7 Бенчмарки, ограниченные возможностями процессора

Тук-тук.

Прогнозирование ветвления.

Кто там?

Классическая шутка программистов

Одно из самых распространенных мест заторов в бенчмарках — это центральный процессор. Правильная разработка и анализ бенчмарков, ограниченных возможностями процессора, требуют знания различных характеристик среды исполнения и устройств, способных повлиять на производительность. В любой среде исполнения .NET есть множество оптимизаций, которые могут улучшить или ухудшить производительность вашего кода. У каждой микроархитектуры процессора много низкоуровневых механизмов, которые также способны повлиять на результаты измерений. Не зная об этих оптимизациях и механизмах, сложно разрабатывать некоторые из бенчмарков и правильно интерпретировать метрики. В этой главе рассмотрим следующие темы.

- **Регистры и стек.**

  Мы обсудим, когда компилятор JIT хранит промежуточные значения в регистрах, а когда в стеке.

- **Инлайнинг.**

  Поговорим о том, когда компилятор JIT может заинлайнить ваши методы и почему это имеет большое значение.

- **Параллелизм на уровне команд (ILP).**

  Рассмотрим одну из важнейших характеристик аппаратных средств — ILP, позволяющую обрабатывать несколько команд одновременно в одном потоке.

- **Прогнозирование ветвлений.**

  Обсудим способность современных процессоров прогнозировать, какие ветви будут включены в программы. Правильные прогнозы помогают улучшать

условия ILP. Это важно для бенчмаркинга, поскольку входные данные могут заметно повлиять на производительность метода, основываясь на количестве правильных прогнозов.

- **Арифметика.**

  Мы расскажем, какие проблемы могут возникнуть с бенчмарками, использующими арифметические операции. Обсудим функции аппаратных средств (числа с плавающей запятой и IEEE 754) и среды исполнения (разные окружения и оптимизации JIT).

- **Интринзики.**

  Рассмотрим случаи, когда компилятор JIT может сгенерировать разумную собственную реализацию конкретных методов и выражений.

Полные описания каждой из тем довольно объемны, потому что включают в себя множество подробностей о внутренних составляющих аппаратных средств и среды исполнения. Однако при бенчмаркинге вам не обязательно знать все эти внутренние составляющие. В этой главе мы обсудим только высокоуровневые концепции, которые полезно знать. В каждом разделе рассмотрены четыре практических примера, демонстрирующих влияние этих концепций даже на небольшие простые бенчмарки. Во всех примерах по четыре раздела.

- **Исходный код.**

  Небольшой набор бенчмарков, показывающий интересное воздействие на производительность. Исходный код всех примеров можно найти в приложении к книге.

- **Результаты.**

  Результаты бенчмарка в *конкретном* окружении. Если вы не можете воспроизвести этот результат на своем устройстве, проверьте версии вашей ОС, .NET Core, .NET Core SDK, среды исполнения, компилятора JIT и модель процессора. Производительность всегда зависит от окружения: что угодно может испортить описанный феномен в области производительности или вызвать еще один.

- **Объяснение.**

  Краткое описание наблюдаемых результатов. Мы часто будем изучать сгенерированный код на промежуточном языке и собственный код, чтобы понять, что происходит в примере.

- **Обсуждение.**

  Общие рекомендации по поводу рассматриваемых эффектов, дополнительная интересная информация, ссылки на вопросы на GitHub и другие источники для дальнейшего изучения. Многие практические примеры основаны на замечательных вопросах и ответах с StackOverflow, соответствующие ссылки приведены в конце раздела.

Вы узнаете о самых распространенных ошибках, совершаемых разработчиками из-за незнания каких-то подводных камней бенчмаркинга. Это поможет вам разработать более качественные бенчмарки, ограниченные возможностями процессора, и правильно интерпретировать их результаты.

Регистры и стек

Если у нас есть локальная переменная, компилятор JIT может поместить ее в регистры или стек. Операции с регистрами обычно гораздо быстрее операций со значениями из стека. Таким образом, решение компилятора JIT может заметно повлиять на производительность. Невозможно хранить все локальные переменные в регистре, поскольку количество регистров ограничено и компилятор JIT должен применять их разумно. В разных наборах команд процессора различное количество регистров.

