Compiler Optimizations and Source Code Analysis

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

"Wisdom is not a product of schooling but of the lifelong attempt to acquire it."

— Albert Einstein

Foreword

At the end of 2001, I joined a very committed research team as a young researcher. In one and a half years, I began to write my first paper with my colleagues. Then new challenges, new projects, interesting research, industrial partners, and so much more happened, it is difficult to list them all. As they say; life is a journey filled with unexpected miracles.

Two decades have just passed, and now I am about to finish another challenge, my PhD thesis. How time flies! Most of this time has been dedicated to research: finding interesting problems, seeking and experimenting with new approaches, and finding the best solutions. This thesis tries to summarize the results of this two-decade research period. Many people around me have contributed to my research over the years. They always gave me good advice, help and knowledge, or just motivated, encouraged and pushed me to achieve my goals. Therefore, at the beginning of this study, I would like to thank them for their support.

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.

Gábor Lóki, 2023

Contents

List of Figures

List of Tables

1 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 [47, 61, 65, 69, 95] 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.

This thesis aims to explore some parts of this diverse area of software and to discuss some other compiler optimizations and technologies that can be useful for the users of compilers. Part I explores binary optimizations of compilers that are designed to aid resource consumption, especially the ones that can reduce code size. Part II presents how optimization techniques can be used in Just-In-Time compilers, focusing on the search for improved performance and code analysis.

Part I

Executable Code Optimizations

2 Overview

Software optimization is a key element in improving the overall performance and efficiency of computer programs. As computer architectures evolve and the demand for faster and more robust software applications increases, the role of code compilers is increasingly important. A code compiler is a fundamental tool for translating human-readable source code into machine-readable binary code, facilitating software execution on the target computer system. The optimized binary code, achieved through advanced compiler technology, plays a key role in meeting the challenges of today's computer landscape.

The importance of optimizing binary code cannot be overstated. Optimized binary code allows software to run faster and consume fewer storage resources, thus increasing the overall performance of computer programs. In today's computing environment, software efficiency can be a decisive factor in improving competitiveness.

Optimization techniques use a wide range of strategies during the compilation process, including, but not limited to, loop transformations, register allocations, instruction scheduling, inter- and intra-procedural analysis, and code transformations. Together, these technologies are designed to reduce execution time, memory consumption, energy consumption, and executable code size, which have real benefits in various areas, such as real-time systems, embedded devices, scientific simulations, and high-performance computing.

I. 2. Overview

Among the many code compilers available, the GNU Compiler Collection (GCC) [29] is one of the most prominent and widely used open-source compilers, known for its versatility. It supports many programming languages and targets a wide range of hardware platforms. Its open-source nature has encouraged a vibrant community of developers and researchers who have been continuously working to improve its optimization capabilities for decades. This part of the thesis aims to present some effective code size optimization techniques used by code compilers, with a particular focus on the GCC compiler.

GCC supports many architectures and is also widely used in mobile phones and other embedded devices. When compiling software for devices such as mobile phones, embedded computers, and routers where storage capacity is limited, compilers have a very important feature: providing the smallest binary code. GCC already contains code size reduction algorithms, but in special cases, the amount of free space saved is very important, so further optimization techniques can be very useful and affect our daily lives. And from the industrial's point of view, it can also mean huge savings in costs.

This part of the thesis will give an overview of the code factoring algorithms in the different intermediate language (IL) representations of GCC in Chapter 3. Chapter 4 will then give a detailed description of how to calculate the code size of a compiled application. As a result of the presented method, the official GCC code size benchmark environment was born. These chapters are based on the results of our previously published papers [10, 11, 55, 67, 68]. The results and the conclusions of this part are summarized in Chapter 6.

3

Code Factoring Techniques in the GCC Compiler

Code factoring is a class of useful optimization techniques that have been specifically developed to reduce code size [13, 14, 16, 18]. These approaches are aimed at reducing size by redefining the code. The following sections will discuss two ideas and mechanisms related to code factoring, one for individual instructions and the other for longer instruction sequences. Next, we present the implementation details of them in GCC. Finally, the evaluation of their result and the efficiency of these optimization techniques will be shown.

A common characteristic of these algorithms is that they operate on the control flow graph [64] (CFG). The CFG is a directed graph, its primary purpose is to model the program's control flow. In short, it shows how a program's control structures, such as loops, conditionals, and function calls, affect the order of execution. The key components of a control flow graph are:

Basic blocks (or nodes). In a control flow graph, nodes represent basic blocks of code. A basic block is a sequence of instructions that are guaranteed to be executed sequentially without any jumps or jump targets. There are two special *basic blocks*: the *entry block*, through which control enters into the CFG, and the *exit block*, through which all control flow leaves.

Edges. Edges in the control flow graph connect two nodes and indicate the flow of control between them. There are several special edges. A *back edge* is an edge that points to a block that has already been met during a depth-first traversal of the graph. The back edges are typical of loops. An *abnormal edge* is an edge whose destination is unknown. The exception-handling constructs can produce them. These edges tend to inhibit optimization. There are two more: *critical* and *impossible* edges. The first needs to be split which also requires changes in the CFG. The latter is the fake or technical edge which is used in the domination relationship calculation [64].

The other important feature that these algorithms rely on is the Data Flow Graph [64] (DFG). It is also a directed graph where nodes represent operations or computations, and edges represent the flow of data between these operations. The data flow graph shows the data dependencies and relationships between various operations in a program.

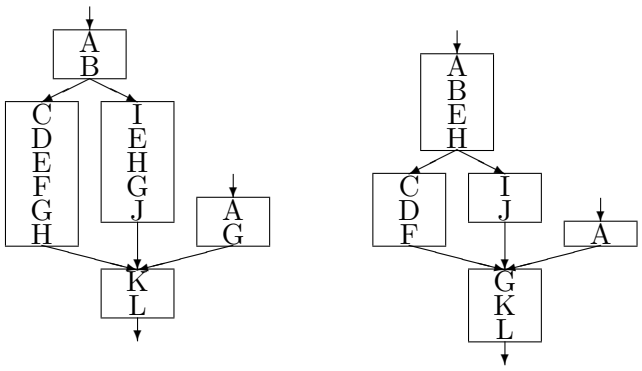
3.1 Local Code Factoring

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 3.1a 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 3.1b 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.

Now, let us examine some more complex cases. 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. Figure 3.2 shows this case. A similar situation occurs in which the basic blocks have more than one common successor (see Figure 3.3). Furthermore, in the case of sinking, even those instructions that are not present in all of the blocks

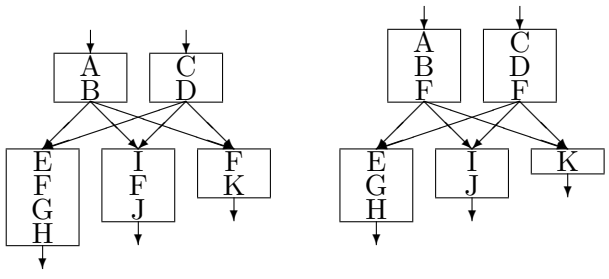
can be moved by creating a new successor block for them. Figure 3.4 shows an example of a CFG for this case.

Except for this last case, which involves the creation of a new basic block, local factoring also has the additional advantage of being good for runtime performance. For example, shorter basic blocks can help to use a shorter form of jumps, or improve the instruction scheduling algorithm by moving the data definition close to its usage or vice versa.



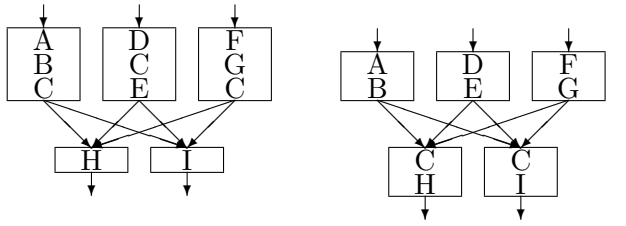
(a) before transformation (b) after transformation

Figure 3.1: *The effect of local code factoring on the CFG*



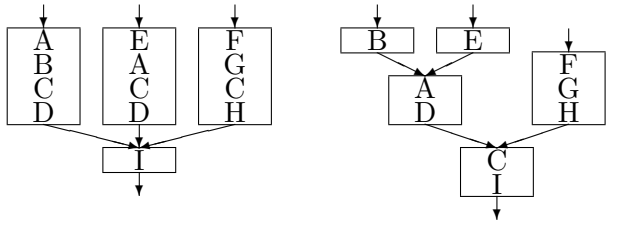
(a) original (b) after local factoring

Figure 3.2: *Effect of local factoring on basic blocks with multiple common predecessors*



(a) original (b) after local factoring

Figure 3.3: *Effect of local factoring on basic blocks with multiple common successors*



(a) original (b) after local factoring

Figure 3.4: *Effect of local factoring on basic blocks with multiple common successors but only partially common instructions*

3.2 Procedural Abstraction

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 [14, 18] 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 3.5a). These solutions can be used to abstract the longest possible sequence into a function and leave the shorter sequence unabstracted (Figure 3.5b) 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 3.5c).

Here, we propose to create *multiple entry subroutine* in the cases described above to allow 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. Each matching sequence must be replaced with a call to the appropriate entry point of the new function. Figure 3.5d shows the optimal solution to the problem indicated in Figure 3.5a.

Needless to say, procedural abstraction can lead to a decrease in runtime performance due to the insertion of call and return code. Furthermore, the size overhead of the inserted code must also be taken into account when looking for candidate sequences to perform the procedural abstraction. Thus, the abstraction shall only be performed if the gain resulting from the elimination of duplicates exceeds the loss resulting from the insertion of additional instructions.

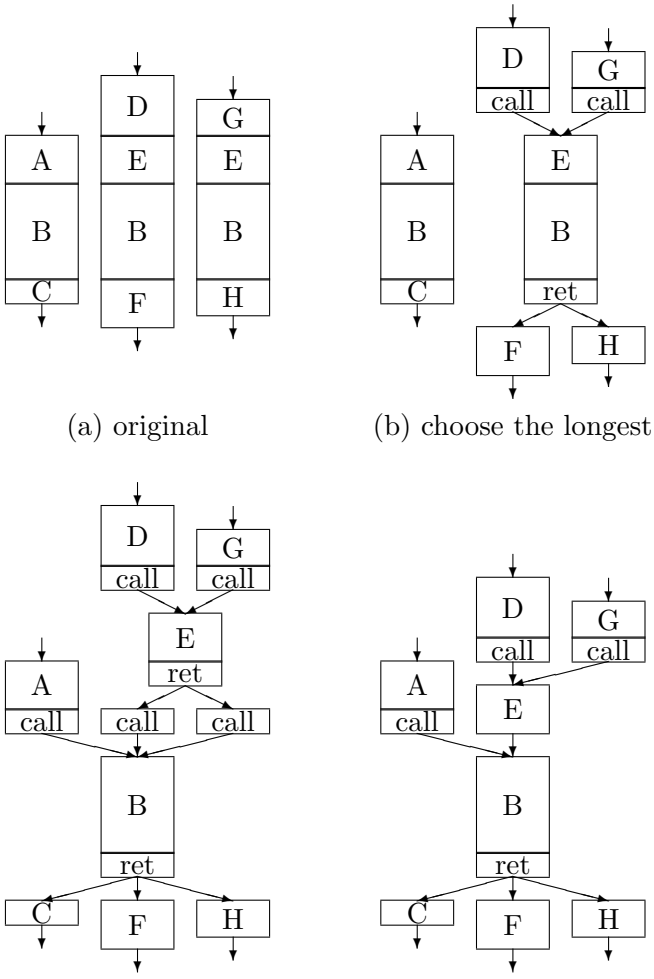


Figure 3.5: *Different kinds of procedural abstraction strategies*

3.3 Implementation Details

GCC already contains some algorithms similar to those discussed in Section 3.1 and Section 3.2, but they usually reduce code size only if the transformation does not introduce a (significant) performance overhead. In addition, they generally have fewer potential than the ones described above. The *cross-jumping* [64] algorithm combines the identical tails of basic blocks, but this approach can only handle very limited subsets of the general problems of procedural abstraction. Another algorithm, called *if-conversion* [58], has a similar effect on the code to local factoring when followed by a *combine* phase. In contrast to local factoring, *if-conversion* is bound to conditional jumps only.

Both of the new algorithms have been implemented on different intermediate languages as new optimization phases of GCC. First, it was implemented in the *Register Transfer Language* (*RTL*) [30], then *Tree-SSA* [70, 71] (which is based on Static Single Assignment, or SSA technique [15]). Finally, a more abstract approach was implemented on the *GENERIC* IL level, introduced in the *Interprocedural Abstraction Analysis* [44] (*IPA*) optimization phase.

The implementation of these algorithms is publicly available. They are part of the GCC codebase.

3.3.1 Implementations in RTL

Local Code Factoring. The local factoring algorithm is divided into two parts and is implemented as two individual optimization phases in GCC. One algorithm executes the hoisting of instructions, i.e. to move them upward to their predecessor blocks, while the other one is responsible for the sinking of the instructions, i.e. to move them downward to their successor basic blocks. A central problem for both algorithms is to decide whether an instruction may be freely moved out of its block. An instruction cannot be moved across others that use parameters defined by the instruction itself, or define parameters used or defined by the candidate instruction. GCC provides methods for collecting the required definition/use information for the whole processed function. However, from the local factoring point of view, these methods are too expensive, as only a small part of the calculated information is used. Therefore, the implementation contains a light version of the definition/use calculation code. As we are also sensitive to the compilation time in the implementation, we have made it possible to parameterize the maximum number of instructions that algorithms should analyze starting from the top or bottom of the basic blocks when

searching for candidates of motion.

Procedural Abstraction. Using the *RTL* representation algorithms can optimize only one function at a time. Although procedural abstraction is essentially an interprocedural optimization technique, it can be adapted to an intraprocedural working environment. Instead of creating a new function from the identical code fragments, one representative instance of them has to be kept in the body of the processed function and all the other occurrences will be replaced by code transferring control to the retained instance. However, in order to retain the original program's semantics, the code location where the control should return to after executing the retained instance must be remembered in some way, so the subroutine call/return mechanism must be mimed. In the current implementation, we use labels to mark the return addresses, registers to store references to them, and jumps on registers to transfer control back to *mimed callers*.

3.3.2 Implementations in Tree-SSA

As a general rule in compilers, the higher the abstraction level of an intermediate language is the more architecture-specific instructions are represented by a single IL instruction. In some sense, this can ease our work, since some architecture-specific information is hidden, but it may also make the optimizations less efficient by removing several possible candidates.

The biggest challenge in implementing any kind of optimization algorithm on the higher abstraction level is that later optimization phases might change the effect of the original algorithm, usually not in our favor. This is one of the focus areas of the research topics that try to find the best order of optimization algorithms [96].

Local Code Factoring. In the case of local factoring, when moving statements from one basic block to another, we have to pay careful attention to *phi nodes*, *virtual operators*, and *immediate uses*. A *phi node* is a special kind of assignment in basic blocks with multiple predecessors that indicate which definitions (or assignments) to the given variable reach the current join point of the CFG [1]. The *virtual operands* are for non-scalar variables (e.g. arrays). The compiler stores references to the non-scalar's base object within the virtual

[1]The concept of *phi nodes* is inherent to the SSA representation [15] and not specific to GCC internals.

operand. In this way, the definition and use dependencies can be tracked on this node as well.

For example, a variable of the movable assign statement's left hand appears inside a phi node (e.g. Figure 3.6). In these cases, after copying the statement to the children or parents' blocks, we must recalculate the phi node. Later, in both sinking and hoisting cases, we must walk through the immediate uses of the moved statements and replace the defined variables with the new definitions.

```
if (a>100) {
  a=b+a;
  c=a*10;
  a=a-c;
} else {
  a=a+b;
  c=a*12;
  a=a-c;
}
return a;
```

(a) Original source C code

```
  int D.1770;
<bb 0>:
  if (a_2 > 100) goto <L0>;
  else goto <L1>;
<L0>:;
  a_9 = b_5 + a_2;
  c_10 = a_9 * 10;
  a_11 = a_9 - c_10;
  goto <bb 3> (<L2>);
<L1>:;
  a_6 = a_2 + b_5;
  c_7 = a_6 * 12;
  a_8 = a_6 - c_7;
  # a_1 = PHI <a_11(1), a_8(2)>;
<L2>:;
  D.1770_3 = a_1;
  return D.1770_3;
```

(b) Tree-SSA representation

Figure 3.6: *An example code for Tree-SSA form with moveable statements.*

Procedural Abstraction. The Tree-SSA IL is much closer to the original programming language than the RTL, so it is likely to detect similar sequences to code clones which the developers tend to introduce [46]. Our idea was that if the procedural abstraction algorithm finds similar sequences at a higher IL, it might lead to a better code size reduction than in a lower IL since no other optimization algorithm can change any candidate of the procedural abstraction.

In Tree-SSA, the implementation of this algorithm has fewer restrictions than in RTL. For example, we do not have to worry about register representa-

tions. Thus, the algorithm is able to find more sequences as good candidates for abstraction, while in RTL we have to make sure that all references to registers are the same in every subsequence.

There is one great challenge in implementing the algorithm in Tree-SSA, which is that it is in the very early stages of the GCC optimization pipeline. Many other optimizations follow the Tree-SSA algorithms. This can simply result in other algorithms, followed by abstraction, simply destroying the initial result of the algorithm. For example, there are some cases where the abstraction does a merge, which is not really useful as one or more sequences are later deleted (e.g. they are dead code), or there are other batches of algorithms that could have applied different, more efficient transformations to the candidate sequences. This last case occurs mainly when the algorithm attempts to merge short sequences.

In general, it can be said that the implementation of any algorithm in Tree-SSA IL is much more straightforward than in RTL IL. The support of SSA may facilitate work, and the target-independent IL makes our algorithm more universal across platforms and helps to avoid additional attention to hardware constraints (e.g. register allocations, and supported operators). The main disadvantage is that the optimization algorithms are not as powerful compared to an implementation in a hardware-aware IL. However, other follow-up optimization algorithms can change the modified code so drastically that this removes the advantages of our algorithm.

3.3.3 Implementations in IPA

The main idea of interprocedural analysis (IPA) optimization is to support algorithms that work throughout the entire program: either across procedures of a compilation unit or even across file boundaries. For a long time, open-source GCC had no powerful interprocedural methods because its structure was optimized to compile functions as units. In the initial development of this optimization pipeline, the new IPA framework and passes were introduced by the IPA branch [44]. After some time, and of course, during the development phases of stabilization, the framework landed in GCC's main branch, and now it is part of the GCC optimization passes officially.

Procedural Abstraction. We implemented the inter-process version of our algorithm in the first stage of the development of the IPA framework. This implementation is very similar to that used in Tree-SSA, but with a major

difference: we can merge sequences from any function into a new real function, not a mimed one.

With this approach, we have to deal with more code size overhead coming from the function call API than the other two cases described above. In addition, we can also merge sequences using different variables and addresses because it is possible to pass variables as parameters to the newly created function. The procedural abstraction identifies and compares the structure of sequences to find good candidates for abstractions, and with this method, we can merge sequences more effectively than previously discussed implementations.

