РОЧЕВ К. В.
АНАЛИЗ БЫСТРОДЕЙСТВИЯ ТИПОВЫХ ОПЕРАЦИЙ ЯЗЫКА C# НА ПЛАТФОРМАХ DOT.NET И MONO

УДК 004:4’2, ВАК 05.13.18, ГРНТИ 50.41.01

Анализ быстродействия типовых операций языка C# на платформах DOT.NET и Mono

К. В. Рочев

Ухтинский государственный технический университет, г. Ухта

Статья посвящена изучению быстродействия часто используемых функций стандартных классов языка C# в разных окружениях, таких как WPF, Windows forms, Unity и ASP.NET. Реализован асинхронный механизм инструментальной оценки быстродействия участков кода. Рассмотрены несколько версий фреймворка, включая Mono, Core и традиционный .NET Framework, чтобы выявить разницу в скорости выполнения тех или иных функциональных возможностей платформ.

Ключевые слова: *быстродействие, оптимизация кода, производительность, C#, WPF, Windows Forms, DOT.NET Framework, Unity, Core*

Standard operations performance analysis of the C# language on platforms DOT.NET and Mono

К. В. Рочев

Ukhta State Technical University, Ukhta

The article is devoted to the study of C# language frequently used functions performance in different environments, such as WPF, Windows forms, Unity and ASP.NET. Implemented an asynchronous mechanism tool of evaluating the performance of code regions. Several versions of the framework, including Mono, Core and the traditional .NET Framework, are reviewed to determine the difference in the speed of execution of certain platform functionalities.

Keywords: *performance, code optimization, C#, WPF, Windows forms, DOT.NET Framework, Unity, DOT.NET core*

Введение

Оптимизации быстродействия программного обеспечения многими разработчиками в настоящее время уделяется недостаточно внимания. Это хорошо заметно по медленной работе некоторых современных приложений и их большим размерам. В качестве примера можно привести несколько самых распространенных по назначению приложений, реализующих такие функции как набор текстов, просмотр страниц в сети Интернет, голосовая связь (рис. 1), с которым справлялись и машины конца прошлого века, обладавшие в десятки и сотни раз меньшими мощностями, чем нынешние [1].

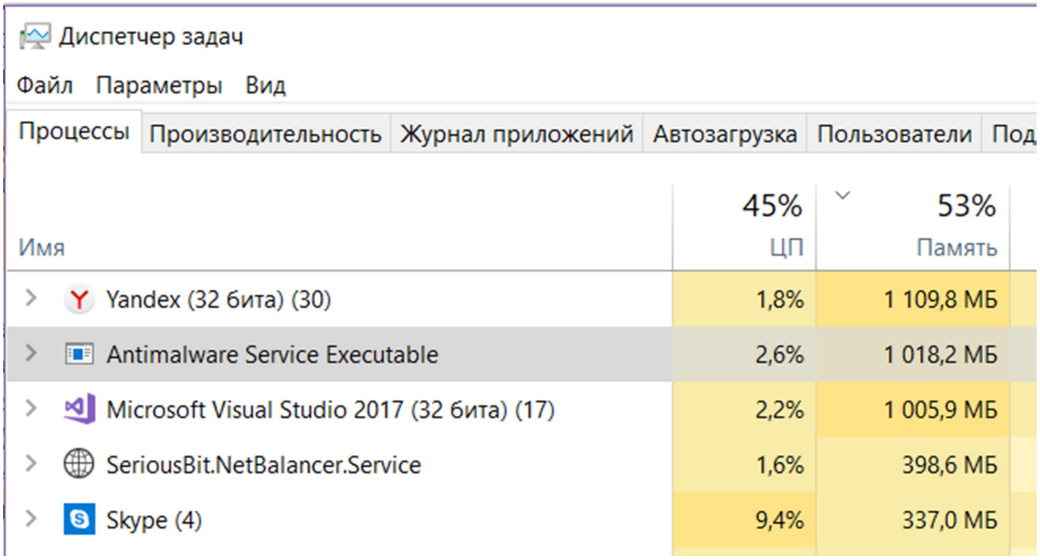

Рисунок 1. Пример объёмов памяти,
занимаемых современным программным обеспечением

Проблема кроется, зачастую, в использовании излишне тяжелых библиотек и компонентов, которые тяжелы из-за того, что сами используют другие библиотеки и это дерево уходит своими корнями глубоко в историю. Не меньшее влияние оказывает и неправильное применение структур данных, не учитывающие сложности алгоритмов при обработке множеств элементов. Элементарные операции и типовые функции из стандартных библиотек, на первый взгляд, влияют на быстродействие в меньшей степени, однако они, ввиду частого использования тоже, порой, вносят своё воздействие в изменение быстродействия. Особенно это становится заметно при обработке больших объемов данных.

