

# Compiler Optimizations and Source Code Analysis

Summary of the Ph.D. Dissertation

by  
**Gábor Lóki**

Supervisor:  
Dr. Ákos Kiss

Doctoral School of Informatics  
Department of Software Engineering  
Faculty of Science and Informatics  
University of Szeged



Szeged  
2023



# Introduction

In the world of modern computing, the need for speed, efficiency, and resource utilization is ever-increasing. As a result, the importance of compiler optimizations is becoming increasingly evident in the software development process. Compilers are essential tools that convert human-readable source code into machine-executable binary code, thus bridging the gap between the programmer's intent and efficient execution. They are a set of techniques that refine and improve the generated code and are essential for achieving optimal program performance, resource efficiency, and code quality.

The complexity of software has been steadily increasing, as shown by the exponential growth in the size of codebases and the requirements for software applications. This has led to a change in the traditional view of compilers. Compiler optimizations are a range of strategies used to reduce bottlenecks, improve algorithms, reduce memory usage, and make the most of hardware resources. The changing requirements go beyond the usual code generation, requiring the compiler toolchains to identify additional opportunities in the codebases. Thus, the compiler technologies, used in optimizations become more valuable.

The use of compiler technologies is widespread in many areas of software development, from aiding resource-hungry systems to improving software quality, and even helping with software visualization. Its areas of use are also large, from embedded systems to scientific simulations, from real-time applications to high-performance computing.

In the first part of this thesis, the main goal is to introduce efficient code size optimization algorithms for one of the most well-known compilers, for the GCC, the GNU Compiler Collection. In addition, a reliable code size measurement method and a stable benchmark environment are presented.

In the second part, the main goal is to examine and validate JavaScript guidelines, presented on various web pages, that can improve the efficiency of JavaScript runtime performance. In addition, one approach of JavaScript's source code analysis is presented that can aid other optimizations or analyses to rely on call graph information.

Between these two main parts of the thesis, the author states four main results as listed below:

1. Implementation and evaluation of different code factoring algorithms in GCC compiler.
2. Introduce binary code size measurement methods and benchmark for compiler's code size optimizations.
3. Collect, analyze, and evaluate JavaScript guidelines.
4. Introduce a dynamic JavaScript call graph generator and evaluate other call graph generators.

In the rest of the booklet, we summarize the results for each thesis point.

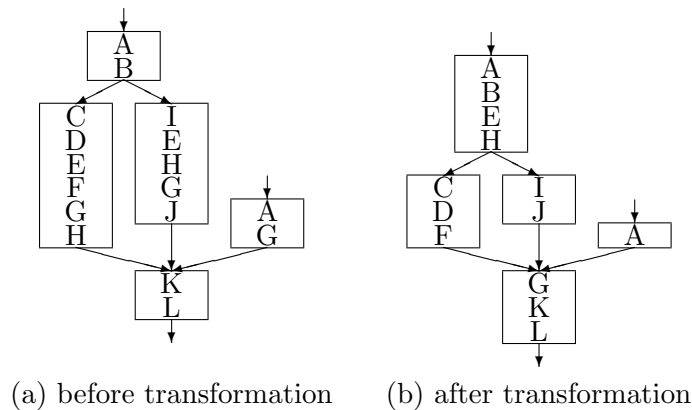
# I. Executable Code Optimizations

In this thesis group, the main goal is to introduce efficient code size optimizing algorithms for one of the most well-known compilers, for the GCC, the GNU Compiler Collection. In addition, a reliable code size measurement method and a stable benchmark environment are presented.

Code factoring is a class of useful optimization techniques that have been specifically developed to reduce code size. These approaches are aimed at reducing size by redefining the code. There are two main code factoring algorithms that are in the focus, one for individual instructions and the other for longer instruction sequences.

## 1. Code Factoring Techniques in GCC Compiler

**Local Code Factoring** is the one that deals with individual instructions. The optimization strategy of local factoring (also known as local code motion, code hoisting, and code sinking) is to move identical instructions from the basic blocks to their common predecessor or successor in the CFG, if they exist. Of course, the semantics of the program have to be preserved, so only those instructions that do not invalidate any existing data dependencies or introduce new ones can be moved. Figure 1a shows a control flow graph (CFG) with basic blocks containing identical instructions. To achieve the best size reduction, some of the instructions are moved upwards to the common predecessor, while others are moved downwards to the common successor. Figure 1b shows the result of the transformation. It should be noted that for the sake of simpler representation, the identical letters denote identical instructions, and jump or branch instructions are omitted from these kinds of CFGs.

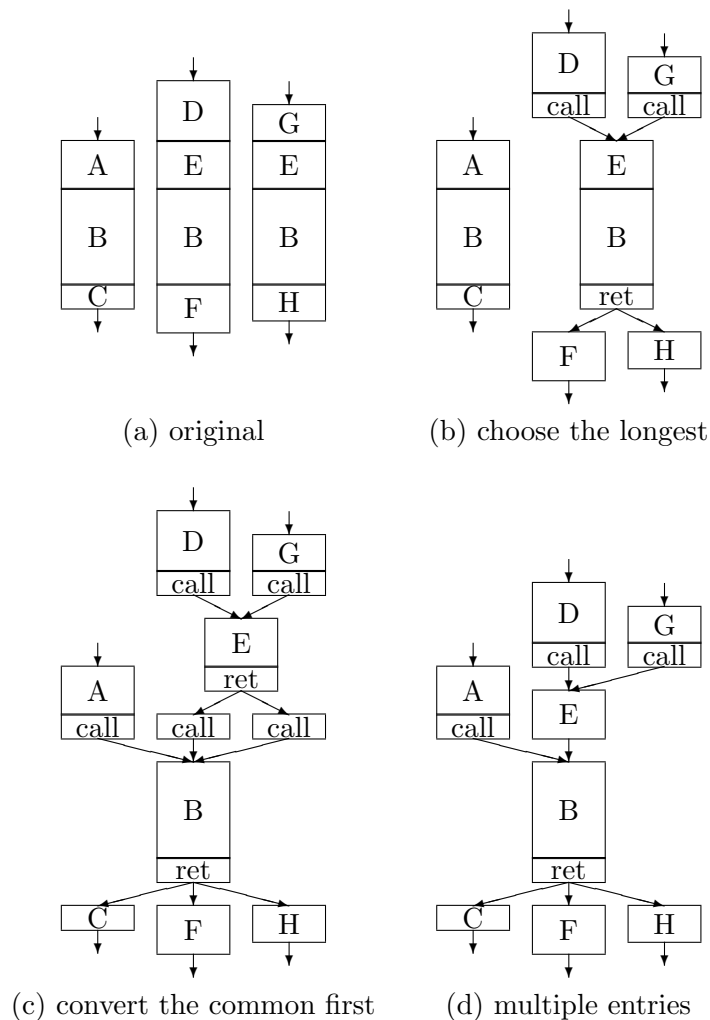


**Figure 1:** *The effect of local code factoring on the CFG*

Although it is not frequent, it may happen that multiple basic blocks have multiple predecessors, all of which are common. In this case, if the underlying basic blocks in question have identical instructions and the number of predecessors is lower than the number of examined blocks, then the instructions are moved to all the predecessors. A similar situation occurs in which the basic blocks have more than one common successor. Furthermore, in the case of sinking, even those instructions that are not present in all of the blocks can be moved by creating a new successor block for them.

**Procedural Abstraction** is a size optimization method that works with single-entry single-exit code regions (like instruction sequences smaller than a basic block, complete blocks, or even larger units) rather than single instructions only, unlike local factoring. The main idea of this technique is to find such code regions that can be converted to procedures and replace all occurrences with calls to newly created subroutines.

Existing solutions can only handle such code fragments that are identical or equivalent, or can be converted to equivalent forms in some way (e.g. through register renaming). However, these methods do not find an optimal solution for the cases where one sequence of instructions is identical to the candidate sequence, while the third sequence is identical only with its suffix (as shown in Figure 2a). These solutions can be used to abstract the longest possible sequence into a function and leave the shorter sequence unabstracted (Figure 2b) or to convert the common instructions in all sequences into a function and create another new function from the rest of the common parts of the long sequence, introducing the overhead of adding additional calls and return code (Figure 2c).



**Figure 2:** *Different kinds of procedural abstraction strategies*

We proposed to create *multiple entry subroutine* that allows the abstraction of instruction sequences of different lengths without the overhead of superfluous call/return code. The

longest possible sequence shall be chosen as the body of the new function, and entry points must be defined according to the length of the matching sequences. Figure 2d shows the optimal solution to the problem indicated in Figure 2a.