Despite these advantages, there are also disadvantages. IPA is also very early in the compilation process, earlier than Tree-SSA, so other algorithms can likely optimize candidates better than procedural abstraction (the same way as in the procedural abstraction case on Tree-SSA). Another disadvantage is that real function calls may require a lot of instructions at the machine code level, and it is difficult to estimate the cost of them at this early stage. This means that we can only use heuristics to estimate the gain of a given abstraction because it is not possible to determine how many assembly instructions are in a GENERIC statement.

3.3.4 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.

In general, the hash table can be described as a data structure that implements an associated array, mapping keys to the corresponding values. The key is transformed by a hash function into an array index or address, called the hash code. These hash codes determine where the corresponding values

are stored in the table. The advantages of hash tables are well-known: fast access time, efficient insertion, deletion, and flexibility. Of course, the greatest disadvantage is also well-known: hash collisions.

When we implemented it, GCC also contained structures based on hash tables (e.g. string tables) but had not been used on instructions before. It was strange to us that GCC's developers did not use this quite well-known technique to compare instructions. 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. The main reason why we do not need to take care of hash collisions is that we are looking for a sequence of hash codes in case of procedural abstraction. This will highlight those sequences which should be compared against each other to identify the real candidates for the abstraction. This way the search space for candidates is much smaller compared to the case of the mentioned in the beginning of this section. We introduced three hash functions, one for each intermediate level. Thus, the complexity of our candidate search in algorithms became $O(n \log n)$ [60]. This was a huge improvement, especially for large compilation units. It was strange to see how an old but elegant solution could mitigate one of the general issues of GCC if it was used in the right way and place.

3.4 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* [10], 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 3.1 and 3.2, where the binary size in bytes and the relative improvement to `'-Os'` in percentage can be seen.

The results show (in Tables 3.1 and 3.2) that these algorithms are really effective methods for the optimization of code size, but further improvements may be needed in higher-level intermediate language representations due to the problems already mentioned to get better efficiency. For the target *i686-elf*, by running all algorithms implemented in addition to the `'-Os'` flag, we can achieve a maximum saving of 57.05% and 2.13% of average code size.

On the other hand, running all implemented algorithms together gives us a lower percentage of code savings than the sum of percentages for individual

algorithms. This difference arises because algorithms work on the same source code (function, compilation unit, or program), and previous passes can optimize the same cases that would also be modified by later methods. For *i686-elf* targets, running local factoring at RTL level and Tree-SSA allowed us to save 0.19% and 0.10% of the average code at *CSiBE*, while running these two algorithms only gave us 0.27%. This difference also proved that the same optimization method on different ILs may find different optimizable cases, and running the same algorithm on more than one IL will lead to better performance.

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 3.1: *Code-size reduction with code factoring algorithms.*

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

The proposed algorithms are publicly available [52] and part of the GCC code base in the CFO branch.

I. 3. Code Factoring Techniques in the GCC Compiler

4

Binary Code Size Measurement Methods and Benchmark

The most frequently used performance metric of compilers is the speed of the generated code. Although it is important, the generated code size is also a significant indicator, since from the industrial's point of view it can also mean huge savings in costs for embedded or other systems (e.g., for IoT devices, smartphones, TVs).

This is also represented in the compiler flags. For example, all compilers have a fine-tuned scale for runtime performance, implemented in combined optimization flags, such as `-O0` ... `-O3` (or even `-O4`). However, for code size optimization, compilers have only one single flag, the `-Os`. In addition, this can also be seen in the various benchmarks. Most of them are to test the runtime performance, not the code size (for example SPEC [89] benchmarks).

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, we cannot rely only on the assembly code, other parts of the software should be included in the measurement. The following sections discuss which parts of the software should be measured and how.

4.1 Fundamentals of Measuring Code Size

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).

One could expect there to be no difference between objects on these different targets. However, some differences between the libraries affect the objects as well. The library headers have to contain the same standard prototypes (e.g. standard functions), but there can be differences in the implementation of

certain features. For example, some standard names can be implemented both with macros and function calls. These can have various effects on the size of the code.

Clearly, then, measuring the size of executables incorporates a much larger impact of library code. It is apparently measurable in standalone executables. However, the situation becomes more complex when we investigate executables that are built for Linux. This is because Linux executables often do not embed the library code in the application binary, but keep only references to so-called shared objects linked at execution time. Even if static linking is used, some functions will be implemented in the operating system rather than in the executable.

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.

The different types of object files (e.g. *elf* [99] and *coff* [7]) can have different types of sections, and, in addition, different compilers can use different strategies to organize the code and the data into sections. More specifically, different compilers can divide some code into several parts or combine other components into a single section. For example, the elf file contains one (or more) initialized read-write data section, and the *coff* file contains program code to initialize the data during runtime. Thus, no common treatment could be used and the combination of sections to be incorporated in the measurements had to be determined separately for each measurement target.

In each case, we summarize only the size of the sections that contain generated code that is directly used by the program. These sections contain executable code and constant or initialized read-write program data. However, since executable code and constant data cannot always be clearly separated (there are constant data elements *hidden* in the executable code), we treat them together during comparison.

We have investigated two types of section combinations:

1. the size of sections containing program code or constant data (referred to as *read-only sections*), and

2. the size of sections containing any type of program code, constant, and initialized data (referred to as *all sections*).

We decided to follow the second approach because it seemed the most reasonable due to the various types of initialized read-write data and their relationship to the program code mentioned above.

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* [28]) is an appropriate tool to extract the size of specified sections from *elf* files. We were not aware of a similar tool for *coff* files. However, the *coffdump* [6] program extracts the sizes of sections from *coff* files, albeit not in a summarized form. Fortunately, all *coff* files have almost the same sections with the same names. We examined the contents of these sections and counted the appropriate sizes by hand.

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. If the host and target architectures are the same, the execution of the binaries is very easy. Otherwise, if the target architecture differs from the host (*cross-compilation*), a simulator could be a good choice. Another approach is to have specific hardware that can receive and execute standalone programs. In our environment, we can handle both mentioned cases. The simulator can be set via an environment variable, and specific hardware can be accessed via *SSH* command or another custom script.

Thus, we ran the programs and checked their outputs to validate the compiler toolchain with components of different versions and to check the correctness of different combinations of compiler options. In all our measurements, only the configurations that produced the correct and running programs were used.

4.2 CSiBE Benchmark

The fundamentals of measuring code size described in the previous section 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 [93]. The benchmark is called *CSiBE*, the Code Size BEnchmark [19].

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 back-end server and the web client represent a typical three-tier client/server system. It serves as a data server (via *Postgres*), implementing various query logics, and providing HTML presentations. All services run on Linux systems.

This online system is controlled by a so-called *master phase* on the front-end servers, which is responsible for the timely CVS checkout, compiler build, measurements using offline *CSiBE* benchmarks, and data population to the relational database. The main challenges of the *master phase* were the availability and correctness of the system. In the first decade of 2000, two mid-end PCs were sufficient to measure all target architectures. Basically, we donated this resource to the GCC community to have a *continuous integration-like* service that can detect code size changes daily. Later, the interest in code size increased rapidly, and dominant industrial companies and organizations began to enter the field of code size optimization (such as ARM, Linaro, LLVM Foundation, RT-RK).

The core of the *CSiBE system* is the offline *CSiBE* benchmark [20], which consists of the testbed and the required measurement scripts. The package can be downloaded from the official website [19] 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.).

The following projects are in *CSiBE* v2.1.1 (references are in Table 4.1):

bzip2 is a freely available, patent-free, high-quality data compressor. It typically compresses files to within 10% to 15% of the best available techniques (the PPM family of statistical compressors), whilst being around twice as fast at compression and six times faster at decompression.

cg-compiler-opensrc is a toolkit that provides a compiler for the Cg language, runtime libraries for use with both leading graphics APIs, and runtime libraries for CgFX.

compiler is the *vc* compiler for VSL that can be used to produce VSL abstract machine code for the VAM interpreter. VAM is a 32-bit machine, with 16 general-purpose registers, a program counter, a single-bit status flag, and up to 4G bytes of byte-addressed memory. It is a byte stream design but with a very reduced instruction set.

flex is a tool for generating scanners: programs that recognize lexical patterns in text. It can be used to build programs that handle structured input. It was designed originally to build compilers, but it has proven to be useful in many other areas.

jikespg, the Jikes Parser Generator is a parser generator that accepts as input an annotated description for a language grammar and produces text files suitable for inclusion in a parser for that language. It is similar in function and use to the widely available parser generators Yacc and Bison.

jpeg is a compression and decompression tool for JPEG images.

libmspack is a library for some loosely related Microsoft compression formats: CAB, CHM, HLP, LIT, KWAJ, and SZDD.

libpng is the Portable Network Graphics (PNG) Reference Library, an open, extensible image format with lossless compression.

linux kernel is a free and open-source, monolithic, modular, multitasking, Unix-like operating system kernel. This test case has our *test platform* implementation, which contains only C code.

lwip is a widely used open-source TCP/IP stack designed for embedded systems. The focus of the lwIP network stack implementation is to reduce resource usage while still having a full-scale TCP stack. This makes lwIP suitable for use in embedded systems with tens of kilobytes of free RAM and room for around 40 kilobytes of code ROM.

mpeg2dec is a free library for decoding mpeg-2 and mpeg-1 video streams.

mpgcut is a command line MPEG audio/video/system file cutter that allows to cutting of MPEG streams into playable chunks in many ways including time intervals, file offset intervals, or several parts. It can also handle the demultiplexing of video and audio streams from an MPEG file.

OpenTCP is a highly robust and portable implementation of the TCP/IP and Internet application-layer protocols intended for those who want to implement TCP/IP functionality in truly resource-constrained environments (8/16-bit MCUs).

replaypc is a simple text mode utility for extracting mpg files from ReplayTV Personal Video Recorders via TCP/IP. The ReplayPC utility is Win32 native but is being developed with an eye for easy porting to *nix operating systems.

teem is a coordinated group of libraries for representing, processing, and visualizing scientific raster data. Teem includes command-line tools that permit the library functions to be quickly applied to files and streams, without having to write any code.

ttt is the standalone traffic monitor program in the *ttt* program suite. It displays traffic data of a local interface. The *ttt* program suite is a descendant of *tcpdump* but it is capable of real-time, graphical, local, and remote traffic monitoring. It does replace *tcpdump*, rather, it helps to find out what to look into with *tcpdump*.

unrarlib, the *UniquE RAR File Library* is a platform-independent, small, and fast static library for decompressing RAR files. Full RAR v2.0 file format support of all compression methods, including multimedia compression and encoding is available.

zlib is a fast and unobtrusive compression library. It is designed to be a free, general-purpose, legally unencumbered - that is, not covered by any patents - lossless data-compression library for use on virtually any computer hardware and operating system. The zlib data format is itself portable across platforms.

Name	Reference URL
bzip2	`https://sourceware.org/bzip2/`
cg-compiler-opensrc	`https://developer.nvidia.com/cg-toolkit`
compiler	`https://www.jeremybennett.com/publications/download.html`
flex	`https://github.com/westes/flex`
jikespg	`https://jikes.sourceforge.net/`
jpeg	`https://www.ijg.org/`
libmspack	`https://www.cabextract.org.uk/libmspack/`
libpng	`http://www.libpng.org/pub/png/libpng.html`
linux kernel	`https://www.kernel.org/`
lwip	`https://savannah.nongnu.org/projects/lwip/`
mpeg2dec	`https://libmpeg2.sourceforge.io/`
mpgcut	`https://mpgcut.sourceforge.net/`
OpenTCP	`https://sourceforge.net/projects/opentcp/`
replaypc	`https://replaypc.sourceforge.net/`
teem	`https://teem.sourceforge.net/`
ttt	`https://github.com/esoule/ttt`
unrarlib	`https://unrarlib.org/`
zlib	`https://www.zlib.net/`

Table 4.1: *Tests and their references in the CSiBE benchmark*

The testbed is composed of two parts: one for code size measurement and the other for testing executable projects. This separation encompasses the entire benchmark, so the user would be able to add any custom test cases.

Since the code size measurement is the main focus of the benchmark, every project has a compilation method to produce binary files. Those projects that can be used for runtime measurement have a separate code path to produce the runtime results. For them, execution test cases have been selected to represent typical executions of the programs. The total size of the execution test input is currently about 60MB.

The table in Figure 4.2 shows some statistics on test projects. The number of source files, the size of the source code in bytes, the number of objects, the total size of objects measured using *CSiBE* and GCC 3.4 on *i686-gnu-linux* with *-O2* flag, and the number of executable programs for each project have been listed. Later, as the benchmark evolved, additional tests were introduced in *CSiBE*, such as Common Microcontroller Software Interface Standard [5] (*CMSIS*), *Servo* [63], and other small ones. In addition, new configuration files have been implemented to support other compilers, e.g. *LLVM*.

Project	# Src.	Src. bytes	# Obj.	Bin. bytes	# Exec.
bzip2-1.0.2	11	242,034	9	80,112	2
cg-compiler-opensrc	42	813,343	22	148,838	-
compiler	9	202,938	6	27,928	1
flex-2.5.31	33	658,799	22	240,206	1
jikespg-1.3	29	978,833	17	267,712	1
jpeg-6b	81	1,119,991	66	156,078	3
libmspack	40	319,611	25	76,506	-
libpng-1.2.5	21	859,762	18	128,941	2
linux-2.4.23-pre3-testpl.	2,430	34,238,976	271	993,815	-
lwip-0.5.3.preproc	30	928,538	30	86,486	-
mpeg2dec-0.3.1	43	461,047	29	62,873	1
mpgcut-1.1	1	28,889	1	29,845	-
OpenTCP-1.0.4	40	545,358	22	38,221	-
replaypc-0.4.0.preproc	39	1,692,413	39	64,221	-
teem-1.6.0-src	370	2,786,644	293	1,210,365	2
ttt-0.10.1.preproc	6	311,311	6	19,049	-
unrarlib-0.4.0	4	93,894	3	16,339	-
zlib-1.1.4	27	305,136	14	42,422	1
Total	3,256	46,587,517	893	3,689,957	14

Table 4.2: *CSiBE v2.1.1, testbed statistics*

4.3 Results

Since the start of the first code size measurement in 2003 [11], the *CSiBE* benchmark has made a lot of progress. It took less than a year and became

the official code size benchmark for GCC [93]. 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 [20] now contains even more complex tests, from the CMSIS, a common software package for microcontrollers, 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 [51] also uses *CSiBE* in their development workflows. After presenting the advantages of *CSiBE* [8] to their developer community, they began demonstrating improvements in code size with *CSiBE* (e.g. some recent public uses of *CSiBE* [50] are made by ARM and Linaro).

The Github statistics show that *CSiBE* v2.1.1 has been downloaded more than 54,000 times since 2016.

Today, the benchmark is maintained mainly by the author of the thesis. The present and future of *CSiBE* benchmarks are very clear. Industrial companies and other organizations still use benchmarks to represent their results. Today, *CSiBE* is used not only to optimize the code size but also for regression testing. We have seen many emails (e.g. in [81]) and papers (e.g. in [50], and [76]) focusing primarily on performance or memory improvements, but also taking the size of the code into account.

5
Related Work

At the time of the evaluation of this topic, there were very few algorithms implemented in GCC to reduce the binary code size, and none of them was based on code factoring, which grew in popularity in the 2000s. Developers recognized the power of these methods, and several applications used these algorithms for optimization purposes. One of these applications was the *Squeeze Project* [17] maintained by Saumya Debray, and was one of the first to use this technique.

Another application is the *aiPop* (Automatic code compaction software), a commercial program published by AbsInt Angewandte Informatik GmbH with a functional abstraction (reverse inline) for common basic blocks' function [3]. The application is an optimizer software suite supporting *C16x/ST10*, *HC08*, and *ARM* architectures, and is also used by Siemens.

Although these tools entered the market, they have focused on a very small segment. None of them tried to cooperate with any of the open-source compilers, which could have helped them to reach a larger audience. Our solutions are more general and can be used on the architectures supported by GCC.

Nowadays commercial compilers adopted at least the procedural abstraction algorithm. Most of the time they refer to it as reverse inlining. For example,

the LLVM-based Wind River Diab compiler[2] of Wind River Systems, compiler toolchains of HighTec EDV-Systeme[3], and MPLAB compilers of Microchip Technology[4] have this algorithm implemented in their compiler toolchains.

As for code size measurement, we are not aware of any prior art or attempt to create a code size benchmark for compilers before our paper. According to our best knowledge, compiler code optimization research and papers were focused on the proposed algorithms instead, and let the readers decide which measurement methodology and tests should be used. There was no de facto standard for that purpose. Thus, the measurement methods were neither discussed by others.

[2] https://www.windriver.com/themes/Windriver/pdf/PN_Compiler_0110.pdf

[3] https://hightec-rt.com/en/products/development-platform

[4] https://ww1.microchip.com/downloads/en/DeviceDoc/MPLAB_C18_Users_Guide_51288c.pdf

6 Conclusions

In this part of the thesis, topics related to executable code optimizations were presented, which cover not only compiler optimizations to save code size but also measurement methods and a benchmark, which became the official benchmark of GCC.

First, several code size optimization techniques have been introduced which had impressive effects. These algorithms were local code factoring and procedural abstraction. Both have been implemented in the RTL and Tree-SSA intermediate languages of GCC, and procedural abstraction has also been provided for the interprocedural abstraction phase. We found, with the help of the *CSiBE* benchmark, that the algorithms can achieve a maximum of 61.53% and an average of 2.58% reduction in code size compared to the baseline option `'-Os'` of GCC. In addition, a very simple optimization technique, hashtables, was used to improve the running time of the algorithms and thus the total compilation time. The proposed code factoring algorithms are publicly available [52] in the CFO branch. The papers that form the basis of this part of the thesis have also inspired recent research works [79, 80, 91], making code size optimization important in other areas as well.

As the second topic, the measurement method of code size has been presented, which led to the birth of *CSiBE*, GCC's official code size benchmark. Many aspects of code size measurements have been evaluated to make the

benchmark the de facto code size measurement standard for compilers. During the many years of development of *CSiBE*, it has taken its well-awarded place alongside other benchmarks, such as SPEC [89] and the later Openbench [23].

After its original release, as code size measurement and monitoring were getting more important, the developers of another major open-source compiler toolchain also started using it. This was the LLVM Compiler Toolchain, which is used to produce many modern applications from smartphone OS to powerful laptops. For the validation of code size changes and for regression testing, companies and organizations still use *CSiBE* (e.g. in [50, 76, 81]).

Part II

Just-In-Time Compilers' Optimizations and Analyses

7 Overview

In recent years, the spread of web applications with various complexity and functionality has made JavaScript [88] a cutting-edge technology in modern programming languages. The popularity of JavaScript in web development is still increasing due to its ability to facilitate the creation of dynamic and interactive web interfaces. However, the growing demand for complex web applications with real-time functionality [37] highlights the need to optimize JavaScript source code to improve performance and efficiency.

JavaScript developers generally have two main options to enhance their JavaScript programs. One dates back to a time, when the JavaScript execution engines were based on traditional interpreters, and the developers followed various tips, tricks, and advice to improve the source code. The other option is the usage of different tools which can reveal the structural relationship between the source code and the bottleneck of their software.

