Dynamic compilation as a very fast mathematic parser in .NET

Petr Čápek, Erik Král

***Abstract*—** This work aims to investigate how to use the process of a dynamic compilation as a very fast mathematic parser. Firstly, it focuses on describing the process of a standard compilation in .NET platform and describing how a parser engine works. The core of this work is the implementation of our own mathematic parser which is based on a dynamic compilation to provide a very fast mathematic parser engine. This implementation is compared to existing mathematic parser libraries using various benchmarks.

***Keywords*—**.NET, math parser, dynamic compilation, computing, benchmark

I. Introduction

IN the world of science, you very often complain about the evaluation of some mathematic formulas. You have some data and you need to apply functions to this data. Small amounts of data can be calculated by hand but for large amounts of data you need to use the power of computer.

For simple calculations you can use a type of spreadsheet software which allows you to easily modify functions expression if you need to. For complex data processing you very often need to create and compile you own program. [1] [2] For example compiling process of .NET framework used 2 way compilations as is shown in Fig. 1.

In some cases, there is a requirement to give users abstract control to change the mathematic expression in a program without recompiling or reinstalling the program. If your program uses a method of data processing with formulas that can be changed, you need to choose the right techniques to allow users to do that.

One of the solutions is to provide a predefined set of functions for users so a user can choose a function formula from it.

Another solution is to provide the ability for users to design and use their own formulas.

In the latter option you must implement some sort of mathematic parser engine which allows users to enter new formulas into the software.

All authors are with Faculty of Applied Informatics, Tomas Bata University in Zlín, Nad Stráněmi 4511, 760 05 Zlín (phone: +420-57-603-5188; fax: +420 57-603-2716; e-mail: ekral@ fai.utb.cz

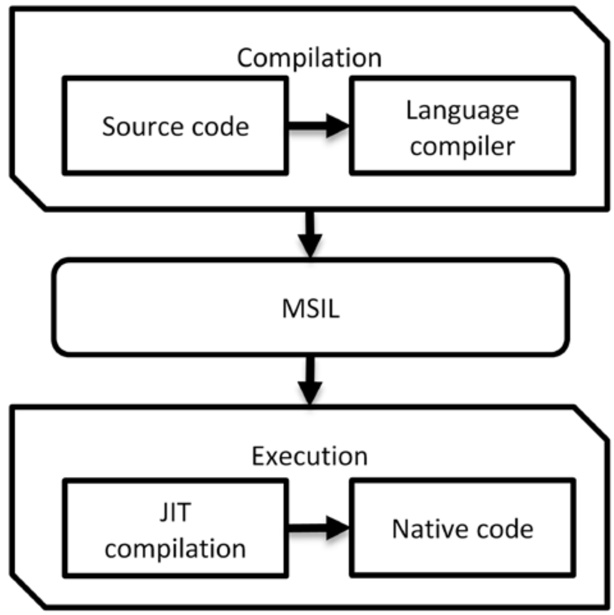

Fig. 1 Principe of .NET code execution

II. Problem description

A Parser engine is a complex system which has specific phases [3] [4] [5] [6] [7]. In general, we can describe the principle of parsing as in Fig. 2.

When we want to implement our own parser system we need to know each component of the parser system. It can be hard to implement it without bugs and implement it to achieve a comparable performance in relation to the native code.

We compared an existing mathematic expression parser and we also compared it to our solution. We chose an alternative approach to implement a mathematic parser system to simplify the complexity of parser engine. There is a similarity between the compilation C# code and the process of expression parsing. Our solution is a string replacement engine based on processing Regex expressions which translate the mathematic expression into C# code. Finally, the .NET dynamic code compilation is used to “revive” this code.

