On the Benefits and Limits of Incremental Build of Software Configurations: An Exploratory Study

Georges Aaron Randrianaina, Xhevahire Tërnava, Djamel Eddine Khelladi, Mathieu Acher

▶ **To cite this version:**

Georges Aaron Randrianaina, Xhevahire Tërnava, Djamel Eddine Khelladi, Mathieu Acher. On the Benefits and Limits of Incremental Build of Software Configurations: An Exploratory Study. ICSE 2022 - 44th International Conference on Software Engineering, May 2022, Pittsburgh, Pennsylvania / Virtual, United States. hal-03547219v1

HAL Id: hal-03547219

https://hal.science/hal-03547219v1

Submitted on 28 Jan 2022 (v1), last revised 11 Feb 2022 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On the Benefits and Limits of Incremental Build of Software Configurations: An Exploratory Study

Georges Aaron Randrianaina
Univ. Rennes 1, IRISA
Rennes, France
georges-aaron.randrianaina@irisa.fr

Xhevahire Tërnava
Univ. Rennes 1, Inria, IRISA
Rennes, France
xhevahire.ternava@irisa.fr

Djamel Eddine Khelladi
CNRS, IRISA, Univ. Rennes 1
Rennes, France
djamel-eddine.khelladi@irisa.fr

Mathieu Acher
Univ. Rennes 1, Inria, IRISA,
Institut Universitaire de France (IUF)
Rennes, France
mathieu.acher@irisa.fr

ABSTRACT

Building software is a crucial task to compile, test, and deploy software systems while continuously ensuring some quality assurance of a project. As software is more and more configurable, building multiple configurations is a stressing need, yet, costly and challenging to instrument. The current state of practice is to build independently (*a.k.a.*, clean build) a software for a subset of configurations. While incremental build has been considered for software evolution and relatively small modifications of the source code, it has not been surprisingly investigated for software configurations. In this first novel exploratory study, we investigate the benefits and limits to incrementally build software configurations, instead of always clean building them. By using five real-life configurable systems as subjects, we explore whether incremental build works, outperforms a sequence of clean builds, is correct *w.r.t.* clean build, and can be used to find an optimal ordering for building configurations. Our results showed that incremental build is feasible in 100% of the time in four subjects and in 78% of the times in one subject. In average, 88.5% of the configurations could be built faster with incremental built while also finding several alternative faster incremental builds. However, only 60% of faster incremental builds are correct. Still, when considering those correct incremental builds with clean builds, we could always find an optimal order that is faster than just a collection of clean builds with a gain up to 11.76%.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

Configurable, Variability, Build Systems, Exploratory Study

1 INTRODUCTION

Building software is a crucial activity in every non-trivial software project. Different artefacts are assembled, compiled, tested, and eventually deployed, hopefully with success. The rise of continuous integration has amplified this trend with build services integrated into major code platforms (*e.g.*, GitHub, GitLab). The interest is to continuously ensure some quality assurance of a project, being in terms of functionality or non-functional properties (*e.g.*, security, execution time). Although widely adopted, building software is increasingly complex and expensive in terms of time and resources [11, 15, 23, 31].

Software configurations are adding further complexity to the problem of building software. Different variants of the artefacts can be assembled due to *e.g.*, conditional compilation directives #ifdef in the source code. Different external libraries can be compiled and integrated as well. The way the build is realized can also change *e.g.*, with the use of different compiler flags. Developers and maintainers of a project want to ensure that, throughout the evolution, all or at least a subset of software configurations build well. As most of today's software is configurable in order to fit constraints, functional and performance requirements of users, it is no surprise to observe that many organizations build different software configurations of their project. For instance, initiatives like KernelCI or 0-day build thousands of default or random Linux configurations each day [7]. Another example is JHipster, a popular Web generator, that builds dozens of configurations at each commit, involving different technologies (Docker, Maven, grunt, etc) [21].

The current state of practice is to build independently a subset of configurations *i.e.*, in a fresh environment. In this paper we propose and explore an approach, called *incremental build of configurations*. The idea is simple: instead of starting from scratch and cleaning build's artefacts, a configuration can be built from an existing and already completed configuration build (incrementally). The hope is to reuse artefacts of previous configurations' build and thus save some computations, resources, and time. This is just a hunch and the real question is to quantify how much and when you can gain (or lose) compared to a more conventional build. Moreover, the approach is not without risk: an incremental build may not work or may be incorrect compared to a conventional build. The build system may forget to recompile some necessary artefacts, for example. In fact, it is an hypothesis that needs inquiry. Another unknown is about the strategy to schedule the incremental build of configurations. Given a configuration to build, with which other configuration should the incremental build be carried out? Should incremental build be used all times? There are numerous possible orderings, possibly with different effects on the correctness and overall build time. Our goal is to explore these hypothesis and address, to the best of our knowledge, new open questions: **(RQ1)** Is incremental build simply possible in real-world configurable systems? **(RQ2)** Does incremental build outperform clean build?

(RQ3) Is the result of incremental build the same as that of clean build? **(RQ4)** Is there an order of configurations that brings an optimal (overall) incremental build time?

Novelty. Surprisingly, while incremental build is supported and has actually been designed for relatively small edits, it has not been explored for software configurations. A possible reason is that the usual compilation workflow and build process targets small, local modifications (*e.g.*, modification of one source file). Building several configurations involve large modifications that span numerous source files thus challenging build systems. There are numerous works in the software product line community about variability-aware analysis [25–28, 40, 45, 50]. The idea is to process the configurable code base, exploiting similarities among individual variants with the goal of reducing analysis effort. Incremental build shares the same principle and aims to leverage similarities among configurations' build. However, we are unaware of works that consider the actual build of configurations in an incremental way. To the best of our knowledge, investigating the benefits and limits of incremental build at the configuration level has not yet been considered.

Significance of the problem. The promise is to reduce the cost of building software, a stressing topic when you think about the environmental and financial costs that companies and public organizations should have to bear [11, 15, 23, 31]. Society relies on software, but building software has an enormous cost: we aim to mitigate this trend. Beyond Linux and JHipster examples already mentioned, numerous real-world software projects are configurable and actually build several configurations. The build is a necessary step to check the correctness of the code, to dynamically test the system, to observe non-functional properties (*e.g.*, execution time, security), to synthesize performance models [16, 20, 22, 29, 34, 39, 42, 43, 46, 49]. Owing to the cost and the frequencies of software builds, any improvement is more than welcome for developers and organizations.

Practical scenarios. Incremental build is mainly intended to be part of a continuous integration. With each commit, rather than building configurations separately, an ordering strategy can be used to reduce build time or to build much more configurations given a budget. Once the (optimal) order of the configurations is established, the benefits can be obtained several times during the evolution of a software project. Of course, the ordering can be updated in case of major modifications. Out of an order, the distribution of incremental build of software configurations on different machines is also possible but out of the scope of the paper.

Methodology. This paper designs and performs the first study about incremental build of configurations: the purpose is to understand the challenges and practices in a real-world setting and to generate hypotheses about other, similar contexts [47]. Our contribution is limited to the study of incremental build on existing and real projects. On the other hand, this study opens new perspectives and brings insights that can be used to design new build systems or scheduling heuristics. Our data set covers five subject systems, namely `x264`, `sqlite`, `xz`, `curl`, and `xterm` covering different domains, respectively, video encoding, database, compression utility, network communication, and terminal emulator. Though all written in C language, their build process highly differs (more details in the paper). We explore whether incremental build works (*i.e.*, produces something without errors), whether an incremental strategy outperforms a sequence of clean builds, whether the result is correct *w.r.t.* clean build, and what an optimal ordering of configurations brings in terms of build time.

