Cloud Native JVM

Cloud Native Compiler

Владимир Воскресенский

Azul Systems

Distinguished Engineer

vladimir@azul.com

azul

Облака очень популярны

azul

Cloud Native

azul

Рантаймы очень популярны

azul

(Managed Language) Рантаймы

azul

?

?

?

?

Scala

Ruby

Java

Go

F#

?

?

Python

Kotlin

Clojure

C#

JavaScript

PHP

Erlang

Objective-C

Swift

?

C

C++

?

?

?

?

azul

Kotlin Java

Scala

Clojure

C# F#

Go

PHP

Python

Erlang

JavaScript

Objective-C Swift

Ruby

azul

Почему?

azul

Продуктивность?

Надежность?

Много экспертов?

Стабильность?

Почему?

Скорость разработки?

Крупные Экосистемы?

Производительность?

azul

Почему?

Таков текущий выбор...

azul

Cloud Native JVM

Что это?

azul

JVM (раньше & сейчас)

- Изолирована от других JVM
- Нет “памяти” о прошлых/других JVM запусках
- Полностью полагается лишь на себя
  - Ограничена локальными ресурсами
  - Ограничена своим функционалом
  - Должна выбирать на что тратить ресурсы

azul

JVM (раньше & сейчас)

- В отсутствии “магического облака”
  - Ограниченные вычислительные ресурсы
  - Ограниченное место для хранения данных
  - Ограничена функционально
  - Ограничена умением анализировать и обучаться
  - Ограниченные “знания” об устройстве мира

azul

Cloud Native JVM

azul

Cloud Native JVM могла бы...

- Присоединяться к сообществу других JVM
- Использовать и расчитывать на:
  - Внешние ресурсы
  - Внешнюю функциональность
  - Внешний опыт
- Создавать новые (и пополнять старые) знания

azul

Cloud Native JVM могла бы...

- Имея доступ к “магическому облаку” получить
  - “безграничные” вычислительные ресурсы
  - “безграничные” размеры хранилища данных
  - “безграничные” аналитические возможности
  - “знания”, “опыт”, ...

azul

Cloud Native JVM (опредеделение)

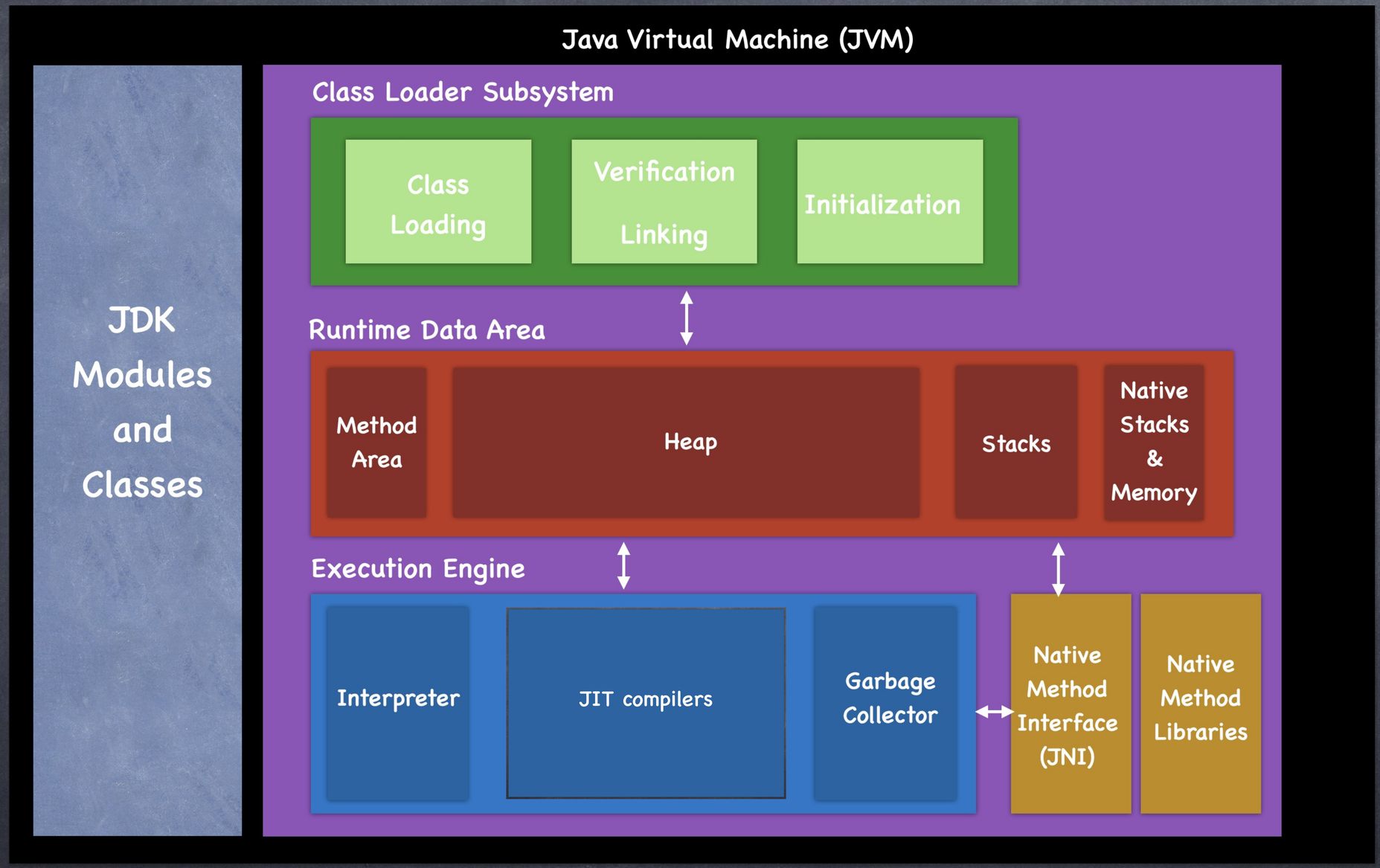
- Обладает всеми свойствами JVM
- Делегирует (аутсорсит) некоторые ключевые функции в облако
- Сообщает в облако опыт и знания, которые могут быть использованы другими JVM, присоединенными к этому облаку

azul

Устройство JVM

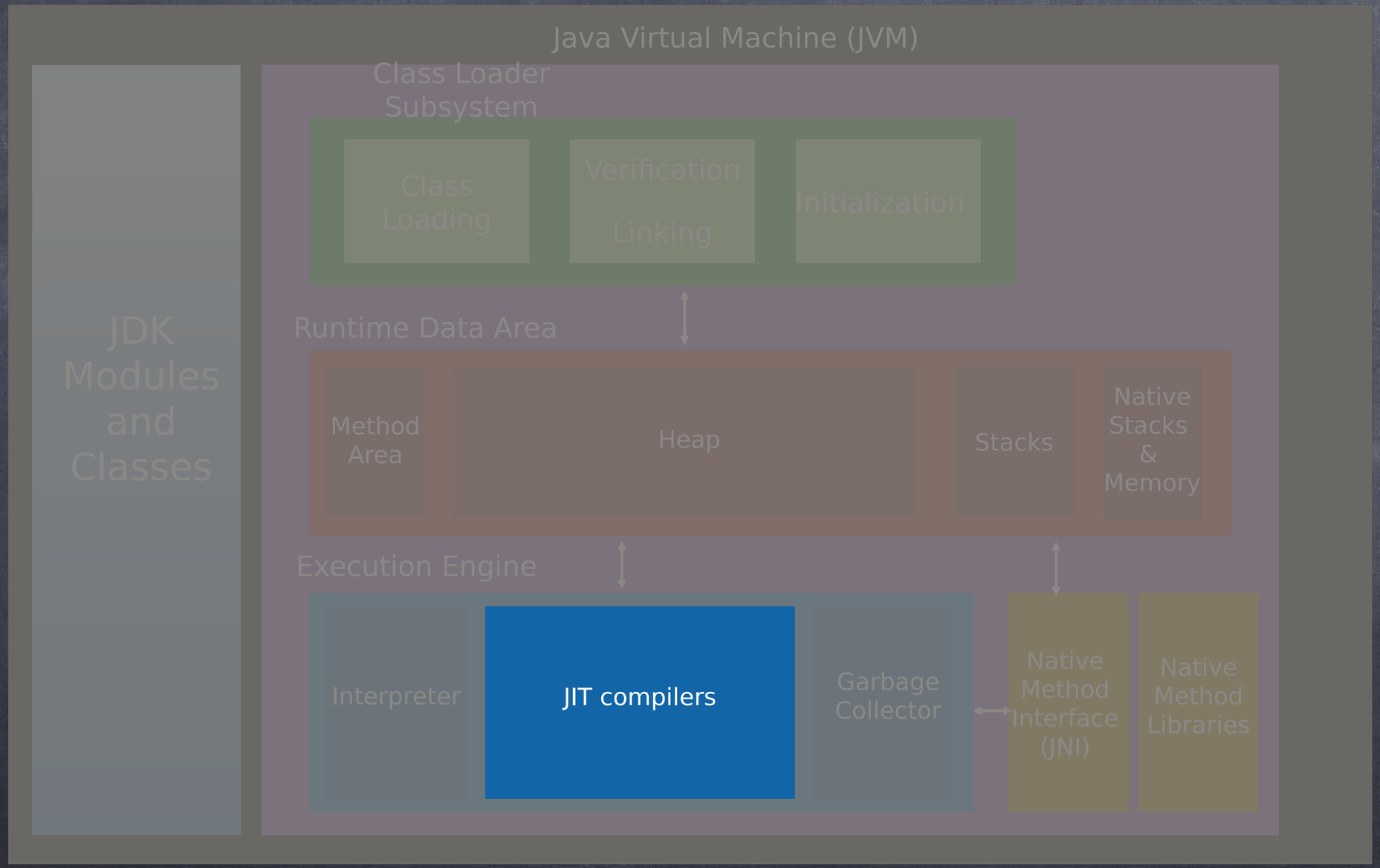
azul

Устройство JVM



azul

Устройство JVM: JIT Компиляторы



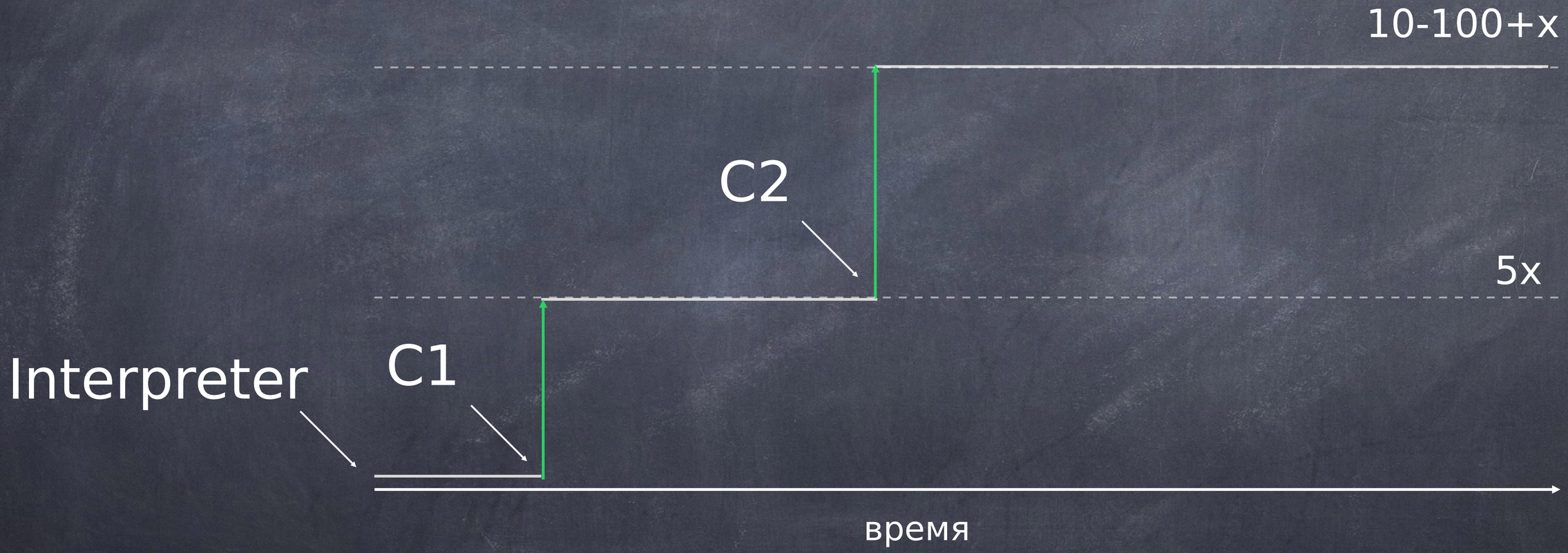
azul

JIT Компиляторы

Стараются перевести байткод в оптимальное машинное представление

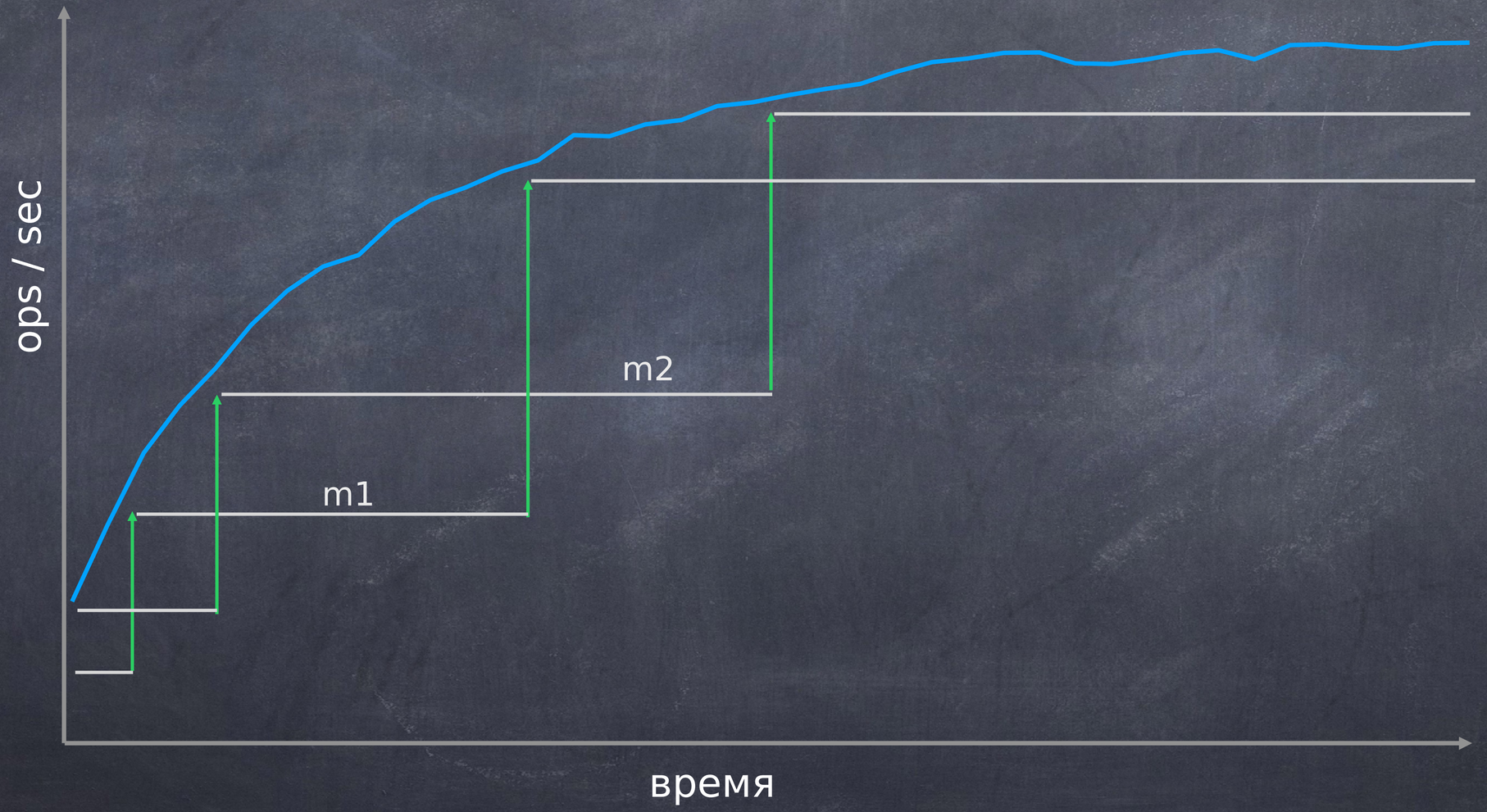
azul

Ментальная модель жизни одного метода



azul

Ментальная модель скорости приложения



azul

Внутренности JVM:
общение с JIT компилятором

- JVM сообщает: “этот код я бы хотела исполнять быстрее”
-

azul

Внутренности JVM:
общение с JIT компилятором

- JVM сообщает: “этот код я бы хотела исполнять быстрее”
- JIT Компилятор задаёт вопросы: “а что тебе известно про X/Y?”
-

azul

Внутренности JVM:
общение с JIT компилятором

- JVM сообщает: “этот код я бы хотела исполнять быстрее”
- JIT Компилятор задаёт вопросы: “а что тебе известно про X/Y?”
- JIT Компилятор заканчивает: “вот код для твоего случая”
-