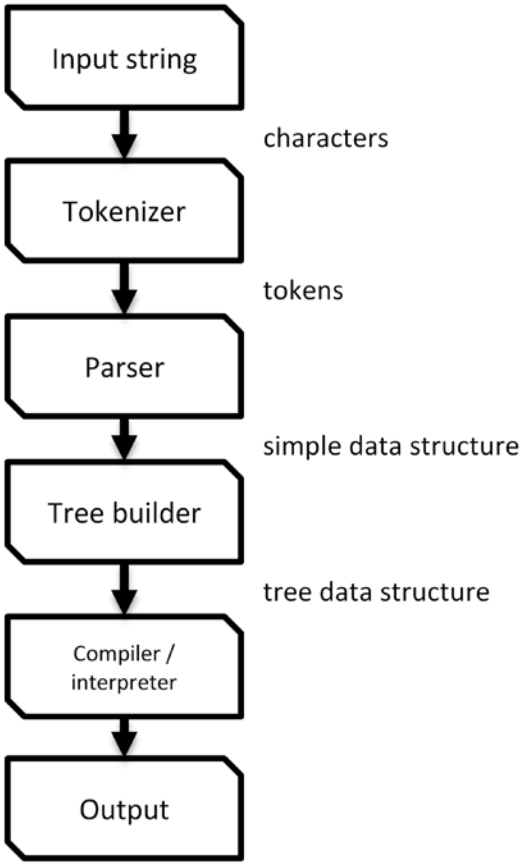

Fig. 2 Principle of parsing [3]

A. *Description of parser libraries*

1) *NCalc*

NCalc is a mathematical expressions evaluator in .NET. NCalc can parse any expression and evaluate the result, including static or dynamic parameters and custom functions. [8]

2) *Sprache.Calc*

This library provides an easy-to-use extensible expression evaluator based on the LinqyCalculator sample. The evaluator supports arithmetic operations, custom functions and parameters. It takes the string representation of an expression and converts it into a structured LINQ expression instance which can easily be compiled to an executable delegate. In contrast to interpreted expression evaluators such as NCalc, compiled expressions perform just as fast as native C# methods. [9]

3) *Flee*

Flee is an expression parser and evaluator for the .NET framework. It allows you to compute the value of string expressions at runtime. It uses a custom compiler, strongly-typed expression language, and a lightweight codegen to compile expressions directly to IL. This means that the expression evaluation can be fast and efficient. [10]

4) *Jace.NET*

Jace.NET is a high performance calculation engine for the .NET platform. It stands for "Just Another Calculation Engine".

Jace.NET can interpret and execute strings containing mathematical formulas. These formulas can rely on variables. If variables are used, the values can be provided for these variables at the execution time of the mathematical formula.

Jace can execute formulas in two modes: in an interpreted mode and in a dynamic compilation mode. If the dynamic compilation mode is used, Jace creates a dynamic method at runtime and generates the necessary MSIL opcodes for native execution of the formula. If the formula is re-executed with other variables, Jace takes the dynamically generated method from its cache. It is recommended to use Jace in the dynamic compilation mode. [11]

5) *Mathos Parser*

Mathos Parser is a mathematical expression parser, built on top of the .NET Framework, which allows you to parse all kinds of mathematical expressions, and in addition, add your own customised functions, operators, and variables. [12]

6) *xFunc*

xFunc is a simple and easy-to-use application that allows you to build mathematical and logical expressions. It is written in C#. This project consists of two libraries and an execution file. The libraries include a code that converts strings into expressions. [13]

7) *muParser*

muParser is an extensible high performance math expression parser library written in C++. It works by transforming a mathematical expression into bytecode and precalculating the constant parts of the expression.

The library was designed with portability in mind and should compile on every standard compliant C++ compiler. There is a wrapper for C and C#. The parser archive contains a ready-to-use project and makefiles files for a variety of platforms. The code runs on both 32 bit and 64 bit architectures and has been tested using Visual Studio 2013 and GCC V4.8.1. Code samples are provided in order to help you understand its usage. The library is open source and distributed under the MIT license. [14]

8) *Expression Evaluator*

Expression Evaluator is a fast-growing, lightweight, simple and free library capable of parsing and compiling simple to medium complexity C# expressions.

Expression Evaluator can take a string that contains C# code, compile it and return the value of the expression, or a function that executes the compiled code. You can also register types or instances of classes to access their properties and methods, essentially allowing you to dynamically interact with those objects at runtime. [15]

9) Dynamic Expresso

Dynamic Expresso is an expression interpreter for simple C# statements. Dynamic Expresso embeds its own parsing logic, and really interprets C# statements by converting it into .NET delegates that can be invoked as any standard delegate. It does not generate assembly but it creates dynamic expressions/delegates on the fly.

