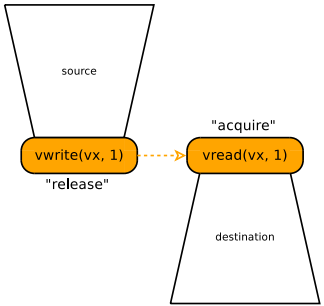
- Либо последнюю запись в HB
- Либо любую запись вне HB (гонка)
- Казуистика: *гонкой* называется конфликтующий доступ к переменной, не связанный happens-before

SC-DRF: Определение

SC-DRF: «Correctly synchronized programs have sequentially consistent semantics»

- Перевод: В программе нет гонок $\Rightarrow$ результат исполнения программы можно объяснить каким-нибудь SC-исполнением
- Интуиция №1: Операции над локальными данными (как правило) не ломают SC
- Интуиция №2: Операции над глобальными данными синхронизованы SC-примитивами (доступными прямо в железе)

SC-DRF: Publication



Это приводит к механизмам безопасной публикации:

- Работает только на одной и той же переменной, одном и том же мониторе
- Работает только если мы *увидели* ту самую `release`-запись
- Всегда парные действия! Нельзя сделать `release` в одной стороне, и не делать `acquire` в другой.

SC-DRF: Quiz

Сейчас соптимизируем...

```
class C<T> {
  T box;
  public synchronized void set(T v) {
    if (box == null) { box = v; }
  }
  public synchronized T get() {
    // TODO FIXME PLEASE PLEASE PLEASE:
    // THIS ONE IS TOO HOT IN PROFILER!!!111ONEONEONE
    return box;
  }
}
```


SC-DRF: Quiz

Ну, кто недавно так упарывался?

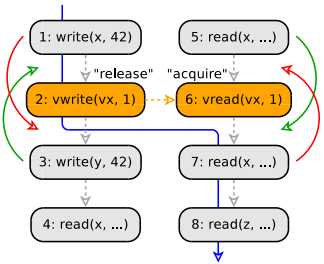
```
class C<T> {
  T box;
  public synchronized void set(T v) {
    if (box == null) { box = v; }
  }
  public T get() {
    // This one is safe without the synchronization.
    //   (Yours truly, CERTIFIED SENIOR JAVA DEVELOPER)
    return box;
  }
}
```


SC-DRF: Quiz

Так уже лучше: вернули SW-ребро, восстановили HB.

```
class C<T> {
  volatile T box;
  public synchronized void set(T v) {
    if (box == null) { box = v; }
  }
  public T get() {
    // This one is safe without the synchronization.
    // <Sigh>. Now it's safe.
    //  ($PROJECT techlead, overseeing certified idiots)
    return box;
  }
}
```

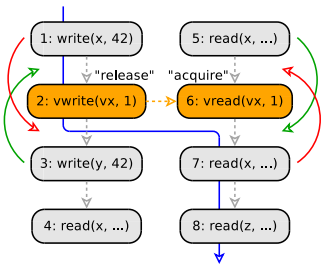

SC-DRF: Roach Motel



Одна из интерпретаций этой модели разрешает простой класс оптимизаций, «Roach Motel»:

(3) можно переставить перед `release`, потому что он не мешает никаким зависимостям по x, и чтения справа могут видеть эту запись через гонку

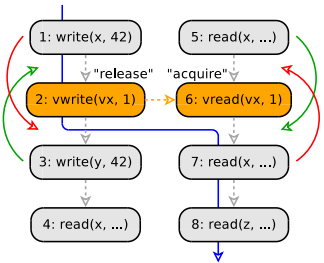
SC-DRF: Roach Motel



Одна из интерпретаций этой модели разрешает простой класс оптимизаций, «Roach Motel»:

(5) можно переставить за `acquire`, потому что он всё равно читает x через гонку, может увидеть и 42

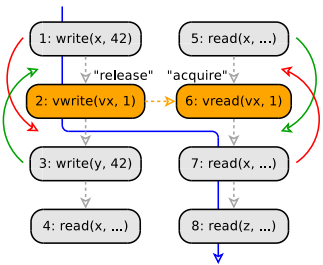
SC-DRF: Roach Motel



Одна из интерпретаций этой модели разрешает простой класс оптимизаций, «Roach Motel»:

Тащемта, «Вносибельно после `acquire`» + «Вносибельно перед `release`» = «Вносибельно в `acquire`+`release` блоки» $\Rightarrow$ работает lock coarsening, например.

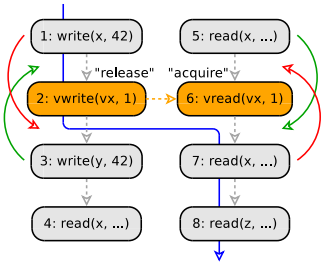
SC-DRF: Roach Motel



Консервативным реализациям некоторые перестановки запрещены:

(1) нельзя просто переставить за `release`, потому что мы потенциально выносим его из HB. Консервативная реализация не знает, есть ли дальше (7), которая должна его увидеть. ГМО мог бы это узнать и таки переставить.

