VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS INSTITUTAS
PROGRAMŲ SISTEMŲ BAKALAURO STUDIJŲ PROGRAMA

Translation Unit Granularization

Kompiliatoriaus transliuojamo vieneto smulkinimas

Bakalauro baigiamasis darbas

Atliko:	Andrius Bentkus	(parašas)
Darbo vadovas:	asist. dr. Vytautas Valaitis	(parašas)
Darbo recenzentas:	Karolis Uosis, Lekt.	(parašas)

Vilnius – 2021

Summary

This thesis presents an approach on how to speed up recompilation in a day to day programming scenario, benchmarks the performance improvements with various inputs and compares the gathered performance metrics to an already existing compiler.

Classic compilers treat a source file as an atomic translation unit in their compilation pipeline, the proposed approach tries to leverage the structure of modern programming languages to granularize the translation unit in order to improve recompilation times.

The chosen target language is a subset of the Scala programming language. Instead of modifying the standard Scala compiler to accommodate the proposed approach, a compiler is written from scratch in order to have a simple program enabling straightforward modifications for an easier accommodation of the proposed approach.

Performance measurements of the prototype compiler are taken with the approach enabled and disabled, various dynamic inputs are generated with different translation unit sizes to gain insights in the effectiveness of the approach. The results show an improvement of 5% to 30% depending on the complexity of the input source.

Keywords: Compilers, Translation Unit, Compiler optimization

Santrauka

Šiame darbe pristatomas metodas kompilerio transliavimo našumui pagerinti, kai identiškas išeities tekstas yra transliuojamas pakartotinai. Metodo našumo patobulinimas yra pamatuojamas skirtingiems išeities teksto variantams. Taip pat našumas palyginamas su egzistuojančio kompilerio našumu.

Klasikiniai kompileriai išeities tekstą vienoje byloje traktuoja kaip nedalomą transliuojamą vienetą. Pristatomas metodas bando pasinaudoti modernios programavimo kalbos struktūra ir susmulkinti transliuojamą vienetą į mažesnius bendram transliavimo našumui pagerinti.

Pasirinkta kompilerio kalba yra Scala. Metodui sparčiam įgyvendinimui yra sukūriamas naujas kompileris nepernaudojant egzistuojančio kompilerio funkcionalumo. Naujo kompilerio architektūra nuo pradžių yra pritaikoma metodui įgyvendinti. Dėl sumažinto sudėtingumo metodas yra pritaikomas lengviau.

Sukurtam kompileriui atliekami įvairūs matavimai pritaikius bei nepritaikius naują metodą. Geresniam greitaveikos pokyčio įvertinimui sukūriamas skirtingo dydžio ir sudėtingumo išeities tekstas. Matavimo rezultatai rodo, jog priklausomai nuo išeities teksto sudėtingumo transliavimo greitaveika pagerėja nuo 5% iki 30%.

Raktiniai žodžiai: Transliavimas, Transliavimo optimizacijos

CONTENTS

Introduction

The majority of the research done on compilers is focused on making compiler output optimized programs in terms of code execution speed and memory usage [LR18] while neglecting or willfully sacrificing [Cra19] actual runtime performance of the compiler itself. As computers get faster more resources are available, but programmers tend to utilize these newfound resources to make the implementation of programs simpler rather than making the programs faster [Wir95].

Another vector of academic improvement within the compiler sphere is to add new features or utilize paradigms like dependent types or formal specifications [AMP+18] in order to allow the compiler to do more sophisticated type checking and verification than classic type checking allows while significantly increasing runtime compilation .

When assessing the adoption of a new programming language the runtime speed of the available compilers are often gauged as an important metric [San21] among others such as language complexity, available libraries and spread of ecosystem. Dealing with enormous compilation times of huge projects can have negative effects on the productivity of programmers and even drive away people from an entire ecosystem [Hal19; San21].

Modern organizations moving towards continuous delivery practices called DevOps [Ebe16] experience an ever increasing need to run the compilations over and over within their continuous integration pipeline. Slow compilation speeds go against the main goal of the short cycle times promoted in DevOps and has spurred creation and adoption of minimal programming language focused on compilation speed and productivity to counter the effects of giant code bases on the CI process [Mey14; Pik12].

Expected results

This thesis will analyze the effects of translation unit granularization in a compiler on the recompilation speed. The compilers language will be a subset of the Scala programming language.

It will be achieved by the following tasks:

1. Create a prototype compiler.
2. Implement compilation unit granularization in the prototype compiler.
3. Validate correctness for the prototype compiler.
4. Measure the performance impact of the implemented approach.
5. Compare it to an existing compiler.

The expected results will be a compiler with compilation unit granularization and various performance measurements together with a comparison to an existing compiler.

Challenges

The goal of this bachelor thesis is to create a prototype compiler for a small subset of the Scala programming language. Since writing a compiler is a difficult task in itself and Scala is known to have a multitude of advanced and complicated features [Ode19], the prototype compiler implementation in this thesis will support only basic features in order to allow for an easier realization of the proposed augmentation to the compiler.

