Stroll: a build system that doesn't require a plan

Andrey Mokhov
Jane Street
London, United Kingdom

ABSTRACT

This paper presents Stroll — a build system that doesn't require the user to specify inputs and outputs of individual build tasks, or provide a build plan in any other way. This makes Stroll very convenient to use, but comes at the cost of slowing down the very first build where Stroll needs to infer the build plan on its own. The inference works by tracking file accesses of individual build tasks and restarting the tasks as needed to ensure the correctness of the final build result.

The paper describes key ideas behind the implementation of Stroll, and positions it in the landscape of existing build systems. We also quantify the cost of inferring the build plan during the initial build.

CCS CONCEPTS

• **Software and its engineering**; • **Mathematics of computing**;

KEYWORDS

build systems, functional programming, algorithms

ACM Reference Format:
Andrey Mokhov. 2021. Stroll: a build system that doesn't require a plan. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Most build systems require the user to specify both *build tasks* as well as *dependencies* between the tasks. The latter essentially provides the build system with a "plan": by building the tasks in an order that respects the dependencies, the build system can guarantee that no task is executed more than once [Mokhov et al. 2020].

Describing dependencies between tasks can be problematic for two reasons:

- Providing an accurate description of dependencies, and then keeping the description up to date as build tasks evolve, is difficult and is often a source of frustration and subtle correctness and performance bugs [Spall et al. 2020].
- To describe dependencies, one needs a suitable domain-specific language, and indeed (almost) every build system comes with its own: Make [Feldman 1979] uses makefiles, Bazel [Google 2016] uses a Python-inspired language called Starlark [Bazel Team 2018], Shake [Mitchell 2012] uses a Haskell EDSL, Dune [Jane Street 2018] uses a combination of OCaml and a higher-level S-expression based configuration language, etc. While learning a new task description language is not a big deal, migrating an existing build system to a new language may take years[1].

Stroll takes a radically different approach. Given a collection of build tasks, it treats them as black boxes and discovers dependencies between them by executing the tasks and tracking their file accesses. This approach is not optimal in the sense that a task may fail because one of its dependencies has not yet been built. A task may therefore need to be restarted multiple times until all of its dependencies have been discovered and brought up to date. In the end, Stroll will learn the complete and accurate dependency graph and will store it to speed up future builds.

At the first glance, this approach may seem hopelessly slow. As we show in this paper, in the worst case, the number of restarts that Stroll's algorithm needs to perform is linear with respect to the size of the dependency graph. Furthermore, the restarts are not on the critical path, which means one can recover performance by giving Stroll a sufficient number of parallel workers. We also show that a simpler version of the problem requires at most 2x work compared to the optimum.

Stroll was inspired by Fabricate [Hoyt et al. 2009] that also tracks file accesses to automatically compute accurate dependencies. Fabricate itself was preceded by Memoize [McCloskey 2008] and succeeded by Rattle [Spall et al. 2020]. Unlike Stroll, all these build systems require the user to provide a build plan by listing the tasks in a topological order. Stroll takes the idea of file access tracking to the limit and doesn't require any plan, thus occupying a unique point in the design space of build systems.

Note for IFL 2021 reviewers

This is a last-minute submission that only describes two key algorithms used in Stroll, to make it possible for the reviewers to evaluate the performance claims. If this submission is accepted for a presentation at IFL 2021, the presentation will include a high-level Haskell model of Stroll, building on the modelling framework from [Mokhov et al. 2020]. The final version of the paper will include: (i) a Stroll model, elaborated further to highlight interesting aspects of Stroll's implementation (Stroll is written in Haskell); (ii) illustrated examples taken from the original blog post about Stroll [Mokhov 2019], clarifying how Stroll discovers the full dependency graph by executing and restarting "black box" tasks; and (iii) proofs of correctness and efficiency of Stroll's build algorithms.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] The author was involved in two such migrations and both are still under way after investing multiple man-years; the first one – migrating GHC's build system from Make to Shake [Mokhov et al. 2016] – started in 2014!

