Delft University of Technology, Bachelor Seminar of Computer Science and Engineering

Optimizing SGLR Parser Performance

Justin de Ruiter

Supervisors

Eelco Visser, Jasper Denkers, Daniël Pelsmaeker

Delft University of Technology

The Netherlands

Abstract

The Scannerless Generalized-LR (SGLR) parsing algorithm allows parsing of declarative and modular syntax definitions. However, SGLR is notorious for having low performance, negatively impacting its adoption in practice. This paper presents several performance optimizations for JSGLR2, which is an implementation of SGLR. All optimizations are implemented and evaluated in parallel, which is possible due to JSGLR2's modular architecture. The evaluation is performed using existing sources from three different languages. A combined speed-up of 9% up to 44% is achieved, improving the practicality of JSGLR2.

Keywords SGLR parsing, performance optimization

1 Introduction

Computers usually apply parsers to perform source code analysis. Parsers transform the source code to a more computer-readable parse or syntax tree for a given language. Creating or maintaining a parser can require substantial effort, especially for complex or rapidly evolving languages.

Parser generators aim to simplify creating and maintaining parsers. They can generate parsers based on a formal language description, eliminating the need to write or modify parsers by hand. However, traditional parser generators such as ANTLR [10] use deterministic parsing algorithms. These algorithms only support a subset of possible languages, sacrificing declarativeness [5] for performance.

The Scannerless Generalized-LR (SGLR) [13] parsing algorithm solves the declarativeness issue of traditional parser generators. SGLR extends GLR [8], which means it supports non-deterministic and ambiguous languages. Furthermore, support for scannerless parsing [11] allows character-level instead of token-based languages. These features result in SGLR supporting the entire class of context-free languages, enabling modular language definitions for increased maintainability.

However, the greater complexity of SGLR does have a performance cost, limiting its adoption in practice. GLR's non-determinism results in a worst-case run-time complexity of $O(n^3)$ [6], where traditional parsing algorithms run in linear time. Ideas from Elkhound [9] improve this by hybridly switching to a deterministic parsing mode whenever possible. It is nonetheless still slow for heavily non-deterministic languages. SGLR's lower performance impacts its practicality, e.g., for use in interactive IDE environments.

To facilitate improving SGLR's performance, Denkers introduced a modular architecture for SGLR [3]. The architecture splits the algorithm into several components, allowing systematic improvement of individual parts of the algorithm in parallel. Furthermore, it simplifies maintaining several parser variants with different features. An implementation of this architecture known as JSGLR2 is available as part of the open-source Spoofax Language Workbench[1]. JSGLR2 supports defining languages using SDF3 [2] which includes features such as modular language definitions.

As Denkers mentioned, there is a limited amount of time spent on optimizing JSGLR2's performance. Spending additional time on optimizations can therefore lead to further improvements. Furthermore, the usefulness of the modular architecture for implementing optimizations can be evaluated more extensively.

This paper presents several performance optimizations for JSGLR2 to improve its practicality. The performance of the optimizations is evaluated using existing sources from three different languages. An additional evaluation of all optimizations is also performed. Furthermore, the usefulness of its modular architecture for implementing optimizations is evaluated.

The remainder of this paper is structured as follows. First, an overview of the SGLR parsing algorithm is provided in Section 2. Section 3 describes the method used to discover and evaluate optimizations. Following in Section 4, the optimizations including implementation details are presented. Section 5 shows the benchmark results for these optimizations. In Section 6 the reproducibility of the results is briefly discussed. Finally in Section 7 the resulting conclusions are given.

[1] `http://www.metaborg.org/en/latest/`

2 Scannerless Generalized-LR Parsing

This section gives an overview of the relevant parts of the SGLR algorithm used as a reference for Section 4.

2.1 LR Parsing

SGLR is a left-to-right (LR) [7] shift-reduce parsing algorithm. It incrementally builds a parse tree from a given input by performing `Shift` and `Reduce` actions on a stack. The stack consists of stack nodes connected by links, where each node corresponds to a specific state. Each link contains a parse node, which represents a sub-tree of the final parse tree. `Shift` actions add a new link to the top of the stack. The parse node on this link corresponds to the current token. The input is advanced to the next token. `Reduce` actions reduce stack nodes on top of the stack to a single new node. The state of the new stack node is the goto state of the given `Reduce` action. The parse node on the new link is a parent of the parse nodes on the reduced stack links. Actions are queried from a parse table depending on the current input and the state of the top stack node. When a parse is complete, the stack consists of a single link where its parse node represents the final parse tree. This parse tree is transformed into a syntax tree, which is known as imploding.

