Методы повышения производительности .NET-приложения на примере программы поиска дубликатов

Юрий Малич
NP4 GmbH
yurymalich@yandex.ru

DOTNEXT

О себе

- Закончил ПГУПС (Автоматика и телемеханика на ж/д транспорте)
- Программирую примерно с 1996 года, с MS DOS на Intel i386
- На C# .NET с первых версий. Microsoft Certified Professional 2016
- Писал статьи по архитектуре микропроцессоров для сайтов iXBT.com, fcenter.ru, 3dnews.ru
- Принимал участие в разработке системных утилит для Windows: в Nero AG (Nero Burning Rom ) и в TuneUp (TuneUp Utilities )
- Работаю в NP4 GmbH, разрабатываю Desktop и Backend приложения

О программе Duplicate Files Search & Link

О программе

- Приложение для Windows 7-11 (начинал с WinXP)
- Собирается под 4 платформы: .Net Fx 4.7.2, .NET 6.0 - 8.0 (32+64 бит)

О программе Duplicate Files Search & Link

О программе

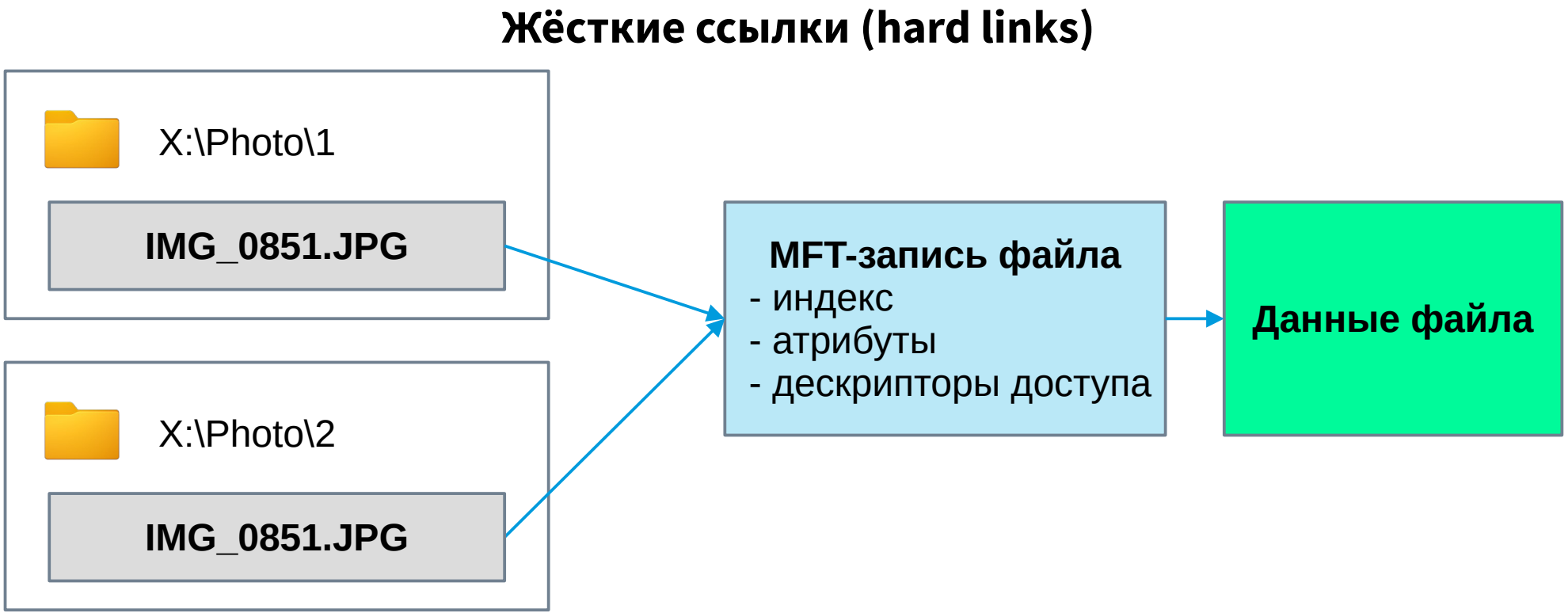
- Находит побайтно идентичные файлы в выбранных папках
- Находит одинаковые медиафайлы (.mp3, .mp4, .jpg etc) по медиаконтенту с игнорированием метаданных (тэгов и т.п.)
- Находит жёсткие ссылки и симлинки на файлы (поддерживаются в файловых системах NTFS, ReFS, EXT2,3,4)
- Умеет заменять дубликаты файлов на жёсткие ссылки и симлинки или удалять файлы (в корзину или насовсем)

О программе Duplicate Files Search & Link

Цель

- Освобождение места на диске за счёт замены файлов жёсткими ссылками или удаления лишних одинаковых файлов
- Наведение порядка, расхламление

О программе Duplicate Files Search & Link



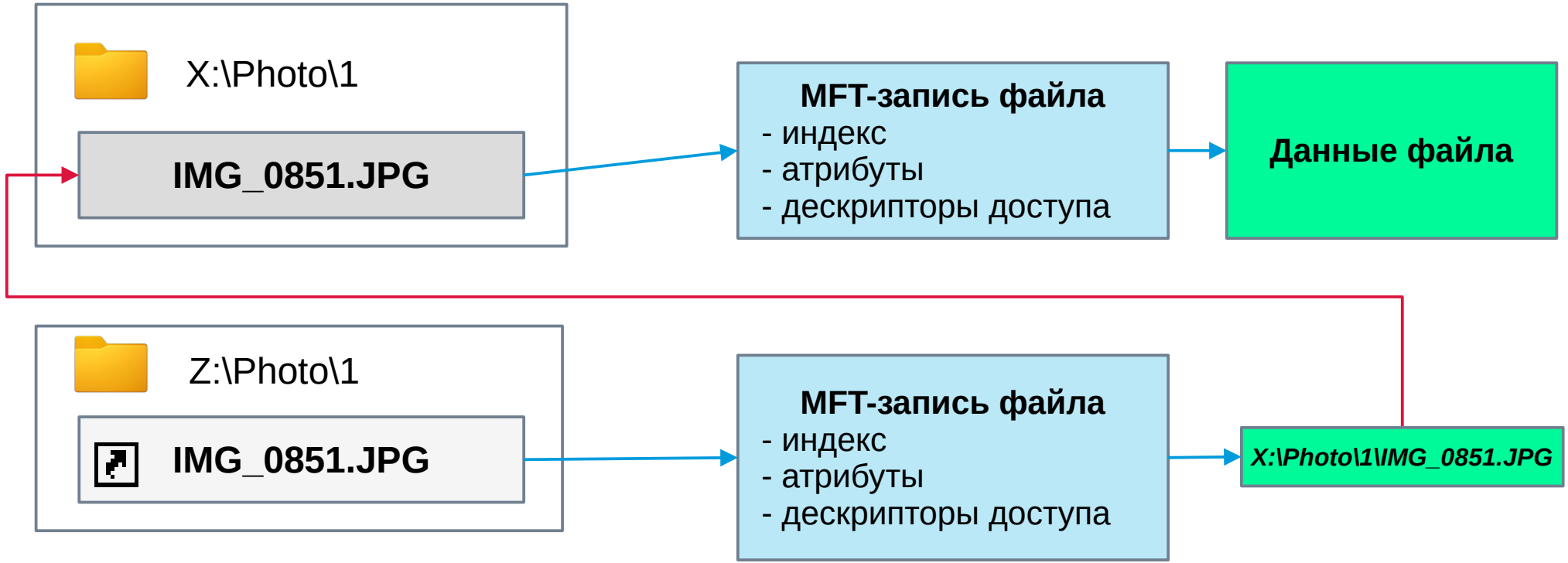
О программе Duplicate Files Search & Link

Жёсткие ссылки (hard links)

- Жёсткая ссылка — это запись в каталоге (имя файла), которая ссылается на файловый дескриптор в MFT с атрибутами, метаданными и контентом
- Каждый файл имеет минимум одну жёсткую ссылку
- Каждый файл в NTFS может иметь до 1023 жёстких ссылок
- Все ссылки равноправны
- Удаление последней ссылки приводит к удалению файла с диска
- Все ссылки на файл могут быть только внутри одного раздела

О программе Duplicate Files Search & Link

Симлинки - символические или символьные ссылки (symbolic links)



О программе Duplicate Files Search & Link

Симлинки - символические или символьные ссылки (symbolic links)

- Симлинк — это специальный текстовый файл, содержащий путь к файлу или каталогу, к которому перенаправляются чтение и запись
- Самостоятельный файл со своими атрибутами и дескрипторами
- Путь может быть полный или относительный
- Может ссылаться на файл на любом разделе, а также быть точкой монтирования раздела (reparse point)
- Если удалить файл, на который ссылается симлинк, или переименовать файл или каталог, то симлинк будет недействительным

О программе Duplicate Files Search & Link

Какие проблемы приходится решать?

- Больше файлов => больше времени нужно на чтение и сравнение
- Чем больше файлов, тем больше памяти требуется программе
- Антивирусы замедляют доступ к файлам
- Огромное разнообразие конфигураций железа и версий софта у конечных пользователей

Алгоритмическая оптимизация поиска

Алгоритмы поиска дубликатов

- Побайтное сравнение файлов одинакового размера между собой
- Подсчёт хэша каждого файла по отдельности и группировка по хэшам

Алгоритмическая оптимизация поиска

Побайтное сравнение файлов между собой

Достоинства:

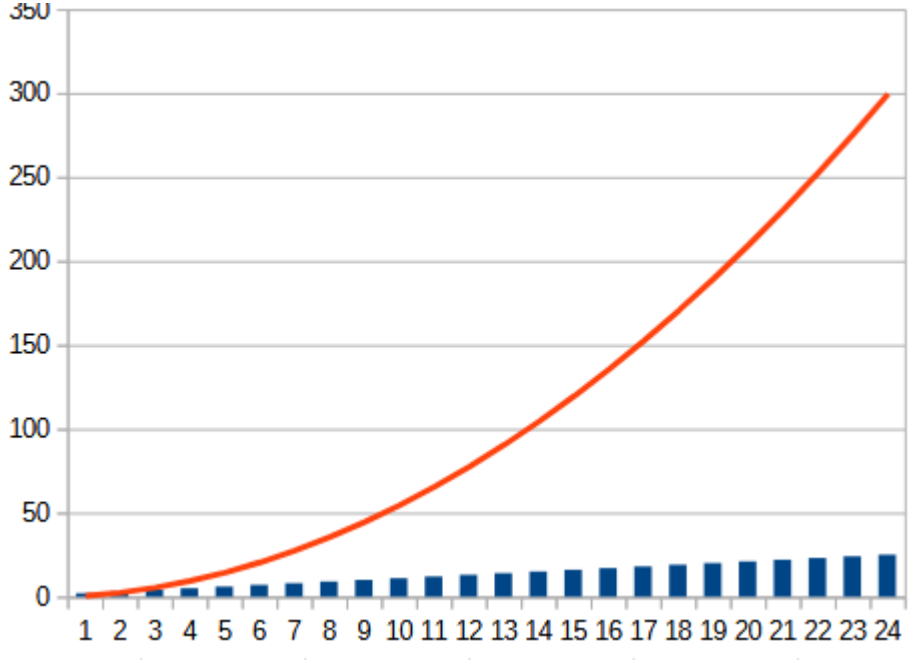
- Если первые байты файлов не равны, то не нужно считывать файлы с диска до конца
- Высокая скорость сравнения групп с малым числом файлов
- Низкая ресурсоёмкость и нагрузка на процессор
- Надёжность