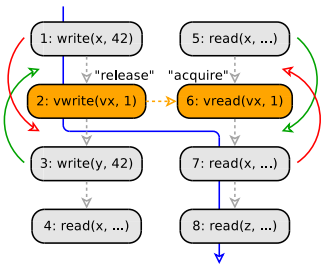
SC-DRF: Roach Motel



Консервативным реализациям некоторые перестановки запрещены:

(7) нельзя просто переставить перед `acquire`, потому что это выносит его из HB. Консервативная реализация не знает, есть ли вверх по течению (1), значение которого мы должны увидеть. Опять же, ГМО мог бы это проанализировать и таки переставить.

SC-DRF: Roach Motel



Консервативным реализациям некоторые перестановки запрещены:

ГМО мог бы перетащить и (8) вперёд acquire, если бы доказал, что в HB-пути никто не делает записей в z. Консервативным реализациям такое счастье недоступно.

SC-DRF: Quiz

Что напечатает? Варианты: 0, 41, 42, 43, <ничего>

```
                int a = 0;
volatile boolean ready = false;
```

`a = 41;`	`while(!ready);`
`a = 42;`	`println(a);`
`ready = true;`	
`a = 43;`	

SC-DRF: Quiz

Что напечатает? Варианты: 0, 41, 42, 43, <ничего>

```
int a = 0;
volatile boolean ready = false;
```

`a = 41;`	`while(!ready);`
`a = 42;`	`println(a);`
`ready = true;`	
`a = 43;`	

Напечатает или 42, или 43.

SC-DRF: Quiz #2

Что напечатает? Варианты: 0, 41, 42, 43, <ничего>

```
int a = 0;
boolean ready = false;
```

`a = 41;`	`while(!ready);`
`a = 42;`	`println(a);`
`ready = true;`	
`a = 43;`	

SC-DRF: Quiz #2

Что напечатает? Варианты: 0, 41, 42, 43, <ничего>

```
int a = 0;
boolean ready = false;
```

`a = 41;`	`while(!ready);`
`a = 42;`	`println(a);`
`ready = true;`	
`a = 43;`	

Напечатает или 0, или 41, или 42, или 43, или <ничего>.

SC-DRF: Немножко бенчмарков

`https://github.com/shipilev/jmm-benchmarks/`

- 2x12x2 Xeon E5-2697, 2.70GHz
- OEL 6, JDK 7u40, x86_64
- Измеряем не производительность спеки, а производительность некоторой её реализации

SC-DRF: Немножко бенчмарков

`https://github.com/shipilev/jmm-benchmarks/`

- 2x12x2 Xeon E5-2697, 2.70GHz
- OEL 6, JDK 7u40, x86_64
- Измеряем не производительность спеки, а производительность некоторой её реализации

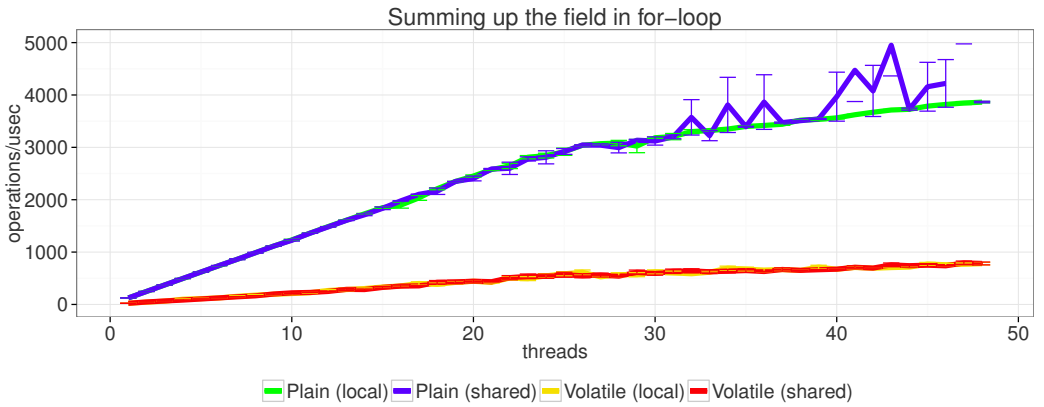
SC-DRF: Hoisting

```
@State(Scope.(Benchmark|Thread))
public static class Storage {
    private (volatile) int v = 42;
}

@GenerateMicroBenchmark
public int test(Storage s) {
    int sum = 0;
    for (int c = 0; c < s.v; c++) {
        sum += s.v;
    }
    return sum;
}
```


SC-DRF: Hoisting

Не так страшен `volatile`, сколько поломанные оптимизации:



