End to end JIT

Карлен Симонян
разработчик

Синопсис

- Реализация методов в CLR
- Кооперация GC & JIT
- Бенчмарки

Warning! ⚡

- Доклад направлен на тему оптимизации уже оптимизированного кода
- Есть блок по теории GC (привет, академ)
- Будет много asm-кода
- WinDBG

Tools & co.

- Visual Studio 2013/2015
- .NET 4.6.1/CoreCLR (RyuJIT)
  - no Mono, sorry ☺
- WinDBG + SOS
- BenchmarkDotNet 0.10.1

Test Environment

- Intel® Xeon® E5-2670 v3 (2.60 GHz) (12x2)
  - 30M Cache
  - Turbo Boost 3.3 GHz
  - Haswell
- OS: Windows Server 2012 R2

Demo samples

https://github.com/szKarlen/JitDemo

Preamble

Simple benchmark

```
TimeSpan firstRun  = Measure(() => action());
TimeSpan secondRun = Measure(() => action());
```

Вопрос:
Какой вариант быстрее и почему?

Вопрос:
Насколько велика разница?

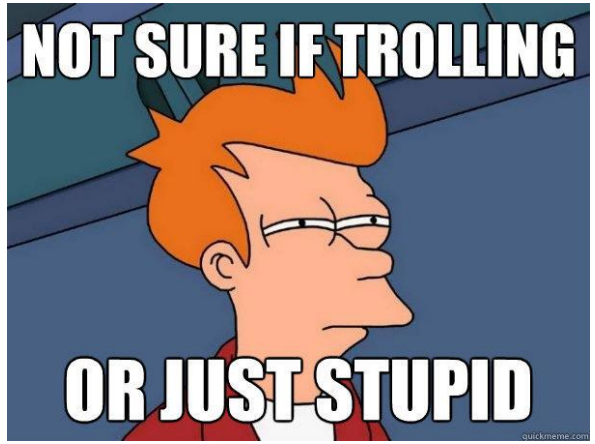
Simple hot loop

```
class Plain {
      public void Print() { }
}
```

```
for (int i = 0; i < length; i++)
      plain.Print();
```

```
for (int i = 0; i < length; i++)
      plain.Print();
```

Вопрос:
Какой вариант быстрее и почему?



When?

- Case: Hot small loops

µArch	JIT(s)	Variation (runs)
Sandy Bridge	Legacy x86 & x64	~ Small or zero
Sandy Bridge	RyuJIT	~ Small or zero
Haswell	Legacy x86 & x64	~ Small or zero
Haswell	RyuJIT	~30% (visible)

How to fix?

- (Semi)Manual loop unrolling

µArch	JIT(s)	Variation (runs)
Sandy Bridge	Legacy x86 & x64	~ Small or zero
Sandy Bridge	RyuJIT	~ Small or zero
Haswell	Legacy x86 & x64	~ Small or zero
Haswell	RyuJIT	~ Small or zero

JIT method inlining

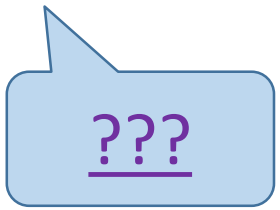
So you think you can optimize?

- Задача: написать реализацию метода Math.Abs(integer)

```
static int Abs(int value) {
     if (value < 0)
          return value * -1;
     return value;
}
```

vs

```
static int Abs(int value) {
     if (value < 0)
          value *= -1;
     return value;
}
```



So you think you can optimize?

- Задача: написать реализацию метода Math.Abs(integer)

```
static int Abs(int value) {
     if (value < 0)
         return value * -1;
     return value;
}
```

inlining

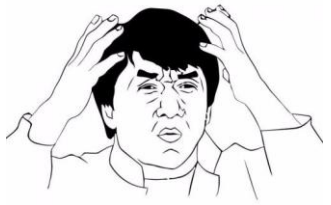
vs

```
static int Abs(int value) {
     if (value < 0)
         value *= -1;
     return value;
}
```

stdarg

CLR Inlining?!

- Inlining есть гуд!
- Но CLR автоматически инлайнит мало и то, если метод отвечает критериям:
  - Размер тела (IL) не более 32 байта
  - Отсутствие манипуляций аргументов (stdarg; или ref & out)
  - Сложный граф
  - И т.д.
- `MethodImplOptions.AggressiveInlining` может помочь, но не всегда



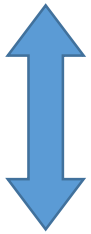
.NET methods trivia

.NET Methods & co.

- Non-virtual
- Static
- Instance
- Virtual (abstract)
- Generic methods
- Generic virtual methods (GVM)
- Properties (getter/setter)
- Events
- Delegates

.NET Methods structure

```
var formatter = new Formatter();
formatter.Format(someObj);
```

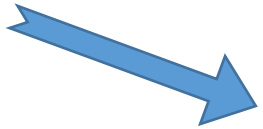


```
Formatter.Format(instance, someObj);
```

Вопрос:
равнозначны ли
данные методы?

