VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

OPTIMALIZACE LINQ PRO .NET

.NET LINQ OPTIMIZATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **DANIEL ŠERÝ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JAN PLUSKAL,**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Šerý Daniel**
Program: Informační technologie
Název: **Optimalizace LINQ pro .NET**
.NET's LINQ Optimization
Kategorie: Algoritmy a datové struktury

Zadání:

1. Nastudujte nezbytné prerekvizity pro optimalizaci LINQ v .NET, zejména pak Microsoft.Build.Utils.Task, System.Reflection.Emit, System.Data.Objects.CompiledQuery.
2. Vytvořte sadu testů a benchmarků, které ověří výkonnost LINQ pro různé verze .NET CLR, jako jsou .NET Framework, .NET CORE a Mono. Verze CLR zvolte po konzultaci s vedoucím.
3. Na základě měření navrhněte způsoby optimalizace, např. nahrazování LINQ dotazů za odpovídající for cykly, inlinování lambda výrazů, či využití známých dat pro optimalizaci for cyklů.
4. Návrh implementujte jako knihovnu a změřte urychlení. Vytvořenou knihovnu distribuujte jako NuGet balíček.
5. Diskutujte dosažené výsledky a navrhněte další možné rozšíření.

Literatura:

- Kokosa, K. Pro .NET Memory Management. 2018. Apress, Berkeley, CA.
- Akinshin, A. Pro .NET Benchmarking. Chicago. 2019. Apress, Berkeley, CA.
- Schwichtenberg, H. (2018). Dynamic LINQ Queries. In *Modern Data Access with Entity Framework Core* (pp. 295-304). Apress, Berkeley, CA.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Pluskal Jan, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 14. května 2020
Datum schválení: 25. října 2019

Abstrakt

Tato práce se zabývá LINQ (Language integrated query) a řeší možnosti jeho implementace a optimalizace v jazyce C#. Je vybrána a implementována metoda přepisu dotazů na procedurální kód před překladem programu. Cílem práce je poskytnout LINQ využitelný, i při potřebě velké rychlosti.

Pro vytvořený program přepisující dotazy LINQ, bylo pro většinu operátorů dosaženo zrychlení od 1.2x do 20x rychlosti System.Linq v závislosti na prováděném algoritmu, zdroji dat a množství poskytnutých informací přepisovacímu programu.

Abstract

This thesis deals with LINQ (Language integrated query) and investigates possibilities of its implementation and optimization in C# language. Method of rewriting of query to procedural code is chosen and implemented. The goal is to provide a LINQ that can be used in code with the need for high speed.

Regarding the program created for rewriting LINQ queries, the performance of most operators has been increased by 1.2x to 20x of System.Linq speed depending of rewritten algorithm, data source and provided information to rewriting program.

Klíčová slova

LINQ, C#, .NET, přepis kódu, analýza kódu, optimalizace, BenchmarkDotNet, algoritmy, datové struktury, dynamické dotazy, paralelní programování, SIMD, vkládání metod

Keywords

LINQ, C#, .NET, code rewriting, code analysis, optimization, BenchmarkDotNet, algorithms, data structures, dynamic queries, parallel programming, SIMD, method inlining

Citace

ŠERÝ, Daniel. *Optimalizace LINQ pro .NET*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Pluskal,

Optimalizace LINQ pro .NET

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Pluskala. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Daniel Šerý
28. května 2020

Poděkování

Chtěl bych poděkovat panu Ing. Janu Pluskalovi za jeho vedení, rady a odbornou pomoc.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Častým úkolem programátora je implementace dotazů nad kolekcemi. Příkladem může být filtrace prémiových uživatelů z databáze, nebo sečtení množiny čísel. Možných operací nad kolekcí může být mnoho, od řazení, třídění, filtrace, různé modifikace dat, spojování více kolekcí, nebo třeba shlukování prvků podle určité vlastnosti. Většina dotazů odpovídá cyklu procházejícímu všechny prvky a provádějící pro každý prvek stejné operace.

Při programování zmíněných dotazů jsou pro programátora C# kladeny obvykle dvě možnosti. Buď použije systémový LINQ, nebo dotaz napíše pomocí procedurálního kódu. `System.Linq` je knihovna předpřipravených částí dotazu (operátorů) a jejich řetězením lze složit výsledný dotaz. Použití LINQ má výhodu jednoduché syntaxe, množství implementovaných funkcí, jednotný přístup ke všem typům dat, jednotnost dotazů a odložené vyhodnocování (vyhodnocení až v případě potřeby výsledku). Velkou nevýhodou ale je, že je v mnoha situacích znatelně pomalejší a vytváří více alokací než odpovídající imperativní kód. (viz kapitola č. 3).

Napsáním imperativního kódu namísto LINQ dotazu se docílí potřebného zrychlení, ale zároveň se tím zpomalí vývoj, opakuje a znepřehledňuje se kód, je náročné ho upravovat a i při malé změně dotazu se musí změnit velká část kódu. Pro některé dotazy navíc není lehké napsat odpovídající kód, protože například není známá výsledná velikost kolekce, nebo programátor nezná potřebný algoritmus. Ještě těžší je napsat optimální kód.

Jelikož se chceme vyhnout psaní imperativního kódu a `System.Linq` nám to v různých situacích znemožňuje (většinou při potřebě velké rychlosti), je potřeba najít alternativu, která by spojila rychlost a malý počet alokací imperativního kódu a zároveň výhody jazyka LINQ a umožnila by používat LINQ i na místě, kde je potřeba vysoké rychlosti. Cílem této práce bude tedy prozkoumat současné i možné implementace LINQ a vybrat, naprogramovat, případně zlepšit z nich takovou, která by tento problém řešila. Tím se pak sníží čas trávený na optimalizacích dotazů a vznikne tak přehlednější a lépe spravovatelný kód a sníží se tedy náklady projektů.

V kapitole 2 budu popisovat programovací jazyk C#, platformu .NET, LINQ a jak spolu tyto pojmy souvisí abych položil teoretický základ na kterém budu v této práci stavět. V kapitole 3 na to navážu popisem existujících implementacích LINQ a popisem jejich přínosů a úskalí a v kapitole 4 na to navážu možnými optimalizacemi dotazů LINQ. V kapitole 5 je popsána implementace LinqRewrite (nová implementace LINQ navržená v této bakalářské práci) a v kapitole 6 jsou popsány výsledky testování funkčnosti a rychlosti této implementace.

Kapitola 2

C#, .NET a LINQ

Abych se mohl vůbec zabývat optimalizací LINQ pro C# .NET, musím nejdříve popsat jazyk C#, vysvětlit postup jeho překladu a jeho souvislost s .NET. Také musím popsat jazyk LINQ, který byl do C# integrován, jeho princip, způsob použití a jeho přínos. Touto kapitolou tedy položím základ, na kterém můžu v dalších kapitolách stavět.

Jazyk C# a jeho první verze byl vydán roku 2002 firmou Microsoft a od té doby je vyvíjen se současnou verzí C# 8. Od verze 1.0 bylo doprogramováno mnoho nových konstrukcí, jako generické typy, asynchronní programování, nebo interpolace textových řetězců [4]. C# je objektově orientovaný jazyk, vycházející z jazykové rodiny C, a poskytuje funkcionalitu k tvorbě stabilních aplikací, příkladem může být tzv. „garbage collector“, který uvolňuje nepotřebnou, nebo nedosažitelnou paměť. Další poskytovaná funkcionalita je například zpracování výjimek, nebo typově bezpečný systém [24]. Kvůli tomu je lehčí se vyhnout častým chybám programátorů, jako je únik paměti, nebo špatnému používání ukazatelů.

Základní konstrukcí jazyka C# jsou třídy, které by šlo popsat jako předpis objektu s daty a množiny operací, které lze nad tím objektem provést. Třídy můžou mezi sebou dědit (přejímat vlastnosti). Každá třída může dědit z jednoho přímého předka a z neomezeného počtu nepřímých předků (předek předka). Třída může implementovat neomezený počet rozhraní, kde rozhraní, je předpis definicí metod, které implementující třída musí obsahovat. Pro rozdělení tříd se stejným jménem můžeme použít jiný jmenný prostor. Třída může obsahovat data a metody. Pokud je metoda, nebo data třídy označená modifikátorem `static` patří pak samotnému typu třídy, namísto konkrétní instanci. Pokud všechna data a metody třídy jsou statické, lze udělat celou třídu statickou a tím zakázat vytváření instance dané třídy.

Ve výpisu 2.1 lze vidět zdrojový kód programu `HelloWorld` naprogramovaného v jazyce C#. Všechny metody C# programu musí patřit do nějaké třídy a každý spustitelný program musí mít statickou metodu `Main`, která je vstupním bodem programu. V kódu je použit jmenný prostor `System`, aby mohla být volána metoda `Console.WriteLine`.

2.1 Platforma .NET

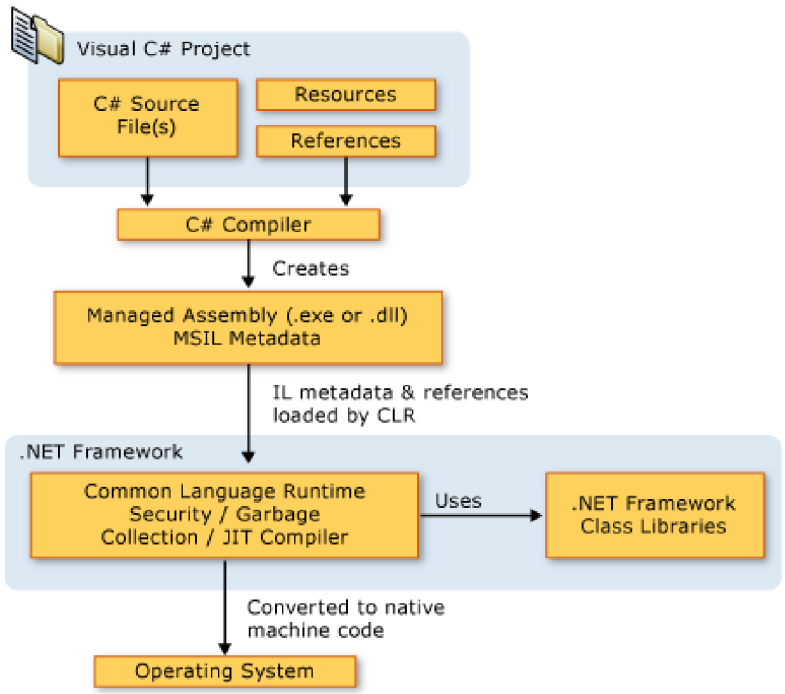
Jazyk C# není překládán (v běžném nastavení) do nativního kódu, jako například u jazyků C, nebo C++, protože pro různé architektury musí být vytvořen různý nativní kód a tedy jednou sestavená aplikace není kompatibilní se všemi architekturami. To řeší platforma .NET, která byla poprvé vydána s jazykem C# jako .NET Framework firmou Microsoft roku 2002 [18]. Jazyky .NET jsou překládány pomocí kompilátoru do mezikódu CIL (Společný mezijazyk), který je následně spouštěn uživatelem pomocí CLR (Společný jazykový

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

Výpis 2.1: Program HelloWorld naprogramovaný v jazyce C#

runtime). CLR pomocí JIT-kompilátoru (kompiluje až za běhu aplikace) převádí CIL do nativního kódu a následně ho spouští [26]. Výhodou je přenositelnost programu mezi různými architekturami, ale nevýhodou je pro uživatele nutnost instalace CLR a vyšší režie CLR a JIT-kompilace, než samotné provádění nativního kódu. Kromě jazyku C#, lze použít nad .NET také jazyky C++/CLI, F#, nebo Visual Basic .NET. Proces překladu je znázorněn na obrázku 2.1.



Obrázek 2.1: Průběh překladu pro .NET Framework převzato z [20]

Jelikož JIT-kompilace zabírá čas, lze u Mono [10] a .NET Native [25] udělat kompilaci před během programu. To znamená, že není potřeba kompilovat při běhu programu a je ušetřen čas spouštění aplikace. *Aby byla aplikace 100% nativní, je potřeba použít alternativu k CLR. Toho se snaží docílit projekt CoreRT, který mění způsob překladu a spouštění .NET aplikací (přeloženo a parafrázováno z [28])*.. Při kompilaci do nativního kódu dochází ke ztrátě přenositelnosti aplikace a také se musí brát v potaz, že CoreRT je zatím experimentálním projektem [6].

2.2 Implementace .NET

Existuje více implementací .NET, které určují, jak má JIT-kompilátor přeložit mezikód do nativního kódu. *Každá implementace rozhraní .NET musí zahrnovat jeden nebo více modulů runtime a obsahovat knihovnu tříd, která implementuje .NET Standard (specifikace rozhraní .NET API, které tvoří jednotnou sadu kontraktů, se kterými kompilujete kód) a může implementovat další rozhraní API. Může obsahovat aplikační architektury a vývojové nástroje (Např. .NET Core s CoreCLR a knihovnou základních tříd .NET Core). Společnost Microsoft aktivně vyvíjí a udržuje čtyři implementace v rozhraní .NET: .NET Core, .NET Framework, Mono a UWP (parafrázováno z [2]).* Tedy při splnění zmíněných podmínek lze program považovat za implementaci .NET.

První implementací .NET byl roku 2002 .NET Framework pro platformu Windows ze kterého vycházejí ostatní implementace. Mono byl vydán dva roky poté jako otevřený software s cílem podpory širokého rozsahu architektur a operačních systémů. Důležitou událostí bylo vydání .NET Core roku 2014, který je také otevřený software pro systémy Windows, Linux a macOS [10].

Tyto implementace usilují o zpětnou kompatibilitu. Tedy kód naprogramovaný v .NET Framework 4.6, lze přeložit a výsledná aplikace bude dělat stejné věci i v .NET Framework 4.8, stejně .NET Core 2.1 lze přeložit v .NET Core 3.1 a měl by mít stejné chování. Dopředná kompatibilita, ani kompatibilita mezi implementacemi .NET není zaručena [27]. Kvůli tomu je pro programátora jednodušší s různými verzemi jedné implementace pracovat, ale je náročnější optimalizovat a upravovat kód při programování jednotlivých implementací. Což je pravděpodobně jeden z důvodu vyšší rychlosti .NET Core, než .NET Frameworku, protože je .NET Core novější a při optimalizacích nemuseli dodržovat zpětnou kompatibilitu s verzemi .NET Framework.

Každá z popsaných platforem má přednosti i nedostatky. Na .NET Framework existuje kvůli délce existence nejvíce kódu a knihoven (Např. do 23.9.2019 neexistovala knihovna pro tvorbu okeních aplikací pro Windows v .NET Core). Mono lze spustit na různých architekturách a .NET Core 3.1 disponuje velkou rychlostí a je podporován na platformách Windows, Linux a MacOS. Proto nelze říct, že by jedna implementace byla nejlepší a věnovat se v této bakalářské práci jen jedné z nich. Zároveň kvůli rozsahu práce nemůže být všechen výzkum a výpočty prováděn pro všechny platformy. Proto bude hlavním zaměřením optimalizace .NET Framework, protože pro tuto implementaci pravděpodobně existuje nejvíce kódu a navíc LINQ pro .NET Framework není tolik optimalizovaný, jako u .NET Core.

2.3 Jazyk LINQ

Častým úkolem programátora je implementace dotazů nad kolekcemi. Příkladem může být filtrace prémiových uživatelů z databáze, nebo sečtení množiny čísel. Do .NET Frameworku verze 3.5 bylo potřeba implementovat pro každý dotaz kód, který ho vykoná, ale kvůli tomu byl vytvářen z velké části opakující se kód, který nebyl tolik testovaný a navíc bylo náročnější kód programovat a hlavně pochopit, což lze vidět ve výpisu 2.2 (pro zjednodušení se v kódu nekontroluje existence nalezeného prvku).

Kvůli opakujícímu se kódu a pomalému programování, byl do .NET Frameworku 3.5 přidán LINQ (integrovaný dotazovací jazyk). Dotaz jazyka LINQ se skládá ze tří částí. Nejdříve je získán zdroj dat, poté je nad ním vytvořen dotaz a nakonec je dotaz vykonán [19]. Zdroj dat dotazu musí implementovat rozhraní `IEnumerable<T>`, kde `T` je typ prvků

```
public void MaxEven(int[] items)
{
    if (source == null)
        throw new ArgumentNullException(nameof(source));

    int maxValue = int.MinValue;
    foreach (var item in items)
    {
        if (item % 2 != 0) continue;
        if (item > maxValue) maxValue = item;
    }
    return maxValue;
}
```

Výpis 2.2: Hledání maximálního sudého prvku z pole

kolekce. Rozhraní `IEnumerable<T>` musí obsahovat metodu `GetEnumerator` která vrací `IEnumerator<T>` a `IEnumerator<T>` obsahuje definice metod `MoveNext`, která pokud je to možné posune enumerátor na další prvek a vrátí `true` a pokud ne vrátí `false`, `Current`, která vrátí hodnotu současného prvku a `Reset`, která resetuje enumerátor.

Pomocí `IEnumerable<T>` lze vytvořit jak konečnou stejně i nekonečnou sekvenci prvků, třeba procházení pole, listu, textového řetězce, nebo třeba posloupnost přirozených čísel či generátor náhodných čísel. Vyhodnocování nekonečného zdroje dat může trvat až nekonečně dlouho. Ve výpisu 2.3 lze vidět kód iterátoru vracející posloupnost přirozených čísel. Pro vytvoření dotazu LINQ je dotaz rozdělen na části, kde každá část má za vstup i výstup `IEnumerable<T>` a lze je tedy libovolně kombinovat. Na konci dotazu pak může být vyhodnocovací operátor, který nemá za výstup `IEnumerable<T>`, ale například pole, číslo, nebo pravdivostní hodnotu. Aby mohly být dotazy prováděny nad základními kolekcemi, pro všechny systémové kolekce je implementováno rozhraní `IEnumerable<T>`.

Operátory LINQ se typicky vytvoří jako metoda, která vrací instanci objektu typu `IEnumerable<T>`, kterého enumerátor obaluje předchozí enumerátor, například při volání metody `MoveNext` se volá `MoveNext` předchozího enumerátoru a pak je výsledek zpracován. Jelikož je k obalovanému objektu zacházeno jako s `IEnumerable<T>`, můžeme operátory LINQ libovolně kombinovat. Pro různé implementace .NET se způsob obalování `IEnumerable<T>` může lišit. Vyčíslovací operátory LINQ vracejí jiný typ, než `IEnumerable<T>` a jsou implementovány obvykle jako vyhodnocení všech hodnot získaných z předchozího `IEnumerable<T>` (tedy dokud vrací `MoveNext` hodnotu `true`, jsou získávány nové hodnoty).