When writing software developers tend to create only small changes in existing source files before running the compilation again for a quick feedback loop of introduced errors. A translation unit is the entire input of source code that is used to produce a compilation output and is usually reprocessed in its entirety without consideration of previous outputs compilation outputs, hence even small changes in a giant class requires a complete recompilation of the entire class which might be costly if the codebase is large.

The core idea of the thesis is to granularize the translation unit size order to increase performance of subsequent compilations of the same source file with minimal source modifications to allow for an measurable speed improvement in this kind of feedback loop.

Such granularization requires additional compiler logic and might overcomplicate an already sophisticated computer program. Additional challenges might arise when ensuring compilation output correctness and preserving compilation output similarity. The lack of complex compiler features in the prototype might skew the results against the advantages of granularization since complex compiler features incur the most significant compilation runtime penalties.

Investigation method

Time measurements in the millisecond range will be taken of subsequent compilation runs changing only parts of the source code using the prototype compiler written in this thesis with the translation unit granularization feature enabled and disabled.

Various source code input sizes and complexity of source code will be evaluated. A study of a classic looking case of average code complexity will be chosen, emphasized, discussed and evaluated.

An automated but definitive and consistent source code generation tool will be used to create consistently reproducible results for the given inputs listed in the following list.

1. Translation Unit

The classic approach of compilers is to read an entire source file and generate a corresponding output file with the translated code.

For example in C++ and C source files with the extensions .cpp or .c are compiled to an intermediate object file with the extension .o, containing the generated assembly code of a specific target architecture, x86, ARM, etc. Once all source files are compiled into object files, all object files are combined together with optional resources into a singular output file called the executable.

1.1. File scope

The Java language forces the programmer to define every Java class in its own file and every .java file is then compiled to a .class file, which can be later packaged to .jar files together with additional resources for easier deployment. A .class file contains all the Java bytecode generated from the Java source file, a condensed representation of the a Java class in a high level format which is abstract and does not target any specific architecture, but an abstract architecture called the Java Virtual Machine (JVM). Code targeting the JVM can be then executed provided a Java Runtime Environment (JRE) is present on the target machine which Just in Time (JIT) compiles the bytecode to the target machines assembly code and executes it. Java bytecode is much more abstract and conceptually closer to the original representation of a class than the C++ compiler generated object files, which contain directly executable machine code. Since the translation unit in Java is an entire file and every Java class needs to be defined in its own file, naturally the translation unit size in Java is also class scoped.

1.2. Class scope

A Scala source file can contain multiple class definitions in a singular file. Because it is possible to define a class in a very concise manner, Scala source files tend to contain more than one class as shown in listing 1.

Listing 1. Example of multiple classes in a single Scala source file

```
1 sealed trait Expression
2 class Int32(val value: Int) extends Expression
3 class Sum(val left: Expression, val right: Expression) extends Expression
4 class Multiply(val left: Expression, val right: Expression) extends Expression
```

The compiler creates in total 4 corresponding .class files as depicted in figure 1. The classes all depend on the same trait definition called Expression, an empty interface. A similar concept exists in Java and is called a Marker Interface [Blo08]. There is no need for a strict compilation order, the compiler can utilize the meta information to compile all result .class files in parallel or whatever order it sees fit. When running the compiler with the source file displayed in listing 1 as the input, 4 .class files will be created.

A compiler might check if the input file was already compiled by memoizing the a corresponding hash value. However, cosmetic changes or reordering the classes within the source file will still trigger a complete recompilation, even though the output will be identical.

In the given examples the classes are trivial and the performance when recompiling is negligible. However, working in giant codebases consisting of thousands and thousands of source files which result in multiple class files, the total amount of file operations needed to create all resulting files add up.

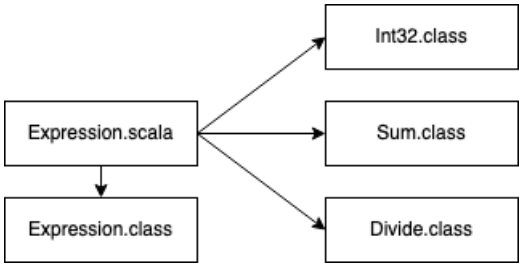


Figure 1. Compilation result of Listing 1

One might argue that multiple sizable class implementations with numerous methods contained in a singular source file are rare in actual code bases and optimizing for this case would yield minuscule benefits outside of tailored benchmarks, but the opposite is the truth. In the following example a typical usage of a popular Scala unit testing library Specs2 [Tor21] is shown.