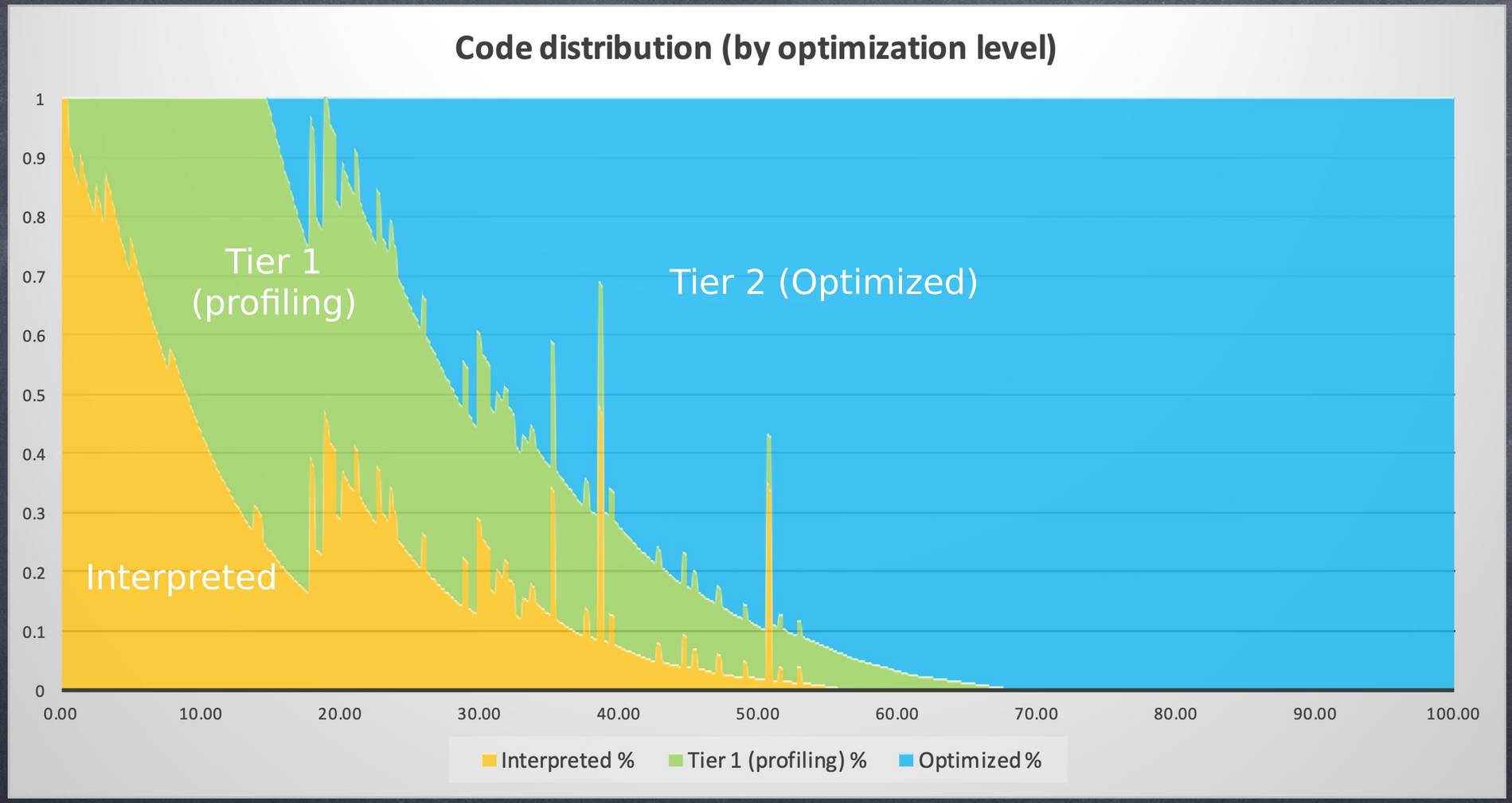
azul

Внутренности JVM: общение с JIT компилятором

- JVM сообщает: “этот код я бы хотела исполнять быстрее”
- JIT Компилятор задаёт вопросы: “а что тебе известно про X/Y?”
- JIT Компилятор заканчивает: “вот код для твоего случая”
- JVM заменяет существующий код метода на новый, в надежде, что он стал быстрее

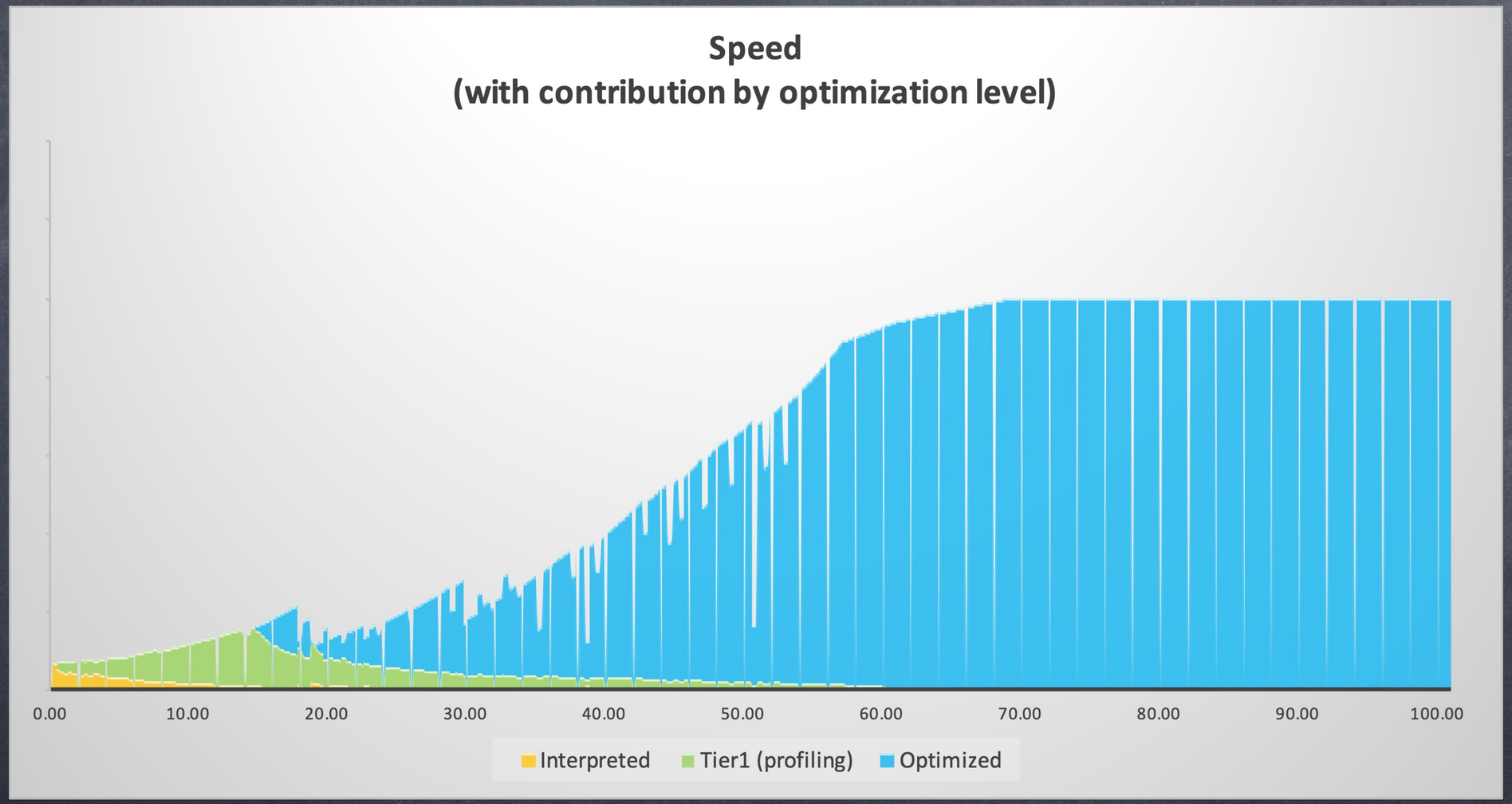
azul

Внутренности JVM: Распределение кода



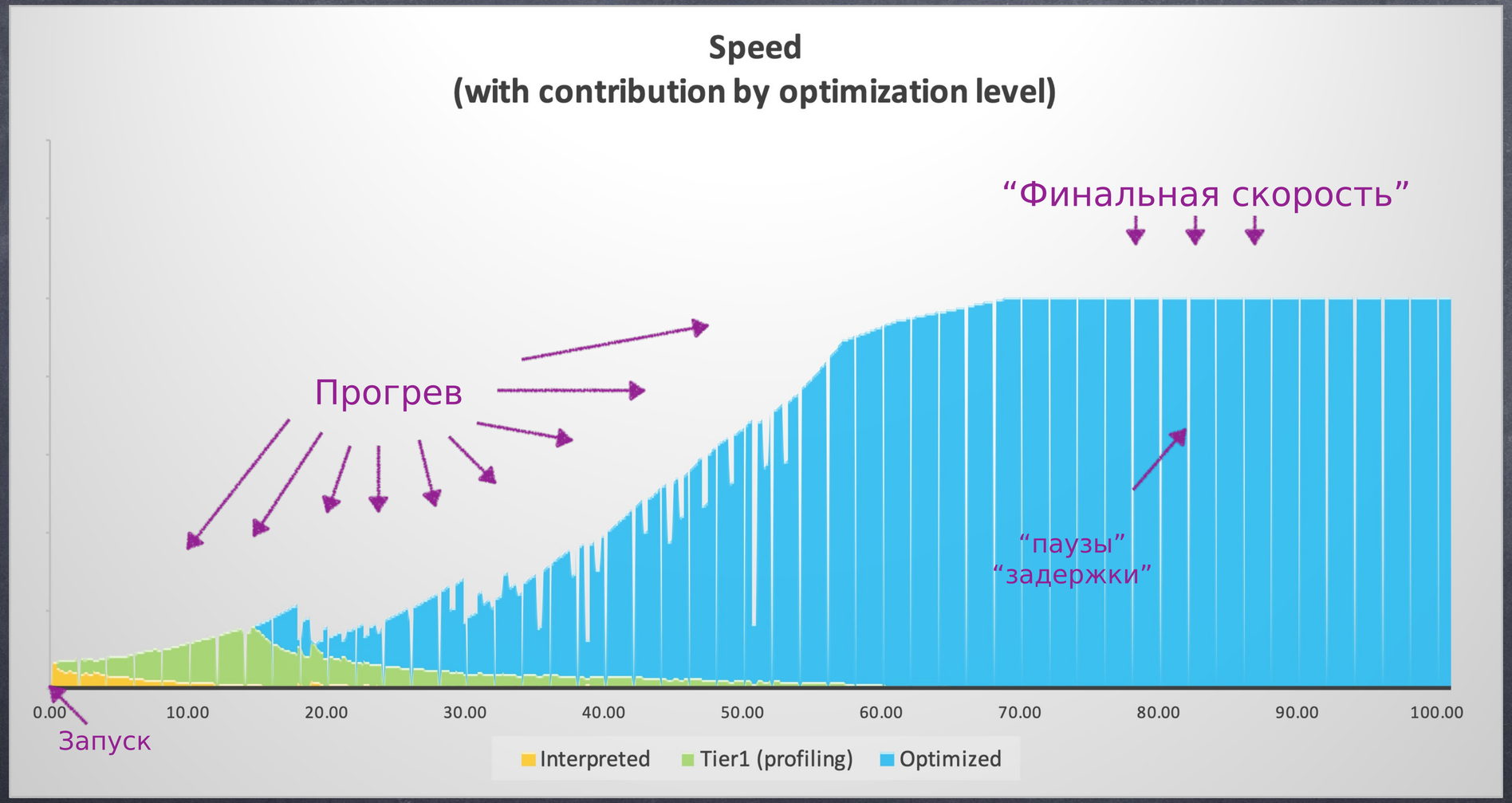
azul

Внутренности JVM: Влияние на скорость



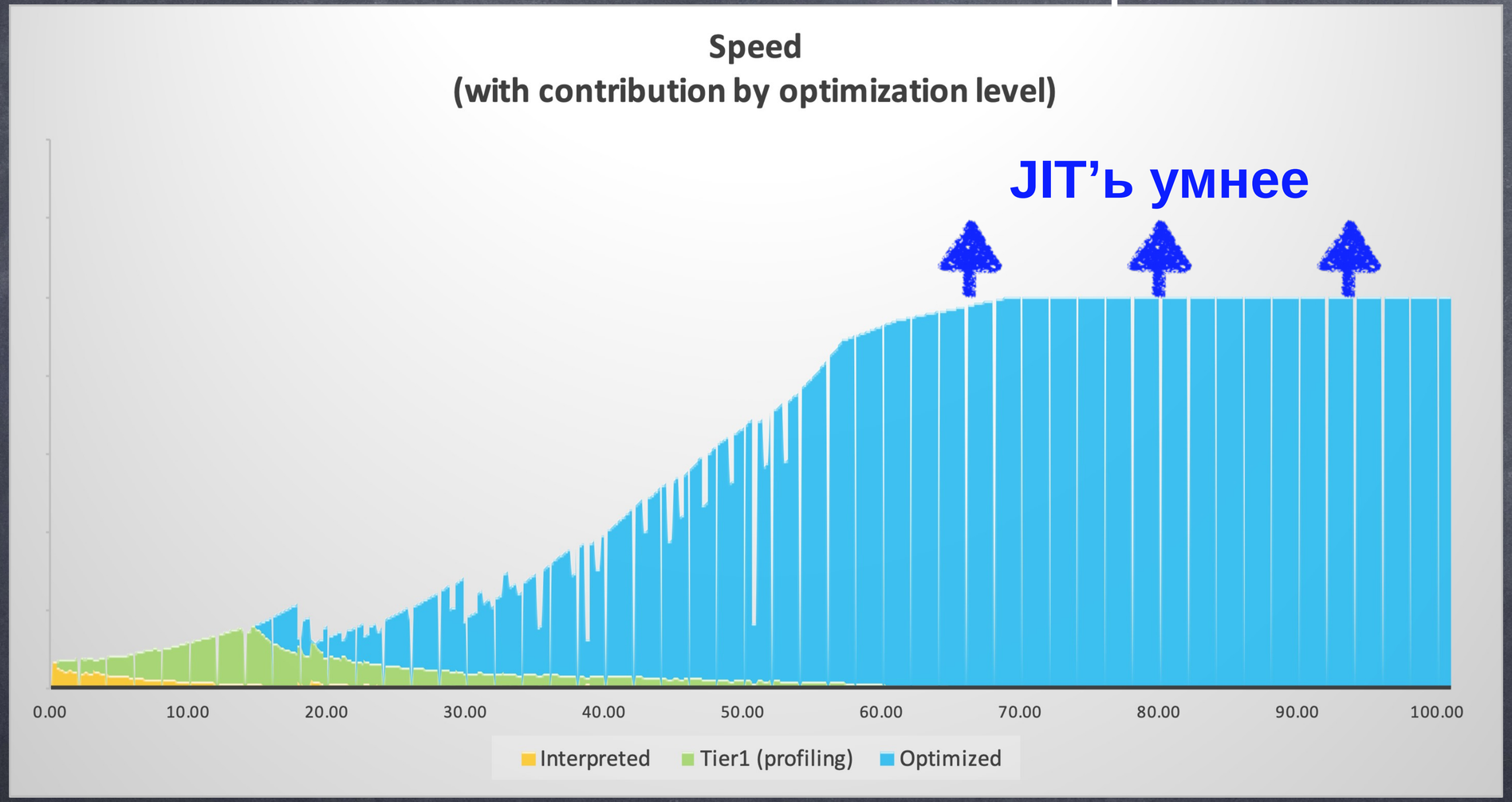
azul

Внутренности JVM: Фазы работы приложения



azul

JIT компилятор:
А можно ли исполнять быстрее?



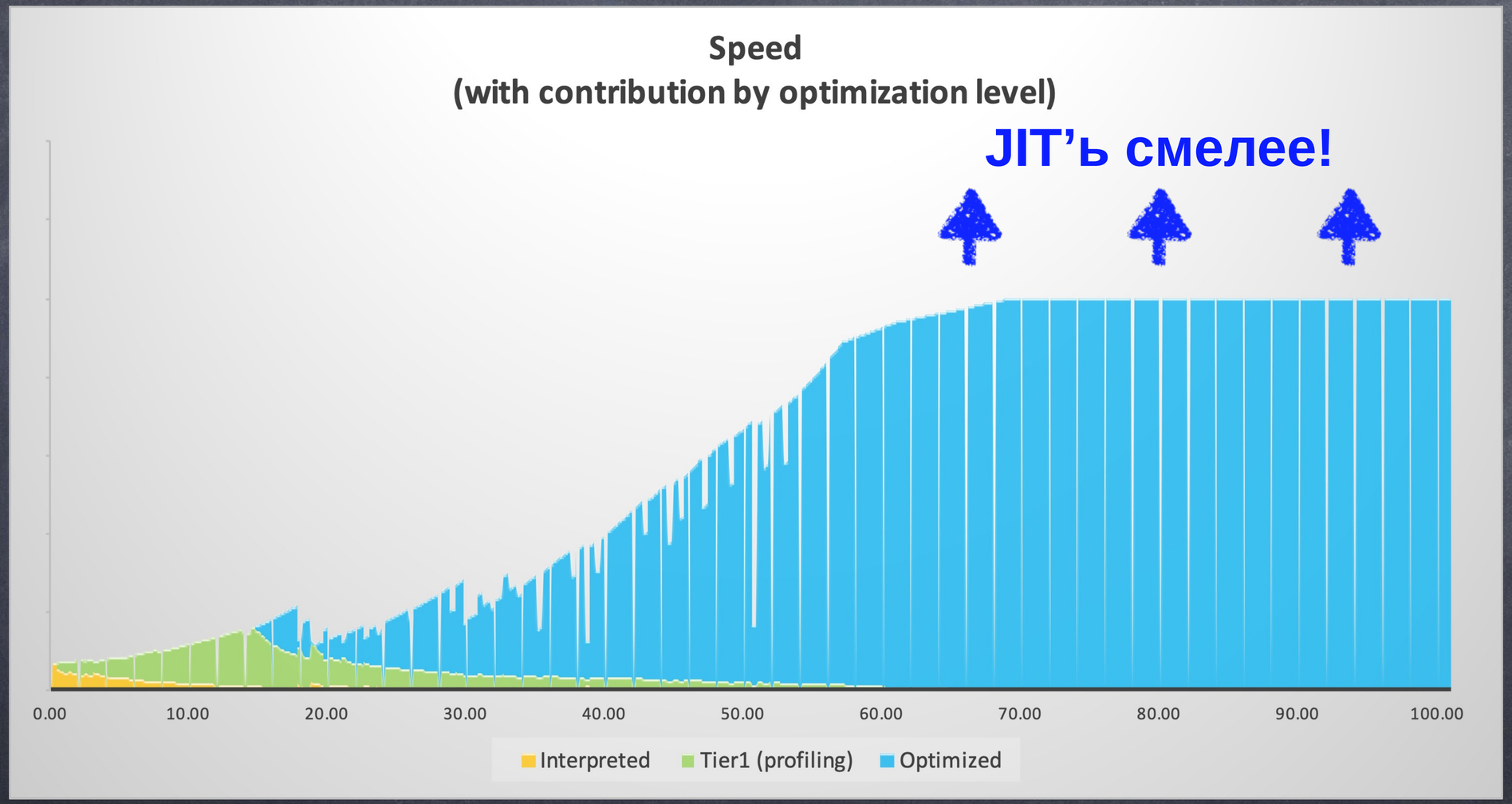
azul

“Умные” Оптимизации

Инлайнинг
Escape анализ
Векторизация
....

azul

JIT компилятор:
А можно ли исполнять еще быстрее?



azul

“Героические” (спекулятивные) оптимизации

“секрет” скорости Java

azul

....
Class Hierarchy Analysis (CHA)
Truly Final fields
Effectively Final fields
Deopt on Throw
....

