[DRAFT] DistGo: Prototype and analysis of distributed Go's tooling

Sebastiaan Gerritsen (2818056), Hanjun Liu (2720974)

1 INTRODUCTION

There is a growing interest in running applications in massively distributed systems in order of milliseconds. Advancements are being made in different domains, for example with RAI [20] in the field of AI and with Dragon for stream processing [18]. The objective of this report is to extend the ecosystem by proposing a massively scalable prototype for a software compiler, focusing on the distributed compilation of Go programs. Compared to languages like C/C++ Go was built from the ground up to support fast compilation times [21]. Consequently, Go ended up being one of the fastest compiling languages ready for production use. However, currently the Go compiler functions only on a single machine, limiting performance increases to vertical scaling. For this report we introduce a prototype which implements horizontal scaling, potentially decreasing compilation times even further. First we start by giving an overview of the internal workings of the local version of the Go compiler, next a preliminary performance analysis is performed to identify possible bottlenecks in the current implementation. After identifying the optimisation candidate we describe the proposed design and its implementation. Finally we perform a performance analysis on the distributed version, and compare the results to the local version.

2 INTERNAL WORKINGS

For analysis we consider version 1.21.6 of Go, as this was the most recent version at the start of this project. The compilation process in Go can be conceptualized into two distinct levels: intra-package compilation, which handles the transformation of source files within individual packages, and inter-package compilation, which orchestrates the compilation across multiple packages. In this section we make use of path references to point to files or relevant sections in the go codebase hosted at GitHub.

2.1 Inter-package

The `go build` command compiles the packages along with their dependencies in the given directory. It does not however install the results like `go run` or `go install`, thus the build command gives a clearly defined entry-point to the Go compiler [5]. When executing `go build` the flags and arguments are parsed to determine the compiler behaviour. For the passed files it performs package patterning, if a single package is passed a binary is built, if multiple packages are passed the end result is discarded. Next the packages are loaded recursively, it does so by reading the package which was passed as argument and checking for any imports. If the imports are from the standard library Go will try to resolve the dependencies by checking `GOROOT` or `GOPATH`, if the imports are external it will first try to resolve the corresponding module at `GOMODCACHE` using the `go.mod` file. When the package cannot be found on disk the corresponding module is fetched from the Go proxy [19]. While recursively resolving the dependencies each package is assigned to one of two types of actions:

(1) **Compile**: This actions contains the definitions and configurations for calling the `go tool compile` command internally on the respective package, it is responsible for converting the Go source files to an intermediate representation called object files.
(2) **Link**: This action takes one or more object files produced by the compile action and combines them into a single executable binary.

Both actions make use of a caching system, in which GO computes a cache key based on the action type and package. The assignment of packages to actions results in a so called action-graph, it is a DAG which defines the relationships and dependencies between different actions. The actions contained in the action-graph are then executed in depth-first post-order priority, which, upon completion, will result in the final binary [1].

2.2 Intra-package

The `go build` command calls upon the functionality of the `go tool compile` each time a package needs to be transformed from source files into an intermediate representation, which can then be used to link the final binary. The `go tool compile` command is thus responsible for the compilation itself, this process can be split in four phases, parsing, AST-transformations, SSA and generating machine code [6].

2.2.1 Parsing: In the first phase each source file is transformed into a concrete syntax tree (CST). A CST is an exact representation of the respective source file, where nodes in the CST correspond to elements in the source file. A CST is built by first tokenizing the source file by performing lexical parsing. During lexical parsing the Go compiler opens multiple source files concurrently using go routines and starts scanning the source files

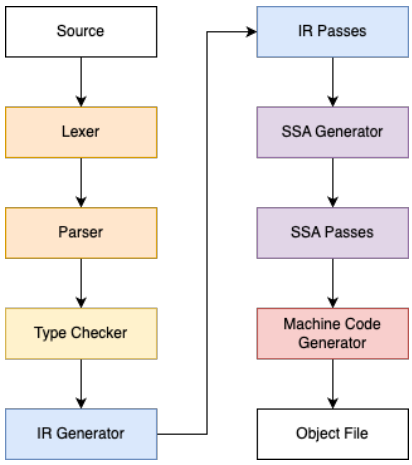


Figure 1: Different stages in the Go compiler

character by character from left to right until it reaches an EOF (`src/cmd/compile/internal/noder/noder.go`). While scanning a source file it tries to map characters to predefined tokens which can be found in `src/cmd/compile/internal/syntax/tokens.go`. The syntax parser consumes the tokens produced by the lexer, and based on grammar rules places the tokens in nodes at the correct place in the syntax tree [23].