By using Dynamic Expresso developers can create scriptable applications and execute .NET codes without compilation. The statements are written using a subset of C# language specifications. Global variables or parameters can be injected and used inside expressions. [16]

B. Dynamic compilation

Our approach is not to make a whole parser engine but instead to try using a kind of hybrid technique.

Our technique can be described like this:

- Take the input string
- Find incompatible tokens and replace it with C# code
- Insert the string into a pre-prepared class
- Use C# feature, dynamic compilation, to compile the code “on-fly”
- Load this compiled class into a current program and load “evaluation” function into the cache

Our approach is trying to achieve maximum performance for evaluating a large amount of data against a small number of functions.

III. Benchmark description

Due to the varied complexity of expressions, we categorized the expressions depending on the complexity of the expressions. There are categories based on expression complexity, in which the complexity is defined by the number of operators, operands and variables:

- Simple expressions – up to 5 operands and 5 operators
- Medium expressions – up to 10 operands and 10 operators, up to 3 function nesting
- Complex expressions – more than 10 operands and operators, more than 3 function nesting

There are two test scenarios for evaluating expressions because there are two main factors that influence the test performance, the expression processing time and expression evaluation time. Let the N is the number of different expressions which are used in the test and M is the number of expression evaluation with given input variables.

The first scenario is focused on measuring the performance of processing different expressions (N >> M). In this case it is the measured time of the evaluation.

The second scenario is focused on measuring the performance of evaluating the same expression against different input variable values (N << M). In this case, the measured time represents the expression processing.

High level scheme of benchmark is described in Figure 3.

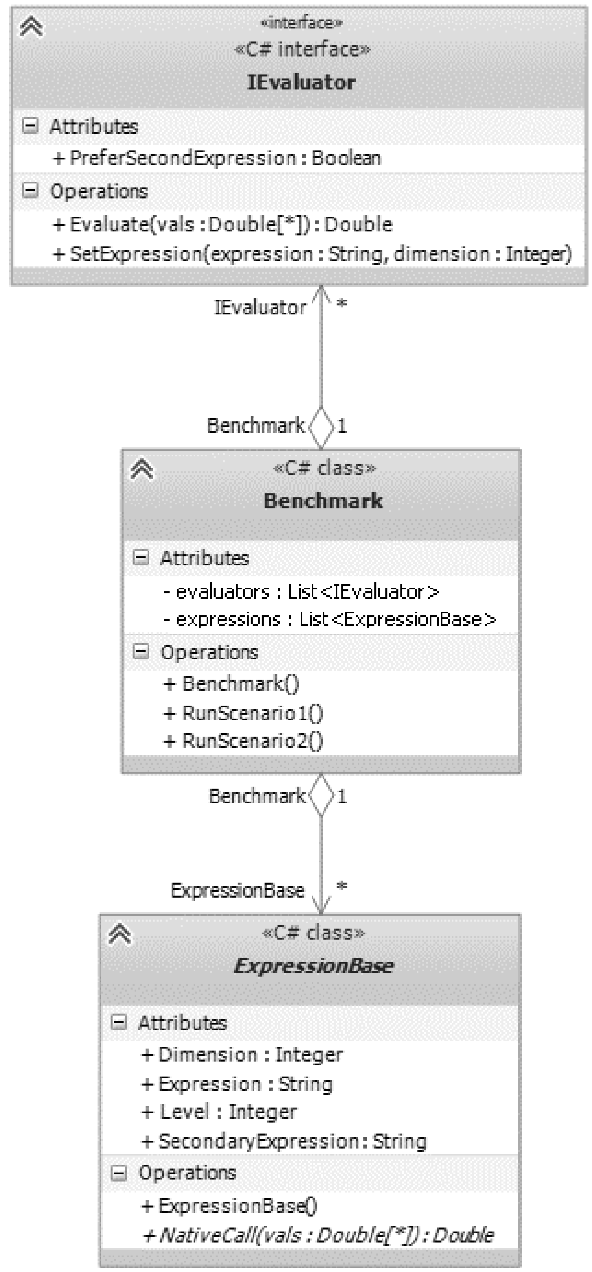

Fig. 3 Benchmark class diagram

Table 1 List of used function in benchmark [17]