Pro psaní LINQ dotazů jsou v systémové knihovně implementovány tři možnosti psaní dotazů. Tyto možnosti jsou syntaxe dotazů, syntaxe metod a kombinace obou možností. Syntaxe metod implementuje dotazy pomocí volání metod, které vytvářejí iterátory. Syntaxe dotazu je více podobná na SQL a je při překladu přepisována na stejný mezikód, jako u syntaxe metod. Pro podporu operátorů v syntaxi dotazů musel být rozšířen jazyk C# (například o kontextové slova `from`, `group`, `into`, ...). Třetí možností je kombinace obou možností [21]. Jelikož je syntaxe dotazu překládána na stejný mezikód, jako u syntaxe metod, není žádný rozdíl v rychlosti při napsání ekvivalentního kódu. Příklad možného kódu napsaného imperativně, pomocí syntaxe metod a pomocí syntaxe dotazu lze vidět ve výpisu 2.4.

```
public class SimpleEnumerator : IEnumerable<int>, IEnumerator<int>
{
    private int _counter = 0;

    public IEnumerator<int> GetEnumerator() => this;
    IEnumerator IEnumerable.GetEnumerator() => this;

    public bool MoveNext()
    {
        _counter++;
        return true;
    }

    public int Current => _counter;
    object IEnumerator.Current => _counter;

    public void Reset() => _counter = 0;

    public void Dispose()
    {
    }
}
```

Výpis 2.3: Jednoduchý enumerátor sekvence zvětšujících čísel

```
public int[] ImperativeSelect(int[] source)
{
    var result = new int[source.Length];
    for (var i = 0; i < source.Length; i++)
        result[i] = source[i] * 2;
    return result;
}

public int[] MethodLinqSelect(int[] source)
{
    return source
        .Select(x => x * 2)
        .ToArray();
}

public int[] QueryLinqSelect(int[] source)
{
    return (from num in source
        select num * 2).ToArray();
}
```

Výpis 2.4: Metody zobrazující různé způsoby zapsání dotazu pomocí LINQ v jazyku C#

Oproti syntaxi metod, syntaxe dotazu zachovává vytvořené proměnné, tedy není potřeba je znovu deklarovat v lambda výrazech. Další výhodou syntaxe dotazu je možnost použití klíčového slova let pro nadefinování proměnné a jednodušší zápis některých příkazů. Syntaxe metod obsahuje vyhodnocovací operátory, které v syntaxi dotazu nejsou (např. `Count`, `Max`, `ToArray`). Také lze syntaxi metod jednoduše doplnit o nový operátor. U syntaxe dotazu by bylo potřeba pro nový operátor pozměnit překlad programu.

2.4 Výhody a nevýhody použití LINQ

LINQ nám umožňuje psát dotazy nad daty deklarativně, namísto klasického imperativního. Automaticky řeší kontrolu chyb, jako kontrola existence a velikosti vstupního pole, nebo také řeší způsob procházení daty, či způsob zvětšování výsledného pole. Proto je výsledný kód přehlednější, lépe čitelný a upravitelný (lze vidět ve výpisu 2.4). Programátor se tedy může soustředit na samotné řešení problému a ne na psaní cyklů a řešení algoritmů pro implementaci dotazu. Pomocí LINQ můžeme dělat stejný dotaz nad kterýmkoli objektem implementující `IEnumerable<T>`. Může to být pole, nebo jiná kolekce, XML, nebo SQL data. Další výhodou je pozdní vyhodnocování. Data jsou vyhodnocována až v případě potřeby (je možné, že dotaz vůbec nebude vyhodnocen, nebo že k vyhodnocení bude stačit jen vyhodnocení části kolekce). Lze ale vyhodnocení vynutit, takže nejsme donuceni čekat. Při implementaci nového operátoru lze operátor libovolně kombinovat s ostatními operátory. S použitím LINQ se snižuje množství opakujícího se kódu a tedy i velikost výsledné knihovny, či programu a času spotřebovaného na opakovanému psaní stejných algoritmů.

Hlavní nevýhodou LINQ je rychlost. Kvůli obecnému přístupu k různým druhům dat dochází k provádění zbytečných kroků a tedy se dotaz provádí déle, než při ručním napsání imperativního kódu. Dále pak predikáty, selektory, metody agregace a další metody předávané algoritmu jsou předávané pomocí delegátů, které musí být volány a tím znovu dochází k zpomalení. Kvůli pozdnímu vyhodnocování může být dotaz vyhodnocován několikrát a tím znovu vzroste čas vykonávání dotazu. Stejné kontroly vstupní kolekce jsou často prováděny mnohokrát. Při vytváření obalů iterátorů vzniká režie, způsobená voláním nově vytvořených funkcí, které jelikož se nemění, můžou být jejich těla vloženy do volající metody a tím by šlo ušetřit zbytečných volání. Jelikož se při použití vnějších parametrů při lambda výrazech vytváří pro parametry uzávěry, vzrůstá tím i počet alokací a snižuje se znovu rychlost. LINQ je v systémových knihovnách implementován jako kolekce metod o kterých mnoho programátorů neví, co reprezentují za kód, a tedy kvůli této nevědomosti můžou jednodušeji nastat chyby jako při paralelním programování čtení z Listu a zároveň jeho modifikace.

Zjednodušeně lze říct, že knihovna LINQ je skvělý nástroj pro urychlení vývoje, ale při potřebě velké rychlosti se projevují nevýhody jeho implementace. Velice často tento problém nemusí být řešen, protože při programování informačních systémů, nebo mnoha desktopových aplikací je větší rychlostní problém u zobrazování uživatelského rozhraní, nebo komunikace přes síť a tedy není potřeba dotazy nad kolekcemi řešit. Pro ty situace, kde je rychlostní problém implementace LINQ, musíme najít alternativu. Jak už jsem psal, jednou alternativou je napsání imperativního kódu namísto dotazu LINQ, jinou možností je použití jiné implementace knihovny LINQ.

Kapitola 3

Implementace LINQ

Existuje několik různých implementací dotazovacího jazyka LINQ (integrovaný dotazovací jazyk). Všechny usilují o poskytnutí možnosti deklarativního dotazování nad kolekcemi a jelikož hlavní nevýhodou základní implementací knihovny LINQ je její rychlost, některé knihovny se to snaží zlepšit. V této kapitole budu rozebírat jednotlivé implementace, jakým způsobem jsou naprogramované, jakým způsobem se snaží optimalizovat LINQ a jaké jsou výhody a nevýhody jejich použití. Budu také porovnávat jak je implementováno vykonání dotazu ve výpisu 3.1. Při uvádění obecné kombinace operátorů nebudu uvádět závorky s parametry, aby byl jednodušší a kompaktnější zápis a zároveň bylo naznačeno, že zápis platí pro obecnou kombinaci volání operátorů s libovolnými parametry, např. kombinace operátorů `Select.Where`.

3.1 System.Linq

V .NET Frameworku[1], .NET Core[2] a Mono[3] je obecně LINQ implementován stejně, jako je popsáno v kapitole 2.3 s tím, že různé implementace mají různé množství optimalizací. Implementace LINQ, které jsou v různých implementacích .NET zahrnuty v systémových knihovnách budu nazývat System.Linq. Při vykonávání porovnávaného dotazu z výpisu 3.1 je nejprve vytvořen iterátor, který získá metoda `ToArray` a ta jím začne procházet a ukládat výsledné prvky. Tam, kde nebyly provedeny další optimalizace se ke všem kolekcím chová jako k `IEnumerable<T>`. Tedy aby jím mohl procházet, musí získat enumerátor a pro každý prvek se zeptat, jestli enumerátor obsahuje další prvek a případně ho získat. Teprve poté je zavolán delegát, který vypočítá novou hodnotu prvku a ta může být uložena do výsledné kolekce. Jelikož z `IEnumerable<T>` nejde získat velikost, musí se výsledná kolekce postupně zvětšovat. To je naprogramováno vytvářením nových polí o dvojnásobné velikosti než předchozí pole a kopírování obsahu starého pole do nového. Na konci pokud je vytvořeno

[1] Referenční kód https://referencesource.microsoft.com/
[2] Referenční kód https://source.dot.net/
[3] Referenční kód https://github.com/mono/referencesource

```
var result = source
    .Select(x => x * 2)
    .ToArray();
```

Výpis 3.1: Dotaz pro porovnávání implementací LINQ

větší pole, než je výsledný počet prvků, vytvoří se nové pole o správné velikosti a prvky se překopírují.

Pro .NET Core a vyšší verze jsou provedeny různé optimalizace. Například u .NET Core je při dotazu z výpisu 3.1 vytvořen iterátor `SelectArrayIterator<TSource,TResult>`, pokud je kolekce pole (namísto obecného `SelectIterator<TSource,TResult>`). Tím je docíleno využití velikosti pole při volání operátoru `ToArray` a není tedy potřeba postupně zvětšující se pole, ale stačí vytvořit jedno pole o správné velikosti. Také může být místo procházení pomocí enumerátoru použit jednoduchý cyklus a tím je ušetřen čas. Zvětšovací algoritmus je kvůli tomu potřeba použit jen při neznámém počtu prvků, například u dotazu obsahujícím operátor `Where`.

Dále jsou optimalizovány kombinace dotazů, například často používaná kombinace operátorů `Where.Select`. Kombinace těchto operátorů je optimalizována tím, že místo vytvoření obalujícího iterátoru je vytvořen enumerátor kombinující oba operátory (obsahuje data k provedení obou operátorů). Tím může být snížen počet virtuálních volání pro funkci enumerátorů a je docíleno zrychlení provedení dotazu. Jelikož je stále vytvořen nový enumerátor, který je navíc složitější, není snížen počet alokací, ale zároveň může být předchozí enumerátor uvolněn, takže může být nižší celková paměťová náročnost provedení dotazu. Tato metoda lze využít jen pro omezený počet kombinací, protože pro každou kombinaci iterátorů je potřeba vytvořit novou třídu a tím vzrůstá velikost výsledné knihovny, nebo programu. V tabulce 3.1 lze vidět výhody a nevýhody použití neoptimalizovaného `System.Linq` (některé nevýhody použití neplatí pro některé implementace .NET).

Výhody	**Nevýhody**
V systémových knihovnách	Vytváření delegátů a uzávěrů
Jednoduché přidat operátor	Procházení všech kolekcí jako `IEnumerable<T>`
Podpora všech operátorů	Nevyužívá známou velikost výsledku
Spolehlivě funkční	Kombinací operátorů vznikají virtuální volání
Podpora paralelního zpracování	Nutnost zmenšování výsledného pole
Možnost pozdního vyhodnocení	Možnost vytvoření většího pole, než zdrojového
Možnost vyhodnocení jen potřebné části kolekce	Vytváření iterátorů

Tabulka 3.1: Výhody a nevýhody použití System.Linq

3.2 LinqFaster

LinqFaster byl vytvořen pro zrychlení jednoduchých dotazů LINQ. *Usiluje spíše o doplnění implementace System.Linq, než o její úplnou náhradu (parafrázováno z [11]).* Je implementován jako sada předpřipravených funkcí pro provedení konkrétních dotazů (např. kombinace operátorů `Select.ToArray` jako metoda `SelectF`, nebo kombinace `Select.Where.ToArray` jako `SelectWhereF`). Aby mohl kód být proveden pomocí LinqFaster musí být upraven (viz

```
var result = source.SelectF(x => x * 2);
```

Výpis 3.2: Dotaz upravený pro LinqRewrite

porovnávaný kód ve výpisu 3.1 přepsán na výpis 3.2). Metody LinqFaster operují jen nad poli a každý operátor je okamžitě vykonán a nejsou tedy vytvářeny žádné iterátory. Při neznámé velikosti výsledného pole vychází z předpokladu, že výsledné pole bude maximálně tak velké, jako vstupní pole, proto ukládá do pole stejné velikosti a nakonec všechna data překopíruje do pole výsledné velikosti.

Kvůli okamžitému provedení všech metod alokuje při řetězení metod zbytečně pole pro mezivýsledky. Lze optimalizovat vytvořením metodu kombinující předcházející dvě metody. Výhody a nevýhody použití LinqFaster lze vidět v tabulce 3.2.

Výhody	**Nevýhody**
V NuGet balíčku	Vytváření delegátů a uzávěrů
Jednoduchá implementace operátoru	Nepodporuje jiné kolekce než pole
Podporuje paralelní zpracování	Nepodporuje pozdní vyhodnocování
Podporuje SIMD zpracování (jedna instrukce nad více daty)	Zbytečné alokace pro operátor `WhereF`
Procházení pole cyklem	Kombinace metod znamená vytváření zbytečných polí
Některé kombinace operátorů optimalizované	Jiný psaní dotazů, než u `System.Linq`
Nevytváří iterátory	Kombinace dotazů nejsou přehledné
Možnost vyhodnocení jen potřebné části kolekce	Nepodporuje mnoho operátorů

Tabulka 3.2: Výhody a nevýhody použití LinqFaster

3.3 VirtualMethodLinq

Můj první návrh implementace LINQ navrhoval změnu řetězení LINQ operátorů oproti `System.Linq`. VirtualMethodLinq by oproti `System.Linq` neprocházelo zvenku dovnitř, ale zevnitř ven (tzn. prochází se smyčka pole a postupně se volají delegáty jednotlivých operátorů). To by mělo za výhodu menší počet virtuálních volání a jednoduchou implementaci procházení podle typu kolekce. Nikdy nebylo plně implementováno kvůli nevýhodám, které přináší. Výhody a nevýhody použití VirtualMethodLinq lze vidět v tabulce 3.3.

Výhody	Nevýhody
V NuGet balíčku	Nebylo implementováno
Jednoduché přidat operátor	Vytváření delegátů a uzávěrů
Možnost pozdního vyhodnocení	Nelze kombinovat s `IEnumerable<T>`
Procházení podle typu kolekce	Mnoho přístupů k `this` (instrukce navíc)
Využívá známou velikost výsledných dat	Kombinací operátorů vznikají virtuální volání
	Nutnost zmenšování výsledného pole
	Možnost vytvoření většího pole, než zdrojového
	Vytváření iterátorů

Tabulka 3.3: Výhody a nevýhody použití VirtualMethodLinq

3.4 LinqOptimizer

LinqOptimizer vytváří dotazy pomocí stromové struktury výrazů (`Expression`) [14]. Kompilování stromů výrazů je založeno na generování dynamických metod [22]. Pro vytvoření dotazu pomocí LinqOptimizer je potřeba nejdříve zavolat nad kolekcí operátor `AsQueryExpr`, který změní způsob vytváření dotazu a nad poskytnutým zdrojem začne vytvářet stromy výrazů. Místo volání operátorů LINQ jsou volány operátory LinqOptimizeru, které analyzují volané operátory a doplňují tvořený strom výrazů. Pro spuštění dotazu je potřeba zavolat funkci `Compile`, která zavolá kompilaci stromu výrazů. Kompilace je založená na dynamických metodách, tedy po analýze stromu je pomocí generátoru mezikódu (`ILGenerator`) emitován mezikód, ze kterého je zkompilována dynamická metoda a její delegát, který lze následně volat. Jelikož má kompilace velkou režii, musí se pro zrychlení běhu oddělit fáze kompilace a samotného spouštění, kdy kompilace je provedena jen jednou. Ve fázi kompilace je také provedena většina analýzy a optimalizací dotazů.

Operátory s funkcemi jako parametry jsou pro práci s výrazovými stromy upraveny, aby jejich parametry byly typu `Expression`. To umožňuje vkládat lambda výrazy do kompilovaného kódu, ale stále jsou vytvářeny uzávěry pro vnější proměnné. Z lambda výrazů obsahující blok výrazů nejde automaticky vytvářet strom výrazů a tedy musí být strom výrazů ručně napsán, což výrazně snižuje čitelnost dotazu. Dotaz z výpisu 3.1 upravený pro LinqOptimizer lze vidět ve výpisu 3.3 a dotaz přepsaný pomocí ručně napsaného stromu

```
var compiledExpression = source.AsQueryExpr().Select(x => x * 2)
                                    .ToArray().Compile();
var result = compiledExpression();
```

Výpis 3.3: Dotaz upravený pro LinqOptimizer

```
var compiledExpression = Range(0, 10).AsQueryExpr()
               .Select(Lambda<Func<int, int>>(
                   Multiply(
                       numParam,
                       Constant(2, typeof(int))), numParam))
               .ToArray().Compile();
var result = compiledExpression();
```

Výpis 3.4: Dotaz upravený pro LinqOptimizer pomocí stromu výrazů po importování statických tříd

výrazů lze vidět ve výpisu 3.4. Výhody a nevýhody použití knihovny LinqOptimizer lze vidět v tabulce 3.4.

Výhody	**Nevýhody**
V NuGet balíčku	Velká režie kompilace
Vkládá těl lambda výrazů do kompilovaného dotazu	Složitější volání jako u `System.Linq`
Procházení podle typu zdroje dat	Pro složitější dotazy náročná tvorba stromu výrazů
Využívá známou velikost výsledných dat	Kompiluje se i se zpracováním výsledku
Podporuje paralelní zpracování	Vytváření stromu výrazů (více alokací, než u iterátorů)
Nevytváří iterátory	Nutnost zmenšování výsledného pole
	Možnost vytvoření většího pole, než zdrojového
	Náročné doprogramovat další operátory
	Nepodporuje mnoho operátorů
	Nepodporuje pozdní vyhodnocování

Tabulka 3.4: Výhody a nevýhody použití knihovny LinqOptimizer

3.5 Roslyn linq rewrite

Oproti LinqOptimizer (viz sekce č. 3.4) provádí převod na procedurální kód už při překladu [8]. Převod je proveden automaticky není třeba měnit kód oproti dotazu napsanému

pro `System.Linq`. Převod je možné pro určitou třídu, nebo metodu vypnout pomocí atributu `NoLinqRewrite`. Pro přepis je třeba překládat projekt pomocí překladače roslyn-linq-rewrite, ten prochází kód a při nalezení volání metody zkontroluje, jestli má být přepsána. Pokud má být přepsána, zanalyzuje celý dotaz, parametry volání metod a vytvoří novou metodu obsahující přepsaný procedurální kód odpovídající dotazu. Samotný dotaz je pak nahrazen voláním na tuto metodu. Výhody a nevýhody použití programu roslyn-linq-rewrite jsou vidět v tabulce 3.5.

Výhody	**Nevýhody**
Vkládá těl lambda výrazů do výsledného kódu	Nelze jednoduše rozšířit o nové operátory
Stejné volání jako u `System.Linq`	Potřebuje speciální překladač
Procházení podle typu zdroje dat	Podporuje jenom projekty typu json
Využívá známou velikost výsledných dat	Pomalejší kompilace
Přepis při překladu	Složité zprovoznit
Nevytváří iterátory	Nutnost zmenšování výsledného pole
	Možnost vytvoření většího pole, než zdrojového
	Pomalý algoritmus pro neznámý počet prvků
	Opakující se kód zvětšuje velikost výsledného programu
	Funguje jen pro C#
	Nepodporuje mnoho operátorů

Tabulka 3.5: Výhody a nevýhody použití Roslyn linq rewrite

3.6 Výsledná implementace

Ze zmíněných možností implementace lze nejrychlejší kód vygenerovat pomocí principu přepisu kódu před či při překladu aplikace, který byl implementován v Roslyn linq rewrite. Běh výsledného programu nezdržuje analýza informací ani kompilace dotazu. Umožňuje provádět dotazy až při vyžádání a nealokuje zbytečná pole ani iterátory. Kvůli jeho nevýhodám ho ale často není vhodné, nebo možné použít. Proto by výsledná implementace měla co nejvíce nevýhod (zmíněných v tabulce 3.5) eliminovat, aby byla knihovna co nejpoužitelnější. Dále by bylo dobré implementovat optimalizace z kapitoly č. 4.

