Department of Mathematics and Computer Science
Database Research Group

Improving JavaScript performance using call graphs and graph databases.

Patrick Strookappe

Supervisor:
dr. Daniele Bonetta
Other assessment committee members:
dr. Odysseas Papapetrou
dr. Renata Medeiros de Carvalho

Eindhoven, May 2022

Abstract

Language Virtual Machines often feature a Just-in-time (JIT) compiler that can optimize an application at runtime, without the need for any Ahead-of-time knowledge. GraalVM is an example of such a virtual machine. It supports multiple programming languages and is able to both use AOT and JIT compilation. This thesis researches how static AOT information can be used to improve the performance of the JIT compiler. The information is gathered by building a polyglot tool that implements a call graph profiler. The resulting call graph is analysed by a graph database and provides the JIT compiler with targets for compilation. This thesis shows that the static information from a call graph can provide valuable information that improves both the startup time and total performance of a virtual machine.

Contents

List of Figures

Chapter 1

Introduction

A Language Virtual Machine is an engine that provides a platform for compatible languages to be executed regardless of the underlying hardware. Besides their portability these languages are also flexible as they can inter operate with each other, due to a shared underlying bytecode. Because optimisation efforts can be focused on a single virtual machine instead of on each language individually, these virtual machines often manage to obtain good performance.

One problem in optimising dynamic languages, such as JavaScript and Python, is that many optimisations can only be inferred at runtime. This forces an underlying virtual machine to try to perform these optimisation with incomplete runtime knowledge, such that heuristics have to be used. These heuristics often require trade offs to reach good performance. Just In Time (JIT) compilation is the process of using runtime information to make decisions that improve virtual machine performance. It also uses this runtime information to produce machine code.

One way to increase the performance of virtual machines using JIT compilation is to provide the virtual machine with additional information that is provided Ahead Of Time (AOT). This is also called profile-guided optimisation.

In order to help the virtual machine make compilation decisions, information about the functions in the applications is needed. Additional information can be found by looking at the calls between functions. These calls can suggest interprocedural optimisations to be performed, further improving the virtual machine performance.

Call graphs are one of the profiling methods that can gather information on both the functions and calls in a program execution [4]. A call graph is a data structure that shows calls made from one function to another function. These call graphs can not only help virtual machines make decisions on optimisation, but also help developers debug their program. Call graphs are therefore ideal candidates to use for gathering AOT information to assist the virtual machine in making compilation decisions.

In order for a virtual machine to extract useful data from a call graph, it must be in a readable data model. Graph databases are a logical candidate for this. A graph database is a database type used for highly connected and very large graphs. Graph databases have especially large advantages in querying speed for queries often used in graphs, such as path queries.

Once the call graph is stored in a graph database, queries can be used to identify functions that might provide useful information to the virtual machine. Another approach is to look for interprocedural optimisations that can be performed between functions, for example using monomorphization or inlining. These suggestions can then be sent to the virtual machine, which will use this information to improve compilation performance [16].

GraalVM is a polyglot JDK distribution that is developed to provide a high performance runtime for various programming languages [30]. It provides a framework that allows developers to build profiling tools that support multiple languages. This instrumentation framework leverages the underlying VM, which is compatible with multiple languages, such that a single profiling tool built with this framework can be used with all compatible VM languages.

Besides a framework to build polyglot tools in, GraalVM also provides a high performance runtime environment. This runtime makes JIT decisions about which parts of the source code to interpret and which to compile. Using the call graph in the graph database, queries can identify parts of the source code that may benefit from compilation and pass these on to be compiled instead of interpreted.

Because GraalVM supports multiple languages, the profiling tool should be compatible with all supported languages of GraalVM. However, choosing a single language for benchmarking purposes makes the process of validation and comparison much easier. For this reason JavaScript is chosen as the focus of this thesis. Multiple JavaScript benchmarks will be profiled and optimised. GraalJS is an implementation of JavaScript that is built to work on GraalVM [32]. GraalJS will be used to exceute the JavaScript benchmarks, such that they can be further profiled and optimised in GraalVM.

1.1 Problem statement

This master thesis aims to use the GraalVM instrumentation framework to create a tool that can analyse an instance of the execution of an application. This analysis tool then produces a dynamic call graph, representing the execution of this application. Although performance of the call graph analysis tool is not a focus, it should still excecute without excessive overhead. The call graph tool will be tested and validated on JavaScript, but should be able to be easily extended to other programming languages.

The second goal of the thesis is to output this call graph in a format that can be easily imported into a graph database. The graph database should be able to create visualisations that help understand and debug source code. The graph database should also allow efficient querying over the call graph. These queries should analyse the graph and return functions that assist the virtual machine in making decisions on compilation strategy.

These returned functions should focus on two areas: Functions that have a high performance impact when compiled and functions that benefit from interprocedural optimisations.

Finally, the compilation strategy of the virtual machine should be adapted such that it can use these returned functions in a way that improves it's compilation strategy.

1.2 Outline

Section 2 describes the required background information on call graphs, graph databases and GraalVM necessary for the succeeding sections. Section 3 discusses the related work on call graph profiling and using call graphs for optimisation. Section 5 describes the methodology and implementation of the call graph profiler in GraalVM. Section 6 discusses how the call graphs are imported into Neo4j and what queries are run on the database, and how the output is transformed into a format that can be used by the VM. Section 7 discusses how these query results are used to improve the compilation strategy. Section 8 discusses the benchmarking procedure, the resulting call graphs, and the performance impacts the call graph output has on the VM. Finally the master thesis concludes with the conclusion and discussion in section 9.

Chapter 2

Background

2.1 Call graphs

In order to understand the behaviour of applications either to debug or optimise them they need to be profiled. Profiling is the process of extracting information from an application either dynamically, by executing the program, or statically. One popular profiling technique is to create a representation of an application execution. There are multiple approaches to realise this, but one of the most popular is to use graphs.

A graph is a data structure that consists of nodes and edges. These nodes may be connected to each other by edges. These edges may have a specific direction or they may imply an undirected relationship between nodes. These nodes are generally uniquely identifiable and may have other attributes that are not unique. Edges must be connected on both sides by nodes, either between 2 different nodes or as a self loop.

There are multiple graph representations that can be used to profile an application, such as control flow or data flow graphs. The most widely used and most applicable to this research is the call graph. A call graph is a graph that represents the calling relationships between between functions in a computer program. Each node represents a function and each edge represents a call from one function to another or its self. For this project, each node represents a JavaScript function. These functions are either defined in the source code of a specific benchmark, or by builtin functions of the JavaScript language. Each edge represents a call, or invocation, between functions. These calls can have several attributes, such as arguments types when the call is made, or return types when the call is completed. The nodes can also have multiple attributes, for example, an estimation of its size. Figure 2.3 shows an example of the call graph of a simple computer program.

Figure 2.1 shows a simple JavaScript program that starts with a main being called from the program body. This call can be seen on the first line in figure 2.2, which is a list of all calls. Figure 2.3 shows all the functions in the program and how they are connected by calls. There can be multiple calls that connect the same functions. This is the case when attributes of the call are different, in this case the arguments. There can also be multiple calls for the same nodes when either the call site or the return type differs. The call site is the exact location where the call is made and, if in a function the same call is performed in different locations, these would be considered as different edges. The call target is defined as the function that is being called from some call site.

```
function main(){
  a();
  b();
}

function a(){
}

function b(){
  c(15);
  c("someString");
}

function c(argument){
}

main();
```

Figure 2.1: Source code of example JavaScript application.

Source	Call Argument	Target
:program	null	main()
main()	null	a()
main()	null	b()
b()	integer	c()
b()	string	c()

Figure 2.2: Edge list resulting from profiling call graph from example JavaScript program in figure 2.1.

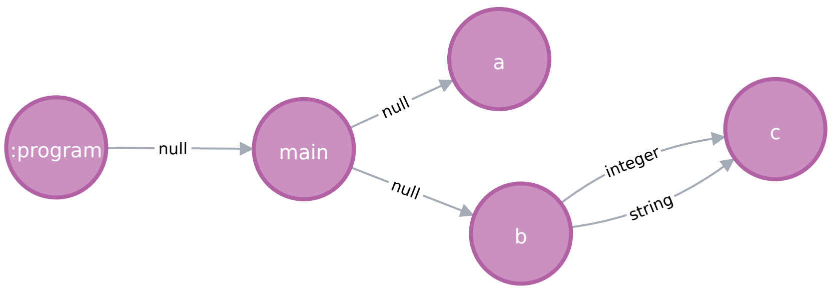


Figure 2.3: Call graph resulting from source code 2.1 and edge list 2.2

Call graphs can either be dynamic or static. Dynamic call graphs represent a specific instance of an execution of an application. Static call graphs model every possible execution of an application without running the program. Both methods have advantages and disadvantages. Static call graphs are great at debugging large software projects, as code need not be run to get a graph. In the introduction there was already a discussion about why JIT compilers are a necessity when optimising dynamically typed languages such as JavaScript. For the same reason, dynamic call graphs are also a necessity when profiling a dynamic language. This is because at compile time the types of variables are not known yet and thus the calls that are going to be made often cannot be known.

In this thesis dynamic call graphs are used. Specifically, context-insensitive call graphs are analysed. This means that each function corresponds to exactly one node in the call graph, and each node in the call graph represents a single function. Contextual information, such as argument types, are not used to uniquely identify a function. However, some optimisations, such as splits, can only be decided on if the arguments of a function are known. To solve this problem the arguments will be stored in the calls [13].

2.2 Graph databases

After profiling a computer program the resulting call graph needs to be imported into a database so that it can be queried on. In order to decide which kind of database to use, it is important to look at the important characteristics of the data and what type of queries are required. In order to do research on interprocedural patterns it is important to be able to traverse the graph and look at local structures. Traditional relational databases generally do not perform well on these type of queries [49].

The solution to this problem is to not use a traditional relational database, but a different database type: A graph database. Graph databases have a data-model that is explicitly designed for efficiently storing and performing queries on large and highly interconnected graphs. These advantages are accomplished by leveraging some unique attributes of graphs and adapting the database schema model towards it. An example of such an attribute is that graph connections have local constructs: A connection is strictly between two nodes. Trying to query over such a local constructs in a relational database will involve traversing over much unrelated data, while graph databases use small local lists within each node or edge that are very quickly traversed [24].

A schema must be used that describes how to store the data retrieved from the application. A convenient way is to only store the edges. Because each edge must be connected on both sides by nodes, the node identifiers can be added to the edge data, together with its attributes. These nodes can then later be abstracted from the edge list when it is loaded in a graph database. This technique is shown in figure 2.2 and is called an edge list. Each tuple in the database corresponds to one edge in the graph. By adding information about this edge to each tuple, a triple is created. such a triple, where there is an subject, an edge, and an object also corresponds to the Resource Description Framework (RDF). The RDF is designed to allow easy communication between different formats on the World Wide Web. Using this format allows the call graph to be easily compared to others and compatible with other applications [5]. This approach has successfully been used to create call graph profiling tools. Gprof, an early example of a call graph profiling tool, suggests a schema that collects the call count information in the edges, and then abstract from these edges the incoming calls for each function [12].

2.3 GraalVM

GraalVM is a polyglot implementation of a JDK. It is designed to let developers use multiple languages, either JVM languages or others, in the same environment without the penalties that would normally be associated with foreign language calls [34].

GraalVM supports multiple programming languages through its Truffle Language Implementation Framework. This framework allows language developers to implement language interpreters that can interact with the underlying GraalVM compiler. An advantage of using such a polyglot framework is that more time can be spent on optimising a single underlying virtual machine and then having all supported languages benefit from these optimisations, instead of having to spend time optimising many different languages separately [36] [33].

An example of such a language is JavaScript. GraalJS uses the Truffle Language Implementation Framework to interpret JavaScript [39]. Since JavaScript can now be run in GraalVM through GraalJS, performing optimisations on GraalVM will increase the performance of JavaScript programs.

Another feature of the Truffle Language Implementation Framework is the ability to create cross-language tools, which work on all Truffle supported languages [35]. This is the framework used for the call graph profiler in this thesis.

In order for all these tools to operate across languages an Intermediate Representation (IR) is needed between the source code language and the JVM. A Truffle language, for example GraalJS, will translate the JavaScript application into this IR, which can then be read by the GraalVM compiler. this IR is in the form of an Abstract Syntax Tree (AST). Each node in the AST represents a single value,statement, or operation in the guest language [44].

2.4 Runtime Dynamic (JIT) Compilation

One of the goals of this thesis is to investigate whether static information gathered ahead of time can assist the virtual machine in making decisions about compilation strategy. In order to

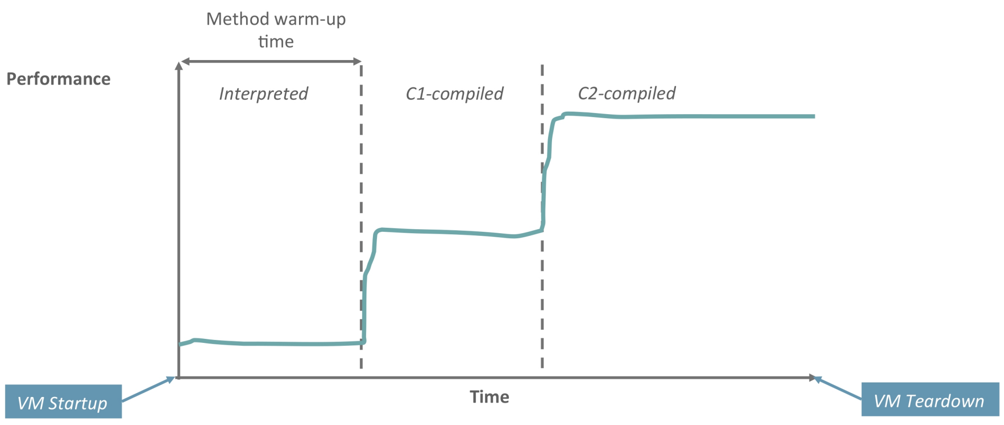


Figure 2.4: Impression of how tiered execution influences performance [48].

understand what sort of static information needs to be gathered, it is important to investigate how the default dynamic compilation strategy of the virtual machine works.

There are three levels of execution available for each JavaScript function. The first level is an interpreter. Here, the JavaScript source code is executed directly by the GraalJS interpreter. The second level is tier 1 optimisation. In this case, the function is compiled and small optimisations are made, such as inlining. The third level is tier 2 compilation. This compilation mode takes the most time and tries to further optimise the already compiled tier 1 code. Figure 2.4 shows an impression of a functions lifespan through a virtual machine. A function is interpreted until it reaches a threshold and then during a short compilation period it gets compiled. After compilation its performance has increased and will now execute faster until it reaches a second threshold, where tier 2 compilation takes place. This thesis focuses on the interpretation and the tier 1 compilation stage.

The virtual machine keeps count of how often every function is executed. This will be referred to as the callCount. In order to make sure that compilation resources are not wasted on infrequently used functions there is a compilation threshold. Functions that are below this threshold are interpreted. Whenever a function hits this threshold it gets submitted to the compilation queue.

The threshold is changed dynamically based on the size of the current compilation queue. This is a measure to ensure that the compilation queue does not grow too large.

Figure 2.5 shows the compilation queue where a new function b() is about to be added. The function first receives a priority based on the callCount and the compilation tier. Since b() has only been interpreted until this point tier 1 compilation is performed. Figure 2.6 shows that based on its callCount b() is placed between SomeFunction2() and SomeFunction3(). Based on its tier it is placed before SomeFunction4() even though SomeFunction4 has a much higher callCount. This heuristic is based on the fact that in the time it takes to perform a tier 2 compilation many tier 1 compilations can be performed, which almost always results in better performance.

Compilation threshold: 300		
New Entry:		
Function	callCount	Tier
b()	301	tier 1
Compilation queue:		
SomeFunction1()	5000	tier 1
SomeFunction2()	400	tier 1
SomeFunction3()	100	tier 1
SomeFunction4()	11000	tier 2
SomeFunction5()	4000	tier 2

Figure 2.5: Compilation queue before new entry is added.

Compilation threshold: 400		
Compilation queue:		
SomeFunction1()	5000	tier 1
SomeFunction2()	400	tier 1
b()	301	tier 1
SomeFunction3()	100	tier 1
SomeFunction4()	11000	tier 2
SomeFunction5()	4000	tier 2

Figure 2.6: Compilation queue after new entry is added.

After the compiler requests a new compilation target the whole compilation queue is scanned for the function with the highest weight value. Weight is a continuously updating value, based on the callCount, and the callCount in the past millisecond. This causes functions that are currently not being called frequently to be moved to the bottom of the queue [38].

Besides looking at when to compile functions, the virtual machine also decides when interprocedural optimisations are performed, such as inlining and splitting.

When a call is made from a call site to a call target, it is possible to place the whole target function at the location of the call site. Doing this has two major advantages. First, a call costs a small amount of compute time. By eliminating calls there a small performance increase is realised. Especially for small and often called functions eliminating call overhead improves performance significantly. Figure 2.7 shows how inlining can decrease the size of a call graph.

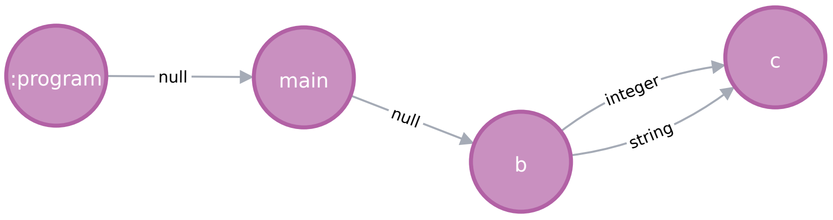


Figure 2.7: Call graph after inlining function a() from call graph in figure 2.3.

The other interprocedural optimisation technique is splitting. Splitting is used in cases where a single function may be called with different arguments, such as function c() in example figure 2.3. When trying to optimise c(), the compiler needs to check what sort of argument is send with each occurrence of the call. Because, for example, String additions are optimised differently than Integer additions, different compilations approaches are necessary. By splitting the function c() into two functions, c1() and c2(), the compiler does not need to perform expensive type checking within the compiled code [37]. Another benefit is that now both these functions can separately be considered for compilation. If, for example, c1() is called a 1000 times and c2() only 50 times, the call graph in figure 2.3 can only compile c() as a whole. In figure 2.8 only c1() can be targeted, improving compilation efficiency.