Алгоритмическая оптимизация поиска

Побайтное сравнение файлов между собой

Недостатки:

- Квадратичная алгоритмическая сложность в самом худшем случае O(n^2)



Алгоритмическая оптимизация поиска

Хэширование файлов и сравнение хэшей

Достоинства:

- Алгоритмическая простота алгоритма сравнения O(n)
- Алгоритмы криптографического хэширования из коробки (MD5, SHA1, SHA256)

Алгоритмическая оптимизация поиска

Хэширование файлов и сравнение хэшей

⛔ **Недостатки:**

- Каждый файл нужно прочитать целиком
- Ресурсоёмкость, высокая нагрузка на процессор
- Скорость хэширования намного ниже скорости побайтного сравнения файлов
- Вероятность коллизий (одинаковые хэши разных данных) больше нуля

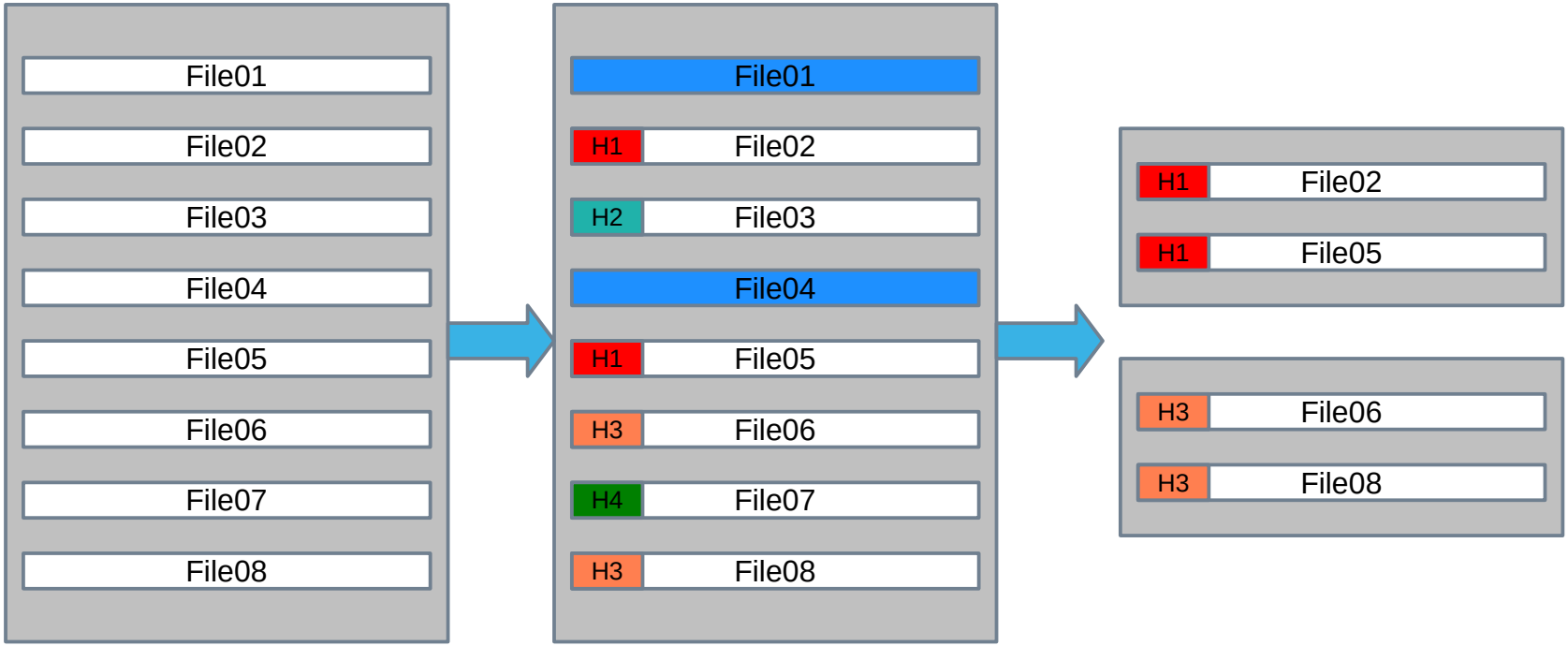
Алгоритмическая оптимизация поиска

Гибридное решение:
скомбинировать достоинства, нивелировать недостатки

- В небольших группах файлы сравниваются побайтно
- Для больших групп используется хэширование, совмещённое со сравнением
  - Для больших файлов хэшируется первый сектор 4КБ
  - Небольшие файлы до 64 КБ считываются, сравниваются и хэшируются целиком
- Результаты хэширования используются для последующей группировки и сравнения по подгруппам

Алгоритмическая оптимизация поиска

Гибридное сравнение



Низкоуровневые оптимизации скорости

1. Простейшее побайтное сравнение буферов

```
for (int i = 0; i < buf1.Length && i < buf2.Length; i++)
{
    if (buf1[i] != buf2[i])
    {
        return false;
    }
}

return true;
```

Слишком медленно, неэффективно

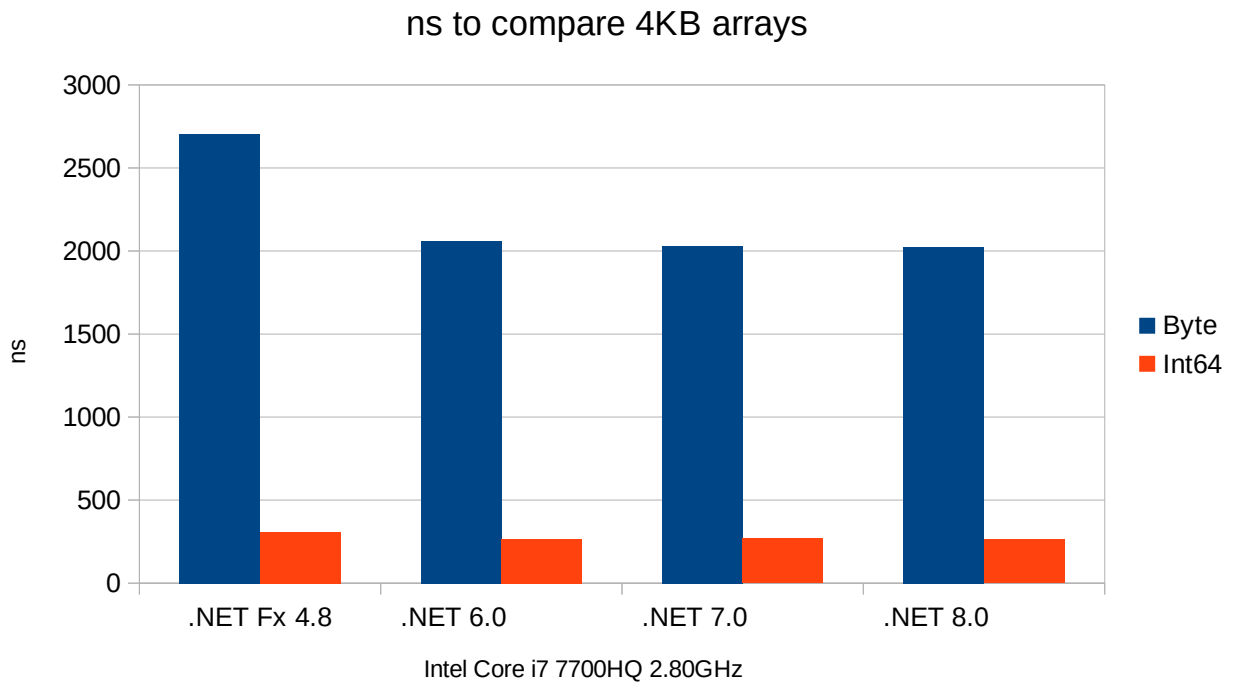
Низкоуровневые оптимизации скорости

2. Сравнение Int64-блоками через fixed-pointer в unsafe-коде

```
fixed (byte* pBuf1 = buf1, pBuf2 = buf2)
{
    int i8 = 0, i = 0;
    int size8 = size / sizeof(long);
    var p1 = (long*)pBuf1;
    var p2 = (long*)pBuf2;

    for (; i8 < size8; i8++)
    {
        if (p1[i8] != p2[i8])
        {
            return i8 * sizeof(long);
        }
    }

    i += i8 * sizeof(long);
    for (; i < size; i++)
    {
        if (pBuf1[i] != pBuf2[i]) { return i; }
    }
}
```

Низкоуровневые оптимизации скорости

Сравнение 4К-массивов побайтно и словами Int64 в BenchmarkDotNet



Ускорение в 9 раз

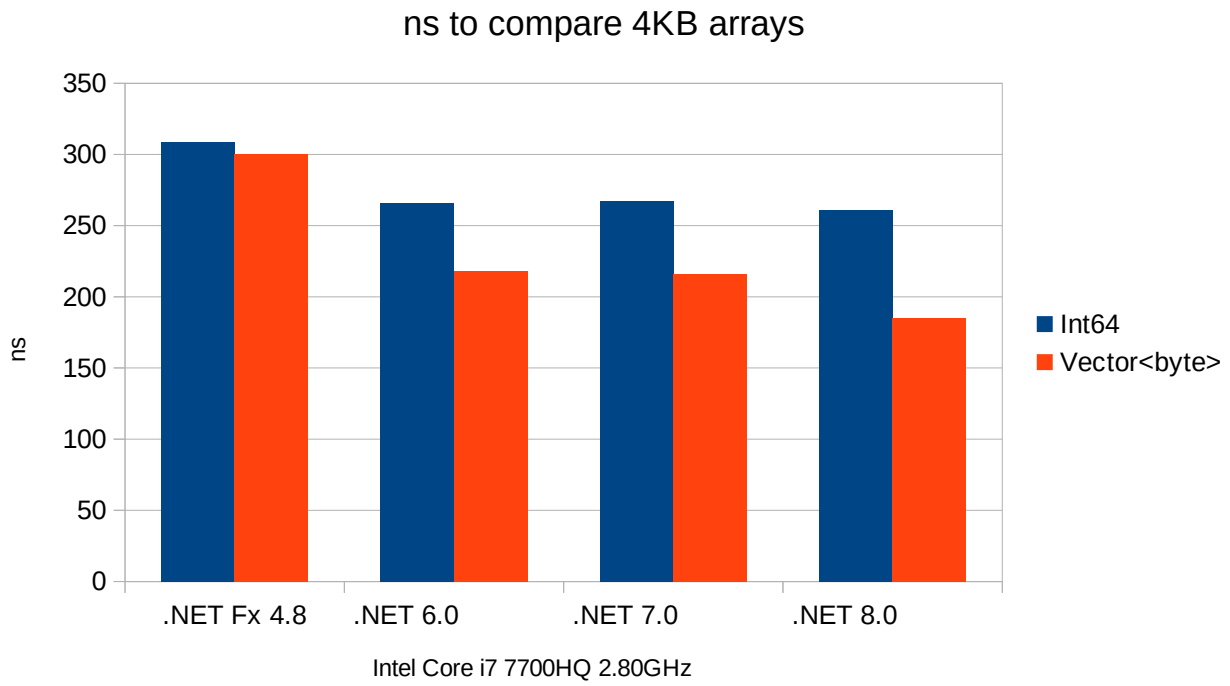
Низкоуровневые оптимизации скорости

3. AVX-сравнение через 32-байтный Vector<byte>

```
var countVectors = size / Vector<byte>.Count;
var sizeAligned = countVectors * Vector<byte>.Count;

for (int i = 0; i < sizeAligned; i += Vector<byte>.Count)
{
    var vector2 = new Vector<byte>(buf2, i);
    var vector1 = new Vector<byte>(buf1, i);

    if (vector1 != vector2)
    {
        return i;
    }
}
```

