Correct, Efficient and Tailored: The Future of Build Systems

Guillaume Maudoux and Kim Mens
Université catholique de Louvain

***Abstract*—Build systems, also known as build automation tools, are used in every non-trivial software project. They contain the knowledge of how software is built, and provide tools to get it actually built as fast as possible. While they are central for day-to-day productivity, they sometimes fail to deliver their promise of being correct, efficient and tailored. The situation gets even worse when used with huge code bases and fast-paced continuous integration pipelines. In this paper we survey state of the art techniques and algorithms that relegate the occasional inconsistent builds, slow execution times and boilerplate makefiles to another age.**

BUILD systems are used to collect all the commands and all the knowledge necessary to build an application. While it simplifies maintenance of the build steps, the real gain (as opposed to a collection of shell scripts for example) resides in the single optimization called incremental builds. Because the build system knows the entire dependency graph, it is able to run only the few required commands to update the build. On non-trivial projects, the potential speed-up is so huge that no-one could live without it. In particular, it is mostly used locally during development where the edit-compile-test loop is most critical. Developers tend to test frequently very small edits. Rebuilding bigger projects completely every time is therefore not an option.

Behind the scenes, build systems all maintain a dependency graph, and schedule the execution of build steps. But each software project want a tailored build system that understands the conventions of their programming language or underlying framework. This explains the plethora of different build systems. In this situation, good ideas found in a given implementation are not propagated to others, and build systems keep using the same old algorithms despite the existing optimizations. In this article, we survey various existing improvements scattered across the different implementations. We grouped them with respect to the three main aspects of build systems: Correctness, efficiency and specificity to their domain.

I. Correctness

A build system is deemed correct when it produces results indistinguishable from a clean build from clean sources. Whatever magic optimisation happens inside the tool must not change the build products. In particular, users should never have to force a full rebuild, nor start the build in a clean tree to get the expected outputs. There exists different definitions of correctness [1], [2], but we prefer the intuitive notion that "any invocation of the build system is equivalent to a clean build".

This is not a trivial requirement. Every developer encountered a situation where the build was broken beyond understanding.

State of the art techniques to enforce correct builds include the persisting build systems state, advanced dependency enforcement and validation as well as redundant checking of build products.

A. Persistent state

To achieve correct builds in all cases, a build system must keep track of the files that were built [1]. For example, this allows to detect when a build product was manually modified and error out. Silently overwriting the file drops user modifications, but keeping it as-is may produce artefacts significantly diverging from the source. Tools like `make` start from scratch on every invocation, with no a priori information. They are therefore bound to miss the above kind of changes.

B. Isolating build steps

To ensure correct incremental builds, build systems need to collect information about dependencies between build steps. A change to an untracked dependency will not trigger the required rebuild for the related steps, which in turn could invalidate the build.

For most build systems, dependencies are provided or inferred statically before the build. In some rare cases, dependencies are collected dynamically during execution [2]. In both cases, it is important to track how a build step accesses it's environment to either enforce the declared dependencies, or to collect them all.

Progress on namespacing (also called sandboxing) techniques to isolate processes enables to perform this enforcement [3]. This is represented in Figure 1. Any access to undeclared resources must be forbidden, because it may invalidate current and future builds. Dependencies unknown to the build system will impede its ability to run the steps in the right sequence, or to detect that a rebuild is needed. Declared but unused dependencies are not as serious but should be eliminated as they reduce the overall efficiency of the build.

C. Managing the environment

Another threat to correctness is the environment where compilations happen. We take the simple definition that the environment is formed by everything outside the source tree and the internal state of the build system (including compilation results). The environment is therefore formed by the

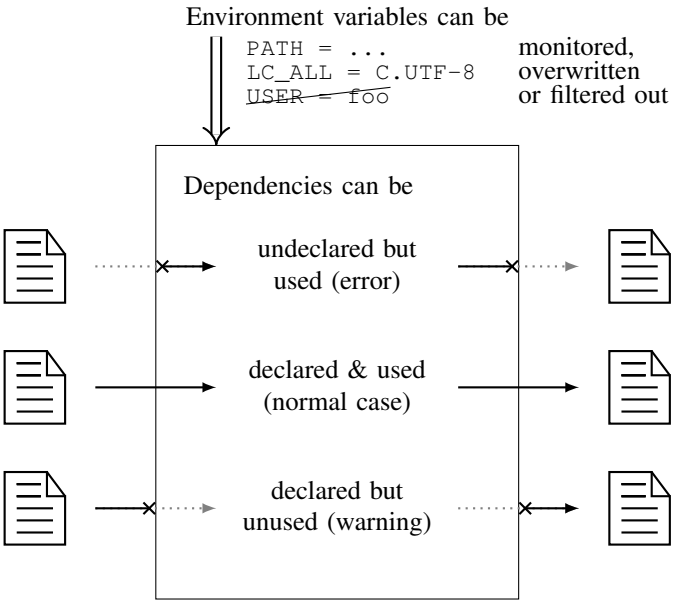

Figure 1. Proper isolation of build steps allow to detect discrepancies between declared and used dependencies. Such an isolation is also instrumental to provide reproducible builds. This is only possible thanks to recent, lightweight sanboxing techniques (also know as namespaces). These allow to capture all the effects of a build step and cache them in a efficient, reusable way. This is discussed in Sections I-B, I-C and II-C.