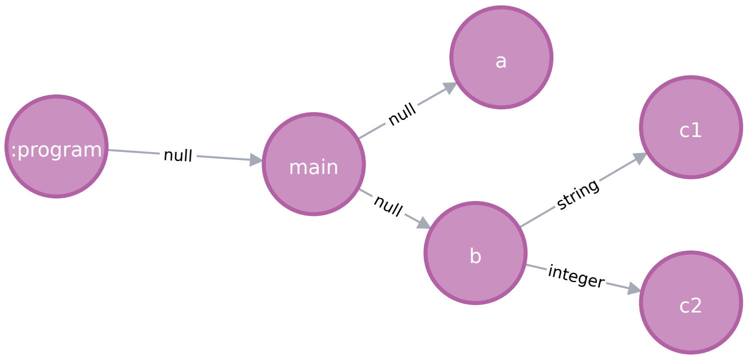


Figure 2.8: Call graph after splitting function c() into c1() and c2().

Chapter 3

Related work

3.1 Call Graphs

Much work has been done in research on profiling static and dynamic call graphs. One major problem in the creation of static call graphs is that the size of the graph increases rapidly as the number of functions in a program increases [20]. A solution for this problem is to use a sampling based approach to create a graph. Sampling, however, may leave unconnected areas in the graph, decreasing the usefulness of the results. Another approach is to survey only a small part of the graph at a time. This can for example be done by only looking at local clusters of functions [1]. Other approaches focus on ignoring a specific subset of functions, such as external or builtin functions [3]. The problem however remains, since no execution of the program is performed, parts of the graph will be overestimated, and, with larger programs, parts will be underestimated. This bias can be especially prevalent in low overhead time based sampling approaches, where some functions may take longer to complete than others. Some work has been done on mitigating this issue by correcting the sampling interval based on the spacing of call events [22]. Further problems occur when trying to create static call graphs for dynamically typed languages such as python [41], since the differing types of variables cause even more possible calls to evaluated.

Dynamic call graphs generally do not have this problem. As an execution of the program is traced, there is only overhead for each call that occurs. This penalty is generally low, and the number of calls will be linearly related to time it takes to run a program. There is, however, still a trade-off between the cost of creating the call-graph and the accuracy of this graph. Running a single execution of a application will very cheaply create a call graph, but it may not be representative of all possible iterations of an application. Running the call graph profiler constantly would negate any potential improvements that additional optimisations provide. Thus still, a suitable time frame should be chosen after which the call graph is deemed to be complete. Then, analysis will be based on this 'complete' graph. The time frame chosen could significantly impact the accuracy of the call graph and thus the analysis that results from it [14].

This complication is deemed outside the scope of this thesis and thus deterministic benchmarks will be used. In a deterministic benchmark a single iteration resembles the correct complete graph. It is important to note that the call graph does still change slightly during compilation due to the compiler making changes to the functions because of interprocedural optimisations, such as inlining. The call graphs are constructed without these optimisations enabled. Thus the call graph may still be slightly different from the one being benchmarked on.

For a dynamic call graph profiler to work, listener code must be inserted either on the source code or byte code level [52]. An advantage to using the byte code level is that for some programming languages the same byte code may be shared. An example is that it is possible to use the underlying JVM byte code to create a generalised call graph constructor for languages that use the same underlying JVM. These systems could be extended, but also made simpler by using polyglot frameworks [2].

After a call graph is created it needs to be made available either to a developer or to a machine so that analysis can take place.

Call graph visualisation is one of the key instruments in software visualisation. One of its key problem is that in both dynamic and static call graphs the numbers of nodes and edges quickly become unreadable due to their size [21]. A popular solution is to not only sample during the creation of the call graph, but also during its visualisation. An example is time based sampling: Parts of the call graph may be separately gathered or rendered. This not only provides the user with a more manageable view, but also provides additional information in the form of the progression of the calls in a computer program [7].

Other research in the time domain suggests an interesting variable to add might be that of execution time. In order to do this during execution time, functions must be kept available if they are active. Since the number of active functions is generally fairly low, this causes low overhead [43].

If knowledge of the execution times is present this may then be further used to optimise the edge collection and visualisation by only looking at parts of the graph that have a high performance impact [8].

Other work on visualising call graphs focuses on the differences between different executions of a program. This technique is interesting both for the ability to show graphs evolving over time, and for being able to compare different executions of a graph [6]. Even if the executions of a graph are the same, as is the case in this thesis, different compilations techniques still change how the graph evolves over time. Looking at these differences together with performance statistics may provide valuable insights.

Comparing call graphs is also important in validating their correctness and performance. Especially when low overhead is a priority sampling decisions may influence the call graph significantly. Research has been done on creating tools which provide similarity statistics and allows users to visually inspect differing areas [23].

A final possible use for call graphs is security. In dynamic languages such as JavaScript there are often many dependencies in large software programs. It is important to know what these dependencies are and how they are used. Call graphs might be a tool to investigate this [29]. Another use case for security uses previously discussed comparison methods. By making comparisons between call graphs of different malware programs classifications can be made about what the malware does. This might help security specialists identify solutions to protect against it [17].

3.2 Profile-guided optimisation

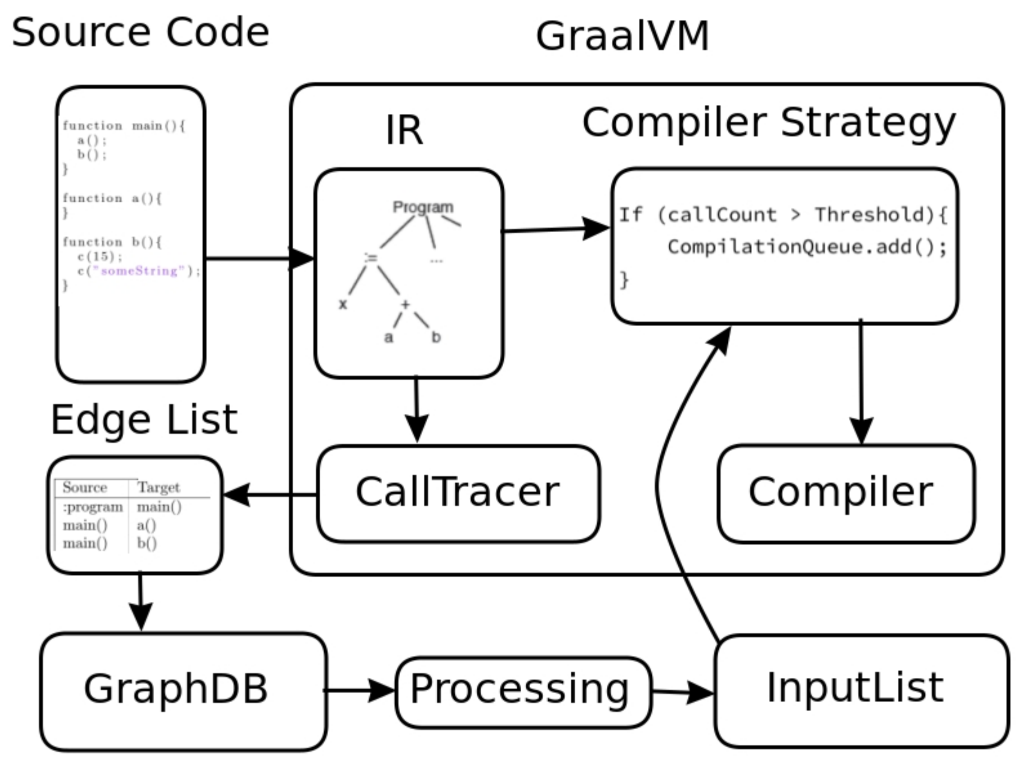
Another use case for call graphs is profile-guided optimisation. Profile-guided optimisation is the process of using profiling information to inform optimisation decisions. Profile guided optimisation is used successfully to analyse dynamic behaviour with for example machine learning [40]. Other profile-guide optimisation approaches target different parts of programs, such as inter process communication [19] or binary level rewriting [51].

One of the problems in AOT compilers is that especially in dynamically typed languages, not all optimisations that JIT compilers can find are also identified by AOT compilers. Profile-guided optimisation may help these AOT compilers find these optimisations. Furthermore, profile-guided optimisation may enable compilers to use both AOT and JIT information to further optimise code [42]. Another solution to optimising dynamic languages is to use profile-guided optimisation to statically infer the types [9] or classes [15].

One of the challenges in profile-guided optimisation is to keep overhead as low as possible. In order to achieve this a trade-off needs to be made between profile accuracy and profile cost. Some profile-guided optimization techniques can achieve performance increases of 15%, while only having close to 1 % overhead cost [50]. Profile-guided optimization techniques have successfully been implemented in large software projects such as Google Chrome [10],GCC, and Intel's C++ compiler [18].

Chapter 4

Methodology



```
function main(){
  a();
  b();
}

function a(){
}

function b(){
  c(15);
  c("someString");
}
```

```
If (callCount > Threshold){
    CompilationQueue.add();
}
```

Source	Target
:program	main()
main()	a()
main()	b()

Figure 4.1: Project structure of all implementation parts of the thesis.

Figure 4.1 shows an overview of the project structure. Starting with the source code of the JavaS-cript benchmarks. GraalJS performs the translation from the source code to the Abstract Syntax Tree (AST) that GraalVM can execute. This AST is a form of an Intermediate Representation (IR).

Section 5 discusses the CallTracer tool. This tool will execute an iteration of the JavaScript benchmark. During this execution CallTracer will profile the IR and the resulting call graph will be exported into an edge list.

Section 6 discusses how this edge list is imported into the graph database. After querying for relevant information the graph database queries return a list of important functions that can be used to improve the compilation strategy. These query results must first be processed into a format that can be used by the virtual machine. This results in an InputList that is given to the virtual machine.

At this point it is important to note that the first two steps occur in a separate process from the last step. Using three iterations of the benchmark, the InputList is created. After this, the benchmark is run again from the start. Section 7 describes how the InputList is used to replace and improve the default compilation strategy.

Chapter 5

The call graph profiling tool

The CallTracer tool is based on the instrument API of GraalVM. The instrument API allows the tracking of events during runtime of an application. Figure 5.1 shows an example of a piece of JavaScript source code. Some important terminology for this chapter is that the location of a call (line 3) is called the callSite. The callTarget is the location of a function being called, in this case: Line 5-7. The callNode represents the call itself and contains both the callSite and callTarget. Figure 5.2 shows how the translation is made between a piece of source code and the AST. This figure shows how the GraalVM manages different languages. It makes a representation that would be the same in different languages. The context encompasses a section of this AST and a tag describes a single node in the AST.

```
1 function main(){
2   i = 5;
3   a(5);
4 }
5 function a(arg){
6     arg + arg;
7 }
```

Figure 5.1: Source code of example JavaScript program.

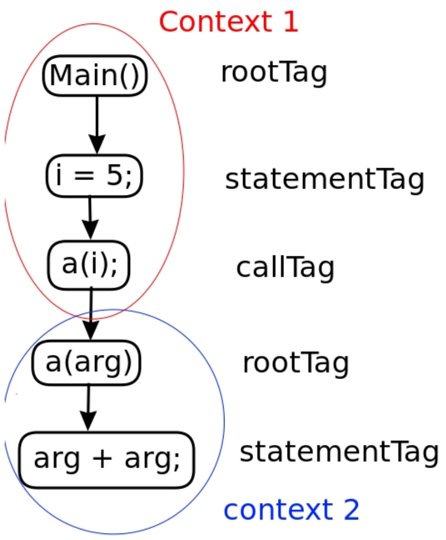


Figure 5.2: AST of the Intermediate Representation of GraalVM of figure 5.1

Figure 5.3 shows the structure of the CallTracer tool. The CallTracerInstrument decides whether or not profiling is enabled. If it is enabled it sends a sourceSectionFilter towards the CallTracer. The sourceSectionFilter decides what parts of the source code are going to be profiled. The CallTracer then profile all these sourceSections and retrieves call and function information from them. It sends this profiled information to the CallTracerCLI. The callTracerCLI transforms this information into a schema that can be read by the graph database: The EdgeList.

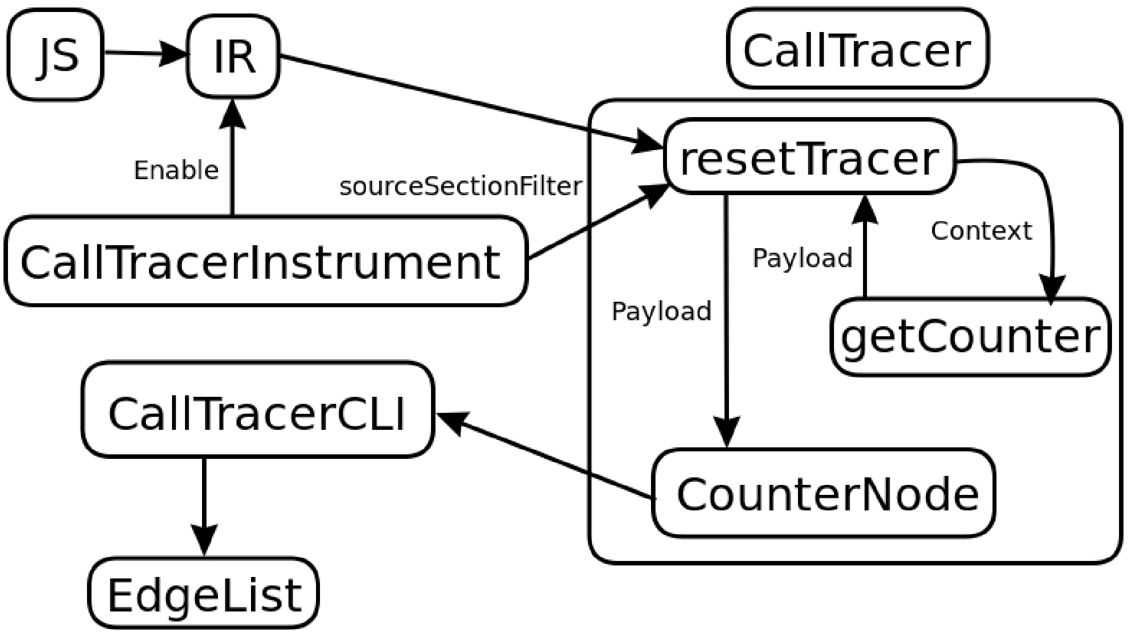


Figure 5.3: Program Structure of the CallTracer profiling tool.

The CallTracer classes are based on the implementation of an existing tool: CPUTracer [31]. CPUTracer tracks the source code location and the number of times either a call is made, a root is visited, or a statement occurs. The most important aspects of the CallTracer and the design decision behind it will be discussed next. Complete code can be found in the the Github repository [46].

5.1 CallTracerInstrument

The CallTracerInstrument is the starting point of the CallTracer tool. It implements the TruffleInstrumentClass, which is the framework of the instrument API. This class enables and initialises the instrumentation and has two important methods: onCreate() and getSourceSectionFilter().

The onCreate() method retrieves all selected options from the CallTracerCLI class. These options determine what data is going the be gathered by the CallTracer. For example, whether to add return types or call times to the call graph. Most importantly however, it sets up the SourceSectionFilter.

The SourceSectionFilter decides what parts of the source code are being profiled. Specifically of interest are the internals: The functions that are not in the original JavaScript files, such as builtins, like Math.div() or array.push(). These functions are an ideal candidate for compilation or inlining, since they are generally small and called often.

Another important part of the SourceSectionFilter is the Roots filter. Enabling the Roots filter causes the CallTracer to look at every AST node that contains a rootTag. The rootTag denotes a node which, in the case of JavaScript, is the root of a function. Figure 5.2 shows the nodes of a simple AST with tags. The CallTracer decides on when a call is made by looking at the rootTag. The idea behind this approach is that whenever a new rootTag appears in the AST it must have been called from somewhere, and thus a call can be recorded.

5.2 CallTracer

The CallTracer class performs the profiling of the AST. ResetTracer() is the entry point for the CallTracer class. It check whether there is an available SourceSectionfilter and enables the

ExecutionEventFactory. This ExecutionEventFactory listens to the nodes from the AST that correspond to the SourceSectionfilter. Whenever the ExecutionEventFactory finds a corresponding node it creates an ExecutionEventNode. This ExecutionEventNode represents an event that needs to be profiled. Additional information that is connected to this node can be found in the context, as can be seen in figure 5.2. For each ExecutionEventNode the information that is useful to the profiling is stored in a Payload Object by getCounter().

Algorithm 1 ResetTracer()

```
Ensure: The tool is still enabled
    Assign SourceSectionfilter
    if no filter is present then
        Retrieve filter from SourceSectionfilter
    end if
    for each Node that corresponds to the SourceSectionfilter do
        payload = getCounter(context)
        CounterNode(payload)
    end for
```

GetCounter() uses the context to retrieve the sourceSection and the rootNode of the ExcecutionEventNode. These two variables are needed to uniquely identify each function and form a hashKey. A new Payload object is created based on the current node and its context. This Payload object stores information the instrumenter has access to such as the rootNode, the sourceSection, and tags. This Payload object is then saved in a hash table payloadMap, using the previously created hashKey and finally the payload is returned.

Algorithm 2 getCounter(context)

```
    Get sourceSection
    Get rootNode
    if RootNode $\neq$ null & sourceSection $\neq$ null then
        Hash(RootNode,sourceSection)
    else
        Hash(rootNode)
    end if
    Assert sourceSection $\neq$ null
    payload = Payload(location,context)
    Add payload to payloadMap(hash,payload)
    Return payload
```

The returned payload is then used to create a new CounterNode object. The CounterNode object stores all necessary information to create a call. When a function is completed it registers the call and sends it for further processing by CallTracerCLI. The reason it registers a call when a function is finished instead of entered is to be able to retrieve the return types, which would be unavailable at the start of a function, because of the dynamic typing.

Counternode notes the first time any object is returned in the whole program. Then, after a warmup time has passed, it starts collecting calls. First, a new Key object is created. Then, from the previous frame on the stack, the callNode is retrieved.

