РОЧЕВ К. В., СЕМЯШКИНА А. В. АНАЛИЗ БЫСТРОДЕЙСТВИЯ СТРОКОВЫХ ОПЕРАЦИЙ ЯЗЫКА C# НА РАЗНЫХ ПЛАТФОРМАХ

УДК 004.4`2, ВАК 05.13.00, ГРНТИ 20.00.00

Анализ быстродействия строковых операций языка C# на разных платформах

К. В. Рочев, А. В. Семяшкина

Ухтинский государственный технический университет, г. Ухта

Данная публикация посвящена анализу результатов измерений быстродействия операций по обработке различных коллекций, предоставляемых средой .NET CLR в разных окружениях, таких как WPF, Windows forms, Unity. Рассмотрены несколько версий инструментальных средств разработки и запуска кода, включая Mono, .NET Core и традиционный .NET Framework с тем, чтобы показать, есть ли разница в скорости выполнения тех или иных функций.

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

Analysis of the performance of C# string operations on different platforms

K. V. Rochev, A. V. Semyashkina

Ukhta state technical university, Ukhta

This publication is devoted to the analysis of the results of measuring the speed of processing operations on various collections provided by the .NET CLR in various environments, such as WPF, Windows forms, Unity. Several versions of the code development and launch tools have been reviewed, including Mono, .NET Core, and the traditional .NET Framework, in order to show if there is a difference in the speed of execution of certain functions.

Keywords: *code optimization, performance, C#, WPF, Windows Forms, DOT.NET Framework, Unity, Core, CLR, collections.*

Введение

Вычислительные возможности современных ЭВМ достаточно велики. Тем не менее, растущие объёмы данных, зачастую, требуют применения оптимальных алгоритмов их обработки. Часто в программировании одного и того же результата можно добиться множеством разных способов, некоторые из которых будут работать в тысячи или даже миллионы раз быстрее других. При обработке больших объёмов данных наибольший прирост и падение производительности вычислений, производимых в оперативной памяти ЭВМ, обычно зависит от правильности использования данных, собранных в последовательности, обычно называемые коллекциями.

Для увеличения производительности программного обеспечения используются различные методы. Наиболее распространёнными из них являются:

1) оптимизация кода средствами разработки, применяемая на разных фазах компиляции [1];

2) механизмы распараллеливания задач для их выполнения на нескольких ядрах вычислительной системы одновременно [2];

3) профилирование быстродействия программного кода на основе инструментальных средств, предоставляемых средой разработки [3];

4) профилирование с помощью собственной реализации замеров быстродействия под конкретную задачу;

5) написание эффективных алгоритмов, правильное использование структур данных и функций над ними, которое, зачастую, требует наибольших знаний и предварительных исследований.

Настоящая статья посвящена методам оптимизации программного обеспечения, обозначенным в п. 3 и 4 представленного списка. Основной упор делается на изучение быстродействия коллекций. Анализ производится на примере языка C# в разных окружениях, таких как WPF, Windows forms, Unity. Рассмотрены несколько версий средств разработки и запуска кода (далее – фреймворк) и видов выполняемого проекта с тем, чтобы оценить различие в скорости выполнения тех или иных функций.

Для измерения производительности участков кода авторами реализована компактная библиотека классов, которая подключается к разным средам её выполнения. Реализация данной библиотеки представлена ранее [4].

Методы профилирования

Профилирование программного кода на предмет скорости его выполнения реализуется по следующему алгоритму.

1) проводится несколько измерений, например, 10 (таблица 1). В каждом из них изучаемый участок кода запускается в бесконечном цикле в течение определенного времени, в данном исследовании по 1 миллисекунде. Подсчитывается количество запусков за эту миллисекунду и, путём деления времени тестирования на это количество запусков определяется время выполнения изучаемого участка кода (как видно в таблице 1 и на рисунке 1, время выполнения теста примерно на 0,5 миллисекунды больше установленного минимального времени из-за ожидания переключения потока).

2) рассчитывается среднее и медианное время выполнения кода по всем измерениям. При этом, далее для анализа использовано медианное значение, т.к. на него не влияют так называемые «выбросы» – экстремальные значения, например, при первом измерении или при переключении процессора на другие потоки.

