Comparing a Tree-walk Interpreter with JIT compilation and embedding via Go-plugins

Evaluating the trade-offs of using the Go-plugin API for JIT compilation while comparing the approach with a Tree-walk interpreter

9525469

May 3, 2024

Contents

1 Introduction **1**

2 Query Language **2**
2.1 Feature set . . . 3
2.2 Evaluation Approach . . . 3

3 Compiler Invocation **4**
3.1 Including the Compiler Source Code . . . 4
3.2 Invoking the local Go Compiler . . . 4

4 Plugin API **5**
4.1 Compiling Go Source Code to Go plugins . . . 5
4.2 Embedding Go plugins . . . 5
4.3 Trade-offs, Issues and Considerations . . . 6

5 Just in Time Compilation **7**
5.1 Meta-tracing & `JIT_CONSTANT` . . . 7
5.2 Connecting the JIT to the Runtime . . . 9
5.3 Function Parameters and Erasing Types . . . 9
5.4 Concurrent Compilation . . . 9
5.5 Type System Clashes . . . 9
5.6 Bailing out to the Interpreter . . . 9

6 Benchmarks **10**
6.1 Arithmetics . . . 10
6.2 String operations . . . 11
6.3 Real world workloads . . . 12
6.4 Discussing Performance Impact . . . 13
6.5 Determining the `JIT_CONSTANT` . . . 14

7 Conclusion **15**
7.1 Usability and Robustness . . . 15
7.2 Performance . . . 15
7.3 Implementation Complexity . . . 15

8 Appendix **III**

List of Listings

2.1 Boolean algebra 2
2.2 Generating boolean algebra 2
2.3 Reducing and formatting objects in lists - source [12] 3
2.4 Applying filter, mapping, join entries with comma - source [12] 3
2.5 Named Functions and unnamed/anonyoums functions - source [12] 3

4.1 Tool-chain invocation with plugin compilation 5
4.2 Plugin compilation, plugin opening and function resolution 6

5.1 `Function[V any] struct` type with meta-tracing 7
5.2 `JIT_CONSTANT` definition 8
5.3 `Function[V any] struct` type with meta data 8

6.1 Heavy load arithmetic operations 10
6.2 Heavy load string concatenating 11
6.3 Real world heavy load benchmark 13

8.1 Tool-chain invocation III
8.2 Passing Go values into the language context - source [12] III

List of Figures

6.1 Benchmark: Arithmetic operations 11
6.2 Benchmark: String concatenating 12
6.3 Benchmark: Real world heavy load benchmark 13

List of Tables

6.1 Arithmetic operations benchmark results 11
6.2 String concatenation benchmark results 12
6.3 String concatenation benchmark results 13

The goal of this paper is to evaluate whether the usage of the Go plugin API is feasible for just-in-time compilation of a query language intended for a high performance in memory data storage. This evaluation is done based upon the criteria of the ease of usability, performance and the robustness of the resulting implementation. For the sake of comparison the query language as well as its features are introduced. A just in time compiler is implemented and benchmarked against the same expressions evaluated with the currently employed tree walk interpreter. The paper explores the different possibilities for accessing the Go compiler, working with the Go plugin API and highlights several benchmarks comparing the performance of the new JIT compiler and the previous language evaluation implementation.

Das Ziel dieser Arbeit besteht darin zu evaluieren ob sich das Einsetzen der Go plugin API für die JIT Kompilation einer Abfragesprache einer effizienten Datenbank, die ihre Daten ausschließlich im Arbeitsspeicher ablegt, rentiert. Diese Abwägung basiert auf den Kriterien der Verwendbarkeit, Effizienz und der Stabilität der angewendeten Lösung. Um den Vergleich unter realistischen Bedingungen zu testen wird vorerst die Abfragesprache sowie einige umfangreicher werdende Beispiele vorgestellt. Diese Abfragen werden dann mit Hilfe des JIT compiliert. Das Ergebnis wird dann mit der derzeitige Implementierung die das Evaluieren mit einem Tree-walk Interpreter umsetzt, verglichen. Auch werden unterschiedliche Art und Weisen des Aufrufes des Go compiler, die Verwendung des Go plugin paketes und Tests auf die Effizienz des neuen JIT und des alten Tree-walk Interpreter vorgestellt und diskutiert.