Category	Function name	Expression
Simple	Constant	$f = 10 + 750$
Simple	Second constant	$f = 10 + \pi + 2^9$
Simple	Sum	$f(x, y) = x + y$
Simple	Linear	$f(x) = 55x - 150$
Simple	Sphere, n = 2	$f(x) = \sum_{i=1}^{n} x_i^2$
Medium	Quadratic	$f(x, y) = 55x^2 - 150x + 44 + 12y^2 - 22 - 4$
Medium	Rosenbrock, n = 2	$f(x) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$
Medium	Beale's	$f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2$
Medium	Booth's	$f(x, y) = (x + 2y - 7)^2 + (2x + y - 5)^2$
Medium	Bukin N.6	$f(x, y) = 100\sqrt{\lvert y - 0.01x^2 \rvert} + 0.01\lvert x + 10 \rvert$
Medium	Matyas	$f(x, y) = 0.26(x^2 + y^2) - 0.48xy$
Medium	Three-hump	$f(x, y) = 2x^2 - 1.05x^4 + \frac{x^6}{6} + xy + y^2$
Medium	Easom	$f(x, y) = -\cos(x)\cos(y)\exp(-((x - \pi)^2 + (y - \pi)^2))$
Medium	McCormick	$f(x, y) = \sin(x + y) + (x - y)^2 - 1.5x + 2.5y + 1$
Complex	Ackley's	$f(x, y) = -20\exp(-0.2\sqrt{0.5(x^2 + y^2)}) - \exp(0.5(\cos(2\pi x) + \cos(2\pi y))) + 20 + e$
Complex	Goldstein-Price	$f(x, y) = (1 + (x + y + 1)^2(19 - 14x + 3x^2 - 14y + 6xy + 3y^2))$
Complex	Lévi	$f(x, y) = \sin^2(3\pi x) + (x - 1)^2(1 + \sin^2(3\pi y))$
Complex	Cross-in-tray	$f(x, y) = -0.0001(\lvert \sin(x)\sin(y)\exp(\lvert 100 - \frac{\sqrt{x^2 + y^2}}{\pi} \rvert) \rvert + 1)^{0.1}$
Complex	Eggholder	$f(x, y) = -(y + 47)\sin(\sqrt{\lvert y + \frac{x}{2} + 47 \rvert}) - x\sin(\sqrt{\lvert x - (y + 47) \rvert})$
Complex	Hölder table	$f(x, y) = -\lvert \sin(x)\cos(y)\exp(\lvert 1 - \frac{\sqrt{x^2 + y^2}}{\pi} \rvert) \rvert$
Complex	Schaffer N.4	$f(x, y) = 0.5 + \frac{\cos(\sin(\lvert x^2 - y^2 \rvert)) - 0.5}{(1 + 0.001(x^2 + y^2))^2}$

The Main class in benchmark is the *Benchmark* class which contains two test scenarios. Also, this class has a constructor which searches and registers all mathematical parsers (class implementing *IEvaluator* interface) and all testing functions (class derived from *ExpressionBase*) via reflection.

The Interface *IEvaluator* provides the abstraction for mathematic parsers. A parser engine is usually loaded in the constructor and then a parsing string expression into tree structure is carried out in the method *SetExpression*. An evaluating tree structure against input data is carried out in the method *Evaluate*. This class also contains the property *PreferSecondExpression* which indicates which string expression should be used for the current parser. This is because some parsers do not allow define operators like '^' power or '%' modulo and those formulas which have these operators must be rewritten without using these operators and stored as *SecondaryExpression* in the Expression class.

The Abstract class *ExpressionBase* provides a base class for all tested mathematical formulas. It provides the following attributes:

- *Dimension* – the number of input arguments for a function
- *Expression* – the string expression of a mathematical formula
- *Level* – the number which indicates the complexity of an expression
- *SecondaryExpression* – the string expression of a mathematical formula which has replaced any incompatible operators

In addition, this contains the method *NativeCall* which contains mathematic formulas expressed in C# code.

IV. Result

We compiled our test program under the .NET 4.5 platform, “Any CPU” platform setting and release configuration. We ran it on a laptop with Intel i7 3517 CPU, 10 GB RAM, SSD disk with Windows 8.1 Pro.

For the first scenario we used 24 different expressions as shown in table 1 and each has been evaluated 1 000 000 times. The evaluation time of the measured functions has been summarized for each category and divided by the total number of functions in the category.