Данная статья посвящена изучению быстродействия элементарных операций и часто используемых функций языка C# в разных окружениях, таких как WPF, Windows forms, Unity. Мы рассмотрим несколько версий фреймворка и видов проекта, чтобы увидеть, есть ли разница в скорости выполнения того или иного функционала.

Для изучения напишем небольшой тестовый класс, разместим его в переносимой библиотеке и будем ее подключать в разные среды выполнения. Компактная переносимая библиотека классов, подключаемая к разным средам выполнения с исходнымм кодом размещена в открытом доступе на сервисе Bit-bucket [2].

Методология и реализация тестового окружения

Основной функционал класса, реализованного для изменения быстродействия, следующий:

1) поток, постоянно выполняющий изучаемую функцию из её делегата *private Action TestAction*, что позволяет частично обойти оптимизацию повторяющихся операций механизмами .Net Framework-а;

2) функция замера, принимающая делегат – замеряет количество выполнений этого делегата потоком в течение 1 миллисекунды;

3) функция подсчета – накапливает результаты одинаковых замеров в словаре для последующего устранения пиковых результатов, усреднения, вычисления медианного значения;

4) механизм минимизации вызова сборщика мусора и замера частоты его во время тестов.

Класс тестировщика производительности представлен ниже:

```
public class TimeTestAsync
{
    public int TimeMilliseconds = 1;

    private Action TestAction = () => { };
    private Thread Thread;
    public int Count = 0;

    public readonly Dictionary<string, List<TimeResult>> Results =
           new Dictionary<string, List<TimeResult>>();

    private void Init()
    {
        if (Thread != null) return;
        Thread = new Thread(TestFunk) { IsBackground = true };
        Thread.Start();
    }

    ~TimeTestAsync() => Stop();

    public void Zamer(string info, Action action)
    {
        Init();
        if (!Results.ContainsKey(info))
            Results.Add(info, new List<TimeResult>());
        var z = Zamer(action);
        Results[info].Add(z);
    }

    private DateTime _start;

    private TimeResult Zamer(Action action)
    {
        var gc = GC.CollectionCount(0);
        TestAction = action;
        Count = 0;
        _start = DateTime.UtcNow;
        Thread.Sleep(TimeMilliseconds);
        var end = DateTime.UtcNow.Subtract(_start);
        return new TimeResult()
        {
            Count = Count,
            Time = end,
            GC = GC.CollectionCount(0) - gc
        };
    }

    private void TestFunk()
    {
        while (true)
        {
            TestAction.Invoke();
            ++Count;
```

```
            }
        }

        public void Stop()
        {
            Thread?.Abort();
            Thread = null;
        }
    }
```

Результат замера выглядит следующим образом:

```
    public class TimeResult
    {
        public int Count;
        public TimeSpan Time;
        public int GC;

        public double Nanoseconds()
        {
            return Time.TotalMilliseconds / Count * 1000000;
        }

        public override string ToString()
        {
            return "\t" + Nanoseconds() + "\t" + Count;
        }
    }
```

По времени замера и количеству операций он определяет время выполнения тестируемой функции в наносекундах.

И, собственно, применение этого класса замеров возможно, например, в таком виде:

```
    public class Tests
    {
        readonly TimeTestAsync Tester = new TimeTestAsync();
        private const string Br = "\t";
        private float ClassProperty { get; set; }
        static float StaticProperty { get; set; }
        static float StaticField = 0;
        float ClassField = 0;
        float ClassField2 = 0;
        string ClassStr = "";
        bool ClassBool = false;
        private const int Min = 50, Max = 100;
…
        public string Test()
        {
            var localRandom = new Random();

            for (int i = 0; i <= 10; i++)
            {
                ClassStr = "";
                ClassStringBuilder = new StringBuilder();
                ClassField = localRandom.Next(Min, Max);
…
                GC.Collect();
                Tester.Zamer("() => { ClassField++; }", () => { ClassField++; });
                Tester.Zamer("() => ClassStr = \"S1\"", () => ClassStr = "S1");
                Tester.Zamer("() => ClassStr = \"S1\" + ++ClassField",
                             () => ClassStr = "S1" + ++ClassField);
                GC.Collect();
```

```
…
                if (i == 0) Tester.Results.Clear();
                StaticField += localField;
            }

            Tester.Stop();
            var tempCount = ClassField + StaticField;

            string s = $"Функция{Br}" +
                       $"Среднее время на выполнение, нс{Br}" +
                       $"Медиананное время на выполнение, нс{Br}" +
                       $"Среднеквадратичное отклонение{Br}" +
                       $"Среднее к-во запусков за тест, раз{Br}" +
                       $"Среднее Время теста, мс{Br}" +
                       $"К-во тестов, раз{Br}" +
                       $"Среднее к-во вызовов сборки мусора на тест, раз\n";
            foreach (var r in Tester.Results)
            {
                var withResults = r.Value.Where(x => x.Count > 0).ToArray();
                var nano = withResults.Select(x => x.Nanoseconds()).ToList();
                s += $"{r.Key}{Br}" +
                     $"{nano.Average(x => x):F2}{Br}" +
                     $"{nano.OrderBy(x => x).ToArray()[nano.Count / 2]:F2}{Br}" +
                     $"{StandardDeviation(nano):F2}{Br}" +
                     $"{withResults.Average(x => (double)x.Count):F0}{Br}" +
                     $"{withResults.Average(x => x.Time.TotalMilliseconds):F2}{Br}" +
                     $"{withResults.Length}{Br}" +
                     $"{withResults.Average(x => (double)x.GC):F2}\n";
            }
            s += "\n\n" + tempCount;// + "\n\n" + Tester.GcCount
            return s;
        }
```