Significance of the results. Our results showed that incremental build is feasible in 100% of the time in four subjects and in 78% of the times in the `xz` subject. On average, 88.5% of the configurations could be built faster with incremental build while also finding several alternative faster incremental builds. However, only 60% of faster incremental build are correct. Still, when considering those correct incremental build with clean build, we could always find an optimal order that is faster than just a collection of clean builds with a gain up to 11.76%. Overall, our results suggest that incremental build of configurations can be beneficial. Owing to the importance and increasing cost of build systems in the field of software engineering, it is worth addressing the open issues for fully realizing the potential of incremental build.

The main contributions of this paper are:

(1) the idea of incremental build of configurations;
(2) the design of an exploratory study to investigate the potential benefits and pitfalls of incremental build;
(3) a quantitative and qualitative analysis of empirical results;
(4) a discussion on the impacts of our works on developers, build systems designers, and researchers in configurable system or continuous integration;
(5) a link to our publicly available data set for reproducibility.

2 BACKGROUND AND MOTIVATION

This section gives a background on the tools and the process that are commonly used to build a C-based, configurable software system, which are the subjects of this study. Then, we motivate our work with an example taken from the `xterm` terminal emulator subject.

2.1 Autotools

When installing a new software package in an environment, the user needs to have its dependencies, such as the required utilities and libraries by that software. In addition, the package needs also to know which are the available features of the current operating system in order to be configured and built accordingly. With the high diversity of operating systems, their features and even local hacks by the users, it is impossible to manage the configuration script for every possible environment. For this reason, the GNU Project introduced the *Autotools* utility to automatically generate configuration and build scripts to fit to the machine it is built upon.

The presented C-based projects in this paper, as subjects, mainly use the GNU Autotools with the utilities of *Autoconf* [1] and *Automake* [2], as shown in Figure 1. Specifically, Autoconf takes as input a file `configure.ac` in which the developer has specified which packages and features can be used~~and which ones are missing~~. Then it generates a `configure` file, which contains a script to check the available packages and features of the current machine and a script to configure the project given the user's configuration. On the other hand, while writing a software with a large amount of files, it is complicated to keep track and specify how to build each one of them. Therefore, the developers of C-based projects mainly use a Makefile to describe how to build the project. But,

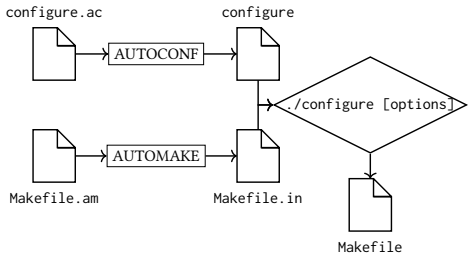

Figure 1: The workflow of Autotools utility

Listing 1 Makefile rules, illustrated in Figure 2

```
1 util.o: util.h util.c
2     cc -c util.c
3 main.o: util.h main.c
4     cc -c main.c
5 main: util.o main.o
6     cc util.o main.o -o main
```

the Makefile support is different depending on every possible computer environments. This is the case when the developer needs the Automake utility. Specifically, the developer first specifies the structure of the current project and how to build it. Then, Automake generates a `Makefile.in` that has the previously build rules with compatibility on the current machine's environment. Once the `configure` and `Makefile.in` are generated, the user needs to run only the `configure` script in order to configure the project. The `configure` script can use different options from the user, which are well-known as compile-time options, and generates the `Makefile` accordingly. After this configuration step, the users need to run the `make` command in order to build the given project. This entire process is also sketched in Figure 1.

Usually, Autotools is installed on the developer's computer, but not on the user's environment. Thus, the `configure`, `Makefile.am`, and `Makefile.in` often are shipped within the compressed folder that contains the project's sources. In this case, the user simply needs to execute the `configure && make && make install` command in order to configure, build, and install that given project.

2.2 Make and incremental build

Make is a well-known build system [17]. Make reads *Makefiles* which contain the build rules to build the current software project. A Makefile can be either generated by using Autotools or written by the developers themselves. To show how Make works, Figure 2 gives an illustrative example of a Makefile for a toy project (its source files are not included here). It is a borrowed example from [38], with the build rules presented in Listing 1.

In the lines 1-2 of Listing 1 is defined the rule `util.o` which depends on the header file `util.h` and `util.c`. In its line 2 is described how to build it by using a `cc` compiler: the flag `-c` and the file to compile `util.c` in order to produce an object file `util.o`. The same thing is done by the next rule of `main.o` in lines 3-4. The last `main` rule, in lines 5-6, specifies how to build the final main product. In this case, it simply links the `util.o` and `main.o` files.

From this Makefile, Make can build a dependency graph like the one shown in Figure 2 (left). During the compilation for the first time, as nothing has been compiled yet, Make will build each of the described rules in the Makefile in order to build the final

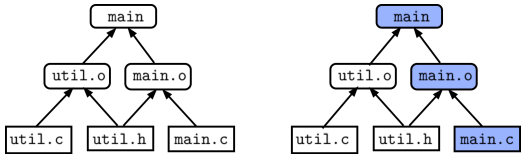

Figure 2: File dependency (left) and incremental build (right)

product. Building a project from scratch and from its clean basis is well-known as a *clean build*.

But, if a developer modifies even a file, then Make will rebuild only the rules that depend on this file. To build only the necessary rules, Make checks their timestamp. A rule must be rebuild if one of its dependencies is more recent than it. For instance, if the developer modifies `main.c`, Make will only build the rule that depends on it, that is, the `main.o` rule. Then, the rule that depends on `main.o`, the `main`, and so on. As `util.o` does not depend on `main.c`, Make will not update it. These updated rules by Make are also highlighted in blue color in Figure 2 (right). This process, where the build system does only the minimum work without rebuilding the unmodified targets is well-known as the *incremental build*.

The incremental build process of Make is meant for file changes of a single software configuration. However, we can leverage it to apply on files changes that are triggered by different compile-time configurations options in a C-based system. Indeed, configuration options are enabled in the code through file addition or C preprocessor directives. Inclusion of more files in a rule forces its update. Moreover, additions of blocks of code through C preprocessor directives changes the file's timestamp to a more recent one, hence it forces again the rebuild of the targets depending on it. Thus, Make will only recompile the necessary targets during an incremental build of configurations.

2.3 Motivating example

Let us consider `xterm`, a standard terminal emulator for the X Window System. `xterm` has 63 compile-time options which may lead to less than 2^{63} configurations (the exact number is certainly lower due to constraints). Building all (or even a subset) of configurations is costly and time-consuming. It is where incremental build can play a positive role. Figure 3 shows the build results for four configurations randomly generated for `xterm`. The top of Figure 3 gives the options of the four diverse configurations, and the bottom of Figure 3 gives the build time for clean build of each configuration separately, and the incremental build of four configurations $c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow c_4$. Note that in the incremental build, the first configuration must be built from scratch with a clean build. Only then we can start building incrementally the rest of configurations.

From Figure 3, we observe that the total sum in time (seconds) of the clean builds of the four configuration is 45,9 s, whereas the total sum in time of the incremental build is 41,42 s. This represents a total gain of 4,48 s and 9,8% of time difference. We also observe individual gain for every incremental build of the three last configurations, respectively 6,16% for c_2, 15,3% for c_3, and 18,23% for c_4. This gain is only on four configurations and can potentially be more significant the bigger the number of configurations that need to be build. To the best of our knowledge, no study exists on exploring the benefits

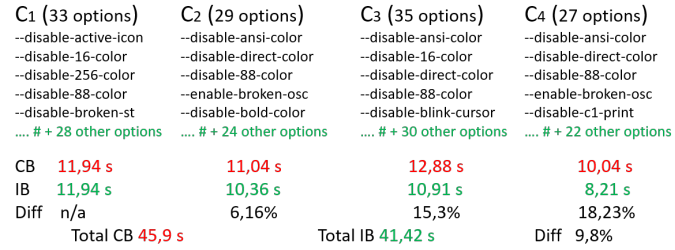

Figure 3: Example of clean build (CB) and incremental build (IB) on four random configurations of `xterm`

and limits of incremental build on configurable software. The rest of the paper designs and reports on our exploratory study.