Table 2 Result for scenario 1 in milliseconds

Library name	**Complexity of expressions**		
	Simple	**Medium**	**Complex**
Native	32	146	162
D. Expresso	51	170	213
muParser	89	244	360
Dynamic	93	202	220
Sprache	610	892	855
EE	838	1149	1184
Flee	875	1339	1374
Jace	1553	2415	3137
NCalc	2105	4565	6561
Xfunc	2297	5425	5158
Mathos	11311	29053	FAILED

In the results table 2, our developed test solution is called “dynamic” and its function evaluation performance is the best of all libraries for function evaluating. However, it must be taken into account that our approach has a relative high starting overheat because of a compilation time of about 50 milliseconds. If a simple function and a small amount of evaluation is used, our approach cannot currently be faster than 50 milliseconds due to the compilation time overheat.

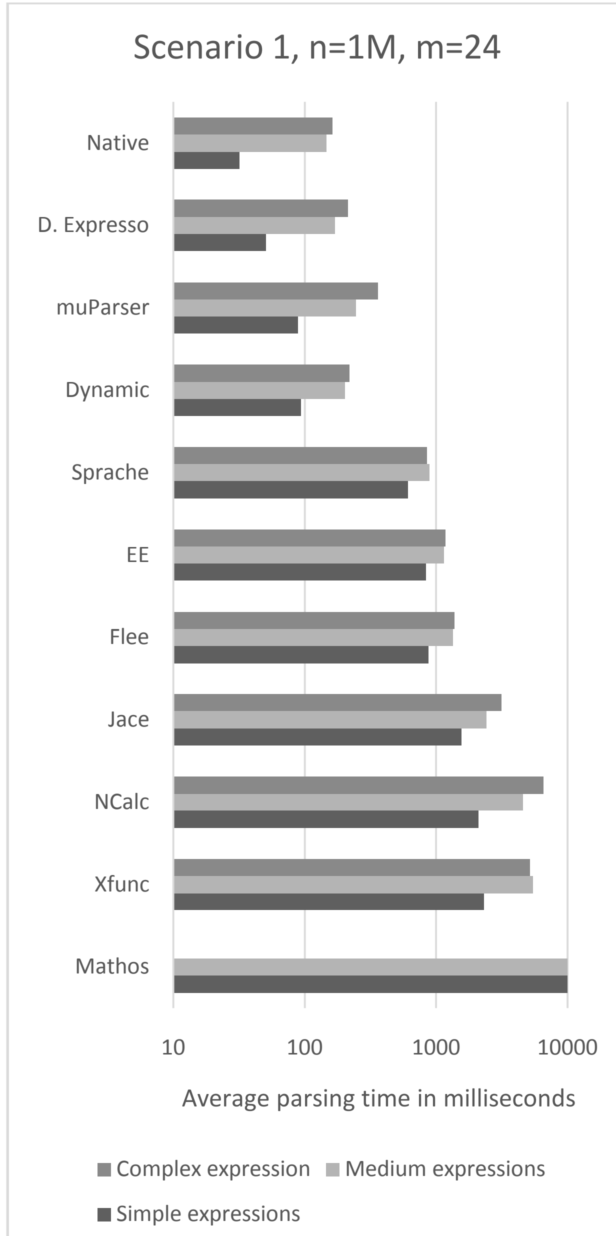

Fig. 4 Benchmark result for scenario 1

In the second scenario, the 1000 function was used (the function set was created by random choice from 24 function sets as shown in table 1). Each of these functions was evaluated only once.

Table 3 Result for scenario 2 in milliseconds

Library name	Complexity of expressions		
	Simple	Medium	Complex
Native	0	0	1
Mathos	41	71	FAILED
Xfunc	93	199	247
NCalc	129	186	263
muParser	145	215	283
Sprache	801	1822	2527
Jace	910	2004	2480
Flee	2536	2961	3269
EE	4526	9425	13441
D. Expresso	5532	5258	6699
Dynamic	48605	51703	52713

In the results table 3, our library is also called “dynamic” and we can see our approach has the worst result against the other libraries. This bad result is due to the .NET compilation time overheat.