Низкоуровневые оптимизации скорости

Сравнение 4К-массивов словами 8-байтными Int64 и 32-байтными Vector<byte> в BenchmarkDotNet



Ускорение всего на 10-20%

Низкоуровневые оптимизации скорости

3. AVX-сравнение через 32-байтный Vector<byte>, дизассемблерный код

```
M01_L00:
        cmp         eax,r10d
        jae         near ptr M01_L09
        lea         esi,[rax+1F]
        cmp         esi,r10d
        jae         near ptr M01_L09
        vmovupd     ymm0,[rdx+rax+10]
        cmp         eax,r9d
        jae         near ptr M01_L09
        cmp         esi,r9d
        jae         near ptr M01_L09
        vmovupd     ymm1,[rcx+rax+10]
        vpcmpeqb    ymm0,ymm1,ymm0
        vpmovmskb   esi,ymm0
        cmp         esi,0FFFFFFFF
        jne         near ptr M01_L06
        add         eax,20
        cmp         eax,r11d
        jl          short M01_L00
```

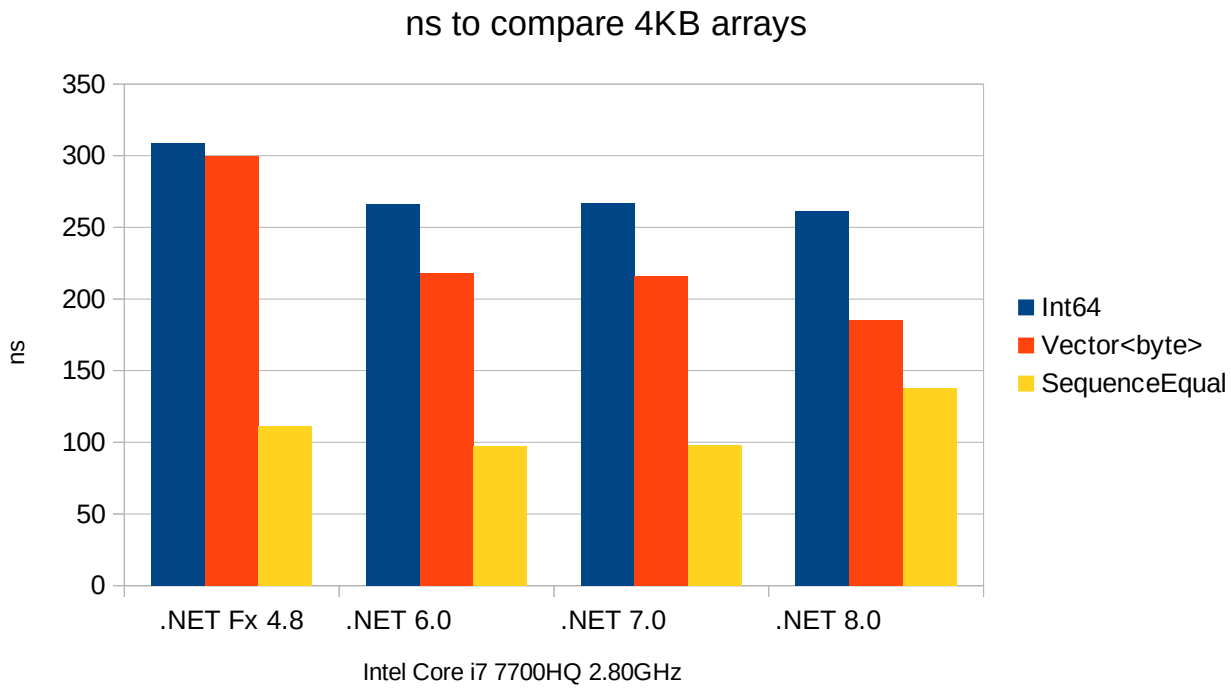
Низкоуровневые оптимизации скорости

4. AVX-сравнение через MemoryExtensions.SequenceEqual (Nuget System.Memory)

```
var span1 = buf1.AsSpan();
var span2 = buf2.AsSpan();
return span1.SequenceEqual(span2);
```

Низкоуровневые оптимизации скорости

Сравнение 4К-массивов тремя разными функциями BenchmarkDotNet



Ускорение в ~2 раза от SequenceEqual

Низкоуровневые оптимизации скорости

AVX Vector<byte> vs SequenceEqual в дизассемблере (.NET 6.0)

SequenceEqual	CompareBuffersVectorized

M01_L01:
 	
vmovdqu ymm0,ymmword ptr [rcx+rax]
 	
vpcmpeqb ymm0,ymm0,[rdx+rax]
vpmovmskb r9d,ymm0
cmp r9d,0FFFFFFFF
jne short M01_L06
add rax,20
cmp r8,rax
ja short M01_L01
	M01_L00:
	cmp eax,r10d
	jae near ptr M01_L09
	lea esi,[rax+1F]
	cmp esi,r10d
	jae near ptr M01_L09
	vmovupd ymm0,[rdx+rax+10]
	cmp eax,r9d
	jae near ptr M01_L09
	cmp esi,r9d
	jae near ptr M01_L09
	vmovupd ymm1,[rcx+rax+10]
	vpcmpeqb ymm0,ymm1,ymm0
	vpmovmskb esi,ymm0
	cmp esi,0FFFFFFFF
	jne near ptr M01_L06
	add eax,20
	cmp eax,r11d
	jl short M01_L00

Выбор оптимального алгоритма хэширования

Хэширование файлов

h = hash(data)


```
SHA1("Hello")  => f7ff9e8b7bb2e09b70935a5d785e0cc5d9d0abf0
SHA1("Hello.") => 9b56d519ccd9e1e5b2a725e186184cdc68de0731
```

Выбор оптимального алгоритма хэширования

Хэширование файлов

- Хэш-функции вычисляют значение фиксированной длины N бит (хэш) для данных произвольной длины
- Длина хэша фиксирована => отсюда конечное число значений
- Количество комбинаций входных данных бесконечно => отсюда коллизии

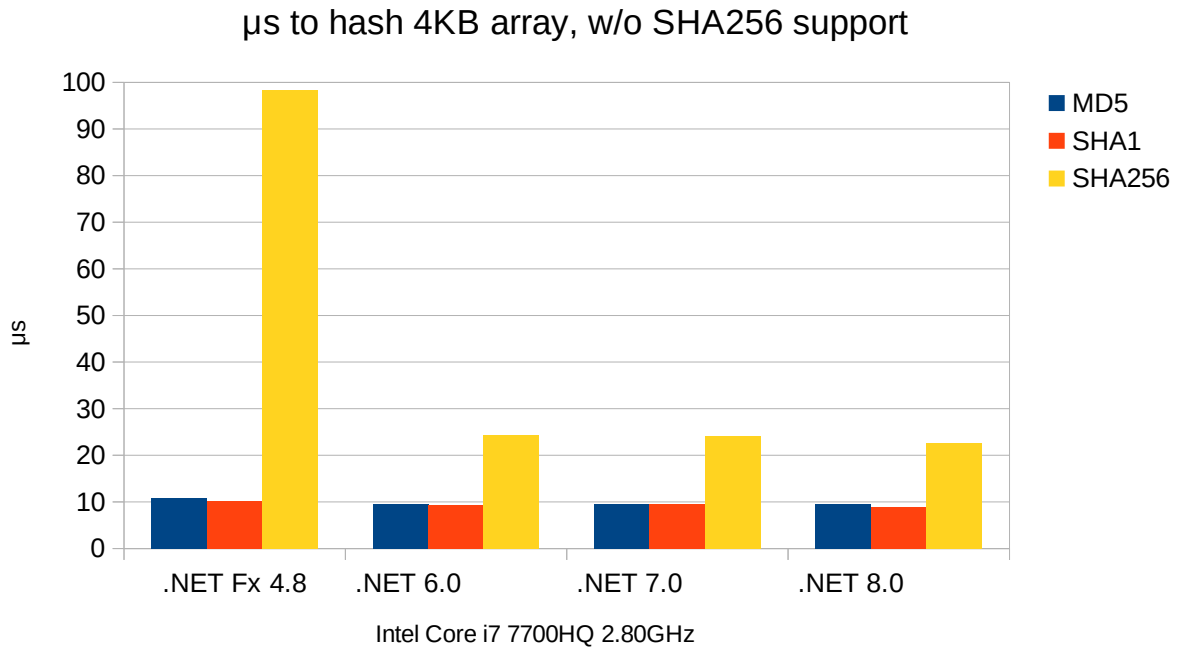
Выбор оптимального алгоритма хэширования

Хэширование файлов

- Хэш-функции
  - криптографические и некриптографические
  - разные по скорости
  - разные по длине хэша
  - разные по качеству распределения
  - разные по частоте коллизий

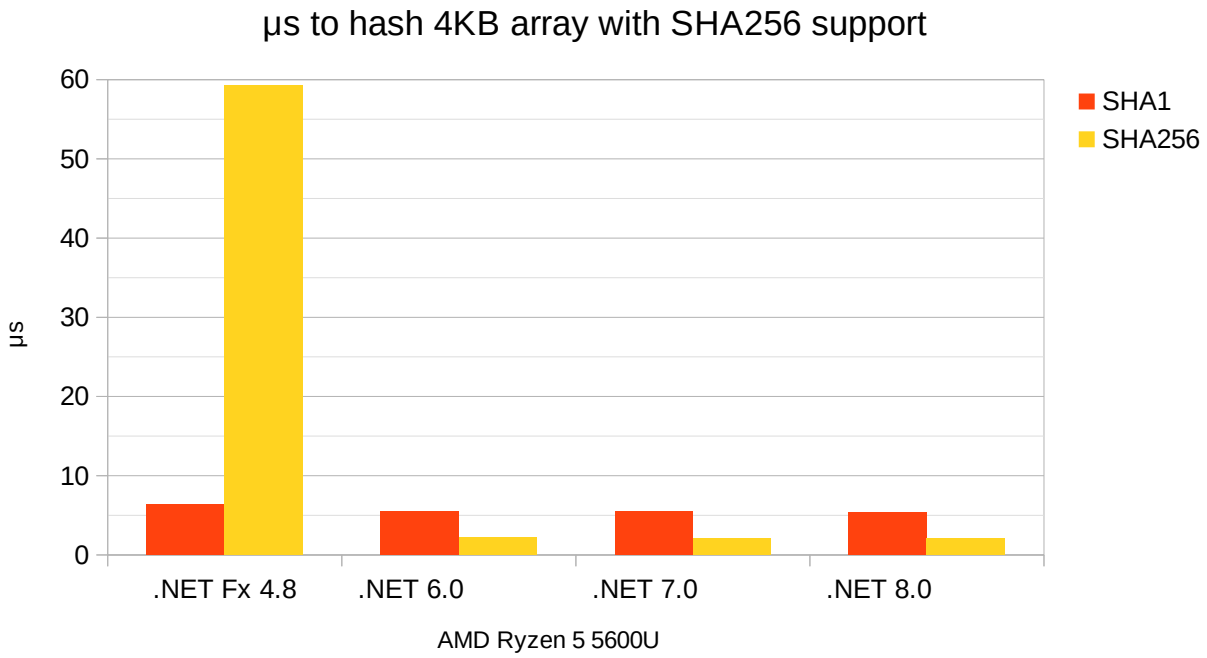
Выбор оптимального алгоритма хэширования

