Bringing Incremental Builds to Continuous Integration

Guillaume Maudoux
Département d'Ingénierie Informatique,
Université catholique de Louvain,
Louvain-la-Neuve, Belgium
Email: guillaume.maudoux@uclouvain.be

Kim Mens
Département d'Ingénierie Informatique,
Université catholique de Louvain,
Louvain-la-Neuve, Belgium
Email: kim.mens@uclouvain.be

***Abstract*—Incremental builds are commonly used to speed up the edit-compile-test loop during program development. By updating only the required parts of a system, build systems can shorten the compilation phase by orders of magnitude. While this technique is commonly used for local builds, it is seldom enabled during continuous integration. Current build system do not offer strong correctness properties on incremental builds. In continuous integration setups, it is therefore difficult to achieve both correctness and efficiency simultaneously. Facing this choice, release engineers tend to favor correct builds over optimized builds.**

In this article, we show that it is possible to obtain both incremental and correct builds. We start by showing that incremental builds are a desirable optimization in continuous integration environments. Different reasons that prevent release engineers to enable incremental builds in practice are discussed. From these, we derive requirements to be met by future build systems to support incremental continuous integration. Whenever possible, we illustrate shortcomings of build systems with insights from current research and industry efforts in new build systems. We also list existing projects that could be combined into a complete tool supporting efficient and correct incremental compilation in continuous integration environments. Ultimately, this paper defines a new research direction at the intersection of build systems and continuous integration.

I. Context

The field of release engineering is concerned with the pipeline of techniques and operations between isolated developers writing source code and end users enjoying released applications. Release engineers maintain and optimize this pipeline from end to end. A classical release engineering pipeline [1], as depicted on Figure 1, starts with source code management (A). When commits are ready, they enter continuous integration (B) where software is built and tested in different stages of increasing complexity and duration. To bring these modifications to end-users, code needs to be deployed (D) and released (E). This pipeline is all about tooling, and a particular importance is attached to the build system (C) used by developers to test their code locally and by continuous integration to both produce artifacts and drive tests. As pressure increases to get frequent releases and shorter development cycles, release engineers try to optimize this pipeline at each stage. In particular, the goal of continuous integration is to have short build times and fast test feedback to developers. Optimizing the edit-compile-test loop shortens developers' critical paths and accelerates software evolution. A recent study on Travis CI [2] showed that " [t]he main factor for delayed feedback from test runs is the time required to execute the build ".

Incremental compilation is the technique by which a build system can efficiently update previous build artifacts. With sufficient information about dependencies between build steps, it is possible to tell which steps are impacted by the updated sources and run only these to generate correct build artifacts. The other optimization based on build task dependencies is parallel execution of independent tasks. These two features explain why developers use build systems despite the added hassle of configuring them. However, during continuous integration (CI) software is generally built from clean sources, making incremental builds impossible.

This paper is motivated by the need to understand why incremental builds are considered essential to developers on their own machines but are seldom used in continuous integration environments where they could seemingly bring the same outstanding advantages. Shunning incremental builds, release engineers voluntarily lose the performance gain of updating only the required build products. We will show that release engineers prefer correctness over optimization, and that it is not so trivial to scale incremental compilation to CI environments. Because continuous integration is mostly automated, release engineers trade slower compilations against better quality guarantees on build products. Computer time is quite cheap, and tracking build issues is time consuming. We will see that in that context their decision makes sense.

But decreasing the edit-compile-test loop time is a pressing concern of most organizations. Developers waiting for CI test results start to work on unrelated issues. When the compilation finishes, developers need to switch context to understand test failures on their previous issue. With very slow compilations, developers may need to juggle with three issues at the same time, losing more concentration to context switches. Some developers may prefer to wait for CI results instead of switching contexts too often, wasting even more time.

That being said, there are other challenges involved in enabling incremental builds on continuous integration servers. For example, builds are distributed on a cluster of workers. Workers may not be assigned to consecutive builds, making incremental builds less useful.

In this article, we show that wasted time on compilation is significant, and that there are solutions to make build systems correct. We outline the features required for the next generation of build system to enable incremental builds in continuous integration setups while guaranteeing correctness in all cases.

II. Incremental build experiment

We claim that incremental builds would benefit CI environments. Were it not the case, there would be no reason to update CI tools to support it. The experiment described in this section illustrates how much resources and developer time could be spared with such an optimization.