3) дополнительно вычисляется среднеквадратичное отклонение для того, чтобы можно было оценить разброс результатов измерений. Подсчитывается количество срабатываний сборщика мусора, чтобы отследить и нивелировать его влияние на результаты измерений.

4) Все эти данные сохраняются в CSV-формате (рисунок 1) для дальнейшей обработки, например, в Excel.

Относительные результаты измерения быстродействия по рассмотренным группам операций оценивались относительно WPF по 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 компьютере ASUS X556UQ: i7-7500U, 2.7 GHz, 20Г ОЗУ, Windows 10 x64. Для оценки быстродействия реализованного тестового окружения (а именно того, насколько велики накладные расходы на вызов функций по ссылкам) были выбраны такие операции, как обращение к функциям, к полям и к свойствам класса (табл. 1) [4, 5]:

Таблица 1. Результаты тестовых измерений на примере проекта WPF в Release-режиме (десять тестов при среднем числе сборки мусора на тест 0)

Функция	Среднее время на выполнение, нс	Медианное время на выполнение, нс	Среднеквадратичное отклонение	Среднее к-во запусков за тест, раз	Среднее Время теста, мс
() => { }	0,61	0,54	0,43	3381908	1,60
() => { ++localInt; }	1,7	2,58	1,18	2409092	1,47
() => { ++ClassIntField; }	1,02	0,33	1,10	3631160	1,49
() => { ++StaticIntField; }	1,58	1,46	0,99	2119760	1,50
() => { ++ClassIntProperty; }	1,45	1,30	0,89	1863567	1,55
() => { ++StaticIntProperty; }	1,98	1,02	1,92	1944740	1,46
StaticFunc() { }	1,22	1,12	0,48	1268418	1,45

По результатам измерений (таблицы 1 и 2) быстродействие тестовой инфраструктуры примерно сопоставимо с обращением к переменной или к пустой функции. Таким образом, можно считать, что тестовая инфраструктура не оказывает существенного воздействия на результаты измерений и может быть применена для дальнейшего профилирования.

Измерения скорости выполнения операций проводились в различных режимах сборки. К их числу относится фаза отладки в Debug-режиме, а также фаза выпуска приложения в режиме Release. Это важно, поскольку во второй фазе компиляции при создании Release-приложения используются дополнительные механизмы оптимизации [3]. Как показали более ранние исследования, элементарные операции довольно существенно оптимизируются при переводе проекта

в фазу выпуска. Например, обращение к свойствам ускоряется в 10-20 раз и становится почти таким же быстрым, как и обращение к полям класса) [4, 5]. Далее перейдём непосредственно к оценке быстродействия при работе с коллекциями для сопоставления результатов оценки быстродействия кода на разных фреймворках, рассмотрим наиболее актуальный режим запуска – Release.

Результаты оценки быстродействия при работе с коллекциями

В таблицах 3-6 представлены результаты измерений быстродействия выполнения различных операций с коллекциями. Рассмотрим результаты измерений в Release-режиме для WPF, Windows Forms, Unity и ASP.NET Core. Рассмотрим также отношение результатов оценок быстродействия для кода Unity и Core применительно к WPF с учетом формулы (1).