To understand the first option we have to look back to the history of JavaScript, which started in 1995 with *Netscape Navigator*. After some time the *browser war* has started. This inspired the entire industry to create better and more efficient JavaScript codes. In addition, the JavaScript language just started to be standardized in 1996. Ecma International[5] was who adopted the language. This is how ECMAScript was born and became the standard

[5] `http://www.ecma-international.org/`

for scripting languages, including JavaScript, JScript, and ActionScript. At that time there was no such sophisticated tool that could aid developers in revealing bottlenecks in their scripts. Thus, they were starting to write tips, measurement reports, suggestions, and advice on the web to help each other. Several people tried to summarize these, which became unofficial guidelines that every JavaScript developer should know, but the landscape changed again, and new state-of-the-art JavaScript engines arose using efficient algorithms to speed up the source code. However, the software engineers still use the guidelines. The question now is if these guidelines are still useful.

On the other hand, nowadays, developers have more tools to improve their software. These tools [1, 25, 33, 35, 86] usually optimize or analyze source code and either automatically perform or interactively suggest changes. Typically, optimization techniques include static pre-execution analysis and code conversion. Although these are effective in many programming environments and languages, the dynamic and event-based nature of JavaScript imposes challenges [48, 78] to static optimization.

Dynamic optimization is an emerging paradigm in programming languages that includes strategies to improve program performance during execution. These strategies [31, 39] use runtime profiles, and adaptive and Just-In-Time (JIT) compilation methods to optimize key code paths based on real usage patterns, data characteristics, and environmental factors. A particularly relevant topic of dynamic optimization, alongside performance, is in the field of call chains. Call chains embody the structural blueprint of applications, revealing the dynamic relationships between functions, especially in dynamic languages such as JavaScript. They aid in code comprehension, debugging, and profiling, forming a foundation for efficient program analysis [12, 87, 104]. In the long run, these can be used to optimize the JavaScript application source to be more fail-proof, effective, and fast.

This part of the thesis will give an overview of the JavaScript guidelines in Chapter 8 which were the first experiments improving the runtime performance. Chapter 9 will then give an overview of dynamic call graph generators which can be used to understand the underlying structural blueprint of JavaScript applications and can be the basis of other software analyses. These chapters are based on the results of our previously published papers [4, 40, 41, 42, 43, 53]. The results and the conclusions of this part are summarized in Chapter 11.

8 JavaScript Guidelines

Before the time of JIT engines, several guidelines were published [74, 101, 105, 106] 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.

8.1 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. In this section, we collect guidelines that have been proposed over the years, for various JavaScript engines and versions, to allow their systematic evaluation (the results of which will

be presented later in Section 8.4). The main focus area is in this section the ECMAScript 5.1 standard [21], the first high-impact version of JavaScript.

Using Local Variables Each time a variable is accessed in JavaScript, a complex search method is started that searches the entire scope chain. However, it is known that all widespread execution engines speed up the lookup of local variables. The guidelines therefore recommend the use of local variables instead of global variables whenever possible. The example transformation can be seen in Figure 8.1

```
for (i = 0; i < 10000000; ++i) ;
```

(a)

```
for (var i = 0; i < 10000000; ++i) ;
```

(b)

Figure 8.1: *Using local variables instead of global ones.*

Using Global Static Data In general, the guidelines recommend the use of local variables rather than global ones, as described above. However, there exist variables that can be defined within functions but last throughout the entire run of the program called static variables. JavaScript (at least the 5.1 version of the standard) does not support the concept of constants or static variables, and initialization is nothing more than assignments.

For example, if a literal array is used in the assignment, as shown in the example in Figure 8.2a, the array is constructed every time the execution reaches the assignment. These unnecessary operations may take a long time, even longer than the cost of searching for a global variable. So, the guidelines suggest that, in this special case, globally initialized variables should be used, as shown in Figure 8.2b.

Caching Object Members If the same object member is accessed several times in a script (e.g. in a loop), it is recommended to cache the member value in a local variable, as shown in Figure 8.3. The reason for this is similar to the explanation given in the Using Local Variables paragraph, where the resolution of the member is slower than the lookup for the local variable.

```
function hexDigit (s) {
  var digits = ["0","1","2","3",
  "4","5","6","7",
  "8","9","a","b",
  "c","d","e","f" ];
  return digits[s];
}
for (var i = 0; i < 5000000; ++i)
  hexDigit(i & 0xf);
```

(a)

```
var digits = ["0","1","2","3",
"4","5","6","7",
"8","9","a","b",
"c","d","e","f"];
function hexDigit (s) {
  return digits[s];
}
for (var i = 0; i < 5000000; ++i)
  hexDigit(i & 0xf);
```

(b)

Figure 8.2: *Moving static data out of functions.*

```
var o = {a: 678,b: 956}
var r
for(var i=0;i<30000000;++i)
  r = o.a + o.b
```

(a)

```
var o = {a: 678,b: 956}
var r
var ca = o.a
var cb = o.b
for(var i=0;i<30000000;++i)
  r = ca + cb
```

(b)

Figure 8.3: *Caching object members in variables.*

Avoiding With The *with* language construct of JavaScript adds a calculated object at the top of the scope chain and executes its body with this augmented scope chain. If the chain of object references or object names is very long, this is a very useful syntactic feature, but in practice, it increases execution time. Again, the guidelines suggest that it is possible to achieve a better performance result if local variables are used to access object members instead of *with* statements (see Figure 8.4).

Creating Objects The most important suggestion of the legacy guidelines on object creation is to avoid creating objects like in object-oriented languages

```
var o = new Object()
o.ext1 = new Object()
o.a = 23
o.ext1.ext2 = new Object()
o.ext1.b = 19
o.ext1.ext2.c = 36
with (o) {
  with (ext1) {
    with (ext2) {
      for(var i=0;i<2000000;++i)
        a = b + c
} } }
```

(a)

```
var o = new Object()
o.ext1 = new Object()
o.a = 23
o.ext1.ext2 = new Object()
o.ext1.b = 19
o.ext1.ext2.c = 36
var ext1 = o.ext1
var ext2 = ext1.ext2
for(var i=0;i<2000000;++i)
  o.a = ext1.b + ext2.c
```

(b)

Figure 8.4: *Avoiding with statements.*

(OO), as this type of object creation must be solved by a function call. It is suggested to use the JavaScript Object Notation (JSON) form to specify the object literals in the script code. Figure 8.5 shows the possible methods for object creation: Subfigure 8.5a implements object creation in an OO way, Subfigure 8.5b shows an inline solution for object creation, and Subfigure 8.5c gives an example of JSON-based object creation.

Avoiding Eval The *eval* function evaluates a string and executes it as a script code. This language function can help conceal or obscure script code, and it can also help execute dynamic script code, but it has its costs. Each string passed to the *eval* function must be analyzed and executed on the fly. This cost must be paid each time the execution reaches an *eval* function call. Thus, avoiding *eval* is regarded as a good idea whenever an alternative solution is possible, as shown in Figure 8.6.

Function Inlining Function inlining is a traditional compiler optimization technique [65] that replaces a function call with the body of the function called. In JavaScript, making a function call is an expensive operation. It takes several preparatory steps to perform: allocate space for parameters, copy parameters, and resolve function names. The function inlining can save the cost of these

```
function create() {
  this.str = "String"
  this.int = 56
  this.num = 6.7
  this.get = function()
  { return this.int }
}
var object
for(var i=0;i<3000000;++i)
  object = new create()
```

(a)

```
var object
for(var i=0;i<3000000;++i) {
  object = new Object()
  object.str = "String"
  object.int = 56
  object.num = 6.7
  object.get = function()
  { return this.int }
}
```

(b)

```
var object
for(var i=0;i<3000000;++i) {
  object = {
    str: "String",
    int: 56,
    num: 6.7,
    get: function()
    { return this.int }
  }
}
```

(c)

Figure 8.5: *Creating objects.*

```
function funcs() { return " " }
function funcd() { return "." }
function funcl() { return "_" }
var code = "dsdllsdsdlls";
var len = code.length
var res = ""
for (var j = 0; j < 50000; ++j) {
  for (var i = 0; i < len; ++i)
    res += eval(
      "func"+code.charAt(i)+"()")
}
```

(a)

```
function funcs() { return " " }
function funcd() { return "." }
function funcl() { return "_" }
var code = "dsdllsdsdlls";
var len = code.length
var res = ""
for (var j = 0; j < 50000; ++j) {
  for (var i = 0; i < len; ++i)
    switch (code.charAt(i)) {
    case 's' : res += funcs() ; break
    case 'd' : res += funcd() ; break
    case 'l' : res += funcl() ; break
  }
}
```

(b)

Figure 8.6: *Avoiding eval.*

steps (e.g., Figure 8.7). (For the sake of completeness, we give two function-call-based implementations in the example next to the inline version; Subfigure 8.7a shows the call of a user-defined function, and Subfigure 8.7b uses a built-in function.)

```
function abs(a) {
  return a>=0 ? a : -a
}
var a
for(var i=0;i<8000000;++i)
  a = abs(4000000-i);
```

(a)

```
var a
for(var i=0;i<8000000;++i)
  a = Math.abs(4000000-i);
```

(b)

```
var a
for(var i=0;i<8000000;++i)
  a = (4000000-i) >= 0 ?
      (4000000-i) :
      -(4000000-i);
```

(c)

Figure 8.7: *Function inlining.*

Common Subexpression Elimination Common subexpression elimination (CSE) is another performance-oriented compiler optimization technique [65], that searches for identical expression instances and replaces them with a single variable that stores the calculated value. In the guidelines, it is recommended to do this manually (see Figure 8.8), as typical JavaScript engines do not support this optimization. The use of a single local variable for a common subexpression is expected to be always faster than leaving the code unchanged.

```
function get_roots(a, b, c) {
  var ret = {
    x1:((-b+Math.sqrt(b*b-4*a*c))
       /(2*a)),
    x2:((-b-Math.sqrt(b*b-4*a*c))
       /(2*a))
  }
  return ret
}
for (var i = 0; i < 2000000; ++i)
  get_roots(i & 0xff, i & 0x7, 10)
```

(a)

```
function get_roots(a, b, c) {
  var sq = Math.sqrt(b*b-4*a*c);
  var ret = {
    x1: ((-b + sq) / (2*a)),
    x2: ((-b - sq) / (2*a))
  }

  return ret
}
for (var i = 0; i < 2000000; ++i)
  get_roots(i & 0xff, i & 0x7, 10)
```

(b)

Figure 8.8: *Common subexpression elimination.*

Loop Unrolling Loop unrolling [98] is another compiler optimization technique (Figure 8.9) suggested by the guidelines to be applied manually. This is much more efficient if the body of the loop is small, but the loop runs long. The performance gain comes from the absence of most loop tests and loop test instructions.

```
var iterations = 100000000
var counter=0
for(i=iterations;i>0;--i) {
  counter++
}
```

(a)

```
var iterations = 100000000
var counter=0
var n = iterations % 8
if (n>0)
  do {
    counter++
  } while (--n)
n = iterations >> 3
if (n > 0)
  do {
    counter++
    counter++
    counter++
    counter++
    counter++
    counter++
    counter++
    counter++
  } while (--n)
```

(b)

Figure 8.9: *Loop unrolling.*

Optimizing Loop Indices With regard to loop indices, the guidelines generally make two recommendations. First, the post-increment and decrement operators of the loop index variable should be replaced, if possible, by pre-increment and decrement operators. Second, it should be preferred to decrementing a loop rather than incrementing it. The reason for the first recommendation is that post-operators require more machine instructions to execute than pre-operators, but JavaScript engines normally do not automatically transform post-increment and decrement operators into their "pre-" counterparts, i.e. if

the value returned by the operation is not used. Secondly, decrementing a loop is preferable over incrementing it because a comparison to zero is much faster in almost every architecture than a comparison to any other number.

HTML DOM Almost all guidelines contain recommendations for optimizing HTML Document Object Model (DOM) based object accesses, such as dynamic HTML generation. The most typical recommendation is not to access DOM objects too often because DOM bindings are slow. However, these guidelines are not JavaScript language-specific but browser-specific, so they are not within the scope of this research.

8.2 ECMAScript 6-based Guidelines

The ECMAScript 6 standard [22] 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.

Arrow Function The *arrow function* is a shorter syntax of a function expression and does not have its own *this*, arguments, or *super* constructs. Many developer discussions suggest using arrow functions when using non-method functions. Figure 8.10a shows an arrow function simulation in ECMAScript 5.1, while Figure 8.10b describes the new standard for it.

Class Definition 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. In ECMAScript 6, the main motivation was to bring JavaScript closer to object-oriented programming languages. Figure 8.11a introduces an example simulation of the class, while Figure 8.11b shows the new construct.

```
array.map(function(it){
  return it * local_var;})
```

(a)

```
array.map(it => it * local_var)
```

(b)

Figure 8.10: *Arrow Function*

```
var cat = function(name) {
  this.name = name;
  this.speak = function () { v++ }
}
var lion = function(name) {
  parent = new cat(name);
  this.speak = function() {
    parent.speak(); v++;
} }
```

(a)

```
class Cat {
  constructor(name) {
    this.name = name;
  }
  speak() { v++; }
}
class Lion extends Cat {
  speak() { super.speak(); v++; }
}
```

(b)

Figure 8.11: *Class Definition*

Enhanced Object Properties 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. This brings the object literals closer to the class definition. Figures 8.12a and Figure 8.12b show the ECMAScript 5 and 6 approaches, respectively.

Template Strings Template strings provide easy-to-use syntax for creating different strings from previously defined templates. Many languages use similar types of templates, such as Linux's Bash, C#, Perl, and Python. The motivation behind this feature is to extend JavaScript with well-accepted template constructions from other languages. Figure 8.13a shows the old simulated way, and Figure 8.13b shows the standard of the ECMAScript 6.

```
var car = {
  make: make, value: value,
  dep: function dep() {
    this.value -= 2500;
} }
car['make' + make] = true;
```

(a)

```
var car = {
  make, value,
  ['make' + make]: true,
  dep() { this.value -= 2500; }
}
```

(b)

Figure 8.12: *Enhanced Object Properties*

```
'a% '+(isOK()?"":(test?"!d":"12"))
```

(a)

```
'a% ${isOK()?"":(test?"!d":"12")}'
```

(b)

Figure 8.13: *Template Strings*

Tagged 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. The simulated (a) and the new standard (b) methods can be seen in Figure 8.14.

```
myTag({0:"that ",1:" is a "},
      person,
      age)
```

(a)

```
myTag'that ${person} is a ${age}'
```

(b)

Figure 8.14: *Tagged Templates*

Destructuring Objects 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

district variables. This could be very useful for developers in many programming situations. Similar language features can be seen in other script languages (e.g. Python). Figure 8.15 presents the simulated (a) and the ECMAScript 6 (b) approaches.

```
a = 10; b = 20;
var _ref = [10,20,30,40,50];
a = _ref[0]; b = _ref[1];
rest = _ref.slice(2);
```

(a)

```
[a,b] = [10,20];
[a,b,...rest] = [10,20,30,40,50];
```

(b)

Figure 8.15: *Destructuring Objects*

Spread Operator ECMAScript 6 has introduced extended parameter handling. The most important is the *spread operator*, which spreads the elements of an iterable collection (such as arrays or strings) into individual elements or function parameters. An example of a simulated version of ECMAScript 5 can be seen in Figure 8.16a, while the ECMAScript 6 standard uses a much more straightforward design (Figure 8.16b).

```
function f(x, y, z) {
  return x + y + z;
}
var a = [1, 2, 3];
f(a[0],a[1],a[2]);
```

(a)

```
f(...a);
```

(b)

Figure 8.16: *Spread Operator*

Constants One of the most significant changes to ECMAScript 6 is that it is possible to create constant values in JavaScript. The *const* construction is

defined to contain only constant values. In ECMAScript 5, only the values stored in the global lexical scope can be configured as constants.

Iterators This feature allows objects to customize their iteration behavior. In addition, it supports the iterator protocol to generate a value sequence and provides a convenient method to iterate all values of an iterable object. Figure 8.17 shows the ECMAScript 5.1 (a) and ECMAScript 6 (b) approaches.

```
for(var i=0,n=array.length;i<n;i++)
  array[i]
```

(a)

```
for (var value of array)
  value
```

(b)

Figure 8.17: *Iterators*

Generators Many languages contain generators and *yield* constructs. The same functionality has been added to the ECMAScript 6 feature set. The generators are subtypes of iterators that include additional *next* and *throw* functions. This allows the values to flow back into the generator so that the *yield* can be returned with the next value. Figure 8.18 shows the ECMAScript 5.1 (a) and ECMAScript 6 (b) approaches.

Map Structure In ECMAScript 6, some effective data structures were introduced for common algorithms (e.g. Map, Set, WeakMap, WeakSet). Since they are not part of the ECMAScript 5 standard, previously function objects have been used to implement the same functionality. In our evaluation, we focus on the structure of *Map*, as the main logic is similar to the others. Figure 8.19 describes an example of this structure in ECMAScript 5.1 (a) and the way in which it can be used with the ECMAScript 6 (b) standard.

Symbols In ECMAScript 6, there is a new feature called *Symbol*, a global symbol indexed by a unique key. Each symbol value returned from *Symbol*() call is unique. Since the simulated implementation of ECMAScript 5 is long, the research does not present examples of this feature.

```
function foo(param){
  this.param = param;
  this.next = function() {
    var res;
    var done = true;
    if (param >= 1) {
      res = param;
      param = param - 1;
      done = false;
    }
    return {value:res, done:done};
} }
```

(a)

```
function* foo(param){
  while (param >= 1) {
    yield param;
    param = param - 1;
} }
```

(b)

Figure 8.18: *Generators*

```
m["abc"] = 123
m[576]
"abc" in m
```

(a)

```
m.set("abc", 123)
m.get(567)
m.has("abc")
```

(b)

Figure 8.19: *Map Structure*

Binary Literals In ECMAScript 6, binary literals can be entered. ECMAScript 5 provides only octal, decimal, and hexadecimal numeric literals. The new standard has added support to describe binary and an alternative syntax for octal numbers (Figure 8.20 (a) and (b) for 5.1 and 6 versions of ECMAScript, respectively). This can help developers when representing numbers for binary operations (such as binary *or*, *xor*, *and*, and negation).

```
parseInt("111110111",2)===503
0767 === 503
```

(a)

```
0b111110111 === 503
0o767 === 503
```

(b)

Figure 8.20: *Binary Literals*

8.3 Static Optimization Difficulties

The previous sections of the guidelines provide instructions to JavaScript programmers on how to write effective code. However, 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.

First, consider Figure 8.21. The loop of the function *test1* is allegedly infinite, and it continuously prints *"Hello World!"* messages. However, the example loop stops after three iterations due to the parameters used in the function call. This parameter is passed to *eval* and redefines the *print* identifier from the internal function to a user-defined one. Furthermore, since the new *print* implementation is defined in the scope of the *test1* function, it can also access its local variables. Since the loop index variable is increased each time the *print* is called, the loop is terminated in this case.