Listing 2. Standard usage of Specs2

```
1  import org.specs2.mock.Mockito
2  import org.specs2.mutable.SpecificationWithJUnit
3  import org.specs2.specification.Scope
4
5  class SampleTest extends SpecificationWithJUnit with Mockito {
6    "test group" should {
7      "first test" in new Context {
8        one must_=== 1
9      }
10
11     "second test" in new Context {
12       (one + one) must_=== 2
13     }
14   }
15
16   trait Context extends Scope {
17     val one = 1
18   }
19 }
```

Scala developers tend to use use many language features to create a Domain Specific Language (DSL) for visually pleasing code. Because many Scala compiler feature are in play in this example, the details are quite intricate, but the result is that for each test case shown in

the example 2 (first test, second test) a stand alone .class compilation output is created (SampleTest$$annon$1.class, $SampleTest$$anon$2.class). Having many unit tests in a sophisticated codebase can therefore increase the compilation times significantly, to a point where an iterative development process becomes impractical. Compilation times longer than a minute can be easily surpassed when meticulously writing unit tests in a Test-driven development (TDD) approach.

TDD also promotes writing tests as a playground to explore approaches and test code feasibility before writing the implementation. Long compilation times of unit tests make such an approach also impractical.

If an action takes more than 10 seconds the user is likely to loose attention and starts to perform different tasks [Nie94]. The norm for compilers it to not show any kind of progress when compiling a single translation unit. This exacerbates the perceived time it takes to complete a process for the user [Nie94].

1.3. Granularization

As such the main idea of the thesis arises. Instead of conceptually treating an entire file as single translation unit which needs to be recompiled in its entirety producing multiple output files, all the objects declared in a source file get treated as as distinct translation units.

It is a natural fit, every class declared in a the source already produces its own output, which is independent from other produced outputs. Thanks to the language structure of modern programming languages, most of the top level constructs within the source files are classes. Once a file has been loaded and parsed, all top level objects can be processed separately. Separation enables the ability to process the translation units in parallel, the ability to skip already processed translation units. In other words, it enables us to granularize the giant translation unit into smaller translation units.

2. Compiler

At the start of the bachelor thesis a thorough investigation was made to analyze how the official Scala compiler works [Ode21] and if it is feasible to apply the targeted optimizations to it. The investigation took very long and yielded little results, the codebase was too humongous and complicated to understand, the compiler too feature packed to adjust with the target objectives. Not being proficient enough in the codebase of the official Scala compiler and taking the scope of the bachelor thesis into account, it was deemed to complicated to undergo such a task.

Instead of adjusting an already existing compiler, a compiler was written from scratch with an iterative approach. In the following numerous advantages and disadvantages of this method are expressed.

Flexibility - mature compilers are harder to change, they have accumulated a lot of code which needs to be adjusted for every modification. The official Scala compiler has a complex pipeline with multiple passes and figuring out where what the best point of incision is a difficult task. A small compiler written from scratch allows the author to focus on the most important aspects.

Correctness - ensuring that a new feature is correctly implemented in a sizable code base is an immense undertaking. In order to ensure that a new feature addition doesn't impact the overall behavior of the compiler every feature needs to be tested for correctness. The multi-pass nature of the official Scala compiler makes reasoning about correctness difficult, because it is a non trivial undertaking to reason about how caching in one pass might affect functionality in subsequent passes.

Simplicity and transparency - a small compiler is simple and easy to reason about. Writing one from scratch means that a complete understanding of the compiler exists. New features are easier to add because of a complete understanding of the entire code base. Debugging is much more pleasant since the number of affected features is smaller.

Accuracy - the results might be skewed since the self written compiler implements only a handful of features of the official language standard. This can go both ways.

Some elaborate Scala features like type-driven implicits and mixing multiple traits are computationally very intensive. The former needs to recurs in a nondeterministic way on distinct implicit types in order to find a correct type inheritance path while the latter creates a proxy method in a target classes for every method within a mixed in non trivial trait. Being able to reuse computation instead of redoing these kind of workloads might add significant performance boosts.

On the other hand, adding a multitude of features might slow down general compilation speeds of the compiler. Features that are not used should not add a significant slow down to execution speed, but code bases are rarely ideal.

2.1. Language

The target compiler programming language is Scala. Scala mixes object oriented programming (OOP) and functional programming paradigms to create a feature rich environment for developers. Functions in Scala are values and all values are objects, a smart way to seamlessly integrate

both paradigms in a potent mix.

OOP has become quite ubiquitous and is a base line feature that most of programming languages have. Many developers are proficient in at least one programming language supporting OOP and the most popular programming languages utilize OOP very prominently.

The language aims to be concise and aesthetically pleasing utilizing features such as comprehensive type inference to minimize verbosity. Functional aspects are prominently represented with a lightweight syntax for anonymous functions, higher order functions, currying, lazy evaluation, Algebraic Data Types (ADT), higher order functions, pattern matching.

Scala is a programming language targeting the Java Virtual Machine (JVM). Running on the JVM allows Scala easily to utilize existing Java code, which is a tremendous advantage because utilizing existing Java code opens access to a plethora of well maintained libraries targeting a wide range of needs. The opposite however is not always true since Scala has a lot of features which can not be directly translated to JVM bytecode which is the underlying instruction set powering the JVM.

2.2. Implementation

Creating a compiler from scratch for a non trivial small language is already a significant task. Programming languages have become quite complex containing multiple paradigms. Scala is also a language with a staggering amount of features that increases the overall complexity of a compiler. Each language feature on its on might look simple and be easy to understand but putting another cog in an already complex machine can prove to be difficult.