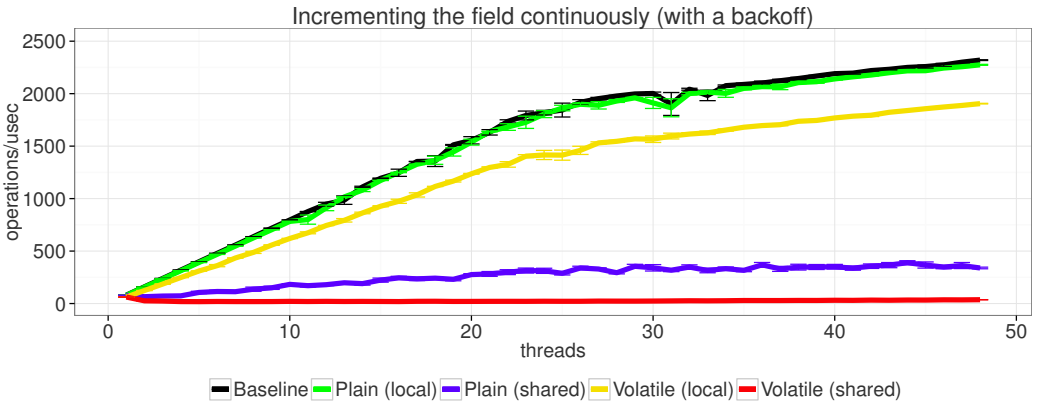
SC-DRF: Writes

```
@State(Scope.(Benchmark|Thread))
public static class Storage {
    private (volatile) int v = 42;
}

@GenerateMicroBenchmark
public int test(Storage s) {
    BlackHole.consumeCPU(8); // ~15ns
    return s.v++;
}
```


SC-DRF: Writes

Не так страшен `volatile`, сколько data sharing:

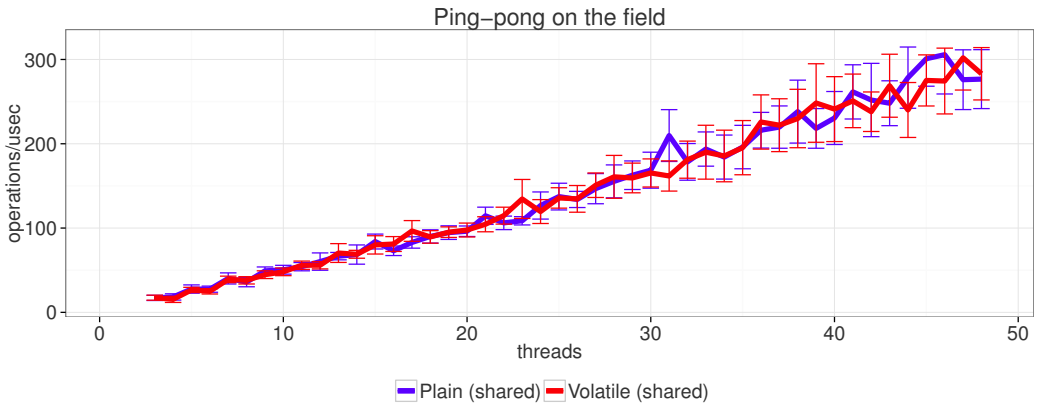


SC-DRF: Ping-pong

```
@State(Scope.(Group|Thread))
public static class Storage {
    private (volatile) int v;
}
@State(Scope.Thread)
public static class Local {
    private int nextV;
}
@GenerateMicroBenchmark
public int test(Storage a, Local l) {
    int nextV = l.nextV;
    while (a.v != nextV);
    l.nextV = nextV + 2;
    return a.v++;
}
```


SC-DRF: Ping-pong

Не так страшен `volatile`, сколько memory latency:



Java ORACLE

SC-DRF: JMM 9

- SC-DRF нынче признаётся наиболее удачной моделью
  - Формально идея существует ещё с 90-х годов
  - Адаптировано в Java в 2004
  - ...и теперь ещё в C/C++ в 2011
- В некоторых местах за SC приходится очень больно платить
  - Пример: PowerPC + IRIW = кровь, кишки, расчленёнка
  - Пример: Linux Kernel RCU = релаксации SC для ARM/PowerPC местами дают конские приросты производительности ...и даже без кажущегося взрыва мозга
- Вопрос: можно ли как-нибудь ослабить это требование, не разрушив всю модель?

ПЕРЕРЫВ

на поесть, попить, покурить, повеситься

SC-DRF: Логика построения модели

Формализм JMM пытается получить ответ на ОВЁС
(и делает это неконструктивно!)

