ризации для такого рода программных приложений, по мнению авторов, является целесообразным: программная реализация является простой, проверка правомочности доступа не требует больших вычислительных ресурсов, предлагаемая процедура авторизации просто встраивается в существующие сетевые протоколы (например, протокол RFC 7519 для создания токенов доступа, основанных на формате JSON [3]).

ЛИТЕРАТУРА

1. Хоффман Л. Современные методы защиты информации. М.: Сов.Радио,1980.

2. Айерлэнд К., Роузен М. Классическое введение в современную теорию чисел . – М.: Мир, 1987.

3. Предлагаемый стандарт RFC 7519 [Электронный ресурс] URL: https://tools.ietf.org/html/rfc7519 дата доступа:10.10.2020.

УДК 004.051

Долговечный В.Н. маг.; П. П. Урбанович, проф., д-р техн. наук
(БГТУ, г. Минск)

СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ СБОРЩИКА МУСОРА И JIT КОМПИЛЯТОРА НА ПЛАТФОРМАХ .NET 5.0, .NET FRAMEWORK 4.8 И .NET CORE 3.1

10 ноября 2020 г. вышел релиз фреймворка .NET 5, который является развитием .NET Core в долгой эволюции фреймворка .NET. .NET 5.0 - это первый релиз на пути к унификации платформы .NET, который позволяет более плавно мигрировать с .NET Framework и использовать преимущества и функции .NET Core. .NET 5 – это объединение его двух предшественников .NET framework 4.8 и .NET Core 3.1.

В данном исследование рассмотрим, улучшилась ли производительность сборщика мусора (garbage collector GC) и компилятора Just-In-Time (JIT) на реальных примерах.

Сборщик мусора .NET управляет выделением и освобождением памяти для приложения. При каждом создании объекта среда CLR выделяет память для объекта из управляемой кучи. Пока в управляемой куче есть доступное адресное пространство, среда выполнения продолжает выделять пространство для новых объектов. Механизм оптимизации сборщика мусора определяет наилучшее время для выполнения сбора, основываясь на выполненных операциях выделения памяти. Когда сборщик мусора выполняет сборку, он проверяет нали-

чие объектов в управляемой куче, которые больше не используются приложением, а затем выполняет необходимые операции, чтобы освободить память.

JIT-компиляция − технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом (сравнимая с компилируемыми языками) за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию. Технология JIT базируется на двух более ранних идеях, касающихся среды выполнения: компиляции байт-кода и динамической компиляции.

Для вызова среды выполнения управляемым кодом требуется относительно небольшое количество накладных расходов, но, когда такие вызовы выполняются с высокой частотой, такие накладные расходы складываются. Новый GC переместил реализацию сортировки массивов примитивных типов из собственного кода в coreclr на C# в Corelib. В дополнение к этому коду, который затем поддерживает новые общедоступные API-интерфейсы для сортировки интервалов, он также удешевил сортировку массивов, где в стоимости этого преобладает переход от управляемого кода.

Используем сортировку двоичных и целочисленных чисел, а также строки. Результаты замеров скорости показаны в таблице 1. Из этих данных видно, что .NET 5.0 работает более быстро чем его два предшественника. Для double типа скорость выполнения сократилась более чем в два раза.

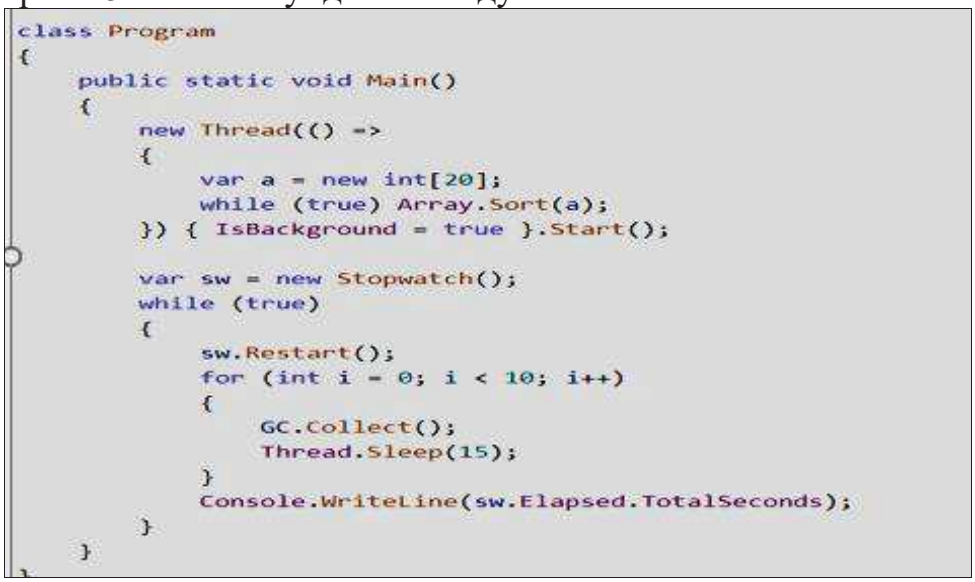
Таблица 1 – Результат выполнения сортировок