2.2 GLR Parsing

Since SGLR extends GLR, it needs to handle non-determinism and ambiguities. For this, it has to maintain multiple stacks in parallel. An efficient way of doing this is using a Graph-Structured Stack (GSS) [12]. A GSS represents the stacks as a directed graph, where the source nodes (nodes without incoming links) indicate the top nodes of the stacks. Ambiguities are represented by allowing multiple parse nodes on the same link. Figure 1 shows an example of such a graph. SGLR internally keeps track of the top stack nodes using a collection called `active-stacks`. It processes the stacks one by one, keeping track of which stacks are yet to be processed using a collection called `for-actor`.

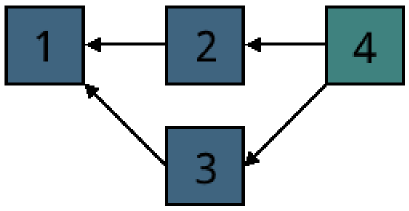

Figure 1: A Graph-Structured Stack with two active stacks: [1, 2, 4] and [1, 3, 4].

2.3 Scannerless Parsing

SGLR is a scannerless parsing algorithm, which means it works using characters instead of tokens. Scannerless parsing introduces additional ambiguities which are handled using reject productions. Reject productions enable the parser to reject ambiguous stacks if applicable. Rejectable stack nodes need to be processed in a specific order and after all other stacks are processed. For this SGLR uses an additional collection called `for-actor-delayed`.

2.4 The Algorithm

The remainder of this section summarizes the relevant parts of each function in the SGLR algorithm.

`PARSE`: The main parse loop. `active-stacks` is initialized after which `PARSE-CHARACTER` is called for each character in the input.

`PARSE-CHARACTER`: A parse round in which a single character is parsed. All active stacks are copied to `for-actor`. Stacks are removed from `for-actor` and `for-actor-delayed` which are then processed in `ACTOR`. `SHIFTER` is called when both for-actor collections are empty.

`ACTOR`: Process actions for an active stack. All applicable actions are queried from the parse table. Shift actions are stored in `for-shifter`. Reduce actions are handled directly in `DO-REDUCTIONS`.

`DO-REDUCTIONS`: Perform a reduce action for a given active stack. First, all possible reduction paths in the graph are collected. Then, for each path, the origin stack node is determined and its child parse nodes are collected. These are passed on in a call to `REDUCER`.

`REDUCER`: Perform a reduce action for a given path. First, a new parent parse node is created for all child parse nodes on the path. Afterward, this parse node has to be added to a link from a stack node with the corresponding goto state to the origin stack node. Three different scenarios are considered to achieve this.

1. The first scenario considers the case where there is no active stack with the goto state. In this case, a new active stack is created with a link to the origin stack. This new active stack is now potentially applicable for actions, so it is added to either `for-actor` or `for-actor-delayed` depending if it is rejectable. The other two scenarios handle the case when there already exists a goto stack with the target state.
2. In the second scenario, the goto stack has a direct link to the origin stack. In this case, the parse node is simply added to the existing link, introducing an ambiguity.
3. The last scenario handles the case when there is no direct link. In this case, a new link is added from the goto stack to the origin stack. Since this possibly introduces new reduction paths, all already processed active stacks have to be reconsidered. The re-processing is done in `DO-LIMITED-REDUCTIONS`, handling reduction paths through the new link.

`DO-LIMITED-REDUCTIONS`: Perform a reduce actions for the given active stack for paths through the new link. This is done similarly to `DO-REDUCTIONS`, but here the paths are additionally filtered on the through-link.

`SHIFTER`: Perform all shift actions for a single character. All shift actions in `for-shifter` are processed, possibly introducing new active stacks for the next parse round.

3 Method

This Section describes the process with which useful performance optimizations are elicited and evaluated. This is done using an iterative approach consisting of the following five steps: Identifying a bottleneck, finding an optimization to improve this bottleneck, implementing the optimization, verifying its correctness, and finally evaluating the optimization. These steps are discussed in more detail in the rest of this Section. Only a single optimization is considered in one iterative cycle to not allow other optimizations to interfere with its evaluation. Additionally, multiple cycles might be needed for a single optimization to further tune its performance.

The first step is to identify bottlenecks. This is useful since bottlenecks usually indicate parts of the system that benefit the most from optimizations. Bottlenecks are identified using a profiler, in this case, the free and open-source async-profiler[2] is used. async-profiler supports generating flame graphs[3], which gives a visual overview of where the most time is spent in a process.

In the second step, optimizations are discovered to improve the bottlenecks found in the first step. Though some bottlenecks might have a simple or known solution, it can be the case that there is no clear-cut improvement. In this case, finding an optimization can be trickier since it relies more on creativity. For these bottlenecks, improvements are mostly found by detecting parts where unnecessary or duplicate work is done. Additionally, JSGLR2 contains measurements that can help with tuning optimizations. These measurements can be extended if necessary. A subset of these measurements can be found in Appendix B.