Kapitola 4

Možné optimalizace LINQ

V kapitole č. 3 jsou popsány různé způsoby implementace LINQ. Mnoho z nich usilovalo o optimalizaci LINQ. Proto popíšu použité a některé další možné metody optimalizace. Optimalizace ve většině případů nejsou výhodné ve všech směrech. Většinou je optimalizace jen přesunutím, či změnou času provádění operace na jiné místo, nebo použití jiného způsobu zpracování. Tedy u většiny optimalizací je v algoritmu někde výhoda a někde nevýhoda. Příkladem může být počítání logaritmu o základu 2 pomocí konverze čísla na desetinné a vrácení jeho exponentu, což oproti funkci `Math.Log` disponuje velkou rychlostí, ale nelze měnit bázi logaritmu a metoda funguje jen pro neznaménková celá čísla. Tedy při zhoršení definičního oboru lze zvýšit rychlost provádění funkce. Lze zvýšit rychlost výrazu $(1 + 1) * x$ na $x + x$, ale musí být provedena analýza výrazu a pokud by byla provedena při běhu programu, z velkou pravděpodobností by samotná analýza výrazu byla pomalejší, než výpočet výrazu samotného. Pokud je ale analýza prováděna při kompilaci může být dosaženo zrychlení, sníží se ale rychlost kompilace a tedy pravděpodobně i rychlost vývoje. Možností je výrazy upravovat jen při vydávání verze aplikace, ale poté je rozdíl mezi verzí, kterou programátor vyvíjel a verzí, která je vydávaná a můžou být ve výsledné aplikaci chyby (například při paralelním programování).

Jelikož má výsledná implementace přepisovat dotazy LINQ, bude velká část optimalizací zaměřená na převod času z běhu programu na analýzu při přepisu. U některých optimalizací ale nelze zjistit všechny informace při analýze a je potřeba vybrat algoritmus pro obecná data, ale různé algoritmy jsou různě dobré nad různými daty. Jelikož bylo potřeba, abych některé z nich vybral jako základní způsob provádění dané operace, vybíral jsem z předpokladů zmíněných u výběru a pokud by statisticky bylo dokázáno, že byly mé předpoklady mylné, bylo by potřeba výslednou implementaci podle toho upravit. Existuje mnohem více možných optimalizací, které by mohly zrychlit implementaci LINQ, ale v této kapitole jsou napsány takové, které byly zvažovány, že budou implementovány ve výsledné implementaci, nebo implementovány byly.

Procházení kolekcemi

Zdroj LINQ dotazu je vždy zdrojová kolekce implementující rozhraní `IEnumerable<T>`. Ale i když je dotaz vykonáván nad polem, pokud je s kolekcí zacházeno jako s `IEnumerable<T>` nelze jednoduše zjistit počet prvků. Proto při každé iteraci kolekcí je potřeba zkontrolovat, jestli obsahuje další prvek a pak ten prvek získat. Navíc pokud je ke kolekci přistupováno

jako k `IEnumerable<T>`, musí být volány virtuální metody `MoveNext` a `Current` (volání metod rozhraní jsou virtuální). Kvůli tomu dochází k zpomalení a proto je potřeba zvolit typ procházení podle typu kolekce.

U některých implementací System.Linq se provádí kontrola typu kolekce za běhu programu na začátku před provádění iterace. Tím se docílí správného procházení kolekce i v situacích, kdy je v parametru předán obecnější typ kolekce, než je opravdu procházen, ale není možné tuto optimalizaci použít v případě přepisování dotazu, protože by bylo nutné udělat několik verzí přepsaného kódu podle typu kolekce a správnou verzi zvolit až za běhu programu podmínkou. Více verzí přepsaného kódu by mělo ale za následek velkých duplicit kódu, pomalejší přepis a hlavně velkou velikostí výsledné knihovny (u `System.Linq` kvůli vytváření iterátorů tyto nevýhody nejsou). Z těchto důvodů jsem implementoval volbu typu procházení za přepisu dotazů, tedy kolekce je procházena podle typu kolekce určené pomocí sémantické analýzy.

Vkládání lambda výrazů

Jelikož se selekce vykonává pro všechny prvky, proto se delegát lambda výrazu předaného parametrem operátoru `Select` volá pro každý prvek kolekce. A jelikož volání delegátu je operace s cenou několikanásobně dražší než je například sčítání, pro jednoduché dotazy to znamená velké zpomalení. Proto je důležité vkládat těla lamba funkcí do výsledného kódu namísto jejich volání (platí například pro operátory `Select`, `Where`, `Aggregate`, `All`, ...).

Pokud je navíc v lambda výraz předané operátoru `Select` vnější parametr, který není definován v tom lambda výrazu, ale mimo něj, vytváří se pro ten parametr uzávěr, tedy když se volá delegát lambda výrazu a chce přistoupit k danému parametru, musí přistoupit k instanci uzávěru a až pak k parametru samotnému, což vytváří další alokace a zpomaluje provádění dotazu. To lze optimalizovat tím, že se vloží tělo lambda výrazu do výsledného kódu a daný parametr předán jako parametr funkce (při měnícím se parametru musí být předán pomocí reference). Oproti uzávěru může nastat chyba, když proměnná přestane existovat, nebo při pozdním vyhodnocení dotazu, když se parametr mění, takže se tento stav musí ošetřit. Parametr předávaný referencí nelze použít při použití příkazu `yield return` v metodě a tedy je nutno použít delegát a uzávěr parametru. Tedy i z tohoto důvodu je důležité vkládat těla lambda výraz do generovaného kódu.

Operace nad vektory

U LinqFaster jsou použity pro optimalizaci SIMD operace (jedna instrukce nad více daty). Pomocí SIMD lze například rychle vypočítat součet pole. Pole je rozděleno na vektory, které jsou sčítány vždy jedna instrukce sečte 2 n-tice čísel a nakonec jsou sečteny jednotlivé hodnoty výsledného vektoru. SIMD je možné použít jen pro číselné operace a funguje jen na některých architekturách, takže před provedením je potřeba vyzkoušet, jestli je architektura podporovaná. V případě vhodného použití může několikanásobně urychlit provedení dotazu. Ve výsledné implementaci není naprogramováno, kvůli jinému přístupu k datům (náročné implementovat), ale je možné použít přepsání dotazu LINQ pro provedení SIMD operací (viz výpis 4.1).

```
[Unchecked]
public int ArraySIMDSum(int[] source)
{
  var simdLength = Vector<int>.Count;
  var vectorSum = Vector<int>.Zero;
  vectorSum = ExtendedLinq.Range(0, source.Length / simdLength, simdLength)
    .Aggregate(vectorSum,
      (x, y) => Vector.Add(x, new Vector<int>(source, y)));

  return Enumerable.Range(0, simdLength).Sum(i => vectorSum[i])
      + source.Skip(source.Length / simdLength * simdLength).Sum();
}
```

Výpis 4.1: Vypočítání sumy pomocí vektorového součtu

Známý počet výsledných prvků

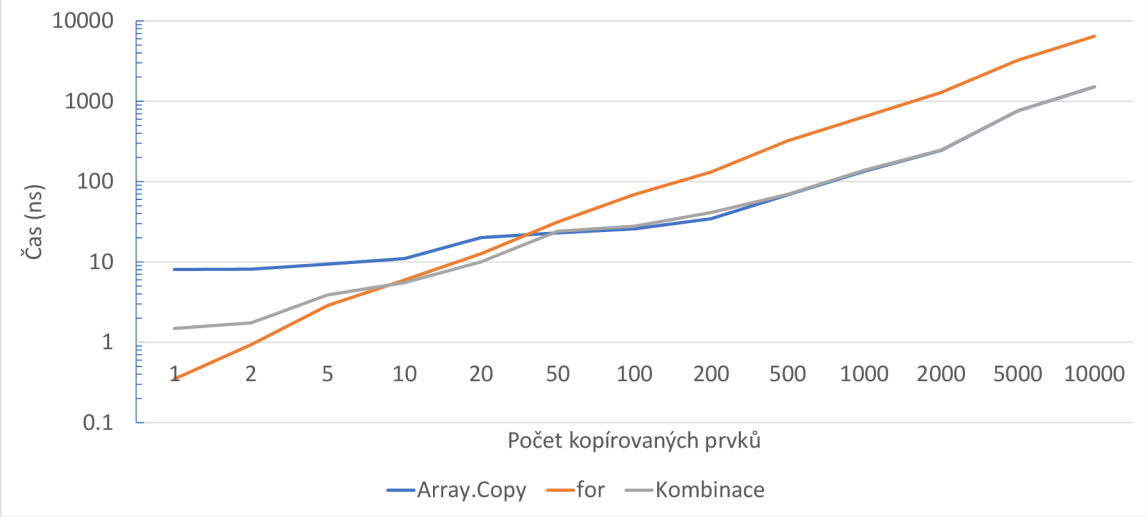
Pokud je znám počet prvků vstupní kolekce dotazu a dotaz sám nemění počet prvků (např. kombinace operátorů `Select.ToArray`, nebo `Skip.ToArray` nad polem), je výhodné tuto informaci využít. Můžeme vytvořit výslednou kolekci o velikosti zdrojové a postupně ji plnit daty, namísto použití algoritmů pro postupné zvětšování kolekce. Tím je ušetřen čas vytváření polí, kopírování prvků a snížen počet alokací. Pokud vyhodnocovací operátor nevrací kolekci prvků a zároveň k výpočtu výsledku není potřeba vytváření kolekcí, neměl by je kvůli rychlosti a alokacím vytvářet (není dodrženo např. u LinqFaster pro mezivýsledky operátorů).

Způsob kopírování dat

V C# jsou dva základní způsoby kopírování pole. První je použití `Array.Copy` a druhá je procházení cyklem a kopírování po jednotlivých prvcích. Existují další, méně standardní způsoby jako funkce `Buffer.Copy`, vektorové kopírování, kopírování pomocí ukazatelů, nebo způsoby měnící procházení kolekce, jako kopírování po skupinách prvků. Provedl jsem testování závislosti délky trvání na počtu kopírovaných prvků pomocí `Array.Copy` a kopírování cyklem. Výsledky benchmarků jsou zaznamenané v obrázku 4.1. Z důvodu velké rozdílnosti výsledků je časová osa zobrazená logaritmicky. Z výsledků lze vidět, že pro malé množství kopírovaných dat (asi do 30 prvků) je rychlejší kopírování pomocí for cyklu a poté začíná být výhodnější použít `Array.Copy`. Vytvořil jsem proto třetí metodu, kombinující předchozí dvě metody (přidána do obrázku 4.1), která podle počtu prvků volí mezi kopírováním cyklem a `Array.Copy`. Kvůli volbě algoritmu za běhu je přibližně 0.5-1 ns režie. Pro malé množství dat není tak dobrá jako kopírování cyklem a pro velké množství jako `Array.Copy`, průměrně má ale nejlepší výkon a proto byla vybrána jako základní metoda kopírování pro výslednou implementaci. Při znalosti počtu kopírovaných prvků je ideální zvolit algoritmus kopírování explicitně mezi cyklem a `Array.Copy`.

Počet prvků	Vytváření (ns)	Kopírování (ns)
0	2.439	4.884
2	2.660	7.170
4	2.962	7.604
8	3.516	6.984
16	4.562	9.389
32	6.205	11.967

Tabulka 4.1: Délka trvání vytvoření a kopírování pole o velikosti n



Obrázek 4.1: Délka kopírování různých algoritmů v závislosti na počtu prvků

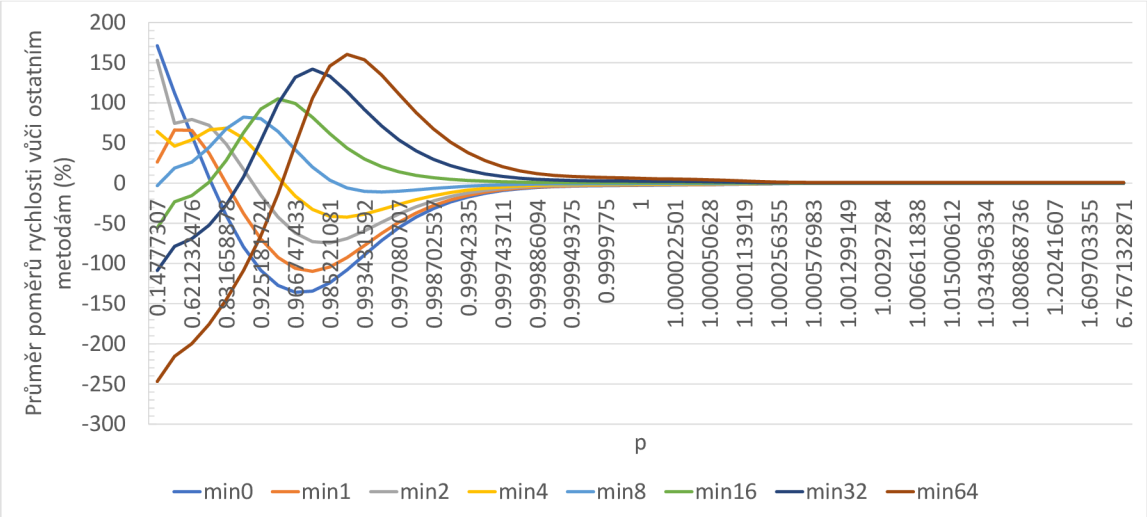
Ořezání výsledného pole

Pokud je použit algoritmus postupného zvětšování pole, je velká pravděpodobnost, že bude potřeba výsledné pole zmenšit, protože výsledné pole je jen částečně plné a to není validní stav. V `System.Linq` se to řeší vytvořením nového pole o správné velikosti a překopírováním prvků. Obě tyto operace jsou relativně drahé. Proto je často rychlostně lepším řešením dát výsledné pole v .NET Core do `Span<T>`, nebo do obdobné kolekce (např. `SimpleList<T>`), protože tím se jen specifikuje délka kolekce, ale délka pole zůstává stejná. Tato možnost redukuje počet alokací polí a zvyšuje rychlost, ale zvětšuje paměťovou náročnost a vytváří instanci generické třídy, proto by mělo být volitelné, jestli tuto optimalizaci použít.

Počáteční počet prvků

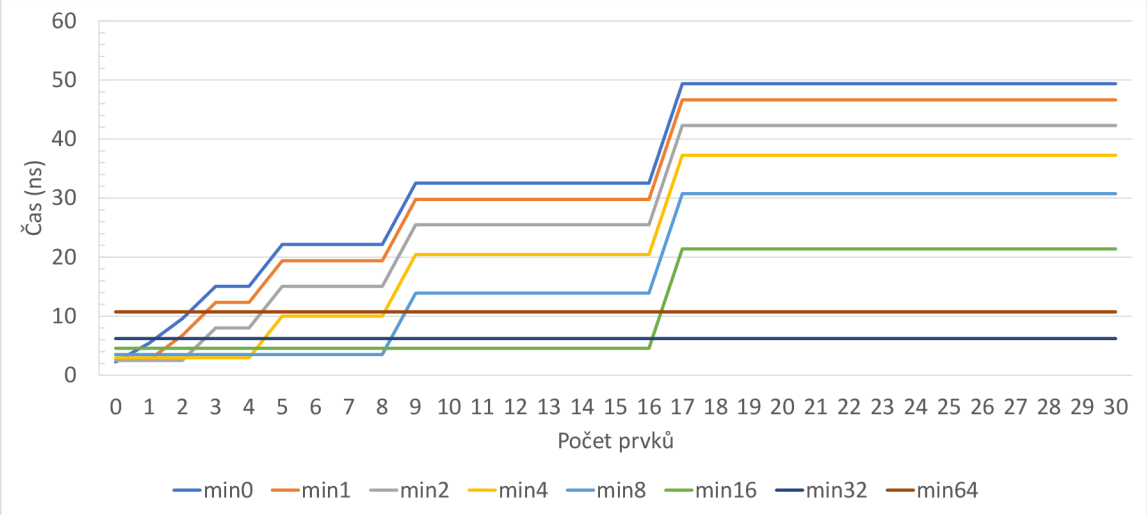
Pro neznámý počet výsledných prvků je pro větší počet prvků rychlostně výhodné nastavit počáteční délku pole (za předpokladu zajištění ořezání výsledného pole). Při postupném zvětšování výsledného pole jsou alokace nového pole a kopírování prvků z minulého pole relativně drahé operace. Pomocí BenchmarkDotNet jsem změřil cenu těchto operací pro některé velikosti polí a zaznamenal je do tabulky 4.1.

Při postupném zvětšování pole, můžeme délku trvání naplnění pole o n prvcích definovat jako součet všech časů (z tabulky 4.1) vytváření a kopírování pole pro počet prvků menší, než n a vytváření pole o velikosti aspoň n. Při průměrování časů pro n od 0 do N by se při velkém N ztratila důležitost optimalizace časů pro malý počet prvků, proto budu počítat průměr počtu procent o kolik je jedna metoda rychlejší než druhá. Budu počítat pro počáteční počet prvků 0, 1, 2, 4, 8, 16, 32 a 64 pro výsledné pole o velikosti 0 až 1000 prvků. Pro simulaci různých rozdělení pravděpodobnosti budou hodnotám přidány váhy o velikostech, které lze spočítat pomocí exponenciální rovnice $|1-p|^{sgn(1-p)x}; p \in (-1, 1)$, kde x je index prvku v poli a p je parametr. Na obrázku 4.2 jde vidět graf závislosti podílu rychlosti vůči ostatním metodám na parametru p.



Obrázek 4.2: Rychlost zvětšování pole pro různé počáteční počty prvků dat v závislosti na rozložení pravděpodobnosti výsledného počtu prvků (p->0 pravděpodobně malý počet výsledného počtu prvků, p->∞ pravděpodobně velký počet)

Z grafu na obrázku 4.2 lze vidět, že počáteční velikost pole ovlivňuje rychlost. Jelikož pro rovnoměrné rozložení dat a dál ($p >= 1$) je relativní rozdíl malý, zobrazím tedy absolutní dobu trvání pro prvních 30 prvků pro jednotlivé metody (viz obrázek 4.3).

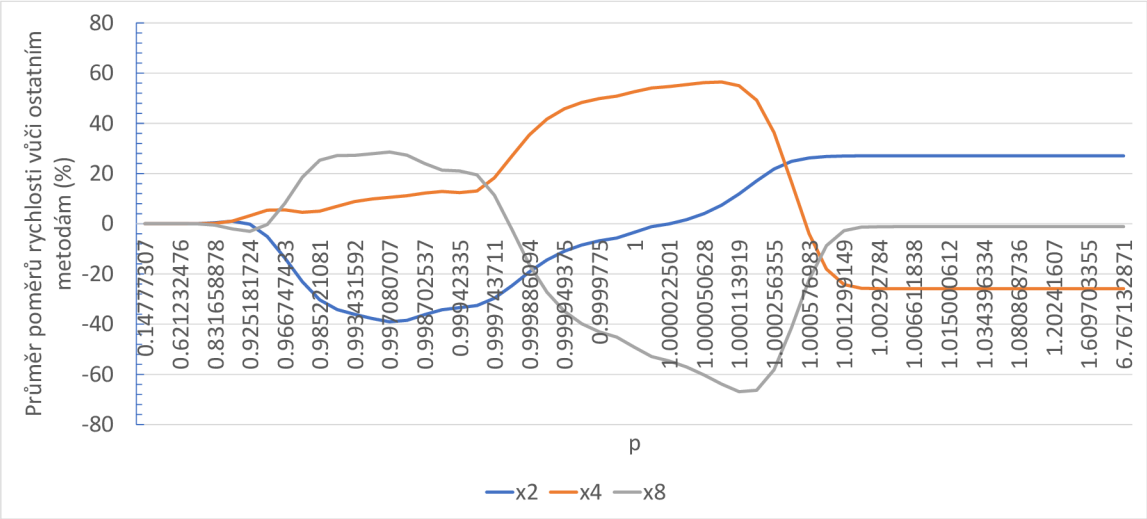


Obrázek 4.3: Čas zvětšování pole pro různé počty počátečních prvků v závislosti na výsledném počtu prvků

Z obou grafů na obrázcích 4.2 a 4.3 lze vidět, že pro menší počet prvků je rychlostně výhodnější použít menší počáteční počet prvků. Proto nastává otázka s nejednoznačnou odpovědí která metoda je nejvhodnější pro implementaci. Jelikož pro větší množství výsledných dat relativní rozdíl rychlostí není velký, oproti malým množstvím dat je pomalejší maximálně o 1.5ns a z grafu na obrázku 4.3 lze vidět, že pro většinu situací má nadprůměrný výsledek, budu pro výslednou implementaci používat jako základní počáteční počet prvků 8. Při neořezávání výsledného pole tím vzroste paměťová náročnost pro malý počet výsledných prvků, ale primárním úkolem bylo rychlostní optimalizace LINQ (při použití vyhodnocovacího operátoru `ToArray` je přebytečná paměť uvolněna).