- JMM строит все возможные исполнения программы
  - Действия программы образуют program/synchronization order
  - Из них рождаются synchronizes-with, happens-before order
- JMM отбрасывает запрещенные исполнения
  - Для этого порядки имеют структурные ограничения
  - Потом на исполнения налагается ещё немножко ограничений
  - Потом исполнения ещё чуть-чуть валидируются
- Оставшиеся исполнения – разрешены

SC-DRF: Executions

Execution: $E =< P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} >$

- P – программа; A – множество действий программы
- $\xrightarrow{po}$ – program order; $\xrightarrow{so}$ – synchronization order
- $W(r)$ – «write seen function», возвращает запись, которую видит данное чтение; $V(r)$ – значение, которое прочитало конкретное чтение
- $\xrightarrow{sw}$ – synchronizes-with order
- $\xrightarrow{hb}$ – happens-before order

SC-DRF: Ограничения на PO

- PO знает своё место, через него в модель «протекает» информация об оригинальной программе:

$\forall t \in Threads, \xrightarrow{po}|_t$ – полный порядок

$\xrightarrow{po}$ не связывает действия разных потоков

- **Inter-thread consistency** даёт мостик между поведением последовательной программы и всей остальной моделью:

$\forall t \in Threads, \xrightarrow{po}|_t$ генерирует действия, совместные с P

SC-DRF: Ограничения на HB

- В happens-before order отсутствуют циклы:

$\xrightarrow{hb}$ – частичный порядок

- **Happens-before consistency** обязывает видеть последние записи в $\xrightarrow{hb}$, или разрешает любые другие вне $\xrightarrow{hb}$:

$$\forall r \in Reads(A) : \neg(r \xrightarrow{hb} W(r)) \wedge$$
$$(\neg\exists w \in Writes(A) : (W(r) \xrightarrow{hb} w) \wedge (w \xrightarrow{hb} r))$$

SC-DRF: Ограничения на SO

- Образует *отдельный* порядок:

$\xrightarrow{so}$ – полный порядок, $\xrightarrow{so}$ совместен с $\xrightarrow{po}$

- **Synchronization-order consistency** обязывает чтения видеть последние записи в SO:

$$\forall r \in Reads(A|_{sync}) : (W(r) \xrightarrow{so} r) \wedge$$
$$(\neg \exists w \in Writes(A|_{sync}) : (W(r) \xrightarrow{so} w) \wedge (w \xrightarrow{so} r))$$

SC-DRF: Ограничения SO, пример

Есть классический тест, IRIW:

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

`x = 1;`	`y = 1;`	`int r1 = y;` `int r2 = x;`	`int r3 = x;` `int r4 = y;`

- Результаты этой программы могут быть описаны только исполнениями, в которых SO – полный
- Все исполнения, приводящие к $(r1, r2, r3, r4) = (1, 0, 1, 0)$ имеют поломанный SO – запрещены!

OoTA

OoTA: Сказка

«SC-DRF. Построй свою любовь»

- Локальные трансформации разрешены, пока не встретим синхронизацию
- Отношения локальных трансформаций и синхронизаций тоже определены (в самом простом случае, «roach motel»)
- Если локальные трансформации переколбашивают конфликтные доступы, значит, там и так была гонка, и девелопер ССЗБ

OoTA: Реальность

Но есть случаи, когда локальные трансформации ломают SC:

```
int a = 0, b = 0;
```

`r1 = a;`	`r2 = b;`
`if (r1 != 0)`	`if (r2 != 0)`
`  b = 42;`	`  a = 42;`

Корректно синхронизована:
все SC исполнения не содержат гонок.
Возможно только `(r1, r2) = (0, 0)`.

OoTA: Оптимизации

Немножко спекулятивных оптимизаций:
почему бы не записать в b, и откатить, «если что»?
(Покажем на компиляторах, хотя может и хардвар спекулировать)

```
int a = 0, b = 0;
```

```
int r1 = a;
if (r1 != 0)
   b = 42;
```

OoTA: Оптимизации

Немножко спекулятивных оптимизаций:
почему бы не записать в b, и откатить, «если что»?
(Покажем на компиляторах, хотя может и хардвар спекулировать)

```
int a = 0, b = 0;
```

```
int r1 = a;
if (r1 != 0)
   b = 42;
```

→

```
int r1 = a;
b = 42;
if (r1 == 0)
  b = 0;
```


OoTA: Оптимизации

Немножко спекулятивных оптимизаций:
почему бы не записать в b, и откатить, «если что»?
(Покажем на компиляторах, хотя может и хардвар спекулировать)

```
int a = 0, b = 0;
```

```
int r1 = a;
if (r1 != 0)
   b = 42;
```

→

```
int r1 = a;
b = 42;
if (r1 == 0)
  b = 0;
```

→

```
b = 42;
int r1 = a;
if (r1 == 0)
  b = 0;
```


OoTA: Как ныне сбирается вещий Эдип...

```
int a = 0, b = 0;
```

`b = 42;`	
	`r2 = b;`
	`if (r2 != 0)`
	`  a = 42;`
`r1 = a;`	
`if (r1 == 0)`	
`  b = 0;`	

- Приводит к `(r1, r2) = (42, 42)`
- В присутствии гонок спекуляция превращается в самоподтверждающееся пророчество!

