Towards a Framework for Stochastic Performance Optimizations in Compilers and Interpreters - An Architecture Overview

Oliver Krauss
Institute for System Software - ISS
Johannes Keppler University Linz
Linz, Austria
Advanced Information Systems and Technology - AIST
University of Applied Sciences Upper Austria
Hagenberg, Austria
oliver.krauss@fh-hagenberg.at

ABSTRACT

Modern compilers and interpreters provide code optimizations before and during run-time to stay competitive with alternative execution environments, thus moving required domain knowledge about the compilation process away from the developer and speeding up resulting software. These optimizations are often based on formal proof, or alternatively have recovery paths as backup.

This publication proposes an architecture utilizing abstract syntax trees (ASTs) to optimize the runtime performance of code with stochastic - search based - machine learning techniques. From these AST modifying optimizations a pattern mining approach attempts to find transformation patterns which are applicable to a software language. The application of these patterns happens during the parsing process or the programs run-time.

Future work consists of implementing and extending the presented architecture, with a considerable focus on the mining of transformation patterns.

CCS CONCEPTS

• **Software and its engineering → Patterns**; *Functionality*; *Search-based software engineering*; • **Computing methodologies** → *Instance-based learning*;

KEYWORDS

Performance Optimization, Stochastic Optimization, Pattern Mining, AST Transformation

ACM Reference Format:
Oliver Krauss. 2018. Towards a Framework for Stochastic Performance Optimizations in Compilers and Interpreters - An Architecture Overview. In *Proceedings of 15th International Conference on Managed Languages & Runtimes (ManLang'18)*, Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

When creating a new programming language or implementing a new execution environment (such as an interpreter or a compiler) for an existing language, one challenge is to achieve sufficient performance to be competitive with other existing environments. The Truffle interpreter Framework [21] as well as the Graal just-in-time (JIT) compiler [12] both address these needs by providing a framework to rapidly prototype new languages, which already have the benefit of optimization built in due to the aggressive optimizations of the Graal compiler.

This publication, which is part of an ongoing research series [6], presents an addition to Truffle and Graal in the form of stochastic performance optimizations. These optimizations are done with Genetic Improvement (GI) [8, 10]. GI applies search based machine learning algorithms, such as genetic or evolutionary algorithms, to code in order to improve it's non functional features such as runtime performance, memory usage or even stability by automatically fixing bugs. The core research question of the series is to find optimizations using GI that improve run-time performance and provide benefits beyond already existing compiler optimizations. The research concentrates on additional optimizations; replacing existing ones is not our goal. This publication primarily addresses a suggested architecture that enables stochastic optimizations of code in Truffle and Graal and a subsequent identification of code transformation patterns that can be applied in a deterministic way, and are also fast enough to apply during program runtime. The GI part is out of the scope of this publication, with details available in previous publications in the series [7].

In the context of this publication stochastic code optimizations are executed on an abstract syntax tree (AST), modifying the AST before it is interpreted by Truffle or compiled by Graal. The originating AST is provided by an implementer, and the modified AST is reached using GI machine learning techniques. These techniques are the reason the optimizations have to be considered stochastic, as the outcome of the optimization cannot be predicted. This provides the challenge of verifying that the modified AST retains the functionality of the original AST. The advantage of this approach lies in the implementation specific optimizations which enable significant performance improvements [10]. The application of these optimizations is considered to be especially useful in domains where some algorithmic accuracy can be sacrificed for faster performance, such as improving the performance of heuristic algorithms, or for example shaders [16].

The remainder of this publication is structured as follows:

- Related work in section 2 presents applications of machine learning in compiler optimizations.
- The proposed architecture which enables stochastic optimizations, and identification of transformation patterns, as well as the application of these patterns is discussed in Figure 3.
- Planned future steps concerning the development and evaluation of the presented framework are outlined in section 4.
- A conclusion is presented in section 5

2 BACKGROUND

The following section gives an overview of the Graal Compiler, the Truffle Interpreter and Genetic Improvement. In the context of this work these technologies are used as follows:

- Graal is used as compiler for optimized languages, It is a highly optimizing JIT compiler that is available in OpenJDK since Java 9 and is used to compile Truffle ASTs from bytecode to machine code.
- Truffle is the interpreter framework that the optimized languages are written in. It can interpret AST nodes on the JVM without the use of Graal, albeit with a lower performance.
- Genetic Improvement is a search based machine learning technique used to optimize the Truffle ASTs before they are interpreted by Truffle or compiled by Graal.