A callNode represents a call from a callsite to a callTarget. For the previous frame, the callTarget is the current payload and the callSite is the function where the call originated. Then the following variables are always gathered and used as key for an edge:

1. **SourceRootNode and targetRootNode:** These are qualified names that should represent a function uniquely. There may be cases where a qualified name is not available and thus the rootNode may not be unique to a specific function.

2. **SourceSourceSection and targetSourceSection:** The sourceSection represents a piece of source code. These are the main identifiers for a function or callSite location. It may happen that a sourceSection is not unique or not available.

Then, if more information needs to be gathered, such as the exact location of the callSite, the following options may be enabled and added to the key.

1. **Arguments:** Arguments play a key role in researching homomorphisation of functions. Splitting decisions can be based on argument types in a function call and provide more inlining opportunities. Arguments can also identify uninlineable functions by scanning for indirect calls.
2. **Return:** Return types may provide additional compilation strategies. Examples being that a boolean return might be easily optimised by a compiler. It may also be used to gauge function compute cost, as for example, returning a float suggests more compute expense than a string.
3. **Object instances:** An object instance is a Javascript specific option that may be used to track objects through the execution of a program. This may be useful for debugging purposes.
4. **Specific source location:** Describes the exact location of where a call is made. This option is important for making inlining decisions. As inlining is an action that is specific to a single call location instead of the whole source function.

Furthermore, the rootSize and targetSize values may be gathered, but not used as identifiers. These values provide an estimation of how difficult it is to compile a function. Every unique Key object is then added to a keyMap.

Finally, for each key in the keyMap, the number of times this key occurs during execution is saved in the countMap. This countMap is where the final call counts are retrieved from.

Algorithm 3 CounterNode(payload)

```
procedure onReturnValue(frame,result)
    if FirstIteration then
        InitialTime = time
    end if
    if (currentTime-initialTime)>warmupTime then
        Create new key Object
        node = getCallNode()
        while node == null do
            Iterate through Frames on the stack
            if node $\neq$ null then
                key.sourceRootNode = node.getRootnode
                key.targetRootNode = payload.getRootnode
                key.sourceSourceSection = node.getSourceSection
                key.targetSourceSection = payload.getRootNode.getSourceSection
                if Any options then
                    key.option = payload.getOptions
                end if
                Add key to keyMap
                Increase count for current key in countMap
            end if
        end while
    end if
end procedure
```

5.3 CalltracerCLI

In the CallTracerCLI class the results of the instrument are processed and exported. Three output foormats are available: JSON, CSV, and a command line interface printout (CLI). In this project CLI was created as a debugging tool. It prints the calls and their counts in the terminal. CSV was used for exporting the call graphs, as it is easily imported into Neo4j. The JSON format has some advantages in processing, since Neo4j returns query results by default in JSON.

The CSV is created by looping through all the keys in the keyMap and creating an edge for each unique key. Using this key, the execution counts are then retrieved from the countMap. The delimiter for the CSV format is ",". It might occur that in some rootNode names, source code, and lists, a comma character "," appears. Thus it is important to strip all "," characters. These are replaced with blank spaces. Algorithm 4 is an example of how a simple Edge List in CSV format looks like.

Algorithm 4 Example edge list

```
SourceLocation,TargetLocation,Count,rootSize,targetSize,sourceNode,targetNode,CallLocation,
acorn.js63-63:24-1, acorn.js64-64:19-1, 1633, 453, 42, pp3.parse, pp6.finish, acorn.js63:200-20,
acorn.js188-188:5-5, builtin1-1:1-1, 94, 109, 22, isKey, RegExptest, acorn.js188208:721-722,
```

5.4 Call graph validation

It is important for the call graph to accurately represent the the execution of a benchmark. If the call graph contains wrong edges or wrong invocation counts, then the strategy that will be based on these numbers may perform worse. There are two methods used to validate the results to a reasonable degree of certainty.

The first method is unit testing. GraalVM provides a unit testing framework that can be used to test correctness of tools implemented with the instrumentation framework. The CallTracerTest is the test program that is created to ensure changes made during development do not change previously verified results.

Two simple JavaScript programs are used for unit testing. These programs are designed such that all options can be tested on them. Figure 5.4 shows the recursive unit test of CallTracerTest. The test contains calls with different arguments, different return types, and different call locations. In total 8 unit tests are performed on these two JavaScript test programs. These unit tests confirm that the number of edges, and the counts of these edges are correct for a variety of combinations of options.

The unit tests confirm that the graph is correct for simple JavaScript patterns. There might still be ways in which JavaScript applications can behave outside of these patterns. Especially for the chosen benchmarks it is not feasible to completely validate if every edge in the graph is profiled

```
function sum(x){
  return x;
}
function calc(x,y){
  if(y>5){return;}
  for (y=y;y<5;y++) {calc(x)};
  for (var i=0;i<10;i++) {sum(x)};
}
function main(){
  calc("hello",0);
  calc(5,0);
}
main();
```

Figure 5.4: Example JavaScript unit test.

and whether every profiled edge is present as a call in the source code. The second approach to validation is to manually inspect parts of the profiled edge list. This is done on simple JavaScript benchmarks and on a selection of the benchmarks. By selecting a random sample of edges from the EdgeList and then checking in the source code whether these edges exist, it can be confirmed with reasonable certainty that all edges in the EdgeList also exist as calls in the source code.

Chapter 6

Using a graph database to analyze a call graph

Now that the JavaScript application is represented as a call graph information can be gathered from it. This information can then later be used to improve the virtual machine. The information is currently in a CSV format as an EdgeList. In order to query on this EdgeList first the CSV needs to be imported into a graph database.

Neo4j community version 4.3.7 [27] is used as the graph database. Neo4j provides a fast way to import and query on large graphs. It also has a tool that enables visualisation of graphs: Neo4j Bloom. These graph visualisations assist in finding mistakes in the call graph and can thus help with verification of correctness. Furthermore, the visualisations provide a way to gain more insight on what is happening in a call graph with respect to hot functions or possibly inlineable structures.

The configuration options for Neo4j can be found in the Github repository [46].

The language used to communicate with Neo4j is called Cypher [28]. Algorithm 5 shows the Cypher command used to import the EdgeList into Neo4j.

First, each row in the EdgeList is loaded into the database. Then, nodes are created with the primary key: SourceNode, SourceLocation or targetNode, targetLocation. sourceNode and targetNode are qualified names and should uniquely identify the nodes. SourceLocation and targetLocation are the sourceSection locations. A pair of these variables uniquely tie each node in the graph database to a function in the JavaScript application.

These nodes then need to be connected to each other with edges. This is done by again scanning through all the rows in the database. For each row in the database the corresponding source and target node are found and the call characteristics are added to the call edge.

Algorithm 5 Cypher command for importing data into Neo4j.

```
LOAD CSV WITH HEADERS FROM file:///bench.csv AS row FIELDTERMINATOR ',',
MERGE (a:Node id:row.sourceNode,location:row.SourceLocation)
MERGE (b:Node id:row.targetNode,location:row.TargetLocation)
CREATE (a)−[r:Call]− >(b)
SET r = row,
r.Count = toInteger(row.Count),
r.targetSize = toInteger(row.targetSize),
r.sourceNode = row.sourceNode,
r.targetNode = row.targetNode
r.SourceLocation = row.SourceLocation,
r.TargetLocation = row.TargetLocation,
r.CallLocation = row.CallLocation;
```

Now that the whole call graph is loaded into Neo4j the only variables in the nodes are the identifiers. Some additional variable are added to the nodes that assist in performing queries and help in visualisation.

The first Cypher command adds the total number of incoming calls to each function. This is done by summing over all counts of the incoming edges. This command also adds the estimated compilation time of the target function to the node.

The second Cypher command adds the distinct number of callSite locations to each node. This call site location is a crucial variable for determining inlining decisions.

Algorithm 6 Additional Cypher commands.

```
MATCH ()−[r:Call]− >(b:Node)
WITH SUM(r.Count) as sum,b, AVG(r.targetSize) as size
SET b.inc = sum,
b.size = size;

MATCH ()−[r:Call]− >(b:Node)
WITH b, count(DISTINCT r.CallLocation) as distinct
SET b.distinct = distinct;
```

Now that the graph database is ready, queries can be performed to extract information that might be useful to the virtual machine. One of the key indicators of how much performance is gained by compiling a function is the number of times it is executed. A minimum for how often a function is executed is the number of times the function is invoked. This is the main measure on which the proposed compilation strategies are based.

Algorithm 7 shows the query that returns the number of times each function is called. First, all nodes are gathered that have an incoming call. Then, for each of these nodes, all the counts of the incoming calls are combined. This results in the total number of times a function is called and thus executed. The final variable that is also returned is the average estimated compilation time.

This is because very large functions may help improve performance, but in the time it takes to compile time them, many other functions could have been compiled. This is the same reason that the default strategy prioritizes tier 1 over tier 2 compilations. The goal of this variable is to investigate whether compiling many smaller functions is better than compiling fewer larger functions. The resulting query returns for each function its number of incoming calls and its estimated compilation time.

Algorithm 7 Query for determining compilation strategies.

```
MATCH ()−[r:Call]− >(b)
WITH SUM(r.Count) as count, r.TargetLocation as id, r.targetNode as TargetName,
AVG(r.targetSize) as size
RETURN count,id,TargetName,size;
```

Besides using the call graph to improve the compilation strategy it can also be used to improve virtual machine performance by looking at interprocedural optimisations. Query 8 is used for gathering a list of potential inlining candidates. The inlining strategy is based on the idea that inlining small callTargets increases the size of the callSite function and thus provides additional opportunities for optimisations. One important note is that these callTarget functions should not be too large as this may complicate later compilation of the callSite function. When inlining a function the first tier of compilation is also performed. When an inlined function still has an outgoing call it may happen that the callTarget of this outgoing call is inlined into the already compiled callSite function. In this case the earlier inlined function must first be deoptimised, wasting the earlier compilation time. To avoid this it makes sense to only look at leaf nodes, where there are no outgoing calls.

The query works by first looking at all nodes. Nodes are selected that have not outgoing calls, that have only 1 distinct incoming call, and that have a size of below 30. Only having 1 distinct incoming call is a requirement for inlining to be possible. The functions are then ordered on the number of incoming calls and returned. The incoming calls are necessary for sorting on the most impactful leaves.

Algorithm 8 Query for determining leaf inlining strategy.

```
MATCH (a)
WHERE NOT (a)−[ ]− >() AND a.distinct < 2 + AND a.size < 30
RETURN DISTINCT a.id, a.inc,a.location
ORDER BY a.inc DESC
```

Now that the query results have been returned they need to be processed so that they are in a format usable by the virtual machine. The lists are reordered based on the chosen strategy. For graph based compilation there are three strategies that use the information provided by the call graph:

1. **Linear size penalty:** The linear size penalty first takes the total number of incoming calls for each function. It then divides this number of incoming calls by the compilation difficulty estimate. Finally, these are then ordered on this new weight and send to the compilation queue in this order.
2. **Square root size penalty:** The square root size penalty uses the same approach as the linear penalty but divides the number of incoming calls by the square root of the size of the function. This results in a less severe penalty that possibly causes larger but more impactful functions to be prioritised.
3. **no size penalty:** The no size penalty strategy does not use the difficulty approximation of the compilations. By only sorting on the incoming calls, the most impactful functions are compiled first. These functions may be very large and hard to compile, causing less total compilations to occur.

In order to execute these strategies the query results need to be reordered based on size and incoming calls. The formula used is:

$$order = count * 1/(size^x)$$

Where *count* is the number of incoming calls, *size* is the estimated compilation time of the target function, and x is a variable that controls the penalty severity of size. The variable x is set to 0 for the no size strategy, 0,5 for the square root strategy, and 1 for the linear strategy. The target functions are then ordered on the *order* variable.

Now there is a list of functions sorted on the importance of compiling them. If this list is directly transferred into the compilation queue in the current order than the compilation strategies can be tested.

This is done by using the fact that the iterations are deterministic and thus the same every time. The virtual machine already keeps track of the number of times a function is called with the callCount variable discussed earlier. The default strategy then uses a threshold to determine when the a function may be entered into the compilation queue. This compilation threshold is the same for every function and thus functions entering the compilation queue is entirely dependent on the callCount of the function.

By making this compilation threshold unique for every function functions can be given more or less priority in being submitted to the queue. Furthermore, since the callCount is deterministic, it can be fairly accurately controlled when a function is added to the compilation queue.

Figure 6.1 shows an example of how the list looks after it is returned from the graph database query and ordered. When setting, for each function individually, the compilation threshold to the callCount all functions will be executed during the first iteration. However it cannot be decided exactly in which order, since there is no knowledge of when exactly during the benchmark these functions were executed. Thus, in this example the compilation queue ordering is completely random.

This can be solved by multiplying the callCount by a larger factor for each successive function down the list. Figure 6.2 is an example of this technique, where the queried callCount of someFunction2() is multiplied by 2 and the callCount of someFunction3() by 3 to produce the compilationThresholds. Since the results are deterministic someFunction2() will be called 3000 times every iteration. Setting the threshold to 6000 causes it to be put in the compilation queue sometime in the second iteration. SomeFunction() will still be entered into the queue during the first iteration. Thus, with a First In First Out queue, the compilation queue in the virtual machine is now in the correct ordering.

The InputList for the virtual machine is thus a list of compilations thresholds for each function.

Function names	compilationThreshold
someFunction()	7000
someFunction2()	3000
someFunction3()	800

Figure 6.1: Example InputList for virtual machine, where the query results are directly used as compilationThresholds.

Function names	compilationThreshold
someFunction()	7000
someFunction2()	6000
someFunction3()	2400

Figure 6.2: Example InputList for virtual machine, where the query results are multiplied by some factor K to result in the compilationThreshold. This ensures that the functions are entered into the compilation queue in the correct order.

Even though sending 1 function to the compiler each iteration will provide the exact desired compilation queue, it will also cause idle time for the compiler, since the compilation queue will often be empty.

The variable entriesPerIteration determines how many functions are sent to the compilation queue each iteration. An example entriesPerIteration = 50 indicates that each iteration 50 functions are sent to the compilation queue. Since these iterations are deterministic, the first 50 functions should be sent in the first iteration and the second 50 functions in the second, so there is a correct ordering between these two batches of functions. Within these batches of 50 functions it could be possible that the ordering is not exact. An example is that if a function's executions all appear in the beginning of an iteration, it will also be sent to the compilation queue early in the iteration. This happens even if it is ordered lower on the list.

This problem can be solved by decreasing the number of functions sent to the compilation queue each iteration. Such that, for example, only 20 functions have a non exact ordering each iteration. This however, causes the compiler to idle or compile tier 2 function, which is not good for performance. 50 entries per iteration was chosen as a good compromise between idle time and ordering precision. Other complications arise when considering function splits and dequeues. These also have an effect on queue length. To avoid complications, entriesPerIteration was fixed at 50. The multiplication by 3 is necessary because the call graph and thus the callCount is based on three iterations of a benchmark.

Algorithm 9 Input List Creation

```
K = 0
for x in output query list do
    Add to InputList: callCount*K +sourceSection location + rootName
    K=K+1/(entriesPerIteration*3)
end for
```

The virtual machine needs two variables to uniquely identify a function and give it a threshold: The rootNode and the sourceSection. The final variable needed is the callCountThreshold: The number of calls after which the virtual machine sends the function into the compilation queue. Algorithm 10 shows an example of the InputList that is sent to the virtual machine.

Algorithm 10 Example InputList passed to GraalVM

```
0@@typescript.js 553614-553616:9-9token
2375@@typescript.js 543323-543323:26-57getStartPos
3479@@<builtin>1-1:1-1Map.prototype.get
```

Chapter 7

Improving GraalVM performance

Improving the GraalVM performance is done by implementing the strategies discussed in previous section. The goal is to provide a better compilation queue to the compiler, such that early and late performance is improved.

This starts by importing the InputList into GraalVM. The InputList consists out of a key and a value. The key is the function identifier which consists out of the sourceSection location (sourceCallTarget) and qualified function name (rootNode). The value is the callCount multiplied by the priority factor K. This value will be used as a compilation queue threshold. The changes to the compiler are made in the OptimisedCallTarget class, which is invoked whenever a callTarget is executed by the virtual machine.

The InputList for the compilation queue and inlining are stored by GraalVM in two hash tables: precomputedCounts and precomputedCountsInlined. Two methods of the OptimisedCallTarget class have been changed to implement the before mentioned altered compilation strategies.

The first is the constructor optimisedCallTarget() in algorithm 11. It is excecuted whenever a function is called for the first time. The optimisedCallTarget() constructor sets initial values for each callTarget, uniquely identified by sourceCallTarget and rootNode. Especially important that here the compilation thresholds are initialised.

In the case that either flags USEGRAPH or INLINE are used the non default initialisation is enabled, which loads the results from the InputList. These initialised values are added to the callTarget as threshold later used in the shouldCompileImpl() method.

First, a locationdescriptor is constructed from the current callTarget, following the same template as the InputList. Then, this locationdescriptor is checked versus all locationdescriptors in the hash tables. If a locationdescriptor matches, then it is identified as a compile target by the query and the threshold for the compilation queue is set as aotCompilationCallCount or aotInlineCallCount.

Algorithm 11 optimizedcallTarget

```
procedure OPTIMIZEDCALLTARGET(sourceCallTarget,rootNode)
    if USEGRAPH || INLINE then sourceSection $\neq$ null
        locationdescriptor = benchName + location + rootNode
    else
        locationdescriptor = empty
    end if
    if precomputedCounts contains locationDescriptor then
        aotCompilationCallCount = precomputedCounts.get()
    else
        aotCompilationCallCount = Max integer
    end if
    if precomputedCountsInlined contains locationDescriptor then
        aotInlineCallCount = precomputedCountsInlined.get()
    else
        aotInlineCallCount = Max integer
    end if

    Use standard initialisation

end procedure
```

Now that for each function the compilation threshold is set the shouldCompileImpl() method will check when the required aotCompilationCallCount threshold has been reached in algorithm 12.

