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

January 20, 2024

Contents

The goal of this paper is to evaluate whether or not 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 store. 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. The code a JIT would have generated for several heavy queries is handwritten 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.

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. Some were already applied to the query language [5]. 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]. To evaluate whether the expensive startup time is rewarding the query language runtime with higher performance will be tested as part of this paper.

[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

2.1 Feature set

2.2 Evaluation Approach

3 Benchmarks

3.1 Tree-Walk interpreter

3.2 Go just-in-time compiler

4 Compiler Invocation

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.

4.1 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 [6, *(gcToolchain).gc*] due to the usage of `internal` packages [7].

4.2 Invoking the local Go Compiler

The remaining method is to start the locally available compiler tool-chain via the `exec.Cmd` interface [8, 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 4.1 shows a simplified implementation of a function invoking the go compiler. Error handling is omitted for the sake of simplicity.

```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 4.1: Tool-chain invocation

5 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 [9, Overview]. This allows the compilation and loading of go code while running a program.

5.1 Compiling Go Source Code to Go plugins

As introduced above the compiler tool-chain accepts different build modes via the -buildmode command line argument [10]. The build mode for compiling a given source file to the go plugin format is named plugin [10] [9, Overview].

Listing 5.1 contains a modified version of Listing 4.1, 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.

```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()
}
```

Listing 5.1: Tool-chain invocation with plugin compilation

5.2 Embedding Go plugins

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

Listing 5.2 modifies Listing 5.1 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 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 5.2: Plugin compilation, plugin opening and function resolution

5.3 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 [9, 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 [9, Overview], possibly opening the program up to race conditions and similar critical bugs due to global state initialisation [11, The init function].

[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

6 Conclusion

6.1 Usability and Robustness

6.2 Performance

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

[6] "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).

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

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

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

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

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