Možnosti zvětšování počtu prvků

Pro postupné zvětšování pole musíme určit počáteční počet prvků (viz kapitola č. 4) a poté algoritmus pro zvětšování pole. Základní algoritmus pro zvětšování pole je při dosažení kapacity pole, n krát pole zvětšit a původní pole do nového překopírovat. Provedl jsem porovnání pro n rovno 2, 4 a 8 (Počítáno pomocí stejného algoritmu jako v kapitole č. 4) a výsledky zaznamenal do grafu na obrázku 4.4.

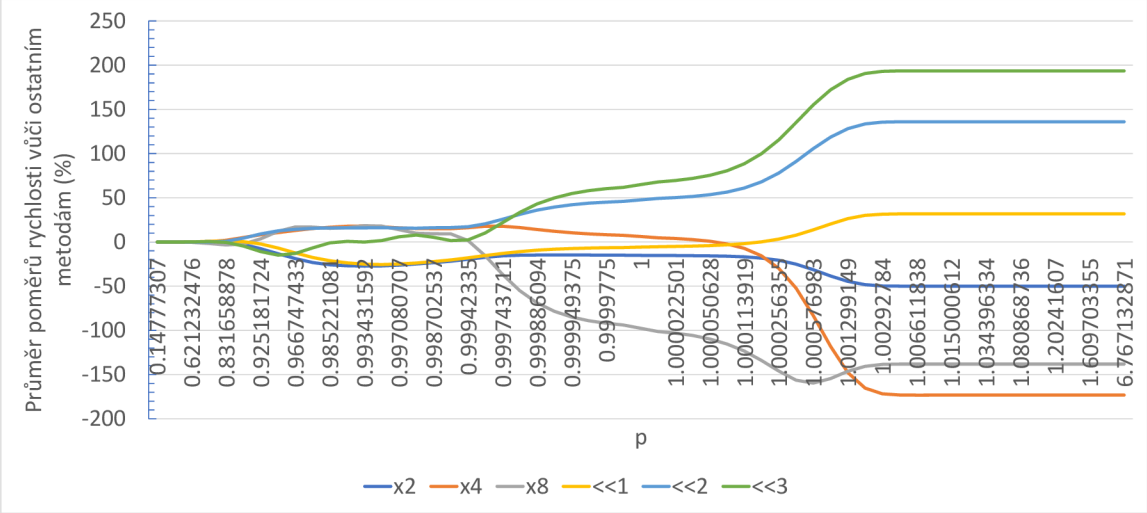


Obrázek 4.4: Rychlost zvětšování pole pro různé zvětšovací koeficienty v závislosti na rozložení pravděpodobnosti výsledného počtu prvků (p->0 pravděpodobně malý počet výsledného počtu prvků, p->∞ pravděpodobně velký počet)

Z grafu na obrázku 4.4 lze vidět, že pro větší počet prvků má nejlepší výsledky n=2. To je způsobeno tím, že i když vstupní pole mělo 10000 prvků při zvětšování x2 má největší vytvořené pole 16384 prvků, ale pro x4 a x8 32768 prvků. Pro zvětšování x8 může mít výsledné pole až x8 více prvků, než vstupní pole (Což způsobuje větší paměťovou náročnost). Metoda násobení větší konstantou než 2 je výhodnější, jen pokud je pro zvětšování n* výsledný počet prvků n* menší, než vstupní počet prvků. Abychom mohli použít větší zvětšovací koeficient, musíme zajistit podmínky, aby nedocházelo ke zbytečným alokacím. Pro menší množství výsledných dat má nejlepší výsledky zvětšování x4. Kvůli menšímu počtu alokací a větší stabilitě algoritmu jsem jako základní algoritmus pro zvětšování pole tedy vybral zvětšování pomocí zdvojnásobení.

Známý počet vstupních prvků

Pokud je znám počet prvků před provedením dotazu a dotaz nemění, nebo zmenšuje počet prvků (např. kombinace operátorů `Where.ToArray` nad polem), můžeme tuto informaci využít tak, že nemůže být maximální počet prvků větší, než velikost zdrojové kolekce. Například pomocí algoritmu postupně se zmenšujícího bitového posunu. Tedy pro n-té zvětšení pole se nebude počítat nová velikost pole jako 2^n, ale jako $source.Count << (log2(source.Count) - n)$ Pro maximální počet prvků 625, pak posloupnost velikostí vytvořených polí je 9, 19, 39, 78, 156, 31, 625, namísto 4, 8, 16, 32, 64, 128, 256, 512, 1024. Při použití optimální metody pro počítání `log2` pro číslo typu `int` na začátku provádění dotazu, zpomalí se algoritmus asi o 0,3ns (docílí se ale redukce paměťové náročnosti a zrychlení metod zvětšování pole pro velké výsledné pole). Provedl jsem porovnání pro zvětšování pomocí násobení koeficientem n rovno 2, 4 a 8 a pomocí bitového posunu o 1, 2 a 3 (Počítáno pomocí stejného algoritmu jako v kapitole č. 4) a výsledky zaznamenal do grafu na obrázku 4.5.



Obrázek 4.5: Rychlost zvětšování pole pro různé zvětšovací algoritmy v závislosti na rozložení pravděpodobnosti výsledného počtu prvků (p->0 pravděpodobně malý počet výsledného počtu prvků, p->∞ pravděpodobně velký počet)

Z grafu na obrázku 4.5 lze vidět, že oproti samotné změně zvětšovacího koeficientu má jiný typ zvětšovacího algoritmu o moc lepší výsledky pro větší množství výsledných dat. Jelikož v algoritmu dochází k méně alokacím, a pro bitový posun o 2 nejsou z rychlostního hlediska žádné očividné nevýhody, bude tento algoritmus zvolen jako výchozí algoritmus pro zvětšování pole, pokud to bude možné. Pokud to nebude možné, bude použit algoritmus zdvojnásobení velikosti pole, protože průměrně má lepší výsledky, než násobení větším koeficientem.

Paralelní provedení dotazu

V několika implementacích je možnost paralelního provedení dotazu. Tím se umožní dotaz zpracovávat více vlákny a může se tím mnohonásobně zvýšit rychlost provedení. Při špatném způsobu práce s vlákny může i zpomalit provedení a také kvůli režii správy vláken při nadměrném používání paralelizace může dojít k zahlcení procesoru. Paralelní provedení nesmí ovlivnit správnost výsledku proto jsou používány při přístupu k datům zámky a při jejich nesprávném použití může dojít k uváznutí. Navíc při použití zámků může nastat zastavení sdílených vláken a následně zpomalení celého vícevláknového programu. Kvůli náročnosti zajištění správnosti výsledku tato optimalizace nebyla ve výsledné implementaci naprogramována.

Optimalizace matematických výrazů

Jelikož různé operace v C# jsou různě rychlé, lze docílit optimalizace úpravou některých matematických výrazů. Například jelikož operátor sčítání je rychlejší, než operátor násobení, můžeme docílit zrychlení výrazu $2x$, přepsáním za výraz $x+x$, ale výraz $x+x+x+x$ je rychlejší přepsat jako $4x$. Optimalizace matematických výrazů je ale náročný úkol, protože sice lze k optimalizaci použít například knihovnu `MathNet.Symbolics`, ale ta počítá matematicky, a ne programátorsky. Tím ale můžou vzniknout pro celá čísla chyby zaokrouhlení

(např. $x/2 + y/2$ je nahrazeno za $(x + y)/2$ pro celá čísla x = 3, y = 3 jsou rozdílné výsledky) a nejsou podporovány operátory jako %, $\ll$, $\gg$, |, &, ~. `MathNet.Symbolics` dále neumí pracovat s voláním metod a někdy může způsobit zpomalení, namísto urychlení, když místo výpočetně optimálního výrazu použije výraz nejkratší (například při nahrazení $x + x$ za $2x$). Při zajištění ochrany proti chybným záměnám je ale výhodou počítání konstant a optimalizace některých výrazů.

Znovupoužitelné hodnoty výrazů

Při provádění LINQ dotazu je většinou potřeba procházet kolekci cyklem. Pokud ale při každé iteraci je potřeba vypočítat stejnou hodnotu, snižuje to rychlost výpočtu a je rychlostně výhodné ji uložit do proměnné. Stejně pokud je proměnná i v jedné iteraci vypočítaná vícekrát, je často rychlostně výhodné ji uložit do proměnné. Příkladem může být počet prvků listu při jeho procházení for cyklem, nebo vícenásobné použití proměnné při dotazu obsahujícím operátor `While`, kdy je potřeba hodnota pro vyhodnocení operátor `While` i pro vyhodnocení operátoru následujícího. Zároveň je rychlostně i paměťově výhodné nevytvářet zbytečné proměnné, tedy uložení hodnoty do proměnné, kdy bude hodnota využita jen v následujícím kroku není optimální.

Použití různých algoritmů pro různá data

Nad různými daty jsou optimální různé algoritmy. Pro `ArraySource.Skip(20).ToArray()` můžeme použít kopírování pomocí obecného algoritmu kopírování pro výslednou implementaci, pro `IEnumerable<T>` to ale udělat nelze. Pokud by ale zdrojové pole, bylo nějak upraveno, například pomocí operátoru `Select`, musíme kopírování přepsat jako cyklus. Můžeme ale využít, že procházíme polem a začít procházení až od 20. prvku. Kdyby byl dotaz upraven pomocí operátoru `Where`, potom bychom museli přepsat operátor `Skip` jako podmínku v cyklu, která docílí přeskočení prvních 20 prvků. Stejně tak i ostatní operátory lze přepsat na různě optimální metody podle toho, nad jakými daty jsou prováděny. Při volbě různých algoritmů podle situace použití narůstá složitost analýzy přepisu a tedy je analýza náchylnější ke vzniku chyb.

Jednoduchý přepis dotazu

Při znalosti informací nemusíme některé dotazy přepisovat na volání metody, ale můžeme je zjednodušit na jednoduchý výraz. Například pro dotaz `ArraySource.Count()`, lze přepsat jako `ArraySource.Length`. Nevýhodou je, že pokud výsledek není zpracován, nastane chyba při kompilace, proto by byl přesnější přepis `_ = ArraySource.Length`. Lze také jednoduše přepsat `ArraySource.Select(x => x + n).Last()`, ale musí se ale zkontrolovat, jestli selekční metoda je lambda výraz a zároveň nemění žádnou proměnnou, kterou sama nevytváří, protože když má metoda vedlejší efekt, může dojít k změně funkčnosti. Při splnění podmínek lze přepsat dotaz na `ArraySource[ArraySource.Length - 1] + n`.

Správa kontrol

Při provedení dotazu LINQ se vykonává mnoho kontrol, které zpomalují běh algoritmu. První možnost optimalizace je, že se některé kontroly můžou provádět už při kompilaci,

```
var p = 3;
var result = Select(x => x + p).ToArray();
```

Výpis 4.2: Jednoduchý dotaz s operátory Select a ToArray

například, že u dotazu `ArraySource.Take(10).ToArray()`, je parametr operátoru `Take` větší než 0 a tedy má dotaz smysl. Ale některé kontroly nejdou určit při kompilaci například u `ArraySource.Skip(3).Take(1).ToArray()` se musí pro operátor `Skip` kontrolovat, jestli má vstupní pole více než 0 prvků, jinak by mohla nastat výjimka. Nejen že se musí provést samotná kontrola, ale zároveň kvůli kontrole nelze provést zjednodušení dotazu na `new int[] {ArraySource[3]}`, protože pokud velikost `ArraySource` je menší, než 3, nastane chyba. Proto je možné přidat optimalizační možnost vypnutí kontrol, která by umožňovala takový přepis s tím, že potřebné kontroly musí zajistit programátor.

Menší počet alokací

Při použití `System.Linq` vznikají alokace, kterým lze předejít. Při provedení jednoduchého dotazu uvedeného ve výpisu 4.2 jsou při použití `System.Linq` alokován enumerátor pro operátor `Select`, delegát pro operátor `Select`, uzávěr proměnné `p`, výsledné pole a v některých implementacích ještě `Buffer<int>`. Přitom pro správné vykonání příkazu je potřebné alokovat jen výsledné pole.

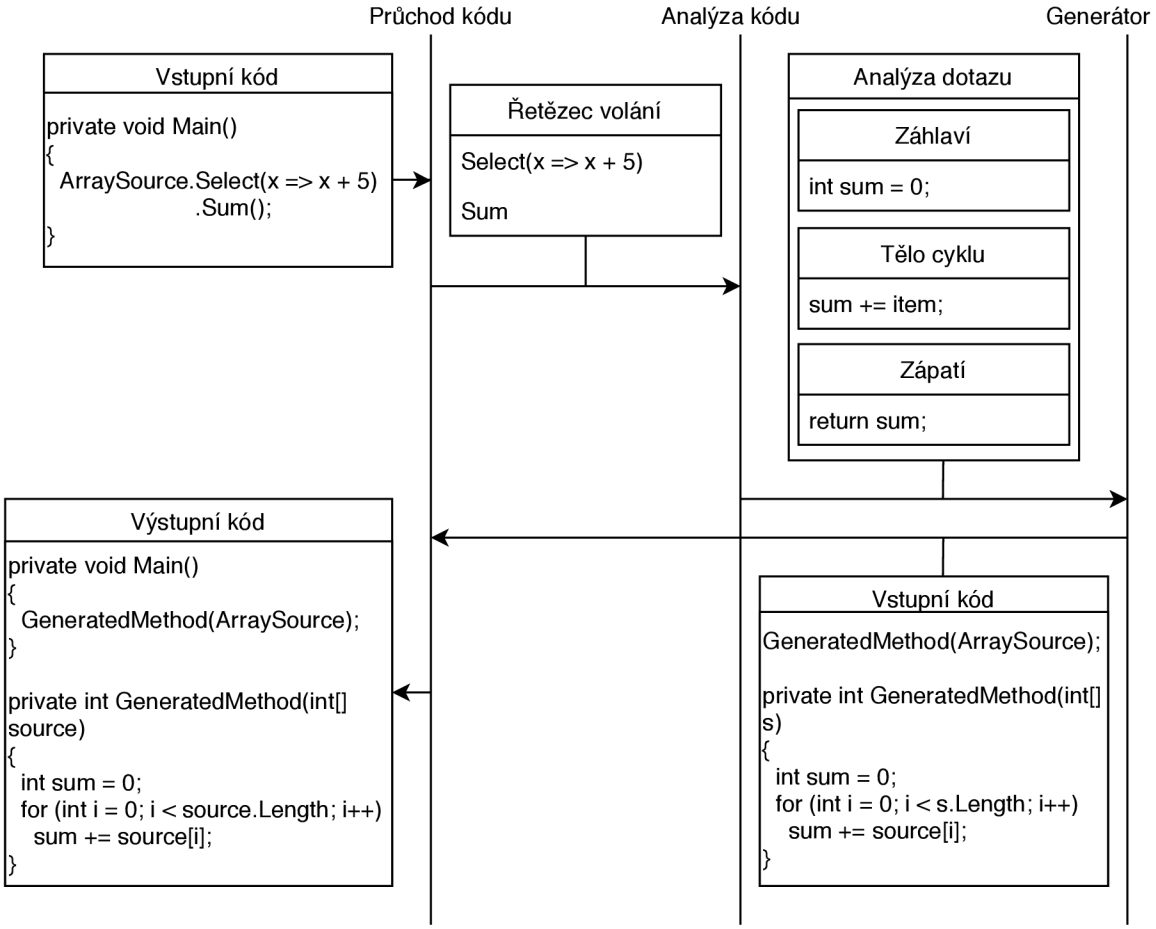
Kapitola 5

Implementace LinqRewrite

Kvůli svým výhodám jsem pro výslednou implementaci vybral pro optimalizaci LINQ dotazů přepis dotazů do procedurálního kódu při překladu (viz závěr z kapitloly 3). Implementace tedy vychází z Roslyn linq rewrite (viz kapitola č. 3.5), ale snaží se opravit některé jeho nedostatky a doplnit program o další optimalizace popsané v kapitole 4. Implementaci jsem nazval LinqRewrite, aby z názvu bylo zřejmé, jaký je hlavní záměr programu.

Přepis dotazů pomocí Roslyn linq rewrite má tři části, které pojmenuji jako průchod kódu, analýza a generátor. V průchodu kódu je jako vstup syntaktický strom získaný pomocí `SyntaxFactory` od Roslyn, který je postupně procházen, kontrolován, zda daný výraz má být přepsán a v případě, že přepsán být má, volá analýzu a nahrazuje kód za nově generovaný. V druhé části programu je získán řetězec volaných metod dotazu a probíhá jeho analýza a volání generátoru. Při samotné analýze jsou přidány podle operátorů vyhodnocující dotaz (např. operátor `Count`) nebo modifikující iteraci (jako `Select`, `Where`, atd.) příkazy do kolekcí příkazů záhlaví, tělo a zápatí cyklu a je vytvořen předpis generování nové metody. Při samotném generování je tento předpis vstupem a je vytvořena (a přidána do třídy odkud je voláno generování) privátní metoda se záhlavím, cyklem obsahujícím tělo cyklu, následované zápatím a volání na vygenerovanou metodu je předáno jako zaměněný kód. Konkrétní příklad tohoto procesu lze vidět na obrázku 5.1.

Samotný proces přepisu, který je využíván v Roslyn linq rewrite lze využít pro vytvoření programu LinqRewrite, ale implementace jednotlivých částí mají značná omezení. Například analýza v Roslyn linq rewrite neumožňuje vytvořit předpis pro generování dotazu, který potřebuje procházet více cykly (například u operátorů `Concat`, `Union`, `Except`, atd.) a tedy ani generátor není připraven pro jejich zpracování. Jelikož se při procesu počítá s vytvořením právě jednoho cyklu, kdy je bráno v potaz jen několik modifikujících operátorů, je analýza dotazu vytvářena v jedné metodě řešící celé vytváření dotazu pomocí stromu podmínek (což nesplňuje principy SOLID a při zpracování libovolného možného LINQ dotazu by byl strom nekonečně dlouhý). Dále generátor neumožňuje přepis na jednoduchý výraz. Roslyn linq rewrite je dále použita malá úroveň abstrakce, a tedy obsahuje velké množství podobného kódu a je náročné implementovat další operátory a optimalizace.