The technology stack was selected as Truffle and Graal allow rapid prototyping of a programming language. Additionally, the AST structure of Truffle enables easy manipulation of a given AST [21, 22]. These reasons make them suitable for stochastic optimizations.

2.1 Graal

Graal [12] is an aggressively optimizing just-in-time (JIT) compiler, written in Java as part of the OpenJDK project. It compiles Truffle ASTs (in bytecode) to efficient machine code and features several optimizations, including speculative optimizations which can be taken back if necessary (called deoptimization). Graal uses an IR that is a directed graph describing both control flow and data flow. The Graal IR can be viewed with the Ideal Graph Visualizer (IGV). Both Graal and IGV are open source and build a basis for research into compiler optimizations. [15, 17]

2.2 Truffle

Truffle [21] is a self-optimizing interpreter framework for implementing new languages based on Abstract Syntax Trees (AST). Truffle itself does not feature a lexer, parser or linker. The focus relies solely on implementing AST nodes that are combined into a Truffle language that can be executed on any Java VM. Every node of an implemented language consists of a generic implementation, and several specializations for each datatype the node supports. This enables dynamic typing of the language, as well as a self-optimization of the language by rewriting the nodes through specialization or generalization. The Truffle optimizer integrates with the Graal compiler for high performance compilation and execution. As Truffle-based languages are executed on the JVM, they provide several services such as garbage collection. Truffle already features several guest language implementations, including Python, Ruby, JavaScript and C. Truffle, as well as several of its guest languages, are open source. [2, 22]

2.3 Genetic Improvement

GI applies search based machine learning algorithms to code, in order to improve it's non functional features such as runtime performance, memory usage or even stability by automatically fixing bugs. The algorithms applied are often from the domain of genetic algorithms, which maintain a population of possible solutions to a problem and breed them into better solutions over several generations by using evolutionary operators such as crossover and mutation. Each individual in this population is assigned a quality which is calculated through a fitness function. [3, 10, 20]

3 ARCHITECTURE OVERVIEW

The architecture for stochastic performance optimizations is rooted in the concept of modifying abstract syntax trees (AST) which are then interpreted or compiled. In order to lay out the essential concepts behind this architecture, a running example is used for the remainder of this section. The example uses a for loop to calculate the sum of all values in a list of positive integers. As can be seen in Listing 1 the iteration is done with accessing each element with a *get* method and an index. Figure 1 shows the running example as a Truffle AST. While this code is functionally correct it presents a significant loss of performance compared to the use of an iterator. While this example is constructed, and thus obvious performance losses like this occur often and can happen simply due to missing domain knowledge of a developer. For example in Java the same problem could be solved with the streaming library, improving the performance even further.

Listing 1: Running Example - for loop calculating the sum of all values in a list of positive integers

```java
int x = 0;
List<Integer> a = listOfPositiveInt();
for (int i = 0; i < a.size(); i++) {
  x += a.get(i);
}
```

3.1 Optimization Lifecycle

The optimization process is split into two lifecycle phases as can be seen in Figure 2.

The offline phase is targeted towards optimizing specific code (a) and towards discovering generally applicable patterns in a Truffle language (b). The core concern when optimizing specific code, is retaining the functional aspects of it, while improving on runtime performance. From a greater number of these optimizations language specific recurring patterns can be identified by comparing in- and output ASTs.