3 EXPERIMENTAL APPROACH

This section details incremental build experimentation. We first present the used research questions to address the goal of our experiment. Then, we describe the used subject systems with their configurations, and the used build approach in out experimentation.

3.1 Research questions

The goal of this study is to explore the feasibility, efficiency, optimization, and correctness of incremental build in real-life configurable software systems. Hence, we define the following four research questions.

RQ_1– **Feasibility: Is incremental build possible in each configurable system?** We first explore whether the build status of each of our subject software systems is successful during the incremental build in its all considered set of configurations, in some of them, or in none of them.

RQ_2– **Efficiency: Does incremental build outperform clean build?** To this end, we propose a *build approach* to measure and compare the clean build and incremental build time of each subject system in its respective set of configurations.

RQ_3– **Correctness: Is the result of incremental build the same as that of clean build?** To this end, we compare whether the resulting executable binary size and its symbols are the same after the incremental and clean build of each respective configuration in five subject systems.

RQ_4– **Optimal ordering: Is there an order of configurations that brings an optimal (overall) incremental build time?** By using our build approach, we analyse whether it exists an order of configurations such that the system with a given configuration can be incrementally build faster while being correct.

3.2 Subject systems

The objects of this experiment are five real-life software systems (see Table 1). To select them, we had in consideration several criteria. Namely, the fact that the system is an open-source and available project, it has compile-time configuration options, is a popular project, and covers a different application domain. As a result, we selected five C-based software systems as subjects. We first selected the command-line video encoder of `x264`, which has been widely studied among the highly configurable systems [9]. Then, we selected the most used SQL database engine of `sqlite`, the general-purpose data compression software of `xz`, the library of `curl`, which supports a wide range of data transfer protocols with URLs, and `xterm`, which is the standard terminal emulator for the X Window System. To reason on a project's popularity, we mostly used as a proxy the recent number of stars (from 29 to 21.6k), commits (from 1300 to 27k), and contributors in its git repository.

Table 1: Subject systems with their respective analysed commit/tag ID, LoC (lines of code), considered compile-time options, range of options within configurations, and batches

System	Commit/Tag ID	#LoC	#Opt.	Range	#Batches
`x264`	ae03d92	115.243	16	3 - 7	2 x 20
`sqlite`	version-3.35.4	318.521	25	7 - 18	2 x 20
`xz`	e7da44d	39.714	87	14 - 23	2 x 20
`curl`	curl-7_78_0	248.713	109	39 - 47	2 x 20
`xterm`	xterm-368	130.850	63	26 - 37	2 x 20

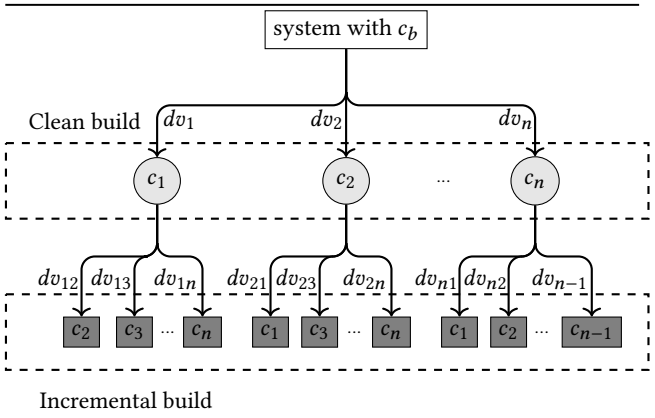

Figure 4: Experimentation workflow. Where c_b is the system's baseline configuration, and c_1 - c_n are its random generated configurations. The dv_n and dv_{nm} are the measured dependent variables (build time, executable binary size, etc.)

3.3 Variables

Our experiment aims to study the incremental build of a given configurable system in contrast to its clean building. Hence, the incremental and clean build of configurations is the *independent variable* we controlled. To answer our research questions, we observed three *dependent variables*, namely: the *build time* of a configuration, the system's *executable binary size*, and the system's *object files* in its build resulting folder, after both the clean and incremental builds.

3.4 Build approach

To be able to explore the qualities of incremental build in configurable systems and answer our research questions, we design the following build approach, and apply it in the five subject systems.

Given a configurable system ψ with compile-time options $O_\psi = \{o_1, o_2, ..., o_m\}$, where $m \in \mathbb{N}$, we first create a sample of configurations $C_\psi = \{c_1, c_2, ..., c_n\}$ for that system. Where $n = 20$, in this study, and each $c_i \in C_\psi$ has a varying size with a random list of

Listing 2 x264's clean build with c1 and incremental build with c2

```
1  /* Clean build with c1 */
2  [x264]$ ./configure  --disable-interlaced --bit-depth=8 \
3                       --chroma-format=444 --disable-bashcompletion
4  [x264]$ time make ; ls -l x264
5  >> 0m20.262s
6  >> -rwxr-xr-x 1904936 x264
7
8  /* Incremental build with c2, after the clean build with c1 */
9  [x264]$ ./configure  --disable-asm --disable-gpl \
10                      --disable-thread --disable-interlaced
11 [x264]$ time make ; ls -l x264
12 >> 0m2.256s
13 >> -rwxr-xr-x 2423016 x264
```

Listing 3 x264's clean build with c2 and incremental build with c1

```
1  /* Clean build with c2 */
2  [x264]$ ./configure  --disable-asm --disable-gpl \
3                       --disable-thread --disable-interlaced
4  [x264]$ time make ; ls -l x264
5  >> 0m2.422s
6  >> -rwxr-xr-x 2423016 x264
7
8  /* Incremental build with c1, after the clean build with c2 */
9  [x264]$ ./configure  --disable-interlaced --bit-depth=8 \
10                      --chroma-format=444 --disable-bashcompletion
11 [x264]$ time make ; ls -l x264
12 >> 0m19.658s
13 >> -rwxr-xr-x 1904936 x264
```

generated options from O_ψ. In Figure 4 is given an overview of our build approach. First, it should be noted that each configurable system has a default configuration, to which we will refer in the following as the baseline configuration (c_b). The system with its c_b is at the root of our build approach. Then, we apply two main build steps, named *clean build* and *incremental build* as in Figure 4.

Clean build. In this step, the baseline system is build with each configuration $c_i \in C_\psi$ from scratch. During this clean build, we measure and record the dependent variables (dv_i) of c_i, namely, the build time, the executable binary size, and the generated files.

Incremental build. Then, over each clean build configuration (*cf.* c_1 to c_n in Figure 4), we incrementally build the same configurations, except the current previous applied configuration during the clean build. For instance, over the clean build of c_n we incrementally build all other configurations from c_1 up to c_{n-1} except the c_n itself. We assume that the incremental build of c_n after its clean build will provide the same results, hence it will be insignificant to study.

Listings 2 and 3 show an illustrative example taken from the first subject system of x264 with two of its used configurations, called c_1 and c_2. In Listing 2, the x264's baseline system is first build with c_1 (lines 2-4), that is, it is clean build, and then it is incrementally build with c_2 (lines 9-11) over the previous build. Similarly, in Listing 3, the system is first clean build with c_2 (lines 2-4) and then it is incrementally build with c_1 (lines 9-11). It should be noted that after the first build, in line 9 of Listings 2 and 3, instead of cleaning the directory with `make clean`, we directly configure the system to host the new configuration: this is what we refer to as an incremental build. Hence, we incrementally build the next configuration on top of the previous clean build configuration.

Further, during each clean and incremental build process, we measure the build time and the executable binary size of the system (lines 4-6 and 11-13). Moreover, we save the state of the build system's directory after each build. In this way, we can retrieve useful data such as the binary, configuration logs and object files for further analysis.