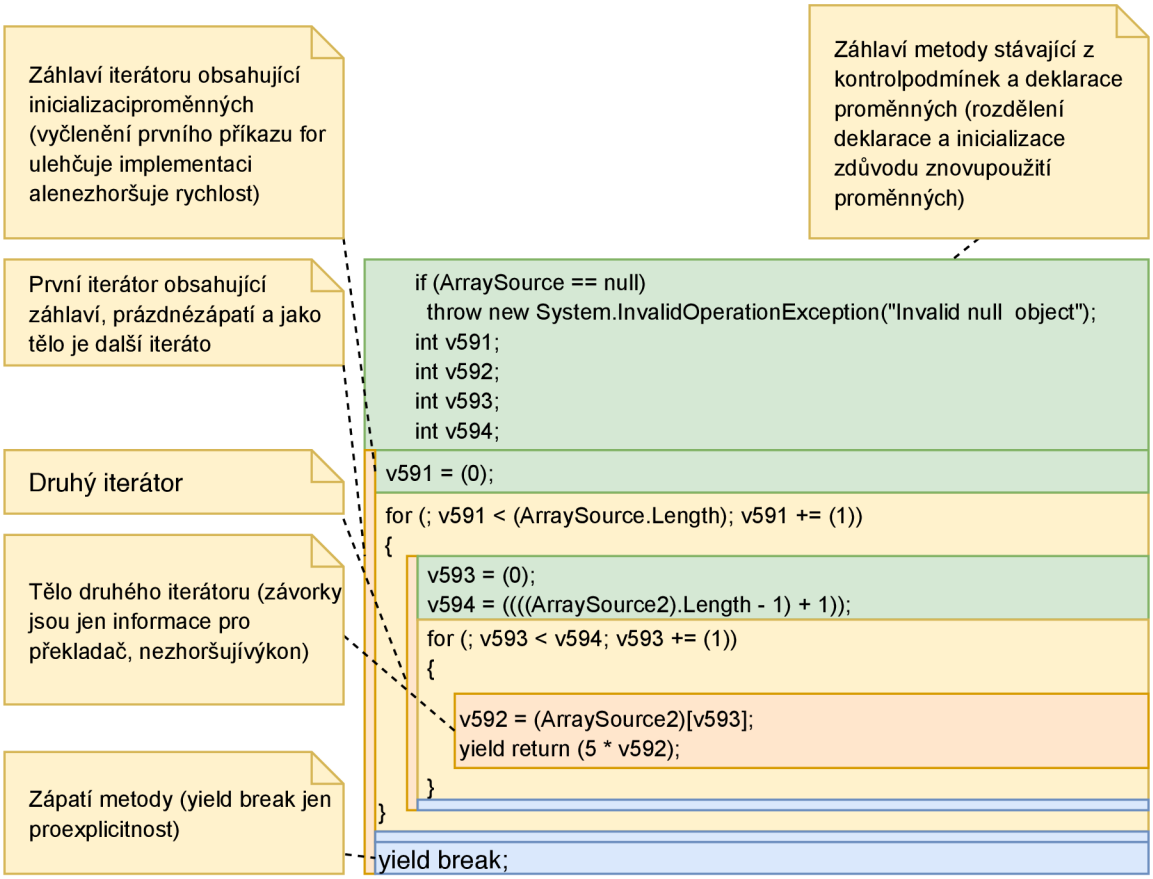
Obrázek 5.1: Průběh přepisu dotazu pomocí Roslyn linq rewrite

5.1 Úprava roslyn linq rewrite

Kvůli změně základních procesů bylo nutné naprostou většinu kódu z původního Roslyn linq rewrite přepsat. Zachoval jsem ale velkou část procházení kódu, část generování kódu a některé datové struktury. Pro začátek jsem musel rozdělit kód do tříd, aby byl zachován princip jedné zodpovědnosti. Jelikož různé implementace .NET nejsou plně kompatibilní a tedy dotaz LINQ může mít nad stejnými daty v různých implementacích .NET různé výsledky a LinqRewrite byl implementován tak, aby byl plně kompatibilní se `System.Linq` v .NET Framework, není tedy plně kompatibilní s ostatními implementacemi `System.Linq`.

Poté jsem upravil analýzu, aby analýza měla záhlaví, 0-n iterátorů, a zápatí a dále několik vlastností, jako velikost vstupní kolekce a typ vstupní kolekce, velikost výstupní kolekce, jména a využitost vytvořených proměnných a další. Každý iterátor obsahuje jeho záhlaví, tělo cyklu a zápatí a v těle cyklu iterátoru může být jako příkaz další iterátor. Tedy pomocí této struktury lze tvořit i vnořené cykly, nebo mít více cyklů pro více kolekcí. Upravil jsem analýzu, aby předpis generování tvořil postupně, tedy při analyzování kombinace operátorů `Where.ToArray` nad polem nejdřív vytvořil iterátor k procházení pole, poté do iterátoru vložil kontrolu predikátu a nakonec pro `ToArray` doplnil kód o vytvoření základního pole, postupné plnění a nakonec kontroly velikosti a vrácení validního pole. Dále jsem přidal kontrolu a možnost přepsání do jednoduchého výrazu. Pro přehlednost jsem udělal pro každý

operátor třídu, která je zodpovědná za analýzu jednoho operátoru, včetně analýzy informací a použitelnosti algoritmů. Upravený předpis generování metody lze vidět na obrázku 5.2.



Obrázek 5.2: Předpis generování metody v LinqRewrite

Musel jsem také upravit generátor, aby reflektoval změny udělané v analýze. Doprogramoval jsem chybějící operátory, kromě `DefaultIfEmpty` (odlišný přístup k datům), `ToLookup` (použití interní třídy) a řadících operátorů, (kvůli nekompatibilitě algoritmu s implementací), které kvůli náročnosti nebyly implementovány. Z optimalizací z kapitoly č. 4 byly implementovány všechny zmíněné optimalizace, kromě operací nad vektory, paralelního provedení dotazu, a procházení podle typu kolekce určenému za běhu programu. Dále jsem opravil mnoho chyb a zavedl podporu `.csproj` souborů. Kvůli náročnosti úlohy jsem implementoval jen režim přepisu kódu. Tedy programu LinqRewrite jsou dány 2 argumenty, název souboru (může být i projekt, nebo řešení), který specifikuje přepisovaný kód, a název složky, kam se bude ukládat výstup. Nebyl však implementován režim doplnění překladače, samotného překladu a ani není možné projekt spouštět z NuGet balíčku.

Typický průběh přepisu začne zpracováním argumentů a buď bude vypsána nápověda k programu a nebo jsou první dva argumenty zpracovány a je zavolána kompilační služba. Ta nejprve pomocí překladače od Roslynu zkompiluje všechny projekty uvedené v souboru specifikované v prvním argumentu (pokud je uveden soubor bez projektu, jednoduchý projekt je vytvořen) a získá tím jejich syntaktické stromy a sémantické modely. Poté je zavolána třída `LinqRewriter`, která prochází syntaktické stromy, kontroluje zda výrazy volání funkcí mají být přepsány a pokud ano, volá třídu `InvocationRewriter`. Ta má dvě fáze. V první

fázi se nejdříve pokusí pomocí pravidel přepisu jednotlivých operátorů přepsat dotaz za jednoduchý výraz a pokud se to nepodaří, vytvoří pro vykonání dotazu metodu, která ho implementuje. Metodu dále přidá do současně přepisované třídy a volání metody pošle jako přepsaný kód. Po přepsání všech syntaktických stromů se zkopírují do výsledné složky (specifikované druhým argumentem) všechny přepsané i nepřepsané soubory specifikované prvním argumentem a program je ukončen.

Výhody	Nevýhody
Vkládá těl lambda výrazů do výsledného kódu	Nelze jednoduše rozšířit o nové operátory
Stejné volání jako u `System.Linq`	Potřeba použití externího programu
Procházení podle typu zdroje dat	Pomalejší kompilace
Optimalizováno pro neznámý počet výsledných prvků	Opakující se kód zvětšuje velikost výsledného programu
Přepis při překladu	Funguje jen pro C#
Nevytváří iterátory	
Implementována většina operátorů	
Přepis jednoduchých výrazů	
Možnost vypnutí kontrol	
Optimalizace proměnných	

Tabulka 5.1: Výhody a nevýhody použití LinqRewrite

5.2 Vytvoření kolekce SimpleList

Při použití kolekce `List<T>`, nejde zvenku nastavit jeho vnitřní pole a proto pokud máme prvky v poli a chceme je mít v listu, musíme je do něj překopírovat. Při použití pole musíme u dotazů LINQ zase často čelit nemožnosti mít pole jen částečně plné a je tedy nutné vytvořit nové pole a validní prvky překopírovat. Řešením tohoto problému může být kolekce `Span<T>`, ale ta je implementována jen pro .NET Core. Tedy u .NET Frameworku při kombinaci operátorů `Where.ToArray` při ukládání do pole musíme na konci pole zmenšit, aby neobsahovalo prvky navíc a při ukládání do listu musíme zase čelit větší době přístupu k prvku (u vyšších verzí .NET Frameworku a .NET Core není až tak velký rozdíl, jako u nižších verzí). Proto jsem pro potřeby výsledné implementace naprogramoval kolekci `SimpleList<T>`, což je kolekce založená na podobných principech jako `List<T>`, ale má veřejně přístupné pole prvků, takže LINQ může pracovat s vnitřním polem namísto s jeho dekorátorem.

5.3 Použití LinqRewrite

Pro funkční spuštění programu LinqRewrite je potřeba nainstalovat .NET Core pro spouštění aplikací[1] a také je potřeba nainstalovat msbuild se všemi závislostmi k přeložení nepřepsaného projektu[2]. Pro korektní funkci LinqRewrite metod je dále potřeba nainstalovat NuGet balíček LinqRewrite.Core do všech přepisovaných projektů, protože přepisované metody obsahují volání pomocných metod a tříd (např. počítání logaritmu).

Pro použití LinqRewrite je potřeba výsledný program spustit s argumenty přepisovaného souboru (podle koncovky souboru .cs, .csx, .csproj, .sln se přepisuje soubor, projekt, nebo řešení) a jako druhý argument musí být specifikována složka, kam má být přepis proveden. Při nespecifikování argumentů, nebo zavolání s argumentem -h, nebo –help je vypsána nápověda. Příklady použití programu LinqRewrite lze vidět ve výpisu 5.1. Pro inspiraci lze využít dvou připravených projektů pro testování a ukázání funkcionality. Přepsat je pomocí LinqRewrite jde spuštěním skriptů RunRewrittenBenchmarks.bat a RunRewrittenTests.bat, nebo napsáním příkazů ze zmíněných skriptů do příkazové řádky.

Pro optimální výsledek pomůže při zajištění podmínek dotazů operátor `Unchecked`, nebo atribut `UncheckedLinq` aplikovatelný na metody a třídy. Při použití `yield` vznikají uzávěry, které mají velký dopad na rychlost a alokace, takže v mnoha situacích není výkonnostně výhodné `yield` použít. Navíc při použití `yield` při měnícím se vnějším parametru v lambda výrazu nelze použít referenční parametr a tedy je potřeba vytvořit delegát a uzávěr pro parametr. `ForEach` je přepisován na rychlejší kód, než `foreach`, pokud je lambda výraz jednoduchý, nebo pokud se iteruje nad už přepisovaným dotazem. Pokud se nemění velikost kolekce je nejrychlejší použít jako výslednou kolekci pole, pokud se mění, je ve většině případech nejrychlejší použít `SimpleList<T>` (viz kapitola č. 5.2). Rychlostně nejlepší je použít jako vstupní kolekci pole, nebo `SimpleList<T>`, protože u kolekce `SimpleList<T>` se iteruje nad jeho vnitřním polem. Pokud to není možné, je nejlepší použít kolekci implementující rozhraní `IList<T>` s tím, že pokud je dotazu předán specifický typ kolekce, dotaz je rychlejší, než při specifikování jen rozhraní, nebo rodiče třídy. Pokud není možné použít jinou rychlejší metodu procházení kolekcí, lze využít i procházení pomocí `IEnumerable<T>`. Při potřebě ještě větší optimalizace je možné nechat kód přepsat a poté ještě ručně upravit a optimalizovat.

[1] Dostupný z https://dotnet.microsoft.com/download,nebopomocíprogramuVisualStudioInstaller

[2] Lze nainstalovat jako součástí Visual Studia, nebo Build Tools for Visual Studio https://visualstudio.microsoft.com/downloads/?q=build+tools

```
dotnet LinqRewrite.exe # Pro zobrazeni napovedy
dotnet LinqRewrite.exe -h # Pro zobrazeni napovedy
dotnet LinqRewrite.exe --help # Pro zobrazeni napovedy

dotnet LinqRewrite.exe soubor.cs directory # Pro prepsani souboru
                      # do specifikovane slozky
dotnet LinqRewrite.exe soubor.csx directory # Pro prepsani skriptu
                      # do specifikovane slozky
dotnet LinqRewrite.exe soubor.csproj directory # Pro prepsani projektu
                      # do specifikovane slozky
dotnet LinqRewrite.exe soubor.sln directory # Pro prepsani reseni
                      # do specifikovane slozky
```

Výpis 5.1: Použití programu LinqRewrite

Kapitola 6

Testování

Při testování přepisu LINQ dotazů je potřeba testovat 2 rozdílné části. První je verifikace (tedy jestli nově přepsané dotazy fungují stejně jako ty systémové) a druhá je rychlost. Jelikož systémový LINQ je pro výslednou implementaci referenční a cílem je naprogramoval LINQ, který pro stejné dotazy bude mít stejné výsledky, můžeme jako test napsat stejný dotaz přepsaný a nepřepsaný pomocí LinqRewrite a pokud jsou výsledky shodné, pro ten test byl přepis validní. Jako výsledek se počítá i výjimka, takže pokud v `System.Linq` nastane výjimka a v přepsané metodě nenastane, je to bráno jako chyba přepisu.

6.1 Verifikace funkčnosti LinqFaster

V LINQ lze napsat teoreticky až nekonečně množství různých dotazů. Ve výsledné implementaci je naprogramováno 24 vyhodnocovacích operátorů, které můžou být v dotazu maximálně jednou na konci dotazu, 3 metody generující data, které můžou být taky pro každý dotaz použito jen jednou na začátku dotazu (pokud je například `Enumerable.Range` jako parametr operátoru `Concat`, tak už je součástí jiného dotazu) a pak je 21 operátorů měnících enumeraci, kterých může být až neomezeně v libovolném pořadí. Dále je mnoho těch metod přetížených a nebo generických. I kdyby se počítalo jen z každého typu zdroje jeden (pole, list, `IEnumerable<T>`, `Enumerable.Range`, `Enumerable.Repeat`, `Enumerable.Empty<T>`), pak maximálně dva operátory měnící enumeraci a maximálně jeden vyhodnocovací operátor, tak i kdyby pro generické metody byly testy prováděny jen pro jeden typ a testovaly se všechny přetížené funkce, tak by bylo potřeba pro otestování vytvořit 6*41*41*83, tedy 837138 různých testů a to by byl jen zlomek funkcionality dotazů LINQ a navíc i kdyby byly testy generované, tak by kód vygenerovaných a přepsaných testů přesáhl 42 milionů řádků a testování by bylo časově náročné udělat. Proto nelze jednoduše udělat verifikace plné funkcionality ani zmíněné částečné funkcionality implementace.

Rozdělil jsem prováděné testování na 3 části. První část je testování kombinací operátorů. Jelikož by bylo náročné jak generováním, stejně i samotným přepisem a kontrolování testů otestovat všechny přetížené metody, vytvořil jsem generátor testů, který kontroloval jen některé přetížení operátorů LINQ. Bylo tedy otestováno z každého typu podporovaných funkcí 1-2 přetížení a byly otestovány dotazy zdroj, 1-2 změny iterace a možné zpracování do výsledku, nebo nechání výsledku jako `IEnumerable<T>`. Celkem bylo provedeno 141120 testů a je tedy ověřená funkcionalita těchto dotazů (Nelze 100% vyvozovat funkčnost ostatních přetížení, nebo delšího řetězení dotazů).

Jako druhou část jsem provedl manuální tvoření testů, které se snaží otestovat různé možnosti předávání parametrů, vytváření parametrů, různé typy předávání delegátů, instanční a statické metody a mnoho dalších možných částí LINQ. Celkem jsem takto vytvořil více než 4000 testů (v mnoha případech testy různých operátorů nad stejnými daty). Poslední částí bylo ověření, že některé operátory se při různé délce zdrojové kolekce chovají správně. Proto jsem vytvořil 40 testů s měnícím se parametrem pro kontrolu tohoto chování.

Tedy jsou otestované některé jevy, a jakákoli maximálně trojice po sobě jdoucích maximálně dvou selekčních operátorů a možného agregačního operátoru (v některých přetíženích). Testování bylo prováděno oproti .NET Frameworku 4.8 a oproti ostatním implementacím nemusí mít stejné výsledky.

6.2 Měření rychlosti implementací LINQ

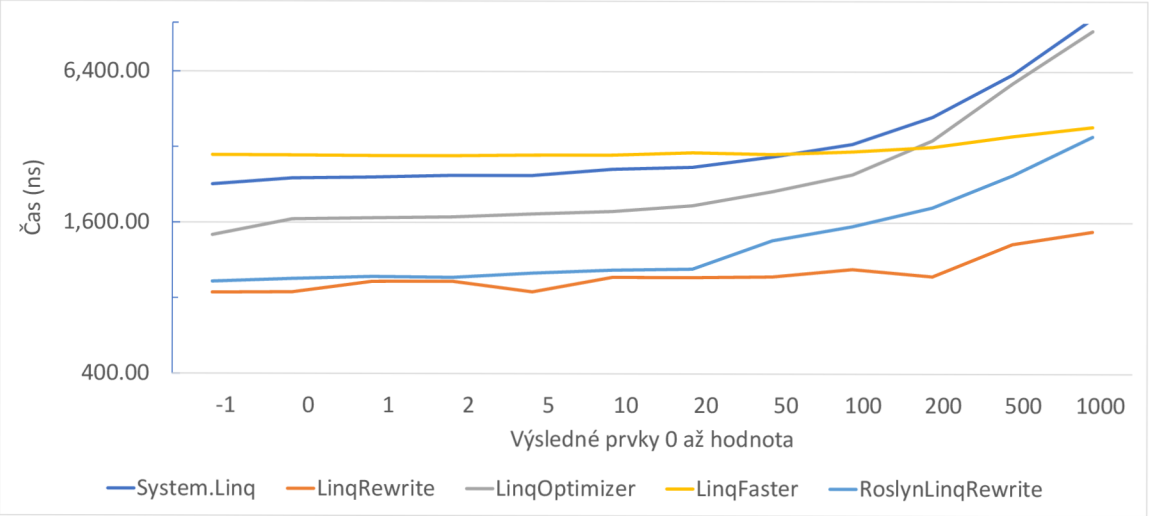
Po zajištění základní verifikace funkčnosti implementace LINQ je potřeba změřit jaké je zlepšení, nebo popřípadě zhoršení, kterého bylo dosaženo. Při snaze změřit, který kód je optimálnější nastávají dvě otázky. Co je kritérium, podle kterého určuji, že je kód optimální? Jakým způsobem změřím, nebo dokážu, že určitý kód toto kritérium splňuje? Optimálnost algoritmu můžeme posuzovat podle různých kritérií, například podle doby běhu algoritmu při vytížení, nebo podle počtu průběhů algoritmu za určitou jednotku času. Tyto kritéria nemusí při určitých podmínkách přímo souviset. Další možné způsoby měření může být podle počtu minutí cache, využití cpu, velikost velkých objektů na haldě (LOH), alokace paměti, délka trvání po dlouhé době a mnoho dalších [1].