If inlining is enabled, the callNode is retrieved. The callNode represent a direct call from a callSite to a callTarget. Thus, if there are multiple calls from different callSites it will become null. If this callNode is not null, inlineable and not already forced to be inlined. Then, the callTarget will be force inlined into the callsite. This happens whenever it is picked up by the compiler from the compilation queue.

Furthermore, if the FORCEINLINE flag is turned on, the callTarget is immediately added to the compilation queue, so that the inlining happens as soon as possible. The earlier a function is inlined, the earlier optimisation can be performed on larger combined parts of the source code.

If USEGRAPH is enabled a check is made whether the callTarget has been executed enough compared to the aotCompilationCallCount threshold, which is set according to the InputList. The callTarget is increased each time the virtual machine interprets the function. After a callTarget has been tier 1 compiled a new threshold will be set for tier 2 compilation. This threshold is determined by the default strategy. Only tier 1 compilations are considered for the graph based strategies.

The default compilation strategy checks whether a function should be compiled based on two criteria: a callThresholdInInterpreter, that dynamically changes size based on the length of the compilation queue, and a callAndLoopThresholdInInterpreter, that looks for loops.

Algorithm 12 shouldCompileImpl

```
procedure shouldCompileImpl(intCallCount,intLoopCallCount)
    if INLINE then
        if intCallCount >= aotInlineCallCount then
            callNode = getSingleCallNode()
            if callNode $\neq$ null & callNode.isInlinable() & !callNode.isInliningForced() then
                Inline callNode
                if FORCEINLINE then
                    Add function to Compilation queue
                end if
            end if
        end if
    end if
    if USEGRAPH then
        if intCallCount >= aotCompilationCallCount then
            Add function to Compilation queue
        end if
    else
        if intCallCount >= callThresholdInInterpreter &
      intLoopCallCount >= scaledThreshold(callAndLoopThresholdInInterpreter) then
            Add function to compilation queue
        end if
    end if
end procedure
```

Chapter 8

Experimental evaluation

In this section first the experimental setup and benchmarks are discussed. Then, a comprehensive discussion of a selection of benchmarks is performed. For these benchmarks the performance results and other compilations statistics are discussed. These will be focused on the in section 6 discussed compilation difficulty penalty compilation strategies.

After the selected benchmarks, a general overview of all benchmarks is shown and a conclusion is made on the effectiveness of the compilation penalty strategies. Then, the leaf inlining strategy and its performance is discussed.

The final section will discus the threats to validity of the performance evaluation. Here, it is discussed how the methodology can have unexpected effects on the performance of benchmarks.

8.1 Experimental setup

All tests were conducted on a AMD Ryzen 5 5600x 6-core processor and 16GB of memory. The benchmarks were performed using the recommendations set out by LLVM [26]. Address space randomization is turned off and SMT is disabled. Then, 3 out of 6 cores were reserved for benchmarking using cpuset [25]: One for executing the Javascript benchmarks, one for compilation, and one for other overhead, such as Python.

To evaluate the performance of the new compilations strategies a custom fork of GraalVM was built using Java 17 [46]. A development version of GraalVM 22.0 was used to build the call graph and perform the benchmarks. Exact versioning of GraalJS and GraalVM can be found in the Github repository [45]. The following flags were enabled or disabled based on the specific tests performed. A selection of flags is discussed in more details. The full Bash command to run the benchmarks can be found in the Github repository.

1. **–vm.DcallTarget.useGraph:** This option enables the usage of information provided by the graph database. It also enables the improved compilation queue strategy instead of the default strategy.

2. **–vm.DcallTarget.inline:** This option enables the use of the leaf inlining strategy.

3. **–vm.DcallTarget.forceCompileInline** This option causes all the leaf functions in the inlining list to also immediately be submitted to the compilation queue. This forces leaves to be compiled earlier in the benchmark for better optimisation possibilities later.

4. **–engine.PriorityQueue:** This option enables the use of a priority number based on callAndLoopCount before a function is put into the compilation queue. If enabled, a function will not be placed into the queue in a First In First Out manner, but given a place in the queue based on its number of function invocations. If turned on together with useGraph, both compiler information and graph information are used to make a decision on placement. If turned off only the graph information is used for compilation queue ordering. This flag

can be enabled to use all information available and in some cases improves performance slightly. It is turned off in the benchmarks to better differentiate between the graph based and default strategies.

5. **–engine.TraversingCompilationQueue:** The traversing compilation queue is the default compilation queue of GraalVM. It reorders functions in the compilation queue based on how hot a function is. It should be turned off when enabling useGraph, as enabling it will cause reordering based on compiler information and disable the FIFO ordering. Another reason to disable it is that useGraph does not use dynamic compilation thresholds. This causes the compilation queue to grow indefinitely. Repeatedly scanning and reordering a large queue may cause performance overhead.

6. **–engine.CompilerThreads=1:** Only one thread is available for compilation by the compiler. This provides more stability in benchmarking and increases the importance of having good ordering, as the number of compilations that can be done per iteration are limited.

7. **–engine.TraceInlining:** Every inlining decision is traced. It provides information on how the altered inlining decisions effect the inlining decisions by the compiler.

8. **–engine.TraceCompilationDetails:** Every compilation and compilation queue entry is traced. This provides an overview of how the compilation queue changes over time and how much functions are compiled every second. Furthermore other compilation information such as tier, split and compilation time are given.

9. **–engine.CompilationStatistics:** General statistics on the performed compilations, such as number of deoptimised compilations and split decisions.

8.2 Benchmarks

14 JavaScript benchmarks were used as benchmarks for performance tests. The benchmarks can be found in the Github repository [47]. The benchmarks are provided by the Google V8 web tooling benchmark [11].

Table 8.2 shows the general characteristics of the call graphs of the benchmarks. The number of functions in the benchmarks vary from 490 to 2560. The number of functions are important as that is one of the factors in estimating how long it takes to go through all tier 1 compilations. As discussed in section 7, the graph based compilation strategies only make decisions on when interpreted functions are sent in for tier 1 compilation. Thus, once all tier 1 compilations have been completed, the graph information is no longer used. This causes the benchmarks with more functions to be more heavily influenced by the chosen compilation strategy.

The other factor in how long it takes for all tier 1 compilations to complete is the time it takes to perform each tier 1 compilations. The number of nodes in the AST of a function is used as a measure for compilation time. This measure is referred to as the size of the function. The difference between benchmarks is smaller here, with the lowest mean size of 61 and the largest 144.

The number of calls is the best available indicator for how long a benchmark will take without performing any compilation. The reason for this is that after every call a function must be executed. The range of these is very large, where the largest benchmark executes over 8 times more functions in 3 iterations than the smallest.

The number of edges and the number of inlineable leaves are interesting to look at for inlining decisions. A function is inlineable if it has only a single incoming caller. When the number of edges is close to the number of nodes, most will have only a single edge connected to them. If the difference between these is very high, then likely every node has multiple incoming edges and can thus not be inlined. Specifically for the leaf inlining strategy it is interesting to look at how many candidate leaves there are.

	Typescript	Acorn	Babel-minify	Babylon	Buble
Number of nodes	2560	608	1720	658	766
Number of edges	7049	1404	4554	1537	1882
Number of calls (millions)	16.4	50.3	18.5	31.4	14.4
Mean function size	108	85	93	92	97
Number of inline-able leaves	466	239	552	239	272
	Esprima	Jshint	Source-map	Prepack	Postcss
Number of nodes	490	620	512	1756	1325
Number of edges	1162	1304	831	6423	2509
Number of calls (millions)	39.2	12.6	44.2	19.7	24.8
Mean function size	95	99	61	120	70
Number of inline-able leaves	206	251	305	346	575
	Prettier	Terser	Chai	Espree	**Combined**
Number of nodes	886	1133	1061	654	980
Number of edges	2542	3574	8232	1331	1847
Number of calls (millions)	23.4	9.7	5.8	51.3	25.8
Mean function size	110	100	144	81	97
Number of inline-able leaves	298	331	187	324	328

Table 8.1: General information about call graphs of benchmarks.

8.3 Warm up and peak performance measurement

An important aspect to discuss about the benchmarks is the warm up time. Warm up times are crucial to get good benchmarking results out of the call graph based compilation strategies. This is because of the way the benchmarks are constructed. Each benchmark starts with a preparation phase wherein the benchmark is made ready for testing. In this preparation phase mostly unique functions are used. These are not used during the iterations of the benchmark.

In the default compilation strategy, leaving these functions in is not much of a problem. Because of the dynamic compilation threshold few of these functions end up in the compilation queue. The startup functions that are in the compilation queue will quickly be pushed towards the bottom by the traversing compilation queue, because they are not being called anymore after the initialisation phase. These functions are then only compiled when all other functions have already been compiled to tier 1. An example can be seen in figure 8.1, where the functions that are useful for performance improvement are still on top of the compilation queue and the startup functions that will not improve the performance are pushed to the bottom.

In the graph based strategy there is no dynamic threshold and thus all initialisation phase functions will be placed in the compilation queue. There is also no queue reordering, so the functions will not be sent to the bottom of the queue, but will be scattered throughout the queue. Whenever one of these functions is compiled no performance improvement is achieved, as the compiled function is not executed anymore. This would effect the performance of the graph based strategies much more than the default strategy. An example can be seen in figure 8.2, where the early iterations need to compile many startup functions that do not improve performance.

Function names	callCount
UsefullFunction()	7000
UsefullFunction()	3000
UsefullFunction()	800
UsefullFunction()	200
StartupFunction()	8000
StartupFunction()	4000
StartupFunction()	800

Figure 8.1: Example compilation queue for default compilation strategy.

Function names	callCount
StartupFunction()	8000
UsefullFunction()	7000
StartupFunction()	4000
UsefullFunction()	200
StartupFunction()	500
UsefullFunction()	30
StartupFunction()	10

Figure 8.2: Example compilation queue for graph based compilation strategy.

A solution to this is to use a warm up time when constructing the call graph, such that calls are only collected when the benchmark iterations have started. This ensures that none of the initialisation phase functions are in the graph database or InputList. Therefore, these initialisation phase functions will not be entered into the compilation queue of the graph based strategies.

A warm up time of 4 seconds was chosen for all benchmarks. This ensures that for no benchmark any of the preparation phase functions are present in the call graph. However, some smaller benchmarks require shorter preparation phase times than 4 seconds. For these benchmarks the start of the call graph would be cut off. To remedy this the first 3 iterations are profiled instead of only the first one. This ensures that all functions are represented in the call graph, but the call count of the first few functions is slightly underestimated.

A problem with this approach is that a warm up is not possible to do in the default compilation strategy, since the compilation queue decisions are decided just in time.

This creates a disadvantage for the default compilation strategy. The preparation phase functions are pushed to the bottom of the list, but will still eventually be compiled. This will happen after all useful functions have been tier 1 compiled, but before the useful tier 2 compilations can be performed. This would then delay these important tier 2 compilations, especially in benchmarks where all tier 1 compilations are performed quickly. It would not affect large benchmarks that still have tier 1 compilations in the queue at the end of the run.

Figure 8.3 shows the situation where the default compilations strategy must first compile preparation phase functions before it can start doing useful tier 2 compilations. Because of the dynamic compilation threshold, this should generally be a low number of functions. Figure 8.4 shows the advantage that the graph based strategy has. Since it does not need to compile any preparation phase functions it can start performing tier 2 compilations straight away, improving performance earlier.

Function names	callCount	Tier
UsefullFunction()	200	tier 1
StartupFunction()	8000	tier 1
StartupFunction()	4000	tier 1
StartupFunction()	800	tier 1
UsefullFunction()	7000	tier 2
UsefullFunction()	3000	tier 2
UsefullFunction()	800	tier 2

Figure 8.3: Example compilation queue for default compilation strategy after most tier 1 functions have been compiled.

Function names	callCount	Tier
UsefullFunction()	80	tier 1
UsefullFunction()	8000	tier 2
UsefullFunction()	3000	tier 2
UsefullFunction()	600	tier 2

Figure 8.4: Example compilation queue for graph based compilation strategy with warm up time after most tier 1 functions have been compiled .

8.4 Typescript

The Typescript benchmark performs the compilation of the Typescript language in JavaScript.

There are four compilation strategies tested for each benchmark:

- **Default compilation:** The default compilation strategy is the one GraalVM uses by default. No graph information is used. The traversing compilation queue, the priority queue, and the dynamic threshold are enabled. In these benchmarks the default inlining strategy is used

The other three strategies use the graph information provided by the graph database. The traversing compilations queue, the priority queue, and the dynamic threshold are disabled. In these benchmarks also the default inlining strategy is used.

- **Linear size penalty:** The linear size penalty first takes the total number of incoming calls for each function. It then divides this number of incoming calls by the compilation time estimate.

- **Square root size penalty:** The square root size penalty uses the same approach as the linear penalty but divides by the square root of the size of the function. This results in a less severe penalty that possibly causes larger but more impactful functions to be prioritised.

- **no size penalty:** The no size penalty strategy does not use the cost approximation of the functions. By only sorting on the incoming calls, the most impactful functions are compiled first. These functions may be very large and hard to compile, causing less total compilations to occur.

Besides only looking at the final performance, it is also interesting to take a closer look at the startup time of a benchmark. This is because after the most impactful compilations have been performed the performance of the different strategies tend to converge. This has however not yet happened at the early iterations and it is here that large differences between strategies are visible.

Figure 8.5 shows, for the first 50 iterations of the Typescript benchmark, the number of iterations per second that are being performed. The opaque bands show the 95% confidence intervals.

Figure 8.5 shows that the no size penalty strategy performs significantly better than the other strategies for the first 50 iterations. With the linear and square root penalty performing worse respectively. This suggest that, for the TypeScript benchmark, it is more important to compile the most impactful functions, rather than to compile many but less impactful ones. Another thing to note is that the default compilation strategy is significantly worse than the graph based strategies. The confidence interval is also much wider than the graph based strategies. Since the graph based strategies receive the same ordered compilation queue for every benchmark run, the difference between benchmark runs is very small. For the default strategy, the compilations are entered into the queue slightly differently each time. This is partly due to the dynamic compilation thresholds. Furthermore, the traversing compilation queue reorders the queue every 1 millisecond. This small time frame causes even small variations in execution speed to change the compilation order.

Figure 8.6 shows the strategies for the full 500 iterations. The default compilation strategy does catch up with the graph based strategies, but the improvements plateau near the end. In this figure it also becomes clear that the three graph based strategies are finished with their 500 iterations earlier than the default strategy, but it is unclear which of the strategies is best and whether the difference between them is significant.

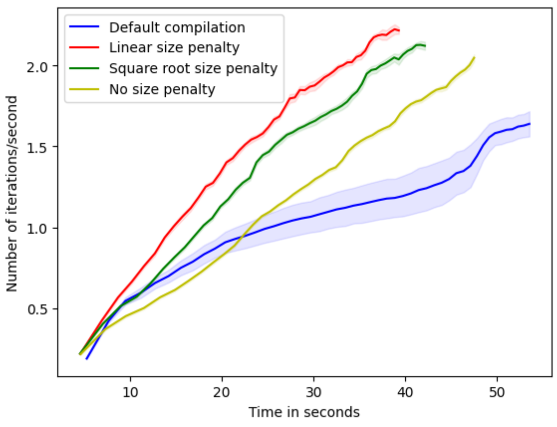


Figure 8.5: Number of iterations per second for the TypeScript benchmark. 50 Iterations per run. 20 runs per strategy. Higher is better.

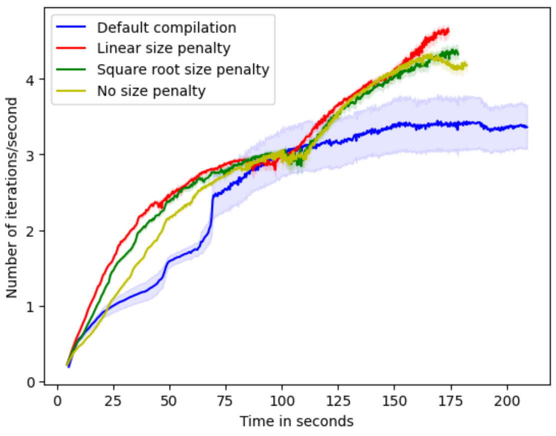


Figure 8.6: Number of iterations per second for the TypeScript benchmark. 500 Iterations per run. 20 runs per strategy. Higher is better.

Figure 8.7 shows the average time at which each iteration is finished for all 500 iterations. The darker the line gets, the more iterations are finished in that period of time. It shows more clearly that the default strategy takes longer to increase its performance early on, as there are less lines in that time period. This figure also shows that the default strategy is significantly slower than the graph based strategies. Furthermore, the no size penalty strategy performs slightly ,but significantly, better than the other graph based strategies.

Figure 8.8 shows the total number of compilations performed for each strategy. It is important to note that this is not the final number of compiled functions. Each successfully completed compilation is counted for both tier 1 and tier 2 compilation levels. It also might happen that functions get deoptimised. These deoptimisations are not taken into account. Again, the large confidence intervals of the default strategy stand out. One reason for the wide confidence intervals is the large total variance in how long a benchmark takes to be completed. Slower benchmarks have a longer time to perform compilations and thus will have compiled more at the end of the benchmark and vice-versa. A large range in total benchmark run time would also cause a large range in number of compilations, even if the compilation ordering is exactly the same. The fact that the no size penalty strategy has less compilations than the square root and linear strategies also makes sense as the linear strategy should penalise costly functions and thus perform more smaller compilations. However, due to the fact that the benchmarks is finished slightly earlier, it could also happen that it performs less compilations, as it had slightly less time to perform them. From this follows the expectation that the default strategy compiles more functions than the other three, but this is not the case.

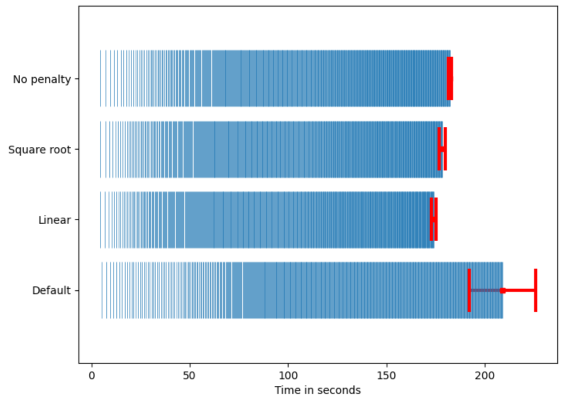


Figure 8.7: Eventplot of TypeScript. Average time each event takes place with 95% confidence intervals for last event. Denser is better typescript.