....
Unreached code
Implicit Null checks
Range checks
Profile Guided Devirtualization
....

azul

“Героические” (спекулятивные) оптимизации

Unguarded спекуляции

azul

Пример: (спекулятивный) Инлайн мономорфик-вызовов

```
public class Animal {
    private int color;
    public int getColor() { return color };
}
...
myColor = animal.getColor();
```

Можно оптимизировать как:

```
...
myColor = animal.color;
```

Корректно до тех пор, пока есть единственная реализация метода getColor()

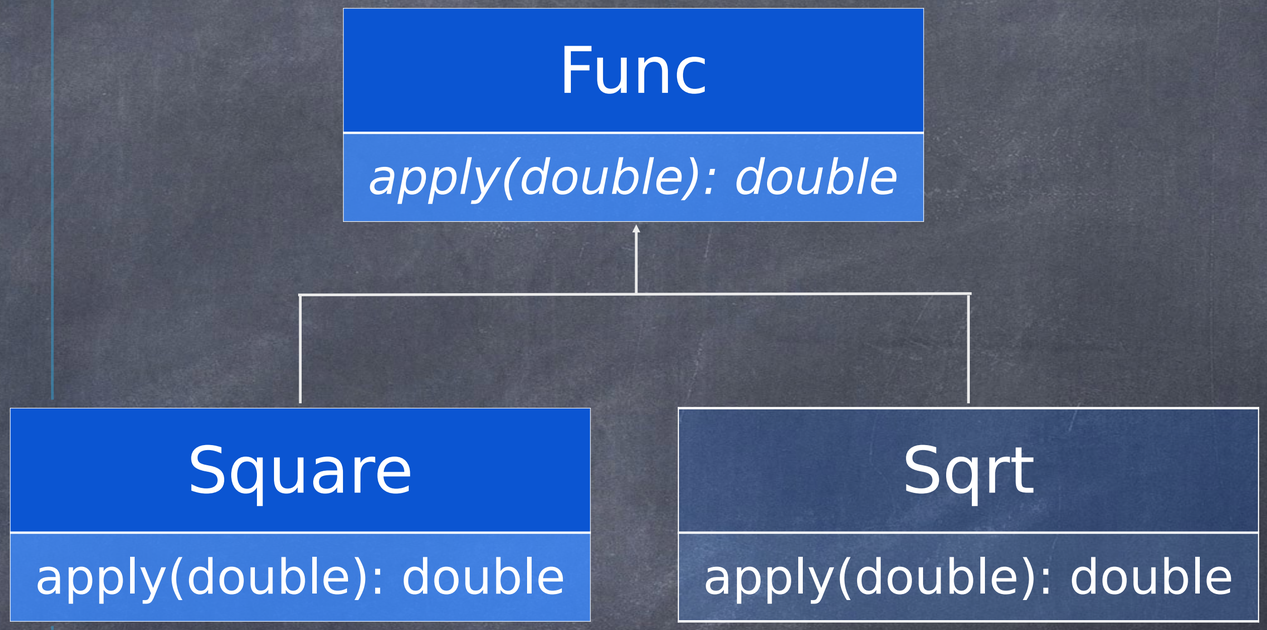
azul

Пример: CHA спекуляции

```
System.out.println("Using Square...");
 Func func = new Square();
 for ( int i = 0; i < 20_000; ++i ) {
      apply1(func, i);
      apply2(func, i);
      …
 }
 Thread.sleep(5_000);

 System.out.printf(
       “Loading %s to Deoptimize Now!%n”,
       Sqrt.class);
 Thread.sleep(25_000);
 …

 double apply1(Func func, double x) {
      return func.apply(x);
 }
 ….
```



azul

Пример: CHA спекуляции

```
double apply1(Func func, double x) {
      return func.apply(x);
}
```

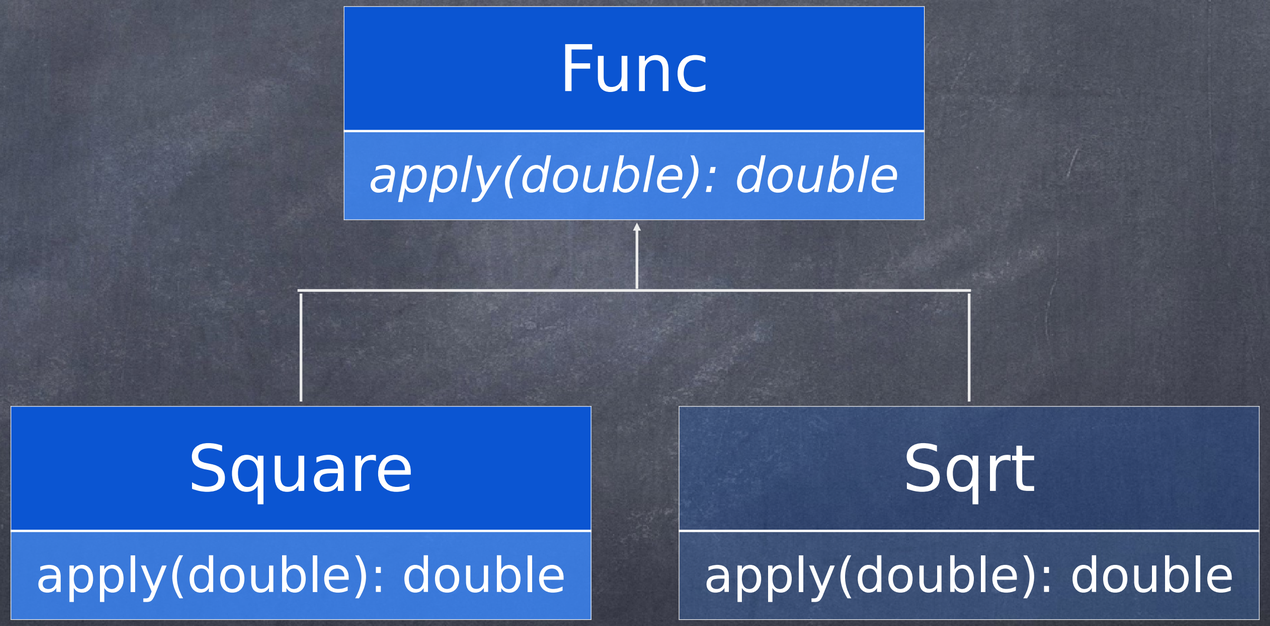
```
return func.apply(x);
```

Девиртуализация

```
// нет проверки типов
return Square::apply(x);
```

Инлайнинг

```
// нет проверки типов
return x * x
```



azul

-XX:+PrintCompilation -XX:+PrintSafepointStatistics -XX:+PrintSafepointStatisticsCount=1

```
 85   29   4   ...ClassDevirtualization::apply1 (7 bytes)
 85   30   4   ...ClassDevirtualization::apply2 (7 bytes)
 ...
 87   36   4   ...ClassDevirtualization::apply8 (7 bytes)
 87   35   4   ...ClassDevirtualization::apply7 (7 bytes)

5091   35   4   ...ClassDevirtualization::apply7 (7 bytes)   made not entrant
5091   36   4   ...ClassDevirtualization::apply8 (7 bytes)   made not entrant
5091   31   4   ...ClassDevirtualization::apply3 (7 bytes)   made not entrant
  ...
5091   33   4   ...ClassDevirtualization::apply5 (7 bytes)   made not entrant
5091   29   4   ...ClassDevirtualization::apply1 (7 bytes)   made not entrant
5091   32   4   ...ClassDevirtualization::apply4 (7 bytes)   made not entrant

vmop  [threads: total initially_running wait_to_block]  [time: spin block sync cleanup vmop]
page_trap_count
5.099: Deoptimize  [ 9  0  0 ] [ 0 0 0 0 0 ] 0
```


azul

Код

Assumptions

Optimized Code

Address	Code	Opcode
0x3000fe50	pushq %rax	0xfff0
0x3000fe52	cmpl $0, %gs:104	0x65833c256800000000
0x3000fe5b	jne 127 ; ABS: 0x3000fedc	0x757f
0x3000fe5d	movl 8(%rsi), %ecx // NPE-> 0x3000fef5	0x8b4e08
0x3000fe60	testq %rcx, %rcx	0x4885c9
0x3000fe63	je 115 ; ABS: 0x3000fed8	0x7473
0x3000fe65	cmpl $7, %ecx	0x83f907
0x3000fe68	ja 20 ; ABS: 0x3000fe7e	0x7714
0x3000fe6a	xorl %edx, %edx	0x31d2
0x3000fe6c	xorl %eax, %eax	0x31c0
0x3000fe6e	nop	0x6690
0x3000fe70	addl 12(%rsi,%rdx,4), %eax	0x0344960c
0x3000fe74	incq %rdx	0x48ffc2
0x3000fe77	cmpq %rcx, %rdx	0x4839ca
0x3000fe7a	jl -12 ; ABS: 0x3000fe70	0x7cf4
0x3000fe7c	popq %rcx	0x59
0x3000fe7d	retq	0xc3
0x3000fe7e	movl %ecx, %r8d	0x4189c8
0x3000fe81	andl $7, %r8d	0x4183e007
0x3000fe85	movq %rcx, %rdx	0x4889ca
0x3000fe88	subq %r8, %rdx	0x4c29c2
0x3000fe8b	je -35 ; ABS: 0x3000fe6a	0x74dd
0x3000fe8d	leaq 28(%rsi), %rax	0x488d461c
0x3000fe91	pxor %xmm0, %xmm0	0x660fefc0
0x3000fe95	movq %rdx, %rdi	0x4889d7
0x3000fe98	pxor %xmm1, %xmm1	0x660fefc9
0x3000fe9c	nopl (%rax)	0x0f1f4000
0x3000fea0	movdqu -16(%rax), %xmm2	0xf30f6f50f0
0x3000fea5	movdqu (%rax), %xmm3	0xf30f6f18
0x3000fea9	paddd %xmm2, %xmm0	0x660ffec2
0x3000fead	paddd %xmm3, %xmm1	0x660ffecb
0x3000feb1	addq $32, %rax	0x4883c020
0x3000feb5	addq $-8, %rdi	0x4883c7f8
0x3000feb9	jne -27 ; ABS: 0x3000fea0	0x75e5
0x3000febb	paddd %xmm0, %xmm1	0x660ffec8
0x3000febf	pshufd $78, %xmm1, %xmm0	0x660f70c14e
0x3000fec4	paddd %xmm1, %xmm0	0x660ffec1
0x3000fec8	phaddd %xmm0, %xmm0	0x660f3802c0
0x3000fecd	movd %xmm0, %eax	0x660f7ec0
0x3000fed1	testl %r8d, %r8d	0x4585c0
0x3000fed4	jne -102 ; ABS: 0x3000fe70	0x759a
0x3000fed6	jmp -92 ; ABS: 0x3000fe7c	0xeba4
0x3000fed8	xorl %eax, %eax	0x31c0
0x3000feda	popq %rcx	0x59
0x3000fedb	retq	0xc3
0x3000fedc	movq %rsi, (%rsp)	0x48893424
0x3000fee0	movabsq $805334400, %rax	0x48b8806d003000000000
0x3000feea	callq *%rax	0xffd0
0x3000feec	movq (%rsp), %rsi	0x488b3424
0x3000fef0	jmp -152 ; ABS: 0x3000fe5d	0xe968ffffff
0x3000fef5	movabsq $805319872, %rax	0x48b8c034003000000000
0x3000feff	movl $7, %edi	0xbf07000000
0x3000ff04	callq *%rax	0xffd0
0x3000ff06	addq $-8, %rsp	0x4883c4f8
0x3000ff0a	jmp -50575 ; ABS: 0x30003980 = StubRoutines::deoptimize	0xe9713affff
0x3000ff0f	int3	0xcc

Только одна реализация метода Animal.getColor()

FastDoof.buf поле финальное

Bar класс не имеет наследников

Сегодня суббота

Assertion выключены

Locale.default() == ENGLISH

Все String размером <128KB

Код функции SomeUtil.computeStuff() это {....} и чексумма класса 0x651712365

azul

Код + Assumptions

Optimized Code

Address	Code	Opcode
0x3000fe50	pushq %rax	0xfff0
0x3000fe52	cmpl $0, %gs:104	0x65833c256800000000
0x3000fe5b	jne 127 ; ABS: 0x3000fedc	0x757f
0x3000fe5d	movl 8(%rsi), %ecx // NPE-> 0x3000fef5	0x8b4e08
0x3000fe60	testq %rcx, %rcx	0x4885c9
0x3000fe63	je 115 ; ABS: 0x3000fed8	0x7473
0x3000fe65	cmpl $7, %ecx	0x83f907
0x3000fe68	ja 20 ; ABS: 0x3000fe7e	0x7714
0x3000fe6a	xorl %edx, %edx	0x31d2
0x3000fe6c	xorl %eax, %eax	0x31c0
0x3000fe6e	nop	0x6690
0x3000fe70	addl 12(%rsi,%rdx,4), %eax	0x0344960c
0x3000fe74	incq %rdx	0x48ffc2
0x3000fe77	cmpq %rcx, %rdx	0x4839ca
0x3000fe7a	jl -12 ; ABS: 0x3000fe70	0x7cf4
0x3000fe7c	popq %rcx	0x59
0x3000fe7d	retq	0xc3
0x3000fe7e	movl %ecx, %r8d	0x4189c8
0x3000fe81	andl $7, %r8d	0x4183e007
0x3000fe85	movq %rcx, %rdx	0x4889ca
0x3000fe88	subq %r8, %rdx	0x4c29c2
0x3000fe8b	je -35 ; ABS: 0x3000fe6a	0x74dd
0x3000fe8d	leaq 28(%rsi), %rax	0x488d461c
0x3000fe91	pxor %xmm0, %xmm0	0x660fefc0
0x3000fe95	movq %rdx, %rdi	0x4889d7
0x3000fe98	pxor %xmm1, %xmm1	0x660fefc9
0x3000fe9c	nopl (%rax)	0x0f1f4000
0x3000fea0	movdqu -16(%rax), %xmm2	0xf30f6f50f0
0x3000fea5	movdqu (%rax), %xmm3	0xf30f6f18
0x3000fea9	paddd %xmm2, %xmm0	0x660ffec2
0x3000fead	paddd %xmm3, %xmm1	0x660ffecb
0x3000feb1	addq $32, %rax	0x4883c020
0x3000feb5	addq $-8, %rdi	0x4883c7f8
0x3000feb9	jne -27 ; ABS: 0x3000fea0	0x75e5
0x3000febb	paddd %xmm0, %xmm1	0x660ffec8
0x3000febf	pshufd $78, %xmm1, %xmm0	0x660f70c14e
0x3000fec4	paddd %xmm1, %xmm0	0x660ffec1
0x3000fec8	phaddd %xmm0, %xmm0	0x660f3802c0
0x3000fecd	movd %xmm0, %eax	0x660f7ec0
0x3000fed1	testl %r8d, %r8d	0x4585c0
0x3000fed4	jne -102 ; ABS: 0x3000fe70	0x759a
0x3000fed6	jmp -92 ; ABS: 0x3000fe7c	0xeba4
0x3000fed8	xorl %eax, %eax	0x31c0
0x3000feda	popq %rcx	0x59
0x3000fedb	retq	0xc3
0x3000fedc	movq %rsi, (%rsp)	0x48893424
0x3000fee0	movabsq $805334400, %rax	0x48b8806d003000000000
0x3000feea	callq *%rax	0xffd0
0x3000feec	movq (%rsp), %rsi	0x488b3424
0x3000fef0	jmp -152 ; ABS: 0x3000fe5d	0xe968ffffff
0x3000fef5	movabsq $805319872, %rax	0x48b8c034003000000000
0x3000feff	movl $7, %edi	0xbf07000000
0x3000ff04	callq *%rax	0xffd0
0x3000ff06	addq $-8, %rsp	0x4883c4f8
0x3000ff0a	jmp -50575 ; ABS: 0x30003980 = StubRoutines::deoptimize	0xe9713affff
0x3000ff0f	int3	0xcc

Только одна реализация метода Animal.getColor()

FastDoof.buf поле финальное

Bar класс не имеет наследников

Сегодня суббота

Assertion выключены

Locale.default() == ENGLISH

Все String размером <128KB

Код функции SomeUtil.computeStuff() это {....} и чексумма класса 0x651712365

Оптимизированный код корректен тогда и только тогда, когда все предположения истины

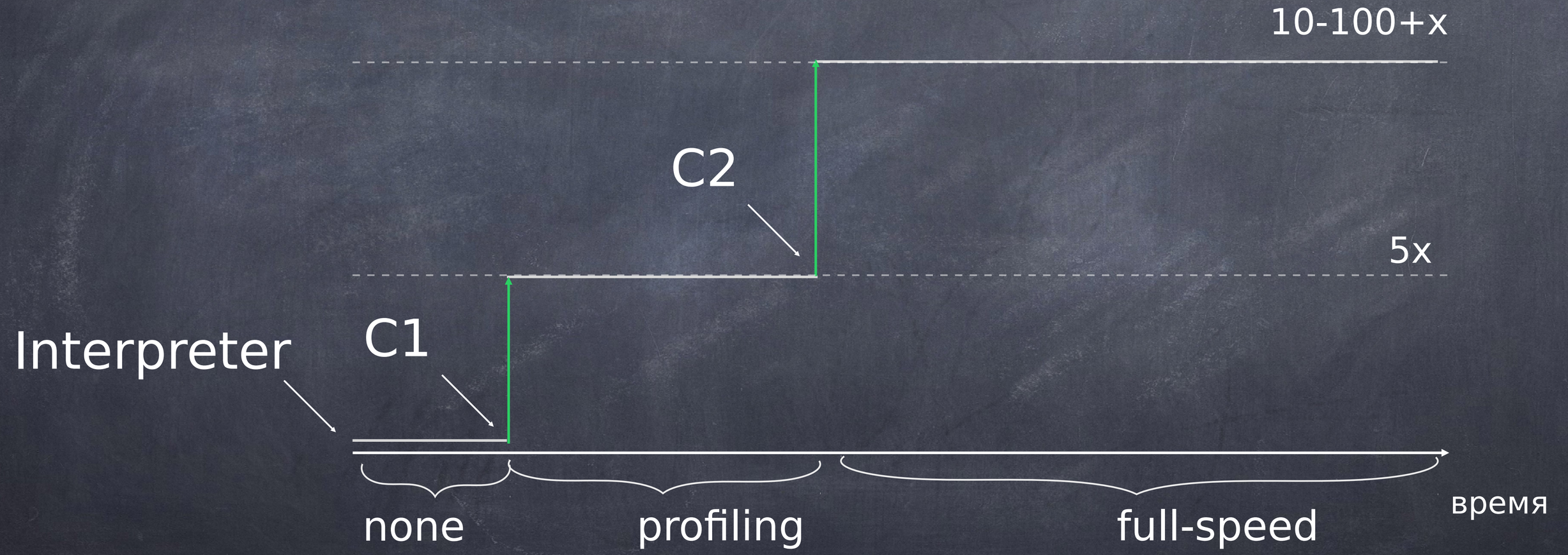
azul

“Героические” (спекулятивные) оптимизации

Guarded спекуляции

azul

Уточненная модель жизни одного метода PGO (Profile Guided Optimizations)



azul

Пример: Guarded девиртуализация

```
func.apply(x);
```

Devirtualize + Inline

```
if ( func.getClass() == Square.class ) {
  return x * x;
} else if ( func.getClass() == Sqrt.class ) {
  return Math.sqrt(x)
} else {
  Return func.apply(x); // virtual call
}
```

```
<klass id='780' name='Square' flags='1'/>
<klass id='781' name='Sqrt' flags='1'/>
<call method='783' count='23161'
  prof_factor='1' virtual='1' inline='1'
  receiver='780' receiver_count='19901'
  receiver2='781' receiver2_count='3260'/>
<predicted_call bci='3' klass='780'/>
<predicted_call bci='3' klass='781'/>
<uncommon_trap bci='3'
  reason='bimorphic'
  action='maybe_recompile'/>
```


azul

Пример: Мертвый код

```
static final void hotMethod() {
  if ( thing == null )
    System.out.print("always");
  else
    System.out.print("never");
}
```



```
static final void hotMethod() {
  if ( thing == null )
    System.out.print("always");
  else
    uncommon_trap(:unreached);
}
```

```
<bc code='199' bci='3'/>
<branch target_bci='17'
 taken='0' not_taken='5800'
 cnt='5800' prob='never'/>
<uncommon_trap
 bci='3' reason='unstable_if'
 action='reinterpret'
 comment='taken never'/>
```

```
<uncommon_trap thread='7171' stamp='5.104'
 compile_id='29' compiler='C2' level='4'
 reason='unstable_if' action='reinterpret' >
 <jvms
  method='...Unreached hotMethod ()V'
  bci='3'.../>
</uncommon_trap>
```


azul

Пример: Неявная проверка на null

```
if ( value == null ) {
  throw new NullPointerException();
}
value.hashCode();
```