Both of the new algorithms have been implemented in different intermediate languages (IL) as new GCC optimization phases. First, it has been done in the *Register Transfer Language* (*RTL*) IL, after the *Tree-SSA* (based on Static Single Assignment technique) version was carried out. Finally, a more abstract approach was implemented on the *GENERIC* IL level, introduced in the *Interprocedural Abstraction Analysis* (*IPA*) optimization phase. The proposed code factoring algorithms are publicly available in the CFO branch.

**Hash tables.** One of the improvements that we introduced during the implementation of these algorithms was the use of hash tables to compare instructions. Most GCC optimization algorithms are subject to the calculation of  $O(n^2)$  while comparing a candidate to another one. Since they compare every candidate to all other ones to find an exact match or similarity. Usually, developers come up with special filters and algorithm-related tricks to speed up computation time. These solutions are also usable and can solve the slow compilation time problem, but are designed and implemented at each algorithm level separately. Since we also faced this problem in the implementation of our algorithms and had not found a general way to solve them, we introduced a common way to compare instructions. Thus, we implemented a hash table-based instruction cache. Our first public implementation of the instruction hash table, or cache, proved to be very effective, although we did not provide any special solution for the hash collisions. We introduced three hash functions, one for each intermediate level. Thus, the complexity of our candidate search in algorithms became  $O(n \log n)$ . This was a huge improvement, especially for large compilation units.

**Results.** When examining the size of the code generated by the compiler, we found that the algorithms of code factoring had significant effects on several tests. We evaluated the algorithms with the help of *CSiBE*, the GCC’s Code Size Benchmark Environment, on three different targets (*i686-elf*, *arm-elf*, *sh-elf*) and found that a maximum of 61.53% and an average of 2.58% of extra code size savings could be achieved compared to the GCC flag ‘-Os’. The detailed results are presented in Tables 1 and 2, where the binary size in bytes and the relative improvement to ‘-Os’ in percentage can be seen.

flags	i686-elf		arm-elf		sh-elf	
	size (byte)	reduction (rel. %)	size (byte)	reduction (rel. %)	size (byte)	reduction (rel. %)
-Os	2,900,177		3,636,462		3,184,258	
-Os -ftree-lfact -frtl-lfact	2,892,432	0.27	3,627,070	0.26	3,176,494	0.24
-Os -frtl-lfact	2,894,531	0.19	3,632,454	0.11	3,180,186	0.13
-Os -ftree-lfact	2,897,382	0.10	3,630,378	0.17	3,179,622	0.15
-Os -ftree-seqabstr -frtl-seqabstr	2,855,823	1.53	3,580,846	1.53	3,149,822	1.08
-Os -frtl-seqabstr	2,856,816	1.50	3,599,862	1.01	3,162,678	0.68
-Os -ftree-seqabstr	2,888,833	0.39	3,610,002	0.73	3,166,054	0.57
-Os -fipa-procabstr	2,886,632	0.47	3,599,042	1.03	3,160,626	0.74
All	2,838,348	2.13	3,542,506	2.58	3,123,398	1.91

**Table 1:** Code-size reduction with code factoring algorithms.

## 2. Binary Code Size Measurement Methods and Benchmark

The measurement of the size of the generated code (i.e. its compactness) is not always trivial. Most of the compilers produce assembly code, after that the *assembler* tool provides the executable binary. So, using the assembly code would be one option to measure the generated code size, but we suggest a different approach. If we recall that the final goal is to reduce the size of the entire software, we must examine several parts of the program that the compiler might influence. The binary objects and executables are the trivial parts of the software that the compiler has an effect on. In addition, it must be carefully examined how the connected libraries can affect the code generation. Finally, the different parts of the binaries (e.g. sections) should be analyzed in order to include or exclude them from the measurements. Thus, it is a research question as to which part of the software should be measured and how.

**Binary objects and executables.** The granularity of the code is an important aspect: Should we measure the size of functions individually, the object code of a complete compilation unit, or investigate the size of the linked executable? For the first option, it is possible to compile one function at a time (compilers used to have a flag for such a case). This approach is very similar to the second option of the previous question, but the function-at-a-time compilation might miss possible optimizations because of this granularity. When comparing the object sizes (compilation unit granularity), the effectiveness of a given compiler is investigated, while in the last option, the entire compiler toolchain is evaluated, including the compiler, linker, and libraries. This is because the size of the linked program also depends on the size of the libraries and the way the linker processes them. Therefore, here we rely mainly on comparing objects that are more informative concerning the optimization potential of a compiler for space.

**Standalone and Linux programs.** Another dimension of the categorization we investigated was two types of targets: standalone executables (i.e. without an operating system) and executables built for a particular operating system (in our case GNU/Linux). Even if the same compiler is used with the same settings, the resulting binaries usually contain several notable differences: some for objects and some for executables. These are mainly due to the different executable production and the different runtime libraries used in these cases (GCC, newlib, and glibc).

flags	i686-elf	arm-elf	sh-elf
	max. reduction (%)	max. reduction (%)	max. reduction (%)
-Os -ftree-lfact -frtl-lfact	6.13	10.98	10.29
-Os -frtl-lfact	4.31	3.51	4.35
-Os -ftree-lfact	5.75	10.34	8.78
-Os -ftree-seqabstr -frtl-seqabstr	36.81	56.92	43.89
-Os -frtl-seqabstr	30.67	45.69	42.45
-Os -ftree-seqabstr	30.60	41.60	44.72
-Os -fipa-procabstr	38.21	56.32	59.29
All	57.05	61.53	60.17

**Table 2:** *Maximum code-size reduction results for CSiBE objects.*

**Sections.** Another problem was to determine which parts of the generated files to take into account (e.g., the size of the binary file, printed by a file manager, is irrelevant due to various file format headers). The generated program code consists of many parts, such as instructions, data, etc., usually separated in a binary file (so-called in sections). However, in many cases, these parts can be mixed (e.g. executable code can embed data). Furthermore, other custom sections are usually placed in binary files and are not mature in terms of code size. These include debug sections, symbol tables, etc.

**Measurement tools.** When assessing both objects and executable sizes, it was necessary to investigate *elf* and *coff* files (they were the most important file formats in the 2000s). As a consequence, different methods were used to extract section sizes due to different binary formats. The program *size* (part of *binutils*) is an appropriate tool to extract the size of specified sections from *elf* files. For *coff* files the *coffdump* tool can be used.

**Execution and testing.** Correctness and validation are also important features. In this field, we should ensure that the compiled executable binaries provide the expected results. Therefore, a measurement environment should be able to execute the built programs.

**CSiBE.** The fundamentals of measuring code size have led to the creation of a prototype of a benchmark. During the discussions with compiler developers and the evaluation of this prototype benchmark, we have released a useful benchmark that has become the official code size benchmark of GCC. The benchmark is called *CSiBE*, the Code Size BEnchmark.

This benchmark has been developed and maintained by the Department of Software Engineering at the University of Szeged in Hungary. Since its initial introduction, *CSiBE* has been used by GCC developers in their daily work to help minimize the size of the generated code. Moreover, the latest results are continuously monitored, and the GCC developers are informed about any code size-related issues, should any occur.

Around the *CSiBE* benchmark, there is a complete framework, which was a variant of a SaaS (Software as a Service) system. We simply call it the *CSiBE system*. The whole system is designed, implemented, and maintained by us.

The *CSiBE system* consists of two main components. The front-end server is used to download daily GCC snapshots and generate raw measurement data. The back-end server acts as a data server by filling the relational database with measurement data and is also responsible for transmitting data to the user via a web interface.

The core of the *CSiBE system* is the offline *CSiBE* benchmark, which consists of the testbed and the required measurement scripts. The package can be downloaded from the official website and can also be used independently of the online system. The testbed consists of 18 projects and the source size is about 50 MB. When compiling, the total amount of binary code is about 3.5 MB. Various types of programs, such as codecs (gsm, mpeg), compilers, compressors, editor programs, and preprocessed units, have been adopted. Some projects are also suitable for measuring performance and constitute about 40% of the testbed. We have also added some Linux kernel sources to the v2.1.1 version of the testbed. Taking the original goal into account, we started with the *S390 platform* and turned it into a so-called *test platform*. On this platform, we replaced all assembly code with code stubs, leaving only C code for important Linux modules (kernel, device, file system, etc.).