Тип объекта	Среда выполнения	Время
Double	.NET FW 4.8	88,88 нс
Double	.NET Core 3.1	73.29 нс
Double	.NET 5.0	35,83 нс
Int32	.NET FW 4.8	66,34 нс
Int32	.NET Core 3.1	48,47 нс
Int32	.NET 5.0	31.07 нс
String	.NET FW 4.8	2193,86 нс
String	.NET Core 3.1	1,713,11 нс
String	.NET 5.0	1,400.96 нс

Одним из показателей для GC является «время паузы», которое фактически означает, как долго GC должен приостанавливать выпол-

нение, чтобы выполнить свою работу. Более длительное время паузы напрямую влияет на задержку, которая может быть важным показателем для всех видов рабочих нагрузок. GC может потребоваться приостановить потоки, чтобы получить единообразное представление о мире и убедиться, что он может безопасно перемещать объекты, но, если поток в настоящее время выполняет код C/C++ во время выполнения, GC может потребоваться дождаться завершения этого вызова, прежде чем он сможет приостановить поток.

Рассмотрим код на рис 1. Код запускает поток, который просто выполняет цикл, снова и снова сортируя небольшой массив, в то время как в основном потоке он выполняет 10 сборок мусора, каждый с примерно 15 миллисекундами между ними.



```
class Program
{
    public static void Main()
    {
        new Thread(() =>
        {
            var a = new int[20];
            while (true) Array.Sort(a);
        }) { IsBackground = true }.Start();

        var sw = new Stopwatch();
        while (true)
        {
            sw.Restart();
            for (int i = 0; i < 10; i++)
            {
                GC.Collect();
                Thread.Sleep(15);
            }
            Console.WriteLine(sw.Elapsed.TotalSeconds);
        }
    }
}
```

Рисунок 1 – Код для проверки «времени паузы»

Результат измерений можно посмотреть в таблице 2. Из таблицы видно, как быстро стал работать GC. Эти цифры доказывают, что новый подход GC является весьма эффективным

Таблица 2 – Результаты проверки время паузы

Среда выполнения	Время
.NET FW 4.8	6,832 с
.NET Core 3.1	6,032 с
.NET 5.0	0,1594 с

Среда выполнения coreclr использует GC, что означает, что сборщик мусора может отслеживать со 100% точностью, какие значения относятся к управляемым объектам, а какие нет; это имеет свои преимущества, но также имеет свои затраты. В отличие от этого, среда выполнения mono использует «консервативный» сборщик мусора,

который имеет некоторые преимущества в производительности, но также означает, что он может интерпретировать произвольное значение в стеке, которое оказывается таким же, как адрес управляемого объекта как действующая ссылка на этот объект.

Одна из таких затрат заключается в том, что JIT необходимо помогать сборщику мусора, гарантируя, что любой локальный объект, который может быть интерпретирован как ссылка на объект, будет обнулен до того, как сборщик мусора обратит на него внимание.

Рассмотрим тестирование кода на рис. 2. Код выполняет простое обнуление.

```
public int Zeroing()
{
    ReadOnlySpan<char> s1 = "hello world";
    ReadOnlySpan<char> s2 = Nop(s1);
    ReadOnlySpan<char> s3 = Nop(s2);
    ReadOnlySpan<char> s4 = Nop(s3);
    ReadOnlySpan<char> s5 = Nop(s4);
    ReadOnlySpan<char> s6 = Nop(s5);
    ReadOnlySpan<char> s7 = Nop(s6);
    ReadOnlySpan<char> s8 = Nop(s7);
    ReadOnlySpan<char> s9 = Nop(s8);
    ReadOnlySpan<char> s10 = Nop(s9);
    return s1.Length + s2.Length + s3.Length + s4.Length + s5.Length + s6.Length + s7.Length
}

[MethodImpl(MethodImplOptions.NoInlining)]
private static ReadOnlySpan<char> Nop(ReadOnlySpan<char> span) => default;
```

Рисунок 2 – Код для тестирования JIT

По результатам из таблицы 3 видно, что улучшение производительности есть, хоть и не такое большое как хотелось бы.

Таблица 3 – Результаты обнуления JIT

Среда выполнения	Время
.NET FW 4.8	22 нс
.NET Core 3.1	18 нс
.NET 5.0	15 нс

В результате данного исследования подтвердилось улучшение производительности GC и JIT-компилятора в новой версии .Net 5.0. Это даёт возможность улучить наши быстродействие наших приложений, а также сэкономить ресурсы для них.