After having found an optimization it is implemented in JSGLR2. Each optimization is implemented in a separate git branch to simplify evaluating each optimization separately. The implementations conform as much as possible to JSGLR2's modular architecture to simplify both the integration into different parser variants and the combination of optimizations.

The last two steps are to verify the correctness and evaluate the performance of the optimization. Verification is done using the existing integration tests in JSGLR2. The evaluation is done using the evaluation suite for JSGLR2. The evaluation suite and the benchmark setup are described in more detail in subsection 5.1.

[2]https://github.com/jvm-profiling-tools/async-profiler
[3]http://www.brendangregg.com/flamegraphs.html

4 Performance Optimizations

This section presents several performance optimizations for JSGLR2 with descriptions of their implementations.

4.1 Merging Stack Collections

SGLR maintains three separate stack collections: `active-stacks`, `for-actor` and `for-actor-delayed`. `for-actor` in conjunction with `for-actor-delayed` are used to iterate over `active-stacks`. `for-actor-delayed` is ordered to handle reject productions, as opposed to the other collections which are unordered.

Since `for-actor` and `for-actor-delayed` are only used to iterate over `active-stacks`, they can be optimized by becoming iterators over `active-stacks`. This eliminates the need to copy to- and remove stacks from both `for-actor` and `for-actor-delayed`.

This is implemented as a single array-based collection consisting of three segments: The first segment represents `for-actor-delayed`, the second segment `for-actor` and the third segment contains the remaining active stacks. An example of this representation can be seen in Figure 2. Adding and removing stacks to `for-actor` is done by moving the second cursor to the right or left respectively. Additionally, the first cursor has to be moved when dealing with delayed stacks.

This ordering of segments is convenient since it avoids having to copy part of the array when adding to the first or second segment. Adding stacks to the second segment is done by first appending the new stack to the end of the array and then swapping it with the first element of the third segment. This places the new stack at the end of the second segment and the first element of the third segment is placed at the end. This is allowed since both the second and third segments are unordered. The same swapping trick can be applied a second time to add an element to the first segment. To maintain the order in the first segment the new stack is then sorted in-place.

This segmentation has one other advantage, namely in the scenario in `REDUCER` where all processed active stacks have to be considered. When using separate stack collections, these stacks are found by iterating over all active stacks and then checking if they are not contained in either `for-actor` or `for-actor-delayed`. These stacks exactly correspond to the third segment, allowing for faster retrieval of these stacks.

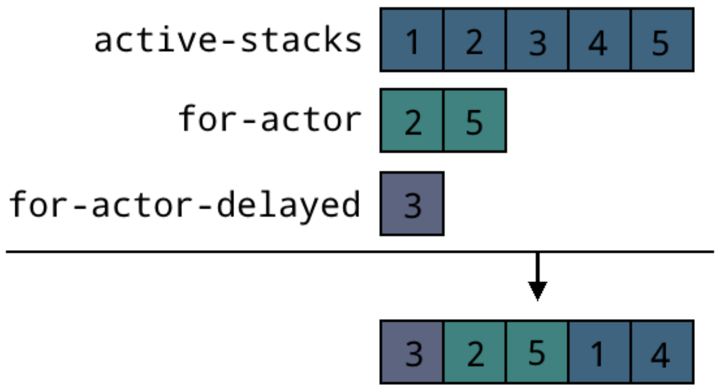

Figure 2: An example of the three segments for the given stack collections.

4.2 Finding Stack Paths

The following few optimizations aim to improve the performance of finding paths in the stack structured graph. All paths are first collected before being processed. This is done since subsequent calls to `REDUCER` alter the graph.

4.2.1 Finding Paths with Through-Link

In `DO-LIMITED-REDUCTIONS`, SGLR needs to find paths that go through a specific link. In JSGLR2 this is done by first finding all applicable paths, using the same method used for `DO-REDUCTIONS`. Only afterward does it filter the paths that do not contain the link. This means that JSGLR2 does extra work since it traverses the paths a second time to check if they contain the through-link. Furthermore, it potentially stores unused paths.

This can be improved by keeping track of whether the current path contains the through-link while finding paths. Paths are then only stored if they contain the through-link. This way it only has to traverse each path a single time and no unused paths are stored. This is implemented as a different method than the one used in `DO-REDUCTIONS`.

4.2.2 Skipping the Path Tree

JSGLR2 stores all found paths as a single tree, where each path corresponds to a leaf in the tree. The child parse nodes of a specific path are obtained by traversing the tree from the given leaf to the root. The children are collected in an array and then consumed by `REDUCER` for further processing.