2 STROLL'S MAIN ALGORITHM

Stroll uses a (slightly optimised and therefore less obviously correct) version of the following algorithm.

- For each previously executed task x, store a *trace* T_x and a *status* S_x. The trace lists all inputs and outputs of the task recorded during its last execution. The status is either *success* or *failure*, depending on the last exit code.
- A task x is *complete* if $S_x = success$ and all inputs in T_x are complete. If one of the inputs in T_x is not complete, the task is *blocked* (on the corresponding input).
- Execute tasks in parallel *rounds*. In each round, execute all tasks that are not complete and not blocked. (Note that in the first round all tasks are executed in parallel.)
- Terminate as soon as the build target requested by the user has been produced, and the task that produced it is complete.
- Report an error if the current execution round is empty: the target cannot be built (e.g., due to a cyclic dependency).

Claim: For a dependency graph with n tasks and m dependency edges, there will be at most n rounds and at most m task restarts.

Proof sketch: Our first observation is that if a task gets executed in a round R and does not become complete, then it will need to be restarted at some later round $R' > R$, when the dependency that currently blocks it becomes complete. A restart therefore requires at least one dependency edge switching from blocked to complete, and since there are m edges overall, there will be at most m restarts. The second observation is that for an edge to switch from blocked to complete, at least one task must become complete. Therefore, there cannot be more than n rounds, since every round is non-empty, which requires at least one task to become complete. In fact, this bound can be strengthened: there will be as many rounds as the number of tasks in the longest dependency chain.

3 BUILDING TASKS WITH KNOWN OUTPUTS

While Stroll doesn't require any information about tasks, providing *some* information may help to significantly speed up the first build. For example, if there is a mapping from outputs to the corresponding tasks, then it is possible to bound the number of restarts by just n for a build graph with n tasks and m dependencies.

Consider the following algorithm.

- Execute the task x corresponding to a requested build target.
- If the task doesn't get blocked, terminate the build. If the task is complete, then the target has been successfully built; otherwise, it can't be built (x must have failed with an error).
- If x gets blocked on another task y, switch to building y. If that gets blocked too, keep building dependencies in the depth-first manner. If there are no cycles, a "leaf" task with no dependencies will eventually be reached, and the previously blocked task will be restarted. Continue until all tasks blocking the "root" x have been built.

Claim: The algorithm does at most n task restarts.

Proof sketch: We can count the number of restarts by counting the number of blocking edges discovered by the above algorithm, since every time a task is blocked, we recurse to build the blocking dependency and then restart the task. Here is a simple observation: each task can be a blocking dependency at most once. This holds because we build tasks in the depth-first order: when we "enter" a task, we never "leave" it until its whole subtree is complete, and so only the very first incoming edge will be blocking. Since there are n tasks and each has at most one incoming blocking edge, there will at most n blocking edges, and hence at most n restarts.

REFERENCES

Bazel Team. 2018. Starlark Language. (2018). https://docs.bazel.build/versions/main/skylark/language.html.

Stuart I Feldman. 1979. Make—A program for maintaining computer programs. *Software: Practice and experience* 9, 4 (1979), 255–265.

Google. 2016. Bazel. (2016). http://bazel.io/.

Berwyn Hoyt, Bryan Hoyt, and Ben Hoyt. 2009. Fabricate: The better build tool. (2009). https://github.com/SimonAlfie/fabricate.

Jane Street. 2018. Dune: A composable build system. (2018). https://dune.build/.

Bill McCloskey. 2008. Memoize. (2008). https://github.com/kgaughan/memoize.py.

Neil Mitchell. 2012. Shake before building: Replacing Make with Haskell. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 55–66.

Andrey Mokhov. 2019. Stroll: an experimental build system. (2019). https://blogs.ncl.ac.uk/andreymokhov/stroll/.

Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2020. Build systems à la carte: Theory and practice. *Journal of Functional Programming* 30 (2020).

Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. 2016. Non-recursive Make Considered Harmful: Build Systems at Scale. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, 170–181.

Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. 2020. Build Scripts with Perfect Dependencies. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 169 (2020), 28 pages. https://doi.org/10.1145/3428237