Практический пример 1: продвижение структуры

Во многих случаях при использовании в локальных переменных значения структурного типа компилятор JIT сохраняет его поля в стеке. В некоторых особых случаях они могут быть сохранены в регистре. Этот подход известен как *продвижение структуры* или *скалярная замена*. Он применяется в RyuJIT, но вручную включить или отключить эту функцию для конкретного метода нельзя. Ознакомимся с примером, показывающим ограничения продвижения структуры.

Исходный код

Рассмотрим бенчмарк на базе BenchmarkDotNet:

```
public struct Struct3
{
  public byte X0, X1, X2;
}

public struct Struct8
{
  public byte X0, X1, X2, X3, X4, X5, X6, X7;
}

public class Benchmarks
{
  public const int Size = 256;

  private int[] sum = new int[Size];
  private Struct3[] struct3 = new Struct3[Size];
  private Struct8[] struct8 = new Struct8[Size];
```

```
  [Benchmark(OperationsPerInvoke = Size, Baseline = true)]
  public void Run3()
  {
    for (var i = 0; i < sum.Length; i++)
    {
      Struct3 s = struct3[i];
      sum[i] = s.X0 + s.X1;
    }
  }

  [Benchmark(OperationsPerInvoke = Size)]
  public void Run8()
  {
    for (var i = 0; i < sum.Length; i++)
    {
      Struct8 s = struct8[i];
      sum[i] = s.X0 + s.X1;
    }
  }
}
```

Здесь две структуры: `Struct3` с тремя полями `byte` и `Struct8` с восемью полями `byte`. У нас два бенчмарка, `Run3` и `Run8`. В каждом из них вычисляем сумму первых двух структурных полей в цикле. Единственное различие между `Run3` и `Run8` заключается в используемой структуре.

Результаты

Вот пример результатов (Windows 10.0.17763.195, Intel Core i7-6700HQ CPU 2.60GHz, .NET Core 2.1.5, RyuJIT-x64):

```
  Method |     Mean |    StdDev | Ratio |
-------- |---------:|----------:|------:|
    Run3 | 1.100 ns | 0.0091 ns |  1.00 |
    Run8 | 1.579 ns | 0.0115 ns |  1.44 |
```

Как видите, `Run8` работает гораздо медленнее. `Run8` использует `Struct8`, которая похожа на `Struct3`, но содержит на пять полей больше. Эти поля на самом деле не используются в бенчмарке, но мы все равно получаем падение производительности на ~ 30–50 %.

Объяснение

Посмотрим на собственный код тела цикла `Run3`:

```
; Run3
lea r8,[r8+r10+10h]               ; r8 = &struct3[i]
movzx r10d,byte ptr [r8]          ; r10d = X0
movzx r11d,byte ptr [r8+1]        ; r11d = X1
movzx r8d,byte ptr [r8+2]         ; r8d = X2
```

```
mov r8,rdx                          ; r8 = &sum
cmp eax,dword ptr [r8+8]            ; if (i > sum.Length)
jae 00007ffe`e62a2b74               ; throw
add r10d,r11d                       ; r10d += r11d
mov dword ptr [r8+r9*4+10h],r10d    ; sum[i] = r10d
```

Как видите, мы находим локацию `struct3[i]` и загружаем три поля, `X0`, `X1` и `X2`, в регистры `r10d`, `r11d` и `r8d` соответственно. Так работает продвижение структуры! Нам не нужно поле `X2`, но JIT по умолчанию загружает все поля. Затем мы складываем `r11d` с `r10d` и сохраняем результат в `sum[i]`.