**Results.** Since the start of the first code size measurement in 2003, the *CSiBE* benchmark has made a lot of progress. It took less than a year and became the official code size benchmark for GCC. Interest in code size has become a central issue in the world of compilers with new vigor. This encouraged the improvement of *CSiBE*. From the first public version v1.0.1 on 2003-08-11 to the most well-known version v1.1.1 on 2004-02-20, the benchmark environment took its present structure. As the years went by and software became more and more complex, *CSiBE* adopted new projects to track various types of programs. The latest official version is *CSiBE* v2.1.1, released on August 15, 2015, but the benchmark is constantly evolving. Its Github page now contains even more complex tests, from the Common Microcontroller Software Interface Standard (CMSIS) to Servo, the parallel browser engine. These projects ensure that the importance of code size is taken into account in compilers.

In addition to the GCC compiler, another compiler, LLVM Compiler Infrastructure also uses *CSiBE* in their development workflows. After presenting the advantages of *CSiBE* to their developer community, they began demonstrating improvements in code size with *CSiBE*.

## The Author's Contributions

The author had a decisive role in the design, implementation, improvement, and maintenance of a significant proportion of the algorithms presented in the “Code Factoring Techniques in GCC Compiler” section:

- Local Code Factoring: The author designed, implemented, and maintained the RTL version while improving and maintaining the Tree-SSA version.
- Procedural Abstraction: The author designed, implemented, and maintained the Tree-SSA version while improving and maintaining the RTL version of it. In addition, the IPA version of it has been designed, improved, and maintained by the author.
- Hash tables: This technique was introduced for optimization algorithms by the author.

The author had a decisive role in the design, implementation, and improvement phases of the code size measurement techniques and the evaluation environment for compilers' binary code optimizations presented in the “Binary Code Size Measurement Methods and Benchmark” section. Besides these, the author has been the main maintainer of the official code size benchmark of GCC, *CSiBE*, since 2004.

The publications of the author, related to this thesis group, are the following:

- [3] Árpád Beszédes, Tamás Gergely, Tibor Gyimóthy, Gábor Lóki, and László Vidács. Optimizing for space: Measurements and possibilities for improvement. In *Proceedings of the 2003 GCC Developers' Summit*, pages 7–20, Ottawa, Canada, 2003
- [2] Árpád Beszédes, Rudolf Ferenc, Tamás Gergely, Tibor Gyimóthy, Gábor Lóki, and László Vidács. *CSiBE* benchmark: One year perspective and plans. In *Proceedings of the 2004 GCC Developers' Summit*, pages 7–15, Ottawa, Canada, 2004
- [9] Gábor Lóki, Ákos Kiss, Judit Jász, and Árpád Beszédes. Code Factoring in GCC. In *Proceedings of the 2004 GCC Developers' Summit*, pages 79–84, Ottawa, Canada, 2004
- [10] Csaba Nagy, Gábor Lóki, Árpád Beszédes, and Tibor Gyimóthy. Code factoring in GCC on different intermediate languages. In *Proceedings of the 10th Symposium on*

*Programming Languages and Software Tools*, pages 79–95, Dobogókő, Hungary, 2007. Eötvös Loránd University Press

- [11] Csaba Nagy, Gábor Lóki, Árpád Beszédes, and Tibor Gyimóthy. Code factoring in GCC on different intermediate languages. *Annales Universitatis Scientiarum Budapestinensis De Rolando Eötvös Nominatae Sectio Computatorica*, 30:79–95, 2009

## II. Just-In-Time Compilers’ Optimizations and Analyses

In this thesis group, the main goal is to examine and validate JavaScript guidelines, presented in various places on the Internet, that can improve the efficiency of JavaScript runtime performance. In addition, one approach of JavaScript’s source code analyses is presented that can aid other optimizations or analyses to rely on call graph information.

### 3. JavaScript Guidelines

Before the time of JIT engines, several guidelines were published on how to write efficient JavaScript code. In this chapter, our research focuses on whether programmers should still comply with these guidelines or can rely on JIT compilers to achieve good performance results as they do with classical compilers to generate optimal code in static languages such as C. We explore the effect of Just-In-Time compilation and programming guidelines on the performance of JavaScript execution. In addition, not only one but two variants of JavaScript standards have been evaluated to get a bigger picture of these guidelines.

**Legacy Guidelines.** In the early days of JavaScript, optimized ideas could only be found in guidelines distributed in various places on the Internet. These proposals were made on the basis of experience and dynamic measurements on small benchmarks. Furthermore, they typically apply only to a single JavaScript engine and only a specific version of it. Therefore, these optimization techniques are more of subjective experiences than objective analyses. We collect guidelines that have been proposed over the years, for various JavaScript engines and versions, to allow their systematic evaluation.

In our evaluation ECMAScript 5.1, the first high-impact version of JavaScript, and the follow-up version, the ECMAScript 6 version are examined. The legacy guidelines give the following suggestions:

- Use of local variables instead of global variables whenever possible.
- Globally initialized variables should be used, instead of constant-like complex local variables.
- Use a cache for object members in variables.
- Avoid *with* construct.
- Use the JavaScript Object Notation (JSON) instead of a function to hold *struct*-like data.
- Avoid *eval* construct.
- Do some function inlining which is a traditional compiler optimization technique.
- Do as many common subexpression eliminations (CSE) as possible. This is also a performance-oriented compiler optimization technique.

- Perform loop unrolling for short loop cycles.
- Replace post-increment and decrement operators of the loop index variable with the pre-increment and decrement variants.

**ECMAScript 6-based Guidelines.** The ECMAScript 6 standard was introduced in 2015. Since then, the main web browsers and JavaScript engines have adopted its features. This is also true for JavaScript engines that target embedded domains. Today, we can see a lot of support for ECMAScript 6 (and later) in the JavaScript engine world. The main purpose of introducing ECMAScript 6 was to improve functionality, bring JavaScript closer to other widespread languages, and facilitate the use of the language for every web developer. To validate and introduce possible new guidelines, we analyzed and evaluated features and components of ECMAScript 6. The goal is not only to introduce new guidelines that help developers improve the performance of applications but also to show how new language constructs affect performance compared to ECMAScript 5. Thus, the following ECMAScript 6 features have been examined:

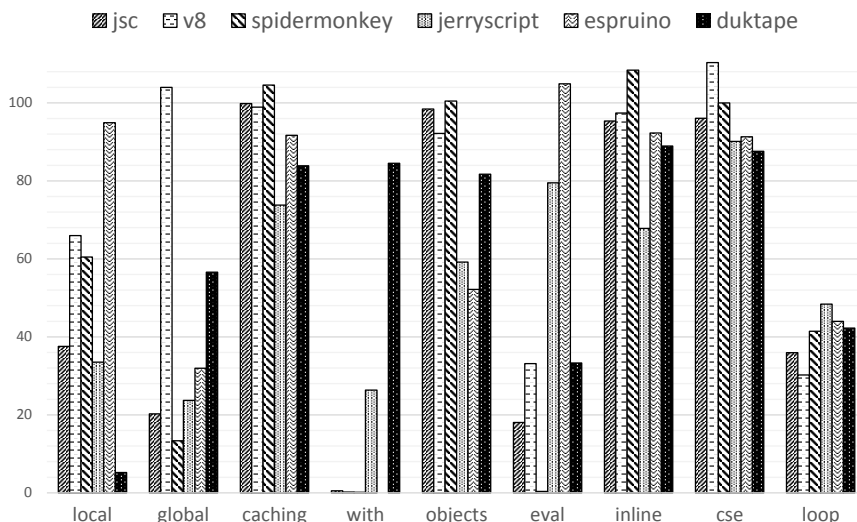
- The *arrow function* is a shorter syntax of a function expression and does not have its own *this*, arguments, or *super* constructs.
- In ECMAScript 5.1, a *class* is nothing but a somewhat specially written function. It has the same syntax as the function expressions and declarations.
- *Object literals* are extended to support setting the prototype for constructions, shorthand for assignments, defining methods, making super calls, and computing property names with expressions.
- *Template strings* provide easy-to-use syntax for creating different strings from previously defined templates.
- A more advanced form of template literals is the *tagged templates*. Tags allow one to parse template literals with the help of a function.
- In the JavaScript world, destructuring objects is a fail-soft action to unbind values from their container. In ECMAScript 6, this is the case when one unpacks values from arrays or properties from objects into distinct variables.
- The *spread operator*, which spreads the elements of an iterable collection (such as arrays or strings) into individual elements or function parameters.
- One of the most significant changes to ECMAScript 6 is that it is possible to create constant values with *const* keyword.
- The standard supports the *iterator* protocol to generate a value sequence and provides a convenient method to iterate all values of an iterable object.
- Many languages contain *generators* and *yield* constructions. The same functionality has been added to the ECMAScript 6 feature set.
- In ECMAScript 6, some effective data structures were introduced for common algorithms (e.g. Map, Set, WeakMap, WeakSet).
- In ECMAScript 6, there is a new feature called *Symbol*, a global symbol indexed by a unique key.
- The new standard has added support for describing binary and an alternative syntax for octal numbers with *binary literals*.