The example in Figure 8.22 produces the same results as in Figure 8.21 but achieves it in different ways. The code shows that one does not have to use *eval* to get hardly predictable results. In this example, the setter function is used to turn variables *b* into function calls. As in the previous example, local variables of the function *test2* can also be accessed in the called function. In addition, as shown in the example, the definition of the setter method can be obfuscated; it is done via the call of the *def* function in this case. Therefore, theoretically, any function call can be a setter function.

```
function test1(cmd) {
  var a = 0
  eval(cmd)
  while (a < 3)
    print("Hello world!")
}
test1("var pr = print; print = function(text) { a++ ; pr(text) } ")
```

Figure 8.21: *Eval, function redefinition, and access to local variables.*

```
var def = __defineSetter__
function test2(name) {
  def(name, function(value) { print("Hello world!") ; a++ } )
  for (a = 0 ; a < 3 ; /* Do nothing */ ) {
    var a
    b = void(0)
  }
}
test2("b")
```

Figure 8.22: *Setter function.*

The last example in Figure 8.23 shows an unusual use of *valueOf*. The *valueOf* method is implicitly called when the operator needs the primitive value of an identifier. Therefore, in this case, the loop test implicitly increases the loop index. Unfortunately, this effect is completely invisible to a static analyzer of the *test3* function.

```
var x = 0
Number.prototype.valueOf = function() { return x++ }
function test3(a) {
  while (a < 3)
    print("Hello world!")
}
test3(new Number(0))
```

Figure 8.23: *Overriding the valueOf() method.*

The examples above show several unpredictable changes that can occur in variables and functions that static optimization algorithms cannot predict. 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. Thus, it seems that guidelines are still just guidelines and not rules that should be applied blindly. The following section describes the effectiveness of these guidelines.

8.4 Results

In this section, we compare the various guidelines. We chose to measure on a *Raspberry Pi 3 Model B*, as it is a platform that all JavaScript engines can be applied to and it is situated between the embedded and desktop worlds. Desktop engines usually offer better performance but at the cost of code size and memory consumption, which is not feasible in a restricted environment. The Raspberry Pi 3 Model B provides the best of both worlds, as it allows desktop engines to be executed without sacrificing performance optimizations.

We used the following hardware and software environment: BCM2835 ARMv7 quad-core CPU, 1GB DDR2 memory, 4GB Class 10 SD card with a Raspbian GNU/Linux 8.0 (jessie) OS and a Linux Raspberry Pi 4.9.35-v7+

kernel image. The measurement framework was written using Python 2.7.9 and Bash scripts.

The measurement methodology was the following:

- Each legacy guideline has an original and a transformed code snippet.
- Each ECMAScript 6 guideline has example codes for versions 6 and 5 as well.
- The measurement framework extends every test with utility functions to measure and save elapsed time within the main test cases.
- Since desktop engines still perform better than embedded ones, an additional loop iteration was introduced for desktop engines.
- Each measurement has been executed twenty times and the median of the results was used to compute the relative percentages on the figures.

The JavaScript engines are tailored to various software and hardware configurations. Several are designed for desktop computers, others for embedded systems, and some are intended to work with both. As a result, certain features may be absent or implemented differently in different JavaScript engines. To find out which features are supported by each engine, there are online comparison tables available [6].

On the other hand, in the JavaScript engines that are targeting low-end hardware or focusing on supporting machine-to-machine communication, it is not so evident to support all JavaScript language features. Our evaluation shows that these embeddable engines do not support the full spectrum of ECMAScript 6 language constructs. There is no single language construct that is supported by all the three embeddable engines (see below), so we cannot do a conclusive evaluation of these engines.

In this research, six JavaScript engines are examined:

- *JavaScriptCore*: It is the JavaScript engine of Safari and WebKit-based web browsers. It can be found on iPhone and Mac desktop machines. (main branch on 2017-12-10)
- *V8*: it is the main JavaScript engine of Google's Chrome and Chromium-based web browsers [34] as well. Most smartphones delivered with Android OS have it, and of course, common desktop machines can use Google's Chrome web browser. (Version: 6.4.99)

[6] `https://kangax.github.io/compat-table/es6/`

- *Spidermonkey*: This engine is used by Mozilla's Firefox web browser. Firefox can run on phones, tablets, and desktop computers. (Version: 59)

- *JerryScript*: This engine is the first on this list that is targeted only at the embedded world. This engine is developed by Samsung, Intel, ARM, and the University of Szeged in partnership with other open-source community members. It is running inside several smartwatches. (Version: 1.0 - da24727)

- *Espurino*: It is a JavaScript interpreter for microcontrollers. The supporting company also built a hardware stack around the software. (Version: 1v95)

- *Duktape*: An easily embeddable ECMAScript E5/E5.1 engine with a small footprint. (Version: 2.2.0)

In this part of the thesis, our target is 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 8.24 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 8.25) on average and the other in which the new standard has more efficient code paths in the engines (Figure 8.26). Both figures show the change in relative execution time (smaller numbers are better), but the basis of comparison is different. For Figure 8.25 the base is ECMAScript 6, and for Figure 8.26 the base is ECMAScript 5 variant code snippets. Based on the findings, we can set out the following guidelines for ECMAScript 6 language features:

- *Arrow Function*: Use the arrow functions if a non-method function is needed. The reason for this is that although JavaScript engines reveal very small performance improvements, the ECMAScript 6 form is clearer and easier to adopt.

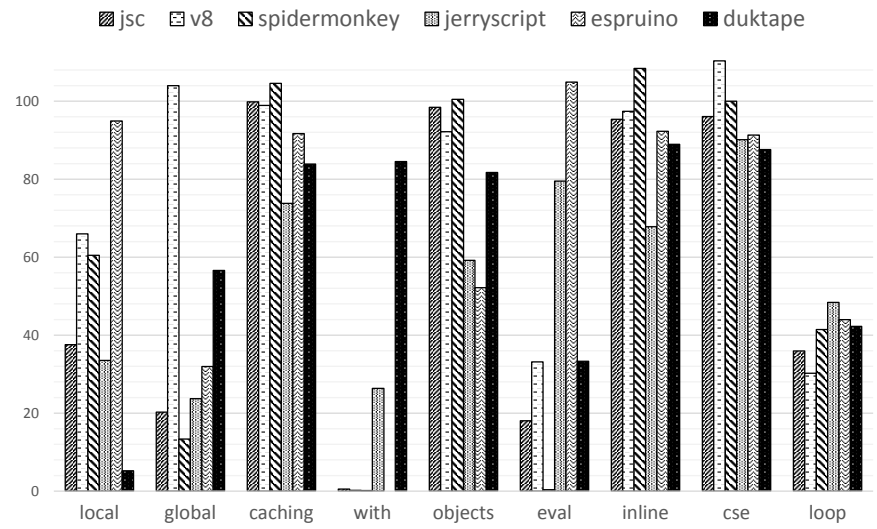


Figure 8.24: *Performance Improvement with Legacy Guidelines*

- *Class Definition*: Use the *class* definition. The reason for this is that most JavaScript engines perform better by using fast-path implementations. The new standard can have as much as a 95% reduction in the running time.
- *Enhanced Object Properties*: Do not use the enhanced object properties in the ECMAScript 6 form. Use the ECMAScript 5 variant instead. The results show that significant speedup can be seen with most execution engines if the old standard is used.
- *Template Strings*: There are no significant changes when using the ECMAScript 5 or 6 version of the template strings, so there is no clear conclusion about this construct. In this case, we suggest following the new standard. The engines may improve this code later.
- *Tagged Templates*: Use tagged templates. Engines implement a special code path for this construct. One of the engines outperforms the others with a runtime improvement above 99%.
- *Destructuring Objects*: Do not use the destructuring construct. The

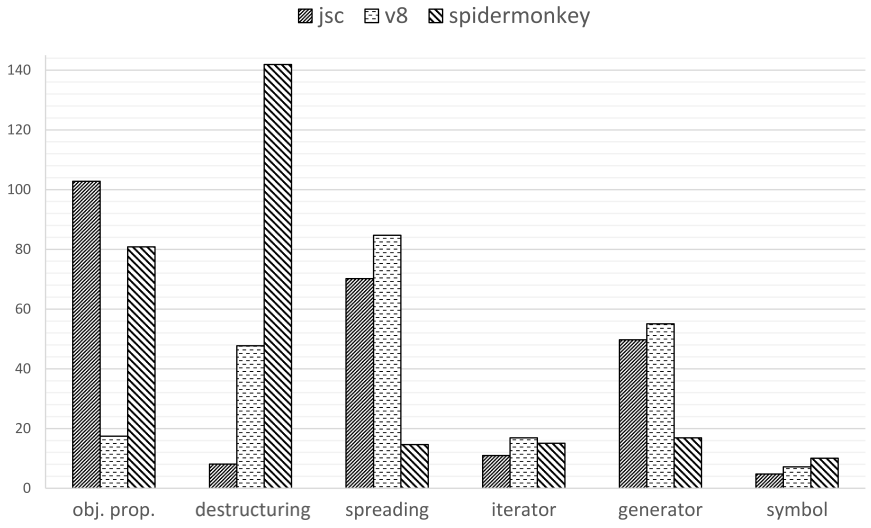


Figure 8.25: *Performance Improvement with ECMAScript 5*

reason for this is that it significantly slows down almost all engines, except one.

- *Spread Operator*: Do not use the spread operator. Most JavaScript engines perform better when using the ECMAScript 5 form.

- *Constants*: Use the constant construct. All engines perform better with *const*. A special code path has been implemented for this.

- *Iterators*: Do not use iterators. The reason for this is that the ECMAScript 5 form is still faster. Currently, there is no fast-path implementation for this construct in the engines.

- *Generators*: Do not use generators. The reason for this is the same as in the iterator case.

- *Map Structure*: Use the new built-in structure, e.g. Map construct. The engines implemented these features with a fast code path.

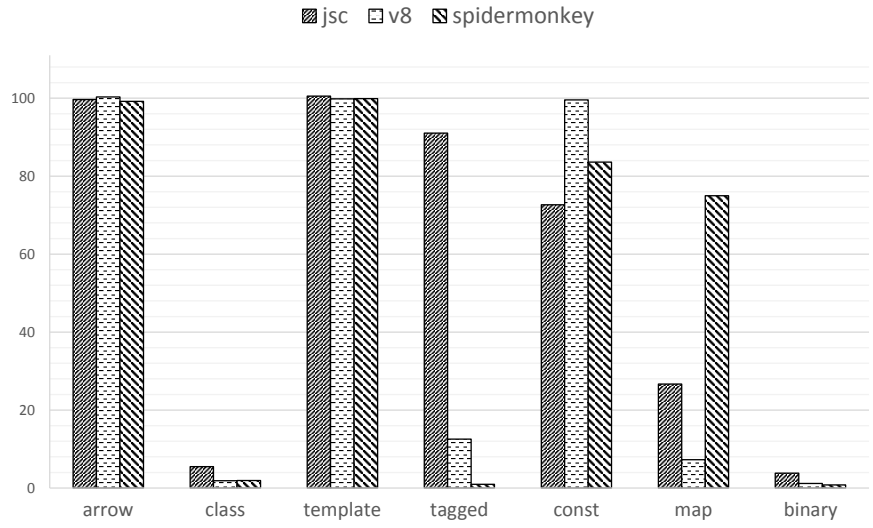


Figure 8.26: *Performance Improvement with ECMAScript 6*

- *Symbols*: Do not use the new symbol standard. Although JavaScript engines have a new code path for this feature, simulating it is currently faster.
- *Binary Literals*: Use the binary literals. The reason for this is that the new feature implementations are very fast and there is no need to call any parsers to read binary literals.

9

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). The former has been extensively studied [26, 27, 45, 56], and first, we focus on the latter to enable the dynamic analysis of JavaScript programs.

Call graphs have been used successfully in fault localization [77, 97], which is the process of pinpointing the cause of a test failure without human intervention. This can save time and money in problem resolution. When comparing the call graphs of successful and unsuccessful tests, it is possible to identify the functions most likely to contain an implementation error and focus on them.

These methods rank functions according to the probability that they contain an implementation error.

A considerable amount of web applications and back-end services have been developed using the Node.js framework [73]. With more than 6.3 million websites using Node.js, it has become the most widely used tool for web development [75, 90] 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).

9.1 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. The following subsections discuss three tools that use distinct approaches to create call graphs.

9.1.1 Call Graph Generator Tools

Jalangi2 The Jalangi2 framework [83], an improved version of Jalangi [82], is the first tool used to dynamically analyze the ECMAScript 5.1 code [21], and is compatible with Node.js and multiple web browsers.

Jalangi2 is an instrumentation framework that works with ECMAScript 5.1 source code, adding event notifications without altering the observed behavior. Events such as variable assignments, expression evaluations, and entering and exiting functions can be captured by Jalangi2. These events are then processed by a JavaScript application called Jalangi2 analyzer. For example, we have created an analysis for dynamic call graph construction [54], which registers handlers for function entry/exit events to collect the nodes and edges of a call graph.

Jalangi2 is distinct from other tools in this category, as it is a JavaScript-based framework that alters the source code of a JavaScript application and then runs the modified code using Node.js.

	nodejs with tracing enabled	Nodejs-cg
running time	48 s	5 s
disk space	161 MB	0.9 MB

Table 9.1: *Running time and disk space consumed by the* express *module.*

NodeProf NodeProf [92] is a dynamic program analysis tool for Node.js applications that is based on the Graal-nodejs project. This project utilizes the Graal.js [102] engine to interpret JavaScript code and convert it into an abstract syntax tree (AST) representation that is then executed by the GraalVM virtual machine. The Graal.js engine is compliant with the ECMAScript 2017 standard.

NodeProf is similar to Jalangi2 in that it adds event notifications to JavaScript programs. 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. To expedite the implementation, NodeProf utilizes Jalangi2 analysis to process events, although the interface is not completely compatible. Fortunately, we were able to modify and reuse the analysis of the call graph generator [54].

Nodejs-cg We have developed a customized version of Node.js, known as Nodejs-cg [54], 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.

Our approach collects all nodes and edges when a Node.js application is executed and stores the entire graph in memory. This allows for much faster execution of JavaScript with minimal printing. Table 9.1 shows that our approach is 10 times faster and requires 100 times less space than postprocessing the tracing output.

Unlike other tools in this section, call graphs are generated directly by Nodejs-cg's JavaScript engine. The source code or the intermediate representation is not modified by the generator tool. Thus, Nodejs-cg's JavaScript engine produces call graphs directly, without altering the source code or intermediate

representation.

9.1.2 Node 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, the locations provided by NodeProf and Nodejs-cg are often different, with Nodejs-cg indicating the start of the function argument list and NodeProf indicating the start of the function. To identify the same nodes in the call graph returned by these two tools, the NodeProf location must be converted to the location returned by Nodejs-cg using the source code.

We have made a significant improvement in that explicit constructor nodes are identified as the same nodes. As illustrated in Figure 9.1, when a constructor is called in JavaScript, the NodeProf' and Nodejs-cg's locations are the starting points of the *class* keyword and the starting points of the constructor argument, respectively. These location places are unified in the final call graph. It is also worth mentioning that JavaScript allows for dynamic script evaluation, meaning scripts are not stored in files, but are strings constructed at the time of execution. This means that they do not have path information. To address this, some heuristics may be designed to add unique identifiers to these strings, however, identifying an element in such a dynamic code is a complex task. For now, all of these scripts are assigned to a single node with an <*eval*> identifier.

```
class ClassWithConstructor {
  constructor(arg) {
    // Prints the "arg" argument.
    console.log(arg);
} }
```

Figure 9.1: *An example for defining a class with an explicit constructor.*

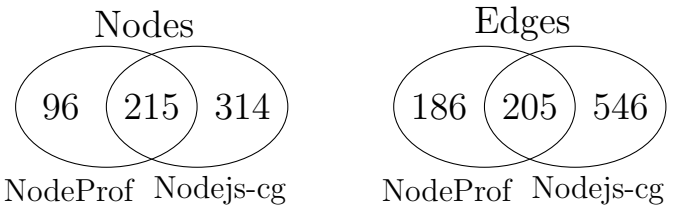


Figure 9.2: *Number of call graph nodes and edges on SunSpider.*

9.1.3 Comparison of Found Nodes and Edges

Figure 9.2 shows a Venn diagram of the nodes and edges encountered during the running of the SunSpider benchmark [100] 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 9.2 shows a large number of unique nodes and edges.

For further examination, the nodes and edges of Figure 9.2 are divided into four categories. Tables 9.2 and 9.3 demonstrate the number of nodes and edges that are part of these categories for these call graph generators. The common group represents the shared nodes and edges, and its values are the same as the values in the overlapping areas of the circles in Figure 9.2. In the following subsections, we will concentrate on the other groups, which represent the distinctions between these two call graphs.

9.1.4 JavaScript Built-ins

The second line in Tables 9.2 and 9.3 contains unique nodes and edges that are called JavaScript, or simply *JS built-ins*. The ECMAScript standard outlines a variety of built-in functions [21, in section 15], some of which are exercised by SunSpider. For instance, the *'string-tagcloud.js'* benchmark program sorts the elements of an array with the assistance of the built-in *sort()* method. Figure 9.3 provides an example of the use of this built-in method.

group name	number of nodes		number of edges	
common	215	(69.1%)	205	(52.4%)
JS built-ins	0	(0.0%)	0	(0.0%)
node.js init	91	(29.3%)	117	(29.9%)
module loading	5	(1.6%)	69	(17.7%)
total	311	(100.0%)	391	(100.0%)

Table 9.2: *Call graph node and edge groups by NodeProf.*

group name	number of nodes		number of edges	
common	215	(40.7%)	205	(27.3%)
JS built-ins	17	(3.2%)	29	(3.8%)
nodejs init	290	(54.8%)	452	(60.2%)
module loading	7	(1.3%)	65	(8.7%)
total	529	(100.0%)	751	(100.0%)

Table 9.3: *Call graph node and edge groups by Nodejs-cg.*

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.

Figure 9.4 illustrates two distinct subgraphs in which the nodes assigned to the *doSort()* and *compare()* functions specified in Figure 9.3 are linked by a route. The left subgraph is a direct edge between these two nodes since the *sort()* function is an internal function implemented in NodeProf and its calls are not monitored. Earlier versions of NodeProf captured the invocation of native functions and allocated the same *<built-in>* source file name, but this feature has been removed from the most recent versions.

The *sort()* function in Nodejs-cg is written in JavaScript and can be seen in the right subgraph of Figure 9.4. However, most of the built-in functions are na-

```
function compare(a, b) {
    if (a < b) {
        return -1;
    }
    return (a > b) | 0;
}

function doSort(arr) {
    arr.sort(compare);
}

doSort([3, 2, 1])
```

Figure 9.3: *An example for sorting an array.*

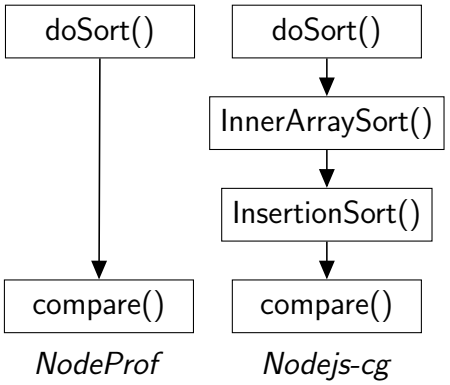


Figure 9.4: *Subgraphs from Figure 9.3 example.*

tive to Nodejs-cg, so the two call graph generators do not provide much insight into the usage of the built-in functions in a module. This could be improved in the future by adding entry/exit notifications to the native functions.