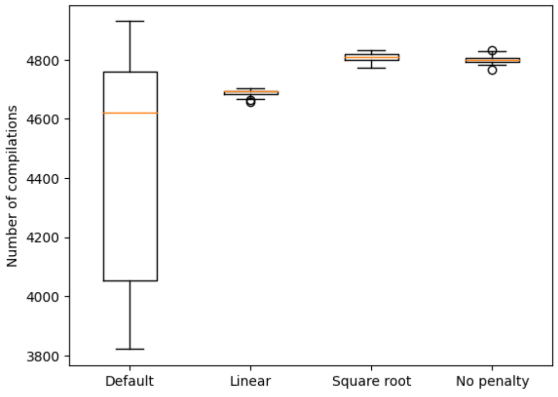


Figure 8.8: Boxplot of total number of compilations (tier 1 and tier 2) for TypeScript.

Figure 8.9 shows the number of tier 1 compilations that were performed as a fraction of the total number of compilations. The remaining compilations are thus tier 2 compilations, since interpreted functions are not measured here. This figure shows that the default strategy performs significantly more tier 2 compilations than the graph based strategies. This might be the reason that the default strategy does not perform more total compilations. These tier 2 compilations take much more time then tier 1 compilations and thus the total number of compilations will be lower if many tier 2 compilations are performed. Due to the priority rules, tier 2 compilations can only be performed when no tier 1 compilations are left in the compilation queue. However, 8.2 shows that Typescript is the benchmark with the largest number of functions and should not be running out of functions to perform tier 1 compilations on.

The reason this happens is the dynamic threshold of the default compilation strategy. While tier 1 compilations will still be performed first, tier 2 compilations are constantly added to the queue. At some point the number of tier 2 compilations queued will make the threshold too large for any tier 1 functions to be added and at this point tier 2 compilations will be performed, because there are no more tier 1 compilations left in the queue.

A similar trend can be seen between the three graph based strategies. Comparing the linear with the no size penalty strategy shows a significant difference between tier 2 compiled functions. In this case the strategies should perform more similar. Since every iteration 50 tier 1 compilations are being added to the compilation queue very little tier 2 compilations should be performed. The reason for this is the earlier discussed problem of determining how fast functions should be added to the compilation queue. The rate of 50 functions per iteration provides good performance later in the benchmark. Early in the benchmark, however, the compilation queue might run out of new tier 1 compilations to perform and switch to tier 2 compilations. This effect is worse in the linear size penalty strategy because the functions are being ordered on the compilation time estimate. Since these functions are compiled faster, the compilation queue will also become empty faster, thus causing more tier 2 compilations to occur.

Figure 8.10 divides the total benchmark time into 10 bins. In this case each bin represents about 25 seconds. In each bin the number of compilations in that time frame per strategy is displayed. This figure shows that the graph based compilations strategies are working as intended. The linear size penalty performs the most compilations early on, even though it is also still doing tier 2 compilations. In the first bins the no size penalty strategy performs the least compilations. Because there is no penalty, the size of these functions are larger and thus take longer to compile.

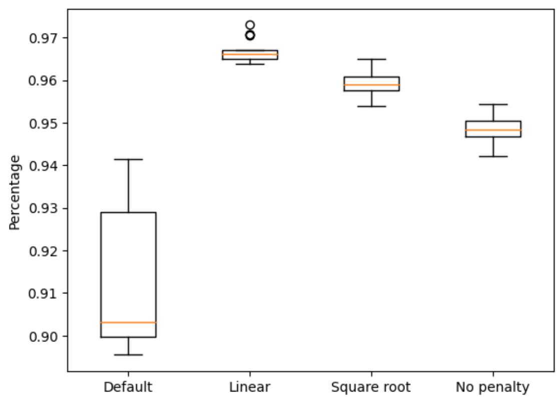


Figure 8.9: Boxplot of percentage of tier 1 versus tier 2 compilations for TypeScript.

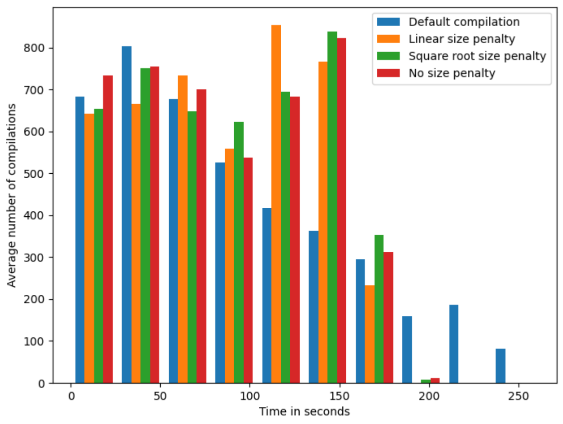


Figure 8.10: Number of compilations per time frame for TypeScript. Tier 1 and tier 2 compilations.

8.5 Terser

Terser is a tool that provides JavaScript compression.

Figure 8.11 shows that the default strategy outperforms the graph based strategies early on. Especially the linear strategy performs worse. This might suggest that there are some important functions early on in the benchmark that the linear strategy delays due to their size. Figure 8.12 shows a large drop in performance for the default strategy. The confidence intervals are not especially wide, so this is not a unique event, but happens at every benchmark run. These drops happen in multiple different benchmarks. One of the reasons for this might be that inlining and splitting optimisations may require the compiler to deoptimise earlier compiled functions. In the end the different strategies do tend to go to the same performance. This makes sense because, as the number of available tier 1 compilations are running out, the strategies have compiled the same functions. Tier 2 functions are not decided on by the graph based strategies, so these are compiled in a similar order as the default strategy, but without reordering and priority.

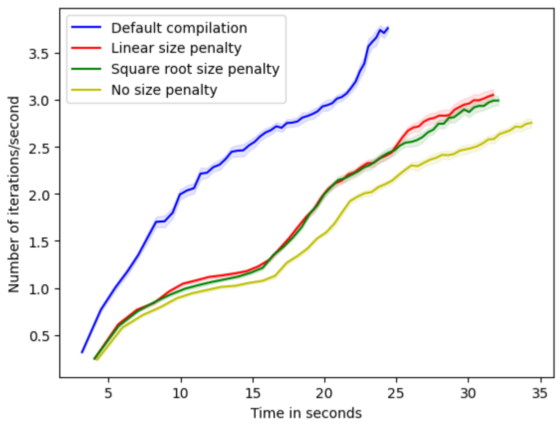


Figure 8.11: Number of iterations per second for the Terser benchmark. 50 Iterations per run. 20 runs per strategy. Higher is better.

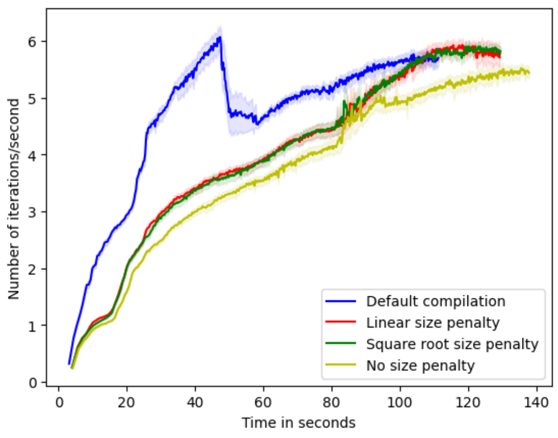


Figure 8.12: Number of iterations per second for the Terser benchmark. 500 Iterations per run. 20 runs per strategy. Higher is better

Figure 8.13 shows that in this benchmark it is better to take extra time to compile less larger functions, instead of compiling more smaller functions, as the linear strategy is significantly worse than all others. It is also clear that the default strategy performs much better than the graph based strategies.

Figure 8.14 shows that the default compilation strategy performs much less compilations then the graph based strategies. The three graph based strategies, however, have very similar number of compilations. This make senses, because when all tier 1 compilations are performed, the fact that the linear strategy performs more compilations early does not matter, as all strategies have the same total number of tier 1 compilations to perform.

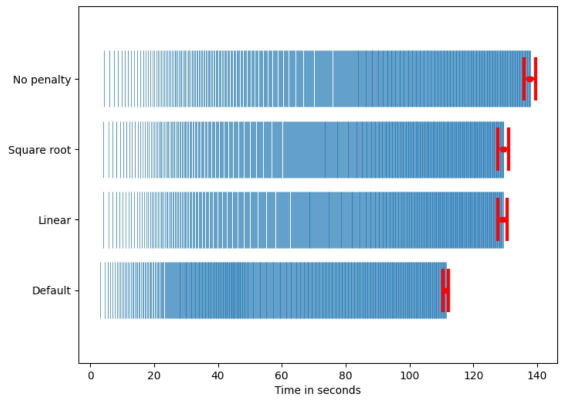


Figure 8.13: Eventplot of Terser. Average time each event takes place with 95% confidence intervals for last event.

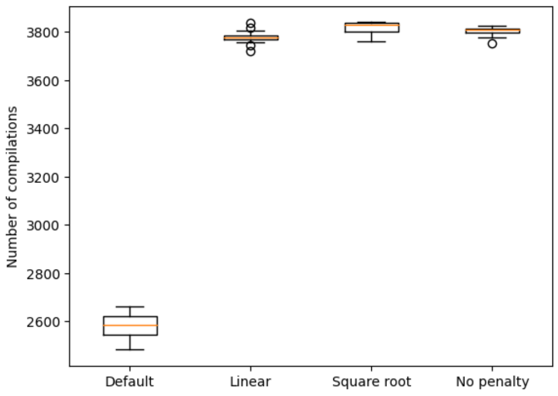


Figure 8.14: Boxplot of total number of compilations (tier 1 and tier 2) for Terser.

Figure 8.15 shows that the graph based strategies have fairly low percentage of tier 1 compilations. This suggests that indeed there were no more tier 1 compilations to be performed near the end and tier 2 compilations have started. Interestingly the default approach has performed more tier 1 compilations than the graph based strategies. This is unusual, as the graph based strategies sends all functions in the graph be compiled and thus it should sent at least as many functions to the compiler. One reason this may occur is the earlier discussed warm up time problem. During the preparation phase of the benchmark some functions will be added to the compilation queue. After the setup phase has ended these functions will not be called anymore and thus provide no optimisation. They are, however, still in the compilation queue and will have priority over any tier 2 compilations. This causes the default strategy to sometimes perform more tier 1 compilations than the graph based strategies. Another reason has to do with how splits are handled differently by both strategies.

Figure 8.16 shows that the difference in number of compilation in figure 8.14 is not only caused by it being finished with the benchmark earlier. Especially early on there are much less compilations performed. This suggests that there should be more tier 2 compilations performed by the default strategy. This furthers the idea that the preparations phase compilations are not the only cause of the extra tier 1 compilations.

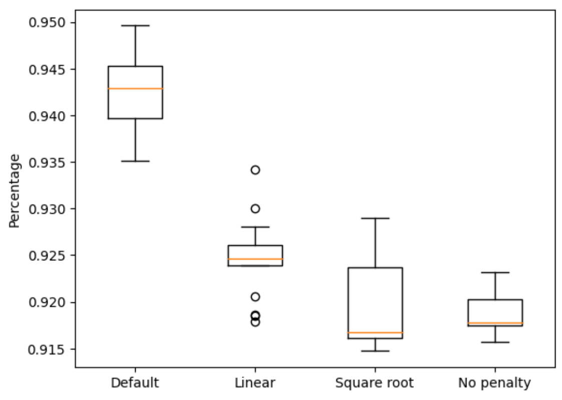


Figure 8.15: Boxplot of percentage of tier 1 versus tier 2 compilations for Terser.

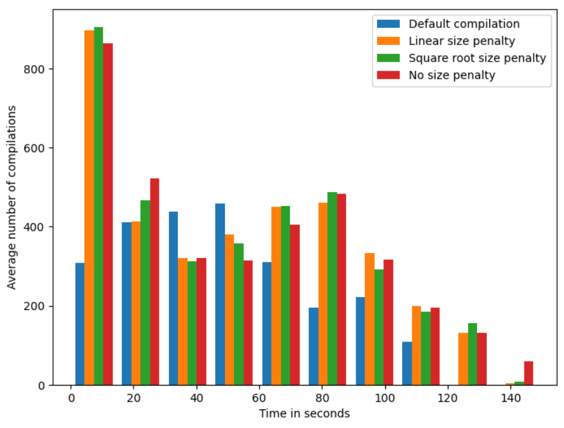


Figure 8.16: Number of compilations per time frame for Terser.

8.6 Prettier

Prettier is a tool that provides consistent formatting and styling for a JavaScript application.

Figure 8.17 shows that in the first few iterations the performance of all the strategies is very similar. The default strategy does increase its performance slightly above the rest. Figure 8.18 shows that indeed this performance difference is kept until the end of the benchmark, where the strategies converge to the same performance.

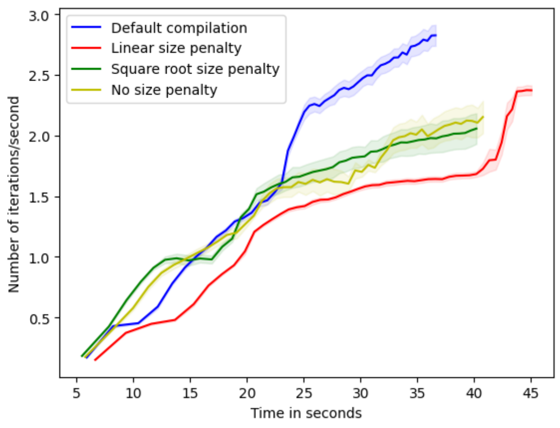


Figure 8.17: Number of iterations per second for the Prettier benchmark. 50 Iterations per run. 20 runs per strategy.

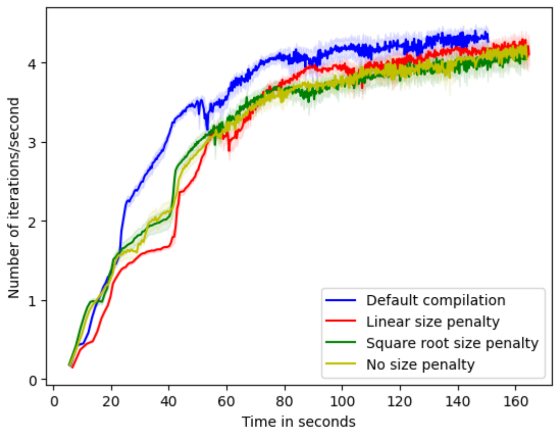


Figure 8.18: Number of iterations per second for the Prettier benchmark. 500 Iterations per run. 20 runs per strategy.

Figure 8.21 confirms that in this case the default strategy performs the best. Figure 8.22 shows that the no penalty strategy performs much more compilations then the others. It is expected to be the other way. The no penalty strategy should have less compilations since the most expensive compilations are performed in this strategy. The reason for this has to do with how splits are dealt with.

A split function inherits the compilation threshold that is given by the graph database to the original function. Figure 8.19 is an example where a splittable function has a low priority. The compiler splits this function and for each split separately keeps the callCount. If this priority is low, its threshold is high and will be hard to reach, since all incoming calls are divided amongst the splits. Thus only the splits with the highest incoming invocations will be sent to the compilation queue. Figure 8.20 shows an example of a situation such as the no penalty strategy in figure 8.22. The splittable function has a high priority and even though the callCounts are divided amongst the splits, still many splits are entered into the compilation queue. This results in more available functions for tier 1 compilation.

InputList from call graph	
Function name	callCountThreshold
someFunction1()	200
someFunction1()	800
SplittableFunction()	2000
Compilation queue	
Function name	Split
someFunction1()	None
someFunction2()	None
SplittableFunction()	split 1
SplittableFunction()	split 2

Figure 8.19: Example graph input and compilation queue in the case where a splittable function has low priority in compilation.

InputList from call graph	
Function name	callCountThreshold
SplittableFunction()	200
someFunction1()	800
someFunction2()	2000
Compilation queue	
Function name	Split
SplittableFunction()	split 1
SplittableFunction()	split 2
SplittableFunction()	split 3
SplittableFunction()	split 4
SplittableFunction()	split 5
SplittableFunction()	split 6
someFunction1()	None

Figure 8.20: Example graph input and compilation queue in the case where a splittable function has high priority in compilation.

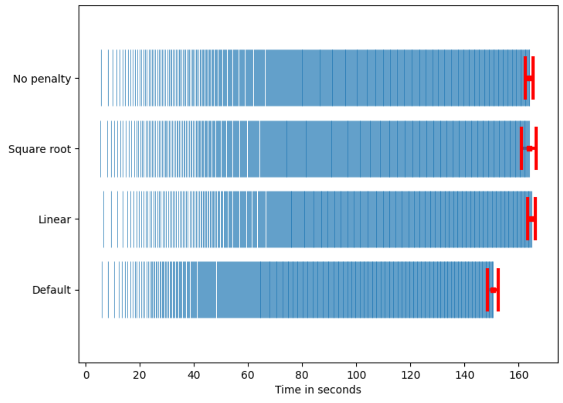


Figure 8.21: Eventplot of Prettier. Average time each event takes place with 95% confidence intervals for last event.

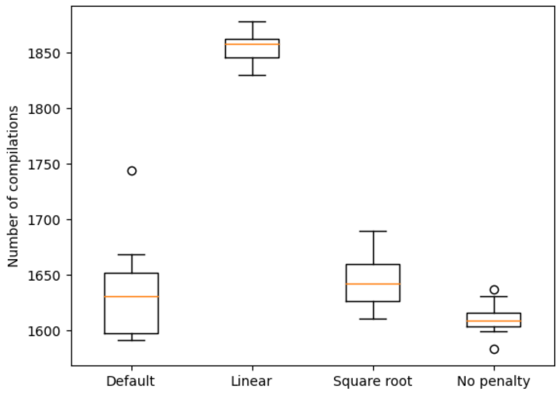


Figure 8.22: Boxplot of total number of compilations (tier 1 and tier 2) for Prettier.