1 Introduction

The query language is the singular interface for accessing, reading, creating and removing data in a database. This requires the query language to provide a high degree of performance in the sense of performing processing intensive queries in a fast enough time for real-time responsiveness, especially for an in memory data store with the aspiration for high performance.

Optimisations for database query languages are common, such as in the embeddable Database *SQLite* with the *SQLite Query Optimizer*[1] and the *Next-Generation Query Planner*[1] both supporting a variety of optimisations after compiling SQL expressions to byte-code instead of walking the AST[1][2]. *PostgreSQL* is an other example of a database optimising its SQL queries, using a JIT compiler[2][4].

There are several optimisations applicable for any programming language and query languages in particular [5, 3.3 Optimisations]. Some were already applied to the query language [6]. Implementing a JIT compiler can significantly improve the performance of a long running highly processing intensive program and allows it to outperform optimisations applied to the AST before walking it or compiling to bytecode instructions and executing them in a dedicated virtual machine [3, 4. Results].

The implementation of a JIT compiler introduces complexity into the codebase for the generation of optimized assembly is inherently complex and time consuming [7, 1. Introduction] while also opening the door for potential attacks [8, 2. Challenges Securing JavaScript JIT] [8, Table 1.]. Another issue is the platform dependence, a JIT compiler has to generate native code for all platforms and operating systems the database and therefore the query language support - increasing complexity and maintenance [9, Abstract]. Furthermore a JIT compiler significantly increases the memory usage [10, Fig. 1.] of the interpreter as well as requiring a not to be disregarded startup time made up of the time to start the JIT compiler and the time to compile the language constructs to machine code. [9, 4.2.7 Breakdown of compilation times].

To evaluate whether the aforementioned positive aspects outweigh the negative points and a JIT therefore does improve the query languages performance, determines the scope of this paper. The examined performance is measured on a fork of the original language project [11], this fork was necessary to pave the way for just in time compilation [12].

[1]Abstract Syntax Tree: tree of syntax nodes

[2]Just in time compiler: compiling methods on demand while a program is running [3, 1. Introduction]

2 Query Language

The query language is generic by design and by usage of the newly introduced generic proposal [13]. This allows for rudimentary and complex language variations depending on the data type the desired language is created with. Such as complexity ranging from boolean algebra (see Listing 2.1 and its corresponding language definition Listing 2.2), over arithmetic operations to complex queries on lists of objects, such as filtering for values, mapping and mutating elements of the list and iterating over the entries of the list.

```
true | false
a & !b
let c=true;
if c then
    a&b
else
    a|b
```

Listing 2.1: Boolean algebra

```
var boolParser = funcGen.New[bool]()
    .AddConstant("false", false)
    .AddConstant("true", true)
    .AddSimpleOp("^", true,
            func(a, b bool) (bool, error) { return a != b, nil })
    .AddSimpleOp("=", true,
            func(a, b bool) (bool, error) { return a == b, nil })
    .AddSimpleOp("|", true,
            func(a, b bool) (bool, error) { return a || b, nil })
    .AddSimpleOp("&", true,
            func(a, b bool) (bool, error) { return a && b, nil })
    .AddUnary("!", func(a bool) (bool, error) { return !a, nil })
    .SetToBool(func(c bool) (bool, bool) { return c, true })
```

Listing 2.2: Generating boolean algebra

2.1 Feature set

To limit the scope of performance evaluation, the feature set is determined as the language definition and dialect[1], specified in the `value` package of the `parser2` [12, `value` package] project. This dialect allows for aggregation, filtering and mutating large lists of data sets consisting of objects with many key value pairs.

```
persons
    .map(p -> p.Name)
    .reduce((a,b) -> a+", "+b)
```