2.2.2 AST-transformations: The next phase transforms the CST to an abstract syntax tree (AST). This transformation is not always required, as some compilers may have implicit CSTs which concurrently are transformed to ASTs. In the case of Go this explicit distinction has a historic reason, as the compiler was first written in C and only later transformed into Go code. A future rewrite may therefore merge the two processes into one [16]. An important distinction between CSTs and ASTs, and why ASTs are necessary, is that CSTs are a more formal mapping of the language grammar whereas ASTs are more simplified representations of the source code, this allows for easier and more efficient analysis and processing in later stages of the compiler [15]. We can see that in `src/cmd/compile/internal/gc/main.go` the compiler performs a multitude of optimizations which include function call inlining, dead code elimination, devirtualizing functions and escape analysis, using the AST representation [17].

2.2.3 SSA:. In this phase the AST intermediate representation (IR) is transformed into a static single assignment (SSA) IR. A program is in SSA form when a variable is assigned only once when looking at the source code [22], as well as the requirement that each variable needs to be defined before it can be used. SSAs are defined in function granularity and are self contained, this has the advantageous property that they can be independently optimized and compiled. The compiler performs multiple passes over the generates SSA code to further optimize the resulting output file. These optimizations can be platform agnostic in the case of TODO, but it can also be platform dependent in the case of lowering instructions to target architecture assembly instructions [7].

2.2.4 Machine code: The resulting SSA code is finally transformed into assembly in the final step of the compilation process. This code is will be emitted in the form of an object file, which is a container for assembly instructions which can be used, in turn, by the linker at the inter-package level.

3 PRELIMINARY PERFORMANCE ANALYSIS

Based on the aforementioned overview of the build process we analyze the different levels and stages to identify targets which may be improved using horizontal scaling.

3.1 Setup

As mentioned previously, analysis will be performed on version 1.21.6 of the Go compiler. During compilation we explicitly disable wrapping C code in Go interfaces by setting `CGO_ENABLED=0` before starting compilation [8], as this report focuses on Go and not C compiler performance. We measure compilation times on 3 real-world applications to find potential bottlenecks in realistic workloads:

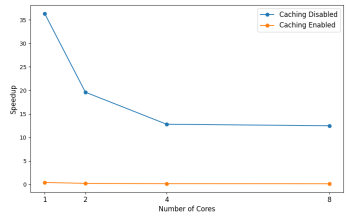
- `go-redis`, a type-safe client for Redis [9].
- `alist`, a file listing program which supports multiple storage backends [10].
- `osmedeus`, a workflow engine for offensive security [11].

The applications were picked to represent a diverse set of functionalities, as well as having a developed user-base. Each application is compiled 10 times, from which an average is derived to prevent outliers skewing the results. As we will be analyzing the performance of the local version of the Go compiler we use `GOMAXPROCS` to explicitly state the amount of processes which is allowed to be used [12]. We furthermore perform the benchmarks on a single node of the DAS-5, where the requested amount of processes will be equal to the maximum allowed `GOMAXPROCS`, where a node of the DAS-5 consists of 2 8-core CPUs with a frequency of 2.4GHz and 64GB of memory [3]. Before the build command is executed we ensure all required modules are fetched with `go mod download`, as the network speed of the Go proxy is not in the scope of this report, we thus assume for all performance metrics that the required modules are already present in the cache of the respective machine. To further enforce reproducibility `GOOS=linux` and `GOARCH=amd64` were set before executing the build command.