The online phase happens during the a programs runtime. Before a parsed Truffle AST is handed to Truffle for interpretation, or to Graal for compilation a check happens if it fits any of the patterns

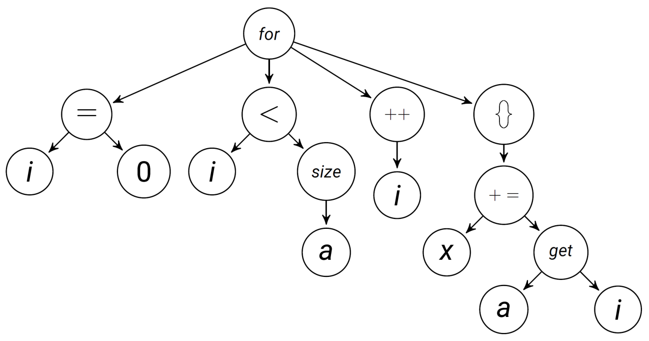

Figure 1: Running Example - Truffle AST representation of Listing 1

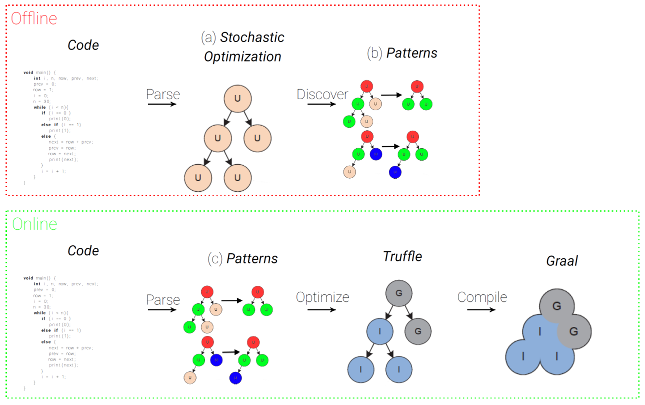

Figure 2: The optimization process is split into an offline and online phase. In the offline phase stochastic optimizations (a) are applied to code, and recurring transformation patterns (b) are identified. These patterns are then applied to code in the online phase (c).

which were identified in the offline phase. If it does the pattern is applied, and the modified AST is handed to Truffle or Graal.

This lifecycle is supported by a *Knowledge Base*, and two closely related frameworks, which can be seen in Figure 3. Both of the frameworks are integrated in a Java Hotspot VM execution environment, with Graal as compiler and Truffle as interpreter. While the frameworks themselves are language independent, the learning of patterns is happening on a guest language, for which a knowledge base is built up. The optimizations themselves are specific to a *Guest Application*.

3.2 Language Analysis

In order to enable optimizations on any *Guest Language*, steps need to be taken to gain a general information about all available node implementations which can form an AST. This is done by the *Truffle Language Analyzer (TLA)*. Using the current ClassLoader the *TLA* collects and analyzes all existing AST nodes in all packages or classes given by an implementer into the *Truffle Language Information (TLI)*. AST nodes are identified by checking if they implement the Node class provided by Truffle.

Listing 2: Example Truffle Implementation

```
@NodeChildren({
  @NodeChild("leftNode"),
  @NodeChild("rightNode")
})
public abstract class
    MinicIntArithmeticNode extends
    MinicIntNode {
  public abstract static class
      MinicIntAddNode extends
      MinicIntArithmeticNode {
    @Specialization
    public int add(int left, int right) {
      return left + right;
    }
  }
}

@GeneratedBy(MinicIntAddNode.class)
public static final class
    MinicIntAddNodeGen extends
    MinicIntAddNode {
  @Child private MinicIntNode leftNode_;
  @Child private MinicIntNode rightNode_;
  private MinicIntAddNodeGen(MinicIntNode
      leftNode, MinicIntNode rightNode) {
    this.leftNode_ = leftNode;
    this.rightNode_ = rightNode;
  }
  public static MinicIntAddNode
      create(MinicIntNode leftNode,
      MinicIntNode rightNode) {
    return new MinicIntAddNodeGen(leftNode,
        rightNode);
  }
  ...
}
```