Figure 8.24 shows that, especially in the start of the benchmark, the no-size penalty performs much more compilations. This strengthens the hypothesis that many splits of a small function are being performed.

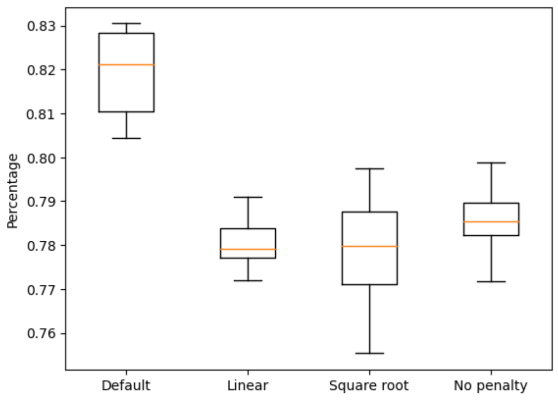


Figure 8.23: Boxplot of percentage of tier 1 versus tier 2 compilations.

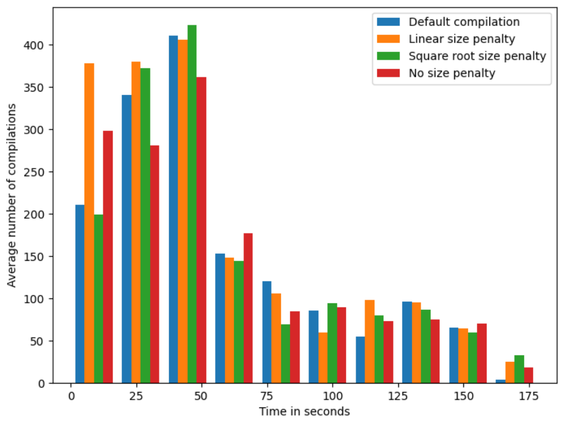


Figure 8.24: Number of compilations per time frame.

8.7 Espree

The third benchmark is Espree. Espree is an ECMAScript parser.

Figure 8.25 shows that, for the first iteration, both the linear and square root strategies perform significantly worse than the default and no size strategy. This suggests that some important larger functions were delayed by these strategies. It also shows that the default strategy has a very slow startup time the first few iteration until about 18 seconds.

Figure 8.26 shows that the default strategy very quickly catches up to the others, but the performance improvements also stalls again around the middle of the benchmark. The other strategies keep improving performance gradually, causing the benchmarks to be finished earlier.

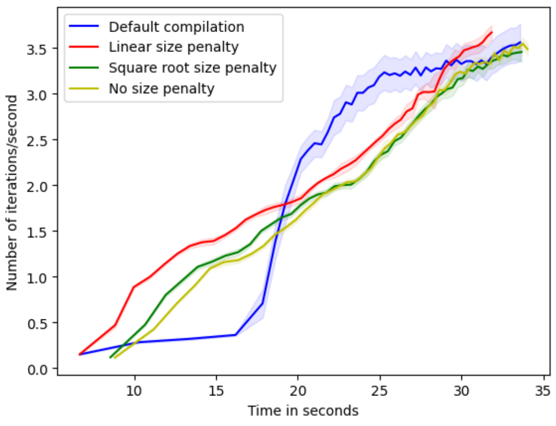


Figure 8.25: Number of iterations per second for the Prettier benchmark. 50 Iterations per run. 20 runs per strategy.

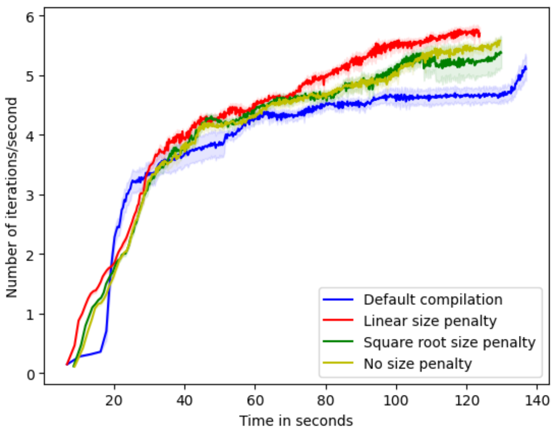


Figure 8.26: Number of iterations per second for the Prettier benchmark. 500 Iterations per run. 20 runs per strategy.

Figure 8.27 confirms that the no penalty strategy is significantly better than the others and that all three graph based strategies perform better than the default strategy. Figure 8.28 shows the no penalty strategy has the least performed compilations. The default strategy having more compilations can again be caused either by a higher number of performed splits, more warmup functions being performed, or because of the longer execution time. Figure 8.29 does show that many tier 2 compilations have been performed for all benchmarks and that thus the tier 1 compilations have run out. Figure 8.30 shows that the no size penalty strategy performs only half the compilations that the other graph based strategies do in the first bin. This shows that performing over double the compilations does not necessary improve the performance if these functions are not impactful. It again confirms that the penalty strategies perform their desired effect.

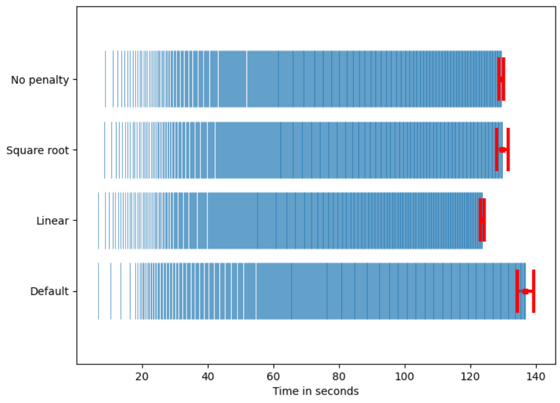


Figure 8.27: Eventplot of Espree. Average time each event takes place with 95% confidence intervals for last event.

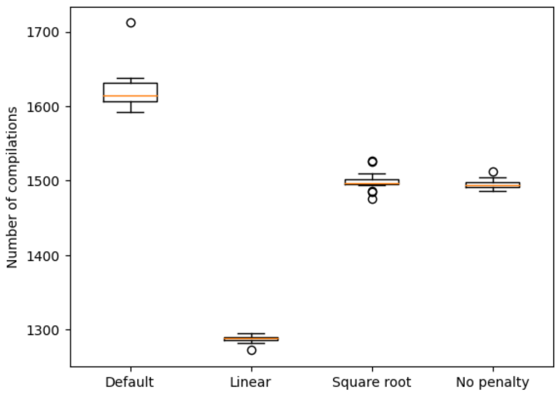


Figure 8.28: Boxplot of total number of compilations (tier 1 and tier 2) for Espree.

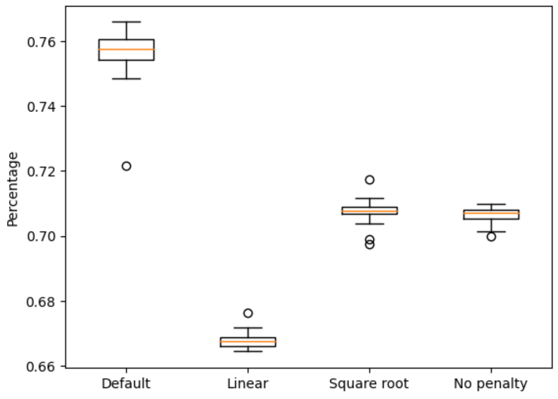


Figure 8.29: Boxplot of percentage of tier 1 versus tier 2 compilations for Espree.

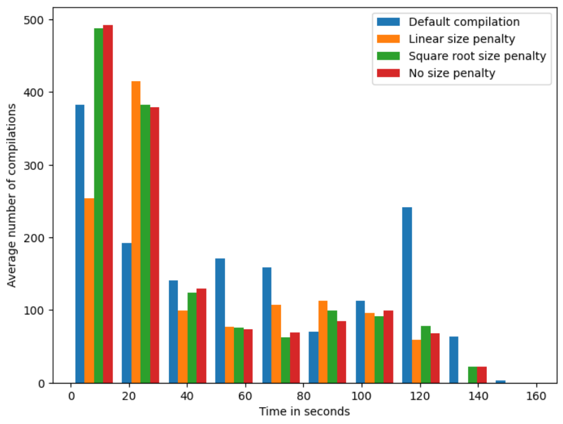


Figure 8.30: Number of compilations per time frame for Espree.

8.8 Babel-minify

Babel-minify is a minifyer for the Babel tool. The Babel tool is a transpiler for translating es6 to older JavaScript versions.

Figure 8.31 shows that all three strategies have very similar performance for the early iterations. The default strategy improves quickly after that and keeps outpacing the other strategies. At some point, the performance of all strategies converge. Interestingly, there is a sudden increase in performance at round the 75 second mark in figure 8.32 for the square root and no size penalty strategies. This increase occurs later in the linear penalty. This might suggest that there is a group of impactful functions that are fairly large. This group of functions is found by the no size strategy quite early, but later for the linear penalty. Another idea might be that it is a group of split functions that only reach a threshold much later in the graph based strategies.

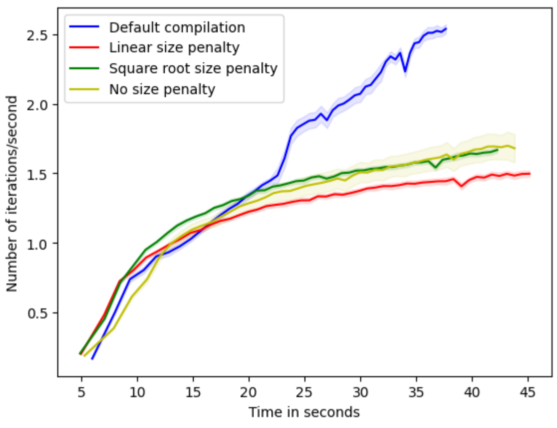


Figure 8.31: umber of iterations per second for the Babel-minify benchmark. 50 Iterations per run. 20 runs per strategy.

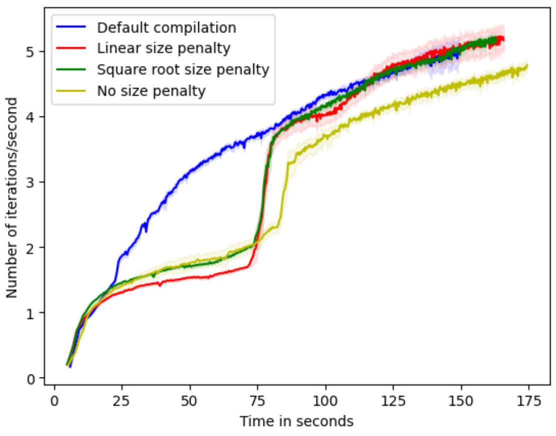


Figure 8.32: Number of iterations per second for the Babel-minify benchmark. 500 Iterations per run. 20 runs per strategy.

Figure 8.33 confirms that the default strategy is significantly better. It is also clear that the square root strategy outperforms the other two graph based strategies. Again, figure 8.34 shows the reverse of what is expected. The linear penalty should compile the most functions. Both because of the penalty, and because the it runs the longest.

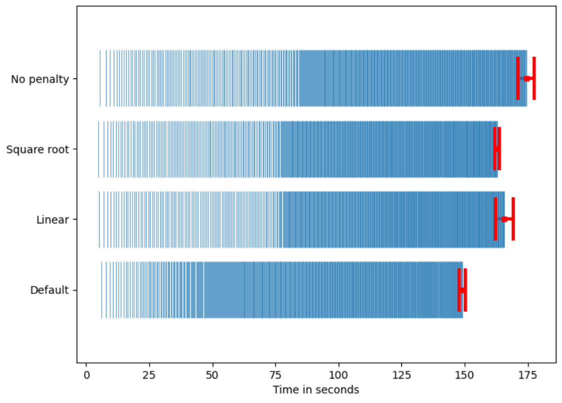


Figure 8.33: Eventplot of Babel-minify. Average time each event takes place with 95% confidence intervals for last event.

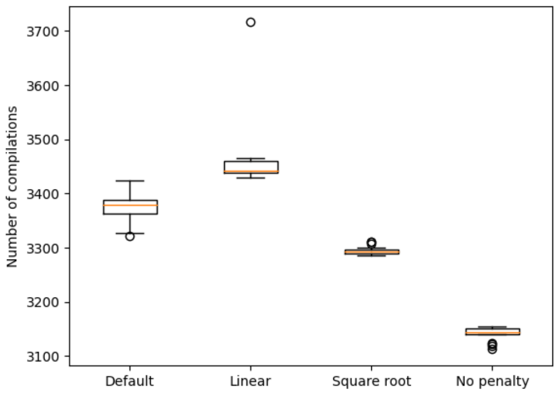


Figure 8.34: Boxplot of total number of compilations (tier 1 and tier 2) for Babel-minify.

Looking at figure 8.36 an explanation can be found. Similarly to previous benchmarks, the same trend is visible comparing figures 8.35 and 8.36. The linear penalty strategy performs more tier 2 compilations. this suggests that there are less available tier 1 functions. The reason might be that the linear strategy has more idle time early on, or that the other strategies perform more compilations of split functions.

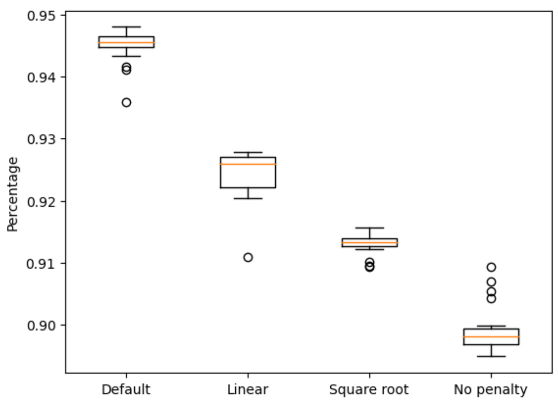


Figure 8.35: Boxplot of percentage of tier 1 versus tier 2 compilations for Babel-minify.

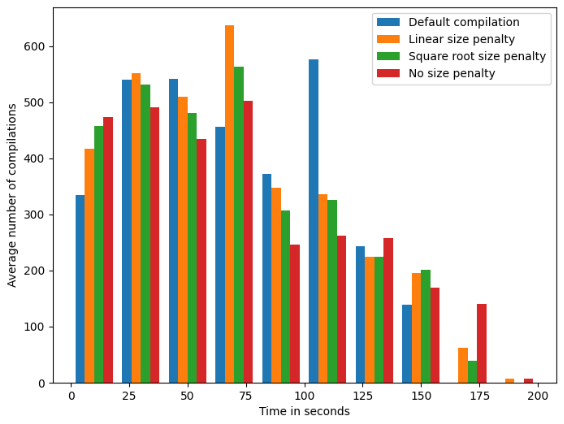


Figure 8.36: Number of compilations per time frame for Babel-minify.

8.9 Acorn.js

Acorn is a JavaScript parser that is also used by many other benchmarks in this thesis, such as Babylon and Babel.

Figure 8.37 shows that for the Acorn benchmark the default strategy has poor performance early on. Figure 8.38 shows that the benchmark barely improves its performance throughout the first 100 seconds. Only at the 125 second mark, when the graph based strategies have almost completed, it sharply increases its performance in only a few iterations. This suggests that there is a very small number of functions whose compilations provides most of the performance improvement. In the ideal scenario these functions could be identified in the very first iterations. The graph based strategies do find these gradually over a longer period of time, but not as sudden as the default strategy does.

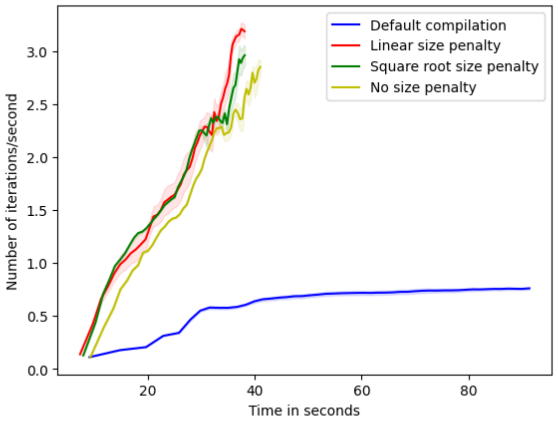


Figure 8.37: Number of iterations per second for the Acorn benchmark. 50 Iterations per run. 20 runs per strategy.

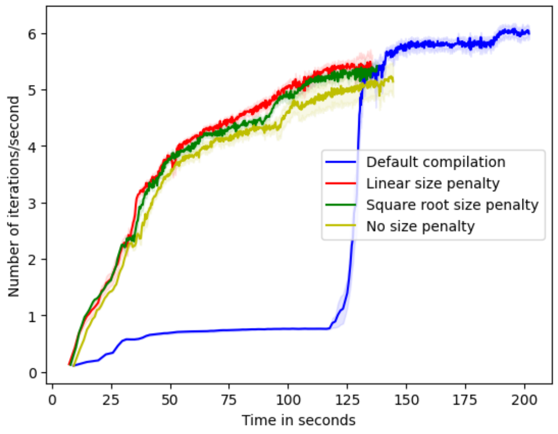


Figure 8.38: Number of iterations per second for the Acorn benchmark. 500 Iterations per run. 20 runs per strategy.

Figure 8.40 shows that the number of compilations performed by the default strategy is much higher. Figure 8.39 shows that this number can be explained by the fact that it runs longer and thus has more time to perform compilations. In the early iterations in figure 8.42 the default strategy performs a similar amount of compilations as the linear penalty strategy. It does however keep compiling when the other strategies have already finished their iterations and stopped compiling.

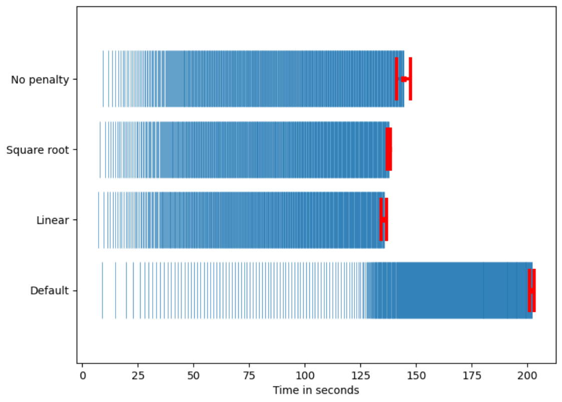


Figure 8.39: Eventplot of Acorn. Average time each event takes place with 95% confidence intervals for last event.