CLR ABI (Application Binary Interface)

```
call(["this" pointer]
     [args…])
```



__fastcall

x64

```
; 4 registers
rcx
rdx
r8
r9
+ XMM0-XMM4
+
stack
```

x86

```
; 2 registers
ecx
edx
+ x87 FPU
+
stack
```

.NET non-virtual call

```
printer.Print();
```

```
mov     ecx,eax
call    00350480
```

Direct (прямой)
адрес метода

.NET Methods structure

```
; !Name2EE *!SampleNamespace.Printer
MethodTable: 001c4d6c
```

```
mov     ecx,eax
call    00350480
```

```
; !DumpMT -MD 001c4d6c
MethodDesc Table
   Entry MethodDe    JIT Name
736c2a98 733261fc PreJIT System.Object.ToString()
736c8930 73326204 PreJIT System.Object.Equals(System.Object)
736cce60 73326224 PreJIT System.Object.GetHashCode()
7361ec04 73326238 PreJIT System.Object.Finalize()
0035005d 001c4d64   NONE SampleNamespace.Printer..ctor()
00350480 001c4d58    JIT SampleNamespace.Printer.Print()
```

.NET Methods structure: Summary

- MethodTable включает **все** методы типа + доп. метаинформацию
- JIT использует адреса невиртуальных методов напрямую (direct method calls), т.е. в обход таблицы

Simple benchmark

```
RangeEnumerable enumerable  = Create(start, count);
foreach (var item in enumerable) {
    // do something
}
```

vs

```
IEnumerable<int> enumerable = Create(start, count);
foreach (var item in enumerable) {
    // do something
}
```

Direct vs. interface call: Benchmark

Method	Median	StdDev	Scaled	Scaled-SD
enumerator (direct)	2.5585 ns	0.0106 ns	1.00	0.01
enumerator (interface)	5.2967 ns	0.0576 ns	2.07	0.02
yield-based	6.2348 ns	0.0685 ns	2.44	0.03

Virtual Stub Dispatch

aka Interface dispatch stubs

Interface method call

```
ICallable target = new SomeImpl();

for (int i = 0; i < count; i++)
{
    target.DoSomething();
}
```

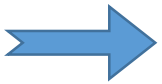
```
mov ecx,esi
call dword ptr ds:[22B0010h]
```

Вопрос:
Что вызывается? ☺

Polymorphic Stub

```
for (int i = 0; i < count; i++)
{
    callable.DoSomething(); // 3…2…
}
```

```
mov ecx,esi
call dword ptr ds:[22B0010h]
```

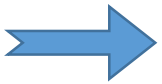


```
; 0x22B0010=022b6012
push eax
push 30000h
jmp clr!ResolveWorkerAsmStub
(73d3c8b0)
```

Polymorphic Stub update

```
for (int i = 0; i < count; i++)
{
    callable.DoSomething(); // 3…2…
}
```

```
mov ecx,esi
call dword ptr ds:[22B0010h]
```

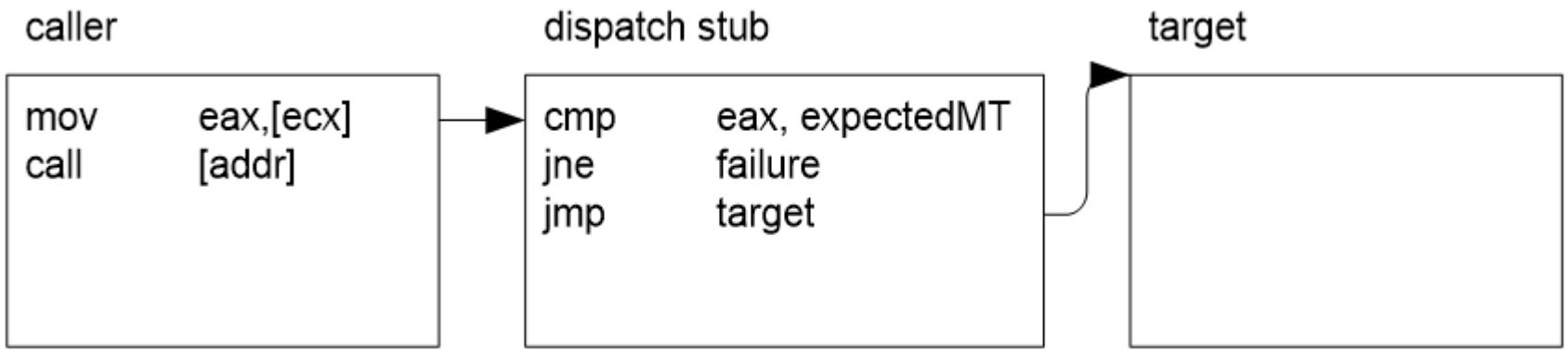


Упс:
Узел вызова (stub) изменился?

```
; 0x22B0010=022b7012
cmp dword ptr [ecx],974DDCh
jne 022ba011
jmp 025204d0
```

Monomorphic Stub

```
; 0x22B0010=022b7012
cmp dword ptr [ecx],974DDCh ; expectedMT=SomeImpl
jne 022ba011
jmp 025204d0
```



1. Virtual Stub Dispatch - https://github.com/dotnet/coreclr/blob/master/Documentation/botr/virtual-stub-dispatch.md

Monomorphic Stub: Benchmark

```
ICallable target = new SomeImpl();
for (int i = 0; i < count; i++)
    target.DoSomething();
```

Method	Median	StdDev	Scaled	Scaled-SD
non-virtual	2.0305 ns	0.0105 ns	1.00	0.00
monomorphic stub	2.7017 ns	0.0420 ns	1.34	0.02

Virtual Stub Dispatch: Notes

- CLR использует отличный от виртуальных методов подход для интерфейсов, кэширующий сами вызовы

- Полиморфный узел заменяется на мономорфный
- Обратная замена при n-ом количестве «промахов»