The purpose of the TLI is to provide information on nodes of a language in a generalized way. It contains information on which nodes are terminal nodes, which nodes are non-terminal and a mapping from each abstract node (represented as an abstract class) to all found implementations of that node is provided. This information is persisted to the *Knowledge Base*, and used by the *Optimizer* for stochastic optimization, and the *Pattern Framework* to identify language specific patterns. An example of an abstract node can be seen in Listing 2 with the classes MinicIntArithmeticNode and MinicIntAddNode. Truffle auto-generated the specific class MinicIntAddNodeGen below the generic classes. Finally, the TLI contains an initialization mechanism for each non-abstract class as each Truffle Node can be instantiated by only one of the following three methods:

- A given public constructor exists only on non-abstract classes created by an implementer

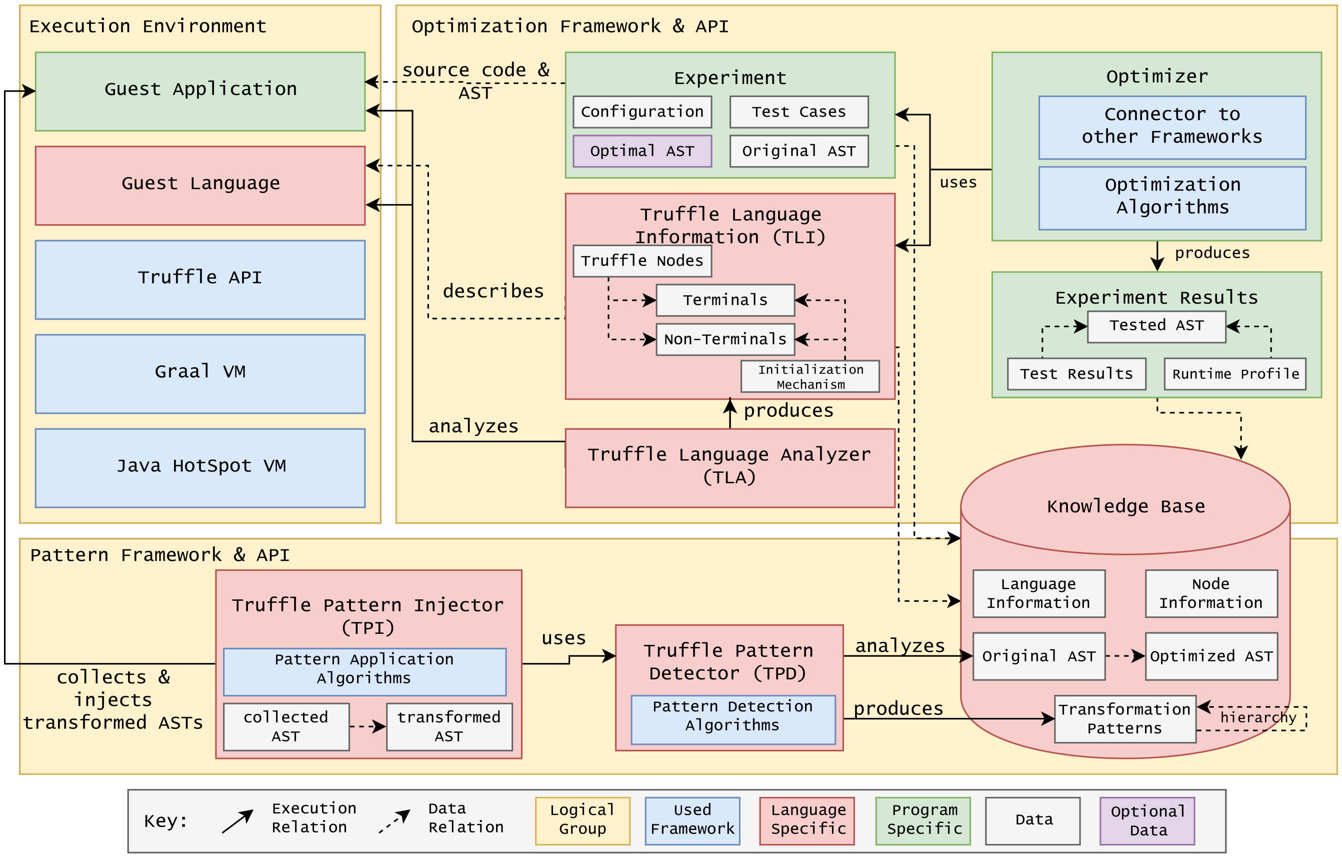

Figure 3: The architecture of the optimization framework in combination with Truffle and Graal (extended from [7].)