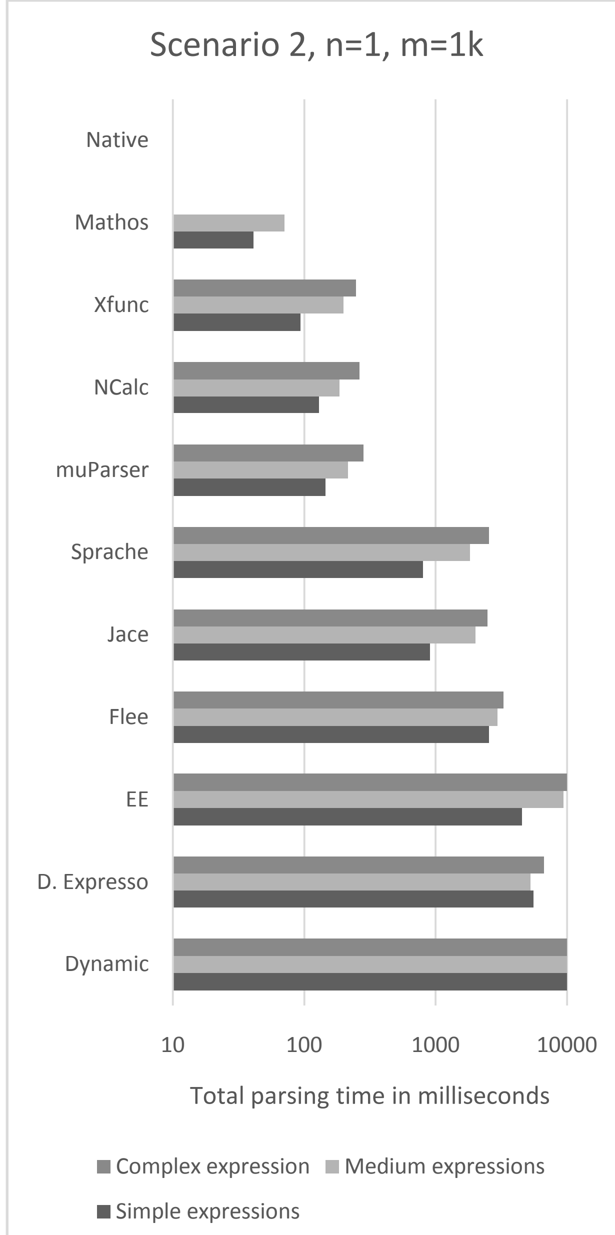

Fig. 5 Benchmark result for scenario 2

V. Improvements

Dynamic compilation is significantly penalised by a bug in the current C# class which is responsible for dynamic compilation. This bug causes an impossibility to create an in-memory assembly [18].

Currently there is only one workaround and it is to switch from the old process of dynamic compilation to the new compilation service “Roslyn”.

The .NET Compiler Platform, better known by its codename "Roslyn", is a set of open-source compilers and code analysis APIs for C# and Visual Basic.NET languages from Microsoft.

The project notably includes self-hosting versions of the C# and VB.NET compilers – compilers written in the languages themselves. The compilers are available via the traditional command-line programs but also as APIs available natively from within .NET code. Roslyn exposes modules for a syntactic (lexical) analysis of codes, semantic analysis, dynamic compilation to CIL, and code emission. [19]

Currently there is only a preview version of Roslyn but it should be usable for our requirements.

We rewrote our codes using this new Roslyn API and compared the results.