Простейший интерфейс позволяет скопировать результаты в Excel и там их обработать:

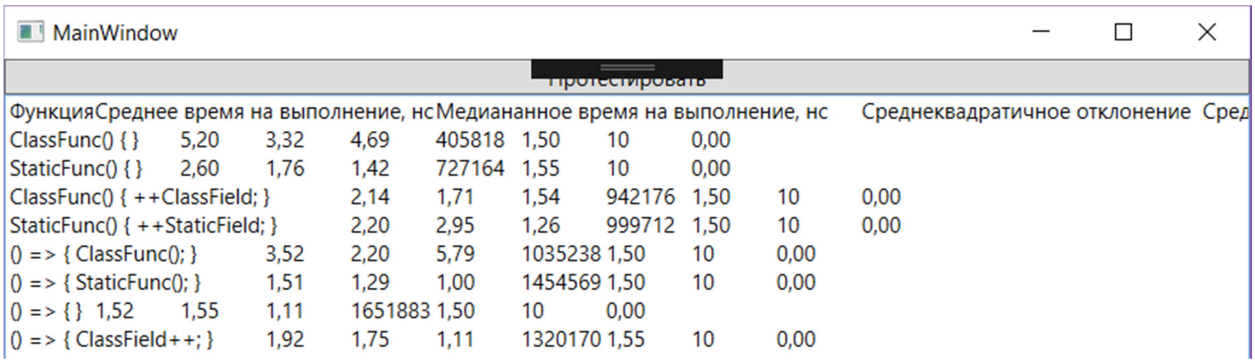

Рисунок 2. Результат измерения в простом окне WPF для последующего копирования в Excel

Относительные результаты измерения быстродействия по рассмотренным группам операций оценивались относительно WPF по следующей формуле:

$$X = \frac{\sum S_t \cdot 2}{\sum S_t + S_{WPF}} \quad (1)$$

где X – относительное быстродействие; S_t – Результат на рассматриваемой платформе; S_{WPF} – Результат на WFP.

Результаты измерений

Измерение проводилось на ноутбуке ASUS X556UQ: i7-7500U, 2.7 GHz, 20Г ОЗУ, Windows 10 x64.

Для оценки быстродействия реализованного тестового окружения были выбраны такие операции, как обращение к функциям, полям и свойствам класса (табл. 1 и 2). Как можно заметить, быстродействие тестовой инфраструктуры примерно сопоставимо с обычным обращением к функции или переменной. Кроме того, несмотря на предпринятые меры по усложнению оптимизации вычислений, в простых операциях заметен разгон при выполнении нескольких однотипных замеров подряд. В дальнейшем было принято решение рассматривать и сопоставлять результаты по медианному времени, поскольку того, что оно более стабильно, чем среднее, т. к. не подвержено влиянию исключительных случаев. Хотя, как ни странно, из-за особенностей разброса результатов (и сдвига в большую сторону при четном количестве тестов) в некоторых замерах медианный результат получен больше среднего.

Таблица 1. Результаты тестовых замеров на примере проекта WPF в Release-режиме

Функция	Среднее время на выполнение, нс	Медианное время на выполнение, нс	Среднеквадратичное отклонение	Среднее к-во запусков за тест, раз	Среднее Время теста, мс	К-во тестов, раз	Среднее к-во сборки мусора на тест, раз
ClassFunc() { }	2,02	1,97	0,23	736911	1,48	10	0
StaticFunc() { }	1,22	1,12	0,48	1268418	1,45	10	0
ClassFunc() { ++ClassField; }	1,74	2,57	1	1241860	1,57	10	0
StaticFunc() { ++StaticField; }	1,34	1,15	0,9	1617269	1,5	10	0
() => { ClassFunc(); }	0,87	0,76	0,5	2067692	1,41	10	0
() => { StaticFunc(); }	0,73	0,55	0,52	2737093	1,55	10	0
() => { }	0,61	0,54	0,43	3381908	1,6	10	0
() => { ClassField++; }	0,46	0,42	0,24	3979307	1,55	10	0
() => { ++ClassField; }	1,02	0,33	1,1	3631160	1,49	10	0
() => { ++localField; }	1,7	2,58	1,18	2409092	1,47	10	0
() => { ++StaticField; }	1,58	1,46	0,99	2119760	1,5	10	0
() => { ++ClassProperty; }	1,45	1,3	0,89	1863567	1,55	10	0
() => { ++StaticProperty; }	1,98	1,02	1,92	1944740	1,46	10	0

Далее приведем результаты замеров в разных режимах выполнения (табл. 2), поскольку на второй фазе компиляции при создании релиз-приложения используются дополнительные механизмы оптимизации [3].