- A create method that is auto-generated from Truffle nodes for abstract classes (as seen in Listing 2)
- A node factory that is auto-generated by Truffle with the GenerateNodeFactory annotation

3.3 Stochastic Optimization

The stochastic optimization (Figure 2 -> (a), Figure 3 -> program specific part of the optimization framework and application programming interface (API), from an interpreter and compiler perspective needs to address the need of retaining the required syntactic correctness of the optimized code. In the architecture, this is achieved by defining *Experiments*, conducted by an *Optimizer* who produces *Experiment Results*, which are also persisted in the *Knowledge Base*. These experiments contain a selection of test cases (in / output value combinations) which are applied to the original AST for verification, and then to the modified AST for a comparison. This results in a correctness score between 0 and 1, formally defined as follows [7]:

$$correctness_{(AST,tests)} = \frac{\sum_{n=1}^{tests} succeeded_{(AST,test)}}{\sum tests} \tag{1}$$

In the context of this work the definition of *required* correctness lies with the person utilizing the optimization framework. For many software domains, compilers and intepreters especially, keeping the syntactical correctness intact, is essential. For other domains the *required* correctness is less than an exact match. Amongst those are, for example are graphics engines, specifically shaders, where a high framerate is more important than rendering an exact image [9, 16].

Since compilers have been proven to introduce bugs and security issues with optimizations even when there exists formal proof of correctness [1, 19], and the applied optimizations can not produce proof, an annotation preventing the optimization technique entirely is provided in the framework. Inversely where the requirement of code correctness can be reduced in favor of performance another annotation is introduced that allows a developer to define how much an optimized version is allowed to differ from the original solution.

To generalize the approach of verifying test correctness, a regression test suite is automatically generated by input fuzzing. This is achieved by utilizing already existing test suite generation frameworks such as Randoop [13] or EvoSuite [14]. As Truffle provides access to the memory, intermediate values can also be monitored during testing. In addition to the automatically generated tests, developers can define manually created tests and add them in the optimization annotation. In order to prioritize parts of the functionality, manually defined tests can be selected as *must pass*, in

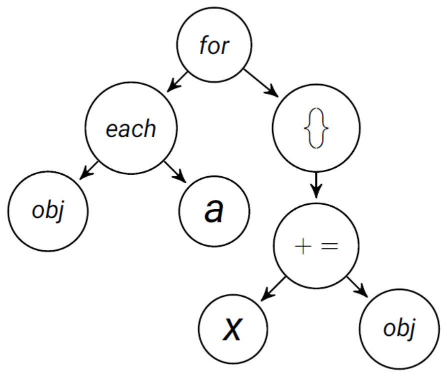

Figure 4: Running Example - Optimized version of Figure 1

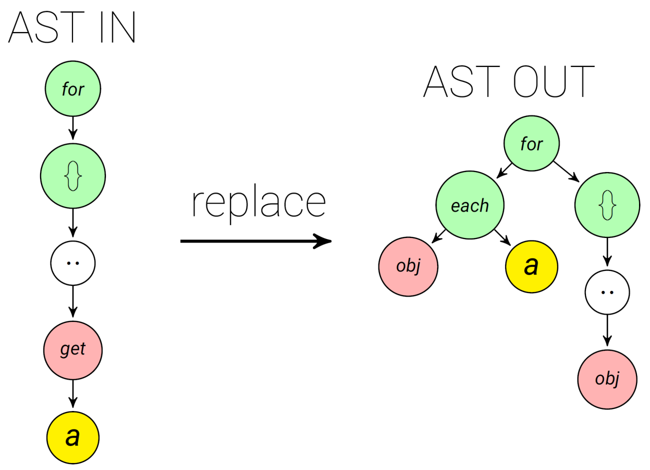

Figure 5: Running Example - pattern for replacement of *for* loops with *foreach* loop

addition to defining a percentage of tests that needs to be passed, or value deviations from an/several inputs.