Сравнение SHA1 и SHA256 в BenchmarkDotNet на процессоре без поддержки Intel SHA extensions



Выбор оптимального алгоритма хэширования

Сравнение SHA1 и SHA256 в BenchmarkDotNet на процессоре с поддержкой Intel SHA extensions



	SHA1	SHA256	Ratio
.NET Fx 4.8	6,4 µs	59,3 µs	923%
.NET 6.0	5,5 µs	2,1 µs	39%
.NET 7.0	5,4 µs	2,1 µs	39%
.NET 8.0	5,4 µs	2,1 µs	39%

Выбор оптимального алгоритма хэширования

Некриптографический хэш-алгоритм XxHash128

Реализован в пакете System.IO.Hashing 8.0 в 2023 году

Достоинства:

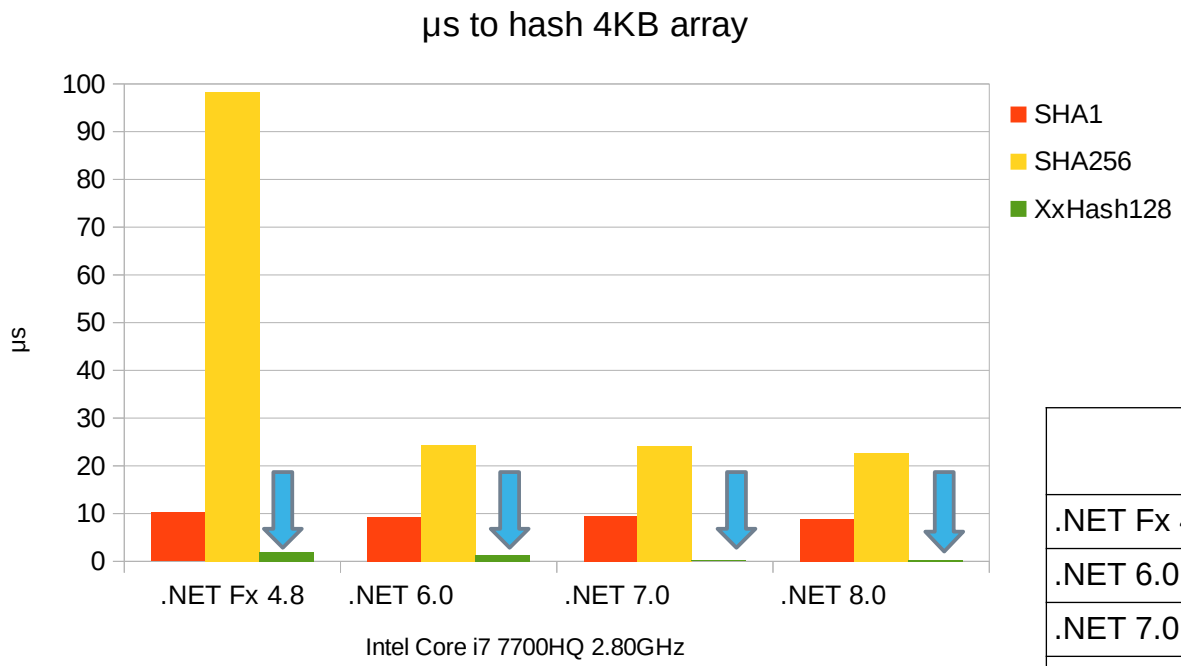
- Скорость, оптимизирован для SSE2/AVX
- Длина 128 бит = 16 байт
- Низкий рейтинг коллизий

https://github.com/Cyan4973/xxHash/wiki/Collision-ratio-comparison

Algorithm	Input Len	Nb Hashes	Expected	Nb Collisions	Notes
MD5 low 64-bit	256	100 Gi	312.5	301	*very long test : 58h !*
XXH128 low 64-bit	256	100 Gi	312.5	314	
XXH128 high 64-bit	512	100 Gi	312.5	291	
XXH128	1024	100 Gi	0.0	0	

Выбор оптимального алгоритма хэширования

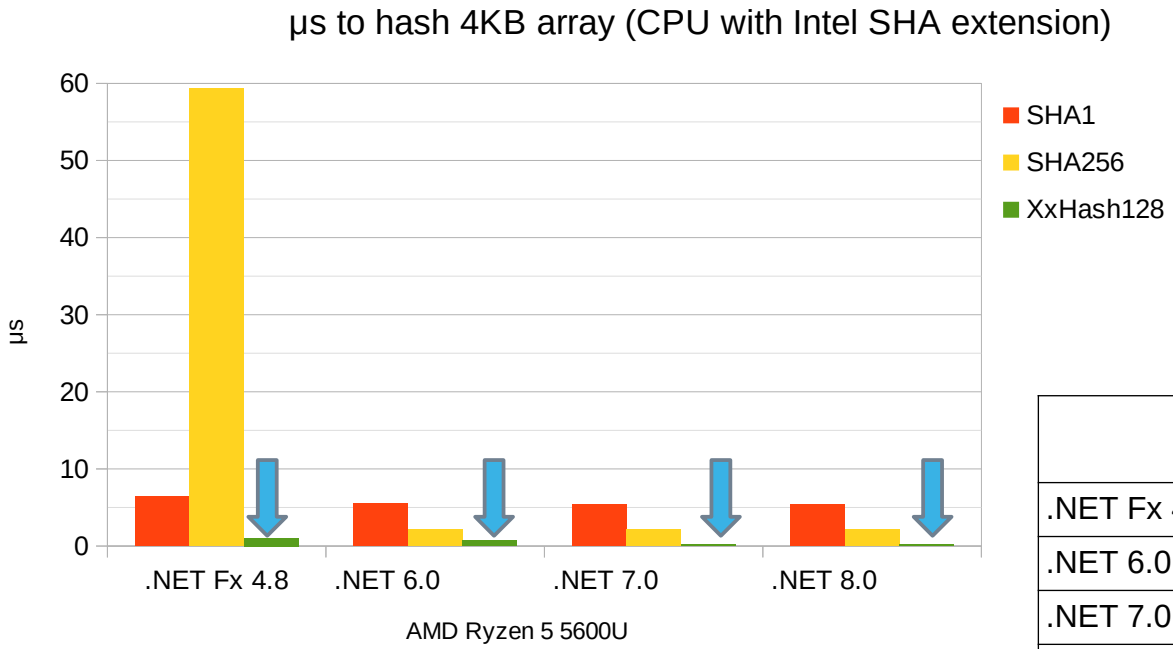
Сравнение SHA1 и SHA256, XxHash128 в BenchmarkDotNet



	SHA1	SHA256	XxHash128	Ratio vs SHA1
.NET Fx 4.8	10,2 µs	98,2 µs	1,90 µs	5,4x
.NET 6.0	9,3 µs	24,4 µs	1,30 µs	7,2x
.NET 7.0	9,4 µs	24,1 µs	0,20 µs	46,3x
.NET 8.0	8,8 µs	22,6 µs	0,27 µs	32,9x

Выбор оптимального алгоритма хэширования

Сравнение SHA1 и SHA256, XxHash128 в BenchmarkDotNet



	SHA1	SHA256	XxHash128	Ratio vs SHA256
.NET Fx 4.8	6,4 µs	59,3 µs	1,04 µs	
.NET 6.0	5,5 µs	2,1 µs	0,70 µs	3,0x
.NET 7.0	5,4 µs	2,1 µs	0,16 µs	13,2x
.NET 8.0	5,4 µs	2,1 µs	0,19 µs	10,9x

Низкоуровневые оптимизации скорости

Сравнение хэш-значений размером 32 байта между собой

1. Сравнение 64-битными словам через fixed-pointer *long

```
fixed (byte* pBuf1 = buf1, pBuf2 = buf2)
{
    var pLong1 = (long*)pBuf1;
    var pLong2 = (long*)pBuf2;

    return pLong1[3] == pLong2[3]
                && pLong1[2] == pLong2[2]
                && pLong1[1] == pLong2[1]
                && pLong1[0] == pLong2[0];
}
```

Низкоуровневые оптимизации скорости

Сравнение хэш-значений размером 32 байта между собой

2. Сравнение 64-битными словам через Span<long>

```
var span1 = buf1.AsSpan();
var span2 = buf2.AsSpan();
var intLong1 = MemoryMarshal.Cast<byte, long>(span1);
var intLong2 = MemoryMarshal.Cast<byte, long>(span2);
return intLong1[3] == intLong2[3] &&
       intLong1[2] == intLong2[2] &&
       intLong1[1] == intLong2[1] &&
       intLong1[0] == intLong2[0];
```

Низкоуровневые оптимизации скорости

Прямой и обратный порядок сравнения элементов Span<long>

Обратный порядок		Прямой порядок		
shr	ecx,3	shr	ecx,3	
shr	r10d,3	shr	r10d,3	
cmp	ecx,3	test	ecx,ecx	[0]([3])
jbe	short M01_L05	je	short M01_L05	
mov	rax,[r8+18h]	mov	rax,[r8]	
cmp	r10d,3	test	r10d,r10d	[0]([3])
jbe	short M01_L05	je	short M01_L05	
cmp	rax,[r9+18h]	cmp	rax,[r9]	
jne	short M01_L02	jne	short M01_L02	
		cmp	ecx,1	[1]
		jbe	short M01_L05	
mov	rax,[r8+10h]	mov	rax,[r8+8]	
		cmp	r10d,1	[1]
		jbe	short M01_L05	
cmp	rax,[r9+10h	cmp	rax,[r9+8]	
jne	short M01_L02	jne	short M01_L02	

Обратный порядок		Прямой порядок		
		cmp	ecx,2	[2]
		jbe	short M01_L05	
mov	rax,[r8+8]	mov	rax,[r8+10h]	
		cmp	r10d,2	[2]
		jbe	short M01_L05	
cmp	rax,[r9+8]	cmp	rax,[r9+10h]	
jne	short M01_L02	jne	short M01_L02	
		cmp	ecx,3	[3]
		jbe	short M01_L05	
mov	rax,[r8]	mov	rax,[r8+18h]	
		cmp	r10d,3	[3]
		jbe	short M01_L05	
cmp	rax,[r9]	cmp	rax,[r9+18h]	
sete	al	sete	al	
movzx	eax,al	movzx	eax,al	