Теперь посмотрим на собственный код тела цикла `Run8`:

```
; Run8
mov     rdx,qword ptr [rdx+r8*8+10h]   ; rdx = struct8[i]
mov     qword ptr [rsp+20h],rdx        ; [rsp+20h] = struct8[i]
mov     rdx,qword ptr [rcx+8]          ; rdx = &sum
mov     r9,rdx                         ; r9 = &sum
cmp         eax,dword ptr [r9+8]       ; if (i > sum.Length)
jae 00007ffe`e6272b7a                  ; throw
movzx   r10d,byte ptr [rsp+20h]        ; r10d = X0
movzx   r11d,byte ptr [rsp+21h]        ; r11d = X1
add     r10d,r11d                      ; r10d += r11d
mov     dword ptr [r9+r8*4+10h],r10d   ; sum[i] = r10d
```

Здесь мы сначала загружаем `struct8[i]` в стек. После этого загружаем первые два поля `struct8[i]` из стека в регистры `r10d` и `r11d`. Затем складываем `r11d` с `r10d` и сохраняем результат в `sum[i]`.

Как видите, RyuJIT смог применить продвижение структуры в `Run3` и не смог — в `Run8`. Этот результат можно объяснить ограничением RyuJIT в .NET Core 2.1: он не может продвигать структуры, в которых больше четырех полей.

Обсуждение

В .NET Core 1.x/2.x у реализации скалярного замещения много ограничений. Например, продвигаемая структура должна соответствовать следующим требованиям (https://github.com/dotnet/coreclr/issues/6733#issuecomment-240623400).

- В ней должны быть только непроизводные поля.
- Она не должна быть аргументом или возвращенным значением, передающимся в регистрах.
- Она не может быть больше 32 байт.
- В ней не может быть больше четырех полей.

В общем-то, не рекомендуется при оптимизации полагаться на эти конкретные эвристические правила, поскольку они могут измениться в будущих версиях RyuJIT. Также они недействительны при использовании других компиляторов JIT,

таких как LegacyJIT-x64 или MonoJIT. Но если вам действительно нужно оптимизировать какие-нибудь горячие методы и версия .NET Core исправлена, можете применить эту информацию, но имеет смысл проверять такие оптимизации после каждого обновления .NET Core (это можно автоматизировать с помощью тестов производительности).

Мы обсудили этот пример, поскольку знание концепции продвижения структур помогает правильно интерпретировать результаты работы бенчмарков. При разработке небольшого бенчмарка, основанного на реальном приложении, не рекомендуется изменять используемые структуры, даже если какие-то из их полей в бенчмарке не задействуются. Любые изменения в составе структур могут повлечь за собой непредсказуемые изменения производительности.

В этом конкретном бенчмарке есть также интересные проблемы с производительностью, связанные с памятью. Обсудим их в главе 8.

См. также:

- coreclr#6839 *Promote (scalar replace) structs with more than fields* (https://github.com/dotnet/coreclr/issues/6839);
- coreclr#6733 *Scalar replacement of aggregates* (https://github.com/dotnet/coreclr/issues/6733);
- документы по разработке CoreCLR: *First Class Structs* (https://github.com/dotnet/coreclr/blob/v2.2.0/Documentation/design-docs/firstclass-structs.md).

Этот практический пример основан на вопросе 38949304 на StackOverflow (https://stackoverflow.com/q/38949304).

Практический пример 2: локальные переменные

Ввод локальной переменной — распространенное перестроение, которое может сделать ваш код более понятным. Эта модификация кода не меняет логику, поэтому разработчики не ожидают того, что перестроение повлияет на производительность приложения. Однако *любые изменения* в исходном коде могут изменить производительность.

Исходный код

Рассмотрим следующие бенчмарки на базе BenchmarkDotNet:

```
public struct Struct
{
  public Struct(uint? someValue)
  {
    SomeValue = someValue;
  }
```

```
  public uint? SomeValue { get; }
}

public class Benchmarks
{
  [Benchmark(Baseline = true)]
  public uint? Foo()
  {
    return new Struct(100).SomeValue;
  }

  [Benchmark]
  public uint? Bar()
  {
    Struct s = new Struct(100);
    return s.SomeValue;
  }
}
```

У нас есть два бенчмарка, `Foo` и `Bar`. Оба метода делают одно и то же — создают копию структуры `Struct` (оберточного кода, значимого для типа `uint?`) и возвращают ее единственное поле. При этом `Bar` отличается от `Foo`: он сохраняет копию структуры в локальную переменную вместо того, чтобы использовать ее в возвращаемом выражении. Исполняемая логика в обоих случаях идентична, но есть небольшие изменения на уровне C#. Обычно при таком простом перестроении кода мы не ожидаем изменения производительности.