Таблица 2. Замеры в WPF в разных режимах выполнения

Функция	**WPF DEBUG с отладчиком, нс**	**WPF DEBUG без отладчика, нс**	**WPF RELEASE с отладчиком, нс**	**WPF RELEASE без отладчика, нс**
ClassFunc(){}	4,83	3,68	2,39	2,10
StaticFunc(){}	4,65	3,10	2,90	1,72
() => { }	4,08	3,15	3,07	1,64
() => { ClassFunc(){} }	7,66	4,20	2,17	1,55
() => { StaticFunc(){} }	7,22	4,82	2,29	1,17
() => { ++LocalInt; }	16,20	4,28	2,12	1,51
() => { ++ClassIntField; }	5,95	3,77	2,33	1,55
() => { ++StaticIntField; }	4,52	3,09	2,16	1,36
() => { ++ClassIntProperty; }	20,18	7,80	1,98	1,61
() => { ++StaticIntProperty; }	11,46	7,47	1,80	1,49

Как можно заметить, элементарные операции довольно существенно оптимизируются при переводе проекта в релиз. Устраняются лишние сложности вызова свойств, и они начинают работать также быстро, как обычные поля, удаляются вызовы пустых функций. Расхождения в замерах быстродействия становятся обусловлены в большей степени случайными флуктуациями. При этом оптимизатор настолько хорош, что можно заметить постепенное ускорение быстродействия для похожих операций, несмотря на их вызов через делегаты.

Далее для сопоставления фреймворков будем рассматривать наиболее актуальный режим запуска – Release.

На рисунке 3 приведены результаты замеров выполнения указанных функций в релиз-проектах на WPF, Windows Forms и Unity. Как можно заметить, WPF и Windows Forms показывают примерно одинаковые результаты (в среднем по рассмотренным операциям формы медленнее на 6 %) ввиду того, что обе платформы реализованы на классическом .Net (при этом, в дебаг-режиме разница между ними более существенная). В то же время, на Unity некоторые операции производятся с существенной разницей в скорости ввиду того, что основаны на MONO-Framwork-е (Unity, в среднем, медленнее на 220 %). В .Net Core реализации заметна не менее ощутимая разница в быстродействии, как в большую, так и в меньшую сторону по разным операциям (в среднем по рассмотренным функциям на 77 % медленнее). Однако выборка функций не является достаточно презентабельной и не даёт полномочий судить о производительности того или иного фреймворка в целом.