3.5 Experiments settings

By using the presented build approach, we conduct an experiment with the five subject systems. Specifically, we build each system by using two batches with 20 random generated configurations each. In this way, we perform $2 * 20$ clean builds and $2 * (20 * 19) = 760$ incremental builds, or in total 800 builds of each system. To correctly handle all of them, we used a local git structure where each system

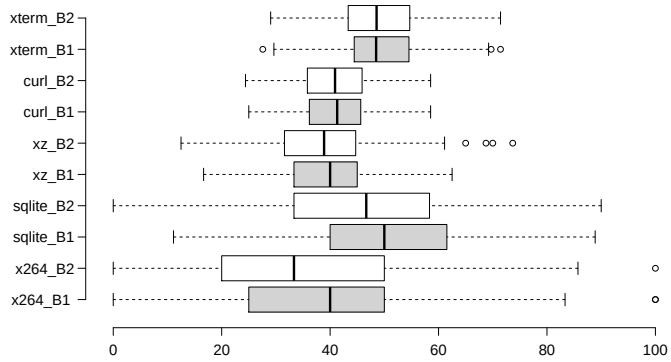

Figure 5: The distance between different generated configurations, per batch, in five subject systems

build is saved in a new git branch. The resulting git structure of a system with all its builds has the same view as in Figure 4.

To generate the random configurations, we use the random product generator in the FeatureIDE framework [36]. It should be noted that the range of options within the sample of configurations (*cf.* column 'Range' in Table 1) changes quite proportionally with the number of considered options among the subjects (*cf.* column '#Opt.' in Table 1). Whereas, for their comparison, we take two batches with 20 sample of configurations in all five subjects. Further, to ensure that the generated set of configurations is diverse enough, we looked at the distance between each pair of configurations within a batch. By distance, we mean the percentage of common options between configurations. Hence, the distance 0 between two configurations $c_i, c_j \in C_\psi$ indicates that the sets of compile-time options in these two configurations are disjoint sets. Whereas, the distance of 1 indicates that c_i is the same or at least a subset of c_j. Figure 5 shows the overall calculated distribution of the distance between all pairs of configurations in our five subject systems. The median distance varies between 33.33% in the second batch (B_2) of x264 and 48.57% in the second batch (B_2) of xterm, indicating that in each subject the considered configurations within a batch are diverse for more than 50% on average. We also observed 82 configurations of 0 distance in x264 (B_1 and B_2) and in sqlite (B_2), that is, completely different. Then, in x264 (B_1 and B_2) there are only three cases with distance 1, that is, entirely included in another one.

We conducted the experiments on a Linux workstation running Fedora 34 with Intel Core i7-10610U CPU and 15.3 GiB of memory. To prevent side effects, all experiments are run twice using Docker and sequentially, as the only processes in the workstation. Moreover,

the Dockerfile of each subject system is made available [1] and can be used to reproduce the experiments. At the same place, we provide the build configurations in all systems with their generated data.

4 RESULTS

We now discuss the results with regard to our research questions.

4.1 Feasibility of incremental build (RQ_1)

In order to measure and compare the build time and system correctness after an incremental build, we first had to ensure that the system is successfully building during the clean build of its configurations, that is, it has a build status 0 and an executable file is created. In cases when the system threw an error during the clean build of a configuration, we analysed the dependencies between compile-time options and resolved it by removing one of the conflicting options. For example, we encountered such option's dependencies in the configurations of `xterm`. As a result, in our experiment with five systems, the clean build of each of their configurations is successful. Put differently, in all our clean build cases the system's build status is 0 and a system's executable file is created.

Despite the successful clean build of all configurations, we noticed that some configurations in some systems have a failed build status during their incremental build. Finding out whether the incremental build is feasible in a system or not was hard to deduce by simply inspecting its Makefile. Therefore, we incrementally build all pairs of configurations in five systems and observed their build status, including the presence of their executable file. As a result, all of the configurations in `x264`, `sqlite`, `curl`, and `xterm` are successfully incrementally build. But, there are 9 configurations in `xz` that have a failed build status during their incremental build. These cases are marked with a dash ('$-$') in Table 2 for batches B_1 and B_2. For instance, by incrementally building `xz` with configuration c_3 over any other configuration, in B_1, it resulted in a failed build status and without an `xz`'s executable file.

Based on our experiments with five subjects, a successful clean build configuration is not necessarily build successfully during its incremental build. Our initial observations, based only on the build status and the created system's executable, are that the incremental build is feasible in 100% of configurations in 4 systems, namely in `x264`, `sqlite`, `curl`, and `xterm`. Whereas, the incremental build in `xz` is feasible in 78% of its configurations. This indicates that the rest 22% of configurations in `xz` always require to be clean build.

> ***RQ_1* insights:** It can be noticed that incremental build is feasible in 78% of the times in the xz software system and in 100% of the time in the other four software systems.

4.2 Incremental vs. clean build time (RQ_2)

To answer the second research question, we recorded the clean build time of each configuration and the incremental build time of their all paired combinations. For this purpose, we follow the described build approach in Section 3.4. In Table 2 is given the resulting clean and incremental build time of each configuration, for two batches B_1 and B_2, in each subject system. All build times

[1] https://anonymous.4open.science/r/icse22-data-0405/

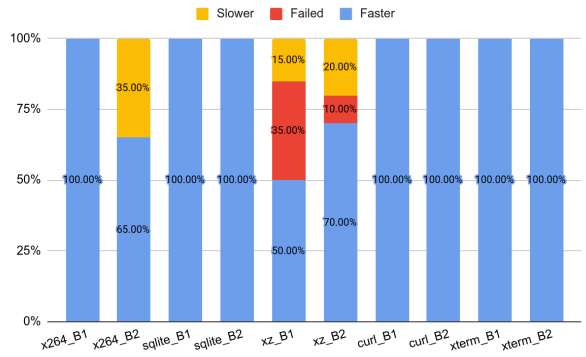

Figure 6: The % of configurations per system that have a faster, slower, and failed incremental build, than clean build

are expressed in seconds. Specifically, the row $CB(t_n)$ shows the obtained clean build time per configuration. For instance, the clean build time of c_{10} in the B_2 of `curl` is 50.13 seconds. The row $mIB(t_n)$ shows the minimum incremental build time of configuration n, from its all paired incremental builds. For instance, the fastest time to incrementally build c_2 of B_1 in `xz` is 10.28 seconds. The next row, $IB(c_m)$, shows which is the clean build configuration for which the incremental build of a given configuration is faster. In the case of `xz`, the c_2 in B_1 is incrementally build faster after the clean build of c_{18}. To easily notice whether the incremental build of a given configuration is faster than its clean build, then the fastest incremental build is colored in green, given in the row of $mIB(t_n)$. Otherwise, when it is slower, it is colored in red. For instance, the incremental build of c_1 in the B_2 of `x264` is 37.95 seconds, or 0.14 seconds faster than its clean build, therefore it is colored in green. On the contrary, the c_{12} in the B_2 of `x264` is 64.56 seconds, or 0.12 seconds slower than its clean build, hence it is colored in red.

From Table 2, it can be observed that in four systems, namely, in `x264` (B_1), `sqlite` (B_1, B_2), `curl` (B_1, B_2), and `xterm` (B_1, B_2), there is always a pair combination of configurations for which the incremental build of a given configuration is faster. However, in `x264` (B_2) there are 7 from 20 configurations for which the incremental build is always slower than the clean build. Similarly, there are 7 from 20 cases in `xz` (B_1, B_2) that always have a slower incremental build than clean build. In Figure 6 is given the percentage of configurations per system that resulted in a faster, slower, or failed build during their incremental build. Specifically, 100% of the configurations in `sqlite`, `curl`, and `xterm` have at least one case where they are incrementally build quicker than during their clean build. Further, 82.50% of the configurations in `x264` and 60.00% of the configurations in `xz` have a faster build during their incremental build. In average, for all five systems, 88.50% of the configurations are build faster during their incremental build than during their clean build. Then, only 7.00% of them are build slower, and 4.50% have a failed build during the incremental build.