CPU Architecture, the version of the tools involved in the compilation, the environment variables, the operating system, etc. As a build can be impacted by such factors, build systems should monitor, hide or otherwise control these parameters. For example, recent build systems clear unknown environment variables and set some others to predefined universal values as shown in Figure 1.

The current state of the art is to manage the environment outside and separately from the build system. Nowadays, most CI builds happen in `docker`, `vagrant` or other containers to provide the same environment across builds. This practice also spreads to local builds of developers, because it avoids surprises when sending patches to CI and makes it easy to set-up the development environment thanks to the existing containers.

Some build systems, like `tup` for example, allow to track the whole environment. This allows to detect changes to the c compiler or any external tool or library. `Tup` does that by tracking all the accessed files outside the source tree in the same fashion as it tracks source files.

D. Validating results

The only way to check that a given build is correct is to compare it with a clean build. In practice, build systems could run the same build steps multiple times and compare the results. While correctness requires only equivalent builds, getting identical results is the simplest way to ensure build equivalence.

While it is not a common practise elsewhere, some package managers strive to provide reproducible packages. This guarantees that a binary package consistently reflects its source tree. Efforts made by open-source projects are concentrated by the reproducible-builds.org initiative. Their goal is to allow anyone to reproduce any package, as reliably as possible.

Correctness is a matter of trust in the build system. When developers are confident enough, they rely blindly on their build system and concentrate on other aspects of their tasks.

II. Efficiency

Correctness is often overlooked in favour of efficiency. There are a lot of speed comparisons amongst build systems, and for a good reason. A benchmark comparing build systems on a C++ project showed time ratios as large as 60 between the tools [4]. One tool took one second while another took a full minute.

The speed of incremental builds is crucial to get a short edit-compile-test loop, and therefore to spare developer's time. When analysing the Travis CI platform, it was shown that "[t]he main factor for delayed feedback from test runs is the time required to execute the build [5]." That being said, getting efficient incremental builds in distributed CI clusters is a complex challenge [6].

An optimal build system should spend as little time as possible figuring out which steps needs to run. It should also run only the steps that are really required, and ensure there are as few as possible. This is made possible by efficiently tracking and propagating change, as well as performing advanced caching.

A. Tracking changes

Detecting changes between builds allows to know exactly what needs to be rebuilt. To do this, make relies on timestamps. This is inefficient because it requires to build the whole dependency graph, and read the timestamps of every source file. This is an $O(n)$ algorithm with n being the number of files. For big software projects, this is a big issue.

Many build systems turned to `inotify` (or simmilar APIs) to receive notifications on every file change. This technique catches can also be used to trigger incremental builds immediately. To work properly it requires a background monitor process. When that process (re)starts, it still needs to check all the files.

Efficiently tracking changes can have a huge impact on build time, especially when there is very little to do to update the build. In huge code bases, this $O(1)$ vs $O(n)$ complexity speeds builds by many orders of magnitude [7]. This benefit is however only possible if the build system does not walk the entire dependency tree on each iteration. To avoid that, we need the efficient encoding of the dependency graph described presented in the next section.

B. Propagating changes

To detect efficiently what steps need to be executed again, the dependency tree (which is technically a DAG) must be stored as reversed arborescence, as depicted in Figure 2. With

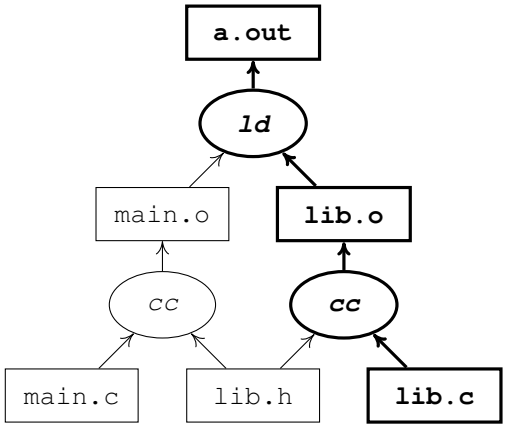

Figure 2. The dependency graph is the common feature of all the build systems. It has a strict alternation of data and processes. The arrows pointing towards the root allow a fast lookup of updates triggered by source changes. In this example, a modification to `lib.c` requires to update the `lib.o` and `a.out` outputs. This is discussed in section II-B.

that data structure modifications propagate naturally upward in the DAG, in an expected $O(\log(n))$ time [1]. This is again a huge improvement over the classical $O(n)$ tree walk that builds the full graph on each invocation. This optimisation, together with efficient change detection described above makes build system feel instantaneous. Only the time taken by the build steps remains significant.