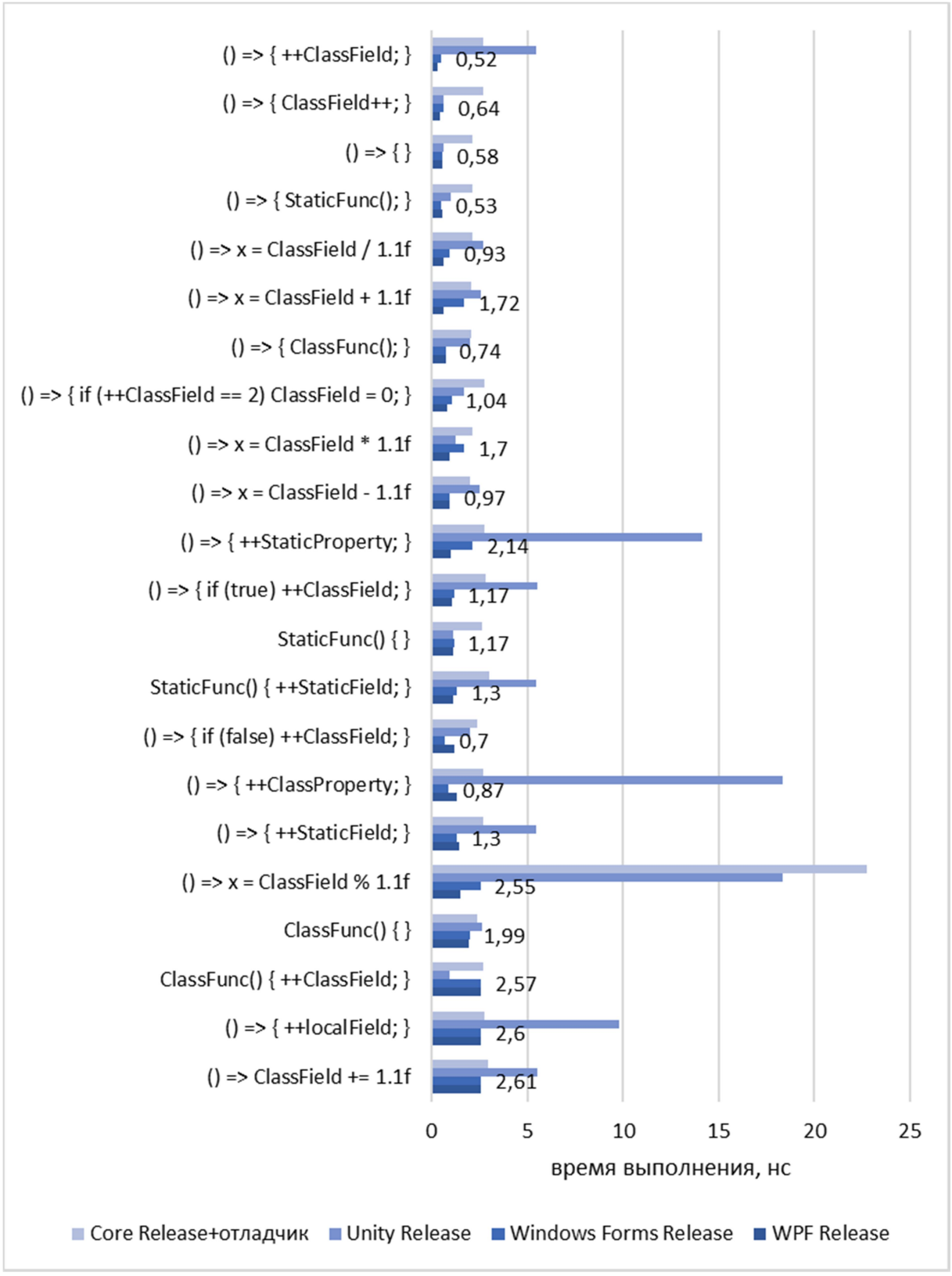

Рисунок 3. Быстродействие обращения к методам, полям и свойствам в различных окружениях

Здесь заметны некоторые сложности с обращением к свойствам в Unity, а также заметно более медленный расчет остатка от деления в Unity и .Net Core – примерно в 10 раз медленнее, чем в обычном .Net, при том, что остальные операции выполняются примерно с той же скоростью. В .Net Core тестовое

окружение (вызов делегата) выполнялось медленнее, учитывая это можно отметить, что остальные обращения и вычисления в нём производились с той же скоростью, что и в обычном .Net.

Быстродействие математических операций

Расссмотрим несколько наиболее популярных математических операций стандартного .Net класса Math (табл. 3 и рис. 4).

Таблица 3. Замеры быстродействия математических операций в различных окружениях

Функция	WPF	Windows Forms	Unity	Core	Unity/ WPF	Core/ WPF
() => Math.Atan(++ClassField) }	54,8	54,87	27,02	2,63	66 %	9 %
() => Math.Atan2(++ClassField, ClassField) }	53,79	53,94	35,93	30,31	80 %	72 %
() => Math.Pow(++ClassField, 2)	45,03	45,14	52,42	2,78	108 %	12 %
() => Math.Pow(++ClassField, 1.4)	44,96	45,85	52,43	2,63	108 %	11 %
() => Math.Round(65.5633, 2)	19,79	21,62	30,31	13,99	121 %	83 %
() => localRandom.Next(10, 10000)	19,44	20,03	15,64	14,04	89 %	84 %
() => ClassField = localRandom.Next(10)	17,03	16,37	16,07	15,82	97 %	96 %
() => Math.Round(0d, 2)	16,01	16,22	10,36	10,88	79 %	81 %
() => Math.Round(0m, 2)	15,39	15,39	17,1	15,74	105 %	101 %
() => localRandom.Next(10)	15,01	15,36	15,12	13,92	100 %	96 %
() => localRandom.NextDouble()	12,65	13,68	14,57	12,14	107 %	98 %
() => Math.Ceiling(++ClassField) }	7,23	7,28	10,3	2,83	118 %	56 %
() => Math.Floor(++ClassField) }	6,69	6,8	10,04	2,72	120 %	58 %
() => Math.Sin(++ClassField) }	3,42	3,06	56,25	2,78	189 %	90 %
() => Math.Sqrt(++ClassField) }	2,6	2,6	9,84	2,75	158 %	103 %
() => Math.Abs(++ClassField) }	2,59	2,59	17,66	2,94	174 %	106 %
() => Math.Log(++ClassField) }	2,21	2,21	14,89	8,94	174 %	160 %
() => Math.Cos(++ClassField) }	1,51	1,54	46,39	2,63	194 %	127 %
() => Math.Tan(++ClassField) }	1,49	2,17	27,49	2,65	190 %	128 %

Как можно заметить, различные математические перации выполняются в разных средах разработки с существенными отличиями в быстродействии. Например, Sin, Cos, корень, логарифм и получение модуля намного медленее работают в Unity, чем в других версиях фреймворка, да и в среднем математика в Unity работает несколько медленее. Хотя, например, арктангенс вычисляется быстрее.