To assert the usefulness of incremental builds in continuous integration, we reproduced the workload of an integration server. A list of consecutive commits are build incrementally from a complete build of their parent. The build time for the first commit serves as an estimate of the duration of a full rebuild from clean sources. Comparing duration of incremental builds with full rebuilds gives an idea of the potential gain. This experiment relies on simplifying assumptions detailed below.

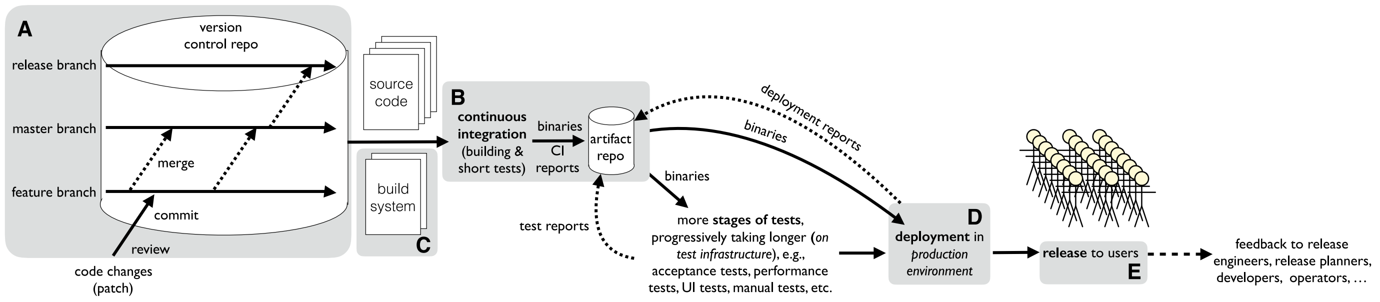
A
version control repo
release branch
master branch
feature branch
merge
commit
review
code changes (patch)
source code
build system
C
B
continuous integration (building & short tests)
binaries
CI reports
artifact repo
deployment reports
binaries
binaries
test reports
more stages of tests, progressively taking longer (on test infrastructure), e.g., acceptance tests, performance tests, UI tests, manual tests, etc.
D
deployment in production environment
release to users
E
feedback to release engineers, release planners, developers, operators, …

Fig. 1. An overview of the release engineering pipeline, borrowed and freely simplified from [1]

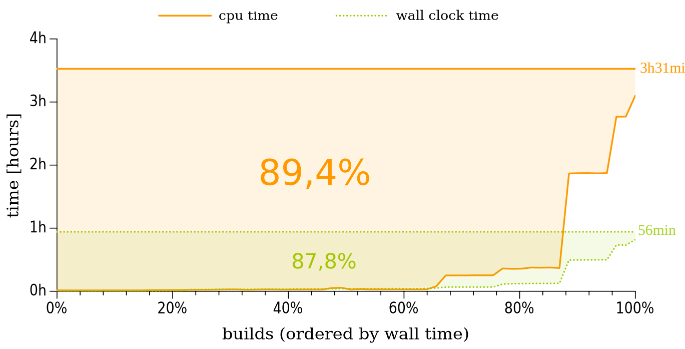

Fig. 2. Empirical distribution function of incremental build times and the potential gain they introduce compared to full rebuilds.

In practice, we analyzed the last 63 commits[1] on mozilla-inbound[2]. As its name suggests, this repository belongs to the Mozilla foundation. It contains mainly the source code of the web browser Firefox. mozilla-inbound accepts briefly tested patches and gets merged daily into mozilla-central if the branch passes extensive tests. It is the normal landing point for non-critical changes [5]. In their analysis of Mozilla's patch management infrastructure, Rodrigo Souza et al. [7] have show that continuous integration is key to early patch backout and stable builds on stable branches.

For each of the 62 most recent commits, we have launched an incremental build, the 63rd was used to bootstrap the incremental build chain, and it's build time was used to estimate the duration of a full rebuild. The results are presented in Figure 2, in an empirical cumulated density function (CDF) where the 62 builds are ordered by build time. The horizontal line represents the time required for a full rebuild, and the other one represents the time for an incremental build of a given commit from it's parent build. The coloured area between these lines represents the potential gain in time due to incremental builds. Two different metrics are reported. The wall time corresponds to the real time elapsed between the start and the end of the incremental build. CPU time on the other hand is the cumulative time spent on the task by all the CPU's. As the experiment was run on an Intel i5 with two cores and four threads, the CPU time cannot be more than four times the wall time. CPU time can be seen as the expected duration of a sequential build on the same machine.