Not only does Scala have a multitude of features but some of the features are quite complex like macros, implicit type conversion and summoning, pattern matching, higher kinded types and type lambdas allowing a user of the language to implement language constructs [Sab09; Sab21] which are challenging to understand even for experienced Scala developers and more so difficult to implement.

Instead of writing the entire compiler from scratch at once containing all features a more conservative approach is taken. The approach starts out with the most simplistic compiler imaginable, compiling a bare minimum example of the target programming language, a simple ”Hello World” program (listing 3).

Listing 3. ”Hello World” in Scala

```
1 object Main {
2   def main(args: Array[String]): Unit = {
3     println("Hello World!")
4   }
5 }
```

Identifier resolution is also omitted in the minimal first implementation, the function identifier ”println” definition is hard-coded in the compiler at first instead of creating an identifier resolution mechanism which would include all identifiers from Predef$ in every compilation unit [EPF02]. Afterwards every feature that is needed is for the thesis is implemented keeping the set of features

concise while allowing to prove the thesis goal. This allows us to deal with the complexity of Scala on a per demand basis, only features that are required are implemented incrementally to keep overall complexity down [Ghu06].

The compiler follows a classic compiler design with a frontend, the parser, an intermediate representation and the backend, code generation targeting the JVM and Java byte code.

2.2.1. Frontend

The frontend starts with a classical lexical analysis step. The entire content of a file is loaded at once into memory to avoid dealing with continuous invocations of slow kernel file system application programming interface (API) calls. Characters are grouped into tokens and a stream of tokens is produced with special meaning attached to each token in accordance with the Scala language specification [Ode19]. Keywords of the language have special tokens to represent them and numbers and identifiers generic ones.

This is followed by a parser which transforms the sequence of tokens into an abstract syntax tree (AST). The chosen approach is heavily inspired by a functional programming approach called "Monadic Parsing" [HM98].

Monadic parsing is a parsing approach where monadic properties of data structures are being heavily utilized to create a parsing mechanism that has a rather simple and concise data structure it operates on, the parser. A parser is a transformation of a token stream to a list of results, with each result containing the parser product and a tail token stream.

Simple parsers can be easily defined on their own (listing. 4)

Listing 4. Parsing a token

```
1 val number: Parser[Expr] = for {
2   number <- token(sat[Number])
3 } yield Num(number.value)
```

Listing 5. Parsing an import

```
1 val identifier = token(sat[Identifier])
2 val fullIdentifier = sepBy1(identifier, '.')
3 val `import`: Parser[Import] = for {
4   _ <- identifierWithName("import")
5   name <- fullIdentifier
6   fullname = name.map(_.value).mkString(".")
7 } yield Import(fullname)
```

Complex parsers can be easily created by combining simple parsers (listing. 5). Utilizing monad composition and Scala's for comprehension the solution is terse and compact. A classic functional programming approach to deal with complexity also widely known as a bottom-up approach.

Scala's functional programming community is thriving and many libraries exist which can be used to aid in the handling of monads. Scalaz [sca02] was used in the implementation of the parser, it had the most candid interface for mondas to implement.

2.2.2. Backend

The compiler targets the JVM just like the official Scala compiler and therefore the output of the backed are instructions for the JVM called Java bytecode.

Compared to other computer architectures the Java bytecode is a fairly high level representation mirroring the functionality of Java closely. As such it has a an explicit notion of classes and groups functionality of classes in a singular file the class file [LYB+13].

Other machine architectures such as x86-64, ARM and Sparc do not have a notion of classes with fields and methods or references to other classes. Compilers directly compiling to these kind of architectures need to map their high-level language constructs to a much more simplistic instruction set, which has only a concept of registers and a linear memory model. This requires a lot of additional that a compiler should do, but in our case the JVM takes care of a lot of this functionality when JITing.

The backend translates the frontend generated AST to an intermediate representation (IR) which is closely modeled after the class file format. Afterwards it serializes the IR to the class file format, which can be executed by the JVM.

The class file format is split into a body and header part. The header parts contains integer, string constants, method and field references. The body part contains the structure of the class. In order to save bytes and to avoid repeating constant definitions, the body references values in the header.

This makes the body depended and makes patching of the file hard. Adding an additional constant in the header means the the entire body needs to be shifted, which is an expensive operation. The main value in the IR is to avoid this split and encode all the constants directly in the IR structure, making each sub expression of the IR self contained and easily transferable.

Class files are designed to be standalone and independent. All the information needed is contained in the class. References to other methods and fields from other classes are a fully qualified identifiers. This property is key for translation unit granularization.

2.3. Translation unit memoization

Computers have a limited amount of resources available, mainly the CPU and RAM get utilized when running programs like compilers. The execution of speed gets determined by how many instructions the application needs to execute in order fulfill its goal. The execution speed of instructions is also variable, as some instructions can be executed faster or accept more data to be processed at once.

Programming language abstractions add more instructions to the final executable that need to be executed. One way to optimize for performance is to remove these abstractions and use lower level abstractions, which are closer to the machine. Achieve the same goal with less.