9.1.5 Module Initialization

The Node.js initialization group is shown in Tables 9.2 and 9.3. The nodes and edges in this group are part of every call graph, regardless of the program.

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.

Regarding Nodejs-cg, 64 modules are loaded and almost 300 functions are executed during the initiation procedure. These functions appear as nodes in the call graph, and their exact amount is displayed in the Node.js init group in Table 9.3. These figures are much lower for NodeProf: It only loads 19 modules and runs almost a hundred functions, as seen in Table 9.2. The cause of these low numbers is that NodeProf loads its analysis script at a later stage of Node.js initialization, and the call graph generator is unable to capture the function calls that happened before loading.

```
console.log('Hello!');
```

(a) Original source code

```
(function(exports, ...) {
  console.log('Hello!');
})
```

(b) Wrapped source code

Figure 9.5: *Example for source code wrapping.*

9.1.6 Module Loading

The last category in Tables 9.2 and 9.3 is the module loading group. This group has a small number of nodes, since most modules are used during the initialization process, and only a few extra support functions are necessary to load other modules.

SunSpider's benchmark suite [100] consists of 26 individual programs, and this is reflected in both tables, which show more than 60 edges. The test driver loads the module by first wrapping the source code into a function expression (as seen in Figure 9.5). 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.

Comparing the differences, the Nodejs-cg and NodeProf-generated call graphs not only have many common edges but also contain a large number of unique edges. For example, only 27% of the edges in the call graph created by Nodejs-cg were part of the common group. This is less than the common edge ratio of NodeProf, which was around 50%. This implies that the call graphs created for the SunSpider benchmark suite reveal more about Node.js than SunSpider. In subsection 9.1.8, we demonstrate how filtering can effectively reduce the discrepancies between call graphs.

9.1.7 Call Graphs of Real-World Programs

In the preceding subsection, we compared the nodes and edges of multiple call graphs created from the SunSpider benchmark suite. We discovered that these call graphs have a considerable number of distinct nodes and edges, for instance, 73% of the edges are exclusive in the call graph produced by Nodejs-cg. Nevertheless, SunSpider is a relatively small benchmark suite, so it would

be advantageous if further research was conducted with other programs before making any conclusions.

In this subsection, we compare the call graphs generated from 12 Node.js modules. Many modules are from the BugsJS [38] framework, while the remaining ones have been chosen by us, namely *doctrine*, *jshint*, and *request*. The BugsJS framework has an additional module, *node-redis*, which was excluded from this comparison. Further information on why this module was not included is provided in Section 9.1.10.

Each module has its own testing system that utilizes Node.js to carry out the tests. The call graph generators also work on Node.js binaries, which can run the tests simultaneously and construct JavaScript call graphs. After the testing is finished, a final call graph is created which is the union of the produced call graphs. The final call graph contains all the function calls that were executed during testing, including the internal calls from Node.js and the JavaScript engine.

9.1.8 Comparison of Nodes

Table 9.4 shows the number of nodes recorded for each Node.js module. 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. Although nodes can be filtered out after the testing is completed, this is not the case for the call-graph edges, as seen in Figure 9.6. When the filter is applied, only the application-related functions and their connections remain in the call graph, such as core module functions, functions related to testing, and functions provided by various external dependencies installed by the package manager.

Table 9.4 is divided into two halves, with three columns in each. The middle column shows the number of nodes that were found by both NodeProf and Nodejs-cg, while the columns on either side show the number of nodes that were only identified by one of the two generators.

The two halves of the table usually have very similar values in the middle columns, which implies that the filter is successful in enhancing the similarity of the call graphs by eliminating only distinct nodes. Nevertheless, the *mongoose* module is an exception: When testing the *mongoose* and *karma* modules, some test cases may be lost. This is further discussed in Section 9.1.10. In addition, the nodes that represent the functions related to these tests are also absent

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

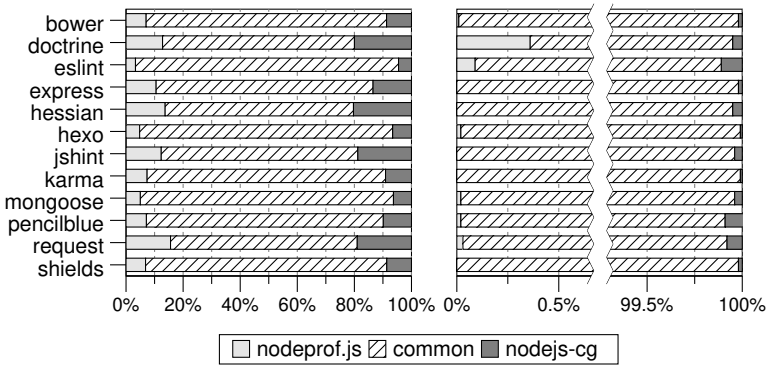


Table 9.4: *Number of call graph nodes found by NodeProf and Nodejs-cg.*

from the call graphs, decreasing the number of common nodes.

Table 9.4 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 (except for *eslint*). 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. As for *eslint*, it creates temporary directories and runs JavaScript source files placed in these directories. Since the source code of these functions is not available later, they are not currently identified as the same nodes in the two call graphs. This was discussed further in Section 9.1.2. The remaining differences will be discussed in the

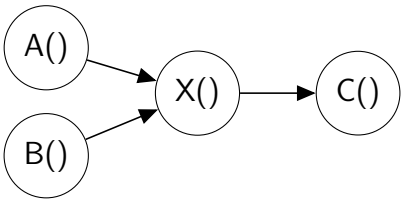


Figure 9.6: *Call graph filtering problem: if the node marked with* X() *is removed, it cannot be decided whether* C() *node is transitively called from* A() *or* B() *or both.*

following subsection, which focuses on the call graph edges.

9.1.9 Comparison of Edges

Table 9.5 shows the number of edges recorded for each Node.js module. The table is divided into two sections: the left side contains the total number of edges, and the right side contains the edges that have been filtered. The filter used is the same as the one described in Section 9.1.8.

We have investigated the effects of filtering by examining Table 9.5. It is evident that after the filter is applied, the number of common edges increases, while the number of unique edges decreases significantly. In 16 of the 24 cases, the number of unique edges is reduced to a single digit, which is less than 0.1% of the edges in the related call graph. As we observed in Section 9.1.8, the unique nodes of the filtered call graphs are also very low, thus we can conclude that the filtered call graphs created by NodeProf and Nodejs-cg are very similar. However, there are two modules, that have hundreds of unique edges even in the filtered call graphs.

The *mongoose* module has hundreds of distinct edges in its filtered call graph, which is mainly due to JavaScript generator functions. An example of this is illustrated in Figure 9.7. When the generator function *g()* is called, both call graph generators record a function call, but the body of the *g()* function is not executed. Instead, an object is created with a *next()* method. When this *next()* method is invoked, the body of the generator function is executed until a *yield* operator is encountered or the function returns. This means that when the *next()* method is invoked, the *f()* function in Figure 9.7 is called, and the Nodejs-cg tool accurately records this as a function call from *g()* to

Name	All call graph edges			Module call graph edges		
	NodeProf	common	Nodejs-cg	NodeProf	common	Nodejs-cg
bower	8302	12250	8734	4	18849	6
doctrine	1751	2788	2237	12	3572	1
eslint	8153	24093	8791	124	29436	73
express	4023	9988	4451	0	11455	1
hessian	2346	2503	2735	0	3399	1
hexo	6904	15460	7578	2	19856	1
jshint	2084	2251	2493	0	3189	1
karma	8463	10787	8892	3	15864	2
mongoose	9468	28889	10735	148	31859	875
pencilblue	5358	6856	5811	4	10000	12
request	4662	3892	5059	9	5776	9
shields	8329	8400	9035	39	13489	259

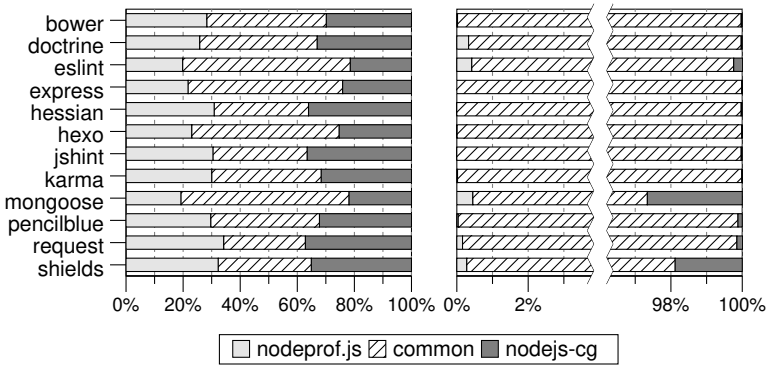


Table 9.5: *Number of call graph edges found by NodeProf and Nodejs-cg.*

f(). However, the call graph generator based on NodeProf is unaware that the execution has entered the body of the *g()* function and it registers a function call from *h()* to *f()*. This is why the call graph generated by Nodejs-cg has six times more unique edges than the one generated by NodeProf. This is because many tests of the *mongoose* module are implemented as generator functions which call the same API functions with different parameters. If these generator functions are disregarded, the test driver becomes the caller of the API functions, resulting in fewer edges being created.

The NodeProf-based call graph generator could be improved in the future to recognize function calls made by generator functions. This is not a simple

```
function f() {
    return 1;
}

function* g() {
    yield f();
}

function h() {
    g().next();
}

h();
```

Figure 9.7: *An example for JavaScript generator functions.*

```
function f() {
}

function r(res, rej) {
   res("Resolved");
}

var p = new Promise(r);

async function g() {
    await p;
    f();
}
g();
```

Figure 9.8: *An example for using Promises and await keyword.*

adjustment, however, as NodeProf only provides the source code location of the call site, not the source code location of the calling function, and the generator would need to search for the relevant function for each site.

The call graphs of the *shields* module have the second-highest number of distinct edges. However, this difference is not due to generator functions, although the cause is somewhat similar. The await expression halts the execution of an asynchronous JavaScript function until a Promise object is fulfilled, as shown in Figure 9.8. When the function *g()* is invoked, it runs until the await expression is encountered, and the function returns with a Promise object. The return value of a function declared with the *async* attribute is always a Promise object, even if the function terminates normally. When the Promise object's argument of the await expression in the *g()* function is fulfilled, the *g()* function continues its execution and calls the *f()* function. In a similar way to generator functions, the call graph generator based on NodeProf is not aware that the execution of the *g()* function is resumed, and it reports that the *f()* function is called by the Promise callback executor, leading to discrepancies between the call graphs.

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.

9.1.10 Performance Overhead

In this subsection, we can discuss the performance impact of constructing call graphs. Table 9.6 shows the effect of the call graph generation with NodeProf and Nodejs-cg. The execution is significantly slower; it is more than two and eight times slower on *eslint* Although the relative slowdown is smaller generating call graphs with NodeProf, it runs around ten times slower than Nodejs-cg on average. Filtering does speed up call graph construction, but the difference is only 10%. Generally, the mentioned slowdown has no negative effect on testing, except for three modules from the BugsJS framework: *mongoose*, *karma*, and *node-redis*. We noticed that some tests may be skipped nondeterministically during testing, and the nodes and edges related to these tests are also absent from the call graphs. This issue can even occur when an unmodified Node.js runs the tests, although rarely. However, when call graph generators are used, we observe more frequent test disappearances. Usually, only a few tests are missing, but sometimes up to 80% of the test cases are absent.

The aforementioned modules interact with external tools: *mongoose* and *node-redis* manage a database server, while *karma* controls a web browser. If an error occurs during communication, the test system captures it and halts the execution of the current batch of tests. Tests that have not been executed yet are not marked as successful or failed tests; they are simply ignored and the test system continues testing with the next batch of tests. As for *mongoose* and *karma* modules, the call graph generators can usually run most of their tests without any issues, but *node-redis* often loses its network connection, so we decided to exclude this module from the comparison. We assume that the overhead of call graph construction is the cause of this issue since *node-redis* runs several time-sensitive tests.

Name	Slowdown factor Slowdown factor without filtering		with filtering	
	NodeProf	Nodejs-cg	NodeProf	Nodejs-cg
bower	1.58	2.40	1.37	2.15
doctrine	1.99	2.59	2.14	1.78
eslint	2.76	8.80	2.53	8.70
express	1.29	1.25	1.13	1.41
hessian	1.61	4.06	1.57	3.51
hexo	1.36	2.51	1.57	2.42
jshint	1.71	1.79	1.50	1.63
karma	1.20	2.30	1.20	1.14
mongoose	1.41	2.33	1.02	2.10
pencilblue	1.35	1.58	1.03	1.58
request	1.52	1.45	1.03	1.17
shields	1.60	4.45	1.55	4.15
average	1.62	2.96	1.47	2.65

Table 9.6: *Performance overhead of generating call graphs.*

9.2 Comparison of Static and Dynamic Call Graphs

In the previous sections, a detailed analysis has been performed of dynamic call graph generators, showing their properties, performances, and applicabilities. 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).

9.2.1 Static and Dynamic Call Graph Generators

We quantitatively evaluated five distinct static analysis-based tools (TAJS [2], ACG [25], Google Closure Compiler [35], IBM WALA [27], and npm call-graph [32]) and two dynamic tools (NodeProf [92] and Nodejs-cg [54]) to determine the various calls each tool can detect and how the results of the static 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.

In order to carry out our analyses, we required inputs. Unfortunately, there is no existing, community-acknowledged standard for assessing JavaScript call graph builder algorithms. To address this issue, we identified two distinct sets of inputs: first is the straightforward, single-file inputs (in this instance, we employed the SunSpider benchmark), and second is the multi-file real projects (we selected a few popular Node.js modules).

We conducted a SunSpider analysis to investigate the variations in numbers, precision, and types of call edges reported by different tools. Our manual evaluation of 348 call edges revealed that TAJS had the highest precision, with more than 97% of the edges found to be true positives. The union of all the true edges found by the five tools showed that ACG and TAJS had the highest recall (93%). Closure, however, detected true positive edges that all other static tools had missed. TAJS had an accuracy of 97%, but it failed to detect any unique edges (edges that other static tools missed). The call graph built by TAJS was also most similar to that of dynamic tools. The combination of static tools did not produce all true edges and the combined precision was only 53%. The similarity between the static call graphs and the dynamically constructed ones varied greatly. We found that many of the missing dynamic edges were not realized in any runs due to incomplete test input. The dynamic nature of JavaScript also hindered static techniques from reliably identifying edges.

The results of the analysis of the Node.js modules showed a large variance. As none of the tools other than ACG could analyze multi-file projects, ACG was

the only static tool used in addition to the dynamic approaches. ACG's two call graph building strategies had a precision of 34.20%, while the dynamic tools had perfect precision. The recall values of the static and dynamic approaches were similar, ranging from 58.40% to 69.52%. The combination of the two approaches achieved perfect recall with a precision of 39.49%. The main contributions of this part of the research are:

- the quantitative and results quality analysis of the static and dynamic tools on 26 SunSpider benchmark programs,
- evaluation and comparison of ACG and dynamic approaches on 12 widely used Node.js modules,
- a manually validated data set of call edges found by these tools.

9.2.2 Fundamentals of Comparison

Let us add some details to the definition of call graph (introduced in Section 9):

- The nodes represent program functions (functions are identified by the name of the containing file and the exact source code position (line and column) where the function starts),
- A directed edge connects two nodes and represents a call from one function to another (i.e., function a() calls function b()),
- In our approach, there is a maximum of one edge between two nodes, thus, we track only if a call from one function to another is feasible, but we ignore its multiplicity. This is because not all tools can detect multiple calls, and we wanted to keep the definition of call graphs simple.

At first glance, it appears reasonable to compare graphs in our research using the conventional method. However, our initial thought was that the tools could overlook certain edges, which would make comparing graphs much more challenging. Therefore, in this research, we will analyze the set of edges that compose graphs. We would like to point out that our call graph outputs can be used to construct a graph at any time, on which any graph algorithm can be applied.

In our research, we needed to use call graph extraction tools. However, it was difficult to select the right set of tools, as a simple search could result in

hundreds of potential options. To make the selection process easier, we established criteria for the tools and only considered those that met the following requirements: i) the ability to generate a function call graph from a JavaScript program, ii) open-source and free, and iii) widely used in practice. The latter was a less formal criterion, where we took into account the number of weekly downloads on npm 2 and the activity on GitHub (stars, issue management, number of forks, and pull requests). As a result, we chose five static and two dynamic tools for our comparative study.

We have not found any benchmark specifically designed to compare JavaScript call graph extraction tools. Therefore, we examined what input other researchers and practitioners use for similar evaluation tasks. One of the benchmarks used by many is the SunSpider benchmark of the WebKit browser engine. This benchmark includes several single-file JavaScript examples in real-world use. The programs are intended to test the WebKit JavaScript engine, and thus contain code of varying complexity, with multiple function types and calls, all within single JavaScript files. These features make them an ideal choice for our single-file test subjects.

Today, many JavaScript modules are made up of multiple files that may contain references to each other. To make our research as realistic as possible, we randomly chose 12 Node.js modules from GitHub that met the following criteria: (1) the module was composed of multiple JavaScript source files, (2) it had a high test coverage of at least 75% statement level, and (3) it was used by at least 100 other modules. These criteria guarantee that our results (based on the chosen inputs) are in line with the results that would be obtained if the tools were used in practice.

We sought to assess the performance of the tools using proto-benchmarks that reflect their real-world application. We also keep in mind that one may want to evaluate the tools with more specific inputs (e.g. domain-specific libraries) or include another tool in the comparison. Therefore, we have designed our study and framework to be easily expandable, with other tools and input.

9.2.3 Overview of Call Graph Generator Tools

In this subsection, we outline the techniques that we used in our comparison study.

WALA WALA is a comprehensive system for analyzing Java programs, both statically and dynamically. It also has a JavaScript front-end based on Mozilla's

Rhino parser [62]. For this research, we only used one of its main components, called static analysis, designed specifically for call graph creation.

Closure Compiler Closure Compiler is a genuine JavaScript compiler that takes JavaScript applications, parses and examines them, eliminates unused code, rewrites and compresses the code, and looks for common JavaScript errors with limitations. Instead of compiling to machine code, it produces improved JavaScript.

ACG ACG (Approximate Call Graph) implements a field-based call graph construction algorithm for JavaScript. The call graph constructor has two distinct approaches, pessimistic and optimistic, which differ in the way inter-procedural flows are managed.

NPM CallGraph Module Gunar C. Gessner developed npm callgraph, a tiny npm package to generate call graphs from JavaScript code. It uses UglifyJS2 [9] to parse JavaScript code. Despite its small size and limited number of commits, it is highly popular, having been downloaded more than 5,000 times.

TAJS A dataflow analysis tool for JavaScript, developed at Aarhus University, called the 'Type Analyzer for JavaScript', allows one to infer type information and produce call graphs.