Of course, this experiment is not perfect. From the number of commits analyzed, it is difficult to draw meaningful conclusions. We also conducted two fast experiments on the much smaller i3 project[3] which exposed a similar distribution of the build times. We can expect that many repositories follow the same pattern, following the intuition that most changes are small and local and very few are spanning the entire project. We made the assumption that the duration of a full rebuild is stable across patches. The duration of the only full-rebuild is represented by the horizontal line on Figure 2. It seems reasonable to think that big, longstanding projects like Firefox have little variations in their full rebuild duration, but this hypothesis remains to be checked. Also, all the times were sampled only once. Displaying the results as a CDF provides some smoothing, but we cannot provide an estimate of the variability of build times. This should also be addressed with more experiments. Finally, we assume that it is possible to build each commit incrementally from its parent. As we will see in the next section, this is not trivial when builds are spread on a cluster.

These limitations of our initial experiment do not invalidated its main conclusion, however. A closer analysis of the Firefox case shows that when there is nothing to rebuild, an incremental build takes 29 seconds +/-1 (wall clock time). This is 100 times faster than a full rebuild, and happened for 10 builds out of 61. The median case spent two minutes and 23 seconds, which is still more than 20 times faster than a full rebuild. This means that for a majority of builds, compilation time can be reduced by one and sometimes two orders of magnitude.

The graph shows that 65% of the commits have little impact on the build. For the first 20% there is no impact at all. The cost of an incremental build in that case is minimal. It is the time needed by the build system to detect that nothing needs to be done. Such commits are more frequent than one would expect. A closer look at their contents shows documentation changes, but also changes to the test suite. For these changes in particular, rebuilding Firefox from scratch seems a real waste of time. After these commits follows a chunk of minor commits. They change files with low impact on the final Firefox executable. They contain for example .js files that are bundled as-is in the final data archive. Incremental builds need only rebuild the said archive to complete. The right part of the curve contains average-impact changes (.c files or their equivalent) requiring some rebuild and high-impact changes (.h files or equivalent) that trigger the recompilation of complete sub-components of the project. Very high impact changes include changes to the build definition itself (change in compiler flags), and mass edits of source files (update of copyright headers for example). Such big changes where not included in our test set.

The potential gain of achieving correct incremental builds during CI can be represented on the graph as the area between the incre-

[1]`hg log -r341198:341136`
[2]https://hg.mozilla.org/integration/mozilla-inbound/
[3]https://i3wm.org/

mental build time (CDF) and the time required for a full rebuild represented by the horizontal line of corresponding color. When applied to CPU time, this gives the amount of wasted CPU time. During that time, CI workers are building products that are known to be the same as before as per the build specification. Applied to the wall time, this gives the useless latency introduced in the edit-compile-test loop of programmers. This has a real impact on programmer workflow. When they work on updates to the test suite, the result of the tests on the CI infrastructure will be delayed by that amount of time.

This problem is obviously not restricted to Mozilla. If, as preliminary experiments tend to show, the cumulative density function of incremental build times has the same shape for most projects, then nearly 90% of CPU time on CI workers is wasted. From this experiment, we see that incremental builds could save a lot resources, either developer attention and time or bare computing power. Despite this, the optimization is not used on continuous integration infrastructures. In the next section, we take a closer look at the reasons behind this absence.

III. Build properties

Having illustrated that incremental builds could provide huge performance gains, we now try to explain why such a well known optimization is never used in automated build farms. We identify three reasons that make incremental builds unfit for CI environments.

a. (lack of) Correctness: Existing build systems implement incremental builds incorrectly. The obtained build result with incremental compilation differs from the one produced by a clean, full rebuild.
b. (lack of) Control over the environment: Build systems rely heavily on their environment. If not strictly controlled, products built from the same source may not be the same. After multiple incremental builds, these impurities stack up, and correctness is impacted.
c. (lack of) Linearity: CI uses clusters to build and test patches. A given worker may not have access to the build products of previous builds, forcing it to rebuild from scratch. This requires communication and caching between builds.

We discuss details of these aspects in the following sections.

A. Correctness

At the heart of correctness resides the idea that a build system should behave like a pure function. For a given initial state, the build system should always produce the same result. This means that incremental builds and full rebuilds from the same sources should always yield the same products. There are various ways to break current build systems. While some build systems are more robust than others, most can be tricked into producing wrong results. This topic has been extensively studied by Mike Shal [6]. We will only give a small example of such incorrect behavior of `make`. Because it has no memory of its previous invocations, `make` cannot tell sources from old build products. When a target 'old' is renamed into a target 'new', `make` does not delete 'old' when it produces 'new'. The resulting state has both 'old' and 'new', which would not be the case if the system had been build from clean sources. This example also shows that correctness requires to keep a trace of previous executions. However `make` is a stateless program. It explores the working tree at each invocation to discover what needs to be done. If it remains stateless, it will never achieve correctness as described here.