OoTA: Логика построения модели

Формализм JMM пытается получить ответ на ОВЁС
(и делает это неконструктивно!)

- JMM строит все возможные исполнения программы
  - Действия программы образуют program/synchronization order
  - Из них рождаются synchronizes-with, happens-before order
- JMM отбрасывает запрещенные исполнения
  - Для этого порядки имеют структурные ограничения
  - Потом на исполнения налагается ещё немножко ограничений
  - Потом исполнения ещё чуть-чуть валидируются
- Оставшиеся исполнения – разрешены

OoTA: Causality loops

JLS TL;DR: OoTA значения запрещены.

- Если мы прочитали значение, то это значит, что кто-то его **до нас** записал
- Самая сложная часть спецификации: весь формализм в JLS 17.4.8 построен для того, чтобы дать определение этому «до нас» = «causality requirements»
- JMM определяет специальный процесс валидации исполнений через «commit»-ы действий

OoTA: Commit semantics

17.4.8 Executions and Causality Requirements

We use $f|_d$ to denote the function given by restricting the domain of f to d. For all x in d, $f|_d(x) = f(x)$, and for all x not in d, $f|_d(x)$ is undefined.

We use $p|_d$ to represent the restriction of the partial order p to the elements in d. For all x,y in d, $p(x,y)$ if and only if $p|_d(x,y)$. If either x or y are not in d, then it is not the case that $p|_d(x,y)$.

A well-formed execution $E = < P, A, po, so, W, V, sw, hb >$ is validated by *committing* actions from A. If all of the actions in A can be committed, then the execution satisfies the causality requirements of the Java programming language memory model.

Starting with the empty set as C_0, we perform a sequence of steps where we take actions from the set of actions A and add them to a set of committed actions C_i to get a new set of committed actions C_{i+1}. To demonstrate that this is reasonable, for each C_i we need to demonstrate an execution E containing C_i that meets certain conditions.

Formally, an execution *E satisfies the causality requirements of the Java programming language memory model* if and only if there exist:

- Sets of actions C_0, C_1, ... such that:
  - C_0 is the empty set
  - C_i is a proper subset of C_{i+1}
  - $A = \cup (C_0, C_1, ...)$

  If A is finite, then the sequence C_0, C_1, ... will be finite, ending in a set $C_n = A$.

  If A is infinite, then the sequence C_0, C_1, ... may be infinite, and it must be the case that the union of all elements of this infinite sequence is equal to A.
- Well-formed executions E_1, ..., where $E_i = < P, A_i, po_i, so_i, W_i, V_i, sw_i, hb_i >$.

Given these sets of actions C_0, ... and executions E_1, ... , every action in C_i must be one of the actions in E_i. All actions in C_i must share the same relative happens-before order and synchronization order in both E_i and E. Formally:

1. C_i is a subset of A_i
2. $hb_i|_{C_i} = hb|_{C_i}$
3. $so_i|_{C_i} = so|_{C_i}$

The values written by the writes in C_i must be the same in both E_i and E. Only the reads in C_{i-1} need to see the same writes in E_i as in E. Formally:

4. $V_i|_{C_i} = V|_{C_i}$
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$

All reads in E_i that are not in C_{i-1} must see writes that happen-before them. Each read r in C_i - C_{i-1} must see writes in C_{i-1} in both E_i and E, but may see a different write in E_i from the one it sees in E. Formally:

6. For any read r in A_i - C_{i-1}, we have $hb_i(W_i(r), r)$
7. For any read r in (C_i - C_{i-1}), we have $W_i(r)$ in C_{i-1} and $W(r)$ in C_{i-1}

Given a set of sufficient synchronizes-with edges for E_i, if there is a release-acquire pair that happens-before (§17.4.5) an action you are committing, then that pair must be present in all E_j, where $j \geq i$. Formally:

8. Let ssw_i be the sw_i edges that are also in the transitive reduction of hb_i but not in po. We call ssw_i the *sufficient synchronizes-with edges for* E_i. If $ssw_i(x, y)$ and $hb_i(y, z)$ and z in C_i, then $sw_j(x, y)$ for all $j \geq i$.

   If an action y is committed, all external actions that happen-before y are also committed.
9. If y is in C_i, x is an external action and $hb_i(x, y)$, then x in C_i.

OoTA: Commit semantics

17.4.8 Executions and Causality Requirements

We use $f|_d$ to denote the function given by restricting the domain of f to d. For all x in d, $f|_d(x) = f(x)$, and for all x not in d, $f|_d(x)$ is undefined.

We use $p|_d$ to represent the restriction of the partial order p to the elements in d. For all x,y in d, $p(x,y)$ if and only if $p|_d(x,y)$. If either x or y are not in d, then it is not the case that $p|_d(x,y)$.