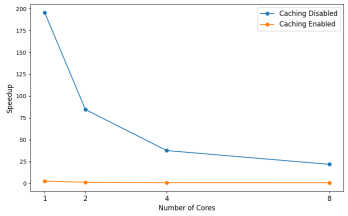
3.2 Results

3.3 Inter-package

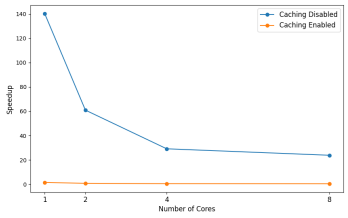
We first consider an analysis of the end-to-end timings and compiler behaviour. For the end-to-end timings we include compiler timings both with and without a cache available, disabling the cache is done by executing `go cache -clean` before every build. As the compilation times are significantly lower when caching is enabled, we can derive that it is an important aspect in the high compilation speeds of the Go compiler. Furthermore, we see in 2 that the compiler benefits from having multiple cores available, this is in line with expectations as parts of the Go compiler support parallel distribution of the incoming workload. It is of note that caching too benefits from having multiple cores available, as can more clearly be seen in 3. This is due to the fact that, even if no compilation



(a) `go-redis`

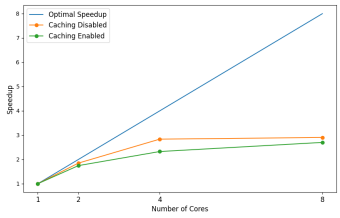


(b) `alist`

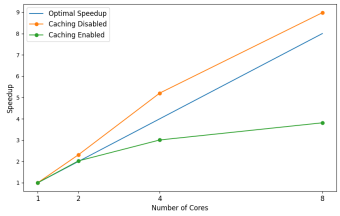


(c) `osmedeus`

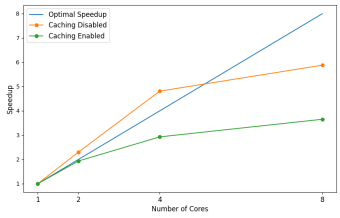
Figure 2: Compilation times for specified applications.



(a) `go-redis`



(b) `alist`



(c) `osmedeus`

Figure 3: Speedup for specified applications.

occurs, dependencies need to be resolved and fetched from the cache, which can also be done in parallel. We also see super linear speedups in 3b and 3c, the reason behind this becomes more clear when we look at the traces of the compilation process.

3.3.1 Traces. Traces can be inspected by supplying the build command with the `-debug-trace` flag, which takes an output path as argument. This produces a file in Trace Event Format [4], this can subsequently be analyzed by tools as Perfetto, which is a tool for performance instrumentation and trace analysis [2]. When comparing the behaviour of the Go compiler using a different amount of cores we see that certain blocking actions like `build runtime` take longer to complete when only having a single core available. This can be explained by the fact that the thread executing the action is also responsible for other parts of the compilation process, reducing the computational power available compared to when the action can be offloaded to another thread. We call the `build runtime` action blocking because as we can see from the traces in 5, other packages depend on it and must await its completion before they can be started. As the packages form a DAG of dependencies which must be compiled, blocking actions will become a relatively larger bottleneck when increasing the parallelisation of the compilation process. This also means that for inter-package compilation there is a maximum degree of parallelisation, as at some point no packages can be further divided over available cores as their dependencies have not yet been compiled. When we compare the traces of the different applications in 5 we see that the larger applications `alist` and `osmedeus` benefit more from inter-package parallelisation compared to `go-redis`. By making use of `actiongraph` [13], a tool to analyze the action DAG, we can analyze how application size affects parallelisation. By listing the actions we see that the `go-redis` DAG has 124 actions whereas the `alist` and `osmedeus` DAG have 771 and 397 actions respectively. A larger DAG generally results in a higher degree of average parallelisation, this can be confirmed by analyzing the DAG itself using the `graph` or `tree` command of `actiongraph`.