Together with his article, Mike Shal developped `tup`[4], a correct build system as per his own extensive definition. Recent build systems like `pluto` [4], `bazel`[5], `buildsome`[6], `button`[7] and many more claim to be correct. Although their definitions of correctness are not equivalent, most agree on the fact that correct build systems never need full rebuilds because their incremental build algorithms will always manage to "do the right thing". This is a feature of new build systems only. Existing projects will need time to switch to these build systems. For example, it is still recommended[8] to clean the source tree before and after every build of the Linux kernel.

The number of new build systems claiming to be correct is a strong hint that old, approximative build systems do not meet today's expectations of quality and usability.

B. Control of the environment

Most definitions of correctness used by next-gen build systems remain local to the source tree. There is a broader scope in which builds happen, commonly referred to as the environment. The environment is composed of all the parameters that are not under direct control of the build system but that may influence the build results and therefore its correctness with respect to the whole environment. Variability in build environments appears with the set of available applications, their installed versions, the processor architecture of the build machine and many other parameters. Even small settings like the hostname, the amount of RAM or the current time can impact builds in subtle ways. For incremental builds to produce correct results, they must either take these impurities into account, or great care should be taken to run successive incremental builds into (nearly) identical environments.

Fortunately, CI platforms have long seen the usefulness of controlling the build environments for reproducibility purposes. Docker and VirtualBox are probably the most used tools to manage build environments. Starting from this, it remains to be ensured that build systems can detect environment changes. Based on that information, they can rebuild the impacted parts or take other appropriate actions. Smarter build system could track important parameters from the environment and try to hide the others from the build tasks.

The `nix` package manager [3] is an interesting project in this context. It installs every software package in a different path. As packages cannot conflict, multiple versions of the same package can be installed at the same time. When using full paths, it is trivial to detect different versions of the same binary; they have different paths. This also means that you have control over which version of libraries are used. `Nix` packages do not depend on common installation paths like /usr or /lib. They reference their dependencies directly by their full path. `nix` allows fine-grained control over execution environments and system configuration, and could be used to wrap old-fashioned build systems that do not take the environment into account.

By properly controlling the environment and taking it into account, build systems will produce correct incremental builds at the level required by release engineers.

C. Lack of linearity

The last obstacle for CI to use incremental builds is that CI builds are usually performed by a cluster of different workers. These workers

[4]http://gittup.org/tup/
[5]https://bazel.build/
[6]http://buildsome.github.io/buildsome/
[7]http://jasonwhite.github.io/button/
[8]https://www.csee.umbc.edu/courses/undergraduate/CMSC421/fall02/burt/projects/howto_build_kernel.html

cannot perform efficient incremental builds because they probably did not build the most recent version before the change being tested. If nothing is done, chances are that the worker will have to catch up with intermediary commits and build more products than strictly needed. On a lower level, it also requires workers to maintain a cache of previous builds. This is not easy to do when workers are provisioned dynamically on the cloud. In both cases, the solution is identical. A cache of build results needs to be maintained.

Caching build results is not a new idea. It is exactly what the `ccache` program performs for the C/C++ compiler. By comparing input files' content and compiler flags, `ccache` is able to cache object files and detect equivalent compiler invocations. In that situation, ccache does not call the compiler but directly produces the object files from its cache. As explained above, this does only work if all the relevant parameters are taken into account. Failing to detect parameter changes, ccache may produce incorrect object files. A similar mechanism could be implemented for entire build steps. Given a build system with enough control on the environment of its sub-processes, it could cache the result of sub-commands execution and query the cache before each sub-command. A well-designed cache could be shared by all the CI workers making the knowledge of one available to the others at once. Many questions remain on the feasibility and practical performance of such an implementation. Much remains to be done in the future.

With these three issues addressed, we have sketched the key issues to be addressed by a build system good enough to be used in continuous integration. With these three features, the build system would meet the correctness and reproducibility requirements of release engineers while still providing the full benefit of incremental builds. All these aspects are individually present in different pieces of software. The challenge resides in making them work together.