Another way is to completely avoid the execution of instructions, reuse work that was already done. Do not work if it the work has already done. This solution is used in the compiler on all top level AST object within a source file.

Before translating the AST, a cache is queried with the list of all hash codes of already compiled AST objects. If the object already exists, no work needs to be done, otherwise the compiler pipeline is invoked as usual. Once the compiler has finished, the cache is replaced with all the new hashes.

3. Benchmarks

In order to measure the efficiency of the proposed granularization technique, multiple measurements have been taken. Measurements were taken of the prototype compiler with and without caching of the translation units enabled and then also a measurement was taken of the official Scala compiler on the same input.

In order to measure the time difference between a cached compilation and a full compilation the cache file containing the hash codes of the already compiled translation units is simply deleted triggering a complete run of the compiler on the next invocation.

In short the sequence during benchmark looked like this:

1. Delete the cache
2. Run the official Scala compiler
3. Run the prototype compiler, which generates the cache
4. Run the prototype compiler, which utilizes the cache of the previous run

The measurements taken can be found in the table 1 in the appendix. The column Official corresponds to the time in milliseconds. The column New is the measurement taken of the prototype compiler without caching, the column Cache is the measurement taken of the prototype compiler with caching enabled. Diff is the difference between New and Cache.

3.1. Measurements

During the compilation runs a general time measurement was taken measuring the entire time it took the compiler from start to finish, labeled ”full” in the following. Additional measurements were taken of the prototype compiler of the following compilation steps:

1. read - reading the source file
2. lex - lexing the input into lexemes
3. parse - parsing the stream of tokens into an ast
4. ast - the transformation step which converts the ast into a high level class file representation
5. class - serializing the higher level class file representation into the binary class file format
6. write - writing the serialized class file from memory files

The table 1 in the appendix contains measurements of all the phases described. Since the caching only affects the ast, class and write part, the results for the cached run was written down in a more terse way in the table. When looking at an entry 100/0, the first number is the time it took for the compiler without caching enabled for that phase and the second number is the result with caching enabled.

These measurements were taken only for the prototype compiler. The official compiler has a much more complex compilation pipeline consisting of more than 12 steps instead of singular ast step in the prototype compiler. Its architecture is also noticeably different, utilizing the same caching technique might not be easy to apply at the same layers. The additional metrics would add little value and would make the benchmarking results harder to read without actually implementing the caching mechanism on in the official compiler.

3.2. Hard to measure metrics

Combining all single compilation pipeline step measurements of the prototype compiler does not add up to the single measurement metric ”full” of the entire application. This is caused by the JVM JIT startup costs and and variable GC interruption time. The JVM needs to load the jars, the class files of a jar, run in interpreted mode and then JIT the class files into executable assembly code. Also the GC can happen unpredictably at any moment between measurements which can add.

When analyzing the collected metrics one should keep in mind the additional overhead that the JVM can have.

3.3. Input variance

In order to measure variance in code complexity, variable input had to be created. The dimensions of the created classes were twofold:

1. Classes - the number of classes a source file contains
2. Methods - the number methods each classes has

The generated input files have the following format:

Listing 6. Generated source inputs

```
1  object A0 {
2    def main(args: Array[String]): Unit = {
3      println("Hello World 1")
4    }
5
6    def main2(args: Array[String]): Unit = {
7      println("Hello World 2")
8    }
9    /*
10     ...
11
12     def mainN(args: Array[String]): Unit = {
13       println("Hello World N")
14     }
15   */
```

```
16 }
17
18 object A1 { /* ... */ }
19 /*
20     ...
21 object AN { ... }
22 */
```

The number of translation units within a source file is controlled by the number of classes. The complexity of each translation unit is controlled by the number of methods each class contains.

The methods are kept very simple, singular expressions containing a basic function call. Even with trivial methods which do not take much computation to compile down a significant uplift in performance is achieved, presented in the later subsection.

To simulate various workloads different inputs where chosen ranging from 1 class to 100 in an input source file and from 0 to 100 methods.

3.4. Discussion

Improvements from 20ms to 480ms can be observed during recompilation (figure 2). This corresponds to an increase of 5% and 30% when considering the total run time of the compiler (figure 3).

20ms is a value so small that it is hard to notice for the user of a compiler. However, shaving off 322ms in a complex translation unit with 50 classes and 50 methods when the entire compilation takes 1084ms (table 1) is significant and actually noticeable to the user of the software.

Applying the approach results in a performance increase of 5% in the most trivial cases with no downside. When a source file is very complex containing 100 translation units with each translation unit also being very complex containing 100 methods an improvement of up to 30% can be observed.

In the graph which displays the different phases for the prototype compiler (figure 4) it is easy to see that the last 3 phases (ast, class and write) are non existent with caching enabled. The the complexity of a translation unit has been fixed to 50 methods to simulate a difficult workload.

In general an improvement in performance from 5% to 15% should be expected in normal scenarios. In a scenario where a lot of code generation is involved an improvement of up to 30% can be expected.