**Static Optimization difficulties.** It would be very convenient if these performance acceleration techniques did not need to be applied manually but could be implemented as

automatic code conversions, i.e. compiler optimizations. The experience with static languages shows that optimization algorithms are worthwhile to apply, since the cost of the technology is paid only during the compilation time, and the performance gains are considerable. Thus, the need for optimization algorithms increases naturally for dynamic languages, and the existing guidelines can serve as a natural starting point for the design of these techniques. However, as we shall see below, the language features of JavaScript render most of the static optimization techniques ineffective.

The main reason comes from the dynamic evaluation of JavaScript code, such as the *eval* function. These kinds of language features can alter the actual context by evaluating a string value as a JavaScript code runtime. In the view of static algorithms, these are unpredictable changes. Because compiler optimizations always have to be safe, these language features make the application of complex static optimization algorithms and automated programming guidelines to JavaScript practically infeasible.

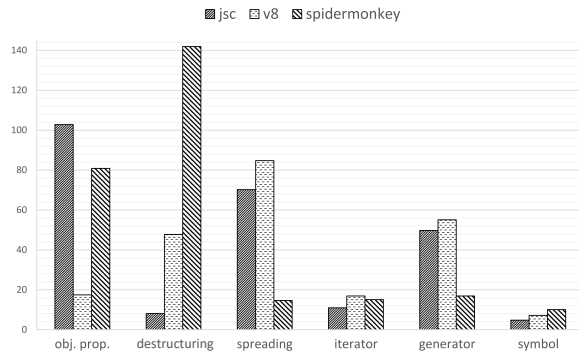
**Result.** Our target was to evaluate JavaScript guidelines to see how they are affecting the different engines. For each language construct and feature the runtime was measured with and without guidelines, and a resulting percentage was calculated.



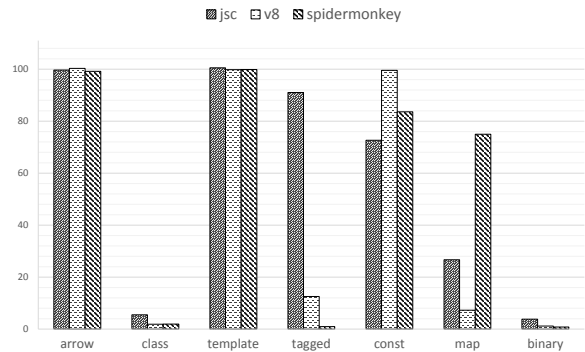
**Figure 3:** *Performance improvement with legacy guidelines*

Figure 3 shows the relative execution time changes when the guidelines have been applied to the source code (smaller numbers are better; the runtime of modified code has been divided by the original one). Thus, the legacy guidelines are still valid and it is worth using them in terms of performance.

An evaluation of ECMAScript 6 has revealed some unexpected and noteworthy modifications to ECMAScript 5. We have divided the results into two categories; one in which the previous standard performs better (Figure 4) on average and the other in which the new standard has more efficient code paths in the engines (Figure 5).



**Figure 4:** Performance improvement with ECMAScript 5



**Figure 5:** Performance improvement with ECMAScript 6

## 4. Dynamic Analysis of JavaScript’s Call Graphs

The fundamental area of understanding the structural blueprints of dynamic applications and even further detecting harmful behavior is the analysis of software function calls. A form of call information is the call graph, which is successfully used in mobile and non-mobile systems to detect both known and unknown harmful code.

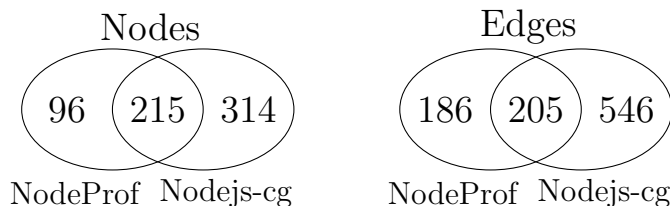
A call graph is a type of directed graph that illustrates the connections between functions in a program. Nodes in the graph represent the called functions, and the edges between them signify the function calls, with the direction of the edge pointing towards the callee. Call graphs can be created without running the program (known as a static call graph) or during execution (known as a dynamic call graph). First, we focus on the latter to enable dynamic analysis of JavaScript programs.

A considerable amount of web applications and back-end services have been developed using the Node.js framework. With more than 6.3 million websites using Node.js, it has become the most widely used tool for web development before React. Consequently, if one is looking to examine the structural designs of dynamic applications, *Node.js*-based applications could be a great option. In this research, we also focused on this framework to unravel the complexity of dynamic JavaScript language in the area of call chains and to provide useful and precise experiences that can be used in other code analyses (for example to detect malicious or fraudulent activities).

**Dynamic Call Graph Generation.** At the time of our analysis, there were no publicly available tools that produced a dynamic call graph for Node.js applications. However, some tools, with different goals, could be extended to generate call information for further processing. In the following, we discuss three tools that use distinct approaches to create call graphs.

The Jalangi2 framework is the first tool used to dynamically analyze the ECMAScript 5.1 code and is compatible with Node.js and multiple web browsers.

NodeProf is a dynamic program analysis tool for Node.js applications that is based on the Graal-nodejs project. This project utilizes the Graal.js engine to interpret JavaScript code and convert it into an abstract syntax tree (AST) representation that is then executed



**Figure 6:** *Number of call graph nodes and edges on SunSpider*

by the GraalVM virtual machine. Instead of instrumenting the source code, NodeProf links events to the program’s Abstract Syntax Tree (AST) representation and applies changes to the AST to report events.

We have developed a customized version of Node.js, known as Nodejs-cg, which features a modified V8 JavaScript engine. This engine is capable of producing an execution trace, which is typically used to print information about functions that are entered or exited. We have replaced this tracing mechanism with our call graph generator.

**Node and Edge Identification.** To compare multiple call graphs of the same program generated by different tools, it is necessary to assign a unique identifier to each node, regardless of the current execution of the program. This identifier can be created from the absolute path of the file, in which the function is defined, and the source code location in which the function starts. However, each tool interprets the file position of each language feature differently. Therefore, these differences should be unified to do any kind of comparison. Identifying the same edge is very straightforward if the nodes have already been unified. Thus, we can compare call graphs.

**Compare dynamic call graph generators.** Two testbeds were used. The first one is the SunSpider benchmark suite, which contains simple JavaScript files that test various parts of the JavaScript engine. The other is based on various Node.js modules that represent real-world applications.

Figure 6 shows a Venn diagram of the nodes and edges encountered during the running of the SunSpider benchmark suite. The number of nodes and edges identified by each generator tool is represented by a circle. The intersection of the two circles indicates the number of nodes and edges found by both tools, which are referred to as common nodes and edges in the rest of the research. The non-intersected regions of the circles represent the unique nodes and edges that are only found by one tool. If the call graphs generated by both generators had been identical, the number of unique nodes and edges would have been zero. However, Figure 6 shows a large number of unique nodes and edges. The differences come from the following three reasons.

*JavaScript Built-ins:* The ECMAScript standard outlines a variety of built-in functions (e.g., `sort()`). A JavaScript engine may incorporate built-in functions that are written in JavaScript or native functions that are not JavaScript-based. When a function is written in JavaScript, the call graph generator may create its node, and the relevant edges may be added to the call graph when the function is invoked or when it calls other functions. However, native built-in functions are usually not included in the call graph, since these

functions usually do not alert the engine when they are used.

*Module initialization:* The Node.js startup procedure, known as bootstrap, is partially written in JavaScript. During bootstrap, Node.js runs a few core modules that set up the module loading system, message queues, timers, and so on. These core modules are part of the Node.js binary to guarantee that they cannot be changed and that Node.js can always depend on them.