In cases when a configuration showed a quicker incremental build, we went further and wondered whether it has more than one pair combination for which it is faster than its clean build. For instance, c_1 in the B_1 of `xterm` is incrementally built over all other 19 clean build configurations, that is, over the clean build configurations of c_2 to c_{20}. When it is built over c_3, given in the row

Table 2: The build time and binary size of subjects for two batches (B_1 and B_2) with 20 configurations each. Legend of colors is in Table 3. *CB* - clean build, *IB* -incremental build, *mIB* - minimum IC, *cIB* - currect IB, " – " - failed build, and "/" - no value

System		c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}	c_{17}	c_{18}	c_{19}	c_{20}
x264 (B_1)	CB (t_n)	38.09	38.60	34.91	70.58	38.24	66.00	21.30	20.79	66.40	32.27	65.72	38.13	20.61	17.90	33.91	18.57	40.74	19.14	35.15	37.87
	IB (c_m)	c_3	c_4	c_{11}	c_{17}	c_4	c_{11}	c_4	c_{20}	c_3	c_7	c_3	c_8	c_5	c_{12}	c_{11}	c_4	c_8	c_4	c_6	c_{18}
	IB (t_n)	37.42	37.85	33.93	69.24	37.41	65.18	20.96	20.33	65.35	31.78	64.58	37.25	20.25	17.65	33.38	18.17	39.99	18.76	34.46	37.22
	BS [MB]	1.83	1.85	2.17	2.91	2.33	2.71	1.53	1.58	2.70	1.67	2.71	1.83	1.50	1.40	2.12	1.48	1.95	1.49	2.18	1.83
x264 (B_2)	CB (t_n)	38.09	18.49	18.31	31.31	69.86	65.46	17.73	34.45	40.12	34.18	34.56	64.44	36.40	18.25	33.62	70.73	34.03	18.43	38.18	31.35
	IB (c_m)	c_{15}	c_8	c_{18}	c_3	c_3	c_{14}	c_3	c_{16}	c_{13}	c_1	c_{16}	c_3	c_{17}	c_1	c_3	c_3	c_3	c_{15}	c_3	c_{17}
	IB (t_n)	37.95	18.54	18.24	31.32	69.84	65.52	17.70	34.53	40.06	34.04	34.51	64.56	36.32	18.23	33.68	70.77	34.03	18.38	38.07	31.34
	BS [MB]	1.85	1.42	1.48	1.64	2.95	2.75	1.37	1.78	1.95	2.17	2.18	2.69	1.79	1.48	2.17	2.96	2.17	1.47	1.85	1.64
sqlite (B_1)	CB (t_n)	61.07	65.67	68.35	68.64	68.55	72.64	57.08	59.41	72.58	71.36	68.11	68.57	54.45	70.82	62.49	65.98	67.85	58.87	72.43	70.91
	IB (c_m)	c_7	c_4	c_{15}	c_7	c_7	c_3	c_4	c_{14}	c_3	c_3	c_{15}	c_4	c_4	c_8	c_8	c_7	c_5	c_2	c_8	c_3
	cIB (t_n)	/	/	/	/	/	70.02	/	/	/	/	/	/	/	/	/	/	/	/	/	/
	mIB (t_n)	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
	BS[MB]	9.20	9.70	9.25	9.73	9.68	9.73	8.07	8.13	9.72	9.64	9.74	9.68	8.17	9.61	8.55	9.52	9.65	8.26	9.73	9.62
sqlite (B_2)	CB (t_n)	103.25	94.24	101.89	103.8	83.63	98.71	90.24	101.67	82.28	91.83	82.95	87.05	89.91	105.17	87.05	96.4	84.8	103.32	103.58	96.58
	IB (c_m)	c_3	c_{16}	c_4	c_1	c_{15}	c_{20}	c_8	c_3	c_3	c_9	c_4	c_3	c_5	c_4	c_5	c_2	c_6	c_{12}	c_3	c_{16}
	cIB (t_n)	63.42	/	/	/	/	94.30	/	/	/	/	/	/	/	/	/	/	/	/	/	56.32
	mIB (t_n)	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
	BS[MB]	9.70	9.51	9.62	9.73	8.41	9.75	9.23	9.60	8.07	8.64	8.58	8.39	9.68	9.73	8.61	9.68	8.62	9.70	9.72	9.70
xz (B_1)	CB (t_n)	12.15	10.78	11.39	10.52	10.24	10.37	11.20	11.89	11.24	10.59	10.29	10.20	10.58	10.44	10.53	12.46	10.53	11.17	11.59	10.19
	IB (c_m)	c_{18}	c_{18}	–	c_{18}	c_{10}	–	–	c_9	c_8	c_5	–	c_{18}	c_{17}	–	c_{18}	–	c_{13}	c_{20}	–	c_{18}
	cIB (t_n)	11.43	10.28	–	10.72	/	–	–	11.28	/	/	–	10.06	/	–	10.65	–	/	10.67	–	10.30
	mIB (t_n)	11.43	10.28	–	10.72	0.40	–	–	1.35	0.42	0.87	–	10.06	0.61	–	10.65	–	0.96	10.67	–	10.30
	BS [MB]	1.16	1.02	1.15	1.02	1.02	1.06	1.09	1.13	1.10	0.96	1.00	0.99	1.02	1.09	1.04	1.23	0.96	1.03	1.16	1.01
xz (B_2)	CB (t_n)	10.38	10.08	11.01	9.75	10.02	10.20	9.94	9.69	10.00	10.90	9.96	9.86	10.74	11.47	9.98	10.18	11.63	11.08	9.82	9.83
	IB (c_m)	c_{13}	c_{17}	c_2	c_2	c_7	–	c_5	c_9	c_2	c_6	–	c_3	c_1	c_8	c_{17}	c_{18}	c_{15}	c_5	c_{17}	c_2
	cIB (t_n)	/	9.93	10.91	9.46	10.08	–	/	9.74	10.10	/	–	10.17	10.42	11.49	/	10.18	11.51	10.95	/	9.77
	mIB (t_n)	0.86	9.93	10.91	9.46	0.81	–	0.47	9.74	10.10	0.95	–	10.17	0.89	11.49	0.28	10.18	1.91	10.95	0.38	9.77
	BS [MB]	1.17	1.01	1.18	0.94	1.02	1.18	1.05	0.92	1.06	1.19	1.09	1.02	1.08	1.17	1.10	1.03	1.16	1.06	0.95	1.01
curl (B_1)	CB (t_n)	23.56	42.42	30.52	44.02	24.99	23.61	22.91	34.05	29.37	24.05	55.98	26.14	40.80	70.95	41.08	44.38	29.34	23.26	21.72	56.86
	IB (c_m)	c_5	c_{16}	c_1	c_{16}	c_9	c_5	c_6	c_5	c_3	c_5	c_{16}	c_5	c_{16}	c_{16}	c_{16}	c_{11}	c_1	c_9	c_{18}	c_{16}
	cIB (t_n)	/	38.17	25.99	39.82	/	/	18.66	/	/	19.66	51.47	22.01	36.34	64.56	37.70	/	24.71	/	17.33	52.37
	mIB (t_n)	0.05	1.06	0.11	0.11	0.06	1.17	0.12	0.05	0.06	0.10	0.11	1.07	0.12	1.07	0.36	1.01	0.12	0.05	1.05	1.08
	BS [MB]	4.67	3.50	0.61	4.15	1.02	4.31	4.09	4.56	0.75	4.31	4.29	0.82	2.61	4.54	3.47	3.89	4.45	0.69	1.10	4.01
curl (B_2)	CB (t_n)	28.61	71.51	33.53	39.86	25.26	67.21	30.87	28.31	29.61	50.13	23.47	41.16	23.40	23.58	44.98	23.35	25.30	22.97	44.97	29.27
	IB (c_m)	c_{13}	c_{11}	c_{11}	c_{11}	c_{11}	c_{13}	c_{11}	c_{13}	c_{13}	c_{11}	c_{13}	c_{13}	c_{11}	c_{13}	c_{11}	c_{13}	c_{13}	c_{13}	c_{13}	c_{11}
	cIB (t_n)	24.69	67.95	29.63	36.75	21.32	65.67	26.98	24.39	25.80	46.52	/	37.59	/	19.61	41.64	19.32	21.18	18.93	41.48	24.82
	mIB (t_n)	1.06	35.01	0.11	19.56	0.12	34.78	1.06	0.11	0.12	24.54	1.01	19.94	0.05	0.11	22.08	0.11	0.11	0.11	21.82	1.07
	BS [MB]	4.47	4.42	4.42	2.85	1.31	4.32	3.60	4.67	3.66	2.83	0.54	3.22	0.58	4.22	3.93	0.70	0.82	3.60	3.93	3.89
xterm (B_1)	CB (t_n)	8.60	8.24	11.64	11.33	11.82	10.23	9.74	9.96	11.53	10.77	9.89	9.08	10.17	8.91	10.26	9.83	7.67	8.63	11.85	10.91
	IB (c_m)	c_3	c_1	c_{10}	c_2	c_1	c_3	c_2	c_1	c_{18}	c_2	c_{20}	c_2	c_1	c_1	c_2	c_1	c_{15}	c_1	c_1	c_2
	IB (t_n)	8.42	8.06	11.43	11.13	11.68	9.83	9.56	9.74	11.42	10.59	9.66	8.94	10.01	8.72	10.11	9.65	7.50	8.49	11.66	10.72
	BS [MB]	2.61	2.35	3.05	3.19	3.30	2.88	2.72	2.77	3.07	2.88	2.77	2.75	2.88	2.62	2.92	2.73	2.28	2.61	3.14	2.89
xterm (B_2)	CB (t_n)	11.41	11.32	10.46	8.88	10.19	10.20	9.08	8.33	12.80	10.76	10.46	10.81	8.52	11.74	11.68	9.88	7.60	11.76	10.59	8.45
	IB (c_m)	c_3	c_1	c_2	c_5	c_3	c_2	c_{10}	c_{10}	c_2	c_2	c_9	c_7	c_6	c_3	c_5	c_3	c_5	c_2	c_2	c_5
	IB (t_n)	11.18	11.11	10.24	8.69	10.05	10.06	8.93	8.17	12.57	10.68	10.31	10.62	8.38	11.53	11.1	9.67	7.4	11.58	10.41	8.24
	BS [MB]	3.11	3.09	2.88	2.55	2.81	2.84	2.52	2.50	3.38	2.91	2.80	3.04	2.44	3.21	3.13	2.61	2.30	3.23	2.75	2.51