The runtime of both the original AST and the optimized one is analyzed and a *RuntimeProfile* is created containing statistical information such as the Median, Quartiles, minimal and maximal runtimes, and the amount of executions in each quartile. This performance analysis must be performed on separate JVM instances. Otherwise every AST being measured would benefit from the optimizations the Graal Compiler already performed on previous ASTs. For correct measurement 200.000 executions are recorded. Graal has a warm-up phase for several of its optimizations which requires a correct performance measurement to ignore executions before the initial warm-up is completed.

When considering the running example, a possible improvement that can be found with the stochastic optimization is replacing the for loop and get access, with a for each loop. This is shown in Figure 4. In this case the developer used neither annotation, leading to a default satisfying all test cases. In the example a developer would not need to manually define test cases either.

3.4 Pattern discovery

When a sufficient amount of *Experiments* has been conducted, the *Truffle Pattern Detector (TPD)* can be used to find *Guest Language* speficic patterns and submit them to the *Knowledge Base*. The reason why they are specific is that the discovery uses the AST node classes. To identify language independent patterns, a mapping between node implementations of two languages would need to be defined.

The pattern detection is done by using pattern mining algorithms on the optimized ASTs in the *Knowledge Base*. Here the implementation work has not started yet. Currently considered approaches go in the direction of substructure pattern mining, as even partial overlaps in a tree can lead to a pattern. This is shown in Figure 5 where the white ".." nodes represent a part of the tree which is not of importance to the pattern [18, 23]. When a pattern has been found in the Optimized ASTs, their corresponding original ASTs are selected for pattern mining as well. If a pattern can be identified in the original ASTs as well, the two patterns, from modified and original ASTs, combine to form a transformation pattern. The transformation pattern contains information about which AST nodes need to be used for identification, which can be copied verbatim, and which nodes must be replaced.

Considering the running example, after several for to for each transformations have been observed, a transformation pattern is discovered as shown in Figure 5. The *for* and *{}* nodes (green background, left) are used as identification in originating ASTs and the transformation is resulting in a base of *for*, *each* and *{}*. The *get* node is replaced with a new variable *obj*. The variable *a* is retained at every AST position it was identified at, while the ".." nodes represent any AST which will be copied verbatim.

Some transformation patterns will relate to other transformation patterns in the sense, that only one of them can, or should be applied. The running example the pattern, as shown in Figure 5, does not consider a modifying access to the collection a. Thus it should be superceded by pattern which does consider that access. This means that the patterns follow a hierarchy where the most specific pattern will be applied, and more general patterns will be applied only if none of the specific patterns match.

The advantage of using this pattern discovery is that the optimization is language dependent, but problem independent. However as all patterns are discovered though a knowledge base, the patterns that can be discovered, depend on the *Experiments* conducted. This means that it is possible to create domain specific knowledge bases. One additional advantage of the pattern discovery is that these patterns can be analyzed by an expert and mathematically (dis)proven, as opposed to the stochastic optimization.

3.5 Pattern application

The final process step is the application of found transformation patterns to any given *Guest Application*. This is handled by the *Truffle Pattern Injector (TPI)* either during runtime or while parsing.

During runtime there are several injection options:

The first is during the parsing process of a given *Guest Language*, and must be added manually by a developer. There the ASTs which are parsed are given to the *TPI* for optimization.

The second option is right before the first time a Truffle AST is handed to the Truffle polyglogt engine for interpretation or to Graal for compilation. This means that when a function is loaded from the function registry the TPI interjects the AST, analyzes it for patterns, and applies applicable ones, and only then forwards it.