Virtual Method Call

Virtual Method Call

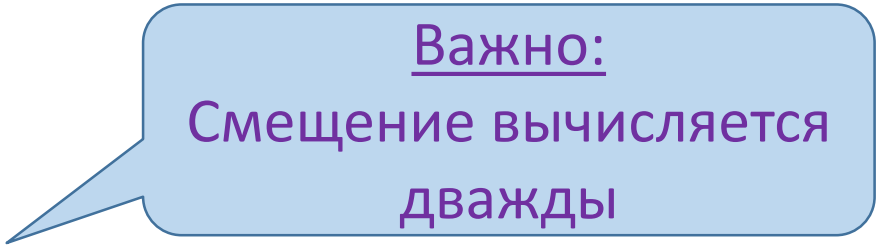
```
Base target = new Derived();
target.Print();
```

```
mov rax,qword ptr [rsi]
mov rax,qword ptr [rax+40h]
call qword ptr [rax+20h]
```

Вопрос:
Что именно вызывается? ☺

Virtual Method Call

```
Base target = new Derived();
target.Print();

mov rax,qword ptr [rsi]
mov rax,qword ptr [rax+40h]
call qword ptr [rax+20h]
; rax+20h=000007fe979200d8
```



Virtual Method Call

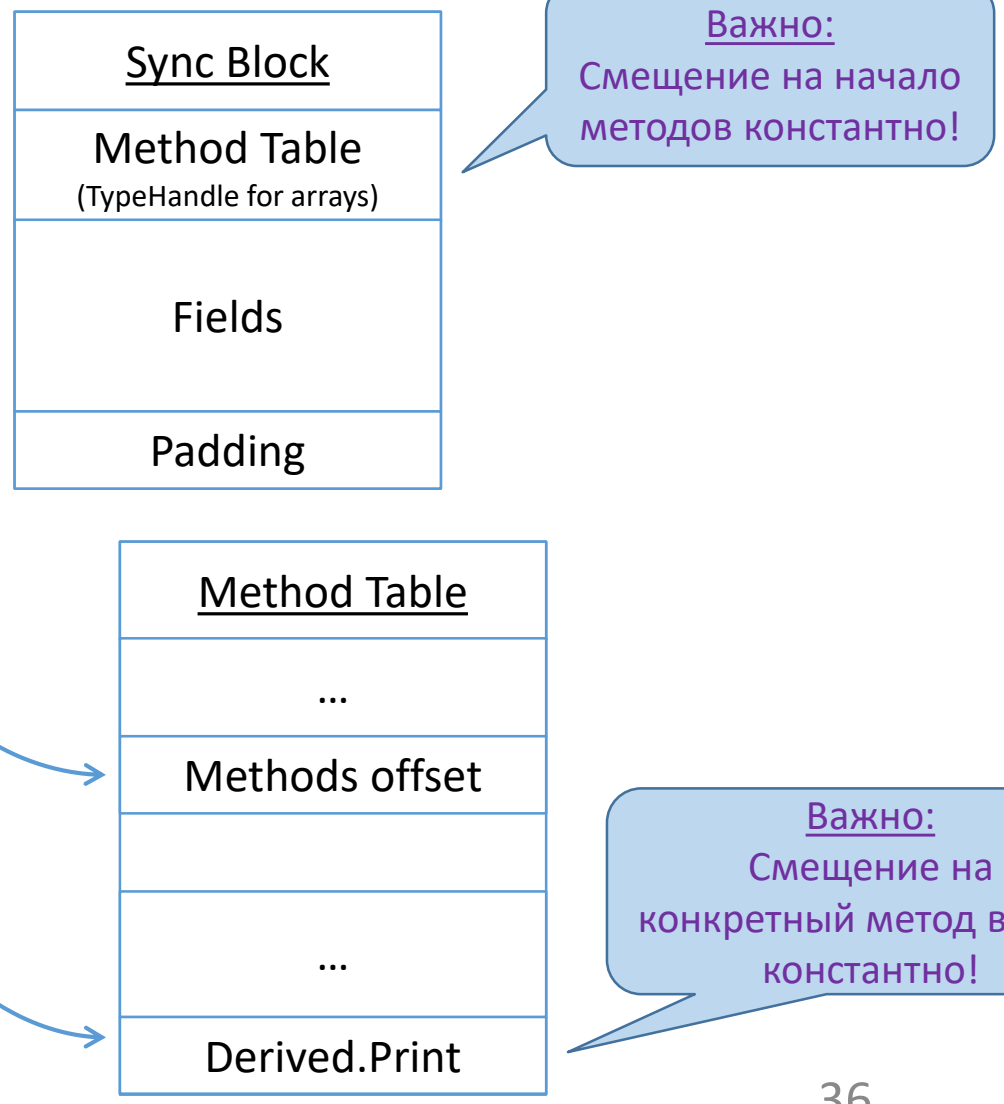
```
; rax+20h=000007fe979200d8
```

```
; !Name2EE *!VirtualStubSample.Derived
MethodTable: 000007fe97815b40
Name: VirtualStubSample.Derived
```

```
; !DumpMT -MD 000007fe97815b40
MethodDesc Table
Entry            MethodDesc       JIT    Name
000007fef5e21260 000007fef59c9288 PreJIT System.Object.ToString()
000007fef5ef0380 000007fef59c9290 PreJIT System.Object.Equals(System.Object)
000007fef5ef03c0 000007fef59c92b8 PreJIT System.Object.GetHashCode()
000007fef5f01d40 000007fef59c92d0 PreJIT System.Object.Finalize()
000007fe979200d8 000007fe97815b30 NONE   VirtualStubSample.Derived.Print()
000007fe979200e0 000007fe97815b38 NONE   VirtualStubSample.Derived..ctor()
```