A well-formed execution $E = < P, A, po, so, W, V, sw, hb >$ is validated by *committing* actions from A. If all of the actions in A can be committed, then the execution satisfies the causality requirements of the Java programming language memory model.

Starting with the empty set as C_0, we perform a sequence of steps where we take actions from the set of actions A and add them to a set of committed actions C_i to get a new set of committed actions C_{i+1}. To demonstrate that this is reasonable, for each C_i we need to demonstrate an execution E containing C_i that meets certain conditions.

Formally, an execution *E satisfies* [illegible] *ents of the Java programming language memory mod*

- Sets of actions C_0, C_1, ... such tha
  - C_0 is the empty set
  - C_i is a proper subset of C_{i+1}
  - $A = \cup (C_0, C_1, ...)$

  If A is finite, then the seq [illegible] et $C_n = A$.

  If A is infi [illegible] ust be the [illegible] ual to A.

  [illegible] V_i, sw_i, hb_i >.

Given these sets of actions C_0, ... and executions E_1, ... , every action in C_i must be one of the actions in E_i. All actions in C_i must share the same relative happens-before order and synchronization order in both E_i and E. Formally:

1. C_i is a subset of A_i
2. $hb_i|_{C_i} = hb|_{C_i}$
3. $so_i|_{C_i} = so|_{C_i}$

The values written by the writes in C_i must be the same in both E_i and E. Only the reads in C_{i-1} need to see the same writes in E_i as in E. Formally:

4. $V_i|_{C_i} = V|_{C_i}$
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$

All reads in E_i that are not in C_{i-1} must see writes that happen-before them. Each read r in C_i - C_{i-1} must see writes in C_{i-1} in both E_i and E, but may see a different write in E_i from the one it sees in E. Formally:

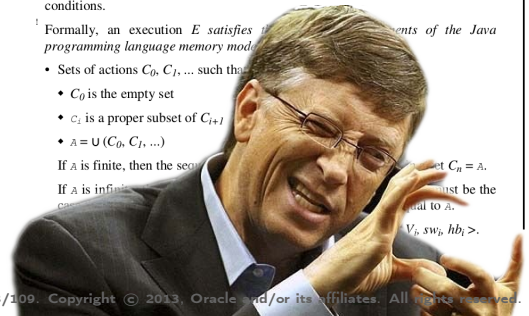
6. For any read r in A_i - C_{i-1}, we have $hb_i(W_i(r), r)$
7. For any read r in (C_i - C_{i-1}), we have $W_i(r)$ in C_{i-1} and $W(r)$ in C_{i-1}

Given a set of sufficient synchronizes-with edges for E_i, if there is a release-acquire pair that happens-before (§17.4.5) an action you are committing, then that pair must be present in all E_j, where $j \geq i$. Formally:

8. Let ssw_i be the sw_i edges that are also in the transitive reduction of hb_i but not in po. We call ssw_i the *sufficient synchronizes-with edges for* E_i. If $ssw_i(x, y)$ and $hb_i(y, z)$ and z in C_i, then $sw_j(x, y)$ for all $j \geq i$.

   If an action y is committed, all external actions that happen-before y are also committed.

9. If y is in C_i, x is an external action and $hb_i(x, y)$, then x in C_i



Java ORACLE

OoTA: C/C++11

Спецификация OoTA настолько сложна, что C/C++11 сдался:
C/C++11 не специфицировала эту часть своей модели

- Избежала всего геморроя со спецификацией (ага, Пиррова победа)
- Особенно в присутствии relaxed atomics, которые нам нужны, потому что мы же любим низкоуровневые оптимизации, да?
- Эксперты в C/C++1x WG чешут темечко, как (в теории) запретить компиляторам спекулятивно рождать значения

OoTA: JMM 9

Пути решения проблемы OoTA:

1. Продолжать в том же духе: предпринять попытку до/перепилить формализм, чтобы он был понимабелен/верифицируем и исправить в нём ошибки
2. Консервативно запретить спекулятивные записи: это будет означать `LoadStore` перед каждой записью
3. Сдаться, и надеяться на вменяемость реализаций

Finals

Finals: Quiz

Что напечатает?

```
class A {
    int f;
    A() { f = 42; }
}

A a;
```

`a = new A();`	`if (a != null)` `  println(a.f);`

Finals: Quiz

Что напечатает?

```
class A {
    int f;
    A() { f = 42; }
}

A a;
```

`a = new A();`	`if (a != null)` `  println(a.f);`

<ничего>, 0, 42, или бросит NPE.