⟵ Возможно,
Но маловероятно

```
0x10795f9cc: mov    0x8(%rsi),%r10d
    ; implicit exception: dispatches to 0x10795fe1d
```

azul

Пример: Проверка инициализации класса

MyClass.getStatic()

```
if (!vm.is_init(MyClass)) {
  vm.init(MyClass);
}
```

MyClass.getStatic()

MyClass.getStatic()

new MyClass()

Увеличение размера 20%

Замедление 5-10%

http://joeduffyblog.com/2015/12/19/safe-native-code/

azul

“Героические” (спекулятивные) оптимизации

de-optimization

azul

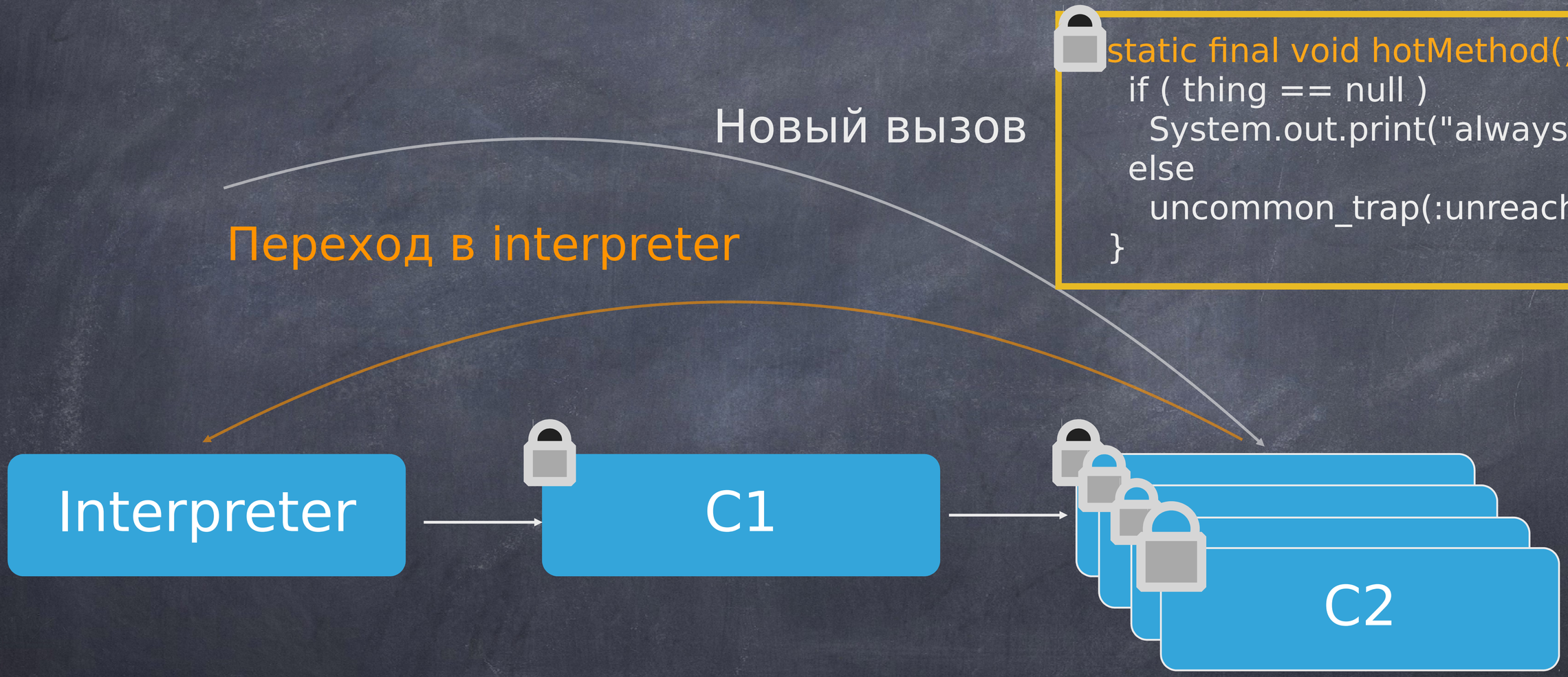
Переход в Interpreter: однократно

```
static final void hotMethod() {
  if ( thing == null )
    System.out.print("always");
  else
    uncommon_trap(:unreached);
}
```

azul

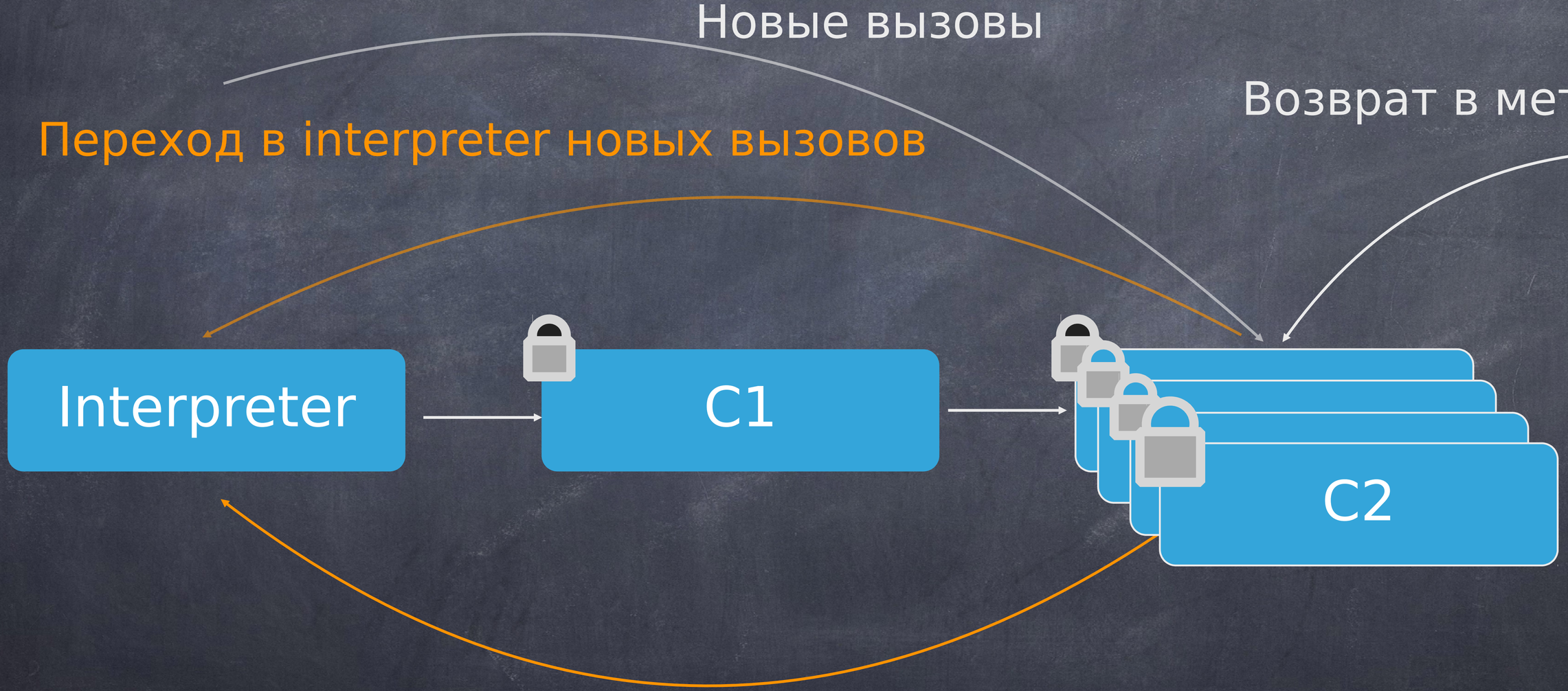
Переход в Interpreter: Всех новых вызовов

```
static final void hotMethod() {
  if ( thing == null )
    System.out.print("always");
  else
    uncommon_trap(:unreached);
}
```



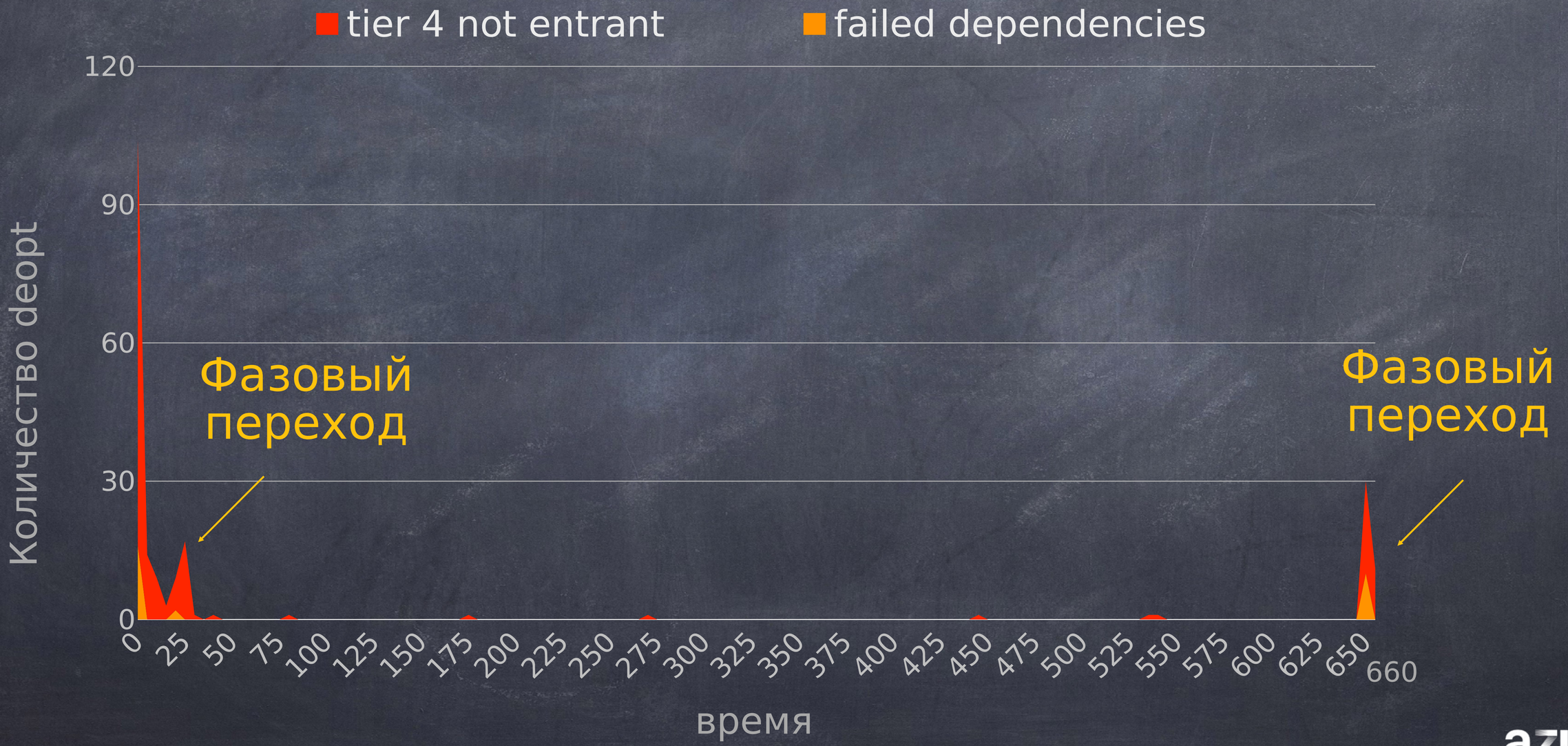
azul

Переход в Interpreter: Всех попыток исполнения



azul

Нарушение спекулятивных предположений



azul

“Героические” (спекулятивные) оптимизации

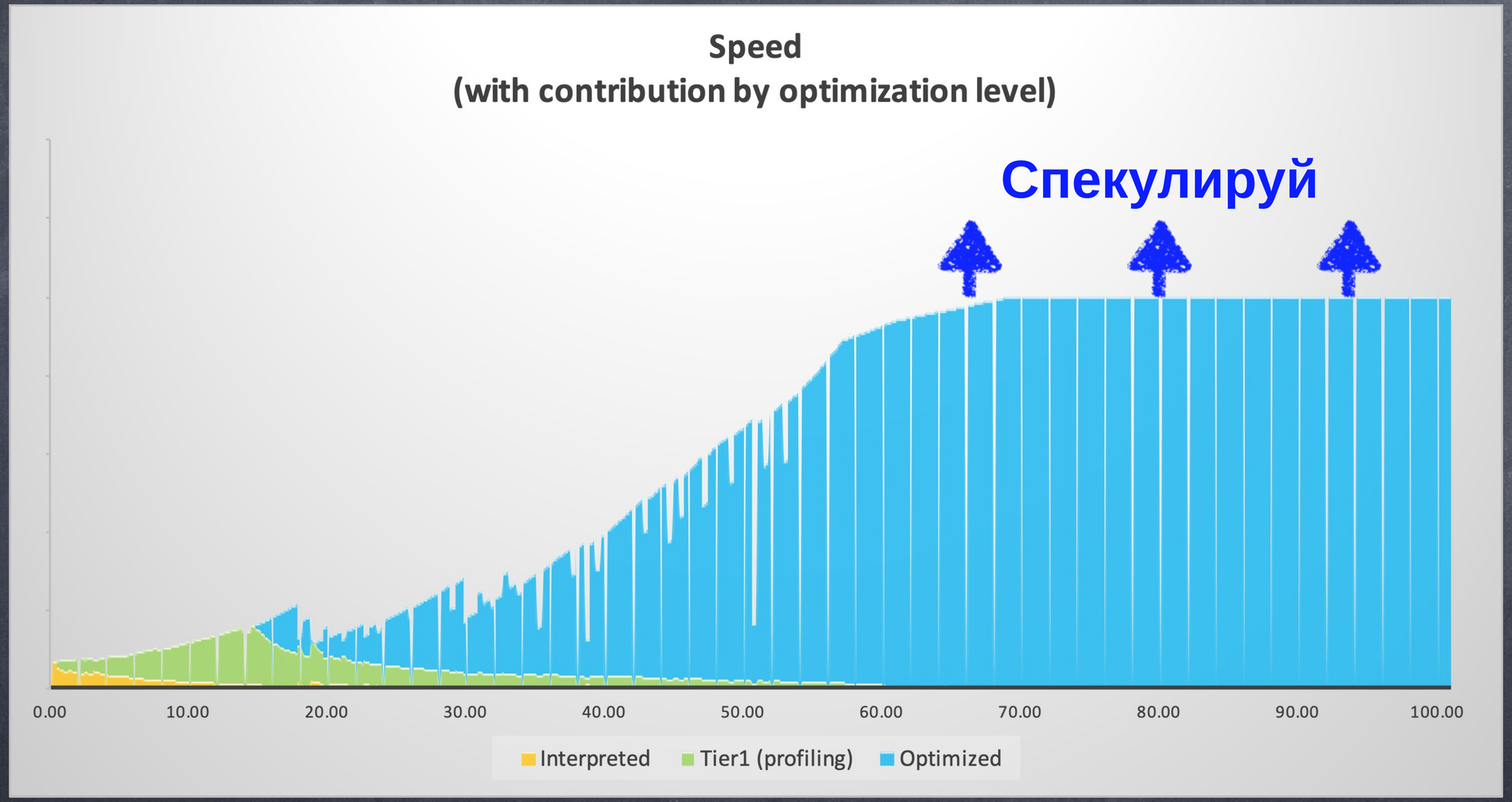
“секрет” скорости Java

azul

25+%

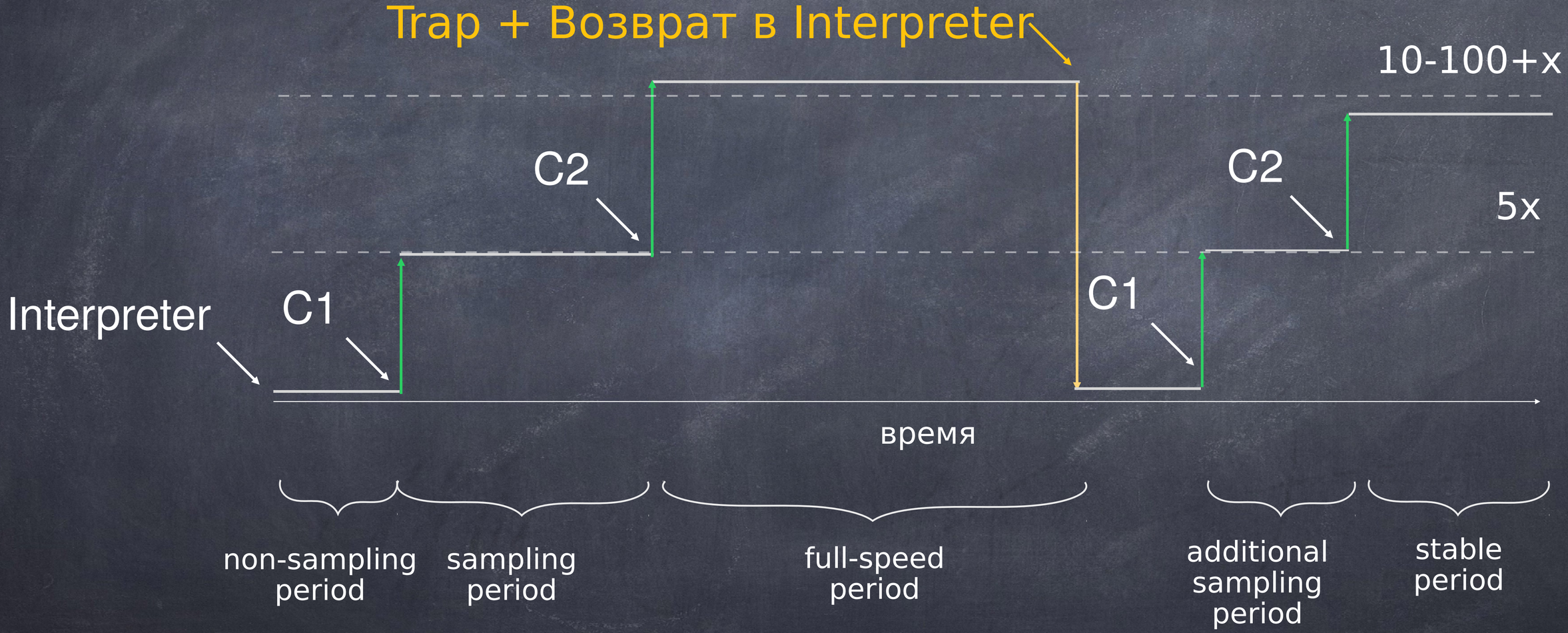
И еще больше в циклах

azul



azul

Уточненная модель жизни одного метода



https://blog.takipi.com/java-on-steroids-5-super-useful-jit-optimization-techniques/