NodeProf NodeProf, introduced in the previous section, is a framework for instrumentation and profiling of Node.js modules. It is able to execute the modules and alert applications, known as analyses, when certain events occur in the JavaScript code, such as the entry and exit of a function or the assignment of a variable. Our dynamic call graph generator tool is one of these analyses, collecting data related to the call graph.

Nodejs-cg We employed Nodejs-cg, a customized Node.js runtime, in this project. Node.js utilizes the V8 engine as its default JavaScript interpreter. It has integrated tracing capabilities, but the default tracing system has significant overhead as it requires a lot of time and space to parse the output and construct a call graph.

9.2.4 Testbed

We created two distinct sets of test inputs for the purpose of conducting a thorough and comprehensive evaluation. These test input groups were carefully chosen to include a wide variety of scenarios, complexities, and usage patterns.

Single file benchmark examples We previously discussed our intention to include real-world, single-file JavaScript examples that can be easily examined by a program or manually. To do this, we used the SunSpider benchmark.

Real-word, Multi-file Node.js Examples We tested several Node.js modules to see how they manage the current ECMAScript 6, and Node.js features (such as module exports or external dependencies, i.e., the *require* keyword) and inter-file connections. Modules that employ the most up-to-date standards and have a large group of developers and users have been chosen for this testbed. Table 9.7 summarizes the details of the selected Node.js modules.

Name	Repository URL	SLOC
debug	`https://github.com/visionmedia/debug.git`	1,083
doctrine	`https://github.com/eslint/doctrine`	5,109
hessian.js	`https://github.com/BugsJS/hessian.js.git`	6,796
request	`https://github.com/request/request`	9,469
express	`https://github.com/BugsJS/express.git`	11,673
hexo	`https://github.com/BugsJS/hexo.git`	16,617
karma	`https://github.com/BugsJS/karma.git`	17,690
bower	`https://github.com/BugsJS/bower.git`	28,087
shields	`https://github.com/BugsJS/shields.git`	47,786
pencilblue	`https://github.com/BugsJS/pencilblue.git`	54,746
jshint	`https://github.com/jshint/jshint`	68,411
eslint	`https://github.com/BugsJS/eslint.git`	284,342

Table 9.7: *The selected Node.js modules and their size (source lines of code)*

9.2.5 Graph Comparison

We quantitatively evaluated the call graphs by comparing the number of nodes and edges, as well as the similarity of entire call graphs. To assess the quality of the results, we implemented a Python-based call graph comparison script based on the work of Lhoták et al [49]. This script was designed to detect

matching edges identified by various tools. Each node and edge was extended with an attribute containing a list of tool identifiers that found the particular node or edge. Because many JavaScript functions do not have names, nodes, and edges were identified using path, line, and column information instead of a unique unified naming scheme (as discussed before in Section 9.1.2).

We manually verified the path and line data generated by the evaluated tools to ensure that the comparison was accurate. TAJS provided precise line and column information in its standard DOT output. We implemented and modified the line information extraction in Closure Compiler, WALA, and ACG tools. Unfortunately, WALA was only able to report line numbers, not column information, so we had to manually refine its output. Since the reported line and column data from npm callgraph were not precise (neither of them), we manually added this information to the created JSON files. As previously noted in 9.1.2, we also implemented a precise line information dump in our dynamic tools.

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.

9.2.6 Precision, Recall, F-measure

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.

Identifying true positive edges was a straightforward task, however, distinguishing between false positive and pseudo-positive edges was more difficult. False positive edges do not exist in the source code, but either the edge's caller or callee has a function signature that is similar to a function signature that is actually present in the program. On the other hand, in the vast majority of pseudo-positive cases, the caller function is never invoked, and thus the call from the caller to callee (which would otherwise be a valid, possible call) is never executed.

9.2.7 SunSpider Analysis

The quantitative analysis shows the number of nodes and edges found by the call graphs generator tools. In Table 9.8, we can see a static tool that produces similar results to the dynamic tool in almost every case. For example, all of the tools agreed on the number of nodes and edges for math-spectral-norm.js and string-fasta.js. For *itops-bitwise-and.js* and *regexp-dna.js*, we can see that none of the static or dynamic tools can find a node. Since *bitops-bitwise-and.js* contains only some statements without calling any function, none of the tools realize a node (or an edge). In the case of *regexp-dna.js*, we can also see some statements, however, some calls to built-in functions happen.

In Table 9.8 one can spot the numerical differences of properties' dynamic call graph reported in the earlier sections. In this follow-up research, we do not take into account the built-in function calls, which allows us to focus on JavaScript analyses rather than the deficiencies of generator tools. Thus, the following aspects do not influence the final result:

- Only the Nodejs-cg can show the built-in calls.
- The benchmark contains several *eval()* calls that do not have path information (see Section 9.1.2). So, such calls should be connected to an artificial node which can be very different in generator tools.

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

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
3d-cube	15	23	15	23	16	24	15	23	15	23	15	23	15	23
3d-morph	2	1	2	1	0	0	2	1	2	1	2	1	2	1
3d-raytrace	22	29	28	41	21	22	27	40	28	39	28	39	28	39
access-binary-trees	3	3	4	5	4	5	4	5	4	5	4	5	4	5
access-fannkuch	2	1	2	1	3	2	2	1	2	1	2	1	2	1
access-nbody	8	11	12	15	8	11	11	14	12	15	12	15	12	15
access-nsieve	3	2	3	2	2	1	3	2	3	2	3	2	3	2
bitops-3bit-bits-in-byte	2	1	2	1	3	2	2	1	3	2	3	2	3	2
bitops-bits-in-byte	2	1	2	1	3	2	2	1	3	2	3	2	3	2
bitops-bitwise-and	0	0	0	0	0	0	0	0	0	0	0	0	0	0
bitops-nsieve-bits	3	2	3	2	3	2	3	2	3	2	3	2	3	2
controlflow-recursive	4	6	4	6	4	6	4	6	4	6	4	6	4	6
crypto-aes	17	16	17	16	13	16	17	16	13	14	13	14	13	14
crypto-md5	21	30	21	30	3	2	21	30	12	15	12	15	12	15
crypto-sha1	18	23	18	23	3	2	18	23	9	8	9	8	9	8
date-format-tofte	18	18	19	20	2	1	3	2	3	2	12	11	12	11
date-format-xparb	0	0	14	14	13	17	14	14	5	5	6	5	6	5
math-cordic	5	5	5	5	5	5	5	5	5	5	5	5	5	5
math-partial-sums	2	1	2	1	2	1	2	1	2	1	2	1	2	1
math-spectral-norm	6	6	6	6	6	6	6	6	6	6	6	6	6	6
regexp-dna	0	0	0	0	0	0	0	0	0	0	0	0	0	0
string-base64	3	2	3	2	3	2	3	2	3	2	3	2	3	2
string-fasta	5	4	5	4	5	4	5	4	5	4	5	4	5	4
string-tagcloud	4	4	12	18	2	1	11	17	3	2	6	6	6	6
string-unpack-code	0	0	13	20	5	8	12	64	13	20	13	16	13	16
string-validate-input	4	3	5	4	5	4	5	4	5	4	5	4	5	4
$\sum$	**169**	**192**	**217**	**261**	**134**	**146**	**197**	**284**	**163**	**186**	**176**	**195**	**176**	**195**

Table 9.8: *SunSpider analysis results*

retrieval metrics (precision and recall). We should note that only simple call edges were evaluated and compared; paths along these edges (i.e., call chains) were not taken into account. The effect of missing or extra edges may vary depending on the number of paths that go through them, and this could influence the precision and recall of the identified call chain paths.

The information in Table 9.9 can be seen in detail. The first column lists the name of the tool or a combination of tools. The second and third columns display the total number of true (TP) and false positive (FP) instances identified by the relevant tool or combination of tools. The column labeled PP displays the amount of pseudo-positive edges, which would be actual if the program's execution had reached the caller's side. Since this does not occur, these edges are not considered true positive edges. The fifth column (labeled All) shows the total number of edges that were identified by the relevant tool

or combination of tools. The sixth (labeled *Prec.*), seventh (labeled *Rec.*), and eighth (labeled *F*) columns contain the precision (*TP / All*), recall (*TP / 195*), and F-measure values, respectively.

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%
ACG+Closure	182	54	73	309	59%	93%	72%
ACG+npm-cg	182	24	73	279	65%	93%	77%
ACG+TAJS	184	6	73	263	70%	94%	80%
ACG+WALA	184	25	73	282	65%	94%	77%
Closure+npm-cg	175	72	72	319	55%	90%	68%
Closure+TAJS	184	54	55	293	63%	94%	75%
Closure+WALA	183	73	55	311	59%	94%	72%
npm-cg+TAJS	183	22	50	255	72%	94%	81%
npm-cg+WALA	149	37	54	240	62%	76%	69%
TAJS+WALA	181	23	6	210	86%	93%	89%
ACG+Closure+npm-cg	182	72	73	327	56%	93%	70%
ACG+Closure+TAJS	184	54	73	311	59%	94%	73%
ACG+Closure+WALA	184	73	73	330	56%	94%	70%
ACG+npm-cg+TAJS	184	24	73	281	65%	94%	77%
ACG+npm-cg+WALA	184	43	73	300	61%	94%	74%
ACG+TAJS+WALA	184	25	73	282	65%	94%	77%
Closure+npm-cg+TAJS	184	72	72	328	56%	94%	70%
Closure+npm-cg+WALA	183	91	72	346	53%	94%	68%
Closure+TAJS+WALA	184	73	55	312	59%	94%	73%
npm-cg+TAJS+WALA	183	41	55	279	66%	94%	77%
ACG+Closure+npm-cg+TAJS	184	72	73	329	56%	94%	70%
ACG+Closure+npm-cg+WALA	184	91	73	348	53%	94%	68%
ACG+Closure+TAJS+WALA	184	73	73	330	56%	94%	70%
ACG+npm-cg+TAJS+WALA	184	43	73	300	61%	94%	74%
Closure+npm-cg+TAJS+WALA	184	91	72	347	53%	94%	68%
ALL	184	91	73	348	53%	94%	68%

Table 9.9: *Precision and recall measures for individual tools and their combinations*

TAJS stands out among the individual tools due to its almost perfect precision (97%) and high recall values (93%). In comparison, ACG and Closure have recall values that are close to TAJS, but their precisions are much lower. Closure had the lowest precision (62%), while WALA had the lowest recall (63%). ACG identified the most true positive edges, but it also found a number of pseudo-positive edges, which explains its lower precision and recall.

Table 9.10 presents the correlation between the call edges identified by the static call graph tools and the dynamic call graph extraction process. We only kept those static edges that were determined to be true positives. As can be

seen, there is considerable variation in the intersections and discrepancies in the call edges among the static tools. The two extremes are npm-cg and TAJS. On the one hand, npm-cg misses 70 valid edges and has the lowest intersection (125 edges) with the dynamic approach. On the other hand, TAJS produces a result that is very similar to that of the dynamic approach. 99% of the edges found by TAJS are also in the dynamic call set, and TAJS also finds 93% of all dynamic edges (i.e., it misses only 14 edges found by NodeProf).

Tool	**Static only**	**Static ∩ Dynamic**	**Dynamic only**	**Precision**$_{dyn}$	**Recall**$_{dyn}$
ACG	71	162	33	0.70	0.83
Closure	55	175	20	0.76	0.90
npm-cg	49	125	70	0.72	0.64
TAJS	1	181	14	0.99	0.93
WALA	5	122	73	0.96	0.63

Table 9.10: *Comparison of static and dynamic edges*

WALA generated only five edges that were not in the dynamic set, resulting in a precision of 96%. Closure, on the other hand, identified 90% of the dynamic edges, but also included 55 edges that were not in the dynamic set. We examined all the edges that were only found by either the static tool or the dynamic approaches.

Edges found by the dynamic approaches Given the highly dynamic nature of JavaScript, it is not unexpected that certain edges were discovered only by dynamic tools. These edges were valid calls between functions, but they are mostly undetectable by static analysis. For example, in Listing 9.9, an anonymous function dynamically adds several functions to its parameter, referred to as *s*. This function is then immediately called with the *String.prototype* parameter, meaning that every String object will be extended with the defined functions and properties. Therefore, *tagInfoJSON* (which is a string) will have a *parseJSON* function that takes a function as an argument. The function call was realized by an inner function called inside *parseJSON*, walk, which calls the *parseJSON*'s parameter named *filter*. Since *parseJSON* was added dynamically, it would have been difficult to detect by static analysis alone.

Dynamic evaluation of strings is a common challenge that is difficult to identify with a static analyzer but can be easily identified with a dynamic tool. As an example, Listing 9.10 shows a program that adds a *formatDate* function to all *Date* instances. This function splits the desired output format (a parameter called input) and iterates through it. If it finds a format character presented in

```
(function (s) {
  // ...
  s.parseJSON = function (filter) {
    // ...
    function walk(k, v) { // line 180
      // ...
      return filter(k, v);
    }
    // ...
    return typeof filter === 'function' ? walk('', j) : j;
    }
    // ...
  };
// ...
})(String.prototype);
// ...
var tagInfoJSON = 'A long string on line 226 in string-tagcloud.js';
// ...
var tagInfo = tagInfoJSON.parseJSON(function(a, b)
  { /*code*/ }); // line 229
```

Figure 9.9: *A call detected only by dynamic tools (String.prototype)*

the predefined variable switches, the function calls the corresponding function with *eval*. This is a straightforward dynamic call, but it is very hard to detect with a static analysis tool.

It is worth mentioning one module, the string-unpack-code.js, which confuses the static analysis. The Listing 9.11 shows the root of the issue. The dynamic tools in string-unpack-code.js identified more nodes than the static ones. We evaluated these nodes and established that they are valid and existing functions. The static analyzers overlooked these nodes since they do not invoke any functions (as they are callbacks that return an element of an array).

Edges found only by static tools As anticipated, certain edges were only identified by the static tools. Generally, static call graphs contain potential call edges that are never executed during runtime. It is important to be aware that the SunSpider benchmark contains a considerable amount of unused code, leading to a large number of potential calls that are not being made. This reveals one of the drawbacks of the dynamic approach, which is that inadequate test input can lead to an imprecise call graph. Nevertheless, there are a number

```
function arrayExists(array, x) {
  for (var i = 0; i < array.length; i++) {
    if (array[i] == x) return true;
  }
  return false;
}
Date.prototype.formatDate = function (input,time) {
  var switches = ["...", "g", "G", "..."];
  // ...
  function g() {/* 12 hour format of the given date */}
  // ...
  var ia = input.split("");
  var ij = 0;
  while (ia[ij]) {  // this will be "g"
    //...
    if (arrayExists(switches,ia[ij])) {  // "g" in switches
      ia[ij] = eval(ia[ij] + "()");  // ia[ij] = g()
    }
    ij++;
  }
  // ...
}
var date = new Date("1/1/2007 1:11:11");
var longFormat = date.formatDate("g");
```

Figure 9.10: *A call detected only by dynamic tools (eval-based)*

of potential edges that are not realized due to certain conditions on the inputs, and we ran the dynamic analysis with only one input vector supplied with the tests.

In Listing 9.12, the function call to *String.escape* (line 19) is never put into action, since the dateFormat was never invoked with an argument that has a backslash.

9.2.8 Node.js Modules Analysis

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

```
var decompressedMochiKit = function(p,a,c,k,e,d)
  {e=function(c){return(c<a?"":e(parseInt(c/a)))+
  ((c=c%a)>35?String.fromCharCode(c+29):c.toString(36))}
  ...
}(...);
var decompressedDojo = function(p,a,c,k,e,d)
  {e=function(c){return(c<a?"":e(parseInt(c/a)))+
  ((c=c%a)>35?String.fromCharCode(c+29):c.toString(36))}
  ...
}(...);
```

Figure 9.11: *A confusing code snippet from 'string-unpack-code.js'*

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.

To gather as much valuable information as we could, we performed the analysis using both applicable strategies that ACG offers (`ONESHOT` and `DEMAND`). The `ONESHOT` strategy tracks the inter-procedural flow but it tracks only for one-shot closures that are invoked immediately. The `DEMAND` strategy (called the optimistic approach) performs inter-procedural propagation along the edges that may end at a call site.

We calculated some basic statistics from the gathered data that is shown in Table 9.11. 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. However, in one case, there is a slight difference between the found edges (*eslint*).

During the evaluation of the edges, we could only validate the existence of the edges, due to the huge size of the input programs. So it is possible that we labeled pseudo-positive edges as true positive edges, since checking whether the execution of such huge programs reaches a particular point (in any way) is cumbersome. Taking every tool into consideration, only 6,818 edges were found by all of the tools, which is approximately 8% of all edges. The number

```
Date.formatFunctions = {count:0};
Date.prototype.dateFormat = function(format) {
  if (Date.formatFunctions[format] == null) {
    Date.createNewFormat(format);
  } // ...
}
Date.createNewFormat = function(format) {
  // ...
  for (var i = 0; i < format.length; ++i) {
    ch = format.charAt(i);
    if (!special && ch == "\\") {
      special = true;
    }
    else if (special) {
      special = false;
      code += "'" + String.escape(ch) + "' + ";
      // ^ This call is never realized but is a valid possible call
    }
}  }  // ...
String.escape = function(string) {
  return string.replace(/('|\\)/g, "\\%$1"); //'
}
var date = new Date("1/1/2007 1:11:11");
for (i = 0; i < 4000; ++i) {
  var shortFormat = date.dateFormat("Y-m-d");
  var longFormat = date.dateFormat("l, F d, Y g:i:s A");
  date.setTime(date.getTime() + 84266956);
}
```

Figure 9.12: *An unrealized edge*

of common edges between the tools can be seen in Figure 9.13 Venn diagram.

Since it is not feasible to double-check every edge by hand, we used a simple statistics approach to achieve a 95% confidence level with a 5% margin of error on found edges. Based on the evaluated samples, we can even give an estimation of the precision and recall of each approach. Table 9.12 shows these estimated numbers.

Node module	Static tools				Dynamic tools			
	ACG `ONESHOT`		ACG `DEMAND`		NodeProf		Nodejs-cg	
	nodes	edges	nodes	edges	nodes	edges	nodes	edges
bower	674	2146	710	2464	790	1177	790	1177
debug	32	29	35	33	22	23	22	23
doctrine	87	179	87	179	92	195	92	195
eslint	2529	13139	2545	13436	3646	6658	3646	6660
express	122	262	133	506	176	345	176	345
hessian	81	201	81	201	117	224	117	224
hexo	440	1173	482	2922	927	1351	927	1351
jshint	351	1019	378	1128	262	360	262	360
karma	443	781	449	817	510	751	510	751
pencilblue	2192	8862	2443	48512	863	1126	863	1126
request	114	217	114	218	169	233	169	233
shields	1410	8062	1524	8832	1609	2236	1609	2237
$\sum$	**8475**	**36070**	**8981**	**79248**	**9183**	**14679**	**9183**	**14682**

Table 9.11: *Node.js analysis results*

9.3 Results

Each approach and tool has advantages and disadvantages. This comparative study yielded the following insights.