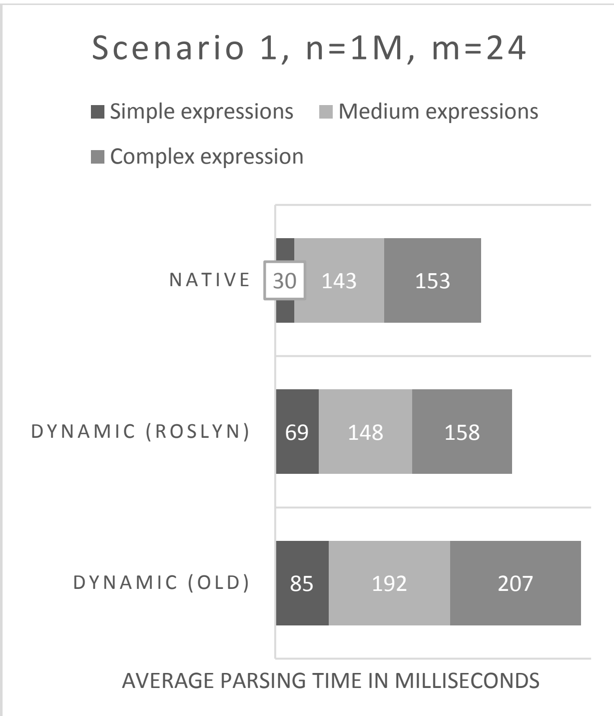

Fig. 6 Benchmark result for scenario 1 with Roslyn

For scenario 1 we can see significant improvements and in the expression category medium and complex we got overhead less than 5% over the native code.

As we can see there is performance drop when we was measuring dynamic compilation of simple expressions with Roslyn. This is because the first time we call the Roslyn compilation service, it must load necessary assemblies in memory.

We are expecting that in the full release of Roslyn this performance drop will be reduced.

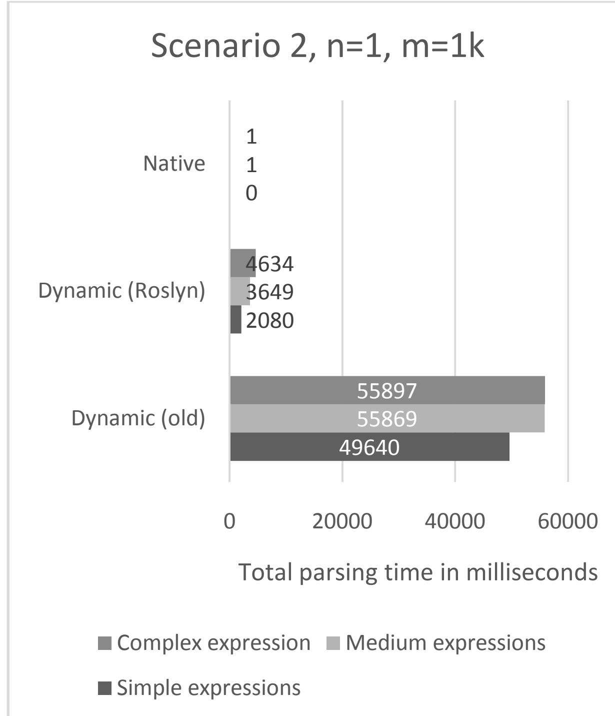

Fig. 7 Benchmark result for scenario 2 with Roslyn

As in scenario 1 there are also significant improvements against the original implementation with the old C# compiler. However, compared to other mathematical parsers there is still low performance. In respect to the way a dynamic compilation works, it will be very difficult or maybe even impossible to achieve a better result than the current one. It is because, now, one dynamic compilation takes about 3 milliseconds, in which the following tasks are included:

- loading the assemblies required for compilation
- creating the compilation service
- parsing the C# code
- compiling the code
- loading the dynamic generated library

VI. Conclusion

The main aim of this work was to investigate how we can use a dynamic compilation for the implementation a very fast mathematical expression parser/evaluator engine. We undertook an investigation into existing mathematical parsers and focused on how they work with parsing mathematical formulas.

As result of our investigation we are able to create an alternative approach to existing mathematical parsers which is based on a dynamic compilation.

We also created benchmarks to measure how efficient our approach is against existing mathematical parsers. The results from those benchmarks were great even though there is a bug in the C# compiler which did not allow us to create only in the memory assembly and we were penalised because of this.

By using the new unfinished Roslyn compiler services we were able to get even more impressive results which ranked us among one of the best mathematical parser engines.

The results of this work were used in heat load modelling. Users can use this application as an excel add-in and the application is built using c# language and the .net framework. [20]

VII. Future work

For our next research we plan to mainly focus on these points.

A. User specific functions

So far we have created a closed system which does not allow users to define user specific functions. This is big limitation for wider application.

B. Use Roslyn compiler

We are also looking forward to the stable release of the C# compiler 'Roslyn' which has a significantly faster dynamic code compilation as we have proven in our benchmarks. Currently Roslyn is in the beta phase and could have some bugs so it is necessary to wait for its full release.

C. Use optimization service

We are also planning to implement an optimization service for mathematic formulas which will modify and simplify mathematic expressions to achieve some more performance.