azul

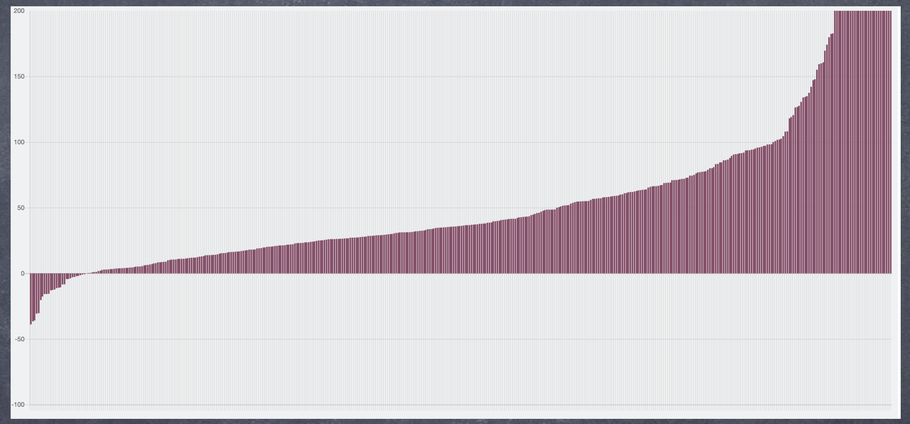
Оптимизации

Инлайнинг
Escape анализ
Векторизация
....

azul

Лучше оптимизировать значит...
быстрее исполнять

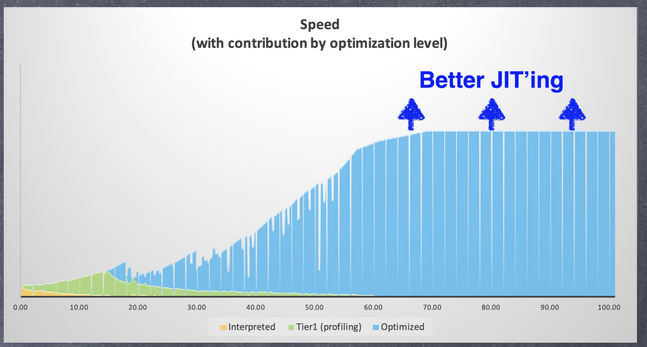
Workload	Zing Throughput as % of OpenJDK
Cassandra	128.42%
jmh-8-stream-v2	153.47%
Renaissance	142.81%
Disruptor	132.46%
Kafka	153.85%



azul

Серьезные JIT оптимизации имеют свою цену

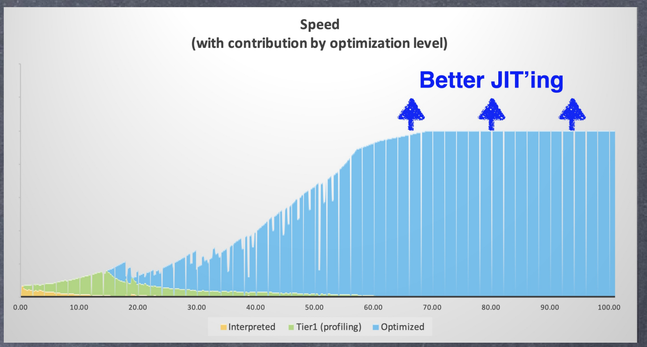
- Мы можем применять более мощные оптимизаци и получить (значительно) более быстрый код
-



azul

Серьезные JIT оптимизации имеют свою цену

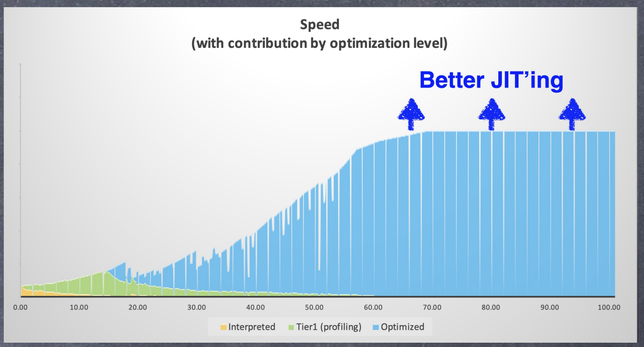
- Мы можем применять более мощные оптимизаци и получить (значительно) более быстрый код
- Но стоимость расходов CPU может значительно увеличиться
-



azul

Серьезные JIT оптимизации имеют свою цену

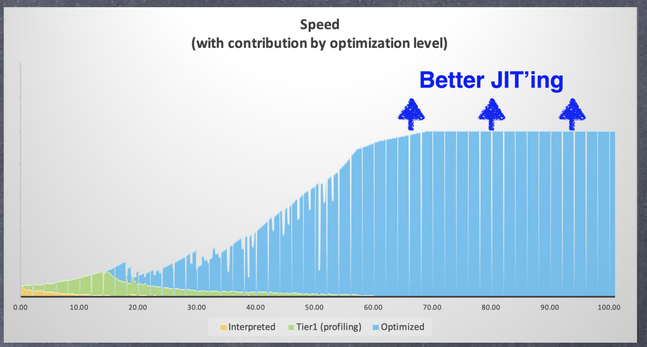
- Мы можем применять более мощные оптимизаци и получить (значительно) более быстрый код
- Но стоимость расходов CPU может значительно увеличиться
- Аналогично увеличение потребляемой (JIT) памяти
-



azul

Серьезные JIT оптимизации имеют свою цену

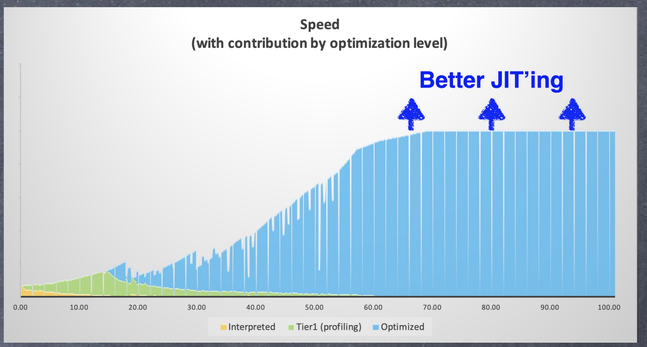
- Мы можем применять более мощные оптимизаци и получить (значительно) более быстрый код
- Но стоимость расходов CPU может значительно увеличиться
- Аналогично увеличение потребляемой (JIT) памяти
- На мощных серверах (64 vcores) такое работает



azul

Серьезные JIT оптимизации имеют свою цену

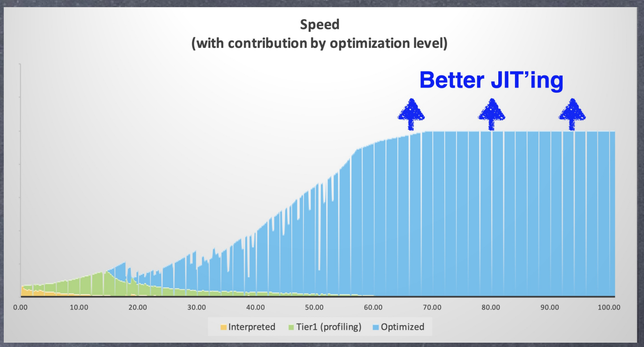
- Мы можем применять более мощные оптимизаци и получить (значительно) более быстрый код
- Но стоимость расходов CPU может значительно увеличиться
- Аналогично увеличение потребляемой (JIT) памяти
- На мощных серверах (64 vcores) такое работает
- Но что с маленькими “constrained” контейнерами…



azul

Серьезные JIT оптимизации имеют свою цену

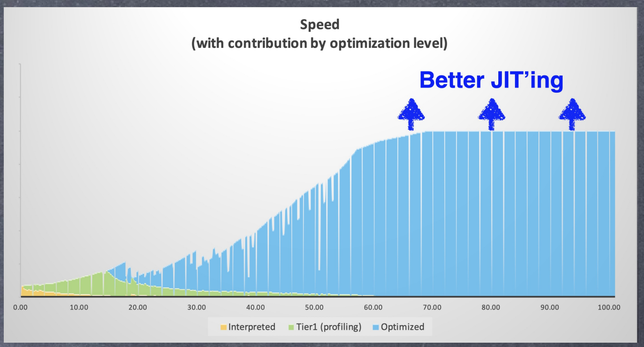
- Мы можем применять более мощные оптимизаци и получить (значительно) более быстрый код
- Но стоимость расходов CPU может значительно увеличиться
- Аналогично увеличение потребляемой (JIT) памяти
- На мощных серверах (64 vcores) такое работает
- Но что с маленькими “constrained” контейнерами…
  - Стоимость JIT CPU (& возможное замедление контейнера) во время оптимизаций может оказаться непомерно высокой
  -



azul

Серьезные JIT оптимизации имеют свою цену

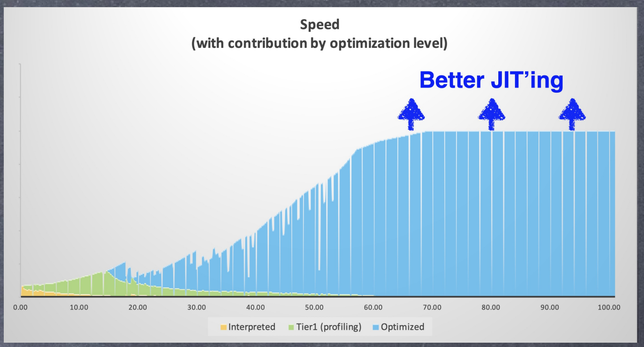
- Мы можем применять более мощные оптимизаци и получить (значительно) более быстрый код
- Но стоимость расходов CPU может значительно увеличиться
- Аналогично увеличение потребляемой (JIT) памяти
- На мощных серверах (64 vcores) такое работает
- Но что с маленькими “constrained” контейнерами…
  - Стоимость JIT CPU (& возможное замедление контейнера) во время оптимизаций может оказаться непомерно высокой
  - увеличение потребляемой (JIT) памяти Jможет заставить сработать OOM Killer



azul

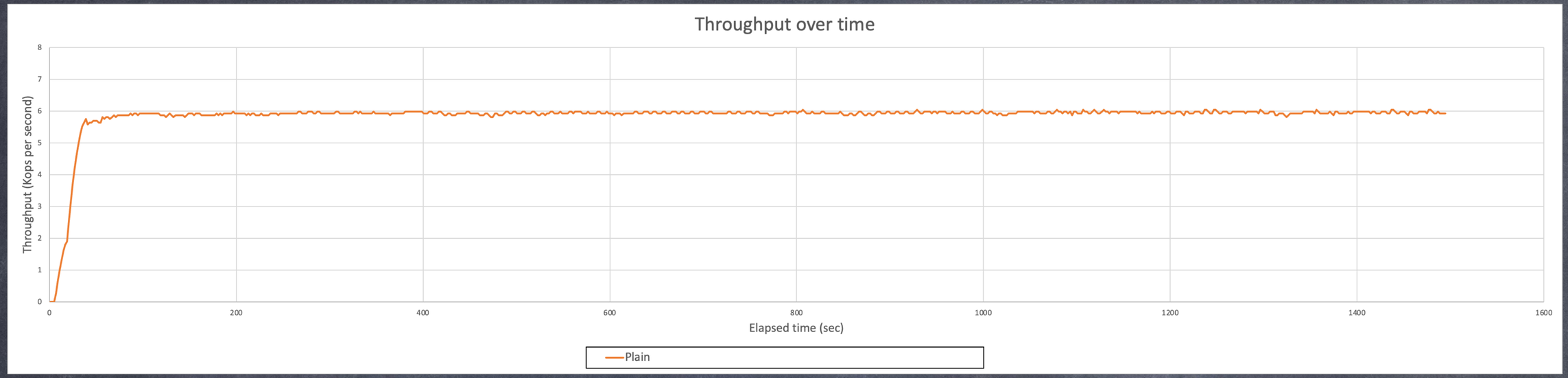
Серьезные JIT оптимизации имеют свою цену

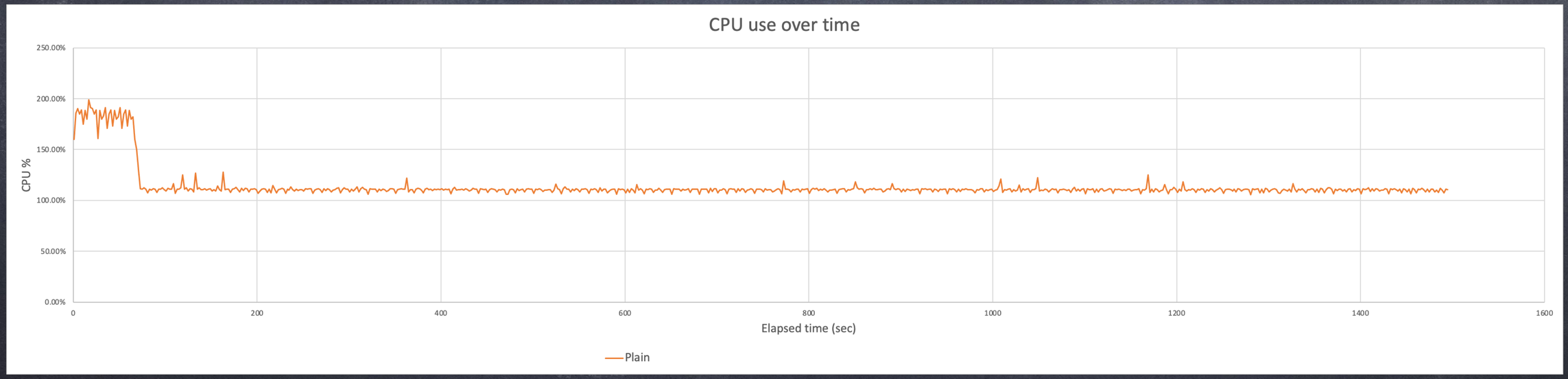
- Мы можем применять более мощные оптимизаци и получить (значительно) более быстрый код
- Но стоимость расходов CPU может значительно увеличиться
- Аналогично увеличение потребляемой (JIT) памяти
- На мощных серверах (64 vcores) такое работает
- Но что с маленькими “constrained” контейнерами…
  - Стоимость JIT CPU (& возможное замедление контейнера) во время оптимизаций может оказаться непомерно высокой
  - увеличение потребляемой (JIT) памяти Jможет заставить сработать OOM Killer
  - Вероятно, придется делать выбор между уровнями оптимизаций и размером контейнера