Таблица 3. Результаты измерений быстродействия операций с массивом

Функция	WPF, нс	Windows Forms, нс	Unity, нс	Core, нс	Unity/WPF, %	Core/WPF, %
() => Array|1|.Contains(ExistElement)	42	41	58	31	116%	85%
() => Array|1|.Contains(NotExistElement)	42	43	58	32	116%	86%
() => Array|1|.Count(x => x == ExistElement)	32	33	74	26	140%	90%
() => Array|1|.Count(x => x == NotExistElement)	36	34	77	29	136%	88%
() => Array|1|.FirstOrDefault()	36	23	17	42	63%	108%
() => Array|1|.FirstOrDefault(x => x == ExistElement)	29	29	72	32	143%	106%
() => Array|1|.FirstOrDefault(x => x == NotExistElement)	34	35	74	35	137%	101%
() => Array|16|.Contains(ExistElement)	51	57	269	40	168%	88%
() => Array|16|.Contains(NotExistElement)	68	71	775	50	184%	85%
() => Array|16|.Count(x => x == ExistElement)	176	175	333	168	131%	98%
() => Array|16|.Count(x => x == NotExistElement)	187	165	311	180	125%	98%
() => Array|16|.FirstOrDefault(x => x == ExistElement)	56	119	133	94	141%	126%
() => Array|16|.FirstOrDefault(x => x == NotExistElement)	163	172	313	193	132%	109%
() => Array|1024|.Contains(ExistElement)	1063	1120	15036	840	187%	88%
() => Array|1024|.Contains(NotExistElement)	1835	1826	47786	1257	193%	81%
() => Array|1024|.Count(x => x == ExistElement)	9217	9229	16667	9398	129%	101%
() => Array|1024|.Count(x => x == NotExistElement)	9281	8351	16215	8786	127%	97%
() => Array|1024|.FirstOrDefault(x => x == ExistElement)	4685	5470	4664	6318	100%	115%
() => Array|1024|.FirstOrDefault(x => x == NotExistElement)	8299	8643	16152	8592	132%	102%
() => Array|8192|.Contains(ExistElement)	5260	5610	167150	4795	194%	95%
() => Array|8192|.Contains(NotExistElement)	14222	14733	334267	9892	192%	82%
() => Array|8192|.Count(x => x == ExistElement)	68359	71657	167017	72177	142%	103%
() => Array|8192|.Count(x => x == NotExistElement)	75180	68314	136427	71832	129%	98%
() => Array|8192|.FirstOrDefault()	40	39	21	41	69%	101%
() => Array|8192|.FirstOrDefault(x => x == ExistElement)	24527	35688	55711	25440	139%	102%
() => Array|8192|.FirstOrDefault(x => x == NotExistElement)	62679	68327	125313	68626	133%	105%

В таблице 3 представлены измерения быстродействия некоторых операций при работе с массивами. В столбцах, показывающих отношение производительности Mono (Unity) и .Net Core к традиционному .Net Framework (WPF), значение 200% означает уменьшение быстродействия в бесконечность раз. Так можно заметить, что операции проверки вхождения элемента в массив с помощью LINQ-функции Contains выполняются в Mono существенно медленнее, чем в классическом .Net – при 1024 элементов массива – в 18 раз, а при 8192 – в 35 раз. В то время, как получение первого элемента с помощью LINQ проходит быстрее на 30-40%. Надо отметить, что быстродействие функций LINQ на платформе Mono постепенно улучшается и в последних версиях уже существенно лучше, чем несколько лет назад.

По итогам оценки прироста быстродействия выполнения поиска элементов в массиве при увеличении размера массива можно отметить, что затраты на создание перечислителя при запуске поиска LINQ-функции весьма незначительны – сопоставимы с затратами на перебор 1 дополнительного элемента. Тем не менее созданный перечислитель попадает в виртуальную память, управляемую сборщиком мусора, и может в последующем дополнительно повлиять на производительность – эти затраты тестом не учтены, ввиду минимизации влияний сборки мусора на процесс изменений, предпринятой для снижения флуктуаций вычислений.

Далее рассмотрим результаты изменения быстродействия более разнообразных функций различных коллекций размером 1024 элементов (таблица 4).

Таблица 4. Результаты измерений быстродействия некоторых функций различных коллекций объемом 1024 элементов

Функция	**WPF, нс**	**Windows Forms, нс**	**Unity, нс**	**Core, нс**	**Unity/WPF, %**	**Core/WPF, %**
Array.Contains(ExistElement)	1063	1120	15036	840	187%	88%
Array.Contains(NotExistElement)	1835	1826	47786	1257	193%	81%
Array.Count(x => x == ExistElement)	9217	9229	16667	9398	129%	101%
Array.Count(x => x == NotExistElement)	9281	8351	16215	8786	127%	97%
Array.FirstOrDefault()	37	38	17	42	64%	106%
Array.FirstOrDefault(x => x == ExistElement)	4685	5470	4664	6318	100%	115%
Array.FirstOrDefault(x => x == NotExistElement)	8299	8643	16152	8592	132%	102%
Dictionary.Contains(ExistElement)	15	16	44	12	149%	88%
Dictionary.Contains(NotExistElement)	8	8	18	9	138%	105%
Dictionary.ContainsValue(ExistElement)	3355	3538	8648	3634	144%	104%
Dictionary.ContainsValue(NotExistElement)	3637	3555	7973	3522	137%	98%
HashSet.Contains(ExistElement)	15	14	39	14	145%	99%
HashSet.Contains(NotExistElement)	9	10	16	11	127%	106%
HashSet.Count(x => x == ExistElement)	10639	11572	13200	11241	111%	103%
HashSet.Count(x => x == NotExistElement)	11231	12533	12123	11803	104%	102%