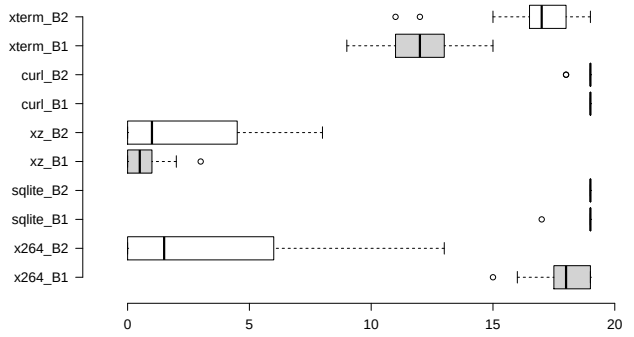

Figure 7: The number of configurations with a faster incremental build (IB) than clean build (CB) in each system

$IB(c_m)$, then it is build faster for 0.18 seconds than during its clean build. But, we noticed that there are also other cases for which the c_1 is incrementally built faster, which cases are not shown in Table 2. Therefore, we count all these cases and show them summarized in Figure 7. It resulted that, except in two case in `sqlite` and `curl`, all their 20 configurations in two batches show a faster incremental build time in all 19 pair combinations with the other configurations. In the B_1 of `x264`, there are between 15 and 19 pair combinations for which 20 configurations are incrementally built faster. In the contrary, in the B_2 of `x264` there are less, between 1 and 13 pairs of combinations. Quite similarly, in `xz` there are configurations that have faster incremental build in up to 3 and 8 pairs of combinations per batch, respectively. In `xterm` there is also a large number of cases for which a configuration can be incrementally build faster, up to 15 and 19 cases, per batch. Hence, in the majority of the cases, one can find more than one pair combination to incrementally build a configuration faster than in its clean build.

In addition, in cases when the incremental built configurations are faster, we then wondered for how much they are. In this way, we want to find out for how many minutes or seconds one can benefit by incrementally building instead of clean building a configuration. To analyse it, we calculated the difference between the clean build

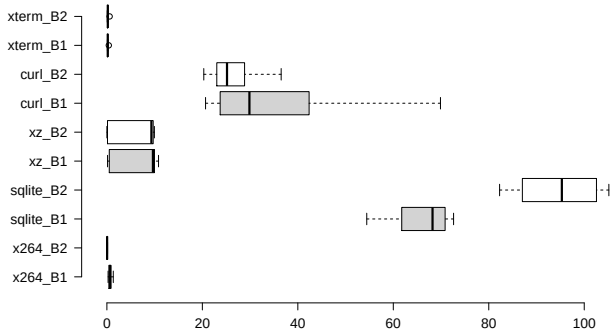

Figure 8: The gained time, in seconds, by the incremental build of configurations instead of their clean build

time and the fastest (the best) incremental build time of each configuration in all systems. The obtained results are shown in Figure 8. The gained time by incrementally building a configuration in `x264` is between 0 and 1.34 seconds. Quite similarly, in `xz` is between 0 and 10.82 seconds. In `xterm` is the smallest gain, between 0.08 and 0.58 seconds. Whereas, in `sqlite` is the largest gained time, between 54.44 seconds up to 1 minute and 45 seconds. This seems to be related also with the taken time to build the system itself. For instance, `xterm` has the smallest build time, less than 12 seconds in all cases. Hence, the gained time during the incremental build is the smallest in `xterm`. The overall gained time for an incremental build configuration, in all systems, is between 0 (*i.e.*, it is the same time as in the clean build) and almost 2 minutes.

These findings show several things. First, in the majority of the cases (88.50%), the incremental build of a system's configuration is faster than its clean build. A successful clean build configuration may have a fail incrementally build (in 4.5% of the cases). Thirdly, in order to benefit on time during the incremental build, the order of build configurations may matter. Then, depending from the system, one can find more than one pair combination of configurations (between 1 and 19) to quickly build a given configuration. The gained time per configuration is quite large (from 0 seconds to 1 minute and 45 seconds). Benefiting in terms of seconds can be significant, for example, in cases when the system needs to be build often and regarding several configurations.

> ***RQ*$_2$ insights:** In average, 88.5% of the configurations could be built faster with incremental built while also finding several alternative faster incremental builds.

4.3 System correctness (*RQ*$_3$)

The observations in the second research question show that in the majority of cases the incremental build of a system's configuration is faster than its clean build. For instance, the clean build time of the configurations in `sqlite` is between 54.45 seconds and 1 minute and 45 seconds, whereas their fastest incremental build time is always 0.01 seconds. Similar examples can be observed also in `xz` and `curl`. Hence, we raised the question whether the resulting incremental build systems in these cases are also correct. That is, whether the system after the incremental build and clean build is exactly the same. To reason on a system's correctness, we analysed the system's executable binary size and symbols of its executable.