Результаты

Вот пример результатов (Windows 10.0.17763.195, Intel Core i7-6700HQ CPU 2.60GHz, .NET Core SDK 2.1.403, .NET Core 2.1.5, RyuJIT-x64):

```
  Method |     Mean |    StdDev | Ratio |
-------- |---------:|----------:|------:|
     Foo | 6.597 ns | 0.0433 ns |  1.00 |
     Bar | 4.975 ns | 0.0439 ns |  0.75 |
```

Как видите, `Bar` работает на ~ 20–30 % быстрее, чем `Foo`. Как такое возможно?

Объяснение

Посмотрим на сгенерированный код на промежуточном языке (Roslyn 2.9.0.63127):

```
// Foo()
.maxstack 1
.locals init (
  [0] valuetype Struct V_0
)
IL_00: ldc.i4.s   100
IL_02: newobj     System.Void System.Nullable`1::.ctor(!0)
IL_07: newobj     System.Void Struct::.ctor(System.Nullable`1)
```

```
IL_0c: stloc.0    // V_0
IL_0d: ldloca.s   V_0
IL_0f: call       System.Nullable`1 Struct::get_SomeValue()
IL_14: ret

// Bar()
.maxstack 2
.locals /*11000001*/ init (
  [0] valuetype Struct s
)
IL_00: ldloca.s   s
IL_02: ldc.i4.s   100
IL_04: newobj     System.Void System.Nullable`1::.ctor(!0)
IL_09: call       System.Void Struct::.ctor(System.Nullable`1)
IL_0e: ldloca.s   s
IL_10: call       System.Nullable`1 Struct::get_SomeValue()
IL_15: ret
```

Как видите, между этими методами есть некоторые различия. `Foo` создает `Struct` с помощью `newobj`, загружает результат в локальную переменную и затем загружает адрес этой переменной. А `Bar` создает `Struct` с помощью `call`, который сохраняет результат в локальную переменную, и мгновенно загружает адрес этой переменной. Обе реализации эквивалентны друг другу, но используют разные команды на промежуточном языке.

Теперь посмотрим на сгенерированный собственный код `Foo()`:

```
; Foo()
sub   rsp,18h
xor   eax,eax;                  ; Initialize Struct
mov   qword ptr [rsp+10h],rax   ; Store Struct into stack

mov   eax,64h                   ; eax = 100
mov    edx,1                    ; edx = 1

xor   ecx,ecx;                  ; Initialize SomeValue
mov   qword ptr [rsp+8],rcx     ; Store SomeValue into stack
lea   rcx,[rsp+8]               ; rcx = pointer to SomeValue
mov   byte ptr [rcx],dl         ; SomeValue.HasValue = 1
mov   dword ptr [rcx+4],eax     ; SomeValue.Value = 100

mov   rax,qword ptr [rsp+8]     ; rax = pointer to SomeValue
mov   qword ptr [rsp+10h],rax   ; Store SomeValue to a different location on stack
mov   rax,qword ptr [rsp+10h]   ; rax = pointer to SomeValue

add   rsp,18h
ret
```

Как видите, пара `stloc.0/ldloca.s` заставляет RyuJIT генерировать избыточные команды `mov`. А вот собственный код `Bar`:

```
; Bar()
push   rax
xor    eax,eax                ; Initialize Struct
mov    qword ptr [rsp],rax    ; Store Struct into stack
mov    eax,64h                ; eax = 100
mov    edx,1                  ; edx = 1

lea    rcx,[rsp]              ; rcx = pointer to SomeValue
mov    byte ptr [rcx],dl      ; SomeValue.HasValue = 1
mov    dword ptr [rcx+4],eax  ; SomeValue.Value = 100
mov    rax,qword ptr [rsp]    ; rax = pointer to SomeValue

add    rsp,8
ret
```

Он выглядит более эффективным, так как не содержит избыточных команд.