*Module loading:* SunSpider’s test driver loads the module by first wrapping the source code into a function expression. Node.js’s JavaScript engine then evaluates the wrapped code, creating internal functions, and executing them. This process is captured by the call graph generator, which adds a new edge to the call graph. The internal function returns with another function object, which is later called by Node.js, resulting in the addition of another edge to the call graph. This explains why the module load group has so many edges.

Similar differences can be seen in Table 3 showing the number of nodes recorded for each Node.js module as a second test after SunSpider. In this test, we have compared the call graphs generated from 12 Node.js modules. The left side of the table contains all the nodes that were identified, while the right side contains the nodes that remain after a filter is applied. This filtering process is conducted during testing and causes the generators to ignore the internal JavaScript functions of the JavaScript engine and Node.js.

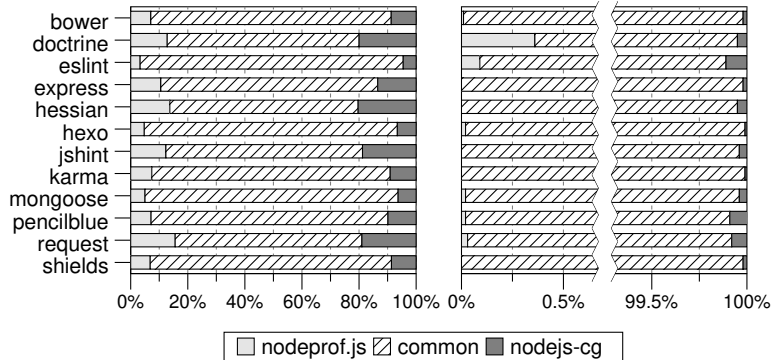
Table 3 reveals that there are multiple distinct nodes in the call graphs when the filter is not applied. However, this discrepancy is significantly reduced to a single digit when the filter is used. This implies that the majority of the nodes in the side columns of the left subtable are internal functions of both the JavaScript engine and Node.js. And, we got a similar result when analyzing the edges.

The differences in the call graph (both nodes and edges) are mainly due to the version of Node.js employed by the call graph generator and some test failures. Each module in our benchmark set checks the versions of Node.js and its supported command line options, which leads to different initialization steps depending on the version of Node.js. Additionally, there are a few test failures that occur only with NodeProf. We have disabled these tests that caused engine crashes, as the test systems cannot continue testing after a crash and a large portion of the call graph would be missing.

**Comparison of static and dynamic call graphs.** Although dynamic call graph generators have many advantages, the classic static version of generators should not be overlooked. Static approaches have the disadvantage of not being able to detect dynamic call edges from nontrivial *eval()*, *bind()*, or *apply()* usages (i.e., reflection). Additionally, they may be overly conservative, recognizing edges that are valid statically, but never realized in practice. However, they are faster, and more memory-friendly than dynamic analysis techniques and do not require a large testbed for the program being analyzed. Dynamic approaches, on the other hand, only identify real call edges, but the completeness of their results is highly dependent on the quality of the test cases for the program. Therefore, it is necessary to learn more about the current static and dynamic JavaScript call graph-building techniques to better understand their capabilities and limitations compared to each other (in terms of tools and approaches).

We quantitatively evaluated five distinct static analysis-based tools (TAJS, ACG, Google Closure Compiler, IBM WALA, and npm callgraph) and two dynamic tools (NodeProf and Nodejs-cg) to determine the various calls each tool can detect and how the results of the static

Name	All call graph nodes			Module call graph nodes		
	NodeProf	common	Nodejs-cg	NodeProf	common	Nodejs-cg
bower	804	9604	996	1	9604	2
doctrine	372	1954	581	7	1954	1
eslint	571	15898	781	15	15898	17
express	727	5239	928	0	5239	1
hessian	437	2103	648	0	2103	1
hexo	541	10076	749	2	10076	1
jshint	412	2299	627	0	2299	1
karma	828	9363	1019	0	9363	1
mongoose	708	12508	890	2	12506	5
pencilblue	539	6265	745	1	6265	6
request	876	3675	1067	1	3675	3
shields	773	9544	976	0	9544	2



**Table 3:** Number of call graph nodes found by NodeProf and Nodejs-cg

analysis-based tools compared to those of dynamic analysis-based tools. We also perform a quality analysis of the results, which involves comparing and validating the identified call edges and analyzing the discrepancies. Furthermore, we compare the results of the static and dynamic tools to gain an understanding of the overall accuracy of static analysis.

**Static and dynamic call graphs.** We quantitatively evaluated the call graphs by comparing the number of nodes and edges, as well as the similarity of entire call graphs. We conducted an analysis of the quality of the results by evaluating all 348 call edges found by the five static and two dynamic tools on the 26 SunSpider benchmark programs. We manually examined the JavaScript sources to determine the validity of the edges in the merged JSON files and added a new attribute ('valid') to the edges of the call graph, which can be either true or false. After evaluating the edges, we cross-reviewed the validation results and resolved any discrepancies. The final validated JSON was created based on consensus.

To ensure the most precise information retrieval metrics, we only took into account edges that were reported by the dynamic tools, that is, those that occurred during the execution of the program. In the case of simple source files, like the SunSpider benchmark, we thoroughly examined all 348 call edges by hand. In the case of Node.js modules, it would be very challenging to examine all the nodes and edges by hand, thus, we rely on our tools (which were validated on SunSpider’s results). Based on our findings we divided the identified edges into three categories:

- true positive (TP): the edge that exists and is realized during execution.
- false positive (FP): edge that does not exist in the source code.
- pseudo-positive (PP): edge that could be a real call but remains unrealized due to lack



of test input of the dynamic analysis.

Benchmark program	Static tools										Dynamic tools			
	npm-cg		ACG		WALA		Closure		TAJS		NodeProf		Nodejs-cg	
	nodes	edges	nodes	edges	nodes	edges	nodes	edges	nodes	edges	nodes	edges	nodes	edges
SunSpider	169	192	217	261	134	146	197	284	163	186	176	195	176	195

**Table 4:** *SunSpider analysis results*

The quantitative analysis shows the number of nodes and edges found by the call graphs generator tools (excluding built-in functions). In Table 4 we can see a static tool that produces results similar to the dynamic tool in almost every case on SunSpider.

We evaluated the static analysis and discovered 184 true positive edges. We then added all edges that could only be identified by dynamic tools as they are certain to occur during program execution. This resulted in a total of 195 edges, which we used as a benchmark. For each tool and all possible combinations of them, we were able to calculate the well-known information retrieval metrics (precision and recall). The Table 5 contains a summarization of these results in static tools.

Tool	TP	FP	PP	All	Prec.	Rec.	F
ACG	182	6	73	261	70%	93%	80%
Closure	175	54	55	284	62%	90%	73%
npm-cg	125	18	49	192	65%	64%	65%
TAJS	181	4	1	186	97%	93%	95%
WALA	122	19	5	146	84%	63%	72%
<b>ALL</b>	184	91	73	348	53%	94%	68%

**Table 5:** *Precision and recall measures for individual static tools*

Testing and analyzing static call graph generators on real-world programs is a challenging task. The npm callgraph and WALA were unable to analyze whole, multi-file projects because they cannot resolve calls among different files (e.g., requiring a module). The Closure Compiler can analyze complex programs as well, however, a manual evaluation of it, on various Node.js modules, showed only 20% precision in the case of found edges. The TAJS framework supports the required command, nonetheless, it was still unable to detect call edges in multi-file Node.js projects. Therefore, we could apply only ACG as a static tool to recognize call edges in Node.js modules. Thus, we used only this static and the two dynamic tools to perform the analysis and comparison on the selected Node.js modules.

We calculated some basic statistics from the gathered data that is shown in Table 6. The table displays the number of nodes (functions) and edges (possible calls between two functions) found by the tools. As can be seen, the results show resemblance, the correlation between these nodes and edges is high. Unsurprisingly, there are no exact matches in the number of nodes and edges for such complex input programs. The two dynamic tools produced almost identical results in terms of the number of nodes and edges. In this case, Table 7 shows these estimated numbers for precision and recall, since it is not feasible to double-check every edge by hand.