Because we are investigating using Wofram Mathematica as an optimization service, we are also considering implementing the application as web service using Service Oriented Architecture. [21]

References

[1] M. P. Radha Thangaraj, "Differential Evolution Algorithm for Solving Multi-objective Optimization Problems," in Proceedings of the 7th International Conference on Applied Mathematics, Simulation, Modelling (ASM '13), 2013.

[2] R. A. Rahmat, "Application of Genetic Algorithm in Optimizing Traffic Control," in Proceedings of the 7th International Conference on Applied Mathematics, Simulation, Modelling (ASM '13), 2013.

[3] Compilers Principles Techniques and Tools (2nd Edition), Boston: Pearson Education, Inc, 2007.

[4] J. A. Farrell, "Compiler Basics," 1995. [Online]. Available: http://www.cs.man.ac.uk/~pjj/farrell/compmain.html.

[5] K. MAEDA, "Quick Parser Development Using Modified Compilers and Generated Syntax Rules," in Recent Advances in Applied Mathematics and Computational and Information Sciences - Volume II, Houston, 2009.

[6] P. M. RICCHETTI, "A Practical Method for the Implementation of Syntactic Parsers," in Proceedings of the 6th WSEAS international conference on Automation & information, Argentina, 2005.

[7] S. I. SOLODUSHKIN, "Access to the Solver of Functional Differential Equations Through," in Recent Advances in Applied & Biomedical Informatics and Computational Engineering in Systems Applications, Florence, 2011.

[8] "NCalc - Mathematical Expressions Evaluator for .NET," 2014. [Online]. Available: https://ncalc.codeplex.com/.

[9] "Sprache.Calc," 2014. [Online]. Available: https://github.com/yallie/Sprache.Calc.

[10] "Fast Lightweight Expression Evaluator," 2014. [Online]. Available: http://flee.codeplex.com/.

[11] "Jace.NET," 2014. [Online]. Available: https://github.com/pieterderycke/Jace.

[12] "Mathos Parser," 2014. [Online]. Available: http://mathosparser.codeplex.com/.

[13] "xFunc," 2014. [Online]. Available: http://xfunc.codeplex.com/.

[14] "muparser - Fast Math Parser Library," 2014. [Online]. Available: http://muparser.beltoforion.de/.

[15] "C# Expression Evaluator," 2014. [Online]. Available: https://csharpeval.codeplex.com.

[16] "Dynamic Expresso," 2014. [Online]. Available: https://github.com/davideicardi/DynamicExpresso.

[17] S. F. University, "Test Functions and Datasets," 2014. [Online]. Available: http://www.sfu.ca/~ssurjano/optimization.html.

[18] Microsoft, "MSDN Forum," 2007. [Online]. Available: https://social.msdn.microsoft.com/Forums/en-US/8e839652-894d-4891-911b-621c89f1c7f7/compilerparametersgenerateinmemory-property-doesnt-really-work?forum=netfxbcl.

[19] Microsoft, ".NET Compiler Platform ("Roslyn")," 2014. [Online]. Available: https://roslyn.codeplex.com/wikipage?title=Overview&referringTitle=Documentation#INTRODUCTION.

[20] P. Č. Erik Král, "A Comparison of the Performance of a Composite Pattern and a Mathematic Expression Parser and Interpreter," in Recent Advances in Applied Mathematics, Modelling and Simulation, Florence, 2014.

[21] S. KAMBHAMPATY, "Service Oriented Architecture for Enterprise Applications," in Proceedings of the 5th WSEAS Int. Conf. on Software Engineering, Madrid, 2006.

Petr Čápek is currently a Ph.D. student at The Department of Informatics and Artifical Intelligence at Tomas Bata University in Zlin. His master's thesis was focused on the State-of-the-Art Methods for Designing Native Multiplatform Mobile Applications. Hi is interested in modern software architectures and evolution algorithms.

Erik Král, Ph.D. is a senior lecturer at The Department of Computer and Communication Systems at The Tomas Bata University in Zlin. Between 2003 and 2006 he worked as a software developer (MS Navision DB, CRM system, .NET, c#). Between 2006 and 2011 he worked as a researcher on the National Research Program II, The intelligent system controlling an energetic framework of an urban agglomeration (successfully finished in 2011).