Обсуждение

Любое перестроение кода, не меняющее логики, может изменить сгенерированный код на промежуточном языке. Любые изменения в коде на промежуточном языке могут непредсказуемо повлиять на эффективность сгенерированного кода. Когда разработчики создают бенчмарки, основанные на сценариях из реальной жизни, они часто делают небольшие перестроения, чтобы бенчмарк проще было понять. К сожалению, такие перестроения могут дополнительно воздействовать на производительность и ухудшить (или улучшить) ее. При перестроении в существующем бенчмарке рекомендуется убедиться, что сделанные вами изменения в коде не влияют на результаты.

В подобных случаях производительность зависит от версии компилятора. Предыдущий пример достоверен для Roslyn 2.9.0.63127 (поставляется в пакете .NET Core SDK 2.1.403), но это может измениться в будущих версиях (см. подробности в roslyn#30284 (https://github.com/dotnet/roslyn/issues/30284)).

Ситуации, когда небольшие изменения в исходном коде дают интересные эффекты в области производительности, встречаются часто. Например, в вопросе 53452713 на StackOverflow (https://stackoverflow.com/q/53452713) представлен простой бенчмарк на Java, ускоряющийся после замены `2 * i * i` на `2 * (i * i)`.

Этот практический пример основан на вопросе 52565479 на StackOverflow (https://stackoverflow.com/q/52565479).

Практический пример 3: попытка-перехват

Правильно работать с исключениями очень важно, если вы хотите разрабатывать стабильные приложения на .NET. Многие разработчики на всякий случай помещают там и сям блоки «попытка-перехват». Они не ожидают снижения производительности, поскольку исключения считаются редкими. Может показаться, что если исходный код не выдает исключений, ограничения из-за блоков «попытка-перехват» будут незаметными. К сожалению, так происходит не всегда, поскольку компилятор JIT может изменить сгенерированный собственный код при добавлении такого блока.

Исходный код

Рассмотрим следующие бенчмарки на базе BenchmarkDotNet:

```
public class Benchmarks
{
  private const int N = 93;

  [Benchmark(Baseline = true)]
  public long Fibonacci1()
  {
    long a = 0, b = 0, c = 1;
    for (int i = 1; i < N; i++)
    {
      a = b;
      b = c;
      c = a + b;
    }
    return c;
  }

  [Benchmark]
  public long Fibonacci2()
  {
    long a = 0, b = 0, c = 1;
    try
    {
      for (int i = 1; i < N; i++)
      {
        a = b;
        b = c;
        c = a + b;
      }
    }
    catch {}
    return c;
  }
}
```

У нас есть два метода, `Fibonacci1` и `Fibonacci2`, которые вычисляют 93-е число Фибоначчи[1]. Однако `Fibonacci2` заканчивает основной цикл блоком «попытка-перехват». Этот код не выдает исключений, поэтому, наверное, нам не стоит ожидать ограничений производительности?

Результаты

Вот пример результатов (Windows 10.0.17763.195, Intel Core i7-6700HQ CPU 2.60GHz, .NET Core 2.1.5, RyuJIT-x64):

```
     Method |      Mean |    StdDev | Ratio |
----------- |----------:|----------:|------:|
 Fibonacci1 |  41.07 ns | 0.1446 ns |  1.00 |
 Fibonacci2 | 102.93 ns | 0.3394 ns |  2.50 |
```

В этом окружении `Fibonacci2` работает в 2,5 раза медленнее, чем `Fibonacci1`.

Объяснение

Посмотрим на сгенерированный собственный код `Fibonacci1`:

```
; Fibonacci1
xor   eax,eax        ; a = 0
mov   edx,1          ; c = 1
mov   ecx,1          ; i = 1
LOOP:
mov   r8,rdx         ; b = c
lea   rdx,[rax+r8]   ; c = a + b
inc   ecx            ; i++
cmp   ecx,5Dh        ; if (i < 93)
mov   rax,r8         ; a = b
jl    LOOP           ; goto LOOP
mov   rax,rdx        ; result = c

ret                  ; return result
```

Эта реализация довольно проста. Переменные `a`, `b` и `c` представлены с помощью регистров `rax`, `r8` и `rdx`.