It is evident that dynamic approaches are highly precise, as they only report call edges that take place. However, this is also their greatest drawback, as they require a very high degree of test coverage to achieve the highest possible recall value. Furthermore, there may be code that relies on the current operating system, environment variables, or even the presence

	Static tools				Dynamic tools			
	ACG ONESHOT		ACG DEMAND		NodeProf		Nodejs-cg	
Node module	nodes	edges	nodes	edges	nodes	edges	nodes	edges
Node.js modules	8475	36070	8981	79248	9183	14679	9183	14682

**Table 6:** *Node.js analysis results*

Tool(s)	Prec.	Rec.	F
ACG ONESHOT	34.20%	58.40%	43.13%
ACG DEMAND	16.93%	63.53%	26.74%
Dynamic (NodeProf)	100.00%	69.50%	82.01%
Dynamic (Nodejs-cg)	100.00%	69.52%	82.02%
ALL	38.10%	100.00%	55.17%

**Table 7:** *Precision and recall values on Node.js modules*

of another service, for which traditional unit tests may not be adequate. In such cases, more complex test cases may necessitate multiple environments with different configurations (or even different interpreters), which can have a major impact on the accuracy of the call graph.

## The Author’s Contributions

The author’s contribution was decisive in the search, formalization, implementation, testing, and evaluation of a significant part of the *JavaScript* guidelines presented in the “*JavaScript Guidelines*” section. The published guidelines and measurement methods are undivided joint results with the co-authors.

The author had a decisive role in the design, implementation, and improvement of the *Nodejs-cg* dynamic call graph generator presented in the “*Dynamic Analysis of JavaScript’s Call Graphs*” section, and in the evaluation of the dynamic results included in the comparison. In addition, in the comparison of different types of call graph generators, the author had a decisive role in discovering and explaining the differences related to dynamic call graphs.

The publications related to this thesis point are the following:

- [6] Zoltán Herczeg, Gábor Lóki, Tamás Szirbucz, and Ákos Kiss. Guidelines for JavaScript Programs. Are They Still Necessary? In *SPLST’09 & NW-MODE’09. Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, pages 59–71, Tampere, Finland, 2009
- [7] Zoltán Herczeg, Gábor Lóki, Tamás Szirbucz, and Ákos Kiss. Validating JavaScript Guidelines Across Multiple Web Browsers. *Nordic Journal of Computing*, 15:18–31, 2013
- [8] Gábor Lóki and Péter Gál. JavaScript Guidelines for JavaScript Programmers - A Comprehensive Guide for Performance Critical JS Programs. In *Proceedings of the 13th International Conference on Software Technologies*, pages 397–404, Porto, Portugal, 2018. SciTePress
- [4] Zoltán Herczeg and Gábor Lóki. Evaluation and Comparison of Dynamic Call Graph Generators for JavaScript. In *Proceedings of the 14th International Conference on*

*Evaluation of Novel Approaches to Software Engineering*, pages 472–479, Heraklion, Greece, 2019. SciTePress

- [5] Zoltán Herczeg, Gábor Lóki, and Ákos Kiss. Towards the Efficient Use of Dynamic Call Graph Generators of Node.js Applications. In *Evaluation of Novel Approaches to Software Engineering.*, volume 1172 of *Communications in Computer and Information Science*, pages 286–302. Springer, 2020
- [1] Gábor Antal, Péter Hegedűs, Zoltán Herczeg, Gábor Lóki, and Rudolf Ferenc. Is JavaScript Call Graph Extraction Solved Yet? A Comparative Study of Static and Dynamic Tools. *IEEE Access*, 11:25266–25284, 2023

## Summary

In this thesis, two main thesis groups have been discussed. Both of them were related to compiler code optimizations and source code analysis. The first was in connection with executable code size optimizations in the classic compiler, and the second was with the Just-In-Time compilers, optimizing and analyzing the source code.

In the first thesis group, the main goal was to introduce efficient code size-optimizing algorithms for one of the most well-known compilers for the GCC, the GNU Compiler Collection. Code factoring is a class of useful optimization techniques that have been specifically developed to reduce code size. These approaches are aimed at reducing size by redefining the code. There were two main code factoring algorithms in focus. The first was the local code factoring that deals with individual instructions. This algorithm moves identical instructions from the basic blocks to their common predecessor or successor. The second was procedural abstraction which works with single-entry single-exit code regions rather than single instructions only, unlike local factoring. The main idea of this technique is to find such code regions that can be converted to procedures and replace all occurrences with calls to newly created subroutines. Both of the new algorithms have been implemented in different intermediate languages (*RTL*, *Tree-SSA*, *IPA's GENERIC*) as new optimization phases of GCC. The proposed code factoring algorithms are publicly available in the CFO branch. The final result reveals that a maximum of 61.53% and an average of 2.58% of extra code size savings can be achieved compared to the GCC flag `'-Os'`. These results would not have been achieved without a reliable code size measurement method and a stable benchmark environment. Based on our research and a detailed discussion with community members the official code size benchmark of GCC, the Code Size BEnchmark (*CSiBE*) was born. The testbed consists of 18 projects and the source size is about 50 MB. When compiling, the total amount of binary code is about 3.5 MB. Various types of programs, such as codecs (gsm, mpeg), compilers, compressors, editor programs, and preprocessed units, have been adopted. The constantly evolving benchmark now contains even more complex tests, from the Common Microcontroller Software Interface Standard (CMSIS) to Servo, the parallel browser engine. In addition to the GCC compiler, another compiler, LLVM Compiler Infrastructure also uses *CSiBE* in their development workflows.

The second thesis group discussed some parts of JavaScript's software analysis, starting with the JavaScript guidelines, that can improve the efficiency of JavaScript runtime performance. However, the guidelines are not official rules, they contain very useful suggestions to speed up the JavaScript code. The most important suggestions have been evaluated and measured with dominant JavaScript engines. This revealed that the legacy guidelines - suggested for ECMAScript 5.1 - are still valid and it is worth using them in terms of performance. On the other hand, as the language evolved, the new standard ECMAScript 6 was released with effective JavaScript engines. This raised the question of whether the new language features are getting better than the old simulated versions in terms of performance. The result we obtained showed that some of the changes are better with the new engine, but some are faster if the old simulated code snippets were used. As a follow-up research, the structural analyses of JavaScript programs have been done. In this part, the fundamental area of understanding the structural blueprints of dynamic applications was the focus with the help of a call graph. Call graphs can be created without running the program (known as a static call graph) or during execution (known as a dynamic call graph). To do that, a new

dynamic call graph generator was created, called *Nodejs-cg*, which is a customized version of *Node.js* with a modified *V8* JavaScript engine. In our research, the newly created call graph generator was compared against two other dynamic generators and later several static ones. As a result, the *Nodejs-cg* performed better in detecting valid nodes and edges for the call graph compared to the other dynamic solutions. Later, a filtering mechanism and other constraints were added so that the output of dynamic call graphs could be compared to the static ones. Thus, the final result stated that dynamic approaches are highly precise, as they only report call edges that actually take place at runtime. However, this is also their greatest drawback, as they require a very high degree of test coverage to achieve the highest possible recall value. Furthermore, there may be code that relies on the current operating system, environment variables, or even the presence of another service, for which traditional unit tests may not be adequate. In such cases, more complex test cases may require multiple environments with different configurations (or even different interpreters), which can have a major impact on the accuracy of the call graph.

Finally, Table 8 summarizes the relation between the thesis points and the corresponding publications.

	[3]	[9]	[2]	[10]	[11]	[6]	[7]	[8]	[4]	[5]	[1]
I.	•	•	•	•	•						
II.						•	•	•	•	•	•

**Table 8:** *Correspondence between the main thesis points and the corresponding publications*

## Acknowledgments

Firstly, I want to thank Ákos Kiss, my supervisor, for guiding and helping me as a researcher and for always presenting interesting and challenging problems. Secondly, Tibor Gyimóthy, my mentor, who introduced me to the fascinating world of research more than twenty years ago. Thirdly, I would like to thank all my colleagues and co-authors, namely Árpád Beszédes, Attila Dusnoki, Balázs Nagy, Csaba Nagy, Dániel Vince, Edit Szűcs, Gábor Antal, István Hegedűs, Judit Jász, László Vidács, Péter Gál, Péter Hegedűs, Rudolf Ferenc, Tamás Gergely, Zoltán Herczeg. In some way, they have all contributed to the creation of this thesis. Finally, I wish to express my gratitude to my mother, my father, my wife, and my children for their continuous support.