Listing 2.3: Reducing and formatting objects in lists - source [12]

```
persons
  .accept(p -> p.PlaceOfBirth="New York" & p.Age>21)
  .map(e -> e.Name+": "+e.Age)
  .reduce((a,b) -> a+", "+b)
```

Listing 2.4: Applying filter, mapping, join entries with comma - source [12]

Listing 2.3 and Listing 2.4 access the global `person` constant using the dialect specific facilities for creating a list of objects of type `Person` as global state (see Listing 8.2).

```
func namedFunction(argument) argument+2;
let m = (argument) -> argument+2;
let add = (a,b) -> a+b;
```

Listing 2.5: Named Functions and unnamed/anonyoums functions - source [12]

The just in time compiler targets named functions and anonyoums functions (see Listing 2.5) and will attempt to compile their contents when conditions for their compilation are hit.

2.2 Evaluation Approach

The runtime currently employs the visitor pattern to walk the AST the parser generated in the previous step of the stages necessary to transform a given character stream to an executable data structure. To evaluate a given input, the runtime generates a function once for each tree node it visits. This improves the performance rapidly compared to a naive tree-walk-interpreter. Furthermore the AST is optimized before being walked by the function generating stage of the runtime.

This evaluation strategy requires the runtime to hold a substantial amount of data structures in memory compared to a bytecode interpreter.

[1]Refers to the usage of [12] to define and generate a language for a specific data type, with static global functions and constant values

3 Compiler Invocation

Instead of manually generating optimized assembly for each function to be compiled on the fly, the go compiler toolchain is invoked and receives the previously generated Go source code that was computed from the query languages AST. This omits the complexity of implementing and maintaining several platform specific machine code generator compiler backends while allowing the JIT compiler to support all platforms supported by the go toolchain.

The method of invoking the compiler tool-chain has significant effects on the startup performance, the robustness and the complexity of the JIT compiler. This chapter highlights two possible approaches for invoking the compiler tool-chain.

3.1 Including the Compiler Source Code

The first idea of invoking the compiler tool-chain, is to include the source code of the compiler as a library and simply start it while passing in the generated go code. This does not require the compiler tool-chain to exist on the target system and omits the overhead of starting the compiler process. However this approach can not be used for the source code since the go compiler is not stable nor accessible outside of the go compiler tool-chain [14, *(gcToolchain).gc*] due to the usage of `internal` packages [15].

3.2 Invoking the local Go Compiler

The remaining method is to start the locally available compiler tool-chain via the `exec.Cmd` interface [16, Overview]. This enables requesting the operating system to invoke the compiler. Approaching the problem with this method has the downside of requiring the compiler to exist on the target system, the overhead of tasking the operating system with starting the compiler, writing the generated code to a temporary file and compiling this temporary file instead of doing all of the aforementioned inside of the JIT by including the compiler as a library as introduced before.

Listing 8.1 shows a simplified implementation of a function invoking the go compiler. Error handling is omitted for the sake of simplicity.

4 Plugin API

The `plugin` package enables the loading of shared objects compiled with the `-buildmode=plugin` compiler flag and the resolution of symbols contained in the plugin [17, Overview]. This allows the compilation and loading of go code while running a program.

4.1 Compiling Go Source Code to Go plugins

As introduced above the compiler tool-chain accepts different build modes via the `-buildmode` command line argument [18]. The build mode for compiling a given source file to the go plugin format is named `plugin` [18] [17, Overview].

Listing 4.1 contains a modified version of Listing 8.1, adding the compiler flags for compiling the generated source code passed via the `code` function parameter, to a go plugin. Instead of producing an executable for the target architecture and operating system the compiler now generates a shared object in the format the `plugin` package requires.

```
func invokeCompiler(code string) {
    f, _ := os.CreateTemp(".", "jit_*.go")
    defer os.Remove(f)
    f.WriteString(code)
    pre := strings.TrimSuffix(f.Name(), ".go")
    c := exec.Command(
        "go", "build", "-buildmode=plugin", "-o", pre, f.Name())
    c.Run()
}
```

Listing 4.1: Tool-chain invocation with plugin compilation

4.2 Embedding Go plugins

The loading of plugins and the resolution of plugins uses the API exposed by the previously introduced `plugin` package.