The creation of this intermediate tree can be completely skipped by directly creating the arrays needed for `REDUCER`. This is implemented by keeping track of the current path in an array while finding paths. Each time a link is visited, its parse node is inserted at its corresponding position in this array. When the end of a path is found, the array contains all its children, which is then copied and stored.

4.2.3 Collecting Paths

JSGLR2 uses an ArrayList to collect the found paths, which is newly instantiated for each `DO-REDUCTIONS` and `DO-LIMITED-REDUCTIONS` call. Afterward, the paths are added one by one. The JSGLR2 measurements have been extended to tune this collection. The new measurements are `pathsAdds` and `pathsMaxSize`, recording the total amount of add calls and the maximum size reached. As can be seen in Appendix B, there are a significant amount of `add` calls. The maximum size is however relatively small.

An improvement is to re-use the same collection between `DO-REDUCTIONS` calls. This removes the need to allocate and grow memory for each call, which is instead done by clearing the collection. The same collection can not be re-used for `DO-LIMITED-REDUCTIONS` since it is called recursively. The implementation extends the optimization from subsection 4.2.1 to differentiate between `DO-REDUCTIONS` and `DO-LIMITED-REDUCTIONS` calls. Additionally using an ArrayDeque is considered over an ArrayList.

4.3 Limited Reductions for Source Stacks

In the third scenario in `REDUCER` as described in Section 2, SGLR needs to reconsider already processed active stacks due to the newly created link. This is the case since the stack to which the link is added might have parents, resulting in new possible reduction paths for those parents. Due to the graph being directed, it is not trivial to find these parents which considerably impacts performance.

This can be improved by keeping track of which active stacks are sources, i.e., stacks without parents). For these stacks, no other stacks would need to be reconsidered. Keeping track of source active stacks is relatively simple. The only way a parent can be added to an active stack is in the first scenario in `REDUCER`. Here, stacks are added to a collection of non-source stacks. Then in the third scenario, it is checked if the stack is contained in this collection. If this is the case, the slow method has to be used. Otherwise, the new improved method can be used. This collection is cleared at the start of each parse round in `PARSE-CHARACTER`.

4.4 Simultaneous Parse Node Visiting

After a successful parse, JSGLR2 performs several checks on the resulting parse tree. It does so by visiting its parse nodes in pre-order and post-order. Each check is performed in a separate visit, where appropriate error and warning messages are generated along the way. Additionally, some checks do not visit the children of specific parse nodes based on certain stopping conditions.

This can be improved by performing all checks in a single visit so that the parse tree is only traversed once. The implementation keeps track of which checks are currently active to handle the unique stopping conditions for each check. For this, two different types of stopping conditions are considered. The first is a hard stopping condition, which skips visiting the child parse nodes for all checks. The second is a soft stopping condition, which only disables the current check, still allowing the other checks to visit the children. The check is then re-enabled when the corresponding parse node is post-visited.

5 Results

This section provides the benchmark setup and results for the previously described optimizations. The figures show comparisons of the optimizations to their original implementation. Complete benchmark data including the different parser variants implemented in JSGLR2 [9, 1, 14] can be found in Appendix C.

5.1 Setup

All benchmarks are performed using the open-source evaluation suite for JSGLR2[4]. The evaluation suite allows simple configuration of which parser variants, languages, and source files should be evaluated. The optimizations are evaluated by batch processing source files, including the imploding of parse trees. The source files are obtained from open-source projects from three different languages: Java, WebDSL [4] and SDF3 [2]. The complete sources corpus for each language can be found in Appendix A.

Internally, the evaluation suite uses the Java Microbenchmark Harness[5] (JMH) to execute the benchmarks. JMH first performs a specified number of warm-up iterations, allowing time for the JVM to perform compiler optimizations. Afterward, it runs the given benchmark for a specified number of iterations. In this case, the benchmarks were run with 10 warm-up and 10 benchmark iterations.

The figures in the remainder of this section are generated on a machine with an Intel Core i7 processor. To improve consistency, the processor was set to a constant clock speed of 2.8 GHz, with Intel Turbo Boost and Hyper Threading disabled. In total 8 GB DDR4 RAM was available to the processor. The operating system used was Manjaro Xfce with kernel version 5.10.36-2-manjaro. Furthermore, OpenJDK version 1.8.0_292 and Apache Maven version 3.8.1 were used. Other processes were reduced as much as possible while running the benchmarks.