Теперь посмотрим на сгенерированный собственный код `Fibonacci2`:

```
; Fibonacci2
sub   esp,10h                   ; Reserve space
lea   rbp,[rsp+10h]             ; on stack
mov   qword ptr [rbp-10h],rsp   ;
```

[1] 12 200 160 415 121 876 738. Это самое длинное число Фибоначчи, которое можно представить с помощью long.

```
xor   eax,eax                  ; a = 0
mov   qword ptr [rbp-8],1      ; c = 1
mov   edx,1                    ; i = 1
LOOP:
mov   rcx,qword ptr [rbp-8]    ; b = c
add   rax,rcx                  ; a += b
mov   qword ptr [rbp-8],rax    ; c = a
inc   edx                      ; i++
cmp   edx,5Dh                  ; if (i < 93)
mov   rax,rcx                  ; a = b
jl    LOOP                     ; goto LOOP
mov   rax,qword ptr [rbp-8]    ; result = c

lea   rsp,[rbp]                ; Recover stack pointer
pop   rbp                      ;
ret                            ; return result
```

Переменные `a` и `b` по-прежнему используют регистры `rax` и `rcx`. При этом переменная `c` размещается не в регистрах, а в стеке (`qword ptr [rbp-8]`). `Fibonacci2` работает намного медленнее `Fibonacci1`, поскольку операции чтения и записи со значениями из стека занимают гораздо больше времени, чем операции с регистрами.

Единственным различием между `Fibonacci1` и `Fibonacci2` является блок «попытка-перехват» в `Fibonacci2`. У нас нет собственных инструкций по работе с исключениями, поскольку `Fibonacci2` не выдает исключений. Однако само существование этого блока заставило RuyJIT поместить переменную `c` в стек, что ухудшило производительность метода.

Обсуждение

Этот пример основан на вопросе 8928403 на StackOverflow (https://stackoverflow.com/q/8928403). Его автор спрашивает, почему метод с блоком «попытка-перехват» работает *быстрее* метода без средства от исключений. Но при использовании RyuJIT мы получаем противоположный результат! Производительность всегда зависит от окружения. Вопрос был задан в 2012 году. Первичные измерения проводились с применением .NET Framework 2.0 с LegacyJIT-x86 и старых версий компилятора на C#. Вот цитата из ответа Джона Скита (https://stackoverflow.com/a/8928476): «Возможно, из-за блока “попытка-перехват” больше регистров сохраняется и восстанавливается, поэтому JIT использует их для цикла... что в целом улучшает производительность. Непонятно, является ли разумным решение JIT не использовать столько регистров в нормальном коде».

Изначальная проблема та же (компилятор JIT в одном случае использует регистры, а в другом — стек), но результат противоположен. Поэтому бесполезно применять подобные знания при оптимизациях производительности: разные

компиляторы JIT используют разные алгоритмы, которые в любой момент могут измениться. Однако знать это очень полезно при исследовании производительности, когда вы пытаетесь объяснить какие-нибудь интересные эффекты в данной сфере.

Практический пример 4: количество вызовов

Количество вызовов в методе — важный фактор для некоторых эвристических правил компилятора JIT. Ограничение их числа может быть небольшим, но оно способно заставить компилятор JIT поменять сгенерированный собственный код для других выражений в том же методе.

Исходный код

Рассмотрим следующие бенчмарки на базе BenchmarkDotNet:

```
public class X {}

[LegacyJitX86Job]
public class Benchmarks
{
  private const int N = 100001;

  [Benchmark(Baseline = true)]
  public double Foo()
  {
    double a = 1, b = 1;
    for (int i = 0; i < N; i++)
      a = a + b;
    return a;
  }

  [Benchmark]
  public double Bar()
  {
    double a = 1, b = 1;
    new X(); new X(); new X();
    for (int i = 0; i < N; i++)
      a = a + b;
      return a;
  }
}
```

У нас есть два метода, `Foo` и `Bar`. Оба складывают одну переменную `double` с другой в цикле. Однако у метода `Bar` есть три дополнительных вызова конструктора.