Listing 4.2 modifies Listing 4.1 for opening the previously compiled plugin. Once opened the `*plugin.Plugin` structure can be used for resolving exported functions and variables included in the plugin. After resolving a symbol[1] its type is `any`, therefore the function `Main`[2] has to

[1] a symbol refers to a function, constant or variable

[2] for the sake of this explanation the generated code in the function parameter `code` is assumed to be contained in the `Main` function

be cast to `func()` before the go type system allows a function call. Upon type casting the function is called and the generation, compilation and calling workflow of the JIT compiler is concluded.

```go
func invokeCompiler(code string) {
    f, _ := os.CreateTemp(".", "jit_*.go")
    defer os.Remove(f)
    f.WriteString(code)
    pre := strings.TrimSuffix(f.Name(), ".go")
    c := exec.Command(
        "go", "build", "-buildmode=plugin", "-o", pre, f.Name())
    c.Run()

    plug, _ := plugin.Open(pre)
    // assumes generated code lives in func Main()
    symbol, _ := plug.Lookup("Main")
    Main, _ := symbol.(func())
    Main()
}
```

Listing 4.2: Plugin compilation, plugin opening and function resolution

4.3 Trade-offs, Issues and Considerations

The `plugin` package provides the program with the unique ability to allow for high performance on the fly code compilation and execution. It therefore fits the use case of a query language implementation well.

However the `plugin` package bears several downsides [17, Warnings], primarily the missing portability due to the package only supporting Linux, FreeBSD and MacOS. Another disadvantage is the strict requirement of both the host application and all plugins needing to be compiled with the same tool-chain version and build-tags - this is particularly difficult in the case of this JIT, due to the requirement of the existence of the local compiler that will most certainly not be of the exact same version as the compiler used for compiling the host application. Is the previously mentioned not strictly ensured runtime errors can occur. A further drawback is the increased difficulty of reasoning about program and plugin initialisation for the special `func init()` function is called upon opening a plugin [17, Overview], possibly opening the program up to race conditions and similar critical bugs due to global state initialisation [19, The init function].

5 Just in Time Compilation

Just in time compilation refers to the process of determining whether a segregated chunk of code is considered "hot"[1] and compiling this code segment into operating system and architecture specific machine code ad hoc. This machine code is then loaded into the memory of the interpreters runtime and executed instead of interpreting the code chunk [21]. The details of just in time compilation, meta tracing, categorizing code segments as "hot", improving the performance of the just in time compiler and error handling are explored in this chapter.

Contrary to the previously introduced definition of a just in time compiler in the context of programming language interpreters, go does not support dynamically loading machine code into memory and executing these chunks. The mitigation for this is introduced and explained in chapter 4.

5.1 Meta-tracing & `JIT_CONSTANT`

```go
// Function represents a function in the interpreter runtime
type Function[V any] struct {
    // ...

    // Counter stores the amount of calls made to the function
    Counter int
}
```

Listing 5.1: `Function[V any] struct` type with meta-tracing

Meta-tracing refers to the process of tracking the actions of the programming language interpreter [22, 4.1 Meta-tracing]. The interpreter uses this functionality to determine the amount of invocations of a function and updates the `Function.Counter` field accordingly, see Listing 5.1. Once this counter reaches the threshold defined in the `JIT_CONSTANT` (see Listing 5.2) the `type Function[V any] struct` instance is passed to the just in time compiler compilation queue, in which it will be compiled with other functions waiting to be compiled. Upon the Function being compiled the interpreter executes the output of the just in time compiler for each function invocation instead of walking the abstract syntax tree and thus is no longer interpreting the function, but instead uses the compiled representation.