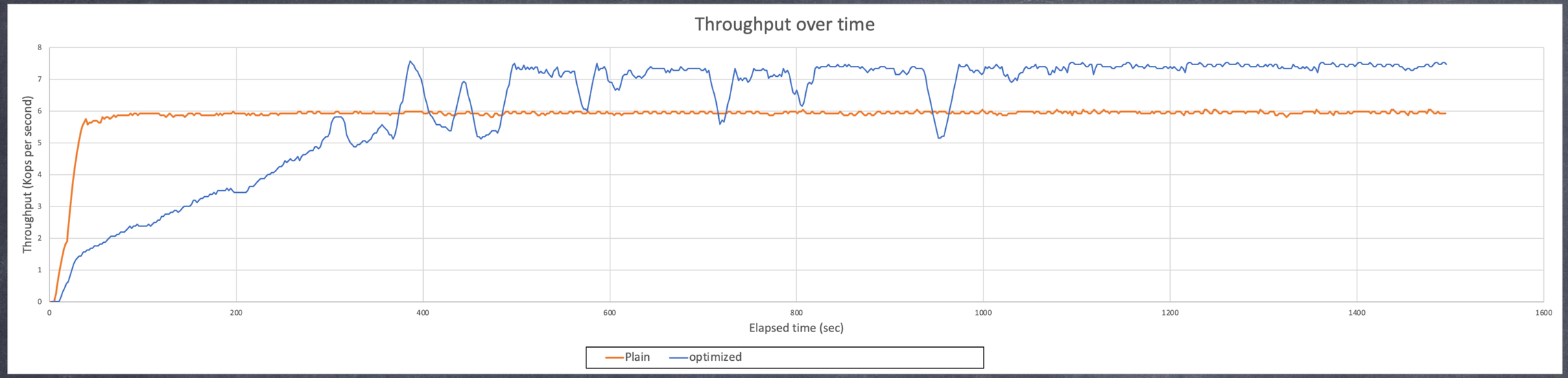
azul

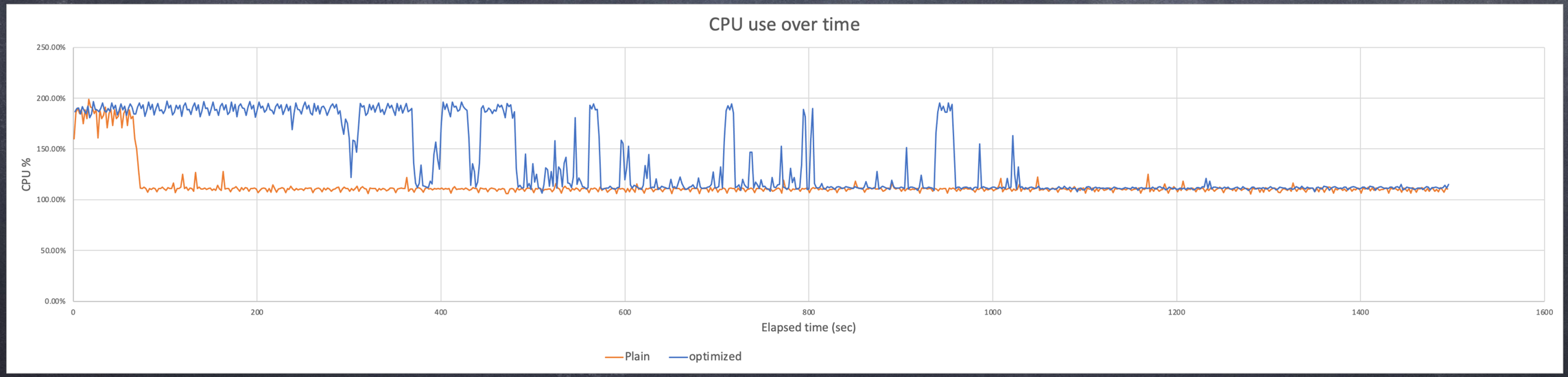
Скорость & CPU по времени (2-vcore контейнер)





Скорость & CPU по времени (2-vcore контейнер)





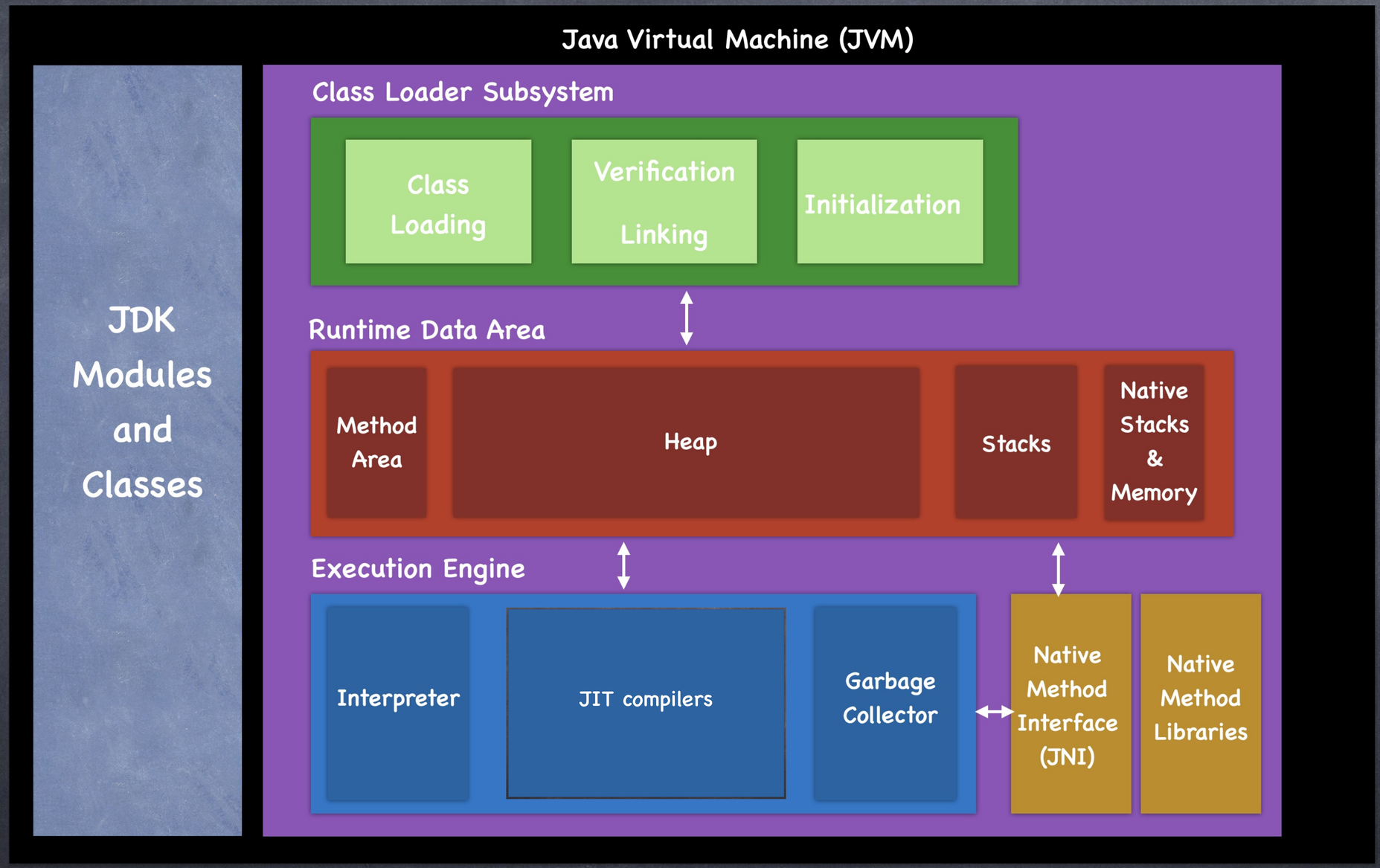
azul

Аутсорсинг

Outsourcing

azul

Локальная самодостаточная JVM...



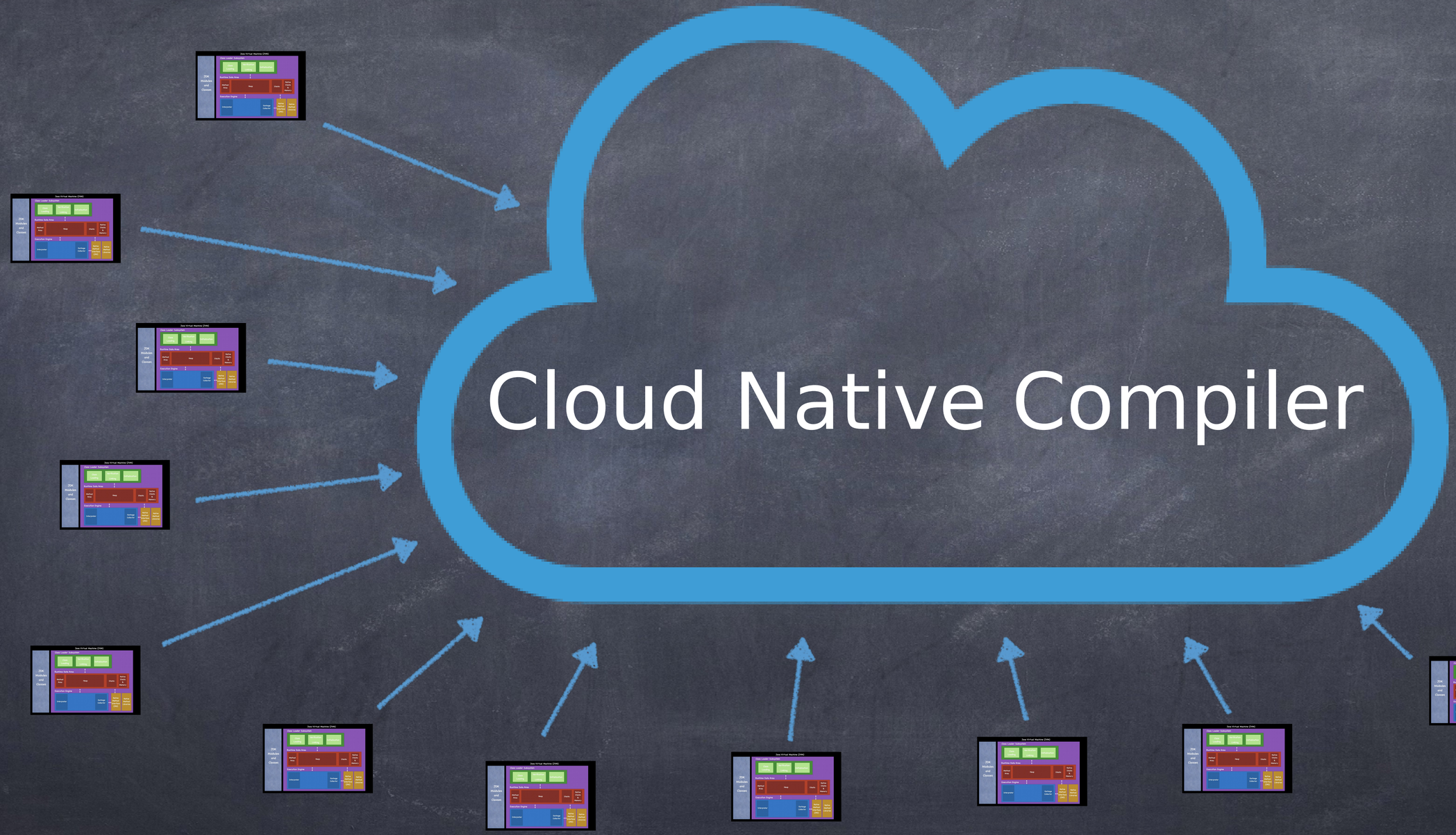
azul

Что если JIT компилятор вынести из JVM?

JIT compilers

azul

И сделать его Cloud Native ресурсом...



azul

Что даёт?

azul

Эффективность

Время прогрева

Скорость

Что даёт?

Память

Эффективность

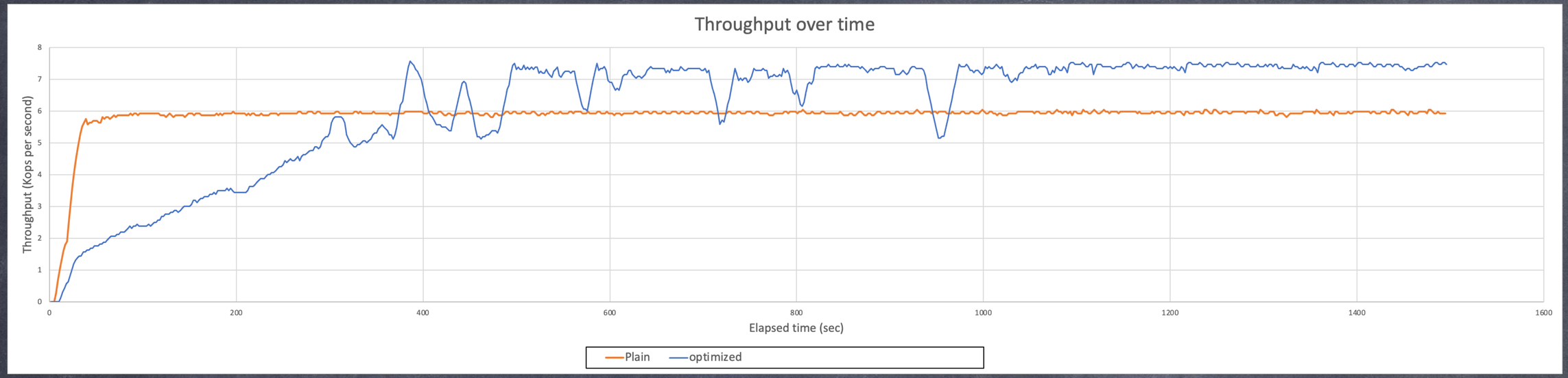
Размер контейнера

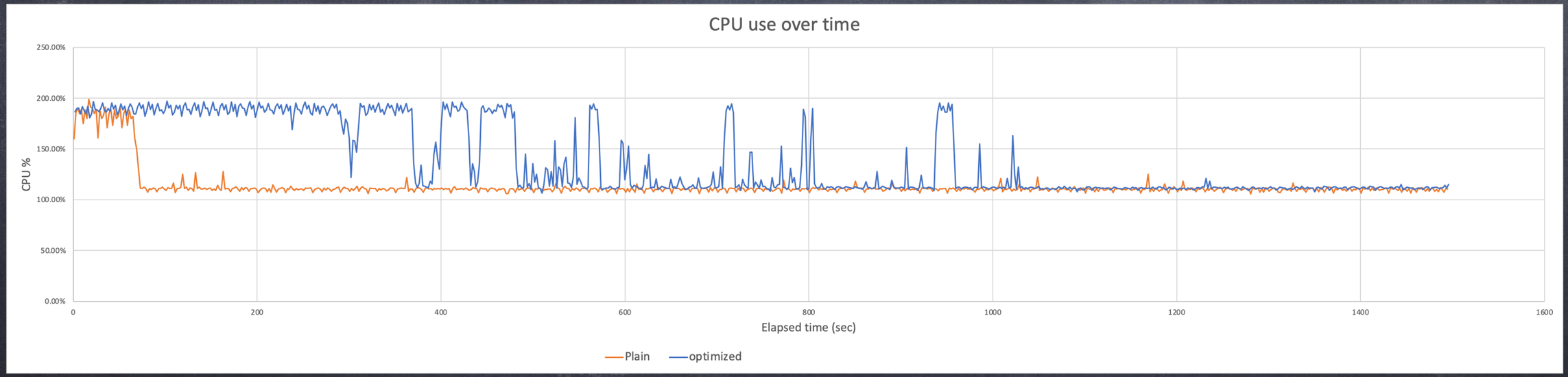
Скорость

Скорость

azul

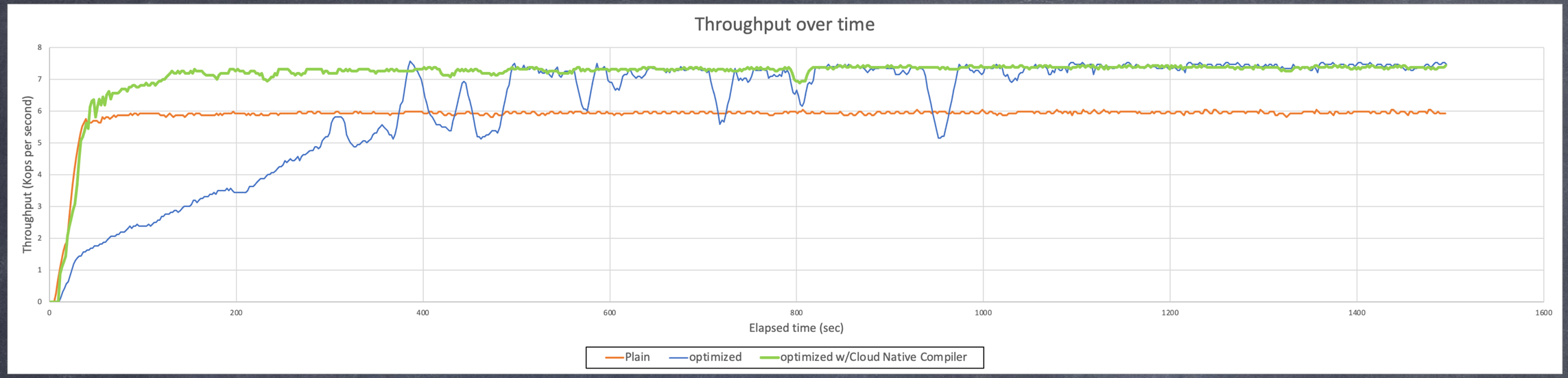
Скорость & CPU по времени (2-vcore контейнер)

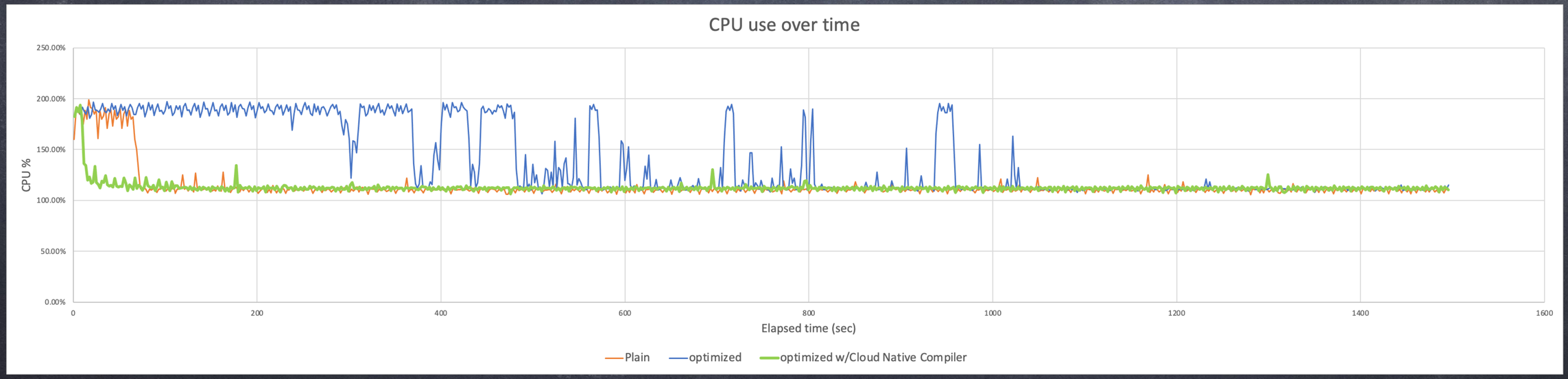




azul

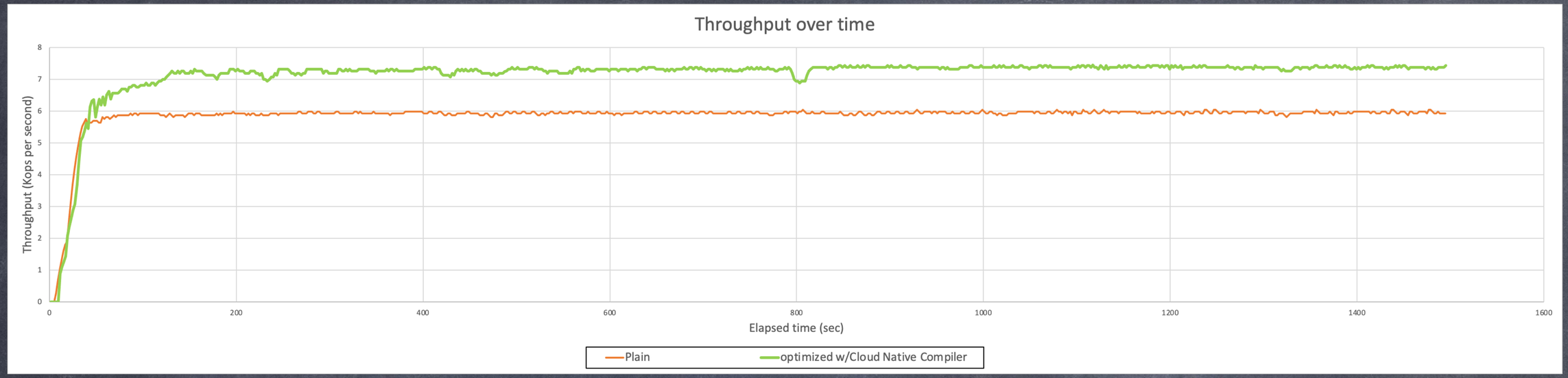
Скорость & CPU по времени (2-vcore контейнер)