Table 3: The percentage of correct systems after their incremental builds, based on their binary size (BS) and symbols

Legend:	`x264` (B_1)	`x264` (B_2)	`sqlite` (B_1)	`sqlite` (B_2)	`xz` (B_1)	`xz` (B_2)	`curl` (B_1)	`curl` (B_2)	`xterm` (B_1)	`xterm` (B_2)
Fast & Same BS	20	12	1	3	5	8	13	18	20	20
¬Fast & Same BS	0	7	0	0	3	5	0	0	0	0
Fast &¬Same BS	0	1	19	17	5	5	7	2	0	0
¬Fast &¬ Same BS	0	0	0	0	7	2	0	0	0	0
Overall same BS (%)	100	95	5	15	40	65	65	90	100	100
Same Symbols (%)	100	95	2	2	2	2	32	46	100	100
¬Same Symbols (%)	0	5	97	98	98	98	68	54	0	0

In the row of $BS[MB]$ in Table 2 is given the executable binary size, in megabytes (MB), of each clean build system for each configuration. These values are summarized in the first four rows in Table 3. As it can be observed, we encountered the four possible cases. Specifically, 60 % of configurations in all systems have a faster incremental build and the resulting system has the same exact binary size as in the clean build. Then, 7.50 % of their configurations have a slower incremental build but still their system has the same exact binary size. On the other hand, there are 32.50 % of configurations in all systems that have a faster or slower incremental build but their resulting system has always a different binary size. Hence, based on the binary size, in 67.50 % of the cases the incrementally build systems are correct, whereas in 32.50 % of the cases they are incorrect. These cases can also be easily identified in Table 2 by using the legend of the colors given in Table 3.

Besides, we compared the system's executable symbols after the clean and incremental build of each configuration in all five subjects. The obtained results are summarized in the last two rows in Table 3. It can be observed that `x264` and `xterm` have the same number (in percentage) of their configurations have a correct system regarding its binary size and its symbols. In other three systems of `sqlite`, `xz`, and `curl`, there are less number of configurations (between 2 % and 46 %) that have the same symbols during their clean and incremental builds. In all systems, there are 48.03 % of configurations for which the resulting system has the same symbols in both build scenarios.

Based on these results, in almost all incrementally build cases `x264` and `xterm` are correct. After investigation, we noticed that the incremental build in these systems is actually not performed by design. In both cases, the `configure` script is generating a configuration file (`config.h` and `xtermcfg.h`, respectively) on which all the other files depend on. Hence, after each configuration everything is rebuilt from scratch. Therefore, the incremental build that we were expecting is actually equivalent to a clean build in these two systems. In the contrast, in `xz` we had incremental build issues. We were getting a same specific linker error telling that some symbols are undefined in some libraries. Hence, the reason of the error is that some libraries were supposed to be rebuild during the incremental build of a configuration, but they were not. Further, the reason for which `curl` and `sqlite` were incorrect is the same. Whenever two configurations shared the same files and generates the same object files then, even when the content in these files changes by a configuration, they are not rebuild and updated. Hence, the incorrect builds in these systems return a warning message that there is nothing to ~~be~~ build.

Table 4: Results for the optimal ordering

System	Total Clean Build	Total Optimal Ordering of Incremental Build	Gain	Reduced N° of Clean Builds
x264 (B_1)	754.92 [sec]	666.12 [sec]	11.76%	20 → 2
x264 (B_2)	747.99 [sec]	747.26 [sec]	0.10%	20 → 8
sqlite (B_1)	1,325.83 [sec]	1323.21 [sec]	0.20%	20 → 19
sqlite (B_2)	1,888.35 [sec]	1803.85 [sec]	4.47%	20 → 17
xz (B_1)	218.35 [sec]	216.89 [sec]	0.67%	20 → 13
xz (B_2)	206.52 [sec]	205.67 [sec]	0.41%	20 → 13
curl (B_1)	710.01 [sec]	652.01 [sec]	8.17%	20 → 7
curl (B_2)	707.35 [sec]	641.14 [sec]	9.36%	20 → 2
xterm (B_1)	201.06 [sec]	197.33 [sec]	1.86%	20 → 1
xterm (B_2)	204.92 [sec]	201.14 [sec]	1.84%	20 → 1

***RQ*$_3$ insights:** The incremental build of a configuration is correct between 48.03% and 67.50 % of the cases, based on binary size/symbols. It is faster and correct in 60% of the time.

4.4 Optimal ordering (RQ_4)

Now that we found that incremental build can be faster and correct, the question is to what extent can we leverage on that to outperform clean build of all configurations. To answer this research question, we use our whole data and not only the minimum time that we report in Table 2, that is, all measured times for the different combinations of incremental build. For this reason, we searched for an optimal ordering of configurations based on build time. To do that, we first start by building a directed graph $g = (V, E)$ where vertices V are all configurations and edges E are incremental builds between configurations [2]. We add an edge only when the incremental build is successful and is correct. Each edge has as a weight the minimum time of incremental build. These information comes from Table 2. Then, we add a root vertex that is linked with all other vertices by edges weighted with their respective clean build times. This is essential for configurations that cannot be built incrementally. After that we run a minimum spanning tree directed graph algorithm [19]. In this way, we calculate the optimal order of configurations that leverages to the best possible time on incremental build.

Table 4 shows our obtained results after running our algorithm on all incrementally build configurations for two batches on five subject systems. From the column `Gain`, we can observe that indeed we are able to find an optimal ordering that allows us to correctly build all configurations while being faster by performing incremental builds. The gain varies from 0.10% to 11.76%. Furthermore, we could observe that we were always reducing the required number of clean builds in a system, which vary from 1, 2, 7, 8, 13, 17 to 19 times. Where 1 meaning one necessary clean build upon which the rest of configurations are incrementally built, and 19 where only one incremental built was possible. In particular, `xterm`, `x264`, and `curl` did not need many clean builds and used several incremental builds contrary to `sqlite` and `xz`. For example, in the optimized order for `xterm` B_1 we had only one clean build and the rest was a tree of incremental builds for the rest of configurations. Whereas in `sqlite` B_2 used only three incremental builds and 17 clean builds.

[2]The graphs are available in our companion page.

***RQ*$_4$ insights:** When considering correct incremental builds, we could always find an optimal order that is faster than just a series of clean builds.

5 DISCUSSION

This section discusses the impacts of our results on three actors.

For developers. From our experiments, we observe that RQ_2 and RQ_4 highlight the benefits of incremental build, while RQ_1 and RQ_3 highlight its limits. Nonetheless, we see that the benefits overcome the limits, in particular, when finding the correct optimal ordering. Hence developers can already benefit from incremental build. A current limitation is that the optimal order is specific to a set of configurations and deserves an upfront computational investment. However there are several projects that have a predefined, fixed set of default configurations to build (*e.g.*, JHipster [21]). Furthermore, the investment can pay for itself with the frequency of commits and thus the use of incremental builds several times throughout the evolution.

Besides, developers should take into account the specifics of their projects and possibly fix their build artefacts to fully realize the potential of incremental build. For instance, the case of `xz` (see Section 4.3) is challenging: each build depends on a library that is also compiled during the same build to be used later. This kind of critical library on which the build depends must be forced to rebuild. Indeed, it is possible to specify that the build rule related to the library must be rebuilt each time, allowing it to propagate the build to its dependency and update the new configuration. Therefore, developers of configurable software can miss opportunities of relying on incremental build due to mismanagement of their build scripts.