[1] hot in the context of just in time compilation refers to a code path or a segment of code that is executed massive amount of times [20], [21]

```go
// JIT_CONSTANT sets the threshold the function invocation meta tracing counter
// has to pass for the function to be considered hot and thus compilable
var JIT_CONSTANT int = 1_000
```

Listing 5.2: JIT_CONSTANT definition

This constant threshold varies from compiler to compiler. The value depends on the performance needs and the hit the runtime performance takes upon invoking the jit compiler. Specifics are discussed in Section 6.5.

The JIT-compiler requires some information about a function before it's being able to start the code generation step. Not only does it require the name of the function[2], but the names of its arguments and the types the JIT can use to compile the given function. The necessary fields are stored in the previously introduced `type Function[V any] struct`, specifically the `type MetaData struct` and `type MetaDataParameter struct` structures (see Listing 5.3).

```go
type MetaDataParameter struct {
        Name string
        Type string
}
type MetaData struct {
        Parameters []MetaDataParameter
}
// Function represents a function in the interpreter runtime
type Function[V any] struct {
    // ...

    // ArgumentNames contains the list of parameter names of the function
    ArgumentNames []string
    // Name holds the name of the function
    Name string
    // MetaData holds the necessary data for the jit to compile valid functions
    MetaData *MetaData
}
```

Listing 5.3: `Function[V any] struct` type with meta data

[2] Unnamed/anonymous functions or closures are named by prefixing a closure counter with `c`, the first encountered closure will therefore be compiled as `func c0()`

5.2 Connecting the JIT to the Runtime

5.3 Function Parameters and Erasing Types

5.4 Concurrent Compilation

5.5 Type System Clashes

5.6 Bailing out to the Interpreter

6 Benchmarks

The following sections will measure the impact the jit compilation has on selected workloads.

Benchmark results may be subject to various influences, including the workload on the system conducting the benchmarks, insufficiently sized data sets for comprehensive and accurate testing, as well as comparing inherently dissimilar benchmarks. Most of the aforementioned can be mitigated by using the `testing` package included in the go programming languages standard library [23].

To avoid the influences of the current workload of the system the tests are performed many times, therefore accounting for statistical outliers and external system influences on the test, which furthermore accounts for the possibility of choosing insufficiently sized data sets.

By benchmarking the execution of the same given input with the JIT-compiler enabled and with the JIT-compiler disabled an inherently comparable data set is created due to the shared purpose of evaluating the runtime performance under differing runtime configurations. Therefore, conducting benchmarks under the two previously mentioned configurations and comparing the results is a valid evaluation of the two benchmarks.

The benchmarks simulate a hot path by executing a given operations for a given iteration count and running the benchmark itself multiple times using the command line benchmarking tool *hyperfine*.

6.1 Arithmetics

```
func b(a)
    a*a/25*a-12+a/a*a*a*a/25*a-12+a/a*a*a*a/25*a
    -12+a/a*a*a*a/25*a-12+a/a*a*a*a/25*a-12+a/a*a
    *a*a/25*a-12+a/a*a*a*a/25*a-12+a/a*a*a*a/25
    *a-12+a/a*a*a*a/25*a-12+a/a*a*a*a/25*a-12+a/a
    *a*a*a/25*a-12+a/a*a*a*a/25*a-12+a/a*a;
let s = list(100_000).map(b).sum();
s
```

Listing 6.1: Heavy load arithmetic operations

Benchmarking the performance of arithmetic operations allows for a first execution efficiency evaluation of the language runtime. The benchmark using Listing 6.1 results of up to 14.51x

improvement ($0.94s$ instead of $13.64s$). While the original runtime scales linearly and proportionally to the input, the runtime supported by the JIT-compiler scales at an almost constant rate, see Figure 6.1 and Table 6.1.