To obtain consistent results across different runs of the evaluation suite, the JVM JIT compiler was serialized with the application. This is done by providing the -XBatch command line option to the JVM, which is an option specific to the HotSpot JVM. Normally the compiler would run in a separate thread, which would result in the compiler optimizing different functions each run depending on the OS's thread scheduler.

[4]`https://github.com/jussyDr/jsglr2evaluation`
[5]`https://openjdk.java.net/projects/code-tools/jmh/`

5.2 Merging Stack Collections.

As shown in Figure 3, this optimization resulted in an overall performance improvement. Due to its implementation, it is not clear how much each aspect of the optimization contributed to this improvement. It could be the case that only the faster retrieval of processed active stacks or only the eliminated need to copy and remove elements lead to this improvement.

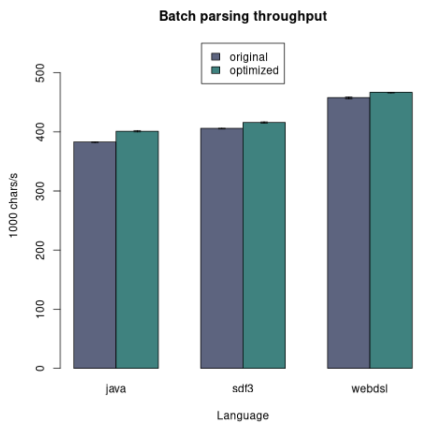

Figure 3: Merging stack collections

5.3 Finding Paths with Through-Link

Figure 4 shows that this optimization increased the performance for all tested languages.

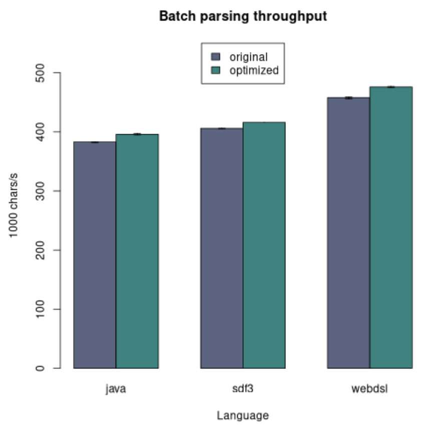

Figure 4: Finding paths with through-link.

5.4 Skipping the Path Tree

As is apparent in the left plot of Figure 5, directly creating the child arrays of paths degrades the performance. A possible explanation of why this happens is that the collection of a path's children now happens while finding paths. This means that in the case that the paths need to be filtered based on a through-link, it potentially spends time collecting the children for unused paths. Originally this happened only after the paths had been filtered.

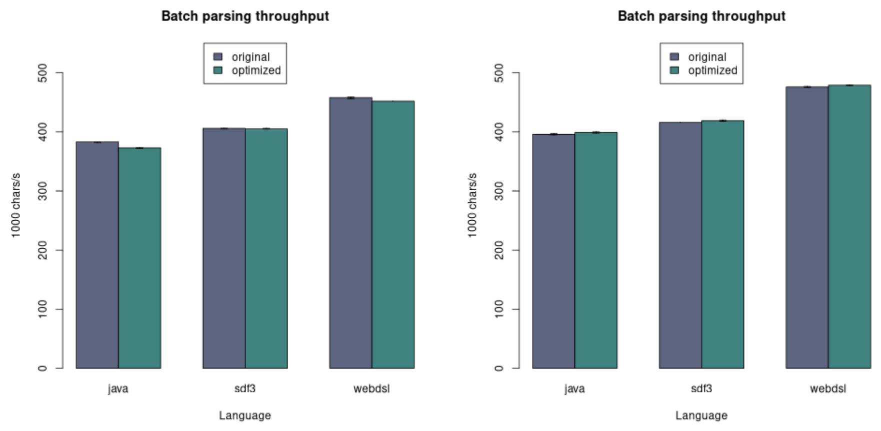

Figure 5: Skipping the path tree (left) applied to finding paths with through-link (right).

By applying the optimization from Section 5.3 this can be negated. This way the paths are already filtered while finding the paths, avoiding the collection of children for unused paths. A comparison with the optimization from Section 5.3 is shown in the right plot of Figure 5. As can be seen, this resulted in a very minor additional improvement.

5.5 Collecting Paths

As shown in Figure 6, both re-using the paths collection and using an ArrayDeque resulted in performance improvements. The right plot is compared with the optimization from Section 5.3. The improvement by using an ArrayDeque instead of an ArrayList is likely due to a more efficient add operation.