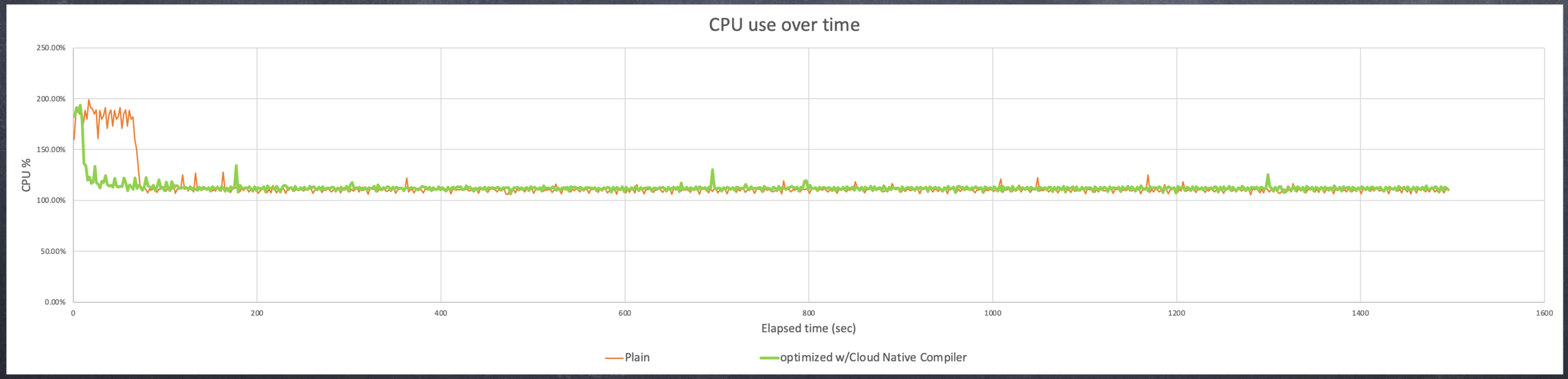




azul

Скорость & CPU по времени (2-vcore контейнер)





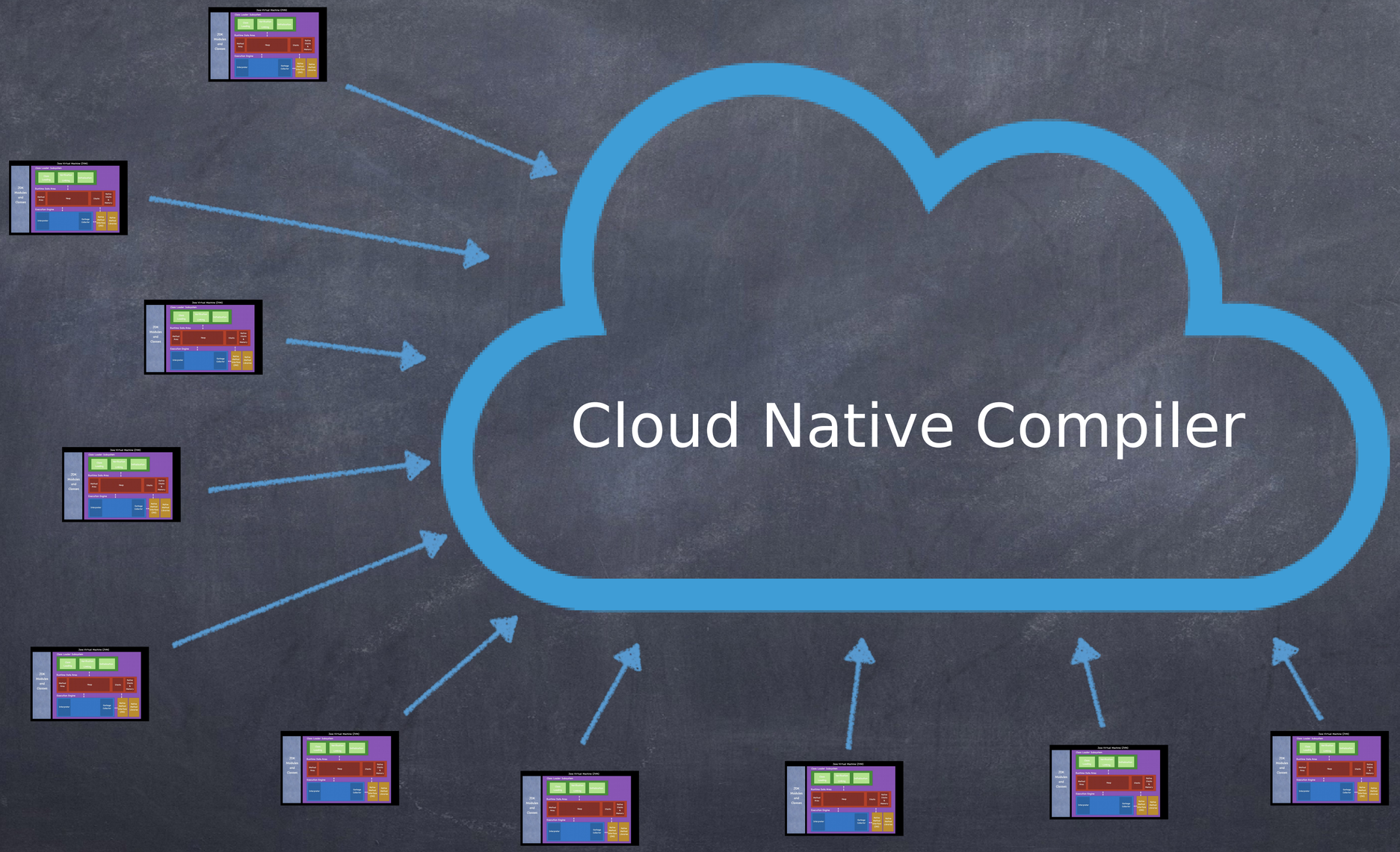
azul

А разве это не перенос оплаты в другое место?

- Вообще-то, именно так…
- Но платим мы меньше, используя более эффективную инфраструктуру
- Ресурсы, необходимые для осуществления оптимизаций расходуются в течение короткого промежутка времени
- Когда JVM оптимизирует локально, то с выделенными ресурсами остаётся навсегда
- Когда JVM аутсорсит в Cloud Native компилятор
  - Ресурсы используются совместно и делятся между всеми
  - Ресурсы могут эластично добавляться и убираться

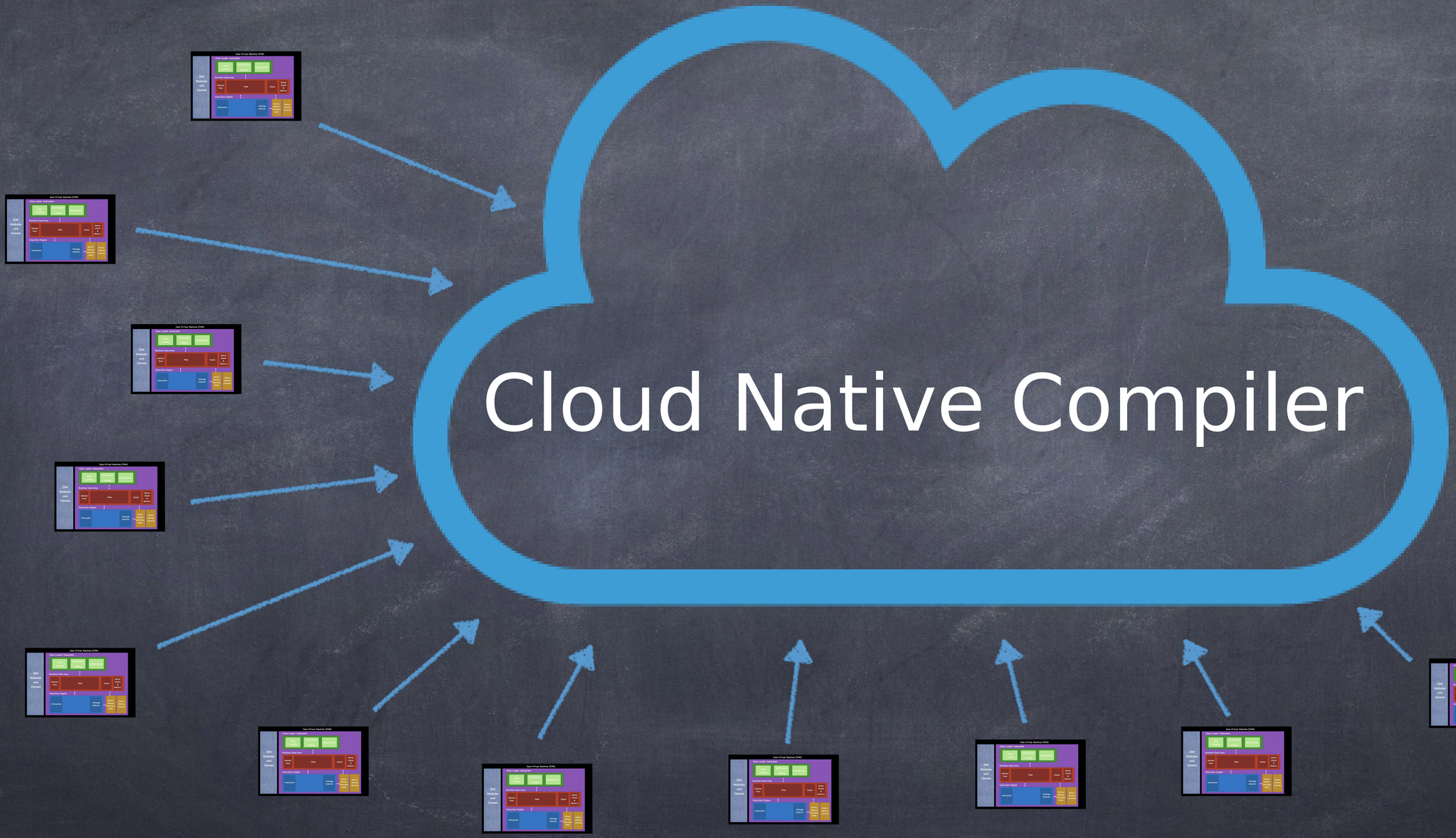
azul

Используются совместно и делятся между всеми...



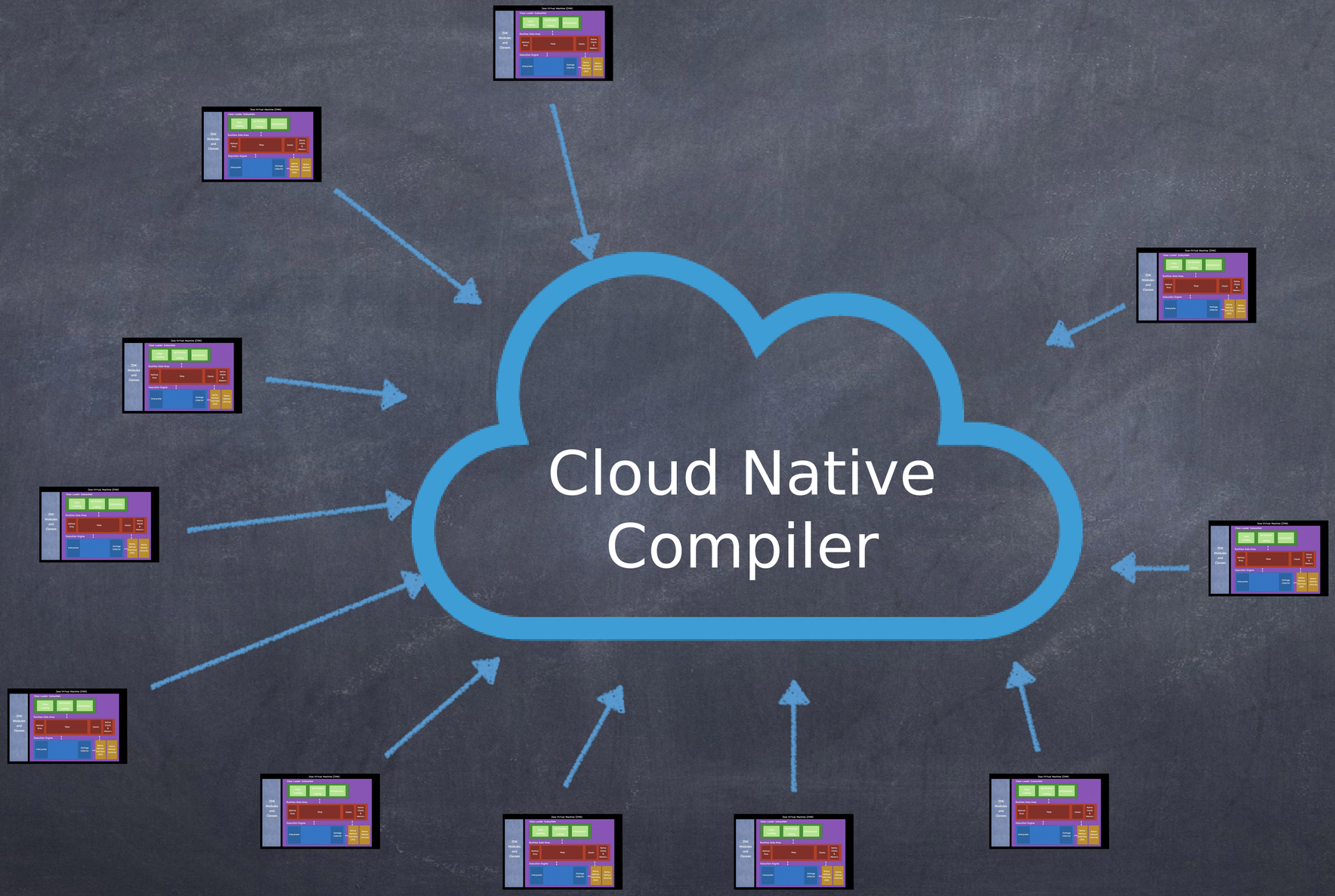
azul

Эластично добавляются...



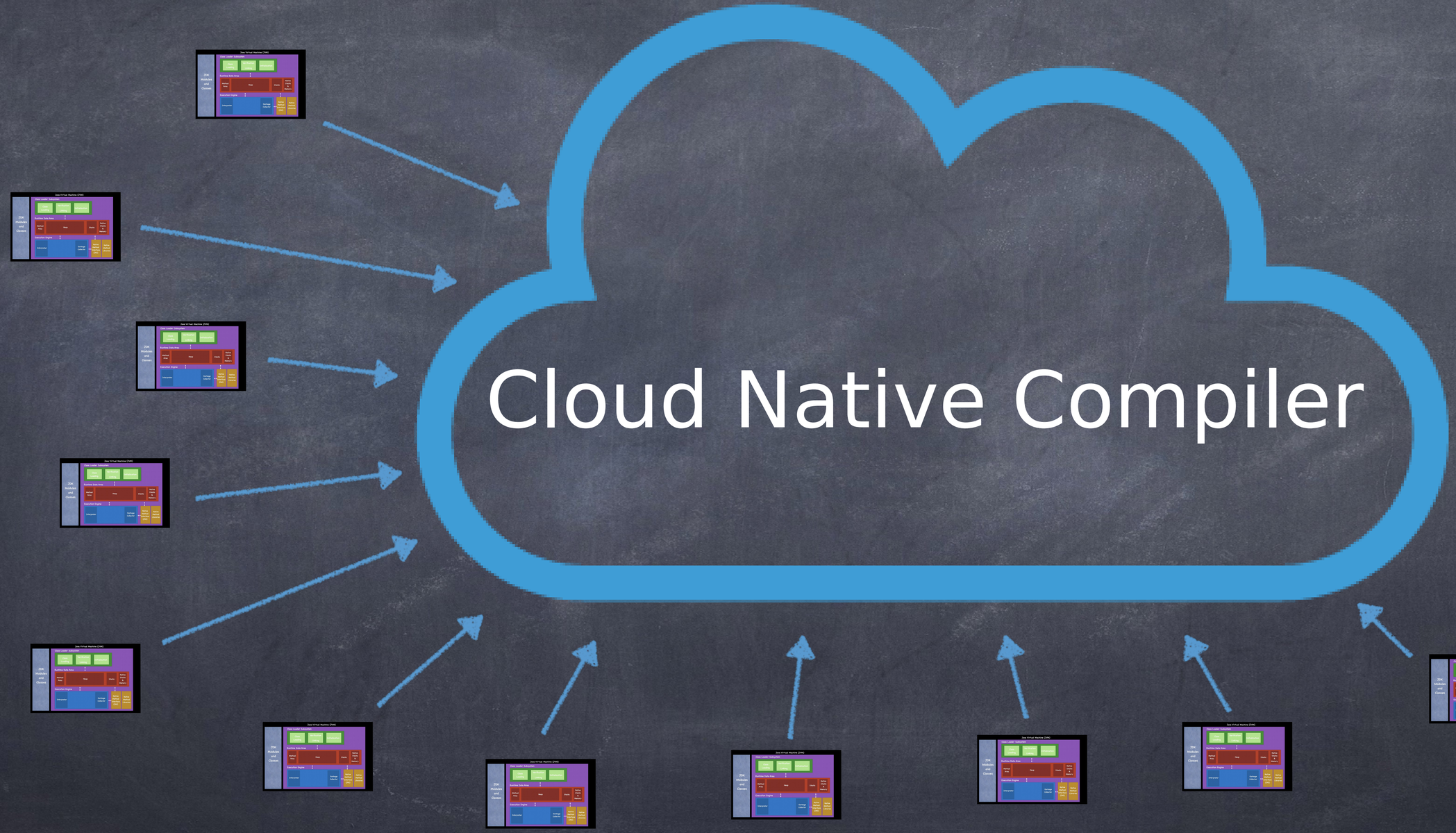
azul

Эластично добавляются и убираются...



azul

Эластично добавляются и убираются...



azul

И еще немного

azul

Простые наблюдения

- В большинстве случаев код приложения:
  - Выполняется множество раз
  - Выполняется на разных устройствах
  - Выполняет одинаковые задачи в разных запусках и на разных устройства
  - Имеет ограниченное множество различных сценариев исполнения
- Это верно даже для часто обновляющихся приложений…
- Когда JVM делает лично для себя оптимизацию и использует её однократно, она “теряется” при перезапуске

azul

Оптимизации можно переиспользовать...