Figure 2. Speedup with caching in milliseconds

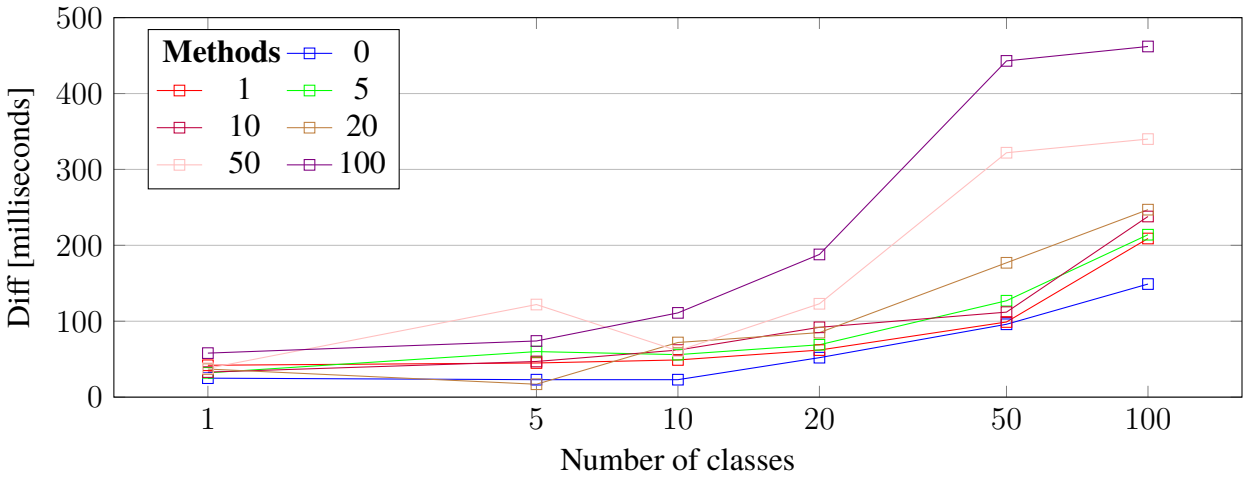


Figure 3. Speedup with caching in percent

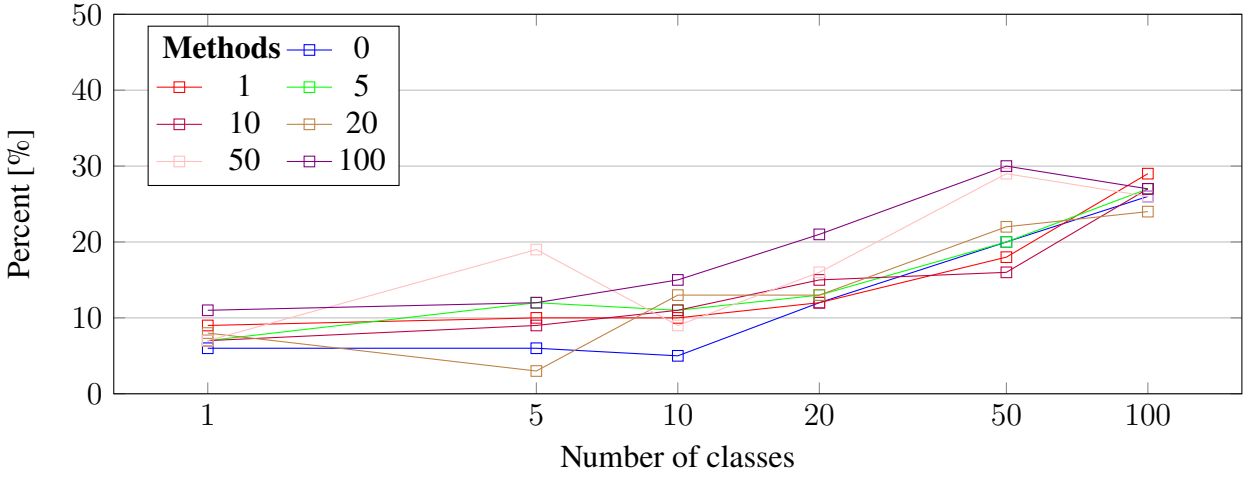
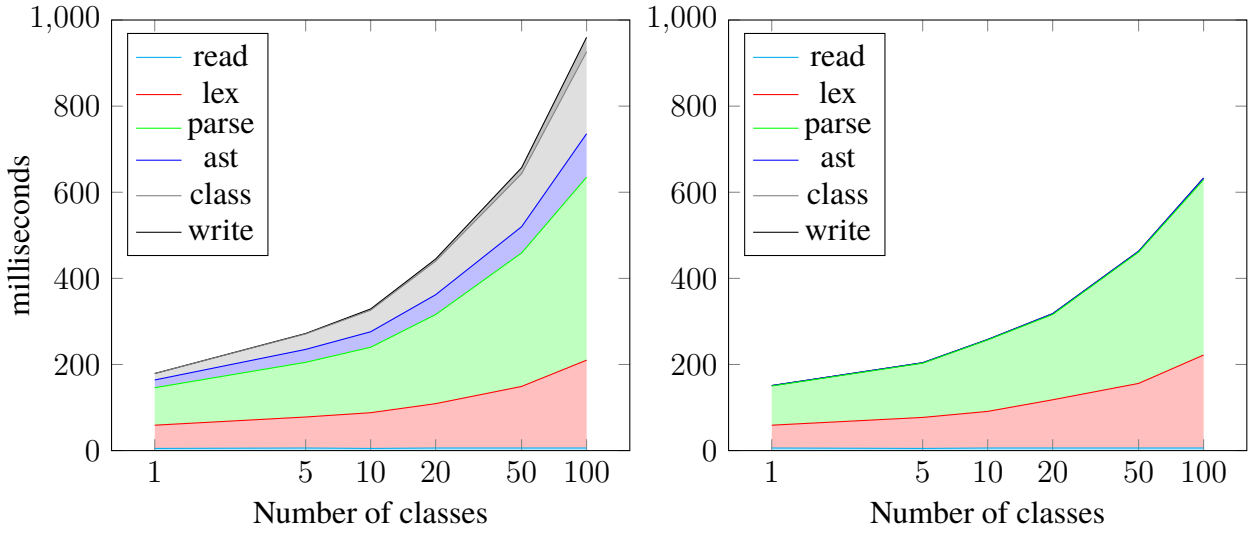