Pro optimalizaci LINQ jsem jako hlavní kritéria měření rychlosti vybral dobu běhu metody při zatížení a pro některé dotazy jsem testoval i množství alokované paměti a dobu běhu při studeném startu. Vybral jsem takto, protože cílem této bakalářské práce je poskytnout LINQ, který bude použitelný v případě potřeby velké rychlosti kódu, což často znamená rychlost při častém opakování dotazu, ale zároveň by neměl alokovat moc paměti pro samotný běh algoritmu a čas studeného startu by neměl být příliš velký. Všechny testy uvedené v kapitole 6.2 jsem spouštěl na notebooku s 64 bitovým procesorem Intel Core i7-8705G, 64 bitovým Windows Home a 16 GB operační paměti. Na jiném zařízení by bylo dosaženo pravděpodobně jiných výsledků.

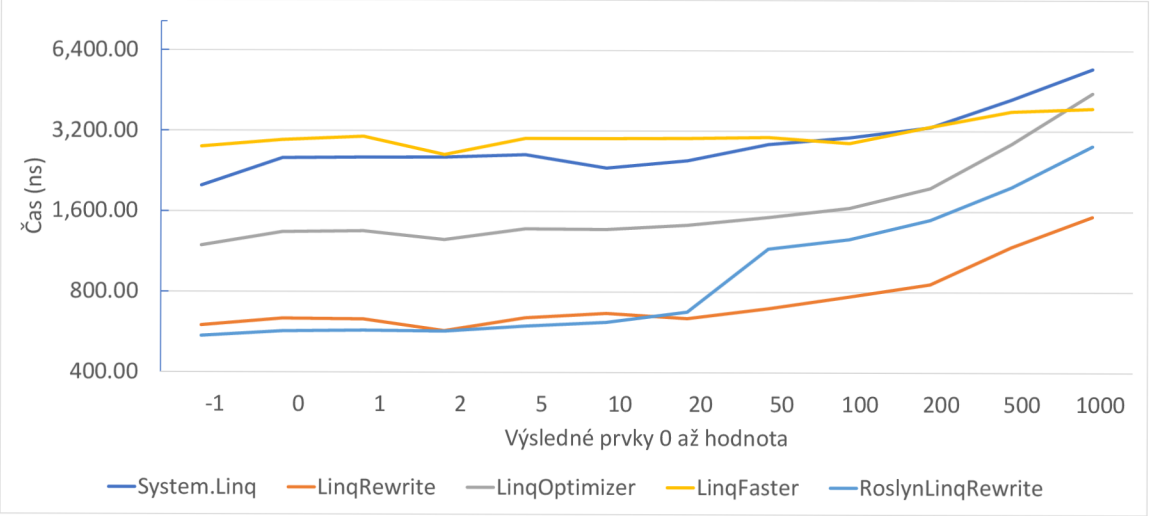
Všechna měření zmíněná v kapitole č. 6.2 umožňuje volně přístupná knihovna BenchmarkDotNet. BenchmarkDotNet pomocí statistických metod při velkém počtu spuštění kódu počítá délku trvání metody, její režii, rozptyl, zajišťuje nezávislost pokusů a umožňuje vyvarovat se nástrahám měření rychlosti běhu kódu. Jelikož pomocí LINQ můžeme vytvořit až nekonečné množství různých dotazů, je v podstatě nemožné dokázat, že je jedna implementace LINQ rychlejší, než ostatní implementace ve většině možných případů. Navíc kombinace operátorů jsou v různých implementacích různě optimalizovány a tedy jako jsme mohli pro kombinaci maximálně 2 selekčních a agregačního operátoru udělat 837138 testů funkčnosti, můžeme udělat i stejné množství testů rychlosti. Jelikož ale provedení testu rychlosti je mnohonásobně náročnější, než test správnosti výsledku, testoval jsem metody doplněné maximálně vyhodnocovací metodou, nebo operátory `Select` a `Where`.

Vytvořil jsem tedy sadu testů rychlosti vždy test nepřepsaného kódu a test stejného kódu přepsaného pomocí LinqRewriter. Jelikož ostatní implementace LINQ nepodporují mnoho z testovaných operátorů a navíc byly pro různé situace nevhodné z jiných důvodů, než kvůli rychlosti například LinqFaster vytváří zbytečné alokace a používá neefektivní způsob řetězení dotazů, LinqOptimizer má více než 30 us režii kompilace a Roslyn linq rewrite chybí velké množství operátorů a je složité ho zprovoznit. Vytvořil jsem tedy jen několik

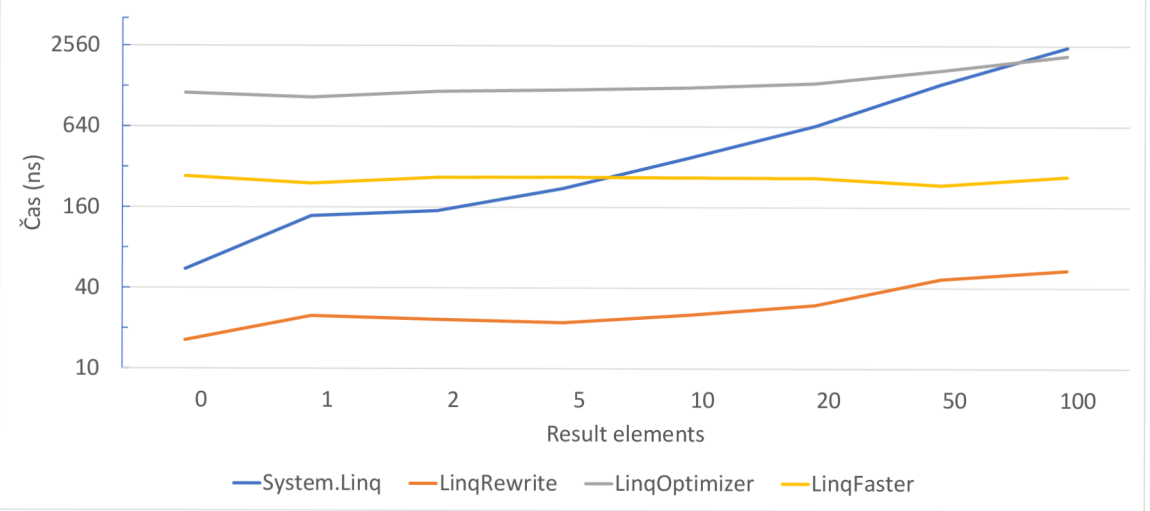
testů pro porovnání mezi jednotlivými implementaci a zbytek testů porovnává jen rychlost `System.Linq` a LinqRewrite. Na obrázcích 6.1 a 6.2 lze vidět rychlost provádění dotazu při neznámém počtu výsledných prvků a na obrázcích 6.3 a 6.4 lze vidět rychlost při známém počtu výsledných prvků. Na všech zmíněných obrázcích je kvůli velkému rozdílu časů použita logaritmická časová osa. Z grafů ale nelze vyvozovat moc závěrů, protože ukazují rychlost jen nepatrné části funkcionality LINQ (například pro kombinaci `ArraySource.Sum` dosáhl při testování nejrychlejší čas LinqFaster). Důležité je zmínit, že ve výsledcích nebyla započítaná režie kompilace LinqOptimizeru, jinak se musí ke všem jeho hodnotám přičíst přibližně 30 us (závisí na složitosti kompilace). Jde vidět, že u kombinace operátorů `Select.Where.ToArray` LinqFaster má skoro konstantní čas kvůli typu zvětšovacího algoritmu (není výhodné pro malý počet výsledných prvků). Ve všech zmíněných grafech lze vidět, že má LinqRewrite oproti ostatním implementacím velice dobré výsledky.



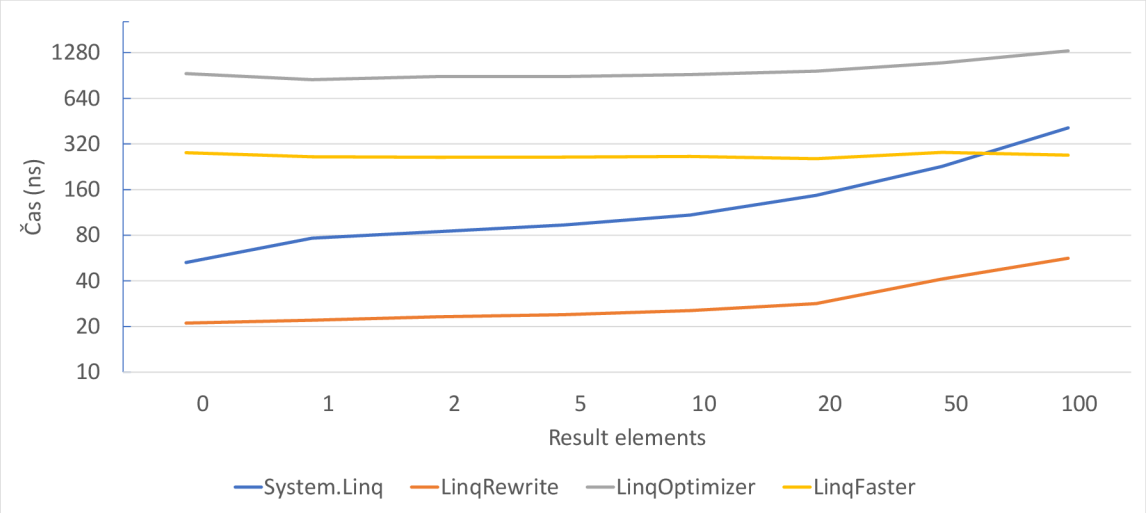
Obrázek 6.1: Porovnání implementací kombinace operátorů Select.Where.ToArray pro různé implementace LINQ nad .NET Framework 4.8



Obrázek 6.2: Porovnání implementací kombinace operátorů Select.Where.ToArray pro různé implementace LINQ nad .NET Core 3.1



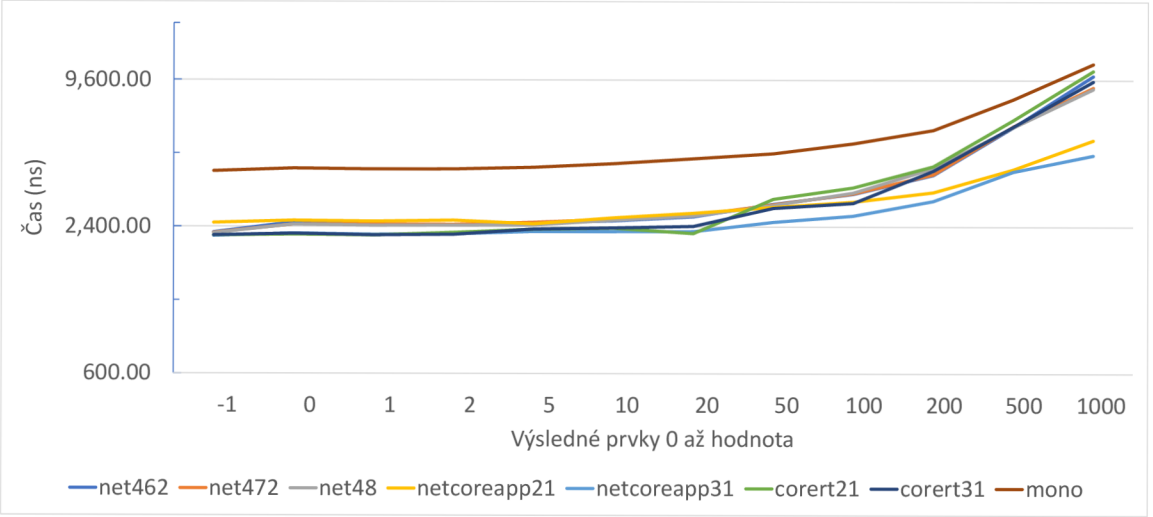
Obrázek 6.3: Porovnání implementací kombinace operátorů Skip.Take.ToArray pro různé implementace LINQ nad .NET Framework 4.8



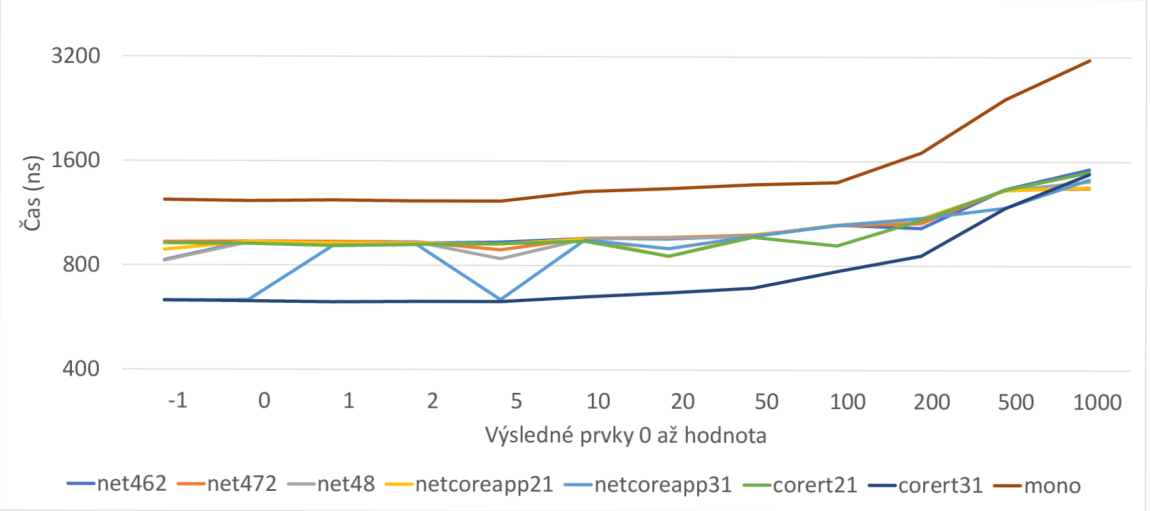
Obrázek 6.4: Porovnání implementací kombinace operátorů Skip.Take.ToArray pro různé implementace LINQ nad .NET Core 3.1

6.3 Rychlost LinqFaster pro implementace .NET

Každá platforma .NET implementuje modul `runtime` trochu jinak a tedy i rychlost je závislá na typu platformy. Jelikož jsem naprogramoval LinqRewrite tak, aby byl kompatibilní s jakoukoli implementací .NET Frameworku od verze 4.0 a .NET Core od verze 2.1, je potřeba zkontrolovat, jak dobrá je optimalizace přepisu ve všech podporovaných implementacích .NET. Pomocí BenchmarkDotNet nelze testovat rychlost pro .NET Framework nižší než 4.6.2, proto pro nižší verze .NET Frameworku nemám data, a při měření rychlosti metod bez pomocné knihovny bych se pravděpodobně dopustil chyb, kvůli náročnosti dosáhnutí stejných startovacích podmínek, proto jsem prováděl testování jen pro implementace podporované knihovnou BenchmarkDotNet. Udělal jsem jednoduchý test pro porovnání rychlosti kombinace operátorů `Select.Where.ToArray` implementované v LinqRewrite a `System.Linq`. Výsledky jsem zaznamenal do grafů na obrázcích 6.5 a 6.6. Z grafů lze vidět, že různé implementace mají pro kombinaci operátorů `Select.Where.ToArray` podobné, ale ne stejné chování. Při porovnání výsledků testů vychází v každé implementaci lépe LinqRewrite, ale z tak malého množství testů nelze generalizovat, že je obecně lepší LinqRewrite. Jelikož se jednotlivé implementace chovají podobně a bude jednodušší a přehlednější testovat jen jednu platformu, budu nadále ukazovat výsledky jen pro .NET Framework 4.8, i když pro přesnou představu by měly být provedeny pro každou podporovanou implementaci .NET.



Obrázek 6.5: Porovnání rychlosti kombinace operátorů Select.Where.ToArray v System.Linq v závislosti na implementaci .NET

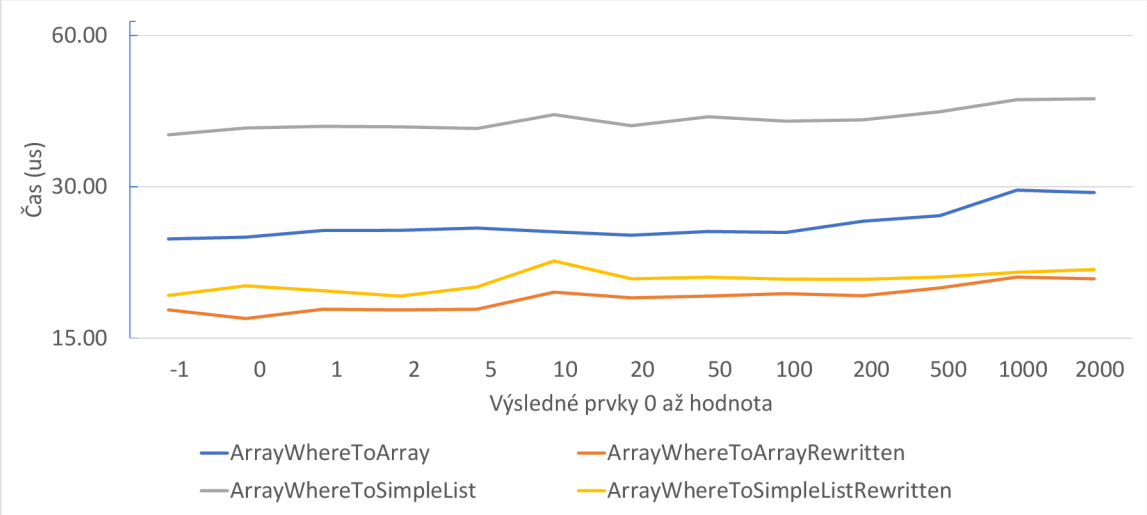


Obrázek 6.6: Porovnání rychlosti kombinace operátorů Select.Where.ToArray v LinqRewrite v závislosti na implementaci .NET

6.4 Rychlost studeného startu LinqFaster

.NET runtime za běhu aplikace provádí optimalizace kódu jako vkládání těl krátkých funkcí do nadřazených funkcí, optimalizace aritmetických výrazů, mazání kódu, který nemá žádný efekt a další. Testoval jsem tedy rychlost studeného startu (kde mnoho zmíněných optimalizací ještě není provedených) pro kombinaci operátorů `Select.Where` pro pole a výsledky zanesl do grafu. Z grafu na obrázku 6.7 pak lze vyčíst, že kombinace operátorů `Select.Where` je při studeném startu rychlejší, pokud je přepsaný pomocí LinqRewriteru. To je pravděpodobně způsobeno tím, že optimalizace prováděné za běhu jsou u LinqRewri-

teru provedené už při přepisování a také to může být režií vytváření a volání iterátorů a delegátů u `System.Linq`.