HashSet.FirstOrDefault()	40	40	66	42	125%	103%
HashSet.FirstOrDefault(x => x == ExistElement)	9521	6572	4940	3524	68%	54%
HashSet.FirstOrDefault(x => x == NotExistElement)	12535	11339	12082	10101	98%	89%
List.Any(x => x == ExistElement)	3275	6041	9926	6603	150%	134%
List.Any(x => x == NotExistElement)	8355	8643	11147	8671	114%	102%
List.Contains(ExistElement)	927	1862	5249	942	170%	101%
List.Contains(NotExistElement)	2681	2944	7468	1340	147%	67%
List.Count(x => x == ExistElement)	10096	10447	10897	11140	104%	105%
List.Count(x => x == NotExistElement)	10407	11478	10870	10984	102%	103%
List.Exists(x => x == ExistElement)	1016	2553	2446	2549	141%	143%
List.Exists(x => x == NotExistElement)	3474	3605	3672	2526	103%	84%
List.Find(x => x == ExistElement)	418	932	2767	1864	174%	163%
List.Find(x => x == NotExistElement)	3657	3633	3225	3189	94%	93%
List.FirstOrDefault()	18	19	13	19	85%	102%
List.FirstOrDefault(x => x == ExistElement)	3251	7197	7164	7051	138%	137%
List.FirstOrDefault(x => x == NotExistElement)	9039	9465	10661	10824	108%	109%
List.LastOrDefault()	16	16	15	18	96%	106%
List.LastOrDefault(x => x == ExistElement)	9495	10160	939	1249	18%	23%
List.LastOrDefault(x => x == NotExistElement)	8952	8693	7062	5343	88%	75%
List.Sort()	20319	19430	81346	10467	160%	68%
ObservableCollection.Contains(ExistElement)	1673	1460	3687	656	138%	56%
ObservableCollection.Contains(NotExistElement)	2694	2687	7318	1326	146%	66%
ObservableCollection.FirstOrDefault()	20	19	19	24	96%	109%
ObservableCollection.FirstOrDefault(x => x == ExistElement)	4823	5223	6007	6031	111%	111%
ObservableCollection .FirstOrDefault(x => x == NotExistElement)	9527	10588	10664	10514	106%	105%
SortedList.ContainsKey(ExistElement)	57	55	98	56	127%	100%
SortedList.ContainsKey(NotExistElement)	50	49	103	54	135%	104%
SortedList.ContainsValue(ExistElement)	1782	1767	4736	1196	145%	80%
SortedList.ContainsValue(NotExistElement)	1806	1906	4763	1252	145%	82%
SortedList.Count(x => x.Key == ExistElement)	14608	14601	23279	21235	123%	118%
SortedList.Count(x => x.Key == NotExistElement)	14334	14461	23321	22676	124%	123%
Stack.Contains(ExistElement)	1713	1163	6876	679	160%	57%
Stack.Contains(NotExistElement)	2957	3544	8042	1240	146%	59%
Stack.FirstOrDefault()	50	49	82	36	124%	83%
Stack.FirstOrDefault(x => x == ExistElement)	6155	3980	5303	3068	93%	67%
Stack.FirstOrDefault(x => x == NotExistElement)	13064	11747	12690	11408	99%	93%

По итогам изучения данной таблицы и данных, не вошедших в неё можно отметить такие особенности, как:

1) Функции LINQ, такие как FirstOrDefault и Contains, работают с относительно одинаковой скоростью в разных коллекциях и фреймворках, хотя расхождения доходят иногда до 30%. Это вполне закономерно – реализация функции сходная для фрейморков и едина для разных коллекций одного фреймворка.