Низкоуровневые оптимизации скорости

Сравнение хэш-значений размером 32 байта между собой

3. AVX- и SSE2-сравнение через Vector<byte>

```
if (Vector<byte>.Count == 32)
{
     var vector2 = new Vector<byte>(buf2, 0);
     var vector1 = new Vector<byte>(buf1, 0);
     return vector1 == vector2;
}
```

Низкоуровневые оптимизации скорости

Сравнение хэш-значений размером 32 байта между собой

4. SequenceEqual (const 32)

```
var span1 = buf1.AsSpan(0, 32);
var span2 = buf2.AsSpan(0, 32);
return span1.SequenceEqual(span2);
```

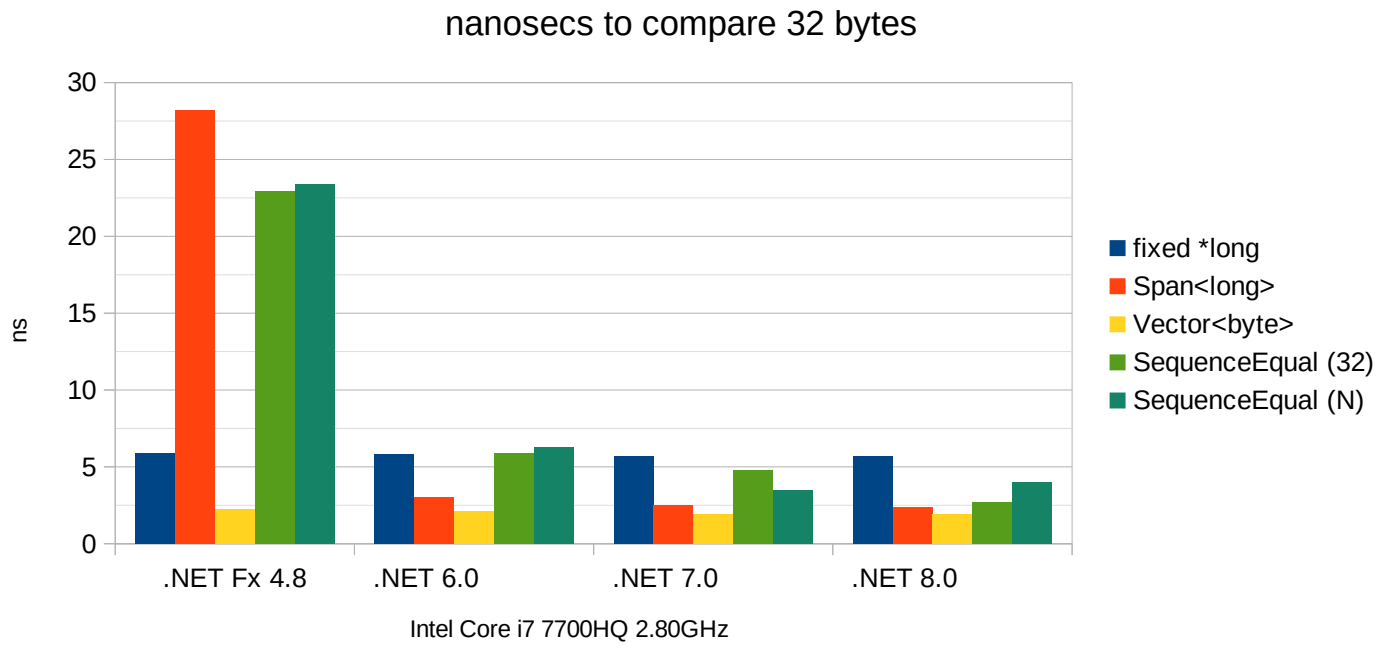
5. SequenceEqual (N)

```
var buf1Span = buf1.AsSpan(0, size);
var buf2Span = buf2.AsSpan(0, size);
var res = buf1Span.SequenceEqual(buf2Span);
return res;
```

Низкоуровневые оптимизации скорости

Сравнение хэш-значений размером 32 байта между собой

Сравнение всех реализаций по скорости в BenchmarkDotNet



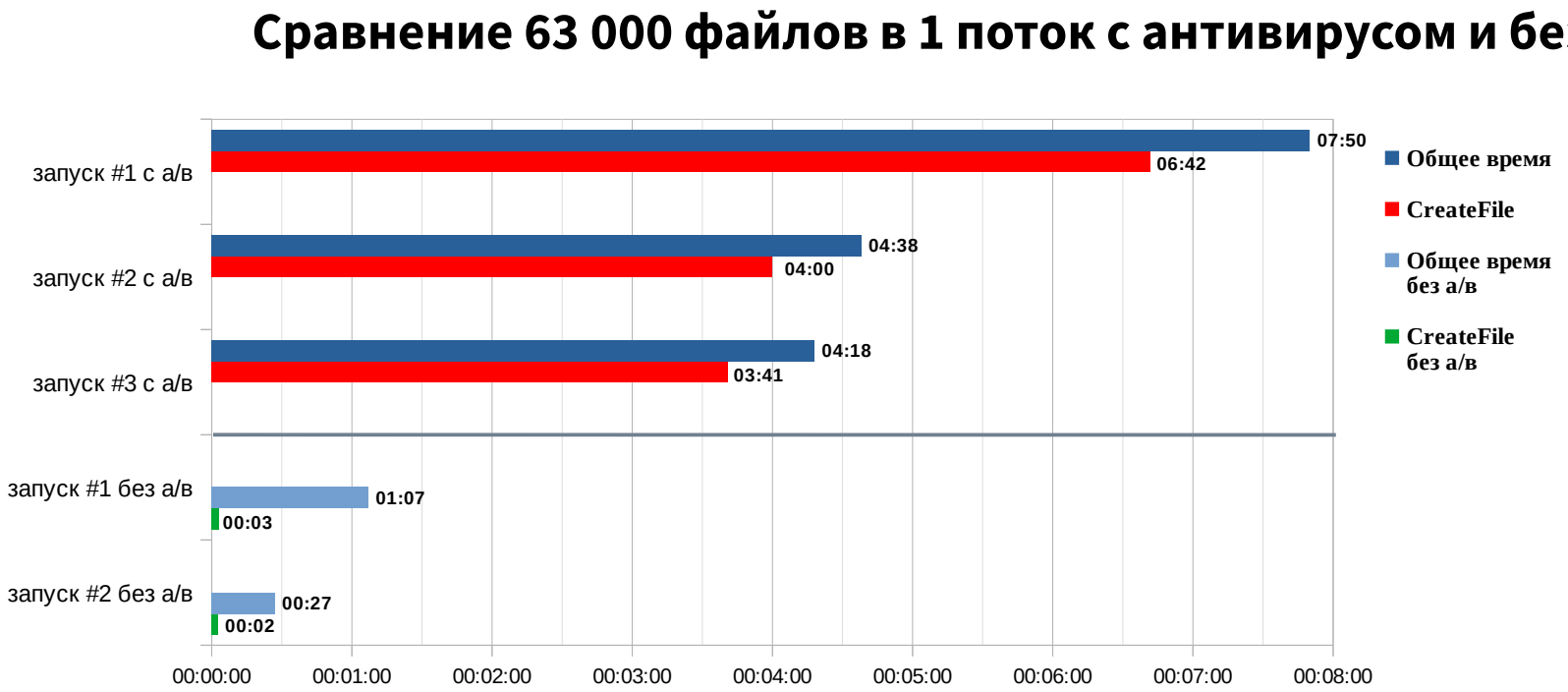
Файловые операции и многопоточность

Какая самая «дорогая»/медленная системная функция в программе?

CreateFile(..) - открытие файла

- Чтение с диска файловых атрибутов метаданных из таблиц
- Проверка прав доступа пользователя
- Самая ресурсоёмкая операция: проверка антивирусом(!)

Файловые операции и многопоточность



4 минуты = в среднем ~4 мс на 1 файл

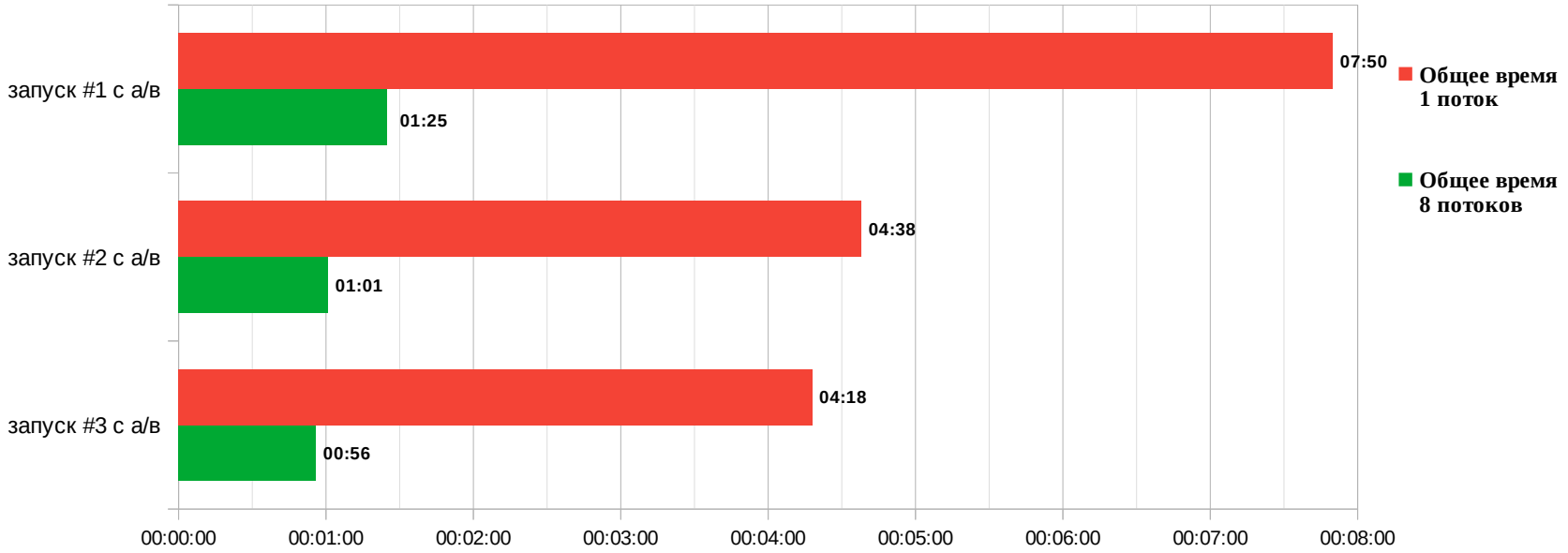
Файловые операции и многопоточность

Оптимизации файловых операций

1. Сокращение количества открытий файлов
2. Совмещение сравнения с хэшированием
3. Закрытие файловых стримов происходит в фоновом потоке
4. Многопоточность для SSD и/или независимых HDD

Файловые операции и многопоточность

Прирост от многопоточности (4C/8T) при поиске дубликатов на SSD



Файловые операции и многопоточность

Задачи при синхронизации доступа потоков к дискам

- Определить, какие физические диски в системе являются SSD, какие HDD, а какие прочими носителями (через **WinAPI**, **WMI**)
- Определить разделы носителей и назначенные им буквы (через **WMI**)
- Обеспечить синхронизацию одновременного доступа с учётом типа носителя

Файловые операции и многопоточность

Кто знает, для чего эти металлические стержни?