The third and final (currently considered) option is intercepting dispatches between functions. This does come with an added runtime overhead, but has the advantages of being completely language independent, as well as enabling profiling, meaning the AST transformation is only conducted at often called functions, or functions that represent bottlenecks.

The option of intercepting dispatches has one additional advantage. Due to the interception, original input and returned values can be observed. The *TPI* can use these values, to test the AST that was created with the transformation pattern, against the results that would have been produced by the original AST, which is kept by the *TPI*. This testing can happen in regular intervals, randomly or with often/rarely observed values. Whenever an instance is observed where the original AST would have produced a different output, the pattern can be invalidated, and the original AST can be replaced back into the running program. The invalidated pattern, together with the observed incorrect instance is then submitted to the *Knowledge Base* and can then be analyzed by a developer at a later date.

4 FUTURE WORK

The presented architecture for stochastic optimization and pattern recognition is currently still under development. Looking at the Figure 3, a *Guest Language*, MiniC, a subset of the C11 standard [4] has been developed alongside the architecture to prototype and test concepts. This language will be continually used during the development and research publications. One important goal is to include more Truffle languages, and make the *Optimization Framework* and *Pattern Framework* applicable to any Truffle language.

Most of the *Optimization Framwork* has already been implemented. Current goals are to finish work here, as the presented automated test generation does not as of yet exist, and the proposed annotations concerning enabling/disabling stochastic optimizations are still in development. The reason for this is that currently all Experiments and Tests were created manually to verify the process and refine the laid out architecture. Another important step is a connector to HeuristicLab [5], a framework for heuristic and evolutionary computation which will provide a solid basis for the stochastic optimization, as well as an option of distributed computation utilizing HeuristicLab Hive [11].

The pattern discovery and injection in *Pattern Framework* do not as of yet exist. The architecture laid out in this publication is a first proposal how to tackle these challenges. Hover the Knowledge Base does already exist, and a manual identification of some simple patterns is possible.

5 CONCLUSION

This publication presents a novel architecture for stochastic optimization of code, and building on that identification of language specific AST transformation patterns which can be used to optimize ASTs in a more general way, and without the overhead the stochastic optimization poses. This is achieved by two distinct lifecycles: *Offline* - stochastic optimization, identification of patterns; *Online* - application of patterns, testing and possible invalidation of patterns.

The core of the architecture consists of a *Knowledge Base* and two Frameworks for *Optimization* and *Patterns* respectively, which directly integrate with the Truffle [21] interpreter and the Graal [12] compiler.

The overarching goal of the architecture is to enable additional AST-modification based optimization techniques in compilers and interpreters, in addition to other already existing optimizations. While the focus of this work lies in the optimization of runtime performance, the resulting architecture and frameworks are general enough to enable research / optimizations in other domains, for example memory use or runtime stability.

ACKNOWLEDGMENTS

This publication is part of an ongoing doctoral thesis which is advised by Hanspeter Mössenböck [1] and Michael Affenzeller [2].

REFERENCES

[1] V. D'Silva, M. Payer, and D. Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *2015 IEEE Security and Privacy Workshops*. 73–87. DOI: http://dx.doi.org/10.1109/SPW.2015.33

[2] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance Cross-language Interoperability in a Multi-language Runtime. *SIGPLAN Not.* 51, 2 (Oct. 2015), 78–90. DOI: http://dx.doi.org/10.1145/2936313.2816714

[3] Gregory S. Hornby. 2006. ALPS: The Age-layered Population Structure for Reducing the Problem of Premature Convergence. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO '06)*. ACM, New York, NY, USA, 815–822. DOI: http://dx.doi.org/10.1145/1143997.1144142

[4] ISO. 2011. *ISO/IEC 9899:2011 Information technology — Programming languages — C.* International Organization for Standardization, Geneva, Switzerland. 683 (est.) pages. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853

[5] Michael Kommenda, Gabriel Kronberger, Stefan Wagner, Stephan Winkler, and Michael Affenzeller. 2012. On the Architecture and Implementation of Tree-based Genetic Programming in HeuristicLab. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '12)*. ACM, New York, NY, USA, 101–108. DOI: http://dx.doi.org/10.1145/2330784.2330801