Finals: Quiz

«Правильно» не бросит NPE:

```
class A {
   int f;
   A() { f = 42; }
}

A a;
```

`a = new A();`	`A ta = a;`
	`if (ta != null)`
	`  println(ta.f);`

Finals: Сказка

Хотелось бы получить только 42:

```
class A {
   ????? int f;
   A() { f = 42; }
}

A a;
```

`a = new A();`	`A ta = a;`
	`if (ta != null)`
	`  println(ta.f);`

Finals: Сказка

Хочется иметь объекты,
которые можно безопасно публиковать через гонки

- ...чтобы не платить за «лишние» synchronization actions
- ...чтобы не нарушить security, если какой-нибудь (злонамеренный) дурак наш защищённый объект опубликовал через гонку

Finals: Быль

Спекуляция приводит к дефолтам и кризисам.
Но совсем сажать за спекуляцию нельзя,
ибо она приносит профит.

- Способ точечно сломать спекуляцию: никогда не показывать неполные объекты: тогда неоткуда ни рантайму, ни железу, ни чёрту в ступе взять левое значение
- ...естественный способ поддержать в языке – новые объекты, ибо про них ничего не известно
- ...включать магическим словом `final`: если пользователи хотят иного, не ставят `final`

Finals: Решение

В конце конструктора происходит freeze action

Freeze action «замораживает» поля

- Если поток прочитал ссылку на объект, то он увидит замороженные значения
- Если поток прочитал из `final`-поля ссылку на другой объект, то состояние того как минимум настолько же свежее, как и на время freeze'а

Finals: Формально

$$\boxed{w \xrightarrow{hb} F \xrightarrow{hb} a \xrightarrow{mc} r},$$

где: w – запись поля, F – freeze action, a – действие, прочитавшее ссылку на объект, r – чтение поля

- Вводятся два новых частичных порядка: *dereference order (dr)* и *memory order (mc)*. Интуитивно: цепочки доступа к конкретным полям через ссылки
- Если единственный путь через dr и mc до записи поля лежит через F, то можем увидеть только замороженное значение, ура! А если есть другие пути...

Finals: Засады

Гарантии на замороженность
пропадают при преждевременной публикации:

T p, q;

T t1 = <new>	T t2 = p	T t4 = q
t1.f = 42	r2 = t2.f	r4 = t4.f
p = t1	T t3 = q	
<freeze t1.f>	r3 = t3.f	
q = t1		

$r4 \in \{42\}$;
но r2, r3 $\in \{0, 42\}$, ибо p «утёк»

Finals: Прагматика

Дать оптимизациям мега-свободу по кешированию `final`-ов

- «All references are created equal»: локальный оптимизатор не перегружен анализом ссылок
- Как только оптимизатор увидел опубликованную ссылку, он может скэшировать все его `final` поля, и баста!
- Т.е. если оптимизатор увидел недоконструированный объект, то хана

Finals: Реализация

Довольно просто реализовать на большинстве платформ

- Достаточно запретить переупорядочивание инициализации `final`-полей и публикации объекта на большинстве архитектур
- Все известные промышленные архитектуры[10] не переупорядочивают загрузку объекта, и загрузку поля из него (dependent loads)

[10] кроме Alpha, но она отправилась к праотцам, куда ей и дорога

Finals: Quiz

Что напечатает?

```
class A {
   final int f;
   { f = 42; }
}
A a;
```

`a = new A();`	`A ta = a;`
	`if (ta != null)`
	`  println(ta.f);`

Finals: Quiz

Что напечатает?

```
class A {
   final int f;
   { f = 42; }
}
A a;
```

`a = new A();`	`A ta = a;`
	`if (ta != null)`
	`  println(ta.f);`

Конечно, 42.

Finals: JMM 9

- Текущая спецификация плохо относится к не-`final`-полям
- Если поле записано в конструкторе, и никогда не модифицируется? (e.g. юзер прошляпил `final` на поле)
- Если поле уже `volatile`? (e.g. `AtomicInteger`)
- Если объект строится билдерами? (примеров не надо)

- Вопрос: не стоит ли дать гарантии на инициализацию для **всех** полей и **всех** конструкторов?

Finals: JMM 9

https://github.com/shipilev/jmm-benchmarks/

- 2x12x2 Xeon E5-2697, 2.70GHz; OEL 6, JDK 8b121, x86_64
- 1x4x1 Cortex-A9, 1.7 GHz; Linaro 12.11, JDK 8b121, SE Embedded
- Измеряем не производительность спеки, а производительность некоторой её реализации

Finals: JMM 9

https://github.com/shipilev/jmm-benchmarks/

- 2x12x2 Xeon E5-2697, 2.70GHz; OEL 6, JDK 8b121, x86_64
- 1x4x1 Cortex-A9, 1.7 GHz; Linaro 12.11, JDK 8b121, SE Embedded
- Измеряем не производительность спеки, а производительность некоторой её реализации

Finals: JMM 9: Initialization (chained)

```
@GenerateMicroBenchmark
public Object test() {
   return new Test_[N](v);
}

// chained case
class Test_[N] extends Test_[N-1] {
   private [plain|final] int i_[N];
   public <init>(int v) {
      super(v);
      i_[N] = v;
   }
}
```