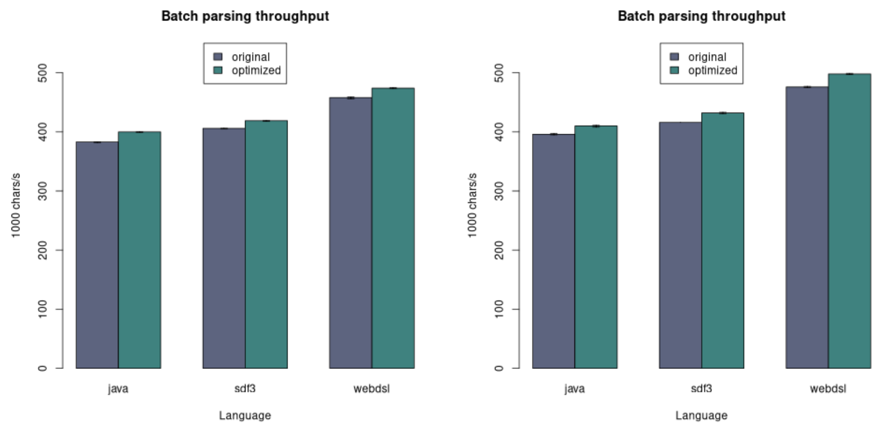

Figure 6: Re-using the paths collection (left) represented as an ArrayDeque (right).

5.6 Limited Reductions for Source Stacks

This optimization resulted in a significant performance improvement as can be seen in Figure 7. This shows that the newly created links are often added to source stacks for the tested languages.

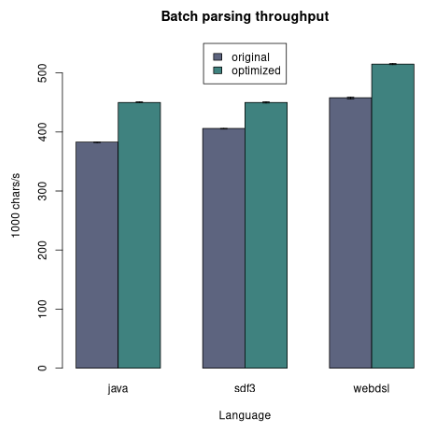

Figure 7: Limited reductions for source stacks.

5.7 Simultaneous Parse Node Visiting

Figure 8 shows that this optimization has a positive performance improvement. Only the checks that are used for all parser variants are considered. Further possible improvements for the recovery variant [1] can be achieved by including checks that are unique to this variant.

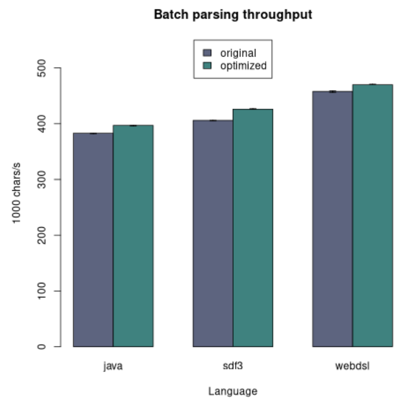

Figure 8: Simultaneous parse node visiting.

5.8 Combined Improvement

Figure 9 show the performance of all optimizations combined. The improvement is less than the sum of individual improvements since some optimizations interfere with each other. Depending on the language, a speed-up of 9% up to 20% is achieved for the Elkhound parser variant and a speed-up of 26% up to 44% for the other parser variants. This performance difference is because most optimizations are in the non-deterministic part of the parser, the usage of which is reduced in the Elkhound parser variant.

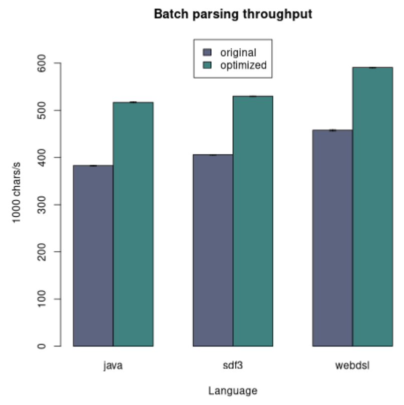

Figure 9: Combined improvement.

6 Responsible Research

The reproducibility of results is important to validate the research. Two main aspects impact the reproducibility: The applied modifications and the evaluation. All modifications are described in detail in Section 4 and are publicly available[6], enabling other parties to replicate them. The evaluation setup is detailed in Section 5.1. All components of the evaluation are publicly available, again enabling other parties to replicate the exact evaluation.

7 Conclusion

This paper presented several performance optimizations for JSGLR2 to improve its practicality. Depending on the language, a speed-up of 9% up to 20% is achieved for the Elkhound parser variant and a speed-up of 26% up to 44% for the other parser variants. JSGLR2's modular architecture in general simplified implementing optimizations by being able to optimize all variants with a single modification. However, some optimizations required extra work to conform to the architecture.