2) Получение данных по ключу (Dictionary, HashSet) наглядно показывает своё существенное преимущество, начиная уже с 3-х элементов (собственно, эти коллекции для того и реализованы, хотя в зависимости от типа данных ключа,

время обращения по нему может возрастать в несколько раз, например из-за, боксинга – упаковки значимых типов в object для получения hash-функции). Dictionary получает данные по ключу на 30-70% быстрее, чем HashSet, а SortedList – на 20-30% медленее.

3) Для примера представлена функция Sort коллекции List. Как можно заметить, для 1000 элементов она всего вдвое дольше, чем поиск элемента (не существующего в списке, т.к. существующий, в среднем, находится за половину времени полного перебора в случайном списке). А для 8000 элементов – в 4 раза.

4) Функции Find и Contains (для List и похожих коллекций), реализованные в самих коллекциях быстрее, чем аналогичные LINQ-функции FirstOrDefault и Any, в среднем, в 3 раза для коллекций разного размера (с 1 до 8000 элементов).

5) Из рассмотренных функций наибольшим быстродействием в .Net Core по сравнению с .Net Framework обладают List.Sort для (в 1,5-2 раза быстрее на 1000-4000 элементов), а наименьшим – FirstOrDefault для ObservableCollection. Для остальных функций отклонения менее существенны.

6) Для Mono быстрее других, по сравнению с .Net Framework, работают FirstOrDefault на массиве и списке (в 1,1-1,5 раза), а медленнее – List.Sort (в 4 раза), добавление в ObservableCollection и, как ни странно, FirstOrDefault на стэке и хэшсете (в 1,5-2 раза).

Заключение

При сравнении результатов быстродействия кода WPF и WindowsForms в Release-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 неудивительно, т.к. используется единый фреймворк. Разница быстродействия с .NET Core тоже не очень велика, хотя для отдельных функций доходит до 2х раз. Различия быстродействия .NET и Mono более существенны и в обработке аналогичных коллекций доходят до 4х раз (а при небольших размерах коллекций и до десятков).

В среднем, по всем проведенным измерениям, Mono (Unity) медленнее, чем .Net Framework (WPF) в 1,9 раз (125% по формуле (1)), а .Net Core на 2% быстрее.

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. Такие различия следует учитывать при написании программного обеспечения, хотя куда более существенное влияние на быстродействие ПО оказывает правильный выбор алгоритмов и типов данных, например, использование словарей, хэш-коллекций и иных специально оптимизированных под конкретные задачи решений.

Список использованных источников и литературы

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

2. Джойша П. Г., Шрайбер Р. С., Банерджи П., Боэм Х.-Дж., Чакрабарти Д.Р. О методике прозрачного расширения возможностей классических оптимизаций компилятора для многопоточного кода // Транзакции ACM для языков программирования и систем. – 2012. – Т. 34. № 2.

3. Дараган Е. И. Система анализа производительности программного кода // Известия Тульского государственного университета. Технические науки. – 2013. – № 9-2. – С. 89-94.

4. Рочев К. В. Анализ быстродействия типовых операций языка C# на платформах DOT.NET и Mono // Информационные технологии в управлении и экономике. – 2019. – № 1. – С. 7-19.

5. Рочев К. В. Анализ быстродействия строковых операций языка C# на разных платформах // Программная инженерия. – 2019. – № 6. – С. 274-280.

List of references

1. Chetverina O. A. Improving the performance of the code during single-phase compilation // Programming. – 2016. No. 1. – P. 51-59.

2. Joisha P. G., Schreiber R. S., Banerjee P., Boehm H.-J., Chakrabarti D. R. On a technique for transparently empowering classical compiler optimizations on multi-threaded code // ACM Transactions on Programming Languages and Systems. – 2012. – V. 34. No. 2.

3. Daragan E. I. System analysis of the performance of program code // Bulletin of Tula State University. Technical science. – 2013. No. 9-2. – P. 89-94.

4. Rochev K.V. Analysis of the speed of typical C# language operations on the DOT.NET and Mono platforms // Information Technologies in Management and Economics. – 2019. No 1. – P. 7-19.

5. Rochev K.V. Analysis of the performance of string operations of the C # language on different platforms // Software Engineering. – 2019. No. 6. – P. 274-280.