Virtual Method Call

```
mov rax,qword ptr [rsi]
mov rax,qword ptr [rax+40h]
call qword ptr [rax+20h]
; rax+20h=000007fe979200d8
```



Virtual Method Call: Benchmark

```
Base target = new Derived();
for (int i = 0; i < count; i++)
    target.Print();
```

Method	Median	StdDev	Scaled	Scaled-SD
non-virtual	2.0305 ns	0.0105 ns	1.00	0.00
virtual	2.0283 ns	0.0189 ns	1.01	0.01

Virtual Method Call: Summary

- Диспетчеризация виртуальных методов производится через MethodTable
  - Возможно разделение таблицы на отдельную для виртуальных методов

- Смещение всегда константно => No-lookup
- Благодаря branch predictor'у современных CPU, разница между direct/indirect вызовами нивелируется

Generic method’s stubs

Generic method's stubs

```
new DataPrinter<Program>()
                .Print();
```

```
mov ecx,eax
call dword ptr ds:[14F10E4h]
```

ECX: указатель на **this**

```
new DataPrinter()
    .Print<Program>();
```

```
mov ecx,eax
mov edx,14F0FE0h
call dword ptr ds:[14F0FB4h]
```

Вопрос:
Какое значение в EDX?!

Generic method's stubs

```
mov ecx,eax
mov edx,14F0FE0h
call dword ptr ds:[14F0FB4h]
```

```
call clr!PrecodeFixupThunk (73d0eba8)
```

```
; !dumpmd 14F0FE0
Method Name: DataPrinter.Print[[System.Object, mscorlib]]()
```

Generic method's stubs: Benchmark

Method	Median	StdDev	Scaled	Scaled-SD
generic class	2.0249 ns	0.0755 ns	1.00	0.05
static generic method	2.0237 ns	0.0822 ns	1.00	0.05
generic method	2.0459 ns	0.0624 ns	1.00	0.04

Generic method’s stubs: Summary

- В CLR при наличии указателя this структура **MethodDesc** передается после (т.е. 2-м аргументом)

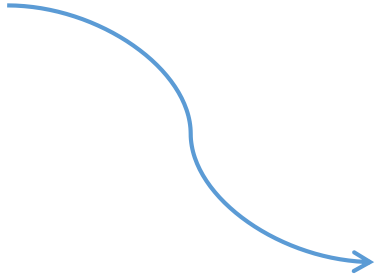
- **MethodDesc** содержит в себе указатель на MethodTable и служит источником метаданных для JIT'а

- Дескриптор хранит в себе все generic-параметры, т.е. экономия на количестве передаваемых аргументов.

Generic virtual methods

Generic virtual methods

```
public class Printer
{
    public virtual void Print<T>() {/* impl */ }
}

printer.Print<Console>();
```



```
push    184E20h
mov     ecx,esi
mov     edx,184D74h
call    clr!JIT_VirtualFunctionPointer
mov     ecx,esi
call    eax
```

GVM stubs

x86

```
push    184E20h
mov     ecx,esi
mov     edx,184D74h
call    clr!JIT_VirtualFunctionPointer
mov     ecx,esi
call    eax
```

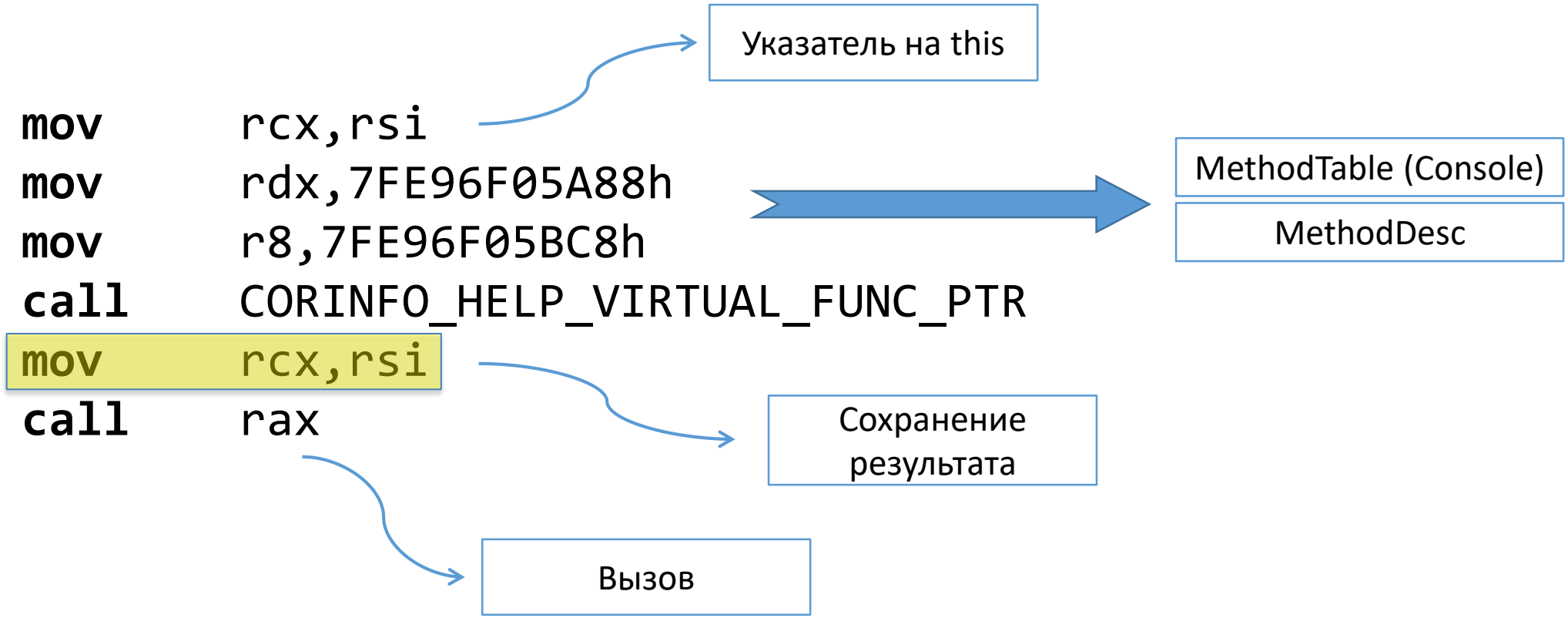
x64

```
mov     rcx,rsi
mov     rdx,7FE96F05A88h
mov     r8,7FE96F05BC8h
call    CORINFO_HELP_VIRTUAL_FUNC_PTR
mov     rcx,rsi
call    rax
```

```
FunctionPtr JIT_VirtualFunctionPointer(Object* objectUNSAFE,
                                       CORINFO_CLASS_HANDLE classHnd,
                                       CORINFO_METHOD_HANDLE methodHnd)
```