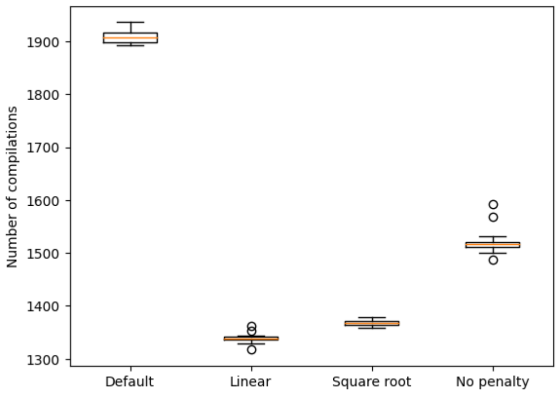


Figure 8.40: Boxplot of total number of compilations (tier 1 and tier 2) for Acorn.

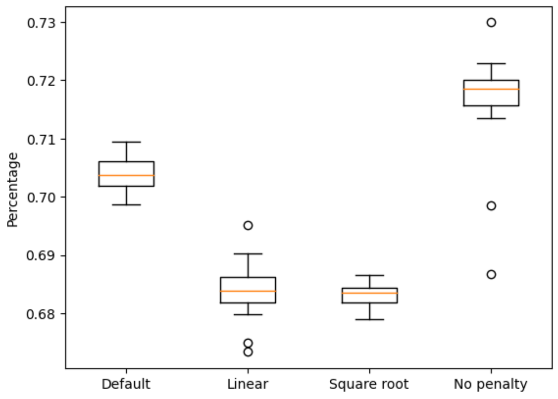


Figure 8.41: Boxplot of percentage of tier 1 versus tier 2 compilations for Acorn.

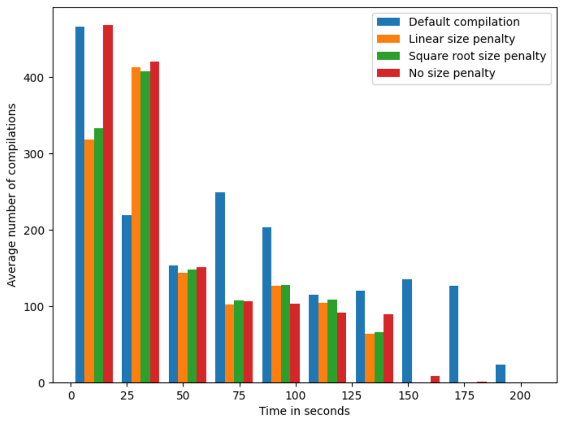


Figure 8.42: Number of compilations per time frame for Acorn.

8.10 Chai.js

Chai is a library that provides testing tools for JavaScript code.

Chai shows the opposite of Acorn. Figure 8.43 shows that the graph based strategies do not increase their performance much after 15 seconds, while the default strategy slowly improves. Figure 8.44 shows that from 15 seconds on the graph based strategies are barely improving their performance. The default strategy, however, already reaches its maximum performance in 25 second and does not increase at all after that. This suggests that there are many functions that are not useful to compile and there are a few that are extremely important.

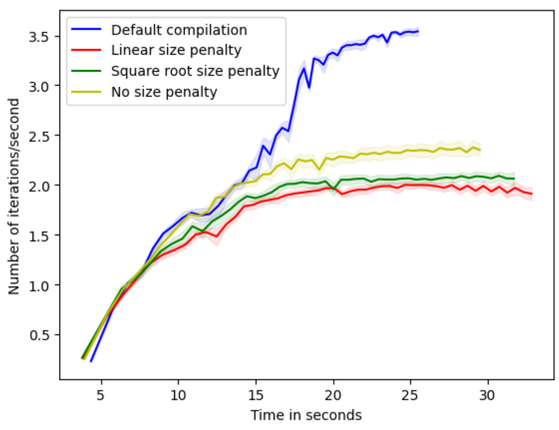


Figure 8.43: Number of iterations per second for the Chai benchmark. 50 Iterations per run. 20 runs per strategy.

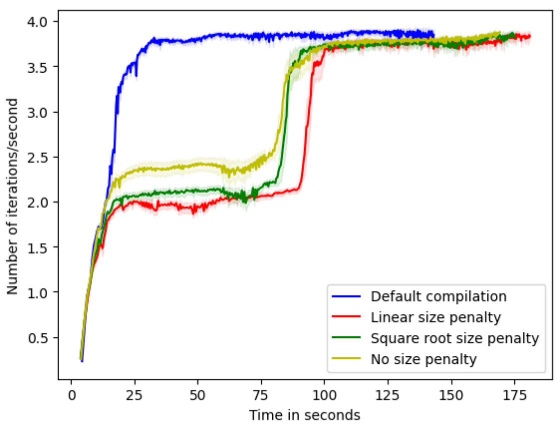


Figure 8.44: Number of iterations per second for the Chai benchmark. 500 Iterations per run. 20 runs per strategy.

Figure 8.45 shows that the default strategy indeed performs much better early and completes significantly earlier.

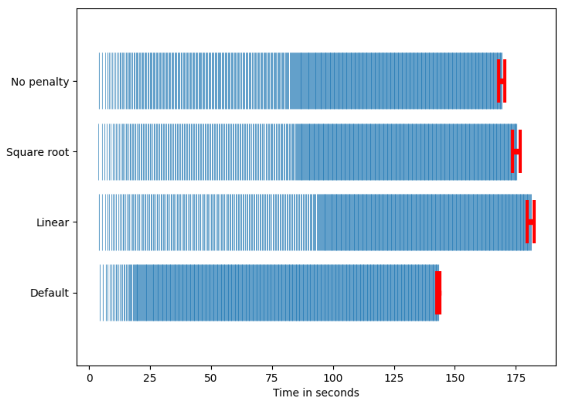


Figure 8.45: Eventplot of Chai. Average time each event takes place with 95% confidence intervals for last event.

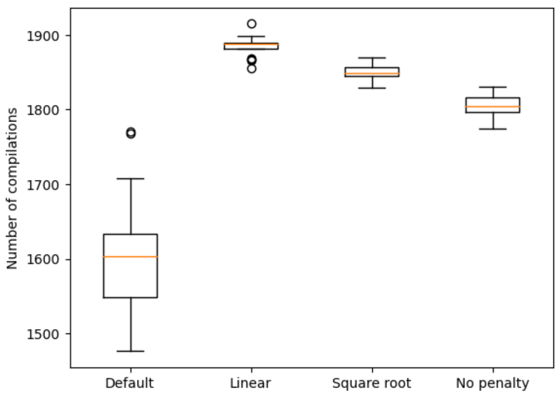


Figure 8.46: Boxplot of total number of compilations (tier 1 and tier 2) for Chai.

Figure 8.48 shows that the number of compilation over time changes much more than in previous benchmarks. At the 25 and 50 second bin the no size penalty has a sharp decrease in number of compilations performed. This suggests that very large functions are being compiled, either very large tier 1 or many tier 2 compilations. The fact that later the no size penalty starts compiling at a normal speed again suggests that these are not tier 2 compilations, as the tier 1 compilations performed later would have been prioritised. Thus, it is likely that there are some very expensive tier 1 compilations. It makes sense that this happens for the no size strategy first, as it does not penalise these large functions. At 75 seconds the same happens for the other three strategies. These times correspond with a period of very little performance improvement in figure 8.44. This suggests that these very large functions do not add much performance, so without them, the benchmarks might have increased performance sooner.

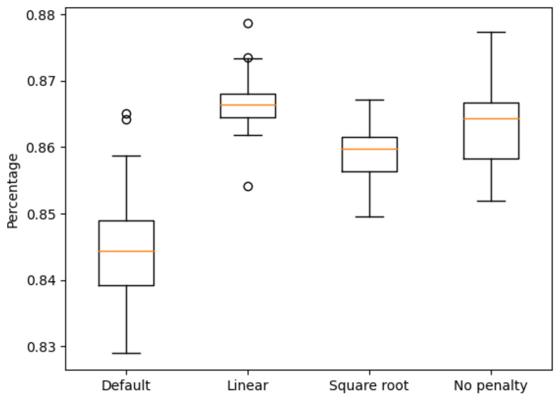


Figure 8.47: Boxplot of percentage of tier 1 versus tier 2 compilations for Chai.

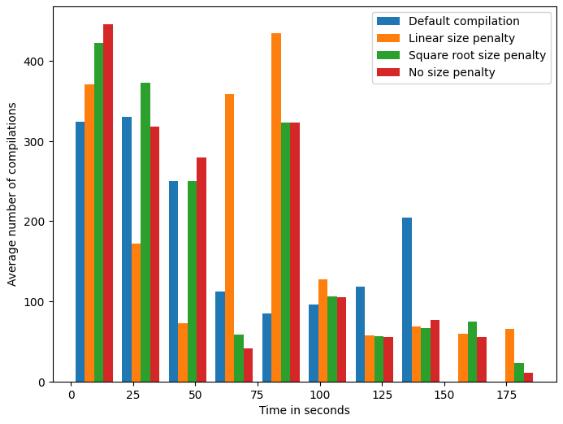


Figure 8.48: Number of compilations per time frame for Chai.

8.11 General Results

Looking at the benchmarks one by one provides ideas about what is going on within the compiler. The results are mixed and do not show a clear overview of the performance of the new compilation strategies. The different benchmarks also vary a lot in their size and duration, making it harder to compare them. In order to make a comparison between benchmarks this section shows the performance of the three graph based strategies as a percentage of the default strategy. This makes a comparison possible between the default and the graph based strategies, with the goal of definitively answering whether the proposed compilation strategies have a positive or negative effect on performance of the virtual machine.

	Typescript	Acorn	Babel-minify	Babylon	Buble	Esprima	Jshint	Source-map	Prepack	Postcss	Prettier	Terser	Chai	Espree	**Combined**
10 iterations Linear	15.9±4.1	87.2±11.0	5.7±2.6	76.1±8.0	-0.8±1.6	72.9±8.2	13.9±2.0	2.5±5.1	4.8±2.4	1.2±5.1	-12.1±1.0	-27.5±1.3	-2.9±1.9	32.6±2.4	19.2±17.7
50 iterations Linear	12.7±6.5	122.7±4.0	-14.1±3.0	106.9±3.6	4.8±2.0	152.0±1.3	6.1±2.6	-16.6±2.5	-4.1±4.8	19.5±8.6	-10.2±2.7	-29.0±0.9	-13.6±1.1	-1.3±3.7	24.0±29.3
500 iterations Linear	14.7±9.3	39.3±2.9	-14.5±1.6	41.0±1.2	7.0±6.1	27.6±0.8	-4.1±1.9	-20.8±1.1	9.5±6.1	17.5±4.1	-8.2±1.4	-19.2±1.2	-15.4±0.7	5.7±1.9	5.8±10.5
10 iterations Square Root	6.9±3.3	92.7±2.4	8.6±2.3	107.9±2.6	-2.7±1.6	124.0±5.8	13.2±2.8	1.3±4.4	-5.1±2.2	6.1±6.0	7.4±1.3	-29.0±1.3	-1.2±2.3	16.1±2.3	24.7±23.7
50 iterations Square Root	27.0±7.3	139.8±4.2	-10.9±1.5	113.3±3.3	-1.8±1.6	160.5±2.1	6.2±3.0	-16.4±2.5	-1.9±5.4	18.3±9.5	-8.9±2.6	-23.9±1.0	-19.6±1.7	-0.3±4.2	27.3±32.4
500 iterations Square Root	17.2±9.4	46.8±0.9	-8.5±1.1	44.2±1.0	8.4±5.7	29.5±1.2	-5.9±1.6	-22.5±1.5	10.3±6.0	14.6±4.0	-8.1±1.6	-14.0±1.1	-18.3±0.7	5.4±2.7	7.0±11.0
10 iterations No Size	-5.0±3.2	68.0±3.2	-0.5±2.1	92.9±3.5	-5.1±1.3	108.6±5.0	12.3±2.0	-1.6±4.3	-11.8±1.7	-1.8±4.6	4.4±1.9	-32.5±1.4	0.9±2.2	10.9±2.2	17.1±21.0
50 iterations No Size	36.2±8.2	140.0±6.5	-16.6±2.0	80.5±6.7	-1.5±1.6	139.8±2.9	-5.6±2.9	-9.0±3.0	11.0±6.2	10.6±8.1	-18.7±1.3	-23.1±1.1	-22.4±1.0	5.5±4.4	23.3±28.4
500 iterations No Size	20.1±9.9	49.1±1.3	-10.1±1.8	38.0±2.7	9.4±5.9	27.1±1.4	-7.5±1.5	-6.4±2.5	16.1±6.8	14.7±4.3	-8.6±1.1	-13.8±1.4	-20.9±0.6	10.6±2.0	8.4±10.5

Table 8.2: Performance statistics of graph based strategies as percentage of the default strategy.

Overall there are 5 benchmarks where both the startup and the total benchmark time of the graph based strategies was lower than the default strategy. There are also 5 benchmarks where the default strategy outperforms the graph based strategies in both situations. The remaining 4 have mixed results. Individually, nearly all results are significant with a 95 % confidence interval. The combined result of all benchmarks show a fairly large mean positive effect after a startup time of 50 iterations. The square root penalty strategy has the largest increase. It is 27% faster then the default strategy. The other two graph based strategies perform a bit worse. This might suggest that having no size penalty is problematic due to large functions being prioritised over more compilations and that a linear penalty might be too harsh and compiles more, but less impactful functions. The differences between the graph based strategies are fairly small. For the full 500 iterations the no size strategy performs the best overall, finishing the benchmark 8 % faster than the default strategy. Even though the number of positive and negative results is about equal, there is a definite difference in how much difference the positive and negative results measure. Especially in the first 50 iterations some of the benchmarks perform over twice as fast with the graph based strategies as opposed to the default strategy.

Figure 8.49 shows the combined performance of the strategies after 10 startup iterations. The linear strategy performs the best with the 50 % box of the benchmarks clearly positive. Table 8.2 also confirms that for the first 10 benchmarks the Linear and Square root strategies are significantly better than the default strategy.

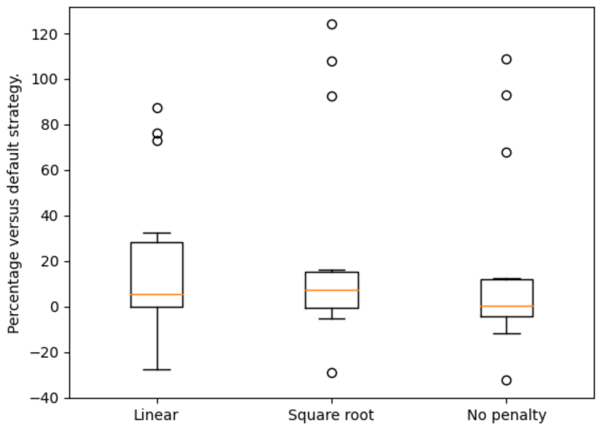


Figure 8.49: Boxplot of combination of all benchmarks for first 10 iterations as percentage of default strategy.

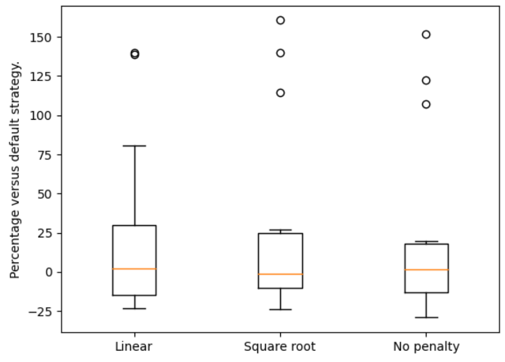


Figure 8.50: Boxplot of combination of all benchmarks for the first 50 iterations as percentage of default strategy.

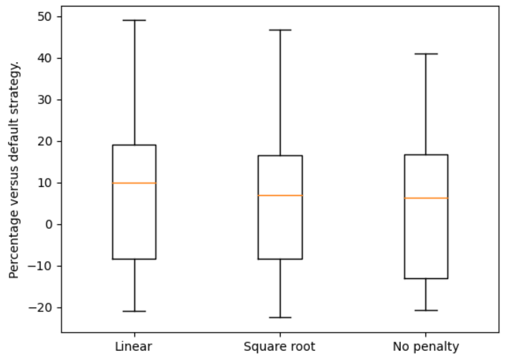


Figure 8.51: Boxplot of combination of all benchmarks for all 500 iterations as percentage of default strategy.

Figure 8.50 further shows this point. The median benchmark has close to 0 % improvement over the default strategy. The 50 % boxes are also largely centered around 0 with a slight improvement in regards to the default strategy. This figure shows that there are some very large positive outliers. These outliers suggest that, especially for the startup iterations, the graph based strategies have a very large performance advantage for some benchmarks.

Figure 8.51 shows that after 500 iterations the differences for these outliers have decreased. There is now a clearer positive effect visible. The median benchmark is positive. The 50 % boxes for the square root and no penalty strategies also suggest that there is a clear performance increase over the default strategy. The whiskers are at most negative 20%, but the positive whiskers go up to 40%.

8.12 Inlining

Besides making decisions on the compilation queue, the virtual machine also decides on interprocedural optimisations, such as inlining and splitting.

The leaf inlining strategy inlines leaf functions early on in the benchmark run in order to get a performance advantage later on. The idea is that early on time is spend on inlining a function that does not have the highest impact. But, as it is now inlined into its calSite function, later on, when the callSite function is being compiled, more optimisations might be performed. Furthermore, inlining functions have the additional performance gain of removing a call.

The expectation is thus that there might be a slight disadvantage early on, as less desirable functions are being compiled in front of functions with higher impact. This should, however, be small, as the leaf functions are chosen to have short compile times.