Интересен тот факт, что возведение в степень с помощью функции Pow работает с одной скоростью для целых и дробных степеней и на пару порядков медленнее умножения (за исключением .Net Core, где оно приближается по скорости к простым арифметическим опрациям).

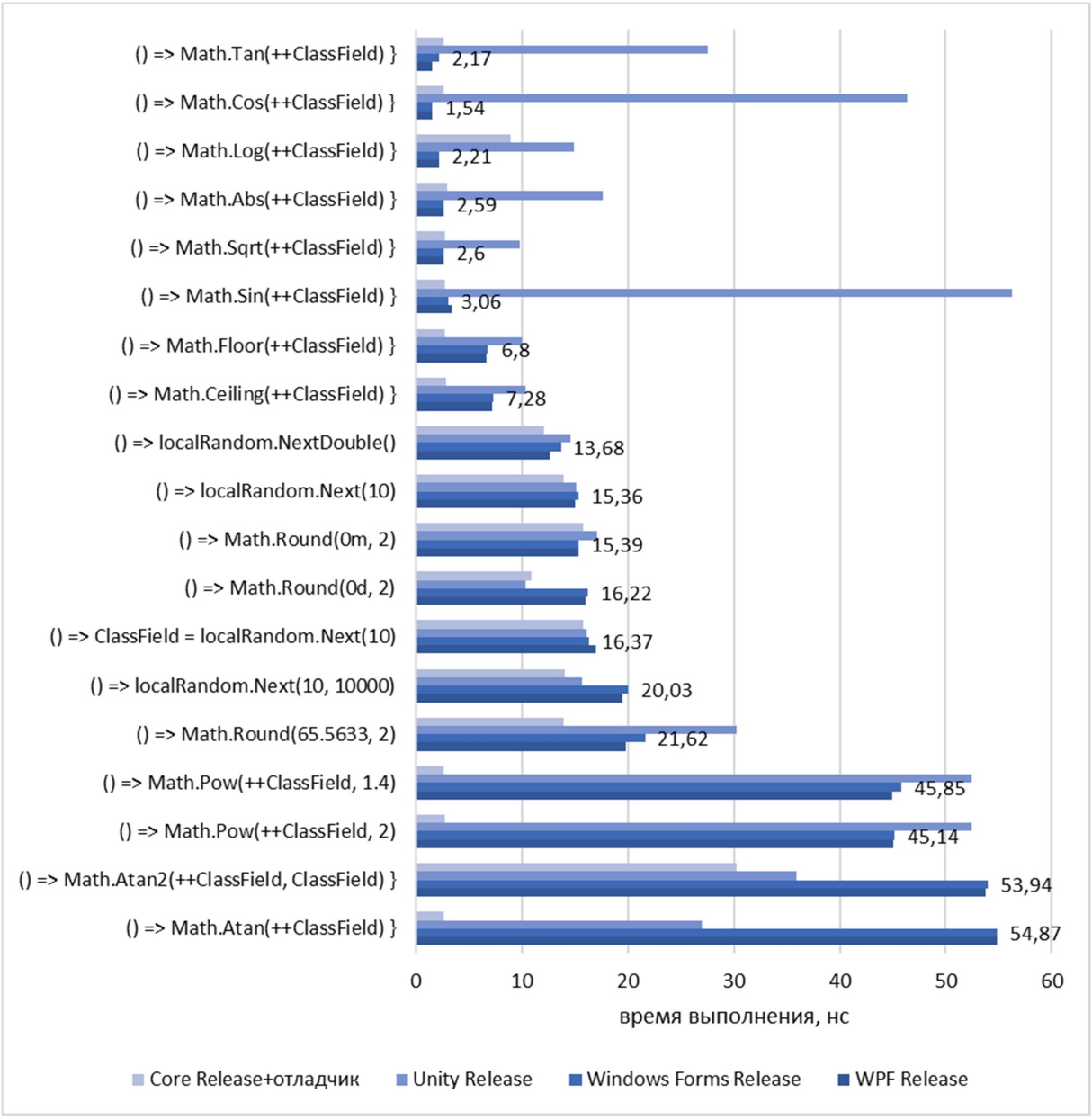

Рисунок 4. Замеры быстродействия математических операций в различных окружениях

Быстродействие функций работы с коллекциями

Приведем результаты исследования работы с коллекциями коротко на примере массивов из 1 и 1000 элементов и таких часто используемых функций, как Contains – поиск элемента, FirstOrDefault – аналогичная функция в LINQ-расширениях, Count – подсчёт с помощью LINQ. ExistElement – случай для элемента, который присутствует в массиве, а NotExistElement – соответственно для несуществующего элемента.