Figure 4. Runtime for 50 methods per class without and with caching



Results

In this thesis the translation unit granularization concept has been presented. An introduction was given to the current state of how compilers handle translation units in various programming languages. An analysis was provided how the scope of a translation unit can be reduced by taking modern language features into consideration.

The following the results were obtained:

1. For demonstration purposes of the viability of the presented approach, a prototype compiler has been written from scratch, implementing only a subset of the programming language, but designing it from the beginning with translation unit granularization in mind.

2. Compilation unit granularization was successfully implemented in the prototype compiler. Scala turned out a to be a perfect target for the implementation. It targets the JVM which in turn expects separate class files for every defined class in the source.

3. The prototype compiler was validated to create correct output allowing it to be benchmarked.

4. The prototype compiler was benchmarked with inputs of variable size and complexity. To generate variable inputs a mechanism was presented and detailed, which creates inputs with a varying number of translation units and complexity for benchmark. Various graphs were presented to better understand the performance improvements and the feasibility of the approach was discussed.

5. A comparison to the official Scala compiler was made and provided in the benchmarking data.

Conclusion

1. Translation unit granularization enables performance improvements during recompilation by simply decomposing work units and then applying uncomplicated approaches to avoid redundant work.

2. Object oriented programming languages which have multiple independent outputs are a natural fit for translation unit granularization.

Literature

[AMP+18] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. https://doi.org/10.3929/ethz-b-000311092, 2018. URL: https://doi.org/10.3929/ethz-b-000311092. accessed 2021-03-08.

[Blo08] Joshua Bloch. *Effective Java™, Second Edition*. Prentice Hall Press, USA, second **edition**, 2008, **pages** 179–180. ISBN: 9780137150021.

[Cra19] David Crawshaw. Fast compilers for fast programs. 2019. URL: https://crawshaw.io/blog/fast-compilers. (accessed: 2021.03.07).

[Ebe16] Christof Ebert. Devops, 2016. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7458761.

[EPF02] Ligthbend EPFL. Scala standard library documentation. 2002. URL: https://www.scala-lang.org/api/2.13.3/scala/Predef%5C$.html. (accessed: 2021-05-12).

[Ghu06] Abdulaziz Ghuloum. An incremental approach to compiler construction, 2006. URL: http://scheme2006.cs.uchicago.edu/11-ghuloum.pdf.

[Hal19] Sam Halliday. Scala almost succeeded. 2019. URL: https://betterprogramming.pub/scala-almost-succeeded-c3b1028b02c5. (accessed: 2021.03.07).

[HM98] Graham Hutton **and** Erik Meijer. Monadic parsing in haskell. *J. Funct. Program.*, 8(4):437–444, 1998-07. ISSN: 0956-7968. DOI: 10.1017/S0956796898003050. URL: https://doi.org/10.1017/S0956796898003050.

[LYB+13] Tim Lindholm, Frank Yellin, Gilad Bracha **and** Alex Buckley. 2013. URL: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html. (accessed: 2021-05-14).

[LR18] Nuno P. Lopes **and** John Regehr. Future directions for optimizing compilers, 2018. arXiv: 1809.02161 [cs.PL].

[Mey14] Jeff Meyerson. The go programming language, 2014. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6898707.

[Nie94] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994, **page** 135. ISBN: 9780080520292.

[Ode19] Martin Odersky. Scala language specification. 2019. URL: https://www.scala-lang.org/files/archive/spec/2.13/. (accessed: 2021.03.09).

[Ode21] Martin Odersky. Scala github repository. 2021. URL: https://github.com/scala/scala/tree/2.13.x/src/compiler/scala/tools/nsc. (accessed: 2021-05-07).