3.3.2 Caching. We revisit the topic of caching to touch upon some other behavioural aspects of the compiler. The timings considered here are either best case or worst case scenario, as we either consider being able to fully resolve the packages from the cache or not being able to solve any packages from the cache at all. During development cycles or when making use of CI/CD there is a higher likelihood that the performance will be somewhere in between either of the aforementioned cases. When looking at the traces in 5 we see that the first major blocking action (colored red) is building the Go runtime, for most workflows this compilation will only happen once per machine as the runtime most likely will not be changed by modifying the application making use of the runtime, given that the version of Go and the environment (compiler flags,

target architecture etc.) stay the same. The duration of compilation times furthermore depends on the depth of changes in the DAG, when a package is modified which is close to the end of the DAG most packages can be fetched from the cache. This is in contrast to when a package is modified close to the start of the DAG, in this case all subsequent dependent packages, which do not necessarily need to be modified themselves, will have to be recompiled.

3.4 Intra-package

3.5 Potential Improvements

When looking at the traces in 5 we see that for all applications the compiler is performing sequential work at the start of the compilation process when executing `PackagesAndErrors`. This may be a target for a cache implementation to improve setup and loading times.

3.5.1 Inter-package. When considering the performance results in 3 we see that for applications with many packages the trend indicates a further increase in core count will decrease compilation times further. This expectation is further enforced by looking at the traces 5b and 5c. For a significant portion of the compilation time the available threads are fully allocated and working on executing build actions, this indicates that the DAG has a larger average width than the currently available threads. Looking at 5a we see that at certain intervals the width of the DAG is larger than the available threads, but for a significant portion of the compilation time the DAG width is smaller than the available threads, which means a smaller performance gain when increasing parallelisation for applications with a smaller package count.

3.5.2 Intra-package.

4 DISTGO DESIGN AND IMPLEMENTATION

4.1 Architecture

4.1.1 The master node. is responsible for generating all the commands required for compilation on the head node. It organizes these commands into various dependency-based groups. Within each group, commands are independent of each other and can be run simultaneously on multiple nodes on DAS-5. Once organized, the master node dispatches these groups to the asynchronous task queue [14] named 'Compile_Group'.

4.1.2 The coordinator node. is to extract groups from the 'Compile_Group' task queue and break them down into single command tasks and send them to the 'Compile_Job' task queue. These tasks are then executed across multiple worker nodes. It manages the compilation process in a sequential manner, extracting each group one after the other. Additionally, the coordinator node is responsible for synchronizing the build result files across the worker nodes, ensuring a cohesive and efficient compilation process.

4.1.3 The worker node. operates on a DAS-5 node. Its ongoing function is to continuously retrieve jobs from the 'Compile_Job' task queue and execute them.

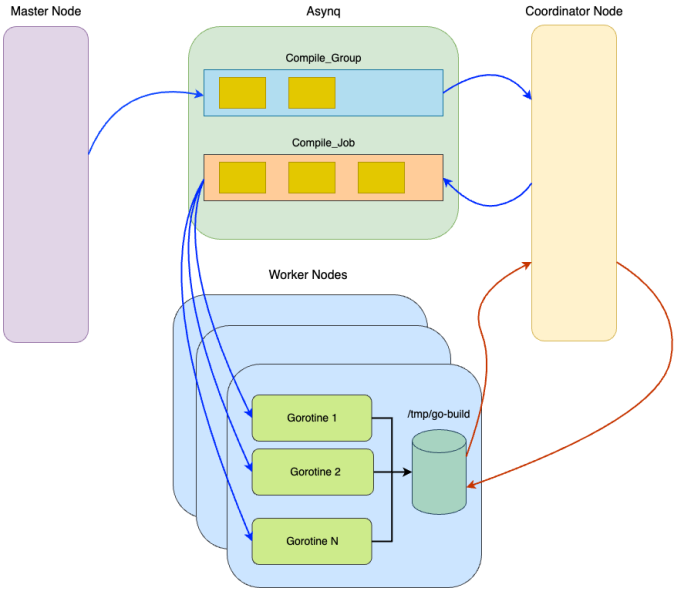


Figure 4: The architecture of distgo

5 PERFORMANCE ANALYSIS

5.1 Setup

The procedure outlined below follows the same environment setup as detailed in Section 3.1:

5.1.1 Cloning the DistGo Repository. Begin by cloning the DistGo repository from `https://github.com/badbubble/distgo`, as well as the specific project you wish to build.