7.1 Future Work

JSGLR2's modular architecture likely introduces a performance overhead. This overhead can be avoided by applying feature-oriented software development (FOSD). Using FOSD, non-modular parser variants are automatically generated based on a modular implementation.

While this work focused on optimizations that improve performance, possible optimizations can be discovered that improve the memory usage of JSGLR2. An example would be to use primitive java collections when applicable. Default generic Java collections box primitive values into an object, which introduces additional overhead. This boxing is avoided by using primitive collections. An example where this could be applied is when querying the goto state from the parse table, which is currently done using an `Integer` to `Integer` map.

[6]`https://github.com/jussyDr/jsglr`

References

[1] Maartje de Jonge, Lennart C. L. Kats, Eelco Visser, and Emma Söderberg. Natural and Flexible Error Recovery for Generated Modular Language Environments. *ACM Transactions on Programming Languages and Systems*, 34(4):15, 2012. URL: `http://doi.acm.org/10.1145/2400676.2400678`.

[2] Luis Eduardo de Souza Amorim and Eelco Visser. Multi-purpose Syntax Definition with SDF3. In Frank S. de Boer and Antonio Cerone, editors, *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings*, volume 12310 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2020. URL: `https://doi.org/10.1007/978-3-030-58768-0_1`.

[3] Jasper Denkers. A Modular SGLR Parsing Architecture for Systematic Performance Optimization. Master's thesis, Delft University of Technology, 2018. URL: `http://resolver.tudelft.nl/uuid:7d9f9bcc-117c-4617-860a-4e3e0bbc8988`.

[4] Danny M. Groenewegen, Zef Hemel, Lennart C. L. Kats, and Eelco Visser. WebDSL: a domain-specific language for dynamic web applications. In Gail E. Harris, editor, *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA*, pages 779–780. ACM, 2008. URL: `http://doi.acm.org/10.1145/1449814.1449858`.

[5] Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 918–932, Reno/Tahoe, Nevada, 2010. ACM. URL: `http://doi.acm.org/10.1145/1869459.1869535`.

[6] James R. Kipps. *GLR Parsing in Time O(n3)*, pages 43–59. Springer, Boston, MA, USA, 1991. URL: `https://doi.org/10.1007/978-1-4615-4034-2_4`.

[7] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965. URL: `https://doi.org/10.1016/S0019-9958(65)90426-2`.

[8] Bernard Lang. Deterministic Techniques for Efficient Non-Deterministic Parsers. In Jacques Loeckx, editor, *Automata, Languages and Programming*, pages 255–269, Berlin, Heidelberg, 1974. Springer.

[9] Scott McPeak and George C. Necula. Elkhound: A Fast, Practical GLR Parser Generator. In Evelyn Duesterwald, editor, *Compiler Construction*, pages 73–88, Berlin Heidelberg, 2004. Springer.

[10] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. volume 49, pages 579–598, 2014. URL: `https://dl.acm.org/doi/10.1145/2714064.2660202`.

[11] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) Parsing of Programming Languages. PLDI '89, page 170–178, New York, NY, USA, 1989. Association for Computing Machinery. URL: `https://doi.org/10.1145/73141.74833`.

[12] Masaru Tomita. Graph-structured Stack and Natural Language Parsing. In *26th Annual Meeting of the Association for Computational Linguistics*, pages 249–257, Buffalo, New York, USA, 1988. Association for Computational Linguistics. URL: `https://www.aclweb.org/anthology/P88-1031`.

[13] Eelco Visser. Scannerless Generalized-LR Parsing. Technical report, Programming Research Group, University of Amsterdam, 1997. URL: `https://eelcovisser.org/publications/1997/Visser97-SGLR.pdf`.

[14] Tim A. Wagner and Susan L. Graham. Incremental Analysis of Real Programming Languages. *SIGPLAN Not.*, 32(5):31–43, 1997. URL: `https://doi.org/10.1145/258916.258920`.

A Test sets

This section show the complete test set for each language used for the measurements and benchmarks. Each source project is limited to thirty files to speed up the evaluation for large projects.

Language	Source	Files	Lines	Size (bytes)
Java	apache-commons-lang	30	6626	271354
	netty	30	3609	130151
WebDSL	webdsl-yellowgrass	30	3665	100600
SDF3	nabl	30	1175	27736
	dynsem	4	454	10248
	flowspec	20	685	14154

Table 1: Test sets.

B Measurements

This section show a subset of the parsing measurements in JSGLR2 for the standard parser variant. The extended measurements for collecting paths are also included.