[6] Oliver Krauss. 2017. Genetic Improvement in Code Interpreters and Compilers. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2017)*. ACM, New York, NY, USA, 7–9. DOI: http://dx.doi.org/10.1145/3135932.3135934

[7] Oliver Krauss, Hanspeter Mössenböck, and Michael Affenzeller. 2018. Dynamic Fintess Functions for Genetic Improvement in Compilers and Interpreters. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '18)*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/3205651.3208308 In Press.

[8] William B. Langdon. 2015. *Genetic Improvement of Software for Multiple Objectives*. Springer International Publishing, Cham, 12–28. DOI: http://dx.doi.org/10.1007/978-3-319-22183-0_2

[9] W. B. Langdon and M. Harman. 2010. Evolving a CUDA Kernel from an nVidia Template. In *2010 IEEE World Congress on Computational Intelligence*, Pilar Sobrevilla (Ed.). IEEE, Barcelona, 2376–2383. DOI: http://dx.doi.org/doi:10.1109/CEC.2010.5585922

[10] W. B. Langdon and M. Harman. 2015. Optimizing Existing Software With Genetic Programming. *IEEE Transactions on Evolutionary Computation* 19, 1 (Feb 2015), 118–135. DOI: http://dx.doi.org/10.1109/TEVC.2013.2281544

[11] Christoph Neumüller, Andreas Scheibenp, Stefan Wagner, Andreas Beham, Michael A enzeller, and Josef Ressel-Centre. 2011. Large Scale Parameter Meta-Optimization of Metaheuristic Optimization Algorithms with HeuristicLab Hive.

[12] OpenJDK. 2018. Graal Project. (2018). http://openjdk.java.net/projects/graal/ Last Accessed - 2018-05-11.

[1] hanspeter.moessenboeck@jku.at

[2] michael.affenzeller@fh-hagenberg.at

[13] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 75–84. DOI: http://dx.doi.org/10.1109/ICSE.2007.37

[14] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering* 22, 2 (2017), 852–893. DOI: http://dx.doi.org/10.1007/s10664-015-9424-2

[15] Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. 2015. Snippets: Taking the High Road to a Low Level. *ACM Trans. Archit. Code Optim.* 12, 2, Article 20 (June 2015), 20:20:1–20:20:25 pages. DOI: http://dx.doi.org/10.1145/2764907

[16] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. 2011. Genetic Programming for Shader Simplification. *ACM Trans. Graph.* 30, 6, Article 152 (Dec. 2011), 12 pages. DOI: http://dx.doi.org/10.1145/2070781.2024186

[17] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, and Thomas Würthinger. 2012. Compilation Queuing and Graph Caching for Dynamic Compilers. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '12)*. ACM, New York, NY, USA, 49–58. DOI: http://dx.doi.org/10.1145/2414740.2414750

[18] Chen Wang, Mingsheng Hong, Jian Pei, Haofeng Zhou, Wei Wang, and Baile Shi. 2004. Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining. In *Advances in Knowledge Discovery and Data Mining*, Honghua Dai, Ramakrishnan Srikant, and Chengqi Zhang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 441–451.

[19] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 260–275. DOI: http://dx.doi.org/10.1145/2517349.2522728

[20] Darrell Whitley, Soraya Rana, and Robert B. Heckendorn. 1998. The Island Model Genetic Algorithm: On Separability, Population Size and Convergence. *Journal of Computing and Information Technology* 7 (1998), 33–47.

[21] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. ACM, New York, NY, USA, 13–14. DOI: http://dx.doi.org/10.1145/2384716.2384723

[22] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, USA, 187–204. DOI: http://dx.doi.org/10.1145/2509578.2509581

[23] Xifeng Yan and Jiawei Han. 2002. gSpan: graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* 721–724. DOI: http://dx.doi.org/10.1109/ICDM.2002.1184038