C. Caching and sharing builds

While incremental builds allow to reuse products of the last build, they are less useful after switching to a new branch, or around merge commits which both introduce many changes at once. It is also difficult to share these products with other users or distant machines. Products caching is the technique used to address these issues.

Caching was first popularised with `ccache`. The tool acts as a drop-in replacement for gcc, but caches compilation resuts in an external directory. When equivalent calls are made to the compiler, ccache fetches the result from the cache, avoiding redundant computations. If the cache is big enough, building different branches remains efficient.

More recently, Mozilla implemented `sccache`, a shared compiler cache, to get caching to work on a cluster of nodes. `Sccache` publishes build products on a central repository available to all the CI workers. This allows CI to access incremental builds intead of clean builds often found there [6]. With `sccache`, Mozilla was able to halve compilation time on its CI cloud workers [8].

Generic caching is much more difficult to achieve, because build steps can perform a huge range of modifications on their environment. Each of these must be either captured of prohibited to ensure that the cache contains exhaustive and valid substitutes to the corresponding build steps. That kind of shared caching is implemented in `gradle`, `buck` and `bazel`. These implementations do not provide complete isolations, and therefore cannot guarantee both correctness and efficiency at the same time. For example, `gradle` cannot correctly cache parallel build steps that write to the same directory. Lightweight isolation and virtualization technologies would help to improve these issues.

As a side note, caching builds also allows individual developers to improve local performances. In fact, caching blurs the difference between local and CI builds.

With good data structure and algorithms, as well as with correct shared caching, software project have improved the speed of their daily builds by some orders of magnitude.

III. Usability

While build systems use similar algorithms behind the scene, they distinguish themselves by their configuration language, the intended uses they support and their community. This is mostly visible in the set of modules and ad-hoc configuration provided with each of them. The advantages of tailoring drive the creation of new build systems. Either when older ones do not support new technologies, or when they require too much boilerplate. While generic, programmable build systems can theoretically build any software, developers tend to prefer tools tailored for their specific language, or with included support for the framework they use.

Build systems mature over time, and collect tips and tricks to handle corner cases for software they build. For example, `bazel` has an in-depth understanding of C compilers options and manages the subtle variations required by different implementations and platforms. The `autotools` tool suite has gained extensive support for building applications on Unix platforms. Picking the right tool therefore avoids a lot of boilerplate to developers and maximises portability.

There is a huge galaxy of different build systems, and all of these could benefit from better performances, and should strive to ensure the correctness of their builds. As these improvements relate to the backend, it should be possible to focus efforts on shared backends, while retaining the rich ecosystem of build system frontends. Alternatively, they could interoperate nicely together, providing their in-depth knowledge to other tools.

IV. Conclusion

We first described existing techniques to guarantee correct incremental builds. We also explained the algorithms to execute incremental (or cached) builds in the blink of an eye. Finally, we outlined the huge set of ad-hoc frontends for every possible use case.

Because build systems share the same core algorithms, these improvements can be easily ported to all of them. This would not sacrifice the embedded domain knowledge that makes the specificity of each implementation. This sketches an exciting future where build systems would no more be in the way. No more `rm -rf`, no more builds from scratch and no more idle waiting on build products. Only sound builds at high performance!

REFERENCES

[1] M. Shal. (2009) Build system rules and algorithms. [Online]. Available: http://gittup.org/tup/build_system_rules_and_algorithms.pdf

[2] S. Erdweg, M. Lichter, and M. Weiel, "A sound and optimal incremental build system with dynamic dependencies," *SIGPLAN Not.*, vol. 50, no. 10, pp. 89–106, Oct. 2015.

[3] M. Shal. (2014) Clobber builds part 2 - fixing missing dependencies. [Online]. Available: http://gittup.org/blog/2014/05/7-clobber-builds-part-2---fixing-missing-dependencies/

[4] N. Llopis. (2015) The quest for the perfect build system (part 2). [Online]. Available: http://gamesfromwithin.com/the-quest-for-the-perfect-build-system-part-2

[5] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17, 2017, pp. 356–367. [Online]. Available: https://doi.org/10.1109/MSR.2017.62

[6] G. Maudoux and K. Mens, "Bringing incremental builds to continuous integration," Jun. 2017, unpublished conference paper. [Online]. Available: sattose.org/2017:schedule

[7] M. Shal. Make vs tup. [Online]. Available: http://gittup.org/tup/make_vs_tup.html

[8] M. Hommey. Analyzing shared cache on try. [Online]. Available: https://glandium.org/blog/?p=3201

Guillaume Maudoux is a PhD student at Université catholique de Louvain (Belgium) since 2013. Member of the RELEASeD research laboratory on software evolution, his research interests cover the whole release engineering pipeline, with a preference for build systems, continuous integration and preferably both at the same time. Contact him at guillaume.maudoux@uclouvain.be.

Kim Mens is a full-time professor at Université catholique de Louvain, where he leads the RELEASeD research laboratory on software evolution and software development technology. His research interests include software development, maintenance, reuse and evolution, as well as programming language engineering. Mens received a PhD in computer science from Vrije Universiteit Brussel. Contact him at kim.mens@uclouvain.be.