5.1.2 Compiling the DistGo Project. Navigate to the DistGo directory using cd distgo and execute make. This process will generate three executable binaries: master, coordinator, and worker.

5.1.3 Configuring the System. Proceed to adjust all configuration files located in the `configs/` directory. This includes setting parameters like the Redis host.

5.1.4 Loading the DAS-5 Module. In the DAS-5 terminal, initiate the module with module load prun.

5.1.5 Deploying Worker Nodes. Deploy the worker nodes by executing `prun -v -NUMBER_OF_WORKERS -np NUMBER_OF_NODES ./worker` The `NUMBER_OF_WORKERS` and `NUMBER_OF_NODES` should be set according to your specific requirements.

5.1.6 Configuring the Coordinator. Update the configs/ coordinator.yaml file to include the IP addresses of the nodes. These IP addresses can be obtained from the initial output lines in step 5.

5.1.7 Deploying Coordinator Nodes. Start the coordinator by running ./coordinator.

5.1.8 Starting Compiling. Launch the master process with the command `./master -p PATH_TO_THE_PROJECT -m PATH_TO_THE_MAIN_FILE`. Monitor the coordinator's terminal to view the time consumed by these processes.

5.2 Results

5.3 Comparison

6 DISCUSSION

6.1 Analysing commands

The process of converting text into command-related data structures and sorting all commands currently takes approximately 1 to 2 seconds. This aspect of the project is not optimized, primarily due to excessive modifications made to the text and latency issues. Therefore, it is essential to consider a rewrite of the code segment responsible for this step. Streamlining this process would involve reducing the number of text modifications and addressing the latency problems to enhance the efficiency of command conversion and sorting.

6.2 Synchronizing files

it was observed that synchronizing files between the coordinator and workers consumed half of the total compilation time, which is deemed inefficient. In the current approach, the project involves synchronizing all files located in /tmp/go-build, a task that is notably resource-intensive. To enhance this process, it is suggested that the coordinator should conduct an analysis of the commands executed by the workers. This analysis aims to identify the autonomous files and dependencies involved. Subsequently, the improved method involves fetching only the autonomous files from the workers and sending solely the necessary dependencies to them, rather than synchronizing the entire directory. This targeted approach is expected to significantly reduce the time and resources required for file synchronization.

6.3 Job distribution

The job distribution phase in the current setup requires roughly 10 seconds, a duration that is contingent upon network performance and the efficiency of the machine running Redis. Given this context, both Redis and Asynq appear to be suboptimal choices for handling compilation tasks, primarily due to their considerable resource demands. To address this inefficiency, it is recommended to shift towards lighter technologies, such as remote procedure call (RPC). Implementing RPC could potentially streamline the job distribution process, as it is typically more lightweight and can offer better performance, especially in environments where network and machine capabilities are limiting factors.

6.4 Load imbalance

The current system exhibits inefficiencies in worker utilization during the compilation process, particularly when processing certain groups of tasks. For instance, we have the following group and 3 workers: group 1: [1, 2, 3], group 2: [4], group 3: [5, 6, 7], while all three workers are actively engaged when processing groups 1 and 3, only one worker is operational for group 2, leading to suboptimal resource utilization. To resolve this issue, two potential solutions are proposed:

6.4.1 Revise the Sorting Algorithm. The first approach involves reworking the sorting algorithm. The aim here is to restructure the distribution of tasks in a way that ensures all groups can be processed in parallel. By doing so, the workload can be more evenly distributed across all available workers, regardless of the group being processed. This adjustment should lead to a more balanced and efficient use of resources.

6.4.2 Parallelize Single Tasks. The second solution proposes parallelizing tasks within singular groups, such as group 2. Sometimes, groups like group 2 take a long time to process, causing the compilation to slow down. This method would involve dividing a single task into smaller, parallelizable components, allowing multiple workers to contribute simultaneously. However, this approach comes with potential drawbacks, such as increased complexity in task management and the possibility of introducing new issues, such as synchronization or dependency challenges.