- Static tools are adept at dealing with recursive calls; the Closure Compiler appears to be the most advanced in this area.
- Edges that lead to functions that are nested within other functions are not always managed properly by static analysis tools, such as WALA, which can result in a large number of false edges.
- In addition to the dynamic tools, only WALA, TAJS, and ACG with the optimistic approach (`DEMAND`) can identify calls of function arguments (e.g. higher-order functions).
- ACG and TAJS cal are able to trace more intricate control flows and recognize non-trivial call connections.
- Closure is frequently based solely on name comparison, which can lead to incorrect or absent connections.
- WALA is capable of analyzing *eval* structures and making dynamic calls from strings to a certain degree.

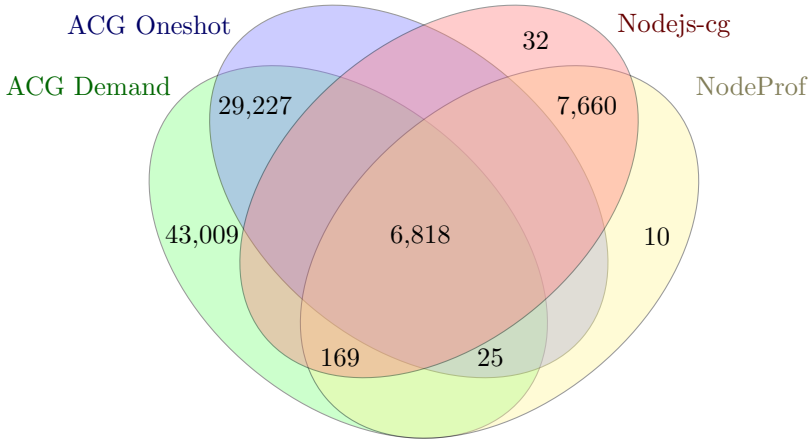


Figure 9.13: *Common edges in Node.js modules*

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%
ACG `ONESHOT`+Nodejs-cg	45.94%	95.52%	62.04%
ACG `ONESHOT`+NodeProf	45.94 %	95.50%	62.04%
ACG `DEMAND`+Nodejs-cg	29.91%	99.85%	46.04%
ACG `DEMAND`+NodeProf	29.91%	99.83%	46.03%
ACG `ONESHOT`+Nodejs-cg+NodeProf	59.49%	95.67%	73.36%
ACG `DEMAND`+Nodejs-cg+NodeProf	39.39%	100.00%	56.52%
ACG `ONESHOT`+`DEMAND`	16.93%	63.53%	26.74%
Nodejs-cg+NodeProf	100.00%	69.67%	82.12%
ALL	38.10%	100.00%	55.17%

Table 9.12: *Precision and recall values*

- Npm-cg mishandles requests from anonymous functions declared in the global context, indicating that the call is coming directly from the global context (as opposed to the anonymous function).
- TAJS generated a call graph that was the most similar to the one produced by dynamic analysis.
- Of the existing static techniques, only ACG is practically suitable for assessing the most recent Node.js modules (owing to its language support and accuracy).
- Both static and dynamic techniques identified edges that the other had overlooked.

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 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.

10
Related Work

Although the topic of writing efficient JavaScript applications and code snippets is very important for the software industry, the main area to evolve the JavaScript software stack is the improvement of the engines themselves. Based on the well-studied research area in static languages [69, 95] the static optimization algorithms could be the first good choice to use in JavaScript engines as well. However, JavaScript is a dynamic language where the static optimization algorithms cannot determine various properties, for example, the types of objects, variables, or even the structure of the input script. For this challenge, user intervention is needed, for example, applying guidelines to improve performance.

There are only a limited number of studies that discuss how to improve any specific characteristic of a JavaScript application with source transformations. There were studies on how to transform JavaScript projects to look like object-oriented source code [84, 85], but the focus of these researches was to improve maintainability and not to improve user experiences (such as performance or memory consumption). Another approach could be to analyze the best practices for JavaScript [72].

JavaScript's dynamic language structure and runtime environment behavior are clearly evident when performance is the main concern; however, the real challenge arises when program analysis is the focus. Call graphs are a useful

```
// shorthand for: let f = function() { ... }
function f() { return true; }

f = function() { return false; }
```

Figure 10.1: *An example for redirecting a JavaScript function.*

tool for program analysis and can be generated statically or dynamically. The first publications mentioning call graphs were published in the 1970s [24, 36]. Based on their construction method, we can divide call graphs into two basic subgroups, they can be either dynamic [103] or static [66]. There are several programs that can generate static call graphs from JavaScript code, regardless of the target platform, such as a web browser, Node.js, or something else [26, 27, 45, 56]. However, the precision of these tools is limited due to the highly dynamic nature of JavaScript. Functions are objects and can be stored in any JavaScript value. Even when a function is declared with a name, it is just a shorthand for assigning that function object to a local variable, which can be changed later, as seen in Figure 10.1. This makes it difficult for static analyzers to track which variable refers to which function object.

Some static analyzers try to improve their prediction by supporting well-known APIs. For example, the Node.js event emitter API allows for emitting named events, and these events can be captured by listener functions. The listener functions activated by a named event can be predicted statically [57] as long as certain conditions apply; e.g., the names of the events are string literals.

In addition to NodeProf, there are other frameworks [59, 83] that can be extended with dynamic call graph generators. These frameworks offer an API to capture events and execute custom JavaScript code in response. While NodeProf provides an example analysis for generating call graphs, it links call sites to called functions instead of connecting two function nodes. Technically, this analysis is not a call graph generator.

There is a tool that has been developed to generate dynamic call graphs for web applications [94]. It runs tests on web applications and collects method-level execution traces, which are then used to construct a call graph. In contrast to our work, this tool is designed for browser-based web applications rather than Node.js.

11 Conclusions

In this part of the thesis, topics related to Just-In-Time Compilers' Optimizations and Analyses were presented.

In the first topic, we evaluated guidelines available for JavaScript engines and presented new ones targeting the ECMAScript 6 feature set. The presented results show that the guidelines are still important and that a significant performance improvement can be achieved by adapting them to a JavaScript project. Although the results are now very conclusive, it is very advisable to revisit and assess the importance of the guidelines from time to time. As the JavaScript engines evolve, it might happen that some of the guidelines become obsolete.

In the second topic, we have compared several call graph generator tools for JavaScript. One of them, Nodejs-cg, is developed and maintained by us. The other tools are Jalangi2 and NodeProf which produce dynamic call graphs, and WALA, Closure, ACG, npm callgraph, and TAJS which create static ones.

First, the effectiveness of dynamic call graph generators has been evaluated and compared to each other. Validation of their efficiency has been done in various test programs. One of them is the SunSpider benchmark, which contains simple JavaScript files that test various parts of the JavaScript engine. The other type of test is based on various Node.js modules that represent real-world applications.

As a result, we have found that a large number of edges are unique in these call graphs and have shown that this is true for nodes as well. The unique nodes and edges have been validated by hand and organized into groups. These groups were compared and the reason for the differences between call graphs was explained

The comparison also provides an investigation of the runtime performance. We have found that the generation of call graphs can slow down execution to up to eight times. Due to this slowdown, unexpected test failures are more frequent for those modules that control external tools, e.g., database servers or web browsers.

The purpose of the second part of the JavaScript analysis was not to determine a victor, but rather to gain empirical knowledge into the abilities and effectiveness of modern static call graph extractors, as well as how they compare to dynamic techniques.

Each tool and analysis approach had its own set of pros and cons. The Closure Compiler was able to detect calls, mainly recursive ones, that had been overlooked by other static tools. Unfortunately, it also identified several false positive edges due to shallow name matching. ACG followed more complicated control flows to detect call edges. This led to a greater recall rate while still keeping precision at an acceptable level; however, it failed to detect higher-order function calls (callbacks). It should be noted that ACG was the only tool that had the ability to evaluate actual Node.js modules. WALA was able to recognize calls to higher-order functions; however, it generated a large number of false positive edges with unidentified nodes and had the lowest rate of accuracy among the tools. The npm callgraph module had a very poor F-measure, with both precision and recall being low, and it did not discover any true positive edges that the other methods had missed. TAJS, on the other hand, had the highest precision and recall values and generated a call graph that was most similar to the one produced by the dynamic approach.

The accuracy of the dynamic call graphs was high, however, they did not capture certain static edges due to the lack of adequate test inputs. The findings also demonstrate that the collective strength of multiple tools is greater than that of individual call graph extractors. Consequently, we believe that cleverly combining static and dynamic techniques could lead to considerable enhancements in the accuracy of the generated call graphs.

Part III

Appendices

A

Summary

The demand for speed, efficiency, and resource utilization in the world of modern computing is still growing. As a result, the importance of compiler optimizations is becoming more and more apparent in the software development process. Compilers are essential tools that transform human-readable source code into machine-executable binary code, thus connecting the programmer's goal and effective execution. They have a set of strategies that refine and enhance generated code and are essential to achieving optimal program performance, resource efficiency, and code quality.

The complexity of software has been steadily rising, as evidenced by the exponential growth in the size of software codebases and the demands for new software applications. This has caused a change in the traditional perception of compilers. The changes need to go beyond the typical code generation, necessitating the compiler toolchains to recognize additional possibilities in the codebases, such as reducing code size bottlenecks, enhancing algorithms, reducing memory usage, and making the most of hardware resources.

In addition, the basis of optimizations, i.e. compiler technologies, has also gained importance. The utilization of these technologies is widespread in many areas of software engineering, from detecting bugs in the software code to improving software quality and even aiding in the visualization of software.

III. A. Summary

I. Executable Code Optimizations

In this thesis point, the main goal was the introduction of efficient code size-optimizing algorithms for one of the most well-known compilers, the GNU Compiler Collection (GCC). In addition, a reliable code size measurement method and a stable benchmark environment were presented.

In Chapters 3 and 4, the contributions to the first thesis group were discussed. This thesis group can be separated into the following two main results.

1. Code Factoring Techniques in the GCC Compiler

Several code size optimization techniques have been introduced which have impressive effects. These algorithms were local code factoring and procedural abstraction. Both have been implemented in the RTL and Tree-SSA intermediate languages of GCC, and procedural abstraction has also been provided for the interprocedural abstraction phase. We evaluated the algorithms with the help of *CSiBE*, 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'`. In addition, a very simple optimization technique, hashtables, was used to improve the running time of the algorithms and thus the total compilation time.

2. Binary Code Size Measurement Methods and Benchmark

The measurement method of code size has been presented which led to the birth of CSiBE, GCC's official code size benchmark. Many aspects of code size measurements have been evaluated to make the benchmark the de facto code size measurement standard for compilers. During the many years of development of *CSiBE*, it has taken its well-awarded place alongside other benchmarks, such as SPEC and the later Openbench.

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 the GCC Compiler" chapter:

- 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. In addition, the IPA version 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" chapter. Besides these, the author has been the main maintainer of the official code size benchmark of GCC, *CSiBE*, since 2004.

The publications related to this thesis point are the following:

[11] 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

[10] 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

[55] 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

[67] 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

[68] 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 point, the main goal was to collect, 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 was presented that can aid other optimizations or analyses that rely on call graph information.

In Chapters 8 and 9, the contributions to the second thesis group were discussed. This thesis group can be separated into the following two main results.

3. JavaScript Guidelines

Available guidelines for JavaScript engines have been evaluated and new ones targeting the ECMAScript 6 feature set have been presented. The presented results showed that the guidelines are still important and that a significant performance improvement could be achieved by adapting them to a JavaScript project. The highest results could be achieved with the *'Avoiding With'* guideline, while the *'Common Subexpression Elimination'* guideline has a small effect on the runtime performance in the case of ECMAScript 5.1 guidelines. On the other hand, the ECMAScript 5.1 simulated version of *'Symbol'* construct achieved the best result over ECMAScript 6, while the *'class'* construct performs way better with the new standard. Although the results are now very conclusive, it is very advisable to revisit and assess the importance of the guidelines from time to time. As the JavaScript engines evolve, some of the guidelines might become obsolete.

4. Dynamic Analysis of JavaScript's Call Graphs

Several call graph generator tools for JavaScript have been compared. One of them, Nodejs-cg, is developed and maintained by the author and his co-authors. The other tools are Jalangi2 and Nodeprof.js, which produce dynamic call graphs, and WALA, Closure, ACG, npm callgraph, and TAJS, which create static ones.

First, the effectiveness of dynamic call graph generators has been evaluated and compared to each other. Validation of their efficiency has been done on various test programs. One of them is the SunSpider benchmark, and the other type of test is based on various Node.js modules that represent real-world applications. As a result, it was found that a large number of edges and

nodes are unique in these call graphs. The unique nodes and edges have been validated by hand and organized into groups. These groups were compared and the reason for their differences between call graphs was explained. The comparison also provided an investigation of the runtime performance. We have found that the generation of call graphs can slow down execution up to eight times. Due to this slowdown, unexpected test failures are more frequent for those modules that control external tools.

After the comparison of dynamic call graph generator tools, additional ones, static call graph generators were added to the next comparison. It is concluded that each tool and analysis approach had its own set of pros and cons. One of them was able to detect recursive calls. The other was able to reach a greater recall rate while still keeping precision at an acceptable level, but only one was able to handle complex modules (ACG). Overall, the accuracy of the dynamic call graphs was high compared to the static ones, however, they did not capture certain static edges due to the lack of adequate test inputs. The findings also demonstrate that the collective strength of multiple tools is greater than that of individual call graph extractors.

The Author's Contributions

The author's contribution was decisive in the collection, formalization, implementation, testing, and evaluation of a significant part of the *JavaScript* guidelines presented in the "JavaScript Guidelines" chapter. The published guidelines and measurement methods are 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" chapter, 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:

[42] 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, Finnland, 2009

[43] Zoltán Herczeg, Gábor Lóki, Tamás Szirbucz, and Ákos Kiss. Validating

JavaScript Guidelines Across Multiple Web Browsers. *Nordic Journal of Computing*, 15:18–31, 2013

[53] 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

[40] 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

[41] 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

[4] 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

Publications

Most of the research results presented in this thesis were published in proceedings of international conferences and workshops, or journals. Table A.1 presents which publications cover which results of the thesis.

Chapter	Title	Publications
3.	Code Factoring Techniques in the GCC Compiler	[55, 67, 68]
4.	Binary Code Size Measurement Methods and Benchmark	[10, 11]
8.	JavaScript Guidelines	[42, 43, 53]
9.	Dynamic Analysis of JavaScript's Call Graphs	[4, 40, 41]

Table A.1: *Summary of thesis topics and corresponding publications*

The author adds that although the results presented in this thesis are his major contribution, the term 'we' is used instead of 'I' for self-reference to acknowledge the contributions of the co-authors of the papers that this thesis is based on.

B
Összefoglalás

A mai, modern számítástechnikában megállás nélkül növekszik a sebesség, a hatékonyság és az erőforrás-felhasználás iránti igény. Ennek eredményeként a szoftverfejlesztési folyamatban a fordítóoptimalizálás jelentősége egyre fontosabbá válik. A fordítók olyan alapvető eszközök, amelyek az ember által olvasható forráskódot gépi futtatható bináris kóddá alakítják, így összekapcsolják a programozó célját és a hatékony gépi végrehajtást. Ezek az eszközök olyan stratégiákkal rendelkeznek, amelyek finomítják és javítják a generált kódot, és elengedhetetlenek az optimális programteljesítmény, az erőforrás-takarékosság és a kódminőség eléréséhez.

A szoftverek összetettsége folyamatosan növekszik, amit a kódbázisok méretének exponenciális növekedése és az új szoftveralkalmazások iránti igény is jól tükröz. Ez változást indukált a fordítók hagyományos felfogásában; túl kell lépni a tipikus kódgenerálási feladatokon. Ezért a fordítóprogram rendszereknek további kihívásokra kell felkészülniük, mint például a kódméret csökkentése, az algoritmusok javítása, a memóriahasználat csökkentése és a hardvererőforrások maximális kihasználása.

Ezek mellett nem csak a fordítóprogramok optimalizációs képessége, hanem a felhasznált a fordítóprogram technológiák is egyre inkább előtérbe kerültek. Ezeknek a technológiáknak a használata a szoftverfejlesztés számos területén elterjedt, a szoftverkód hibáinak észlelésétől, a szoftver minőségének javításán

át, egészen a szoftver vizualizálásáig, megjelenítéséig használják őket.

I. Futtatható állományok optimalizálása

Ebben a tézispontban a fő cél az egyik legismertebb fordítóprogram, a GNU Compiler Collection (GCC) számára kódméret-optimalizáló algoritmusok bevezetése volt. Emellett bemutatásra került egy megbízható kódméret mérési módszer és egy stabil benchmark környezet.

Az első téziscsoport tárgyalása a 3. és 4. fejezetekben történt meg. Ez a téziscsoport a következő két fő eredményre bontható.

1. Code Factoring technikák a GCC fordítóprogramban

A korábbi évek során két kódméret-optimalizálási technika lett bemutatva, melyek jelentős eredményeket tudtak elérni a GCC fordítóprogramba építve. Ezek az algoritmusok a 'local code factoring' és az 'procedural abstraction' voltak. Mindkettőnek készült megvalósítása a GCC *RTL* és *Tree-SSA* köztes nyelveire, a 'procedural abstraction' technikának pedig egy inter-procedurális implemetációja is. Az algoritmusok a *CSiBE*, GCC Code Size Benchmark Environment, segítségével lettek kiértékelve három célarchitektúrán: *i686-elf*, *arm-elf*, *sh-elf*. Ez eredmények azt mutatták, hogy a GCC `'-Os'` kapcsolójához képest maximum 61,53% és átlagosan 2,58% extra kódméret megtakarítás érhető el. Ezen kívül egy nagyon egyszerű, de addig még ezen a területen bevezetetlen technika, a hash-tábla lett felhasználva az algoritmusok futási idejének és így a teljes fordítási időnek a javítására.

2. Bináris kódméret méréstechnikájának módszertana és kiértékelő környezete

A kódméret mérési módszereinek bemutatásával és a figyelembe vehető szempontok megvitatásával létrejött egy új benchmark, a *CSiBE*, mely nem sokkal ezek után a GCC hivatalos kódméret eszköze lett. A kezdeti apró lépések után a *CSiBE* többéves fejlesztése elérte a célját, és a benchmark elfoglalta a kitüntetett helyét más fordítóprogram benchmark mellett (mint például a SPEC és a későbbi Openbench).

A szerző hozzájárulásai

- 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 mellet 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:

[11] 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

[10] 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

[55] 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

[67] 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

[68] 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 fordítóprogramok optimalizálása és analízise

Ebben a tézispontban a fő cél az volt, hogy összegyűjtsük, megvizsgáljuk és validáljuk a különböző weboldalakon megjelent *JavaScript* programozási irányelveket, amelyek javíthatják a *JavaScript* futási teljesítményének hatékonyságát. Ezenkívül bemutattuk a *JavaScript* forráskód-elemzésének egyik változatát, amely segíthet a hívási gráfok információira támaszkodó egyéb optimalizálásokban vagy elemzésekben.

A második téziscsoport tárgyalása a 8. és 9. fejezetekben történt meg. Ez a téziscsoport a következő két fő eredményre bontható.