1. JIT_VirtualFunctionPointer - https://github.com/dotnet/coreclr/blob/32f0f9721afb584b4a14d69135bea7ddc129f755/src/vm/jithelpers.cpp#L4466

GVM Call


mov rcx,rsi
mov rdx,7FE96F05A88h
mov r8,7FE96F05BC8h
call CORINFO_HELP_VIRTUAL_FUNC_PTR
mov rcx,rsi
call rax
Указатель на this
MethodTable (Console)
MethodDesc
Сохранение результата
Вызов

GVM Call: Benchmark

Method	Median	StdDev	Scaled	Scaled-SD
generic method	2.0459 ns	0.0624 ns	1.00	0.04
generic virtual method	15.6322 ns	0.1841 ns	7.60	0.26

GVM: Summary

- Каждый вызов GVM сопровождается обращением к среде исполнения

- На данный момент CLR не имеет подобного VSD (как у интерфейсов) механизма для GVM

GC & JIT

cooperation

Simple stack

```
public class Stack
{
    public void Push(int item)
    {
        Node node = new Node(item);
        node._next = head;
        _head = node;
    }

    public class Node
    {
        // Constructor
        public Node _next;
        public int _value;
    }
}
```

- Стек можно реализовать в виде однонаправленного списка
  - Т.е. каждый элемент списка указывает только на следующий

Simple stack: Unsafe+structs

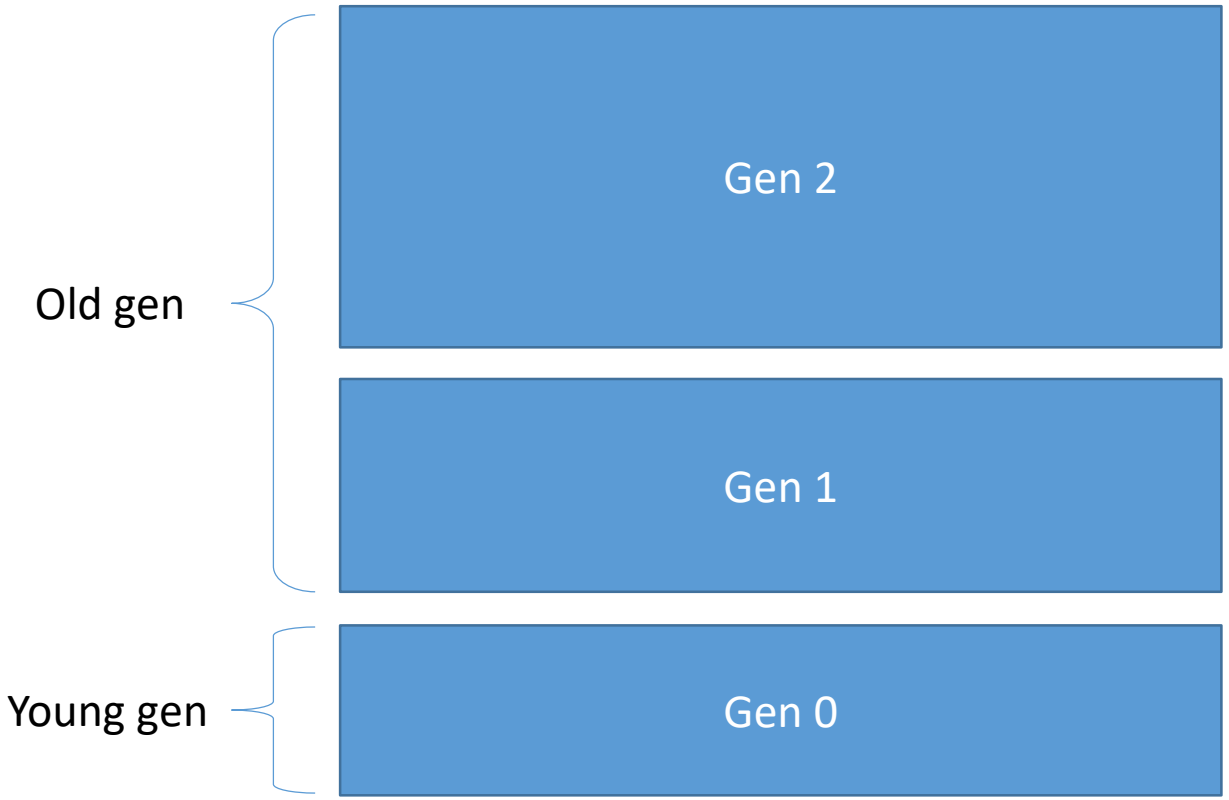
```
public class ValueStack
{
    public void Push(int item)
    {
        Node* node = alloc(item);
        node._next = head;
        _head = node;
    }

    public unsafe struct Node
    {
        // Constructor
        public Node* _next;
        public int _value;
    }
}
```