Obrázek 6.7: Rychlost studeného startu kombinace operátorů Where.ToArray pro LinqRewrite a System.Linq

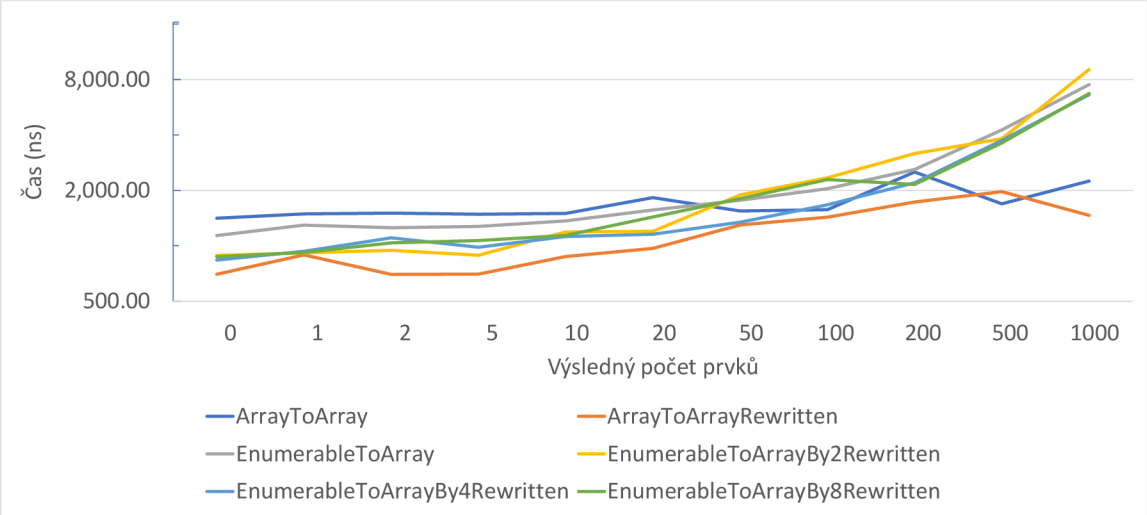
6.5 Rychlost operátorů LinqFaster

V této bakalářské práci se hlavně zaměřuji na optimalizaci LINQ pro .NET Framework, proto následující testy budou zaměřené na současně nejnovější .NET Framework, verzi 4.8. V sekci 6.3 sice testy ukazují pro různé implementace podobné zrychlení, ale pro vyvození nějakých závěru by bylo potřeba provést více testů pro různé operátory a konfigurace. Kvůli velkým rozdílům v rychlostech metod jsou všechny časové osy v následujících grafech logaritmické.

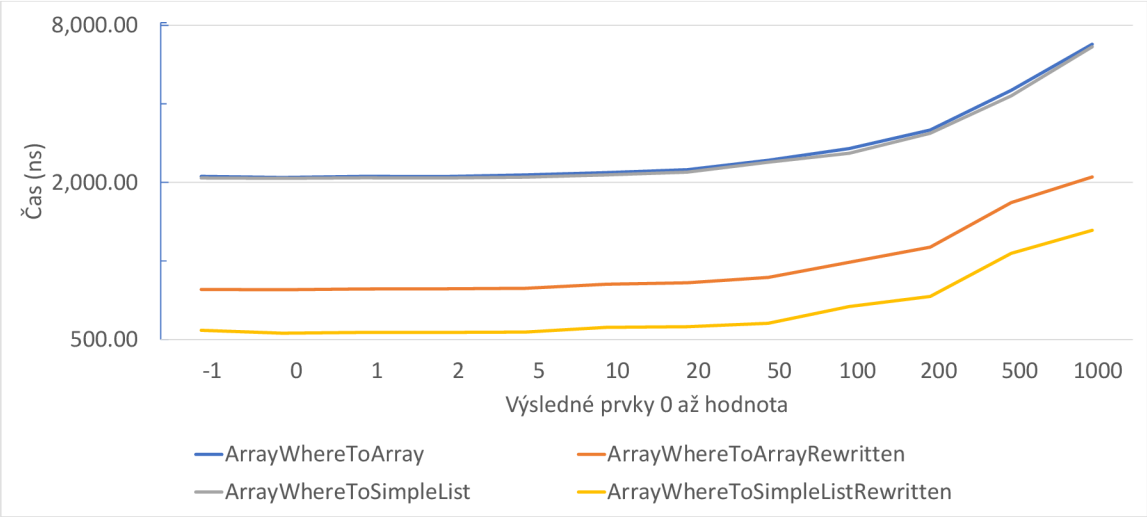
První skupinou testovaných operátorů byly operátory u kterých parametrem můžeme měnit výsledný počet prvků (u operátoru `ToArray` je počet prvků měněn pomocí velikosti vstupního pole). Do této skupiny patří například kombinace operátorů `ToArray`, `Where.ToArray`, `TakeWhile.ToArray`, `Skip.ToArray`, nebo `Take.ToArray`. Při nutnosti použití zvětšovacího algoritmu byly naměřené časy u většiny operátorů LinqRewrite lepší (u `EnumerableSource.ToArray` je počítán jako vztažný algoritmus zvětšování zdvojnásobení velikosti pole a je pro některé hodnoty horší), než časy `System.Linq`. Výsledky testování některých zmíněných algoritmů jsou vidět na obrázcích 6.8, 6.9 a 6.10.

Velké rozdíly rychlosti jsem naměřil u operátorů `Skip` a `Take`, případně jejich kombinaci. Při přeskočení, či vzetí proměnného počtu prvků při velikosti výsledného pole do 100 prvků bylo zrychlení od 16x do 200x a při konstantním počtu přeskočených, či vzatých prvků a použití operátoru `Unchecked` od 31x do 300x. Bez použití operátoru `Unchecked` musí být velikost kontrolována a nelze tedy operátor přepsat na jednoduchý příkaz při malém počtu výsledných prvků. Velké zrychlení u operátorů `Skip` a `Take` je pravděpodobně způsobeno použitím informace známé velikosti výsledného pole, jiným způsobem kopírování a zmenšením režie (nejsou použité iterátory). Rychlejší časy u `EnumerableSource.ToArray` byly pravděpodobně způsobené správným procházení předaného pole, které bylo v metodě

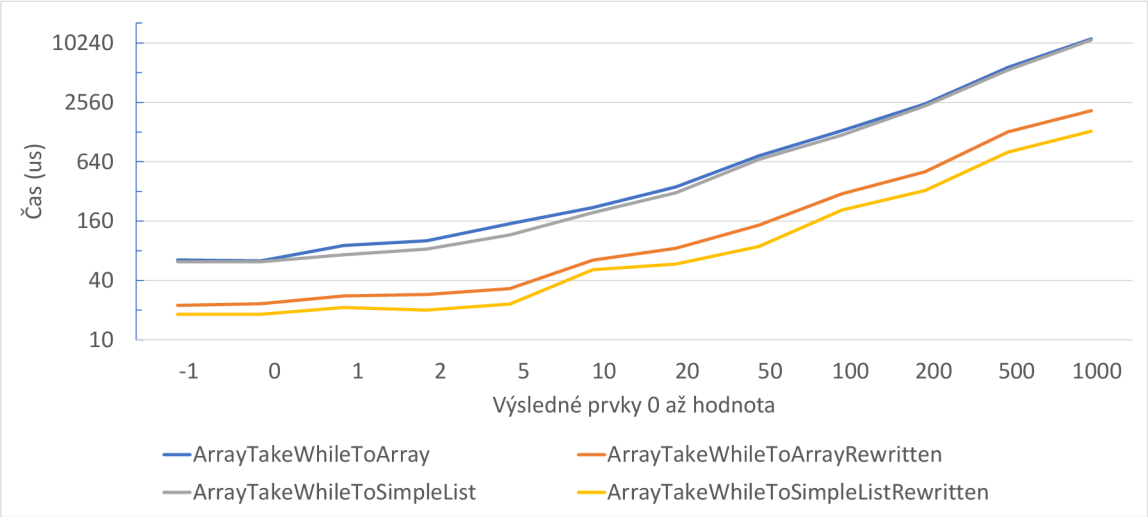
předáno jako `IEnumerable<T>` (v LinqRewrite není podporováno kontrola typu kolekce za běhu programu viz kapitola 4).



Obrázek 6.8: Rychlost operátoru ToArray pro různé implementace v závislosti na počtu prvků

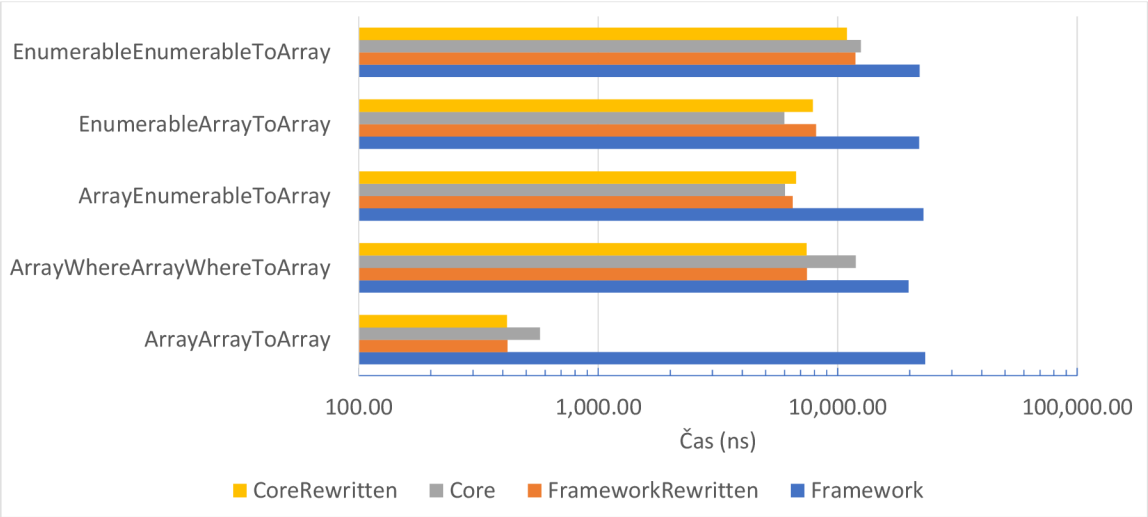


Obrázek 6.9: Rychlost kombinace operátoru Where.ToArray pro různé implementace v závislosti na počtu výsledných prvků

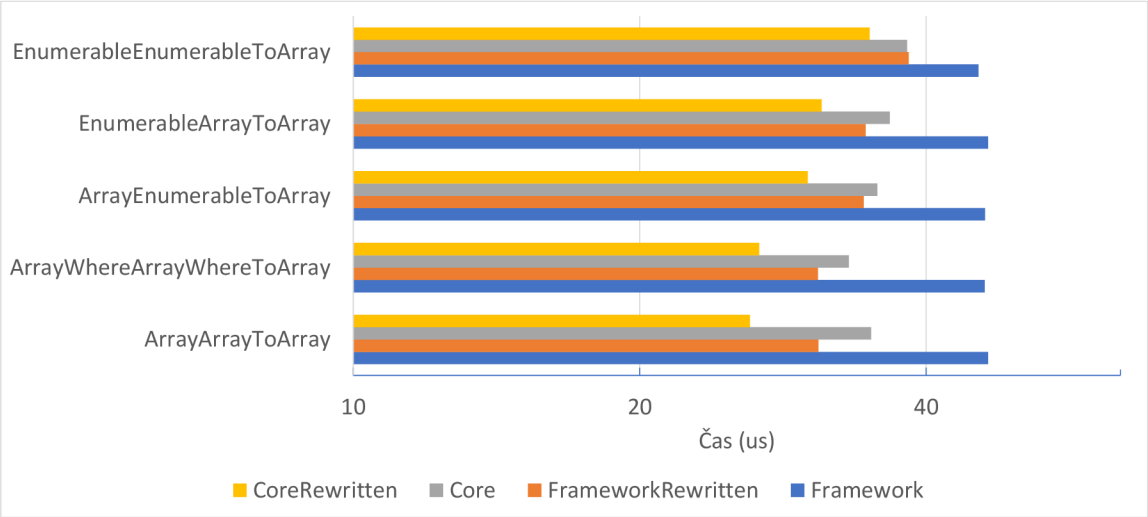


Obrázek 6.10: Rychlost kombinace operátoru TakeWhile.ToArray pro různé implementace v závislosti na počtu výsledných prvků

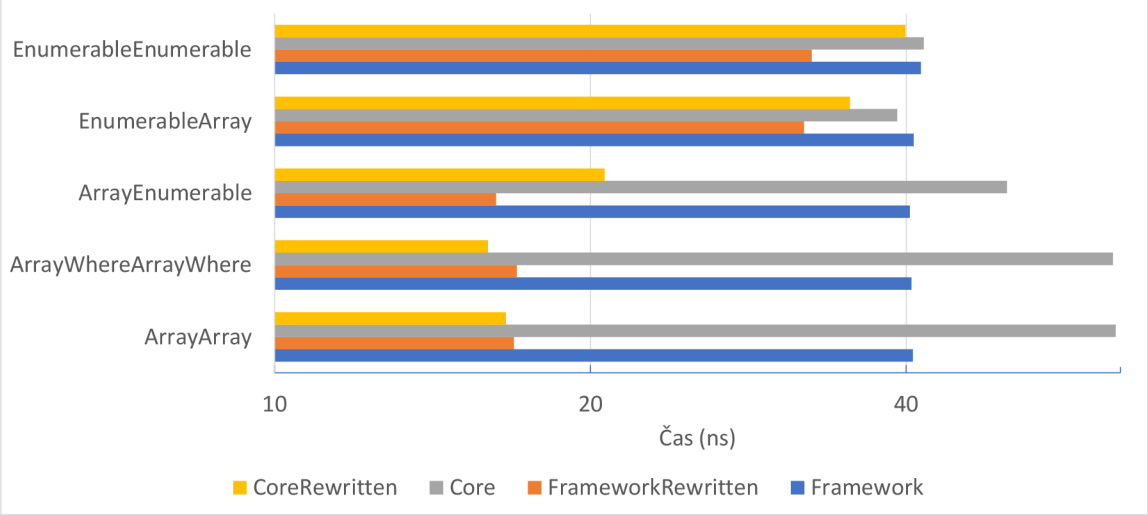
Do druhé skupiny testovaných operátorů, jsem zařadil ty operátory, které nějak spojují kolekce, například `Join`, `Except`, `Concat`, nebo `SequenceEqual`. U .NET Core 3.1 operátor `Concat` umí rozpoznat, že je je při testování použit jako zdroj `Enumerable.Range` (v LinqRewrite není podporováno určování typu kolekce za běhu viz kapitola 4) a tedy nemusí používat algoritmus postupného zvětšování. Největší úspěch LinqRewrite je značné zrychlení operátoru `Concat` (až 50x zrychlení) a `SequenceEqual` (až 2.2x zrychlení) pro pole u .NET Frameworku, ale průměrné zrychlení ze všech testů této kategorie bylo 3.16x a mediánové 1.42x. Výsledky některých testů jsou zobrazeny na obrázcích 6.11, 6.11 a 6.13.



Obrázek 6.11: Porovnání rychlosti operátoru Concat v System.Linq a LinqRewrite

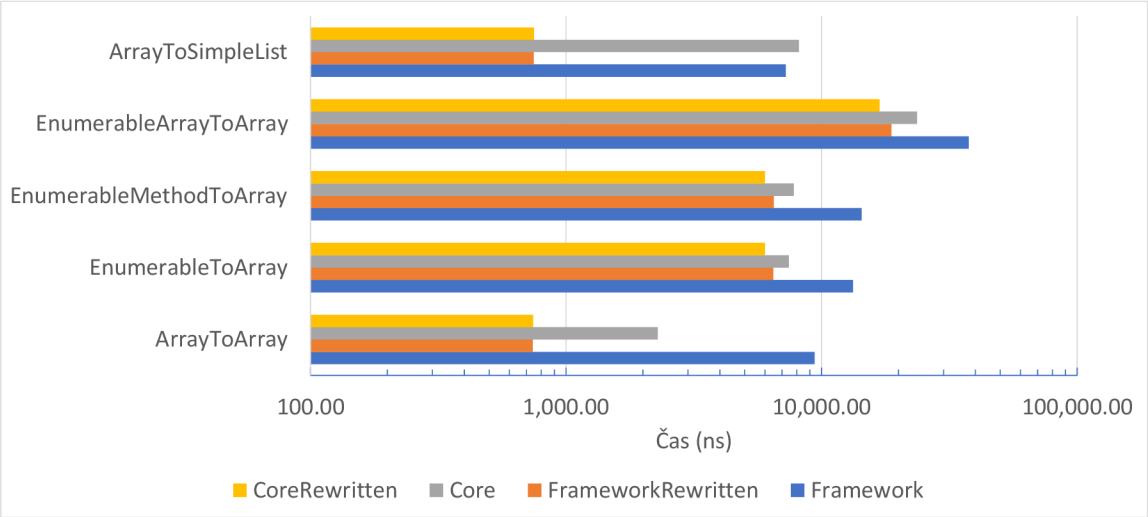


Obrázek 6.12: Porovnání rychlosti operátoru Union v System.Linq a LinqRewrite

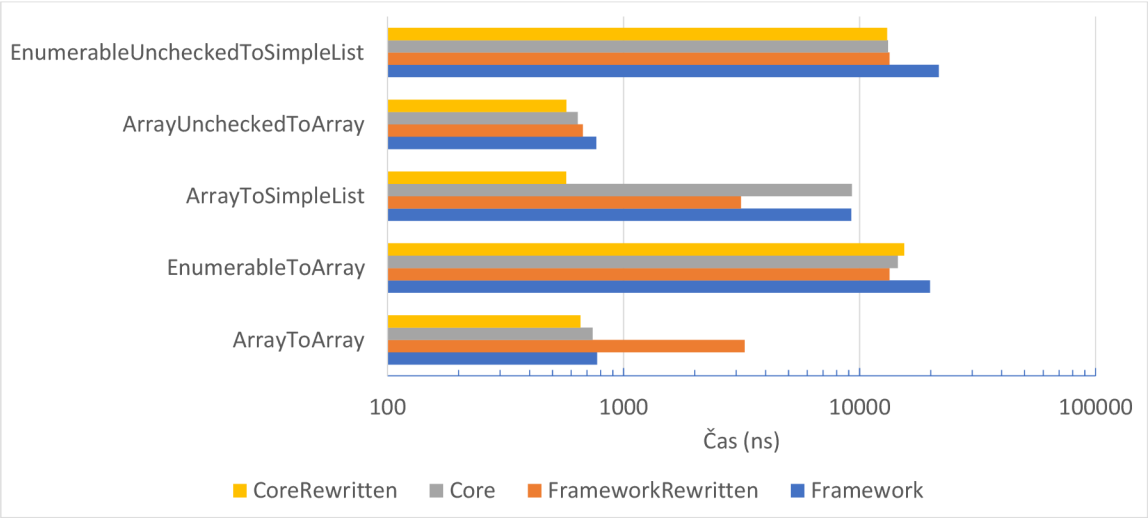


Obrázek 6.13: Porovnání rychlosti operátoru SequenceEqual v System.Linq a LinqRewrite

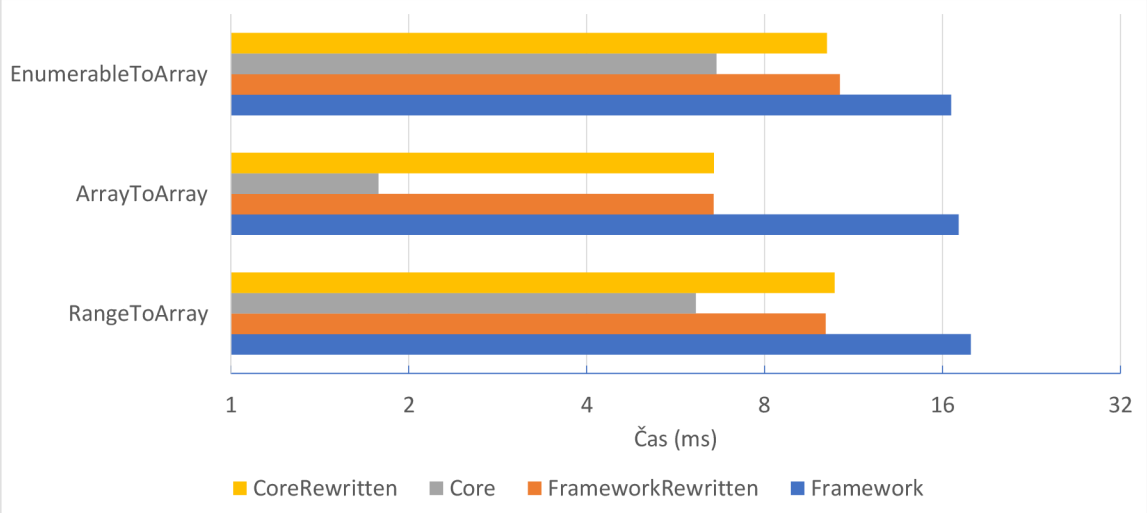
Jako třetí skupinu operátorů jsem testoval operátory měnící enumeraci, zároveň nepatřící do první skupiny. V této skupině se ukázalo, že pokud není použit operátor `Unsafe`, tak v některých situacích je přepsaný operátor `Cast` pomalejší, než u `System.Linq`. Také v .NET Core je implementované více optimalizací pro operátor `SelectMany` a operace používající množiny nad `IEnumerable<T>`, než v LinqRewrite a tedy dosahují lepších výsledků. Ze všech naměřených testů z této kategorie měl LinqRewrite průměrné zrychlení 22.7x a mediánové 2.68x. U .NET Core je pro některé operátory (např. SelectMany) rychlostně výhodnější použít atribut `NoRewrite`. Výsledky některých testů jsou zobrazeny na obrázcích 6.14, 6.15 a 6.16.