7 CONCLUSION

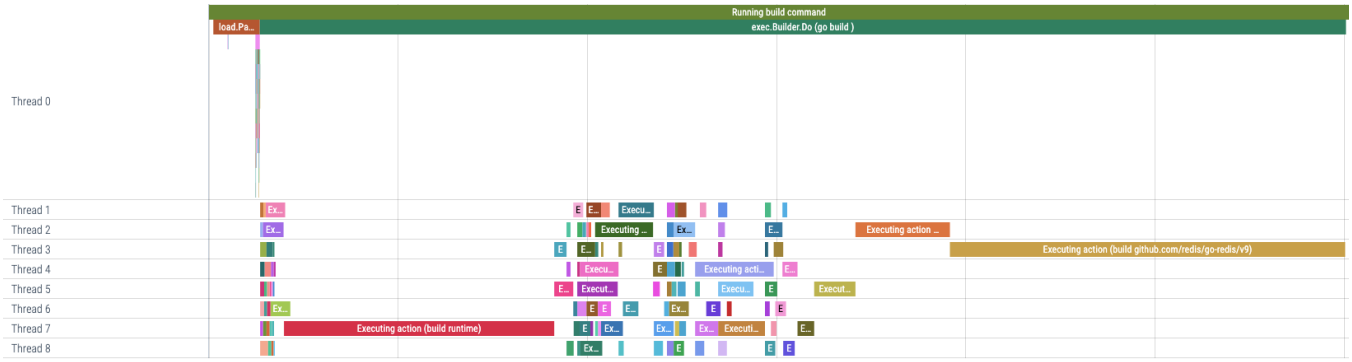
8 ACKNOWLEDGEMENTS

This report was written in conjunction with the use of ChatGPT. Its purpose was to serve as a feedback loop for writing, ideation and implementation. In the case of writing already written paragraphs were submitted to ChatGPT, which was then prompted to suggest writing style changes or list limitations in the paragraph. Its feedback was then analyzed, and if appropriate, (partially) applied. For ideation and implementation already considered approaches and technologies were submitted, following by prompts challenging or extending these considerations.