Файловые операции и многопоточность

Это жезлы (**tokens**) системы блокировки ж/д перегона

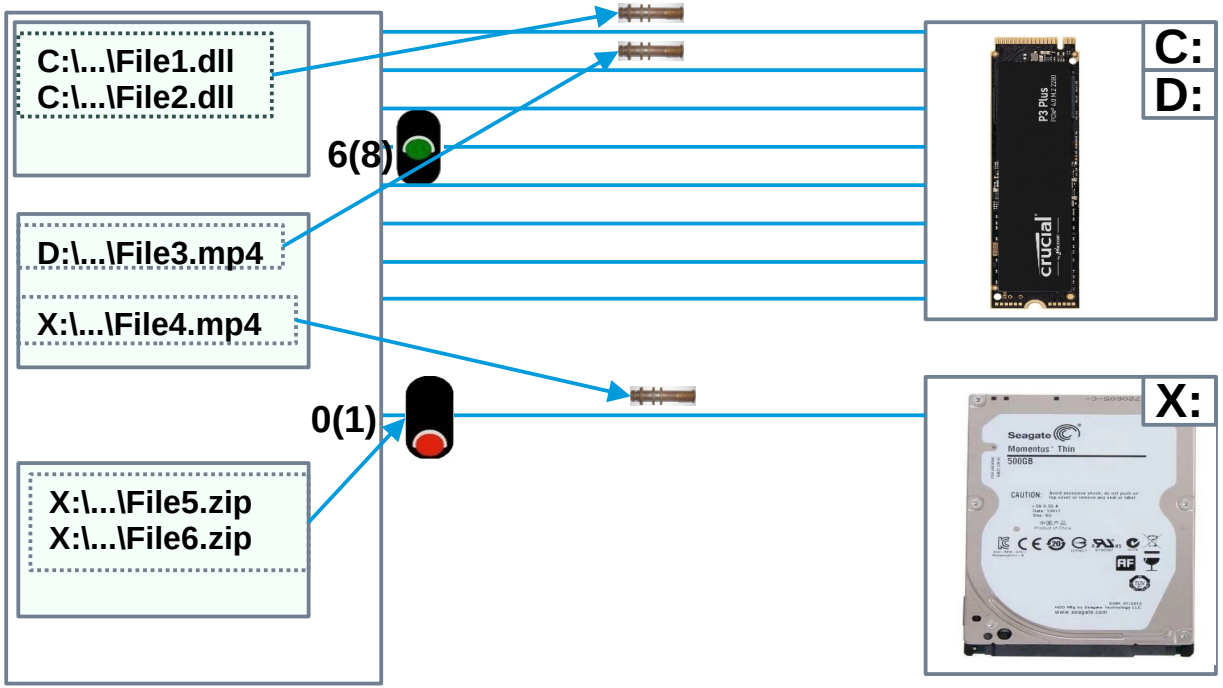
Файловые операции и многопоточность

Распараллеливание сравнения файлов

- Файлы группируются по размеру
- Группы файлов одинакового размера сравниваются параллельно через **Parallel.ForEach()**
- Ограничение доступа к носителям регулируется через семафоры
- На каждый физический носитель выделяется отдельный семафор (initialCount = 1 для **HDD** или CPU Count для **SSD**)
- На каждое сравнение файлов выдаётся токен со ссылкой на семафор, который освобождает блокировку(и) после сравнения

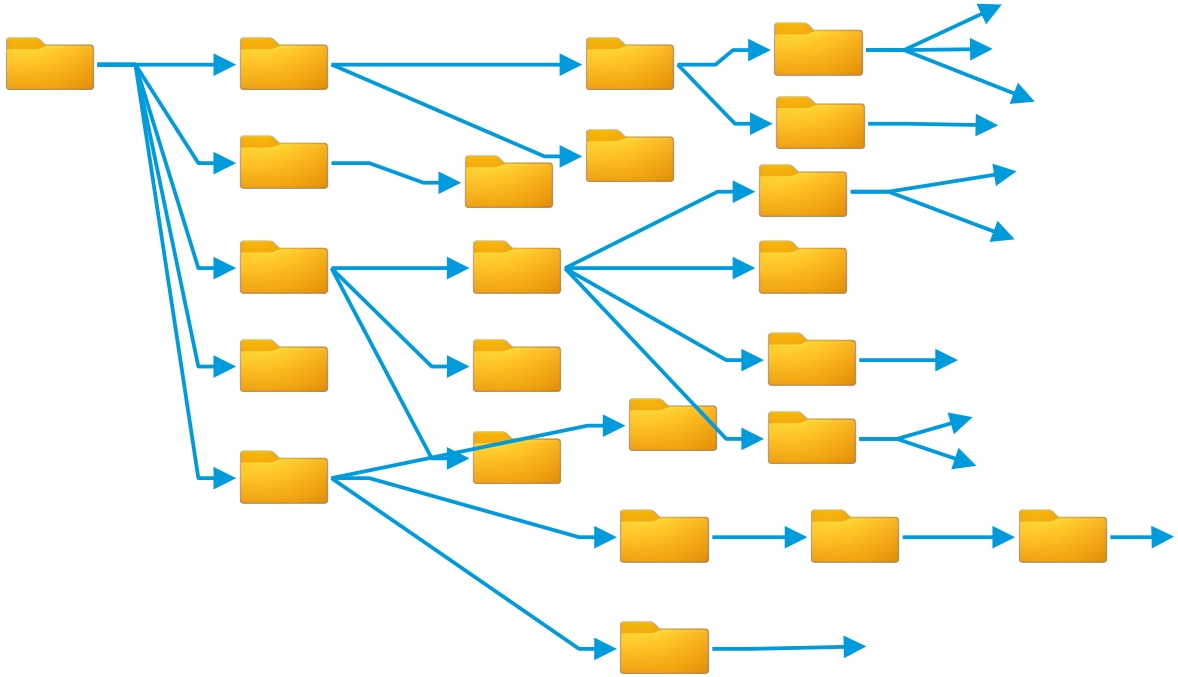
Распараллеливание при сравнении файлов

О семафорах и токенах в программе.



Файловые операции и многопоточность

Параллельное рекурсивное сканирование каталогов



Файловые операции и многопоточность

- **Параллельное рекурсивное сканирование каталогов**

```
internal sealed class InterlockedInt
{
    private int _value;

    public int Value => _value;

    public InterlockedInt(int initValue)
    {
        _value = initValue;
    }

    public int Inc() => Interlocked.Increment(ref _value);

    public int Dec() => Interlocked.Decrement(ref _value);

    public int Add(int val) => Interlocked.Add(ref _value, val);
```

Файловые операции и многопоточность

Параллельное сканирование каталогов

```
// Scan start

int maxDegreeOfParallelism = 1;
if (searchSettings.SelectedProfile.IsScanParallelismAvailable)
{
    var driveLock = _driveAccessManager.GetDriveAccessSemaphore(rootFolder);
    maxDegreeOfParallelism = driveLock.MaxDegreeOfParallelism;
}

var counter = new InterlockedInt(maxDegreeOfParallelism);

// Enter into the root folder
ScanPath(rootFolder, recursive, searchSettings, counter);
```

Файловые операции и многопоточность

- **Параллельное рекурсивное сканирование каталогов**

```
private void ScanPathInternTaskEnter(AbsolutePath path, …, InterlockedInt counter)
{
    counter.Dec();

    // ...

    try
    {
        ScanPathIntern2(path, recursive, searchSettings, files, counter);
    }
    finally
    {
        counter.Inc();
    }
```

Файловые операции и многопоточность

- **Параллельное рекурсивное сканирование каталогов**

```
// здесь перечисление файлов и каталогов
try
{
    availableDegreeOfParallelism = Math.Max(1, counter.Inc());

    var parallelOptions = new ParallelOptions {
          MaxDegreeOfParallelism = availableDegreeOfParallelism
    };

    Parallel.ForEach(subDirectories, parallelOptions, subDir =>
    {
        ScanPathInternTaskEnter(subDir, true, searchSettings, counter);
    });
}
finally
{
    counter.Dec();
}
```

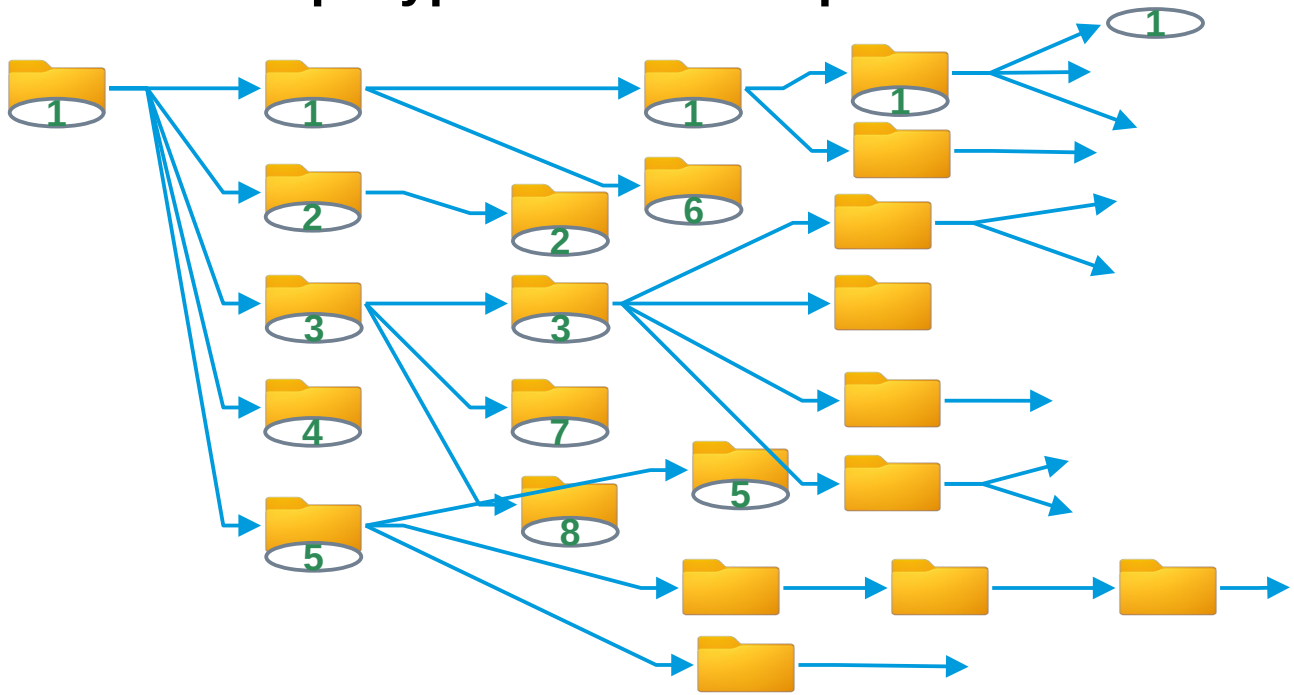
Файловые операции и многопоточность

- **Параллельное рекурсивное сканирование каталогов**

```
private void ScanPathInternTaskEnter(AbsolutePath path, …, InterlockedInt counter)
{
    counter.Dec();

    // ...

    try
    {
        ScanPathIntern2(path, recursive, searchSettings, files, counter);
    }
    finally
    {
        counter.Inc();
    }
```

Файловые операции и многопоточность

- **Параллельное рекурсивное сканирование каталогов**



Parallel.ForEach + MaxDegreeOfParallelism + Interlocked-счётчик

Эффективное использование памяти

Проблематика

- Сотни тысяч, даже миллионы файлов на дисках пользователей
- Ограниченное количество свободной памяти на компьютере пользователя
- Информация о всех найденных файлах в памяти программы

  => OutMemoryException

Эффективное использование памяти

Что общего в этих строках?