Даже *спекулятивные* оптимизации...

azul

Код+Assumptions

Optimized Code

Address	Code	Opcode
0x3000fe50	pushq %rax	0xfff0
0x3000fe52	cmpl $0, %gs:104	0x65833c256800000000
0x3000fe5b	jne 127 ; ABS: 0x3000fedc	0x757f
0x3000fe5d	movl 8(%rsi), %ecx // NPE-> 0x3000fef5	0x8b4e08
0x3000fe60	testq %rcx, %rcx	0x4885c9
0x3000fe63	je 115 ; ABS: 0x3000fed8	0x7473
0x3000fe65	cmpl $7, %ecx	0x83f907
0x3000fe68	ja 20 ; ABS: 0x3000fe7e	0x7714
0x3000fe6a	xorl %edx, %edx	0x31d2
0x3000fe6c	xorl %eax, %eax	0x31c0
0x3000fe6e	nop	0x6690
0x3000fe70	addl 12(%rsi,%rdx,4), %eax	0x0344960c
0x3000fe74	incq %rdx	0x48ffc2
0x3000fe77	cmpq %rcx, %rdx	0x4839ca
0x3000fe7a	jl -12 ; ABS: 0x3000fe70	0x7cf4
0x3000fe7c	popq %rcx	0x59
0x3000fe7d	retq	0xc3
0x3000fe7e	movl %ecx, %r8d	0x4189c8
0x3000fe81	andl $7, %r8d	0x4183e007
0x3000fe85	movq %rcx, %rdx	0x4889ca
0x3000fe88	subq %r8, %rdx	0x4c29c2
0x3000fe8b	je -35 ; ABS: 0x3000fe6a	0x74dd
x3000fe8d	leaq 28(%rsi), %rax	0x488d461c
000fe91	pxor %xmm0, %xmm0	0x660fefc0
0	0fe95	movq %rdx, %rdi	0x4889d7
0x	fe98	pxor %xmm1, %xmm1	0x660fefc9
0x3	fe9c	nopl (%rax)	0x0f1f4000
0x3	fea0	movdqu -16(%rax), %xmm2	0xf30f6f50f0
0x	fea5	movdqu (%rax), %xmm3	0xf30f6f18
0	0fea9	paddd %xmm2, %xmm0	0x660ffec2
	000fead	paddd %xmm3, %xmm1	0x660ffecb
	3000feb1	addq $32, %rax	0x4883c020
0x3000feb5	addq $-8, %rdi	0x4883c7f8
0x3000feb9	jne -27 ; ABS: 0x3000fea0	0x75e5
0x3000febb	paddd %xmm0, %xmm1	0x660ffec8
0x3000febf	pshufd $78, %xmm1, %xmm0	0x660f70c14e
0x3000fec4	paddd %xmm1, %xmm0	0x660ffec1
0x3000fec8	phaddd %xmm0, %xmm0	0x660f3802c0
0x3000fecd	movd %xmm0, %eax	0x660f7ec0
0x3000fed1	testl %r8d, %r8d	0x4585c0
0x3000fed4	jne -102 ; ABS: 0x3000fe70	0x759a
0x3000fed6	jmp -92 ; ABS: 0x3000fe7c	0xeba4
0x3000fed8	xorl %eax, %eax	0x31c0
0x3000feda	popq %rcx	0x59
0x3000fedb	retq	0xc3
0x3000fedc	movq %rsi, (%rsp)	0x48893424
0x3000fee0	movabsq $805334400, %rax	0x48b8806d003000000000
0x3000feea	callq *%rax	0xffd0
0x3000feec	movq (%rsp), %rsi	0x488b3424
0x3000fef0	jmp -152 ; ABS: 0x3000fe5d	0xe968ffffff
0x3000fef5	movabsq $805319872, %rax	0x48b8c034003000000000
0x3000feff	movl $7, %edi	0xbf07000000
0x3000ff04	callq *%rax	0xffd0
0x3000ff06	addq $-8, %rsp	0x4883c4f8
0x3000ff0a	jmp -50575 ; ABS: 0x30003980 = StubRoutines::deoptimize	0xe9713affff
0x3000ff0f	int3	0xcc

Cloud Native Compiler

Assumptions

Only one implementor of Animal.getColor() exists

Assertions are disabled

Bar has no subclasses

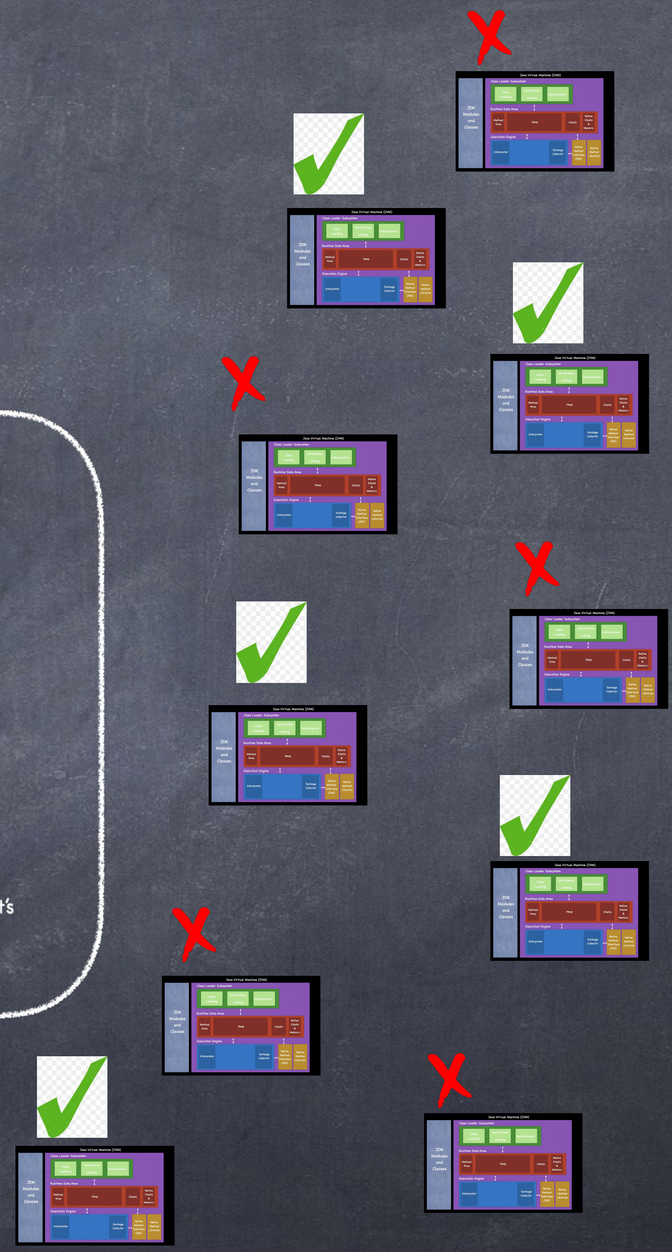
Today is Tuesday

FastDoof.buf is truly final

Locale.default() is ENGLISH

Longest String seen so far is <128KB

The actual code for SomeUtil.computeStuff() is {....} and it's checksum is 0x651712365



azul

А разве это не перенос оплаты в другое место?

- Вообще-то, именно так…
- Но платим мы меньше, используя более эффективную инфраструктуру
- Ресурсы, необходимые для осуществления оптимизаций расходуются в течение короткого промежутка времени
- Когда JVM оптимизирует локально, то с выделенными ресурсами остаётся навсегда
- Когда JVM аутсорсит в Cloud Native компилятор
  - Ресурсы используются совместно и делятся между всеми
  - Ресурсы могут эластично добавляться и убираться
  -

azul

А разве это не перенос оплаты в другое место?

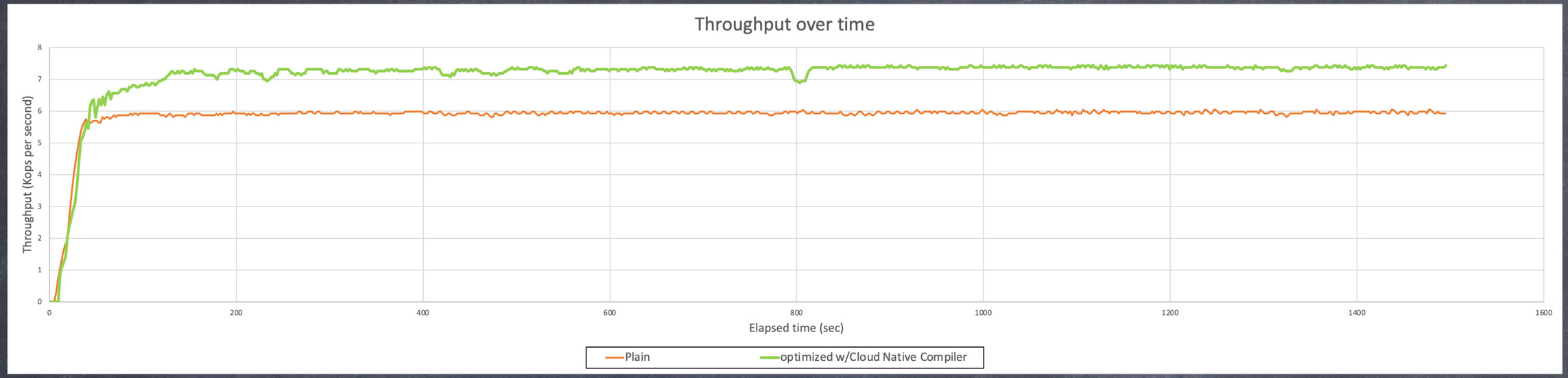
- Вообще-то, именно так…
- Но платим мы меньше, используя более эффективную инфраструктуру
- Ресурсы, необходимые для осуществления оптимизаций расходуются в течение короткого промежутка времени
- Когда JVM оптимизирует локально, то с выделенными ресурсами остаётся навсегда
- Когда JVM аутсорсит в Cloud Native компилятор
  - Ресурсы используются совместно и делятся между всеми
  - Ресурсы могут эластично добавляться и убираться
  - **РАБОТА МОЖЕТ ПЕРЕИСПОЛЬЗОВАТЬСЯ (!)**

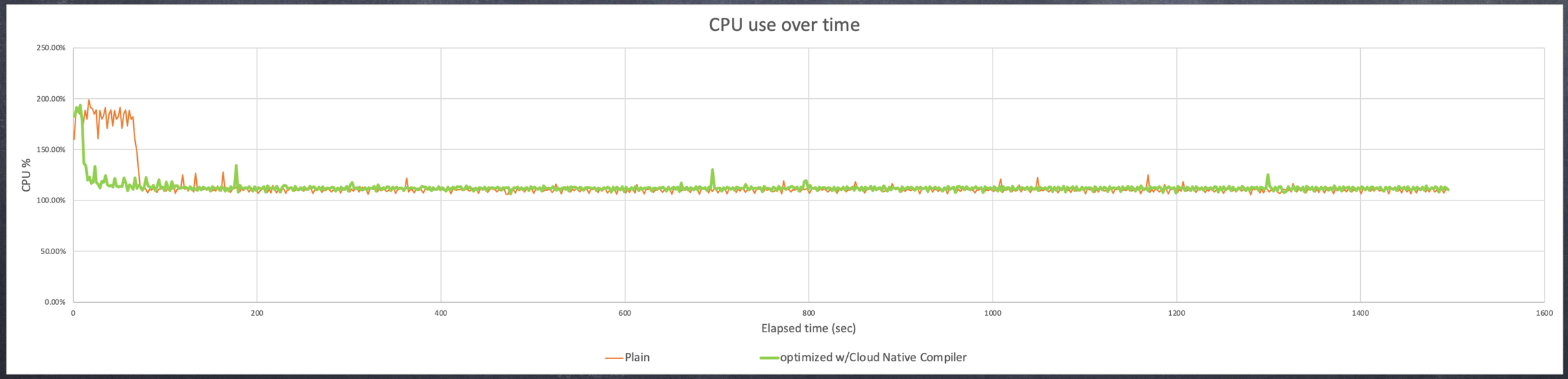
azul

Итого...

azul

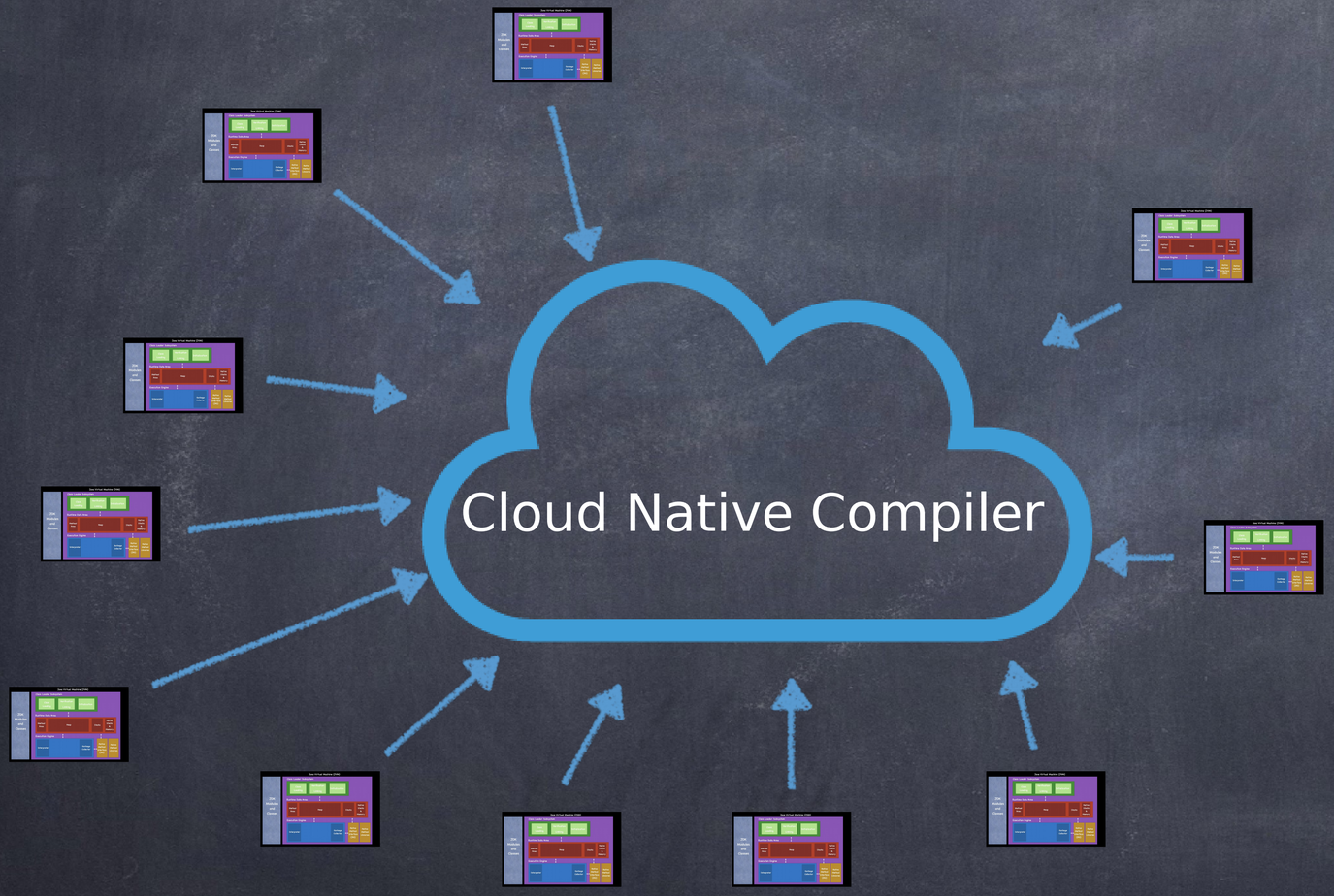
Cloud Native Компиляции == Быстрее

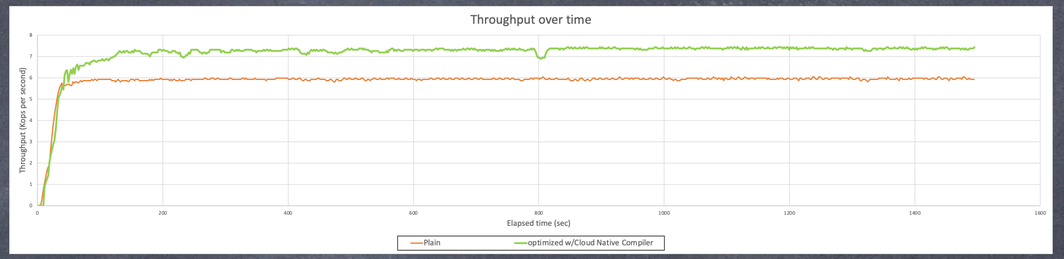


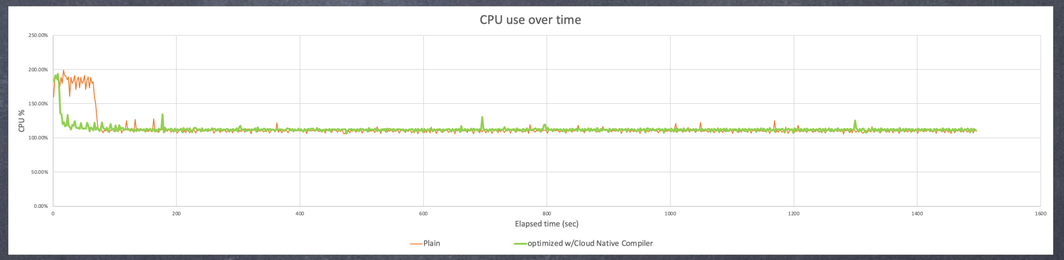


azul

Cloud Native Компиляции == Эффективнее







Cloud Native Compiler уже существует....

azul

Super-Duper optimizations
Application Level Intrinsics

azul

JVMCI++

azul

Cloud Native JVM
Cloud Native Compiler

Владимир Воскресенский
Azul Systems
Distinguished Engineer
vladimir@azul.com

Q&A

azul