Iterations	Mean Execution Time	Mean Execution Time (JIT)	Δ	Improvement
100k	00.28s	0.20s	00.08s	01.40x
500k	01.37s	0.26s	01.11s	05.27x
1mio	02.73s	0.34s	02.39s	08.03x
5mio	13.64s	0.94s	12.70s	14.51x

Table 6.1: Arithmetic operations benchmark results

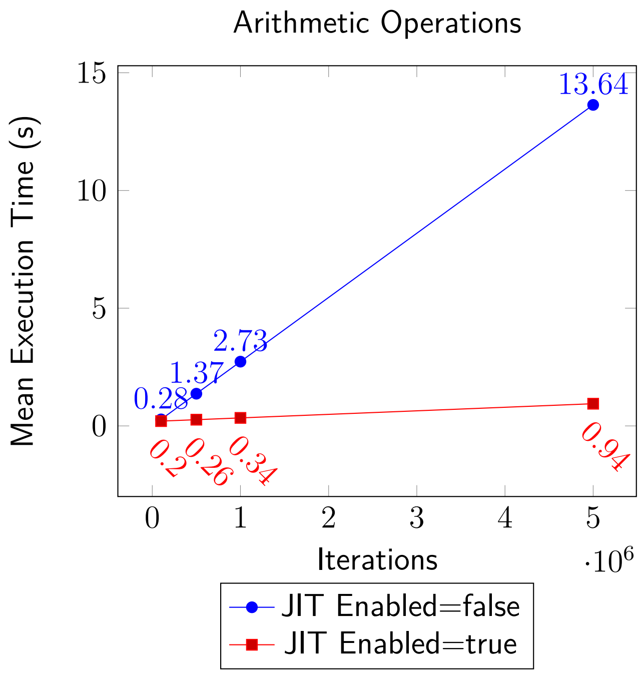

Figure 6.1: Benchmark: Arithmetic operations

6.2 String operations

```
func b(a)
    a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a;
let s = list(100_000).map(e->e.string()).map(b).size();
s
```

Listing 6.2: Heavy load string concatenating

Merging strings is an often used language feature, therefore making the need for high efficiency evident. This benchmark aims to simulate a real world use with heavy load, similar to the benchmark performed before. Both Table 6.2 and its visualisation Figure 6.2 show the performance improvement of at least 1.43x and at most 4.59x - resulting in a mean delta, comparing the current runtime and the runtime enhanced with the JIT, of at most 11.08s and at least 0.10s.

Iterations	Mean Execution Time	Mean Execution Time (JIT)	Δ	Improvement
100k	00.33s	0.23s	00.10s	1.43x
500k	01.57s	0.48s	01.09s	3.27x
1mio	02.99s	0.77s	02.22s	3.89x
5mio	14.17s	3.09s	11.08s	4.59x

Table 6.2: String concatenation benchmark results

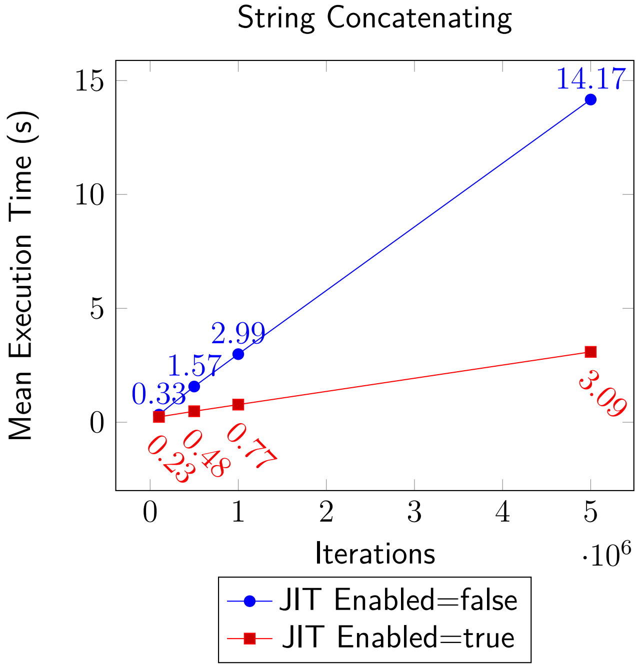

Figure 6.2: Benchmark: String concatenating

6.3 Real world workloads

Contrary to the before examined language features, this benchmark focusses on common operations performed on large datasets, such as aggregating values, reading and overwriting data, chaining list operations and working with lists of objects. Due to the mixture of operations featured in this benchmark the resulting numbers are a combination of different factors.