IV. Ideal build system

In order to support continuous integration for CI, we propose a design for an ideal build system. To anchor this discussion in the real world, we propose to reuse existing software. This would avoid building yet another build system from the ground up. The tools proposed here could most probably be replaced by other equivalent tools.

At the core, we need a correct algorithm to manage incremental builds. The algorithm from `tup` is correct (see Section III-A) and could be reused as-is. However, `tup` does not take into account changes to the host system, and does not provide caching.

To manage the environment, we propose to use `nix` as described in Section III-B. Combining `tup` with `nix`, we can generate build configuration files containing the full path to the command to run. The way `nix` installs packages ensures that the command path will change when the executable or one of its dependencies change. This way `tup` will detect changes to the build command and trigger rebuilds where needed. As `tup` already tracks environment variables, and `nix` allows to remove much variability between systems, the build environment would be under control.

Finally, to provide sharing across the workers, we need an efficient and correct caching mechanism. The cache should be implemented at the lowest level, beneath `tup`. Whenever `tup` invokes a build command, the cache will first check if there is already a build result for that exact compilation. If the same step already happened, the cache produces the build outputs without running the build step. If not, the build step is executed and the results are added to the cache and shared on the cluster. The cache needs to take into account the content of the input files, and all the compilation parameters like the compiler version and its flags as well as the available packages.

While there exist excellent caching tools, significant research and testing is required for this step. In particular it requires to design a proper encoding of a build command and all its dependencies to make lookup of previous identical builds efficient. Also, caching the result of a build may be more complex than it looks. How should we cache the result of a build step that appends to an existing file for example? Finally, caching is never easy to implement correctly, and may turn out to be impractical. For example, network access could be slower than a local rebuild. This part of the tool requires investigations to ensure its feasibility and practicality.

A tool designed like this should provide correct and efficient incremental builds for distributed continuous integration environments. Like most optimizations, this tool would add complexity and introduce potential bugs to be discovered. A careful assessment of the final tool would have to be realized. While we cannot provide guarantees that it would be adopted by industry, we believe that a huge speedup in compilation time could make it attractive.

V. Conclusion

With incremental builds, we have argued that it is possible to improve build performance by one or two orders of magnitude. That speedup is needed because it is on the critical path of developers that must wait before getting test results. We have analyzed the main barriers to the usage of incremental builds as an optimization technique for continuous integration. For each of these we explained how it can be handled and provided examples of existing tools capable of doing it. Taken together, these features address all the remaining issues to deploy incremental builds to CI. A tool that implements them all, possibly by reusing existing partial tools, would make it possible to get the same speedup on CI as for local incremental builds. Having provided the need and path, we are ready to start the journey.

VI. Acknowledgments

We thank Bram Adams for his comments on an earlier version of this paper. We would also like to thank Kim Moir and Mike Shal for their attentive reading and their eagerness at discussing release engineering practices at Mozilla.

Add an acknowledgements section where you explicitly thank Bram Adams for his comments on an earlier version of this paper.

References

[1] Bram Adams and Shane McIntosh. "Modern Release Engineering in a Nutshell – Why Researchers Should Care". In: *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5. Mar. 2016, pp. 78–90. DOI: 10.1109/SANER.2016.108.

[2] Moritz Beller, Georgios Gousios, and Andy Zaidman. *Oops, my tests broke the build: An analysis of travis ci builds with github*. Tech. rep. PeerJ Preprints, 2016.

[3] Eelco Dolstra, Merijn De Jonge, Eelco Visser, et al. "Nix: A Safe and Policy-Free System for Software Deployment." In: *LISA*. Vol. 4. 2004, pp. 79–92.

[4] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. "A Sound and Optimal Incremental Build System with Dynamic Dependencies". In: *SIGPLAN Not.* 50.10 (Oct. 2015), pp. 89–106. ISSN: 0362-1340. DOI: 10.1145/2858965.2814316.

[5] Kim Moir. “Built to Scale: The Mozilla Release Engineering toolbox”. EclipseCon. 2014. URL: https://www.eclipsecon.org/na2014/session/built-scale-mozilla-release-engineering-toolbox.html.

[6] Mike Shal. *Build system rules and algorithms*. Available at http://gittup.org/tup/build_system_rules_and_algorithms.pdf. 2009.

[7] Rodrigo Souza, Christina Chavez, and Roberto A Bittencourt. “Rapid releases and patch backouts: A software analytics approach”. In: *IEEE Software* 32.2 (2015), pp. 89–96.