Language	Java	WebDSL	SDF3
characters	401501	100600	52138
activeStacksAdds	2163018	514594	322731
activeStacksMaxSize	75	57	40
activeStacksIsEmptyChecks	803182	201290	104438
activeStacksFindsWithState	2476014	571677	368041
activeStacksForLimitedReductions	198914	41168	33463
activeStacksAddAllTo	401561	100630	52192
activeStacksClears	401561	100630	52192
forActorAdds	2140308	500853	312492
forActorDelayedAdds	22710	13741	10239
forActorMaxSize	33	13	8
forActorDelayedMaxSize	1	4	4
forActorContainsChecks	1611485	257596	212655
forActorNonEmptyChecks	2564579	615224	374923
actors	2158441	513581	322403
doReductions	1636543	386465	227546
doLimitedReductions	695080	111328	80146
reducers	1917282	444890	278719
pathsAdds	4559872	794224	480211
pathsMaxSize	33	13	13

Table 2: Parsing measurements.

C Benchmarks

This section shows the batch parsing throughput results from the benchmarks. The results are normalized to a thousand characters per second. The other parser variants implemented in JSGLR2 are also included.

Variant	Java			WebDSL			SDF3		
	Score	Low	High	Score	Low	High	Score	Low	High
standard	383	382	383	458	456	459	406	405	406
elkhound	497	495	498	587	586	588	491	490	491
recovery	326	316	336	385	385	386	318	317	318
incremental	277	265	290	340	340	341	290	290	291

Table 3: Original implementation.

Variant	Java			WebDSL			SDF3		
	Score	Low	High	Score	Low	High	Score	Low	High
standard	397	397	398	465	465	465	404	404	405
elkhound	520	519	521	602	601	602	504	504	505
recovery	352	345	359	414	413	414	358	358	359
incremental	293	281	305	347	347	348	290	289	290

Table 4: Merging stack collections.

Variant	Java			WebDSL			SDF3		
	Score	Low	High	Score	Low	High	Score	Low	High
standard	396	395	397	476	475	477	416	416	416
elkhound	511	511	512	594	594	594	509	509	510
recovery	346	337	355	400	400	400	334	333	334
incremental	315	302	329	380	380	381	319	318	319

Table 5: Finding paths with through-link.

Variant	Java			WebDSL			SDF3		
	Score	Low	High	Score	Low	High	Score	Low	High
standard	373	372	373	452	452	452	405	405	406
elkhound	488	487	489	583	583	584	495	494	496
recovery	316	307	326	374	374	374	339	338	339
incremental	270	258	283	334	334	334	281	281	282

Table 6: Skipping the path tree.

Variant	Java			WebDSL			SDF3		
	Score	Low	High	Score	Low	High	Score	Low	High
standard	399	398	400	479	478	479	419	418	420
elkhound	520	519	522	597	597	598	518	517	519
recovery	344	335	355	391	390	391	353	354	352
incremental	331	317	347	379	379	379	325	325	326

Table 7: Skipping the path tree applied to finding paths with through-link.

Variant	Java			WebDSL			SDF3		
	Score	Low	High	Score	Low	High	Score	Low	High
standard	410	408	411	498	497	499	432	431	433
elkhound	513	512	514	589	587	590	513	512	513
recovery	354	344	364	415	415	416	344	344	345
incremental	322	310	336	392	392	392	329	328	330

Table 8: Re-using the paths collection.

Variant	Java			WebDSL			SDF3		
	Score	Low	High	Score	Low	High	Score	Low	High
standard	400	399	400	474	473	475	419	418	419
elkhound	506	505	506	594	593	595	503	503	503
recovery	340	332	350	399	395	403	331	330	331
incremental	287	274	301	349	349	349	294	294	295

Table 9: Collecting paths using an ArrayDeque.

Variant	Java			WebDSL			SDF3		
	Score	Low	High	Score	Low	High	Score	Low	High
standard	450	450	451	515	514	516	450	449	451
elkhound	543	542	544	604	603	605	520	519	520
recovery	366	366	367	433	432	433	373	372	373
incremental	364	361	367	406	405	407	343	343	344

Table 10: Limited reductions for source stack nodes.

Variant	Java			WebDSL			SDF3		
	Score	Low	High	Score	Low	High	Score	Low	High
standard	396	396	397	471	470	471	425	424	425
elkhound	524	523	525	614	613	615	528	527	529
recovery	336	326	346	399	398	400	333	333	333
incremental	286	274	299	348	348	349	299	298	300

Table 11: Simultaneous parse node visiting.

Variant	Java			WebDSL			SDF3		
	Score	Low	High	Score	Low	High	Score	Low	High
standard	517	516	518	591	590	591	530	529	530
elkhound	571	564	578	638	637	639	590	590	590
recovery	452	449	455	484	484	484	446	446	447
incremental	399	395	403	443	442	444	389	388	390

Table 12: All optimizations combined.