3. JavaScript programozási irányelvek

Kiértékelésre került a JavaScript nyelvhez, az ECMAScript 5.1 verzióhoz íródott irányelvek számos példánya. Továbbá kiértékelés alá lett vonva az ECMAScript 5.1-es és a 6-os verziók között bevezetett új nyelvi elemek hatékonysága. A mért eredmények azt mutatták, hogy az irányelvek továbbra is fontosak, jelentős teljesítményjavulás érhető el egyes irányelvek alkalmazásával. A legmagasabb eredményeket az *'Avoiding With'* irányelvvel lehet elérni, míg a *'Common Subexpression Elimination'* irányelvnek kis hatása van a teljesítményre az ECMAScript 5.1 irányelvek esetén. Másrészt a *'Symbol'* konstrukció ECMAScript 5.1 szimulált változata érte el a legjobb eredményt az ECMAScript 6-hoz képest, míg az *'class'* konstrukció sokkal jobban teljesít az új szabvánnyal. Bár az eredmények nagy része meggyőző, tanácsos időről időre felülvizsgálni őket és újból lemérni ezen irányleveket, hogy még a későbbi ECMAScript verziók között és az új JavaScript futtatómotrok használatával is megállják-e a helyüket.

4. JavaScript hívási gráfjainak dinamikus szoftveranalízise

Számos JavaScript-hívási gráf generátor eszköz lett összehasonlítva és kiértékelve. Az egyiket, a *Nodejs-cg*-t a szerző és szerzőtársai készítették, fejlesztik és tartják karban. A bemutatott többi eszköz sorban a Jalangi2 és a Nodeprof.js volt, amelyek dinamikus hívási gráfok előállítására használhatók. Valamint a WALA, a Closure, az ACG, az npm callgraph és a TAJS volt, amelyek statikus hívási gráfok előállítására hoznak létre.

Először is a dinamikus hívási gráf generátorok hatékonysága lett kiértékelve és összehasonlítva egymással. Hatékonyságuk különféle szempontokon keresztül, tesztprogramok segítségével lettek ellenőrizve. Az egyik ilyen a SunSpider benchmark volt, amely egyszerű JavaScript fájlokat tartalmaz, és a JavaScript végrehajtó motorok különböző részeinek tesztelésére hozták létre. A másik típusú teszt különböző *Node.js* modulokon alapult. Ezek hivatottak betölteni a valós alkalmazásokon történő tesztelést.

Az eredmények azt mutatták, hogy nagyszámú él és csomópont egyedi ezekben a hívási gráfokban. Az egyedi csomópontokat és éleket manuálisan kellett ellenőrizni és csoportokba rendezni. Ezek az elemcsoportok kerültek összehasonlításra. Utána a bennük talált különbözőségek részletezése és okainak felfedése történt meg. Továbbá azt lehetett az eredményekből leolvasni, hogy a hívási gráfok generálása akár nyolcszorosan is lelassíthatja a programok végrehajtást. A lassulás miatt váratlan hibák fordulhatnak elő azoknál a moduloknál, amelyek például külső eszközöket vezérelnek.

A dinamikus hívási gráf generátor eszközök összehasonlítása után már statikus hívási gráf generátorok is hozzá lettek adva a következő összehasonlításhoz. Az összehasonlítás eredménye arra mutatott, hogy minden eszköznek és elemzési megközelítésnek megvannak a maga előnyei és hátrányai. Nem lehet egyértelmű győztest hirdetni. Az egyik eszköz képes volt felismerni a rekurzív hívásokat, a másik nagyobb *recall* értéket tudott elérni, miközben a pontosságot továbbra is elfogadható szinten tartotta, de csak egy eszköz volt képes kezelni az összetett modulokat (ACG). Összességében elmondható, hogy a dinamikus hívási grafikonok pontossága kiemelkedően nagy volt a statikusokhoz képest, azonban a megfelelő tesztek hiánya miatt bizonyos statikus éleket nem rögzítettek, nem jelentek meg a hívási gráfban. A kombinált eredmények azt is megmutatták, hogy több eszköz együttes használatával jobb eredmény érhető el, mint ha csak egy eszközre támaszkodnánk. Más hívási gráfokra építő analízisekben a kombinált technikák alkalmazása a javasolt.

III. B. Összefoglalás

A szerző hozzájárulásai

- 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 társszerzőkkel.

- A szerzőnek meghatározó szerepe volt a "*Dynamic 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 tézisponthoz:

[42] Zoltán Herczeg, Gábor Lóki, Tamás Szirbucz, and Ákos Kiss. Guidelines for JavaScript Programs. Are They Still Necessary? In *SPLST'09 & NWMODE'09. Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, pages 59–71, Tampere, Finnland, 2009

[43] Zoltán Herczeg, Gábor Lóki, Tamás Szirbucz, and Ákos Kiss. Validating JavaScript Guidelines Across Multiple Web Browsers. *Nordic Journal of Computing*, 15:18–31, 2013

[53] 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

[40] 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

[41] 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

[4] 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

Publikációk

A disszertációban bemutatott kutatási eredmények többsége nemzetközi konferenciák és workshopok kiadványaiban vagy folyóiratokban jelent meg. A B.1 táblázat pedig azt mutatja be, hogy mely publikációk a dolgozat mely eredményeit fedik le.

Fejezet	Cím	Publikációk
3.	Code Factoring Techniques in the GCC Compiler	[55, 67, 68]
4.	Binary Code Size Measurement Methods and Benchmark	[10, 11]
8.	JavaScript Guidelines	[42, 43, 53]
9.	Dynamic Analysis of JavaScript's Call Graphs	[4, 40, 41]

Table B.1: *A tézisek és a hozzájuk kapcsolódó publikációk összegzése*

A szerző hozzáteszi, hogy bár ebben a dolgozatban bemutatott eredményekhez az ő hozzájárulása volt a meghatározó, az 'én' ('I') helyett a 'mi' ('we') kifejezést használja önhivatkozásként, hogy elismerje a dolgozat alapjául szolgáló cikkek társszerzőinek hozzájárulását.

Acknowledgments

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.

Bibliography

[1] OpenJS Foundation. ESLint. https://eslint.org/. (Accessed on 2023-08-10).

[2] Aarhus University. TAJS (Type Analyzer for JavaScript). https://github.com/cs-au-dk/TAJS, 2009. (Accessed on 2023-08-10).

[3] AbsInt (Angevandte Informatik). aiPop - Automatic Code Compaction. https://www.absint.com/aipop/index.htm. (Accessed on 2023-08-10).

[4] 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.

[5] ARM Ltd. Common Microcontroller Software Interface Standard (CMSIS). https://www.arm.com/technologies/cmsis. (Accessed on 2023-08-10).

[6] AT&T Corporation. COFFDump utility. https://coffi.readthedocs.io/en/latest/get_started.html. (Accessed on 2023-08-10).

[7] AT&T Corporation. Common object file format (COFF). http://bitsavers.org/pdf/att/unix/System_V_386_Release_3.2/UNIX_System_V_386_Release_3.2_Programmers_Guide_Vol2_1989.pdf, 1983. (Accessed on 2023-08-10).

[8] Gábor Ballabás and Gábor Lóki. CSiBE in the LLVM ecosystem. https://www.llvm.org/devmtg/2016-03/Lightning-Talks/EuroLLVM_2016_paper_22.pdf, 2016. (Accessed on 2023-08-10).

[9] Mihai Bazon. UglifyJS: JavaScript parser / mangler / compressor / beautifier toolkit. https://github.com/mishoo/UglifyJS/releases/tag/v2.8.29. (Accessed on 2023-08-10).

[10] 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.

[11] 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.

[12] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 71–80, 2009.

[13] Wen-Ke Chen, Bengu Li, and Rajiv Gupta. Code compaction of matching single-entry multiple-exit regions. In *Proc. 10th Annual International Static Analysis Symposium*, pages 401–417, June 2003.

[14] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 139–149, 1999.

[15] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[16] Bjorn de Sutter, Bruno de Bus, Koen de Bosschere, and Saumya Debray. Combining global code and data compaction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 2001.

[17] Saumya Debray, William Evans, Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. The Squeeze Project: Executable Code Compression. `https://www2.cs.arizona.edu/projects/squeeze/`. (Accessed on 2023-08-10).

[18] Saumya K. Debray, William Evans, Robert Muth, and Bjorn de Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[19] Department of Software Engineering, University of Szeged. Code-Size Benchmark Environment (CSiBE) System. `http://szeged.github.io/csibe/`. (Accessed on 2023-08-10).

[20] Department of Software Engineering, University of Szeged. Offline benchmark of CSiBE. `https://github.com/szeged/csibe`. (Accessed on 2023-08-10).

[21] Ecma International. ECMAScript Language Specification 5.1. `https://262.ecma-international.org/5.1/`. (Accessed on 2023-08-10).

[22] Ecma International. ECMAScript Language Specification 6.0. `https://262.ecma-international.org/6.0/`. (Accessed on 2023-08-10).

[23] ExactCODE GmbH. Openbench. `http://exactcode.com/opensource/openbench/`, 2005. (Accessed on 2023-08-10).

[24] Allen F. Interprocedural Data Flow Analysis. In *Information Processing 74 (Software)*, pages 398–402. North-Holland Publishing Co., Amsterdam, The Netherlands, 1974.

[25] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. ACG. `https://github.com/Persper/js-callgraph`, 2013. (Accessed on 2023-08-10).

[26] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.

[27] Stephen Fink and Julian Dolby. WALA–The TJ Watson Libraries for Analysis. `https://wala.sourceforge.net/wiki/index.php/Main_Page`, 2012. (Accessed on 2023-08-10).

[28] Free Software Foundation. GNU binutils. `https://www.gnu.org/software/binutils/`. (Accessed on 2023-08-10).

[29] Free Software Foundation. GNU Compiler Collection (GCC). `http://gcc.gnu.org/`. (Accessed on 2023-08-10).

[30] Free Software Foundation. GNU Compiler Collection (GCC) internals. `https://gcc.gnu.org/onlinedocs/gccint/`. (Accessed on 2023-08-10).

[31] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 465–478, New York, NY, USA, 2009. Association for Computing Machinery.

[32] Gunar Gessner. NPM Callgraph. `https://github.com/gunar/callgraph`. (Accessed on 2023-08-10).

[33] Google. Chrome DevTools. `https://developer.chrome.com/docs/devtools/`. (Accessed on 2023-08-10).

[34] Google. V8 JavaScript engine. `https://v8.dev/`, 2008. (Accessed on 2023-08-10).

[35] Google. Google Colsure Compiler. `https://github.com/google/closure-compiler`, 2009. (Accessed on 2023-08-10).

[36] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[37] Sacha Greif and Eric Burel. State of JavaScript. `https://2022.stateofjs.com/en-US/usage/`, 2022. (Accessed on 2023-08-10).

[38] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: a benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101. IEEE, 2019.

[39] Jungwoo Ha, Mohammad R Haghighat, Shengnan Cong, and Kathryn S McKinley. A concurrent trace-based just-in-time compiler for single-threaded javascript. *Proc. PESPMA*, 2009.

[40] 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.

[41] 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.

[42] 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, Finnland, 2009.

[43] Zoltán Herczeg, Gábor Lóki, Tamás Szirbucz, and Ákos Kiss. Validating JavaScript Guidelines Across Multiple Web Browsers. *Nordic Journal of Computing*, 15:18–31, 2013.

[44] Jan Hubička. The GCC call graph module, a framework for interprocedural optimization. In *Proceedings of the 2004 GCC Developers' Summit*, pages 65–75, June 2004.

[45] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Static Analysis: 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings 16*, pages 238–255. Springer, 2009.

[46] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[47] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2002.

[48] Seong-Won Lee, Soo-Mook Moon, Won-Ki Jung, Jin-Seok Oh, and Hyeong-Seok Oh. Code size and performance optimization for mobile JavaScript just-in-time compiler. In *Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture*, 2010.

[49] Ond Lhoták et al. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 37–42. ACM, 2007.

[50] Linaro and ARM. The LLVM Embedded Toolchain for ARM. `https://resources.linaro.org/en/resource/cYa6kFWBVrMBLmrZXomN2X`, 2021. (Accessed on 2023-08-10).

[51] LLVM Developer Group. The LLVM Compiler Infrastructure. `https://llvm.org/`. (Accessed on 2023-08-10).

[52] Gábor Lóki. Home page of CFO branch. `https://gcc.gnu.org/projects/cfo.html`, 2004. (Accessed on 2023-08-10).

[53] 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.

[54] Gábor Lóki and Zoltán Herczeg. Dynamic call graph generators for JavaScript. `https://github.com/szeged/js-call-graphs/tree/call-graphs`, 2019. (Accessed on 2023-08-10).

[55] 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.

[56] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 499–509, 2013.

[57] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven node.js JavaScript applications. *SIGPLAN Not.*, 50(10):505–519, October 2015.

[58] Scott A Mahlke, David C Lin, William Y Chen, Richard E Hank, and Roger A Bringmann. Effective compiler support for predicated execution using the hyperblock. *ACM SIGMICRO Newsletter*, 23(1-2):45–54, 1992.

[59] Felix Maier. Iroh a dynamic code analysis for JavaScript, 2017. `https://maierfelix.github.io/Iroh/`.

[60] Ward Douglas Maurer and Ted G Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975.

[61] Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.

[62] Mozilla. Rhino is an open-source implementation of JavaScript written entirely in Java. `https://github.com/mozilla/rhino`, 1997. (Accessed on 2023-08-10).

[63] Mozilla Corporation. Servo, the parallel browser engine. `https://servo.org/`. (Accessed on 2023-08-10).

[64] Steven Muchnick. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[65] Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann Publishers, 1997.

[66] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An Empirical Study of Static Call Graph Extractors. *ACM Trans. Softw. Eng. Methodol.*, 7(2):158–191, April 1998.

[67] 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.

[68] 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.

[69] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Principles of program analysis. 1999.

[70] Diego Novillo. Tree SSA a new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, May 2003.

[71] Diego Novillo. Design and implementation of Tree SSA. In *Proceedings of the 2004 GCC Developers' Summit*, pages 119–130, June 2004.

[72] Henrik Ölund and Jonatan Karlsson. Investigation of the key features in ECMAScript 2015, 2016.

[73] OpenJS Foundation. Node.js: open-source, cross-platform JavaScript runtime environment. `https://nodejs.org`, 2009. (Accessed on 2023-08-10).

[74] Addy Osmani. How to write fast, memory-efficient JavaScript, 2012. `https://www.smashingmagazine.com/2012/11/writing-fast-memory-efficient-javascript/`.

[75] Bhadresh Panchal. Node.js Usage Statistics. `https://radixweb.com/blog/nodejs-usage-statistics`, 2022. (Accessed on 2023-08-10).

[76] Milos Poletanović, Miodrag Dukić, Dragan Mladenović, and Zoran Jovanović. Implementation of machine outliner for nanomips in the llvm compiler infrastructure. In *2022 IEEE Zooming Innovation in Consumer Technologies Conference (ZINC)*, pages 140–144. IEEE, 2022.

[77] Xiaoxia Ren and Barbara G. Ryder. Heuristic ranking of java program edits for fault localization. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*, pages 239–249, New York, NY, USA, 2007. ACM.

[78] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 1–12, New York, NY, USA, 2010. Association for Computing Machinery.

[79] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. Hyfm: Function merging for free. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 110–121, 2021.

[80] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. Effective function merging in the ssa form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 854–868, 2020.

[81] Roger Sayle. PR ipa/103601: ICE compiling CSiBE. `https://gcc.gnu.org/pipermail/gcc-patches/2021-December/586550.html`. (Accessed on 2023-08-10).

[82] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498, 2013.

[83] Koushik Sen, Manu Sridharan, and Christoffer Adamsen. Jalangi2 dynamic analyses framework for JavaScript. `https://github.com/Samsung/jalangi2`, 2015. (Accessed on 2023-08-10).

[84] L. H. Silva, M. Ramos, M. T. Valente, A. Bergel, and N. Anquetil. Does JavaScript software embrace classes? In *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 73–82, 2015.

[85] Leonardo Humberto Silva, Daniel Hovadick, Marco Tulio Valente, Alexandre Bergel, Nicolas Anquetil, and Anne Etien. JSClassFinder: A tool to detect class-like structures in JavaScript. *CoRR*, abs/1602.05891, 2016.

[86] Snyk. Snyk Code. `https://snyk.io/`. (Accessed on 2023-08-10).

[87] Steve Souders. *Even faster web sites: performance best practices for web developers*. " O'Reilly Media, Inc.", 2009.

[88] StackOverflow. Stack Overflow Developer Survey. `https://insights.stackoverflow.com/survey/2021`, 2021. (Accessed on 2023-08-10).

[89] Standard Performance Evaluation Corporation. Standard Performance Evaluation Corporation Benckmark (SPEC). `https://spec.org/benchmarks.html`, 1988. (Accessed on 2023-08-10).

[90] Statista Inc. Most used web frameworks among developers worldwide. `https://www.statista.com/statistics/1124699/`

worldwide-developer-survey-most-used-frameworks-web/, 2023. (Accessed on 2023-08-10).

[91] Sean Stirling, Rocha Rodrigo CO, Kim Hazelwood, Hugh Leather, Michael O'Boyle, and Pavlos Petoumenos. F3m: Fast focused function merging. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 242–253. IEEE, 2022.

[92] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for node. js. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 196–206, 2018.

[93] The GNU Compiler Collection. GCC. GCC benchmarks homepage. https://gcc.gnu.org/benchmarks/. (Accessed on 2023-08-10).

[94] T. R. Toma and M. S. Islam. An efficient mechanism of generating call graph for JavaScript using dynamic analysis in web application. In *2014 International Conference on Informatics, Electronics Vision*, pages 1–6, May 2014.

[95] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers, 2nd edition, 2011.

[96] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 204–215. IEEE, 2003.

[97] Burak Turhan, Gozde Kocak, and Ayse Bener. Software defect prediction using call graph based ranking (cgbr) framework. In *Proceedings of the 2008 34th Euromicro Conference Software Engineering and Advanced Applications (SEAA '08)*, pages 191–198, Washington, DC, USA, 2008. IEEE Computer Society.

[98] Christoph W. Ueberhuber. *Numerical computation: methods, software, and analysis*. Springer, 1997.

[99] Unix System Laboratories. ELF file format. https://refspecs.linuxbase.org/, 1995. (Accessed on 2023-08-10).

[100] WebKit Team. SunSpider Benchmark. https://github.com/WebKit/webkit/tree/main/PerformanceTests/SunSpider/tests/sunspider-1.0.2, 2007. (Accessed on 2023-08-10).

[101] Mark Wilton-Jones. Efficient JavaScript, 2006. https://dev.opera.com/articles/efficient-javascript/.

[102] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 662–676, 2017.

[103] Tao Xie and David Notkin. An Empirical Study of Java Dynamic Call Graph Extractors. *University of Washington CSE Technical Report 02-12*, 3, 2002.

[104] Chuan Yue and Haining Wang. Characterizing insecure javascript practices on the web. In *Proceedings of the 18th international conference on World wide web*, pages 961–970, 2009.

[105] Nicholas C. Zakas. Speed up your JavaScript: The talk, 2009. https://www.nczonline.net/blog/2009/06/05/speed-up-your-javascript-the-talk/.

[106] Nicholas C. Zakas. *Writing Efficient JavaScript – Even Faster Websites*, chapter 7. O'Reilly Media, 2009.