One or more research papers, the result of which were used in this thesis, were partially supported by

- the TÁMOP-4.2.2/08/1/2008-0008 program of the Hungarian National Development Agency,
- the Hungarian Government and the European Regional Development Fund under the grant number GINOP-2.3.2-15-2016-00037 (“Internet of Living Things”),

- the grant TUDFO/47138-1/2019-ITM of the Ministry for Innovation and Technology, Hungary,
- the European Union Project within the framework of the Artificial Intelligence National Laboratory under Grant RRF-2.3.1-21-2022-00004,
- the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA Funding Scheme, under Project TKP2021-NVA-09, and
- the University of Szeged Open Access Fund under Grant 5913.

## References

- [1] Gábor Antal, Péter Hegedűs, Zoltán Herczeg, Gábor Lóki, and Rudolf Ferenc. Is JavaScript Call Graph Extraction Solved Yet? A Comparative Study of Static and Dynamic Tools. *IEEE Access*, 11:25266–25284, 2023.
- [2] Árpád Beszédes, Rudolf Ferenc, Tamás Gergely, Tibor Gyimóthy, Gábor Lóki, and László Vidács. CSiBE benchmark: One year perspective and plans. In *Proceedings of the 2004 GCC Developers’ Summit*, pages 7–15, Ottawa, Canada, 2004.
- [3] Árpád Beszédes, Tamás Gergely, Tibor Gyimóthy, Gábor Lóki, and László Vidács. Optimizing for space: Measurements and possibilities for improvement. In *Proceedings of the 2003 GCC Developers’ Summit*, pages 7–20, Ottawa, Canada, 2003.
- [4] Zoltán Herczeg and Gábor Lóki. Evaluation and Comparison of Dynamic Call Graph Generators for JavaScript. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 472–479, Heraklion, Greece, 2019. SciTePress.
- [5] Zoltán Herczeg, Gábor Lóki, and Ákos Kiss. Towards the Efficient Use of Dynamic Call Graph Generators of Node.js Applications. In *Evaluation of Novel Approaches to Software Engineering.*, volume 1172 of *Communications in Computer and Information Science*, pages 286–302. Springer, 2020.
- [6] Zoltán Herczeg, Gábor Lóki, Tamás Szirbucz, and Ákos Kiss. Guidelines for JavaScript Programs. Are They Still Necessary? In *SPLST’09 & NW-MODE’09. Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, pages 59–71, Tampere, Finland, 2009.
- [7] Zoltán Herczeg, Gábor Lóki, Tamás Szirbucz, and Ákos Kiss. Validating JavaScript Guidelines Across Multiple Web Browsers. *Nordic Journal of Computing*, 15:18–31, 2013.

- [8] Gábor Lóki and Péter Gál. JavaScript Guidelines for JavaScript Programmers - A Comprehensive Guide for Performance Critical JS Programs. In *Proceedings of the 13th International Conference on Software Technologies*, pages 397–404, Porto, Portugal, 2018. SciTePress.
- [9] Gábor Lóki, Ákos Kiss, Judit Jász, and Árpád Beszédes. Code Factoring in GCC. In *Proceedings of the 2004 GCC Developers' Summit*, pages 79–84, Ottawa, Canada, 2004.
- [10] Csaba Nagy, Gábor Lóki, Árpád Beszédes, and Tibor Gyimóthy. Code factoring in GCC on different intermediate languages. In *Proceedings of the 10th Symposium on Programming Languages and Software Tools*, pages 79–95, Dobogókő, Hungary, 2007. Eötvös Loránd University Press.
- [11] Csaba Nagy, Gábor Lóki, Árpád Beszédes, and Tibor Gyimóthy. Code factoring in GCC on different intermediate languages. *Annales Universitatis Scientiarum Budapestinensis De Rolando Eötvös Nominatae Sectio Computatorica*, 30:79–95, 2009.

## Összefoglaló

Ebben a dolgozatban két fő téziscsoportot tárgyaltunk. Mindkettő a fordítóprogramok optimalizálásához és a forráskód elemzéséhez kapcsolódott. Az első a klasszikus fordítóprogramok kódméret optimalizációjához kapcsolódik, a második pedig a ‘Just-In-Time’ fordítók forráskód optimalizálásával és elemzésével foglalkozik.

Az első téziscsoportban a fő cél az volt, hogy hatékony kódméret-optimalizáló algoritmusok készüljenek az egyik legismertebb fordítóhoz, a ‘GNU Compiler Collection’-hez (GCC). A ‘code factoring’ algoritmusokat kifejezetten a kód méretének csökkentésére fejlesztették ki. Ezek a bináris kód méretének csökkentését a kód újradefiniálásával oldják meg. Két fő algoritmust különböztetünk meg. Az első a ‘local code factoring’, amely az egyedi utasításokkal foglalkozik. Ez az algoritmus azonos utasításokat mozgat a tartalmazó ún. ‘basic block’-okból egy közös szülő (un. előd) vagy gyerek (un. utód) blokkba. A második algoritmus a ‘procedural abstraction’, amely a ‘local code factoring’-től eltérően nem egyetlen utasítással, hanem egy be- és kilépési ponttal rendelkező kódrégiókkal dolgozik (‘single-entry single-exit’). Ennek a technikának az alapötlete az, hogy olyan kódrégiókat találjon, amelyeket eljárásokká lehet konvertálni, és minden ilyen kódrégió előfordulást lecserélni az újonnan létrehozott szubrutin hívásra. Mindkét algoritmus különböző köztes nyelveken (*RTL*, *Tree-SSA*, *IPA’s GENERIC*) lett elkészítve a GCC egyes optimalizálási fázisaként. Az elkészült algoritmusok nyilvánosan elérhetők a GCC-nek a CFO fejlesztői ágban. A végeredmény megmutatta, hogy a GCC ‘-Os’ kapcsolójához képest maximum 61,53% és átlagosan 2,58% extra kódméret megtakarítás érhető el az algoritmusok használatával. Ezeket az eredményeket nem lehetett volna elérni egy megbízható kódméret mérési módszer és egy stabil benchmark környezet nélkül. Ezen a területen végzett kutatásunknak és a fordítóprogram közösség tagjaival folytatott részletes megbeszélésnek köszönhetően megszületett a GCC hivatalos kódméret benchmarkja, a Code Size BENCHMARK (*CSiBE*). A *CSiBE* 18 projektből áll, és a források összmérete körülbelül 50 MB. Fordításkor a bináris kód teljes mérete körülbelül 3,5 MB. A tesztrendszer különféle típusú projekteket tartalmaz, például kodekeket (gsm, mpeg), fordítókat, tömörítőket, szerkesztőprogramokat és előfeldolgozó egységeket. A folyamatosan fejlődő benchmark jelenleg még összetettebb tesztekkel tartalmaz, egészen az alacsonyszintű ‘Common Microcontroller Software Interface Standard’ (CMSIS) szabványtól a ‘Servo’, a párhuzamos böngészőmotorig. A GCC fordító mellett egy másik fordítóprogram, az ‘LLVM Compiler Infrastructure’ is használja a *CSiBE* benchmarkot a fejlesztési munkafolyamataiban.

A második téziscsoport a JavaScript szoftverelemzésének néhány részét tárgyalta, kezdve a JavaScript irányelvekkel, amelyek javíthatják a JavaScript futásidejű teljesítményének hatékonyságát. Ezek az irányelvek azonban nem hivatalos szabályok, de nagyon hasznos javaslatokat tartalmaznak a JavaScript kód felgyorsítására. A legfontosabb irányelveket a népszerű JavaScript motorok segítségével lettek lemérve és kiértékelve. Ebből kiderült, hogy az ECMAScript 5.1-hez javasolt régi irányelvek továbbra is érvényesek, és a teljesítménynövelés szempontjából érdemes ezeket alkalmazni. Másrészt, ahogy a nyelv fejlődött, megjelentek az új szabványt, az ECMAScript 6-ot, követő hatékony JavaScript motorok. Ez felvetette a kérdést, hogy az új nyelvi funkciók teljesítményben jobbak-e, mint a régi szimulált verziók. A kapott eredmény azt mutatta, hogy egyes változtatások jobbak az új motorral, de vannak olyanok, amelyek gyorsabbak, ha a régi szimulált kódrészleteket használták. A kutatás következő fázisaként a JavaScript programok szerkezeti elemzése is megtörtént. Ebben a részben a dinamikus JavaScript alkalmazások, hívási gráf analízis segítségével történő, szer-