[Pik12] Rob Pike. Go at google: language design in the serice of software engineering. 2012. URL: https://talks.golang.org/2012/splash.article. accessed 2021-03-08.

[Sab09] Miles Sabin. Unboxed union types in scala via the curry-howard isomorphism. 2011-06-09. URL: http://milessabin.com/blog/2011/06/09/scala-union-types-curry-howard/. (accessed: 2021-05-11).

[Sab21] Miles Sabin. Shapeless github repository. 2021. URL: https://github.com/milessabin/shapeless. (accessed: 2021-05-11).

[San21] Ivan Sanchez. Why are java server-side developers not adopting kotlin? 2021. URL: https://medium.com/google-developer-experts/why-are-java-server-side-developers-not-adopting-kotlin-8eb53e06ee99. (accessed: 2021.03.07).

[sca02] scalaz. Scala library for functional programming. 2002. URL: https://github.com/scalaz/scalaz#scalaz. (accessed: 2021-05-14).

[Tor21] Eric Torreborre. Specs2 github repository. 2021. URL: https://github.com/etorreborre/specs2. (accessed: 2021-05-05).

[Wir95] Niklaus Wirth. A plea for lean software. *Computer*, 28(2):64–68, 1995. ISSN: 0018-9162. DOI: http://doi.ieeecomputersociety.org/10.1109/2.348001.

Acronyms

AST - Abstract Syntax Tree
JVM - Java Virtual Machine
JIT - Just In Time
DSL - Domain Specific Language
TDD - Test Driven Development
OOP - Object Oriented Programming
ADT - Algebraic Data Type
API - Application Programming Interface
IR - Intermediate Representation
JRE - Java Runtime Environment

Table 1. Benchmark results

Classes	Methods	Official	New	Cache	Diff	read	ast	parse	ast	class	write
1	0	1721	376	351	25	6	24	42	8/0	9/0	1/0
1	1	1595	426	384	42	6	28	63	11/0	9/0	0/0
1	5	1681	421	389	32	6	31	68	11/0	10/0	0/0
1	10	1624	430	397	33	6	36	70	11/0	10/0	0/0
1	20	1693	445	408	37	5	41	78	13/0	11/0	0/0
1	50	1760	476	438	38	5	54	87	18/1	15/0	0/0
1	100	1901	516	458	58	6	61	105	23/1	21/0	0/0
5	0	1549	375	352	23	6	26	41	7/1	10/0	1/0
5	1	1636	427	382	45	6	33	68	13/0	13/0	1/0
5	5	1687	473	413	60	6	44	83	14/0	19/0	1/0
5	10	1804	481	434	47	6	54	87	18/1	18/0	1/0
5	20	1941	511	494	17	6	63	108	24/1	25/0	1/0
5	50	2364	614	492	122	6	72	127	30/1	36/0	1/0
5	100	3004	616	542	74	6	80	152	36/1	46/0	2/0
10	0	1616	386	363	23	5	26	43	7/0	15/0	2/0
10	1	1661	448	399	49	5	38	72	12/0	19/0	2/0
10	5	1863	489	433	56	6	56	90	17/1	30/0	2/0
10	10	2088	537	475	62	6	64	105	26/1	34/0	2/0
10	20	2335	553	481	72	6	71	119	28/2	38/0	3/0
10	50	2876	621	560	61	5	83	152	36/1	50/0	3/0
10	100	3794	722	611	111	5	101	203	46/2	69/0	2/0
20	0	1692	413	361	52	5	29	46	7/0	33/0	5/0
20	1	1792	479	417	62	6	43	79	15/0	41/0	5/0
20	5	2073	530	461	69	5	64	104	23/2	40/0	5/0
20	10	2371	580	488	92	5	71	118	29/1	42/0	5/0
20	20	2784	616	531	85	5	79	147	34/1	47/0	6/0
20	50	3688	742	619	123	6	103	207	46/2	78/0	5/0
20	100	4147	894	706	188	6	134	277	63/3	92/0	6/0
50	0	1883	476	380	96	6	35	54	8/0	59/0	12/0
50	1	2282	541	442	99	5	58	89	21/1	62/0	11/0
50	5	2676	625	498	127	6	75	134	33/1	71/0	10/0
50	10	3115	675	563	112	5	84	165	39/1	78/0	12/0
50	20	4140	788	611	177	5	101	208	50/2	114/0	10/0
50	50	5111	1084	762	322	6	143	310	61/2	123/0	14/0
50	100	5452	1440	997	443	6	235	557	126/4	155/0	16/0
100	0	2226	556	407	149	6	42	58	10/1	123/0	20/0
100	1	2599	706	497	209	5	65	110	26/2	163/0	28/0
100	5	3431	780	566	214	5	85	168	38/1	152/0	26/0
100	10	4070	858	620	238	6	106	205	48/2	159/0	23/0
100	20	4750	992	745	247	5	140	277	65/4	167/0	25/0
100	50	5405	1305	965	340	6	204	425	101/4	190/0	34/0
100	100	6779	1710	1248	462	7	281	637	134/8	264/0	36/0