- alloc – метод аллоцирующий память
  - Marshal.AllocHGlobal

Field assignment: Benchmark

Method	Median	StdDev	Scaled	Scaled-SD
unsafe struct node (preallocated)	4.9566 ns	0.0015 ns	1.00	0.00
class-based node (preallocated array)	8.2541 ns	0.0043 ns	1.67	0.01
class-based node (constructor)	13.4473 ns	0.0185 ns	2.73	0.04
unsafe struct node (alloc)	73.5886 ns	0.0148 ns	14.84	0.11

Generational GC scheme

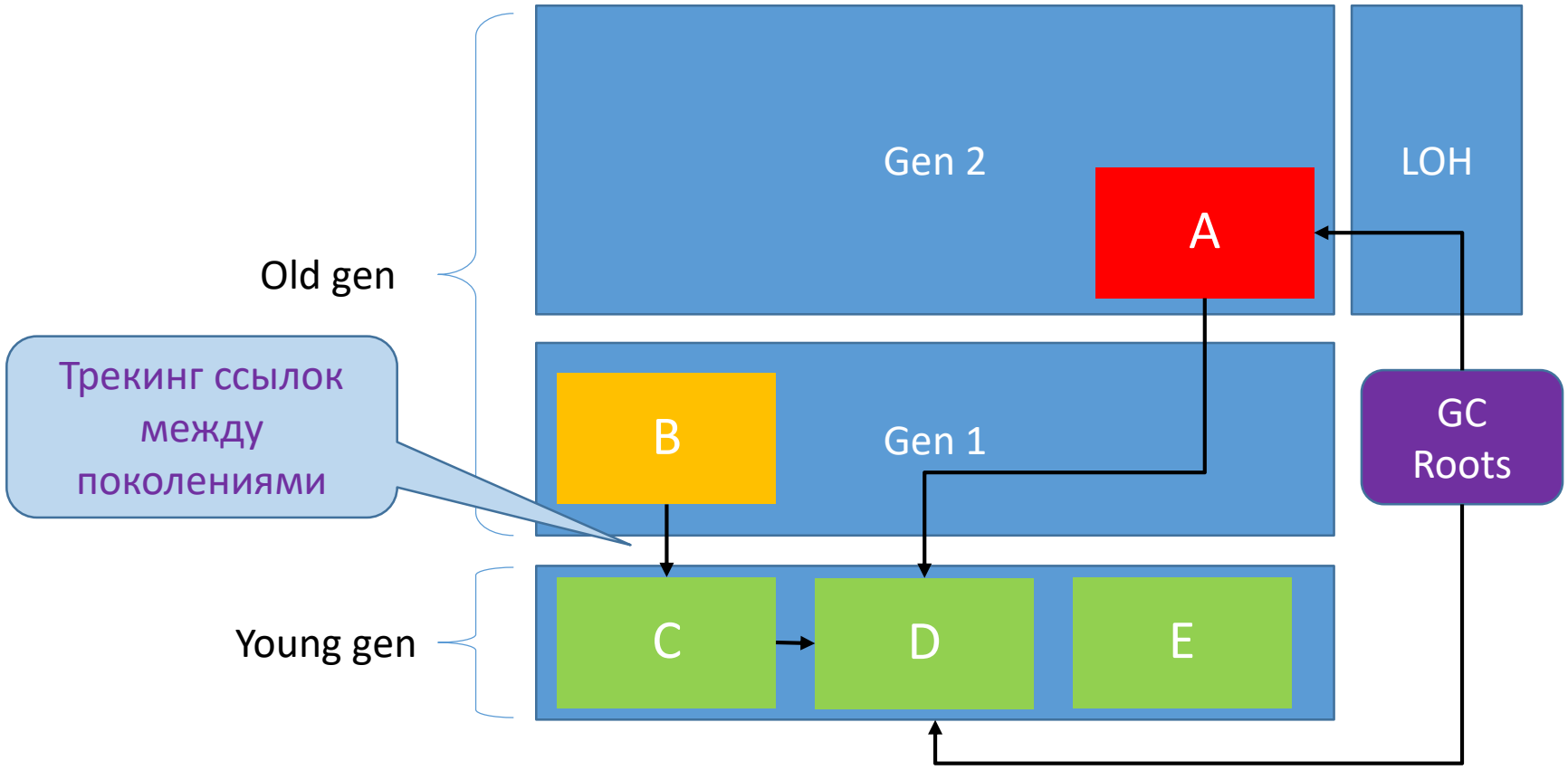


Most allocated objects will die young[1]

Few references from older to younger objects exist[1]