The improvements for smaller iterations, such as 100 thousand and 500 thousand, can be considered negligible. For the larger iterations the JIT improves the performance by at least 0.3s (1.24x) and at most 2.42s (1.431x).

```
list(100_000)
    .filter(e -> e >= 1)
    .map(e -> ((1024*(1024-e))/(e+1024))+(e*e+e*e-e*e+e*e))
    .map(e -> {Key: "Number", Value: e})
    .map(e -> e.Value)
    .map(e -> e.string().len().string())
    .map(e -> e+e+e+e+e+e+e)
    .last()
```

Listing 6.3: Real world heavy load benchmark

Iterations	Mean Execution Time	Mean Execution Time (JIT)	Δ	Improvement
100k	0.18s	0.18s	0.00s	0.000x
500k	0.87s	0.82s	0.05s	1.024x
1mio	1.68s	1.36s	0.32s	1.240x
5mio	8.04s	5.62s	2.42s	1.431x

Table 6.3: String concatenation benchmark results

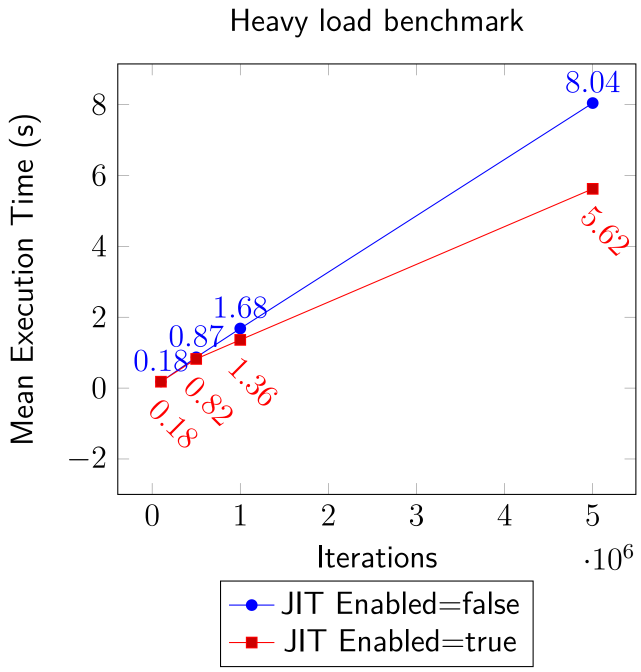

Figure 6.3: Benchmark: Real world heavy load benchmark

6.4 Discussing Performance Impact

Based on the preceding benchmarks, their results and visual representation, the performance impact of the JIT-Compiler is substantial for most workloads, but especially for larger iterations, such as in hot paths. The Go compiler optimises compiled plugins as if it were to compile a Go application, such as reducing memory allocations[24, Escape Analysis], inlining functions[24, Inlining] and not allocating empty structures [24, Interface Values]. This allows the interpreter to leverage the optimized performance the Go compiler applies to binaries it compiles, thus enabling the observed large performance improvements for string manipulation and arithmetic operations.

6.5 Determining the `JIT_CONSTANT`

7 Conclusion

7.1 Usability and Robustness

7.2 Performance

7.3 Implementation Complexity

Bibliography

[1] "The sqlite query optimizer overview." (), [Online]. Available: https://www.sqlite.org/optoverview.html (visited on 01/20/2024).

[2] "The next-generation query planner." (2023), [Online]. Available: https://www.sqlite.org/queryplanner-ng.html (visited on 01/20/2024).

[3] A. Krall, "Efficient javavm just-in-time compilation," in *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*, IEEE, 1998, pp. 205–212. [Online]. Available: https://www.complang.tuwien.ac.at/andi/papers/pact_98.pdf.

[4] "Chapter 32. just-in-time compilation (jit)." (), [Online]. Available: https://www.postgresql.org/docs/current/jit-reason.html (visited on 01/20/2024).