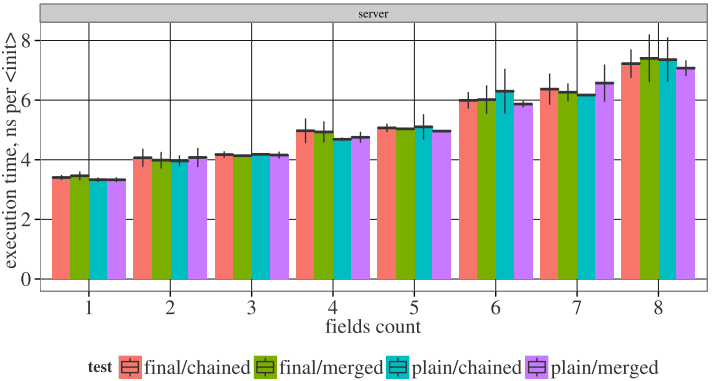

Finals: JMM 9: Initialization (merged)

```
@GenerateMicroBenchmark
public Object test() {
   return new Test_[N](v);
}

// merged case
class Test_[N] {
   private [plain|final] int i_1, ..., i_[N];
   public <init>(int v) {
      i_1 = i_2 = ... = i_[N] = v;
   }
}
```


Finals: JMM 9: Результаты (x86)

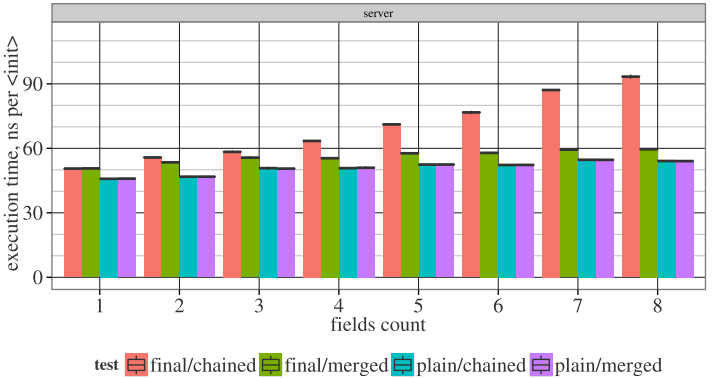
На Total Store Order это вообще бесплатно:[11]



[11] http://shipilev.net/blog/2014/all-fields-are-final/

Finals: JMM 9: Результаты (ARMv7)

На weakly-ordered архитектурах нужно клеить барьеры:[12]



[12] http://shipilev.net/blog/2014/all-fields-are-final/

Заключение

Заключение: Lingua Latina...

«The best way is to build up a small repertoire of constructions that you know the answers for and then never think about the JMM rules again unless you are forced to do so! Literally nobody likes figuring things out from the JMM rules as stated, or can even routinely do so correctly. This is one of the many reasons we need to overhaul JMM someday.»

Заключение: Lingua Latina...

«The best way is to build up a small repertoire of constructions that you know the answers for and then never think about the JMM rules again unless you are forced to do so! Literally nobody likes figuring things out from the JMM rules as stated, or can even routinely do so correctly. This is one of the many reasons we need to overhaul JMM someday.»

(Doug Lea, private communication, 2013)

Заключение: Известные проблемы

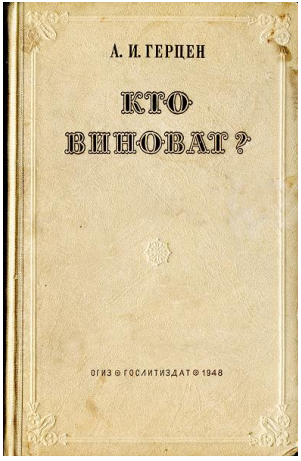
- JSR 133 Cookbook не содержит некоторых машинно-специфичных особенностей, обнаруженных много позднее создания текущей JMM
- Некоторые новые примитивы в библиотеке вообще не специфицируемы в текущей модели (e.g. `lazySet`, `weakCompareAndSet`)
- JMM специфицирована для Java, что делать с JVM-based языками?
- В формальной спецификации JMM есть формальные неточности, которые ставят раком автоматические верификаторы

Заключение: JMM Overhaul

«Java Memory Model update»
`http://openjdk.java.net/jeps/188`

- Improved formalization
- JVM languages coverage
- Extended scope for existing unspec-ed primitives
- C11/C++11 compatibility
- Testing support
- Tool support

Заключение: Чтение



Заключение: Чтение

- Goetz et al, «Java Concurrency in Practice»
- Herilhy, Shavit, «The Art of Multiprocessor Programming»
- Adve, «Shared Memory Models: A Tutorial»
- McKenney,«Memory Barriers: a Hardware View for Software Hackers»
- Manson, «Java Memory Model» (Special PoPL issue)
- Huisman, Petri, «JMM: The Formal Explanation»