1. Garbage Collector Basics and Performance Hints - https://msdn.microsoft.com/en-us/library/ms973837.aspx
2. Garbage Collection in the Java HotSpot Virtual Machine - http://www.devx.com/Java/Article/21977

Generational GC scheme



- Регистры
- Стек
- Глобальные объекты
- Куча (Heap)

Reference assignment

```
[MethodImpl(MethodImplOptions.NoInlining)]
static void SetFieldReference(DataObject dto, object obj)
{
    dto.Val = obj; // breakpoint
}

class DataObject
{
    public object Val;
}
```

Вопрос:
Что в этом коде необычного?

```
; SetFieldReference
push ebp
mov ebp,esp
mov eax,edx
lea edx,[ecx+4]
call 73D0E660
pop ebp
ret
```

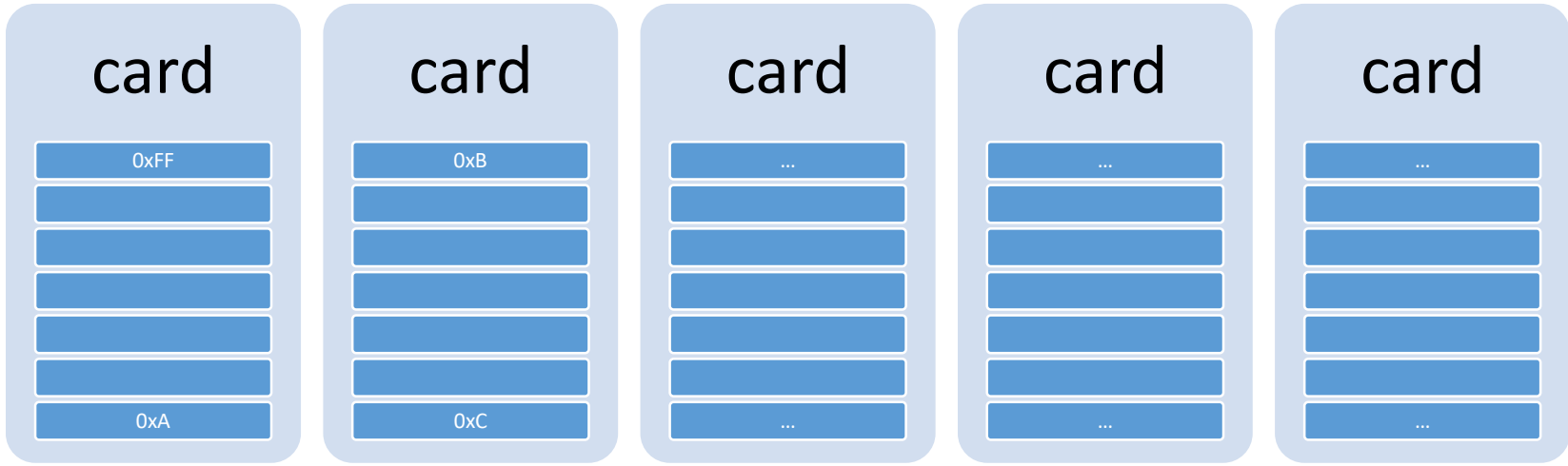
Generational GC: Card Table

- Чтобы избежать сканирования всей кучи, GC необходимо знать какие участки памяти (т.е. объекты или группы объектов) можно **пропустить**.

- Структурой данных, содержащей всю необходимую информацию, является **card table**

Generational GC: Card Table

- Каждая «карта» покрывает свой диапазон памяти



Card table in action

```
static void SetFieldReference(DataObject dto, object obj) {
    dto.Val = obj; // breakpoint
}
```

```
; SetFieldReference
push ebp
mov ebp,esp
mov eax,edx
lea edx,[ecx+4]
call 73D0E660
pop ebp
ret
```

```
; SetFieldReference (with Symbols)
push ebp
mov ebp,esp
mov eax,edx
lea edx,[ecx+4]
call clr!JIT_WriteBarrierEAX
(73d0e660)
pop ebp
ret
```

Write barrier

```
; clr!JIT_WriteBarrierEAX (73d0e660)
mov dword ptr [edx],eax
cmp eax,296100Ch
jb clr!JIT_WriteBarrierEAX+0x17
(73d0e677)
shr edx,0Ah
nop
cmp byte ptr [edx+26D5AA0h],0FFh
jne clr!JIT_WriteBarrierEAX+0x1a
(73d0e67a)
ret
```