Таблица 4. Замеры быстродействия работы с массивами в различных окружениях

Функция	WPF	Windows Forms	Unity	Core	Unity/ WPF	Core/ WPF
Массив из 1 элемента						
Contains(ExistElement)	42	41	58	31	116 %	85 %
Contains(NotExistElement)	42	43	58	32	116 %	86 %
FirstOrDefault(x => x == ExistElement)	29	29	72	32	143 %	106 %
FirstOrDefault(x => x == NotExistElement)	34	35	74	35	137 %	101 %
FirstOrDefault()	36	23	17	42	63 %	108 %
Count(x => x == ExistElement)	32	33	74	26	140 %	90 %
Count(x => x == NotExistElement)	36	34	77	29	136 %	88 %
Массив из 1000 элементов						
Contains(ExistElement)	1063	1120	15036	840	187 %	88 %
Contains(NotExistElement)	1835	1826	47786	1257	193 %	81 %
FirstOrDefault(x => x == ExistElement)	4685	5470	4664	6318	100 %	115 %
FirstOrDefault(x => x == NotExistElement)	8299	8643	16152	8592	132 %	102 %
FirstOrDefault()	37	38	17	42	64 %	106 %
Count(x => x == ExistElement)	9217	9229	16667	9398	129 %	101 %
Count(x => x == NotExistElement)	9281	8351	16215	8786	127 %	97 %

Как можно заметить, различия в скорости выполнения присутствуют и увеличиваются с увеличением объёма массива. Практически во всех рассмотренных случаях Mono реализация Unity несколько уступает по скорости.

Что интересно, проверка наличия элемента с помощью встроенной функции Contains проходит быстрее, чем с помощью LINQ расширения примерно в 3–5 раз в клссическом и Core фреймворках и в 3 раза медленнее Юнити на 1000 элементах. А на 1 элементе LINQ в 1.5 раза быстрее в клссическом фреймворке и быстрее в Юнити.

В любом случае, быстродействие работы с коллекциями – это тема для отдельного большого исследования.

Быстродействие работы со строками

На рисунке 5 приведены замеры быстродействия строковых операций. Здесь Str1 = «1», Str10 = «1234567890», Str100 = «1234567890123…90» – до длины в 100 символов.

Можно отметить, что операции сложения строк в целом довольно тяжеловесны, также, как и преобразования чисел в строки. Быстродействие методов String.Format уступает по скорости интерполяции строк примерно на 15 %, а интерполяция, в свою очередь, уступает конкатенации примерно на столько же. String.Join, естественно существенно опережает обе из перечисленных функций.