From the graph database all leaves are retrieved below the size of 30. This size of 30 represents only very small functions. This is important for two reasons. First, the inlining is performed at the start of the benchmark, even before the most called functions. If these functions are too large, and thus take too long to inline, it can delay the compilation of the more important functions by too much. The second reason is that functions which are very large can become quite expensive to compile. If a large function is inlined into another large function, this might cause performance problems.

Only leaves were chosen that have a direct call from a single call site. This ensures that the function may be inlined straight away, without having to wait for splitting decisions.

In order to ensure that the compilation of the functions, chosen by the graph based strategy, is not delayed by too long, the 50% leaves with the lowest incoming call count are discarded. This is a trade off that needs to be made between performance early and possible increased performance later on.

The no size strategy with default inlining was chosen as a baseline to compare the leaf inlining strategy to. This is done because in the current implementation the default strategy cannot be combined with a different inline strategy. Of all the graph based strategies, the no size penalty strategy is considered to be the base.

Table 8.3 shows that only for Acorn, the hypothesis that early speed is traded in for later performance gain holds true. There are 9 benchmarks that perform worse, 3 that perform better and 2 that are mixed. A difference with the graph based versus default strategy is that for inlining, very little results are significantly different from the default inlining strategy. This implies that not enough changes have been performed to the strategy to cause a measurable increase or decrease in performance.

The combined results are overall negative, but not significantly different from the default inlining strategy.

	Typescript	Acorn	Babel-minify	Babylon	Buble
50 iterations No Size	-3.7±0.9	-3.5±1.6	-0.1±2.9	-1.0±1.9	-3.5±1.7
500 iterations No Size	-2.9±1.2	3.2±1.6	-1.1±3.9	-1.8±1.6	-1.8±1.3
	Esprima	Jshint	Source-map	Prepack	Postcss
50 iterations No Size	-1.5±2.4	-1.8±1.9	4.7±5.0	1.2±1.1	1.9±1.5
500 iterations No Size	-0.8±0.8	-1.1±2.3	2.3±2.3	-0.2±1.8	1.2±1.2
	Prettier	Terser	Chai	Espree	**Combined**
50 iterations No Size	-3.5±2.2	-3.8±2.0	-0.3±1.0	2.1±1.5	-0.9±1.3
500 iterations No Size	-2.0±2.3	-0.1±2.3	-0.1±0.8	1.5±2.2	-0.7±0.8

Table 8.3: Performance statistics of inline leaf strategy vs no changed inline strategy for No size strategy.

Figure 8.52 shows that the median benchmark of the leaf inlining strategy performs slightly worse than the default inlining strategy. Similar to figure 8.50 and 8.51 the range for the first 50 iterations is larger than for the whole 500 iterations, suggesting that the differences do become smaller over time. These results do show that the overall performance has decreased as opposed to the default inlining strategy.

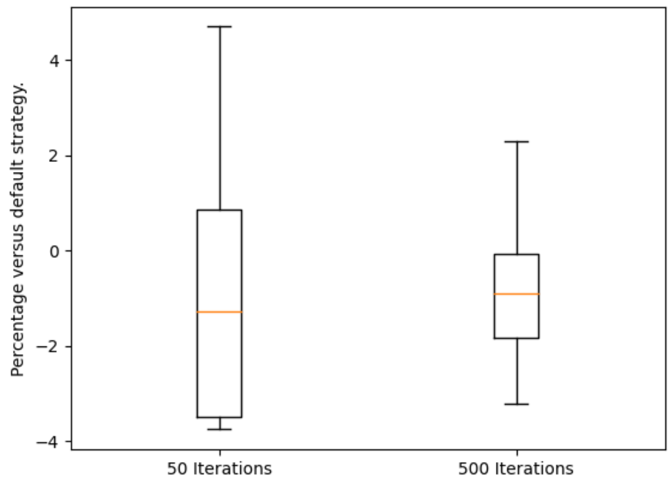


Figure 8.52: Boxplot performance of leaf inlining strategy as percentage of no size strategy with default inlining.

Figure 8.53 shows that for the Source-map benchmark the leaf inlining strategy behaves a bit worse than the default inlining strategy early on. Looking at the whole benchmark in figure 8.54 the leaf inlining strategy seems to catch up to the default inlining and at points is even slightly better, although not significantly at any point.

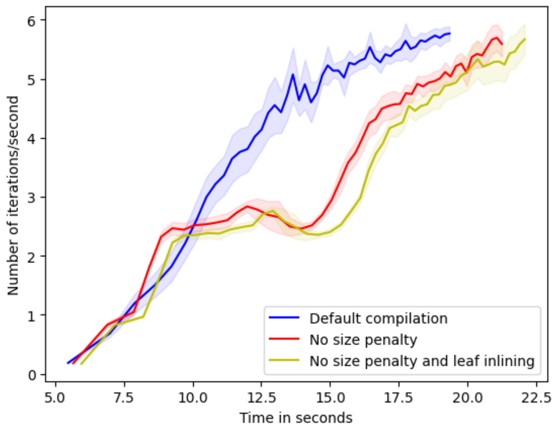


Figure 8.53: Number of iterations per second for Source-map benchmark and inlining. 50 Iterations per run. 20 runs per strategy.

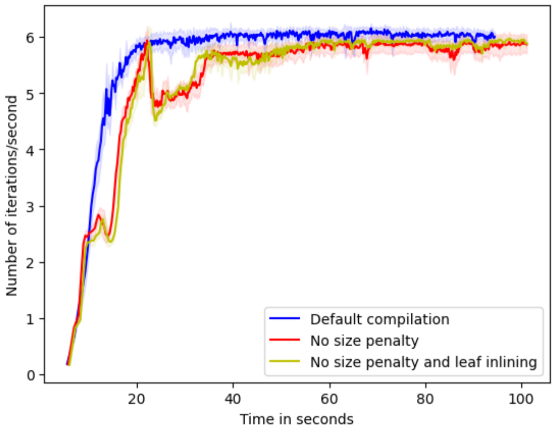


Figure 8.54: Number of iterations per second For Source-map benchmark and inlining. 50 Iterations per run. 20 runs per strategy.

In the Postcss benchmark, the leaf strategy performs very slightly better than the default strategy, although this does seem to be within statistical noice levels. Looking at both figure 8.55 and 8.56 it stands out that the leaf inlining strategy is very slightly better throughout the whole benchmark. This compounded improvement causes the leaf strategy to almost be significantly better than the default inlining strategy according to table 8.3.

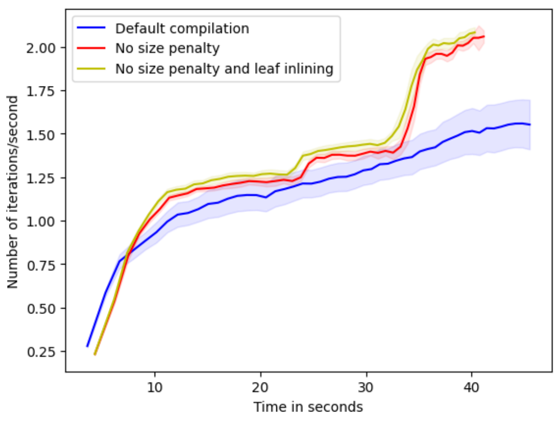


Figure 8.55: Number of iterations per second for Postcss benchmark and inlining. 500 Iterations per run. 20 runs per strategy.

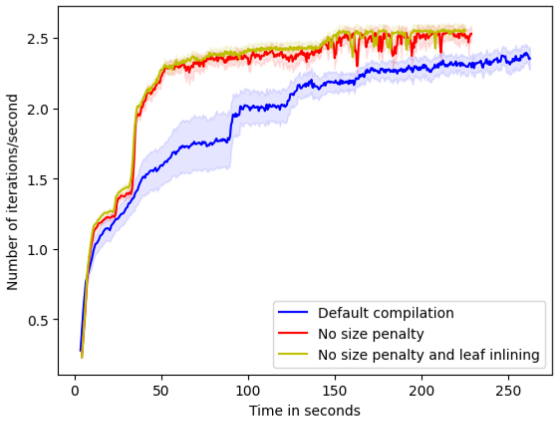


Figure 8.56: Number of iterations per second for Postcss benchmark and inlining. 500 Iterations per run. 20 runs per strategy.

8.13 Threats to validity

Some decisions made in the options, the call graph creation, and in the compilation strategy may influence the performance of the benchmarks in unexpected ways. This section discusses these decisions.

The first decision is to restrict the number of CPU cores available to the compiler. By only allowing the compiler to use 1 core, the results become very consistent. It also slows down the performance improvement of the virtual machine. This especially influences the default strategy, as the reordering of the compilation queue and the dynamic compilation threshold might be optimised for more compilation speed. The restricted number of compilations has the advantage of making it more important in what order functions are being compiled, which highlights the difference in performance of different strategies.

Splitting was already discussed in some of the benchmark sections. The splitting strategy is not taken into account during the creation of the call graphs. If a function is split during compilation, both callTargets are considered separate functions, and thus its callCounts get split between both these newly created functions. The graph based compilation strategy does still assume that both split functions are combined and thus the compilations threshold is much higher than intended. This causes the function to be added to the compilation queue later. This causes the default compilation strategy to perform more compilations on split functions.

A function that has many distinct incoming calls, and thus many split opportunities, will generally also have a high callCount and thus a high compilation priority. This high priority may lead to the function receiving a low compilation threshold. This low threshold is then shared among all splits of this function. In some cases this causes many of the splits of the function to be entered into the compilation queue very early, even though the splits are not called often individually.

Chapter 9

Conclusion

In conclusion, GraalVM provides a good framework for profiling instruments, such as a call graph tool. The instrumentation framework in combination with the Intermediate Representation allows the tool to work with multiple programming languages. The call graph profiling takes approximately 8 times longer than the non-profiled execution. This is reasonable for offline ahead of time computation, but must be optimised further if used just in time.

The graph database allows the call graphs to be quickly and easily imported. Querying is also fast and the visualisation possibilities of Neo4j can help developers who are knowledgeable about their profiled application.

A problem with the virtual machine is that functions are never removed from the compilation queue. The initialisation functions that are present in all benchmarks are submitted in the compilation queue and are then never executed again. These functions still have priority over all tier 2 compilations. In the graph based strategies this can be solved by using warm up times, but in the default strategy this problem remains.

Another issue is that tier 1 compilations always have priority over tier 2 compilations. In the graph based strategies all functions are added to the compilation queue at some point. Some of these functions are only rarely executed, however they still have priority over tier 2 compilations of functions that are executed very often. In the graph based strategies this can be solved by slowly submitting functions to the compilation queue. The default strategy fixes this problem with the dynamic compilation threshold. This threshold will prevent rarely used interpreted functions from being sent to the compilation queue, while very often executed tier 1 compiled functions may still enter the queue to be tier 2 compiled.

The final issue concerns splitting for graph based strategies. It might be a good idea to allow splitting decisions to be made during call graph profiling. Even if not all splitting decisions are made by the compiler in the small time available for profiling, it still helps to relieve the problems discussed in the previous section on threats to validity.

Overall, the performance of the graph based strategies is good. By only performing a single offline iteration good intuitions can be gained about which functions are important to compile. It is important to note that the benchmarks were deterministic and that in real life scenarios the performance may decrease due to increased difference between the call graphs. There is a significant performance improvement in early startup phase for the graph based strategies over the default strategy. There is also a large increase in performance after 50 and 500 iterations, although this increase is not statistically significant. Some benchmarks can definitely be improved greatly by using this offline graph data. The difference between the different graph based compilation strategies is less clear. The effect the compilation time of a function has on the performance of the total benchmark should be researched further.

The leaf inlining strategy overall had a slightly negative effect on performance. More test can be performed on whether it makes sense to inline either larger or more functions.

To conclude, a call graph is a valuable representation of an application and provides information that can help improve runtime performance in a virtual machine.

Bibliography

[1] Rakan Alanazi, Gharib Gharibi, and Yugyung Lee. Facilitating program comprehension with call graph multilevel hierarchical abstractions. *Journal of Systems and Software*, 176:110945, 2021. 9

[2] K. Ali, X. Lai, Z. Luo, O. Lhotak, J. Dolby, and F. Tip. A study of call graph construction for jvm-hosted languages. *IEEE Transactions on Software Engineering*, 47(12):2644–2666, dec 2021. 9

[3] Karim Ali and Ondřej Lhoták. Application-only call graph construction. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 688–712, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 9

[4] M. Arnold and D. Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. pages 51–62, 2005. 1

[5] Maciej Besta, Emanuel K. Peter, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *ArXiv*, abs/1910.09017, 2019. 5

[6] Michael Burch. The dynamic call graph matrix. pages 1–8, 09 2016. 10

[7] Michael Burch, Christoph Müller, Guido Reina, Hansjoerg Schmauder, Miriam Greis, and Daniel Weiskopf. Visualizing Dynamic Call Graphs. In Michael Goesele, Thorsten Grosch, Holger Theisel, Klaus Toennies, and Bernhard Preim, editors, *Vision, Modeling and Visualization*. The Eurographics Association, 2012. 10

[8] Mikhail Dmitriev. Profiling java applications using code hotswapping and dynamic call graph revelation. *SIGSOFT Softw. Eng. Notes*, 29(1):139–150, jan 2004. 10

[9] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. *SIGPLAN Not.*, 44(10):283–300, oct 2009. 10

[10] Google. https://blog.chromium.org/2016/10/making-chrome-on-windows-faster-with-pgo.html. 10

[11] Google. https://github.com/v8/web-tooling-benchmark. 28

[12] S.L. Graham and Peter Kessler. Gprof: A call graph execution profiler. *ACM SIGPLAN Notices*, 17, 06 1982. 5

[13] David Grove and Craig Chambers. Ibm research report an assessment of call graph construction algorithms. 06 2000. 4

[14] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, nov 2001. 9

[15] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. *SIGPLAN Not.*, 30(10):108–123, oct 1995. 10

[16] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Lett. Program. Lang. Syst.*, 1(3):227–242, sep 1992. 1

[17] Mehadi Hassen and Philip K. Chan. Scalable function call graph-based malware classification. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, page 239–248, New York, NY, USA, 2017. Association for Computing Machinery. 10

[18] Intel. https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming/profile-guided-optimization-pgo.html. 10

[19] Erik Johansson and Sven-Olof Nyström. Profile-guided optimization across process boundaries. *SIGPLAN Not.*, 35(7):23–31, jan 2000. 10

[20] Mehdi Keshani. *Scalable Call Graph Constructor for Maven*, page 99–101. IEEE Press, 2021. 9

[21] Mátyás Komáromi, István Bozó, and Melinda Tóth. An efficient graph visualisation framework for refactorerl. *Studia Universitatis Babeș-Bolyai Informatica*, 63:21–36, 06 2018. 10

[22] Byeongcheol Lee. Adaptive correction of sampling bias in dynamic call graphs. *ACM Trans. Archit. Code Optim.*, 12(4), dec 2015. 9

[23] Ondrej Lhoták. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, page 37–42, New York, NY, USA, 2007. Association for Computing Machinery. 10

[24] Justin J. Miller. Graph database applications and concepts with neo4j. 2013. 5

[25] neo4j. https://github.com/lpechacek/cpuset. 27

[26] neo4j. https://llvm.org/docs/benchmarking.html. 27

[27] neo4j. https://neo4j.com/. 19

[28] Neo4j. https://neo4j.com/developer/cypher/. 19

[29] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of node.js applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 29–41, New York, NY, USA, 2021. Association for Computing Machinery. 10

[30] Oracle. https://github.com/oracle/graal. 1

[31] Oracle. https://github.com/oracle/graal/blob/master/tools/src/com.oracle.truffle.tools.profiler/src/com/oracle/
14

[32] Oracle. https://github.com/oracle/graaljs. 2

[33] Oracle. https://www.graalvm.org/22.0/reference-manual/java/compiler/. 5

[34] Oracle. https://www.graalvm.org/22.1/docs/getting-started/get-started-with-graalvm. 5

[35] Oracle. https://www.graalvm.org/22.1/graalvm-as-a-platform/implement-instrument/. 5

[36] Oracle. https://www.graalvm.org/22.1/graalvm-as-a-platform/language-implementation-framework/. 5

[37] Oracle. https://www.graalvm.org/22.1/graalvm-as-a-platform/language-implementation-framework/splitting/monomorphizationusecases/. 7

[38] Oracle. https://www.graalvm.org/22.1/graalvm-as-a-platform/language-implementation-framework/traversingcompilationqueue/. 7

[39] Oracle. https://www.graalvm.org/22.1/reference-manual/js/. 5

[40] Nadav Rotem and Chris Cummins. Profile guided optimization without profiles: A machine learning approach, 2021. 10

[41] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. Pycg: Practical call graph generation in python. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 1646–1657. IEEE Press, 2021. 9

[42] Denys Shabalin and Martin Odersky. Interflow: Interprocedural flow-sensitive type inference and method duplication. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, Scala 2018, page 61–71, New York, NY, USA, 2018. Association for Computing Machinery. 10

[43] J. M. Spivey. Fast, accurate call graph profiling. *Softw. Pract. Exper.*, 34(3):249–264, mar 2004. 10

[44] Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal ir : An extensible declarative intermediate representation. 2013. 5

[45] P Stookappe. https://github.com/patricksss/graal. 27

[46] P Stookappe. https://github.com/patricksss/graal/tree/master/auxiliryfiles. 14, 19, 27

[47] P Stookappe. https://github.com/patricksss/graal/tree/master/auxiliryfiles/benchmarks. 28

[48] Tobias Hartmann, Zoltán Majó . The java hotspot vm under the hood. URL: `chrome-extension://oemmndcbldboiebfnladdacbdfmadadm/http://cr.openjdk.java.net/~thartmann/talks/2016-Hotspot_Under_The_Hood.pdf`, 2 2016. 6

[49] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. volume 10, page 42, 01 2010. 5

[50] Baptiste Wicht, Roberto A. Vitillo, Dehao Chen, and David Levinthal. Hardware counted profile-guided optimization, 2014. 10

[51] David Williams-King and Junfeng Yang. Codemason: Binary-level profile-guided optimization. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, FEAST'19, page 47–53, New York, NY, USA, 2019. Association for Computing Machinery. 10

[52] Tao Xie and David Notkin. An empirical study of java dynamic call graph extractors. 2001. 9