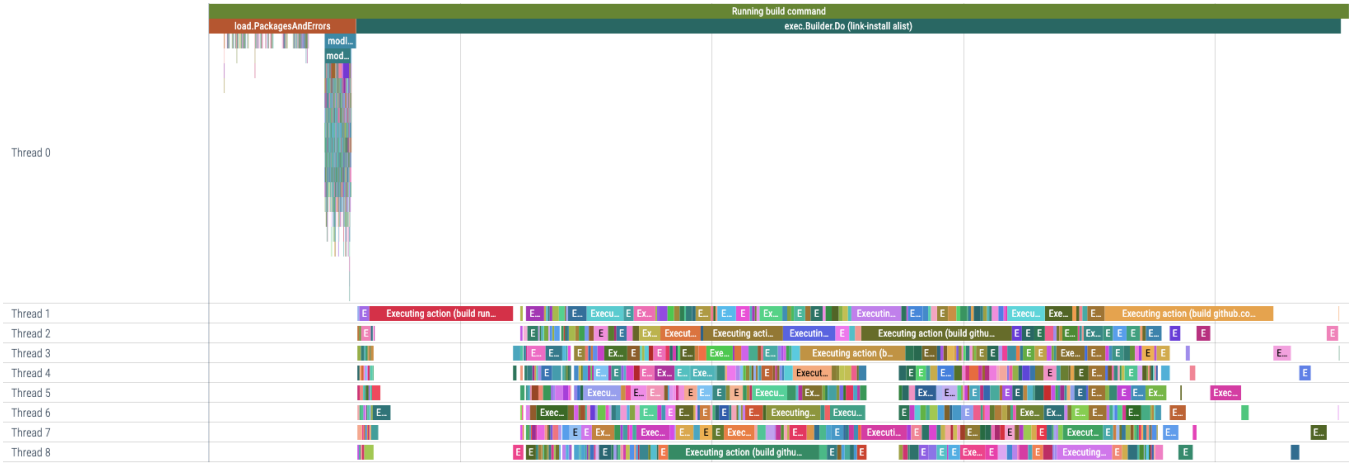
REFERENCES

[1] [n. d.]. https://github.com/golang/go/tree/go1.21.6
[2] [n. d.]. https://perfetto.dev/
[3] 2012. https://www.cs.vu.nl/das5/
[4] 2016. https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/edit#heading=h.yr4qxyxotyw
[5] 2024. https://pkg.go.dev/cmd/go@go1.21.6
[6] 2024. https://go.dev/src/cmd/compile/README
[7] 2024. https://go.dev/src/cmd/compile/internal/ssa/README
[8] 2024. https://pkg.go.dev/cmd/cgo
[9] 2024. https://github.com/redis/go-redis
[10] 2024. https://github.com/alist-org/alist
[11] 2024. https://github.com/j3ssie/osmedeus
[12] 2024. https://pkg.go.dev/runtime
[13] 2024. https://github.com/icio/actiongraph
[14] 2024. https://github.com/hibiken/asynq
[15] Eli Benderskys. 2009. Abstract vs. concrete syntax trees. https://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees
[16] Eli Benderskys. 2019. Go compiler internals: Adding a new statement to go - part 1. https://eli.thegreenplace.net/2019/go-compiler-internals-adding-a-new-statement-to-go-part-1/
[17] Jesus Espino. 2023. Understanding the go compiler - Jesus Espino. https://www.youtube.com/watch?v=qnmoAA0WRgE&t=391s
[18] A. Harwood, M. Read, and Gayashan Amarasinghe. 2020. Dragon: A Lightweight, High Performance Distributed Stream Processing Engine. *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)* (2020), 1344–1351. https://doi.org/10.1109/ICDCS47774.2020.00177
[19] Daniel Marti. 2022. Deep dive into go's build system. https://youtu.be/sTXc_JxmvV0?si=6PbsgUmgCKjoHkN4
[20] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and I. Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. *ArXiv* abs/1712.05889 (2017).
[21] Rob Pike. [n. d.]. Go at google: Language design in the service of software engineering. https://talks.golang.org/2012/splash.article#TOC_5.
[22] Keith Randall. 2017. GopherCon 2017: Keith Randall - generating better machine code with SSA. https://www.youtube.com/watch?v=uTMvKVma5ms
[23] book travel. 2022. https://segmentfault.com/a/1190000041213877

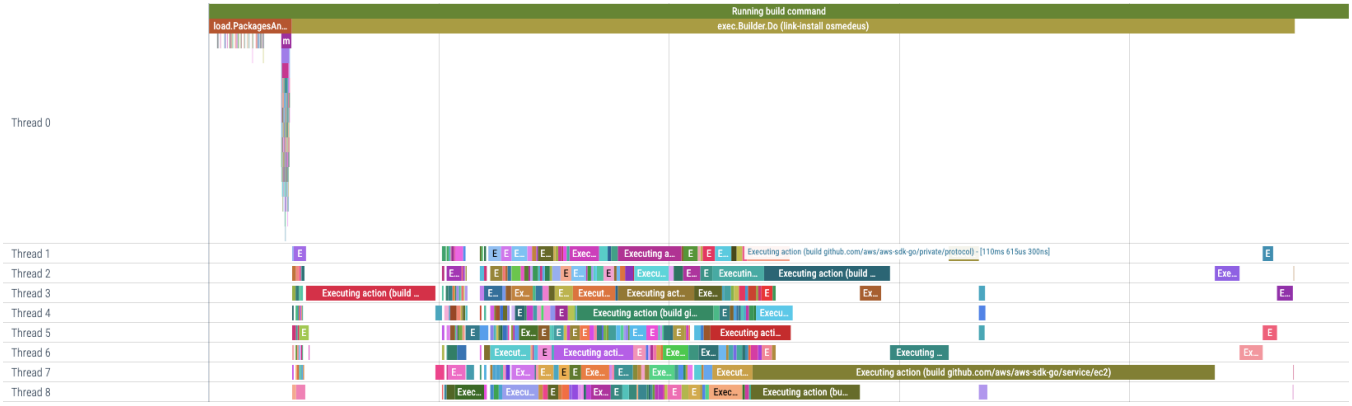
9 APPENDIX



(a) `go-redis`



(b) `alist`



(c) `osmedeus`

Figure 5: Traces for specified applications during the compilation process when using 8 threads.