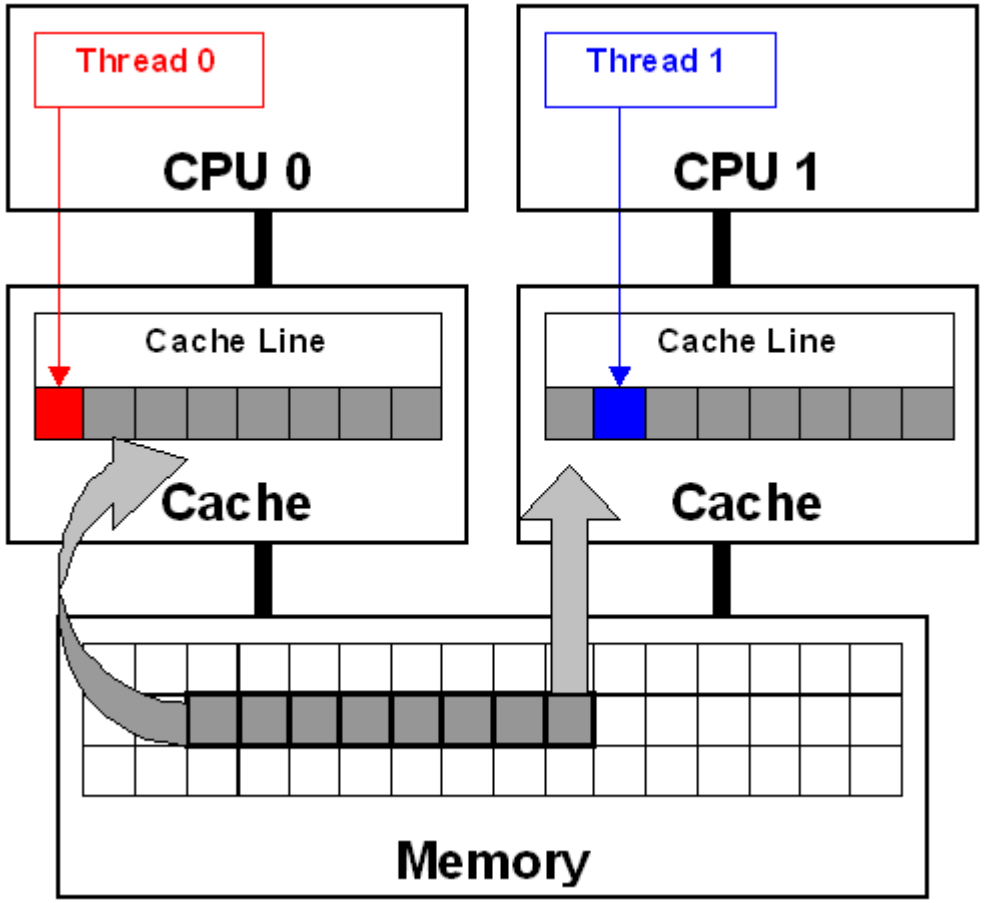
```
; clr!JIT_WriteBarrierEAX+0x17 (73d0e677)
ret
```

```
; clr!JIT_WriteBarrierEAX+0x1a (73d0e67a)
mov byte ptr [edx+26D5AA0h],0FFh
```

```
if (cardTable[offset] != 0)
    cardTable[offset] = 0;
```

Вопрос:
Почему мы проверяем перед маркировкой?

False sharing



Arch	Cache Line size (in bytes)
ARM (32-bit)	32
x86, SPARC	64
ARM (64-bit, modern)	64
PowerPC	128

```
if (cardTable[offset] != 0)
    cardTable[offset] = 0;
```

Меньше нагрузка на подсистему памяти

1. Reduce False Sharing in .NET - https://software.intel.com/sites/default/files/m/3/4/d/7/e/29393-218129_218129.pdf

Write barrier offset

x86

```
; clr!JIT_WriteBarrierEAX (73d0e660)
mov dword ptr [edx],eax
cmp eax,296100Ch
jb clr!JIT_WriteBarrierEAX+0x17
(73d0e677)
shr edx,0Ah
nop
cmp byte ptr [edx+26D5AA0h],0FFh
jne clr!JIT_WriteBarrierEAX+0x1a
(73d0e67a)
ret
```

Сдвиг вправо на 10 бит

x64

```
; JitHelp: CORINFO_HELP_ASSIGN_REF
mov qword ptr [rcx],rdx
nop dword ptr [rax]
mov rax,21E038C1018h
cmp rdx,rax
jb clr+0x3e70
nop
mov rax,21DD7CB8EC0h
shr rcx,0Bh
cmp byte ptr [rax+rcx],0FFh
jne clr+0x3e5c (00007ff9`22f93e5c)
```

Сдвиг вправо на 11 бит

CLR Card table

- Card table – битовая маска (или массив битов), где **1 байт** покрывает **1 KB** в **32-битной системе**.

x86

```
; 2^10 = 1024
shr edx, 0Ah
```

x64

```
; 2^11 = 2048
shr rcx, 0Bh
```

CLR Card table: Cards size

- Card table – битовая маска (или массив битов), где **1 байт** покрывает **1 KB** в **32-битной системе**.

x86

```
; 2^10 = 1024
shr edx, 0Ah
```

x64

```
; 2^11 = 2048
shr rcx, 0Bh
```

Система	Диапазон памяти	Мин. кол-во объектов
32-bit	1 KB	~85
64-bit	2 KB	~85

Write barriers: Notes

- Присваивание полей ссылочного типа в языках с GC, основанном на поколениях – **не бесплатно**!
- CLR пытается минимизировать свое влияние на работу пользовательских приложений
  - см. False sharing prevention
- Write barrier – не уникальная для CLR абстракция. Активно используется практически в любой современной управляемой среде. Например:
  - Java (HotSpot)
  - Mozilla SpiderMonkey & co.
  - Google V8

CLR GC: Summary

- CLR – Tracing Precise Generational GC
- Наиболее распространенный вид сборщиков мусора
- Требует отслеживания объектов (время жизни)
  - => Кооперация с JIT-ом
- При использовании стратегии «поколений» (aka Generational GC) дает преимущество в скорости обхода управляемой кучи

Epilogue

- RyuJIT имеет нюансы
- Inlining есть гуд, но не всегда применим
- Виртуальные методы не страшны
- Интерфейсами злоупотреблять не надо
- Присваивание полей ссылочного типа не бесплатно

Спасибо!

Q&A

Контакты

- Email: szkarlen@gmail.com, KSimonyan@luxoft.com
- Twitter, Habr, GitHub: @szKarlen