Obrázek 6.14: Porovnání rychlosti operátoru Select v System.Linq a LinqRewrite

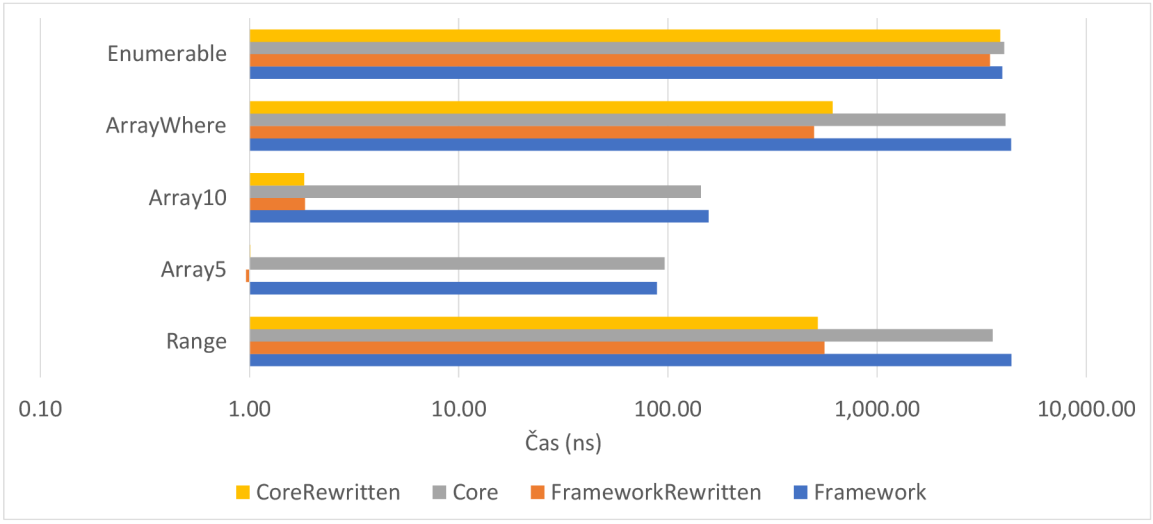


Obrázek 6.15: Porovnání rychlosti operátoru Cast v System.Linq a LinqRewrite

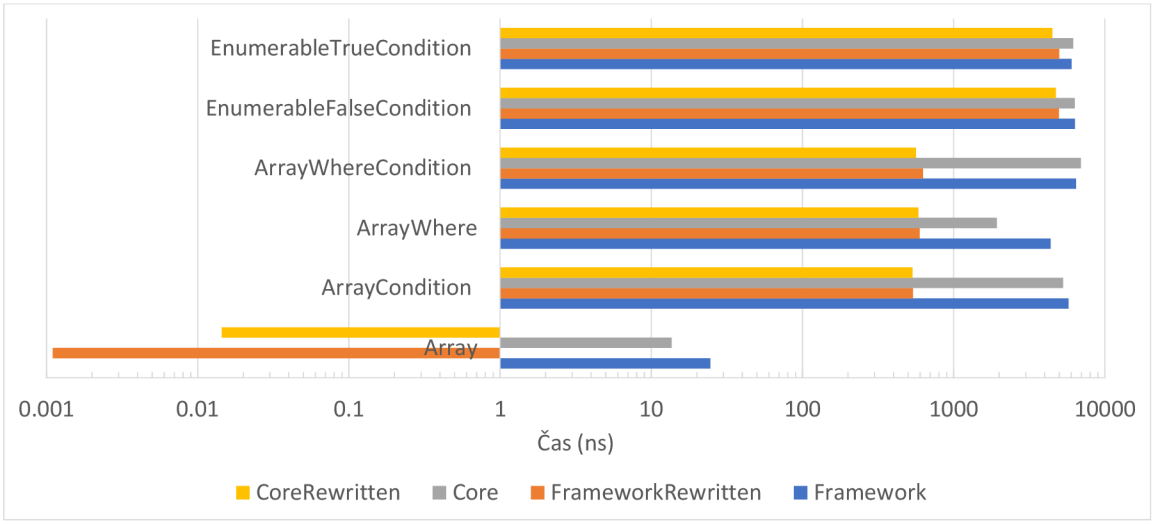


Obrázek 6.16: Porovnání rychlosti operátoru SelectMany v System.Linq a LinqRewrite

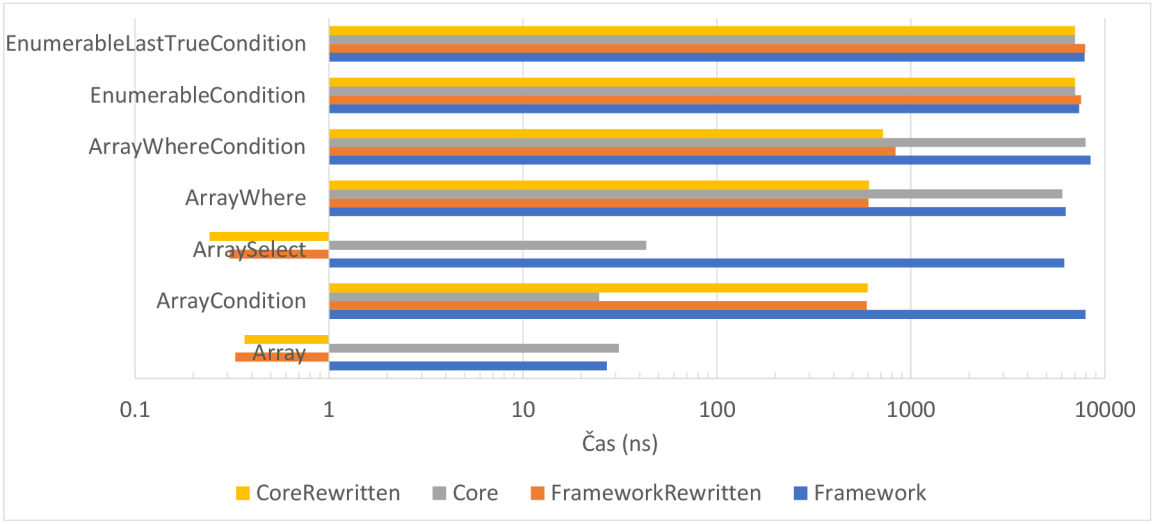
Největší rozdíly jsem naměřil při testování agregačních operátorů. Některé agregační operátory totiž jdou přepsat do jednoduchých výrazů a tedy není potřeba procházet cyklus. Příkladem může být operátor `Count`, nebo `Last` bez predikátu, nebo při zajištění podmínek a použití operátoru `Unchecked` operátor `Sum`, ale jednoduché přepsání je podporováno pro mnoho kombinací operátorů. U .NET Core je při hledání posledního prvku splňujícího podmínku jako optimalizace implementováno procházení cyklu z opačné strany. Na to, aby tento přístup byl validní (u .NET Frameworku), musí být zajištěno správné chování při vytváření nějakého vedlejšího efektu, třeba vypsání do konzole. Takže v tomto kroku se tyto implementace liší. Jelikož je LinqRewrite dělán podle .NET Frameworku, při použití operátoru `Last` je cyklus procházen normálně a pokud je zajištěno, že není žádný vedlejší efekt, může programátor použít kombinaci operátorů `Reverse.First`. Kvůli přepisování na jednoduché výrazy byly naměřeny velké rozdíly rychlosti, a tedy průměrně byla rychlost lepší ve všech testech této kategorie 94850x a mediánově 7.7x. Při započítání jen výrazů, které nemohly být jednoduše zrychleny bylo průměrné zrychlení 6.5x a mediánové 5.4x. Výsledky některých testů jsou zaznamenány v grafech na obrázcích 6.17, 6.18, 6.19 a 6.20.



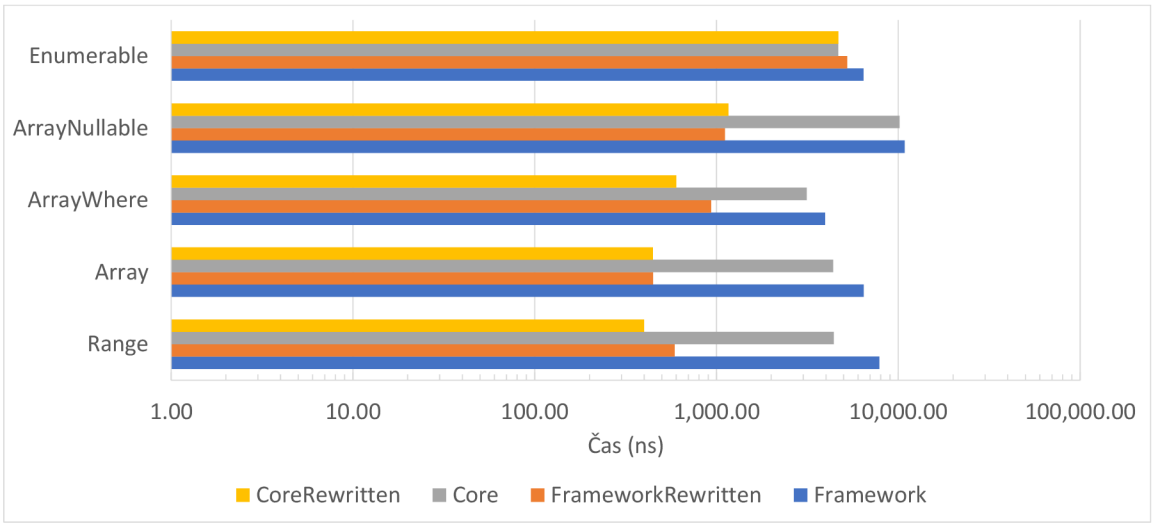
Obrázek 6.17: Porovnání rychlosti operátoru Aggregate v System.Linq a LinqRewrite



Obrázek 6.18: Porovnání rychlosti operátoru Count v System.Linq a LinqRewrite



Obrázek 6.19: Porovnání rychlosti operátoru Last v System.Linq a LinqRewrite



Obrázek 6.20: Porovnání rychlosti operátoru Max v System.Linq a LinqRewrite

Z provedených testů nelze dokázat, že LinqRewrite je rychlejší, než `System.Linq`, protože provedené testy ukazují rychlost jen některých dotazů LINQ a také i naměřené testy ukázaly, že přepsané dotazy jsou rychlejší jen v některých situacích. V provedených testech ale mají přepsané dotazy až na pár případů velice dobré výsledky a umožňuje vložit některé optimalizace implementované až v .NET Core i do staršího .NET Frameworku a jiných platforem a jsou přidány další, nově vytvořené optimalizace. Přepsané dotazy mají také nižší paměťovou náročnost.

Kapitola 7

Závěr

Z výsledků testů lze vidět, že implementace programu pro náhradu LINQ dotazů za jejich procedurální ekvivalentní kód je základně otestovaný, a může zrychlit provádění dotazů LINQ. Zjednodušený závěr by mohl být, že pro jednoduché dotazy lze docílit zrychlení i 50000x, pro agregace a jednoduché změny iterace několikanásobné, a při množinové modifikace iterace, nebo při provádění dotazů nad `IEnumerable<T>` o několik desítek procent. Toto je ale zjednodušený závěr a docílené zrychlení závisí na zdroji dat, použitých operátorech a také na množství poskytnutých informací LinqRewriteru. Pro určité operátory a při nedostatku informací jsou rychlejší jiné implementace a to třeba u .NET Core pro operátor `SelectMany`, nebo operátor `Cast` bez použití operátoru `Unchecked` (či atributu `UncheckedLinq`), nebo při zobecněné zdrojové kolekci (pole předané jako `IEnumerable<T>`).

Při použití LinqRewrite lze dále docílit snížení počtu alokací, množství alokované paměti a také snížit počet uzávěrů. Dle mého názoru jsem navrhl a naprogramoval užitečný nástroj, který může pomoct optimalizovat kód pro různé implementace .NET. Umožňuje mít kód jednodušší a při dotazování nad kolekcemi snížit množství potřebného procedurálního kódu a umožňuje ho použít i v situacích potřeby velké rychlosti kódu (i když pro některé situace je potřeba ještě přepsaný kód ještě více optimalizovat). Kvůli tomu může být urychlen vývoj a zlepšena čitelnost a spravovatelnost kódu. Myslím tedy, že základní cíl této bakalářské práce byl splněn.

Tomuto tématu by bylo dobré se dál věnovat a to nejen zlepšování implementace přepisu LINQ, ale zkoumat další možné způsoby optimalizace, implementovat zbývající operátory v případě potřeby přidávat další operátory usnadňující dotazy nad kolekcemi. Velkou zlepšením knihovny by bylo doprogramování chybějících a zlepšení stávající implementací optimalizací (např. operace nad vektory, paralelní provedení dotazu, optimalizace matematických výrazů). Pro několik operátorů by mohlo rychlostně pomoct vytvořit přepínač prováděného algoritmu (např. pro operátor `OrderBy`). Bylo by užitečné princip přepisu integrovat přímo do některého překladače pod nějaký přepínač, i když pro tyto účely by knihovna musela být upravena a více otestována. Myslím, že tato práce je celkem velkým krokem k optimalizaci dotazů LINQ a doufám, že práce na tomto tématu a projektu bude pokračovat.

Literatura

[1] Akinshin, A. *Pro .NET Benchmarking*. 1. vyd. New York: Apress, červen 2019. 690 s. ISBN 978-1-4842-4940-6.

[2] Carter, P. *.NET architectural components* [online]. Microsoft, 2017 [cit. 25. května 2020]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/components.

[3] Craig, I. *Object-Oriented Programming Languages: Interpretation*. London: Springer London, 2007. 240 s. ISBN 978-1-84628-773-2.

[4] Dietrich, E. *The history of C#* [online]. Microsoft, 2020 [cit. 26. května 2020]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history.

[5] Dotnet. *BenchmarkDotNet* [online]. 0.12.1. GitHub, 2020 [cit. 25. května 2020]. Dostupné z: https://github.com/dotnet/BenchmarkDotNet.

[6] Dotnet. *.NET Core Runtime (CoreRT)* [online]. 0.0.1. GitHub, 2020 [cit. 25. května 2020]. Dostupné z: https://github.com/dotnet/corert.

[7] Gray, J. *Writing Faster Managed Code: Know What Things Cost* [online]. Microsoft, 2007 [cit. 25. května 2020]. Dostupné z: https://docs.microsoft.com/en-us/previous-versions/dotnet/articles/ms973852(v=msdn.10).

[8] Martinelli, A. *Roslyn-linq-rewrite* [online]. 1.0.1.11. GitHub, 2020 [cit. 25. května 2020]. Dostupné z: https://github.com/antiufo/roslyn-linq-rewrite.

[9] Meijer, E. The world according to LINQ. *Communications of the ACM*. ACM. 2011, roč. 54, č. 10, s. 45–51. ISSN 00010782.

[10] Mono. *Mono* [online]. 6.10.0.105. GitHub, 2020 [cit. 25. května 2020]. Dostupné z: https://github.com/mono/mono.

[11] Mott, J. *LinqFaster* [online]. 1.0.0. GitHub, 2019 [cit. 25. května 2020]. Dostupné z: https://github.com/jackmott/LinqFaster/tree/master/LinqFaster.

[12] Mullan, E. *Parallelism on a Single Core - SIMD with C#* [online]. Instil, 2016 [cit. 25. května 2020]. Dostupné z: https://instil.co/2016/03/21/parallelism-on-a-single-core-simd-with-c/.

[13] Murray, D., Isard, M. a Yu, Y. Steno: automatic optimization of declarative queries. *ACM SIGPLAN Notices*. Srpen 2012, roč. 47, s. 121.

[14] Palladinos, N. a Rontogiannis, K. *LinqOptimizer* [online]. Nessos, 2019 [cit. 25. května 2020]. Dostupné z: https://github.com/nessos/LinqOptimizer.

[15] Palladinos, N. a Rontogiannis, K. *LinqOptimizer* [online]. 0.7.0. GitHub, 2019 [cit. 25. května 2020]. Dostupné z: https://github.com/nessos/LinqOptimizer.

[16] Pialorsi, P. *Microsoft LINQ : kompletní průvodce programátora*. 1. vyd. Brno: Computer Press, září 2009. 616 s. Programování. ISBN 978-80-251-2735-3.

[17] Posadas, M. *Mastering C# and .NET Framework*. 1. vyd. Birmingham: Packt Publishing Ltd., 2016. 560 s. ISBN 978-1-78588-437-5.

[18] Troelsen, A. W. *Pro C# 7 : With .NET and .NETCore*. 8. vyd. New York: Apress, listopad 2017. 1351 s. ISBN 978-1-4842-3017-6.

[19] Wagner, B. *Introduction to LINQ Queries (C#)* [online]. Microsoft, 2015 [cit. 26. května 2020]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq-queries.

[20] Wagner, B. *Introduction to the C# language and the .NET Framework* [online]. Microsoft, 2015 [cit. 25. května 2020]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework.

[21] Wagner, B. *Write LINQ queries in C#* [online]. Microsoft, 2016 [cit. 25. května 2020]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/linq/write-linq-queries.

[22] Wagner, B. *Expression Trees (C#)* [online]. Microsoft, 2017 [cit. 26. května 2020]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/expression-trees/.

[23] Wagner, B. *Language Integrated Query (LINQ)* [online]. Microsoft, 2017 [cit. 25. května 2020]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/.

[24] Wagner, B. *A Tour of C# - C# Guide* [online]. Microsoft, 2020 [cit. 25. května 2020]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/.

[25] Warren, G. *Compiling Apps with .NET Native* [online]. Microsoft, 2017 [cit. 28. května 2020]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/net-native/.

[26] Warren, G. *Managed Execution Process* [online]. Microsoft, 2017 [cit. 25. května 2020]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process.

[27] Warren, G. *Version compatibility* [online]. Microsoft, 2020 [cit. 26. května 2020]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/migration-guide/version-compatibility.

[28] Warren, M. *CoreRT - A .NET Runtime for AOT* [online]. 2018 [cit. 25. května 2020]. Dostupné z: https://mattwarren.org/2018/06/07/CoreRT-.NET-Runtime-for-AOT/.

[29] Watson, B. *Writing High-Performance .NET Code*. 2. vyd. Los Gatos: Smashwords Edition, duben 2018. 519 s. ISBN 978-0-990-58349-3.