"c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\ProjectTemplates"

"c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\PublicAssemblies"

"c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\ReferenceAssemblies"

Эффективное использование памяти

Общее начало строки пути => избыточность.

"c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\ProjectTemplates"

"c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\PublicAssemblies"

"c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\ReferenceAssemblies"

Эффективное использование памяти

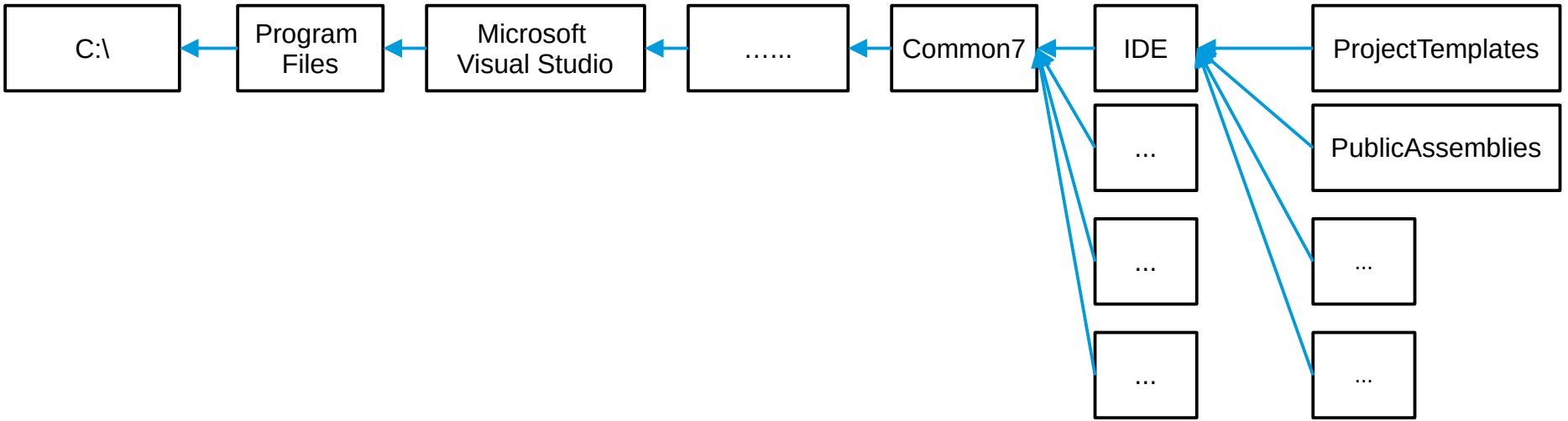
Решение — класс AbsolutePath

```
internal class AbsolutePath : IEquatable<AbsolutePath>, IComparable<AbsolutePath>
{
    public AbsolutePath? Parent { get; }

    public string Name { get; }

    public int Length => GetLength();

    public int Levels => GetLevels();
    // ...
    public AbsolutePath(string name, AbsolutePath? parent)
    {
        Parent = parent;
        Name = GetSharedName(name ?? string.Empty);
    }
```

Эффективное использование памяти

Класс AbsolutePath

Связный список узлов для одного пути, дерево для множества путей



Эффективное использование памяти

Некоторые недостатки AbsolutePath vs string

- Нужно выделять временную строку для WinApi-функций
- Динамическое вычисление длины строк

Эффективное использование памяти

Преимущества AbsolutePath vs string

- Строгая типизация
- Использует значительно меньше памяти при большом количестве файлов
- Ссылки на имя, родительский и корневой каталоги прямо в списке
- Строки одинаковых имен каталогов и файлов не дублируются (подменяются через словарь общих имён)
- Пути сравниваются на равенство между собой быстрее, чем строки

Эффективное использование памяти

Преимущества AbsolutePath vs string

Сравнение на равенство двух таких путей

"**c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\Packages\0009**8"

"**c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\Packages\0009**9"

в BenchmarkDotNet на .NET 8.0

- string.Equals(OrdinalIgnoreCase): **48 нс**
- AbsolutePath.EqualsIgnoreCase: **2 нс**

Эффективное использование памяти

Преимущества AbsolutePath vs string

- Наследование: AbsolutePathDriveRoot - быстрое получение информации о корневой папке и томе по пути.

```
internal sealed class AbsolutePathDriveRoot : AbsolutePath
{
    public override string FileSystemName { get; }

    public override Win32.FileSystemFlags FileSystemFlags { get; }

    public override DriveType DriveType { get; }

    public override string VolumeName { get; }

    public override bool IsOnSsd { get; set; }

    public override uint SectorSize { get; }
```

Эффективное использование памяти

Преимущества AbsolutePath vs string

- Наследование: AbsolutePathSymlinkPoint - встроенная поддержка символьных ссылок

```
internal sealed class AbsolutePathSymlinkPoint : AbsolutePath
{
    public AbsolutePath SymlinkTarget { get; }

    public override AbsolutePath FinalPath => SymlinkTarget;

    public override bool IsSymlink => true;

    public override bool IsSymlinkPoint => true;

    public AbsolutePathSymlinkPoint(string name, AbsolutePath? parent, AbsolutePath symlinkTarget) :
                                    base(name, parent)
    {
        SymlinkTarget = symlinkTarget;
    }
}
```

Эффективное использование памяти

ArrayPool<T>

Вместо статического свойства **ArrayPool<byte>.Shared** я использую 1 разделяемый экземпляр **ArrayPool<byte>.Create()**

Преимущества экземпляра ArrayPool<byte>.Create()

- Меньше суммарный объём памяти, выделенной на массивы
- Позволяет освободить ссылку на выделенную память после использования
- В моих сценариях скорость одинакова на .NET 8.0, но реализация через Create() быстрее на .NET Fx 4.8

Эффективное использование памяти

ListPool<T>

Хранит **List<T>**, как и **ArrayPool<T>** используется для уменьшения трафика памяти

```
internal sealed class ListPool<T>
{
    public static int MaxCount { get; set; } = Math.Max(32, Environment.ProcessorCount * 2);

    public static ListPool<T> Shared { get; } = new ();

    private ConcurrentBag<List<T>> _listContainer = new ();
    private int _countInContainer = 0;

    [ThreadStatic] private static List<T>?[]? _slot;

    private static List<T>?[] CacheSlot { ...}

    public List<T> Rent() { … }

    public void Return(ref List<T>? listRef) { … }
```

Эффективное использование памяти

ListPool<T>

Rent() - аренда

```
var slot = CacheSlot; // <= Read from TLS (threal local storage)
var list = slot[0];
if (list is not null)
{
    slot[0] = null;
    return list;
}

// Read from ConcurrentBag
if (_countInContainer > 0 && _listContainer.TryTake(out list))
{
    Interlocked.Decrement(ref _countInContainer);
    return list;
}

return new List<T>();
```

Эффективное использование памяти

ListPool<T>

Rent() - аренда

```
var slot = CacheSlot; // <= Read from TLS (threal local storage)
var list = slot[0];
if (list is not null)
{
    slot[0] = null;
    return list;
}

// Read from ConcurrentBag
if (_countInContainer > 0 && _listContainer.TryTake(out list))
{
    Interlocked.Decrement(ref _countInContainer);
    return list;
}

return new List<T>();
```

Эффективное использование памяти

ListPool<T>

Rent() - аренда

```
var slot = CacheSlot; // <= Read from TLS (threal local storage)
var list = slot[0];
if (list is not null)
{
    slot[0] = null;
    return list;
}

// Read from ConcurrentBag
if (_countInContainer > 0 && _listContainer.TryTake(out list))
{
    Interlocked.Decrement(ref _countInContainer);
    return list;
}

return new List<T>();
```

Эффективное использование памяти

ListPool<T>

Return() - возврат

```
var list = listRef;
listRef = null;
if (list is null || _countInContainer >= MaxCount) { return; }

list.Clear();

var slot = CacheSlot; // <= Read from TLS (threal local storage)
if (slot[0] is null)
{
     slot[0] = list;
     return;
}

if (_countInContainer < MaxCount)
{
     _listContainer.Add(list);
     Interlocked.Increment(ref _countInContainer);
}
```

Эффективное использование памяти

ListPool<T>

Return() - возврат

```
var list = listRef;
listRef = null;
if (list is null || _countInContainer >= MaxCount) { return; }

list.Clear();

var slot = CacheSlot; // <= Read from TLS (threal local storage)
if (slot[0] is null)
{
     slot[0] = list;
     return;
}

if (_countInContainer < MaxCount)
{
     _listContainer.Add(list);
     Interlocked.Increment(ref _countInContainer);
}
```

Эффективное использование памяти

ListPool<T>

```
[ThreadStatic] private static List<T>?[]? _slot;

private static List<T>?[] CacheSlot
{
    get
    {
        var slot = _slot;

        // Write to TLS (threal local storage)
        if (slot is null) { _slot = slot = new List<T>[1]; }

        return slot;
    }
}
```

Эффективное использование памяти

ListPool<T>

RentedListWrapper<T>

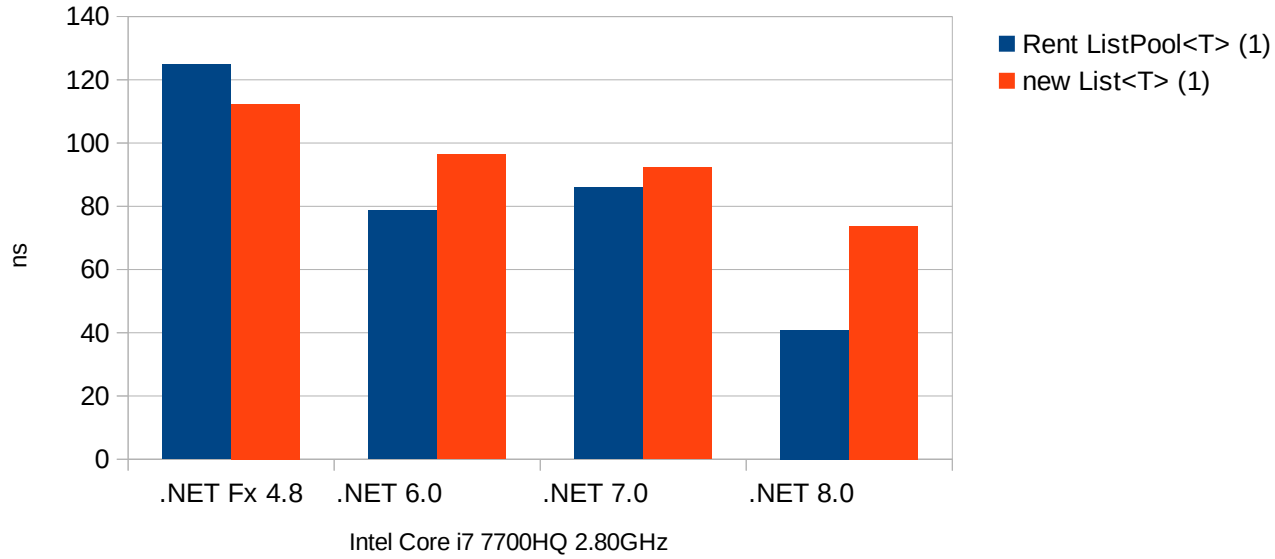
```
internal struct RentedListWrapper<T> : IDisposable
{
    private List<T>? _rentedList;

    public List<T> List => _rentedList ?? new List<T>();

    public RentedListWrapper(List<T> cachedList)
    {
        _rentedList = cachedList;
    }

    public void Dispose()
    {
        ListPool<T>.Shared.Return(ref _rentedList);
    }
}
```

```
using var wrappedList = ListPool<FileNameSize>.Shared.RentWrapped();
```