[5] T. Suganuma, T. Yasue, and T. Nakatani, "A region-based compilation technique for a java just-in-time compiler," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, 2003, pp. 312–323.

[6] "Parser2.optimizer." (2023), [Online]. Available: https://github.com/hneemann/parser2/blob/361d9bd2f6daf0ff76b47a1186ed2cfcd1a96d12/parser2.go#L23 (visited on 01/20/2024).

[7] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, *et al.*, "Design, implementation, and evaluation of optimizations in a just-in-time compiler," in *Proceedings of the ACM 1999 conference on Java Grande*, 1999, pp. 119–128.

[8] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "Jit-picking: Differential fuzzing of javascript engines," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 351–364.

[9] K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, *et al.*, "Effectiveness of cross-platform optimizations for a java just-in-time compiler," in *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, 2003, pp. 187–204.

[10] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 97–113, 2003.

[11] H. Neemann, *Parser2*, 2024. [Online]. Available: https://github.com/hneemann/parser2 (visited on 04/16/2024).

[12] xnacly and dani0611, *Parser2*, 2024. [Online]. Available: https://github.com/xnacly/parser2 (visited on 04/16/2024).

[13] I. L. Taylor. "Type parameters proposal." (2021), [Online]. Available: https://go.googlesource.com/proposal/+/HEAD/design/43651-type-parameters.md (visited on 04/02/2024).

[14] "Gc.go." (2023), [Online]. Available: https://github.com/golang/go/blob/master/src/cmd/go/internal/work/gc.go#L58 (visited on 01/20/2024).

[15] "Internal directories." (2024), [Online]. Available: https://pkg.go.dev/cmd/go#hdr-Internal_Directories (visited on 01/20/2024).

[16] "Os/exec package." (2024), [Online]. Available: https://pkg.go.dev/os/exec (visited on 01/20/2024).

[17] "Plugin package." (2024), [Online]. Available: https://pkg.go.dev/plugin (visited on 01/20/2024).

[18] "Build modes." (2024), [Online]. Available: https://pkg.go.dev/cmd/go#hdr-Build_modes (visited on 01/20/2024).

[19] "Effective go - the go programming language." (2024), [Online]. Available: https://go.dev/doc/effective_go (visited on 01/20/2024).

[20] "-xjit / -xnojit - ibm documentation." (Feb. 12, 2024), [Online]. Available: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=options-xjit#optlevel (visited on 04/16/2024).

[21] A. Krall, "Efficient javavm just-in-time compilation," in *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*, IEEE, 1998, pp. 205–212.

[22] C. F. Bolz and L. Tratt, "The impact of meta-tracing on vm design and implementation," *Science of Computer Programming*, vol. 98, pp. 408–421, 2015.

[23] "Go testing package." Official documentation for the Go testing package. (), [Online]. Available: https://pkg.go.dev/testing (visited on 04/16/2024).

[24] "Go wiki: Compiler and runtime optimizations." (), [Online]. Available: https://go.dev/wiki/CompilerOptimizations (visited on 05/03/2024).

8 Appendix

```go
func invokeCompiler(code string) {
    f, _ := os.CreateTemp(".", "jit_*.go")
    defer os.Remove(f)
    f.WriteString(code)
    c := exec.Command("go", "build", f.Name())
    c.Run()
}
```

Listing 8.1: Tool-chain invocation

```go
type Person struct {
        Name         string
        Surname      string
        PlaceOfBirth string
        Age          int
}

var Persons = []Person{
        {"John", "Doe", "London", 23},
        {"Jane", "Doe", "London", 25},
        {"Bob", "Smith", "New York", 21},
        {"Frank", "Muller", "New York", 22},
        {"Mary", "Green", "Seattle", 21},
        {"Jake", "Muller", "Washington", 22},
}

var PersonToMap = value.NewToMapReflection[Person]()
var persons = value.NewListOfMaps[Person](PersonToMap, Persons)
```

Listing 8.2: Passing Go values into the language context - source [12]