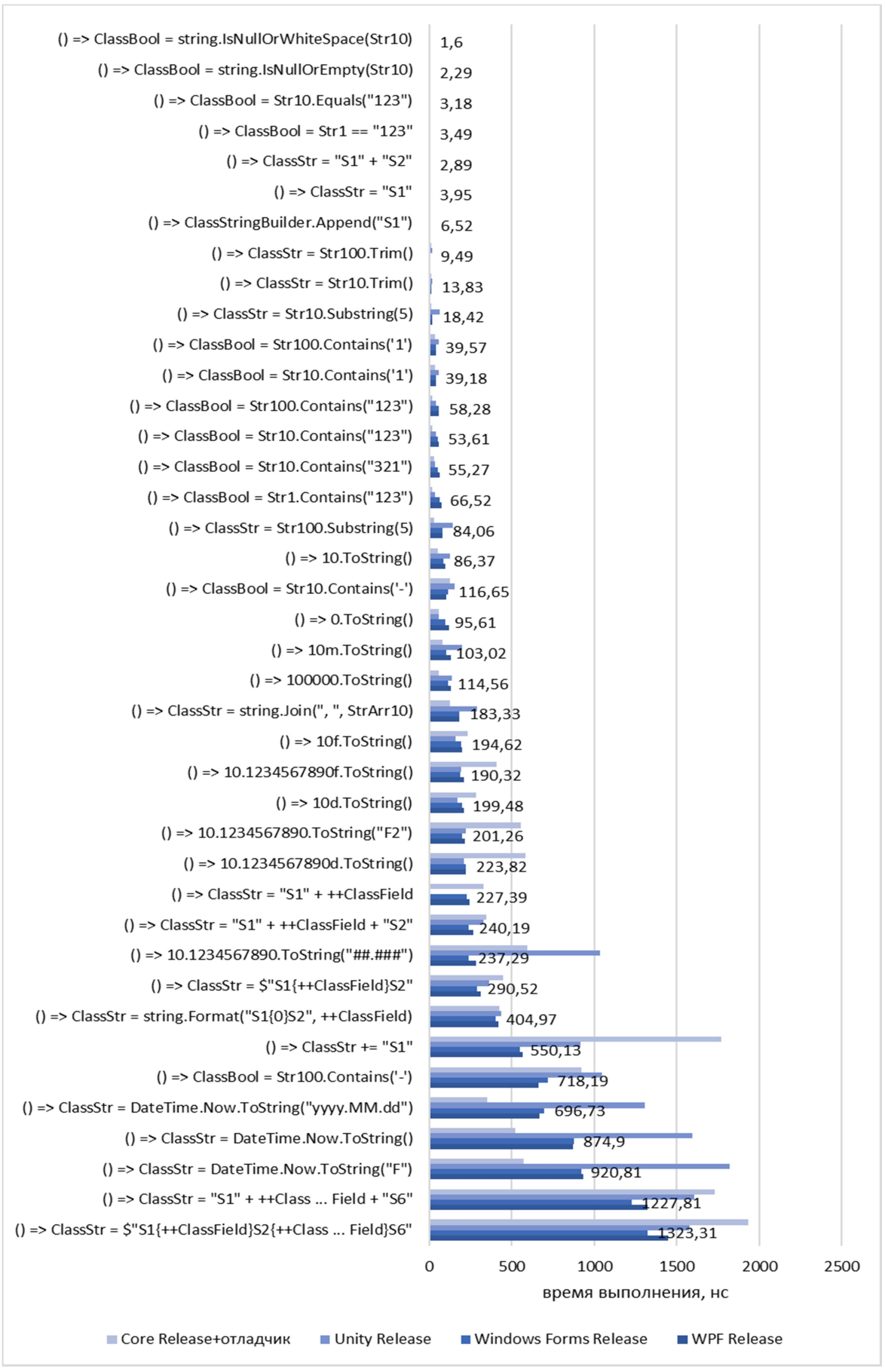

Рисунок 5. Быстродействие работы со строками в различных окружениях

Выводы

При сравнении результатов быстродействия WPF и WindowsForms в релиз-режиме получено, что средняя разница быстродействия операций по разным группам составляет до 10 %, что может быть обусловлено погрешностями измерений. В целом же все операции выполняются примерно с одинаковой скоростью, что не удивительно, ввиду единого фреймворка. Это довольно очевидно и без исследования и приведено в большей степени для того, чтобы можно было со стороны оценить погрешность измерения.

Что касается сравнения с Юнити и .Net Core, то фреймворк уже отличается, ввиду чего и отличия более существенные.

Вызовы пустых функций и обращения к переменным в Юнити выполняются, в среднем, на 43 % медленнее (за счет обращения к свойствам), математические вычисления – на 25 % (за счет таких функций, как sin, cos, tan, sqrt, abs – медленнее в 10-20 раз, тогда как atan, random, pow выполнялись быстрее в 1,5–2 раза), работа с коллекциями – на 25 %, а работа со строками на 10 % медленнее. Первоначальные замеры показывали, что работа со строками в Юнити происходит медленнее, чем в WPF, на 98 % по формуле 1 (в 100 раз), но, после минимизации вызовов сборщика мусора, этот результат существенно улучшился. Тем не менее, при относительно долгом функционировании, сборка мусора в любом случае внесет свой вклад в быстродействие реальной программы.

В .Net Core базовые операции, в среднем, на 43 % медленнее, математические имеют довольно большой разброс и выполняются на 17 % быстрее (до 20 раз быстрее для atan, pow и до 4 раз медленнее для log), коллекции работают немного медленнее с малыми объемами и немного быстрее с большими – в среднем одинаково. Что же касается строк, то здесь среднее быстродействие то же, что и в WPF. Однако быстродействие выполнения отдельного функционала различается до 2–3 раз (наибольшие различия: IsNullOrWhiteSpace в WPF быстрее в 6 раз, а Contains в 3 раза медленнее).

Таким образом можно заметить, что при близких результатах измерений по большинству рассмотренных операций, даже в родственных средах разработки в отдельных случаях есть принципиальные различия быстродействия часто используемых операций, которые имеет смысл учитывать при написании ПО.

Список литературы

1. Моё разочарование в софте [Электронный ресурс] // Хабр. Режим доступа: https://habr.com/post/423889/

2. Публичный репозиторий с кодом проекта [Электронный ресурс]. Режим доступа: https://bitbucket.org/Konstatos/timetest/src/master/readme.md

3. Четверина О. А. Повышение производительности кода при однофазной компиляции // Программирование. 2016. № 1. С. 51–59.

List of references

1. My disappointment in software, Habr. Mode of access: https://habr.com/post/423889.

2. Public repository with project code. Mode of access: https://bitbucket.org/Konstatos/timetest/src/master/readme.md.

3. Chetverina, O. A., “Improved code performance with single phase compilation”, *Programming*, 2016, no. 1, pp. 51–59.