Эффективное использование памяти

ListPool<T>

- Бенчмарк скорости ListPool<T>.**Rent()** vs **new** List<T>() (для 1 списка с добавлением 6-ти объектов в каждый список)

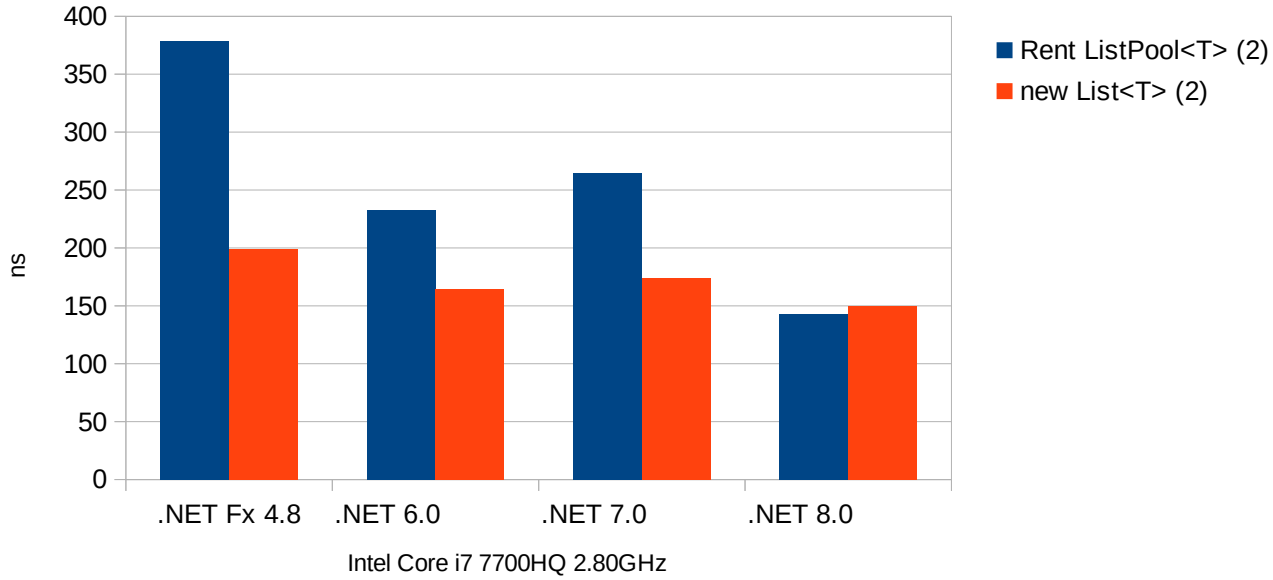


	ListPool<T> (1)	Alloc	new List<T> (1)	Alloc
.NET Fx 4.8	124,8 ns	0	112,3 ns	185 B
.NET 6.0	78,7 ns	0	96,4 ns	176 Б
.NET 7.0	85,9 ns	0	92,3 ns	176 Б
.NET 8.0	40,9 ns	0	73,7 ns	176 Б

Эффективное использование памяти

ListPool<T>

- Бенчмарк скорости ListPool<T>.**Rent()** vs **new** List<T>() (для 2 списков с добавлением 6-ти объектов в каждый список)

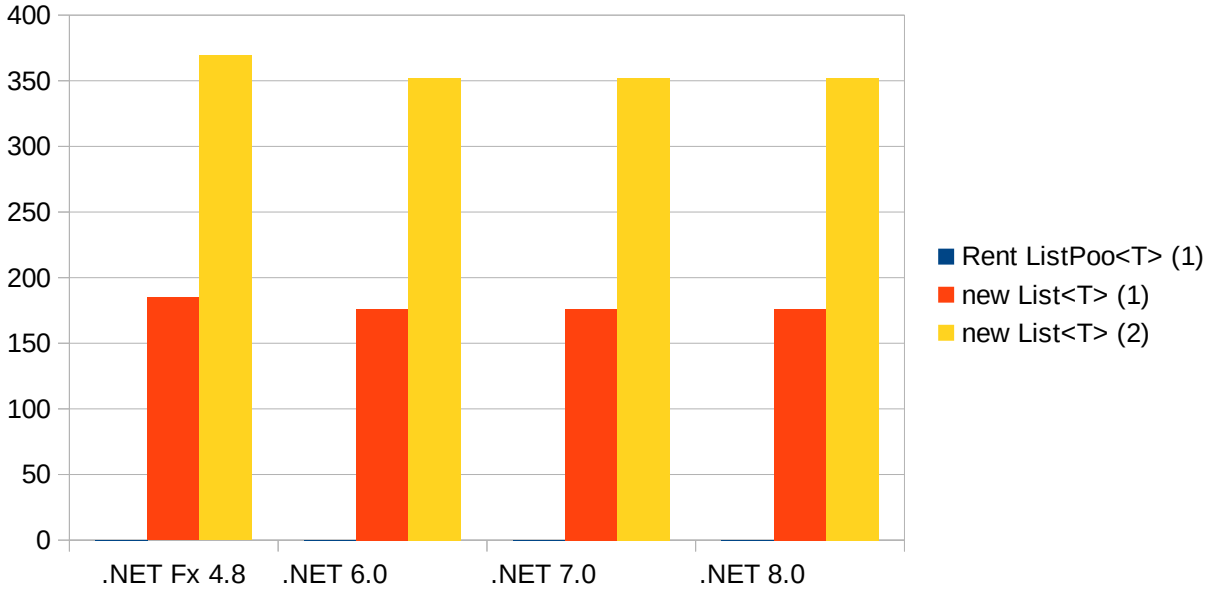


	ListPool<T> (2)	Alloc	new List<T> (2)	Alloc
.NET Fx 4.8	378,3 ns	0	199,1 ns	369 B
.NET 6.0	233,1 ns	0	164,9 ns	352 Б
.NET 7.0	264,8 ns	0	173,9 ns	352 Б
.NET 8.0	142,7 ns	0	149,7 ns	352 Б

Эффективное использование памяти

ListPool<T>

- Выделение памяти ListPool<T>.**Rent()** vs **new** List<T>() для 1 и 2 списков с добавлением 6-ти объектов в список



Эффективное использование памяти

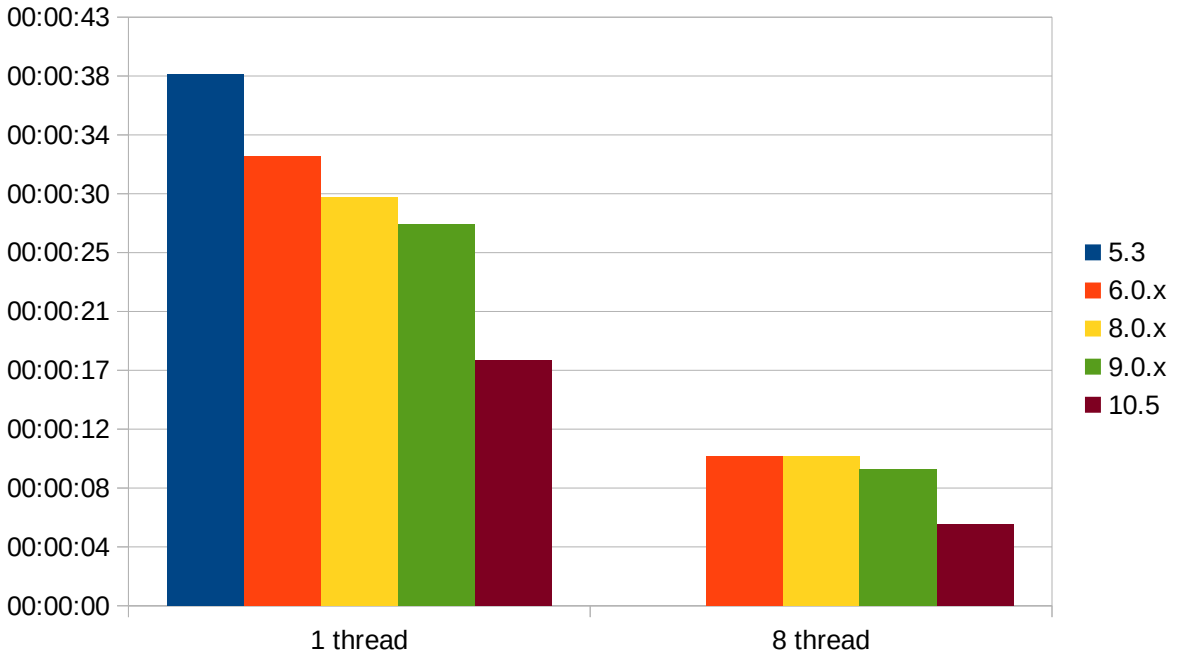
StringBuilderPool

Хранит StringBuilder, аналогичен ListPool<T>

```
internal sealed class StringBuilderPool
{
    public static int MaxCount { get; set; } = 32;
    public static StringBuilderPool Shared { get; private set; } = new StringBuilderPool();

    private int _countInContainer;

    private ConcurrentBag<StringBuilder> _stringBuilders = new ();

    [ThreadStatic] private static StringBuilder?[] _slot;

    private static StringBuilder?[] CacheSlot{ … }

    public StringBuilder Rent() { … }

    public void Return(StringBuilder? sb){ … }
```

Заключение

Комплексный прирост от всех оптимизаций разных версий



Заключение

Выводы

- Используйте **XxHash128** для хэширования данных, когда нужен быстрый хэш, стойкий к коллизиям
- Используйте **MemoryExtensions.SequenceEqual** для сравнения байтовых массивов большой длины
- **Vector<byte>** ограниченно применим для оптимизаций низкого уровня операций с массивами
- Используйте пулы объектов как **ListPool<T>**, чтобы реже выделять память при работе со списками
- Творческий и креативный подход к программированию делает работу интересней, а программы быстрее :)

Спасибо за внимание!

Вопросы?

Контакты:

email: yurymalich@yandex.ru

@Yury_Malich

web: http://www.malich.org/duplicate_searcher

XING: https://www.xing.com/profile/Yury_Malich

GitHub https://github.com/ymalich

https://github.com/ymalich/ObjectPools/