kezeti struktúrájának megértésének területe állt a fókuszban. Hívási gráfok hozhatók létre a program futtatása nélkül (statikus hívási gráfként) vagy végrehajtás közben (dinamikus hívási gráfként). Ehhez egy új dinamikus hívási gráf generátort készítettünk el, *Nodejs-cg* néven, amely a *Node.js* testreszabott változata egy módosított V8 JavaScript motorral. Kutatásunkban az újonnan létrehozott hívási gráf generátort két másik dinamikus generátorral, majd később több statikus generátorral hasonlítottuk össze. Az eredmények szerint a *Nodejs-cg* jobban teljesített a hívási gráf csomópontjainak és éleinek helyes észlelésében, mint a többi dinamikus megoldás. Később olyan szűrőmechanizmus és más megszorítások lettek bevezetve, hogy a dinamikus hívási grafikonok kimenete összehasonlítható legyen a statikus gráfokéval. Így a végeredmény rámutatott arra, hogy a dinamikus megközelítések rendkívül pontosak, mivel csak azokat a hívás éleket regisztrálják, amelyek ténylegesen futásidőben zajlanak le. Ez azonban a legnagyobb hátrányuk is, hiszen nagyon magas fokú tesztlefedettséget igényelnek ehhez a magasabb ‘recall’ érték eléréséhez. Azt is meg kell említeni, hogy előfordulhatnak olyan kódok, amelyek az aktuális operációs rendszerre, környezeti változókra, vagy akár más szolgáltatás jelenlétére támaszkodnak, amihez a hagyományos egységtesztek nem biztos, hogy megfelelőek. Ilyen esetekben javasolt több különböző hívási gráf generátor használata, ami nagy hatással lehet a hívási gráf pontosságára.

## Nyilatkozat

Lóki Gábor “*Compiler Optimizations and Source Code Analysis*” című PhD disszertációjában a következő eredményekben Lóki Gábor hozzájárulása volt a meghatározó:

Az első tézisponthoz, az “*Executable Code Optimizations*” részhez tartozó eredmények:

- A szerzőnek meghatározó szerepe volt a “*Code Factoring Techniques in the GCC Compiler*” részben bemutatott algoritmusok jelentős hányadának tervezésében, implementálásában, javításában és karbantartásában:
  - *Local Code Factoring*: A szerző megtervezte, implementálta és karbantartotta az *RTL* változatát, illetve javította és karbantartotta a *Tree-SSA* változatát az algoritmusnak.
  - *Procedural Abstraction*: A szerző megtervezte, implementálta és karbantartotta a *Tree-SSA* változatát, illetve javította és karbantartotta az algoritmus *RTL* változatát. Ezen felül az algoritmus *IPA* változatának a tervezését, javítását és karbantartását is elvégezte.
  - *Hash Tables*: Ezt a technikát a szerző javasolta és implementálta a különböző kódoptimalizációs algoritmusokhoz.
- A szerzőnek meghatározó szerepe volt a “*Binary Code Size Measurement Methods and Benchmark*” részben bemutatott fordítóprogramok bináris kódoptimalizációkhoz készített kódméret méréstechnikájának és kiértékelő környezetének tervezési, megvalósítási és javítási fázisaiban. Ezek mellett a szerző 2004 óta a fő karbantartója a hivatalos kód méret kiértékelő környezetnek és tesztrendszernek, a *CSiBE*-nek.

A következő felsorolásban lévő publikációk tartoznak ehhez a tézisponthoz:

1. Árpád Beszédes, Tamás Gergely, Tibor Gyimóthy, **Gábor Lóki**, and László Vidács - Optimizing for space: Measurements and possibilities for improvement. In *Proceedings of the 2003 GCC Developers' Summit*, pages 7-20, Ottawa, Canada, 2003.
2. Árpád Beszédes, Rudolf Ferenc, Tamás Gergely, Tibor Gyimóthy, **Gábor Lóki**, and László Vidács - CSiBE benchmark: One year perspective and plans. In *Proceedings of the 2004 GCC Developers' Summit*, pages 7-15, Ottawa, Canada, 2004.
3. **Gábor Lóki**, Ákos Kiss, Judit Jász, and Árpád Beszédes - Code Factoring in GCC. In *Proceedings of the 2004 GCC Developers' Summit*, pages 79-84, Ottawa, Canada, 2004.
4. Csaba Nagy, **Gábor Lóki**, Árpád Beszédes, and Tibor Gyimóthy - Code factoring in GCC on different intermediate languages. In *Proceedings of the 10th Symposium on Programming Languages and Software Tools*, pages 79-95, Dobogókő, Hungary, 2007. Eötvös Loránd University Press.

5. Csaba Nagy, **Gábor Lóki**, Árpád Beszédes, and Tibor Gyimóthy - Code factoring in GCC on different intermediate languages. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eotvos Nominatae Sectio Computatorica, Sectio Computatorica* - Tomus XXX., pages 79-95, 2009.

A **második tézispont**hoz, a “*Just-In-Time Compilers’ Optimizations and Analyses*” részhez, tartozó tartozó eredmények:

- A szerző hozzájárulása volt meghatározó a “*JavaScript Guidelines*” részben bemutatott *JavaScript* irányelvek jelentős részének felkutatásában, formalizálásában, implementálásában, tesztelésében és kiértékelésében. A publikált irányelvek és mérési módszerek osztatlan közös eredmények a társzerzőkkel.
- A szerzőnek meghatározó szerepe volt a “*Dynamic Software Analysis of JavaScript’s Call Graphs*” részben bemutatott *Nodejs-cg* dinamikus hívási gráf generátor megtervezésében, implementálásában és javításában, az összehasonlításban szereplő dinamikus eredmények kiértékelésében. Ezen felül a különböző típusú hívási gráf generátorok összehasonlításában a szerzőnek meghatározó szerepe volt a dinamikus hívási gráfokhoz kapcsolódó különbözőségek felfedezésében és megmagyarázásában.

A következő felsorolásban lévő publikációk tartoznak ehhez a tézispont

6. Zoltán Herczeg, **Gábor Lóki**, Tamás Szirbucz, and Ákos Kiss - Guidelines for JavaScript Programs. Are They Still Necessary? In *Proceedings of the 11th Symposium on Programming Languages and Software Tools SPLST’09*, pages 59-71, Tampere, Finland, 2009.
7. Zoltán Herczeg, **Gábor Lóki**, Tamás Szirbucz, and Ákos Kiss - Validating JavaScript Guidelines Across Multiple Web Browsers. *NORDIC JOURNAL OF COMPUTING*, volume 15, pages 18-31, 2013.
8. **Gábor Lóki** and Péter Gál - JavaScript Guidelines for JavaScript Programmers - A Comprehensive Guide for Performance Critical JS Programs. In *Proceedings of the 13th International Conference on Software Technologies*, pages 397-404, Porto, Portugal, 2018. SciTePress.
9. Zoltán Herczeg and **Gábor Lóki** - Evaluation and Comparison of Dynamic Call Graph Generators for JavaScript. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 472-479, Heraklion, Greece, 2019. SciTePress.
10. Zoltán Herczeg, **Gábor Lóki**, and Ákos Kiss - Towards the Efficient Use of Dynamic Call Graph Generators of Node.js Applications. In *Evaluation of Novel Approaches to Software Engineering*, volume 1172, pages 286-302, Communications in Computer and Information Science, 2020. Springer.
11. Gábor Antal, Péter Hegedűs, Zoltán Herczeg, **Gábor Lóki**, and Rudolf Ferenc - Is JavaScript Call Graph Extraction Solved Yet? A Comparative Study of Static and Dynamic Tools. *IEEE Access*, volume 11, pages 25266-25284, March 2023. IEEE.

Ezek az eredmények **Lóki Gábor** PhD disszertációján kívül más tudományos fokozat megszerzésére nem használhatók fel.

Szeged, 2023.08.25



---

**Lóki Gábor**  
jelölt

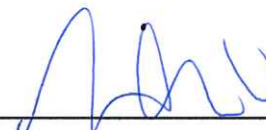


---

**Dr. Kiss Ákos**  
témavezető

Az Informatika Doktori Iskola vezetője kijelenti, hogy jelen nyilatkozatot minden társszerzőhöz eljuttatta, és azzal szemben egyetlen társszerző sem emelt kifogást.

Szeged, 2023. 08. 25.



---

**Dr. Jelasity Márk**  
doktori iskola vezető