For build system designers. Numerous new build systems have been recently developed to support the specific needs of organisations [3–6]. The design space of build system is still to be explored and the case of configurations adds a new dimension. Designers should give an interface between the build rules written by the developer and the build system's back-end which is building the project. In the case of Make, the issues observed when aiming to incrementally build in our case studies cannot be only spotted by the developer in charge of writing build rules. Hence configurations-aware build system are still to be designed and developed. We believe the limits and insights of our study can help.

For researchers. Our results calls for more research on the topic of incremental build of configurations. Researchers can evaluate the incrementality of existing build systems: most have been designed to support variability in time (evolution), not variability in space (configurations). We encourage researchers to assess the feasibility, correctness, and performance of build systems with the novel scenario of building successive software configurations as we did in the study. In general, there is still a lack of evaluation on a wider scope of incremental build of configurations with various build systems [3–6]. Researchers can design further empirical studies (*e.g.*, confirmatory studies) to gain further insights or validate some hypotheses of our work. A major specificity of incremental build of configurations is that there is an order to define. Given a set of configurations, the order of their incremental build can be defined given according to their distance. However, this notion of distance

has to be defined precisely. It can be based on the difference of options or the relation between options and their implementation and the impact on the build. It would be interesting to propose heuristics to find an order before the build, based on configuration similarities and build rules. In other words, how can we automatically find the orders in RQ_4? This hard and open problem can expand the applicability to an arbitrary set of configurations (*e.g.*, random).

6 THREATS TO VALIDITY

Internal validity. To measure the benefits and limits of incremental build, we had to build several pairs of configurations. To reduce the risk of interference with other running software, we isolated the build environment. To do so, we created one docker image per system with the needed build tools and only needed dependencies. The machine we used was dedicated to the experiment. However, some OS processes and services were still running on the computer that can bring measurements noises. Nevertheless, as we dockerized similarly every run of our experiments, we expect that the noises would be similar, in particular, that we did not launch any other task in parallel. We only run twice the experiments owing to the cost of computations. Moreover, there is a threat related to the sampling used to generate configurations. We deliberately used random sampling to diversify our data sets of configurations. However, we observe from Figure 5 that our configurations are relatively disjoint. Nevertheless, for each sample batch, we repeat the experiment process 2 times. Finally, to check for the correctness of an incremental build, we compared the size and symbols of the produced binary with the one produced with a clean build.

External validity. We experimented on five subjects that are C-based configurable software systems with the Make build system. Although we think that the incremental build would be applicable in other build systems and software technologies, we cannot generalize our results. Further experimentation is necessary.

Conclusion validity. Our experiments showed promising results for incremental build of configurations by accelerating the build time. We also shows limits when it comes to correctness. Even though, we could still find faster and correct incremental builds, which we used to find the optimal ordering of configurations. To have more insights and statistical evidence, further evaluation is needed on more subject systems and larger set of configurations.

7 RELATED WORK

Build systems. Many works exists on incremental build systems [3, 16, 17, 22, 29, 34, 37, 43, 49] but without handling a set of configurations. In Cao et al. [12], the authors forecast the duration of incremental build jobs for over 2 thousands of commits in GLib (library) and VTK (Visualisation ToolKit). While incremental build jobs vary in terms of duration, they propose a tooled approach, BuildMÉTÉO, to forecast how long a job will take based on the dependency graph extracted from a first cold build using MAKAO [8]. BuildMÉTÉO can estimate the build time of a project after some modifications in its files. However, it is considering only the evolution of one configuration of the project and not diverse ones like our preliminary study. In Cserep et al. [14], they introduce a way to detect only the necessary files to build by parsing the whole codebase. Instead of parsing every file from scratch each time which takes an important amount of time, they use incremental parsing. In addition, they also check the build rules modifications associated with the files. Hence a file to rebuild is a file that has been either modified or which the build rule has changed. By doing so, they do not check the binaries metadata such as timestamps like Make, and avoid issues we present in Section 4.3. Maudoux et al. [33] present in their paper that incremental build helps to save time on local builds. However, it not available yet on continuous integration (CI) platforms. Indeed, incremental build is not brought on CI because of some factors such as the correctness of the produced binary. In our paper, we show that the correctness of incremental build with real-world projects based on GNU Build System is not ensured due to Make's strategy to perform incremental build over configurations. In [29, 30], Konat et al. provide a DSL to increase the effectiveness of writing build scripts by using their proposed language of PIE. With such expressive build scripts, analysis and error detection could be prevented beforehand. They also introduce a build system that takes track of files and focus only on the part that changed, to avoid having too much information in memory and perform strictly the minimum while being effective.

Several empirical studies on build systems have been performed (*e.g.*, [23, 24, 31, 34, 35, 51]). For instance, a case study at Google reported a large corpus of builds and build errors mainly focusing on static compilation problems [44]. Beller et al. [11] performed an analysis of builds with Travis CI on top of GitHub. About 10% of builds show different behaviour when different environments are used. In our case, we are considering different configurations rather than environments. To the best of our knowledge, incremental build for software configurations has received little attention.

Software product line (SPL) and variability. The SPL community develops numerous methods and techniques to manage a family of variants (or products). Configurations are used to build variants and are subject to intensive research. Formal methods and program analysis can identify some classes of configurations' defects [13, 48], leading to variability-aware testing approaches (e.g., [18, 25–28, 32, 40, 45, 50]). The general principle is to exploit the commonalities among variants, mainly at the code level. For instance, variability-aware execution [10, 27, 40] instruments an interpreter of the underlying programming language to execute the tests only once on all the variants of a configurable system. Nguyen *et al.* implemented Varex, a variability-aware PHP interpreter, to test WordPress by running code common to several variants only once [40]. Reisner *et al.* use a symbolic execution framework to evaluate how the configuration options impact the coverage of the system given a test suite [41]. Static analysis and notably type-checking has been used to look for bugs in configurable software and can scale to very large code bases such as the Linux kernel [25, 26, 50]. Though variability-aware analysis is relevant in many engineering contexts, our interest differs and consists in studying the practice of concretely building a sample of (representative) configurations with an unexplored approach – incremental build.

There are several empirical studies about the build of SPLs and configurable systems. For instance, Halin *et al.* [21] report on the endeavour to build all possible configurations of an industry-strength, open source configurable software system JHipster, a popular code

generator for web applications. We are unaware of studies that consider incremental build of configurations.

8 CONCLUSION

In this paper, we conducted a novel study investigating the benefits and limits to incrementally build software configurations, as opposed to always cleaning as in conventional build. By considering five real-life configurable software systems, we explored whether incremental build works, outperforms a sequence of clean builds, is correct *w.r.t.* clean build, and can be used to find an optimal ordering of building configurations. Our results suggest that incremental build of configurations can reduce build time without trading correctness. Developers and maintainers can already benefit from this simple approach. Owing to the frequencies of build in continuous integration and their increasing cost, we encourage the software engineering community (build system designers, researchers, etc.) to further investigate incremental build of configurations.

As a future work, we plan to replicate our study with other build systems and more batches of configuration samples. We also plan to investigate the possibility of having an heuristic for finding automatically the optimal ordering. This is challenging as it requires to infer a priori the unknown "distances" among configurations. Finally, we aim to synthesize knowledge of patterns and anti-patterns of incremental build to increase benefits and reduce limits.

REFERENCES

[1] [n.d.]. Autoconf - GNU Project - Free Software Foundation. https://www.gnu.org/software/autoconf/

[2] [n.d.]. Automake - GNU Project - Free Software Foundation. https://www.gnu.org/software/automake/

[3] [n.d.]. A fast, scalable, multi-language and extensible build system. https://bazel.build/

[4] [n.d.]. Maven, a software project management and comprehension tool. https://maven.apache.org/

[5] [n.d.]. Ninja, A Small Build System with a Focus on Speed. https://ninja-build.org/

[6] [n.d.]. RECC. https://buildgrid.gitlab.io/buildgrid/user/using_recc.html

[7] 2021. KernelCI. https://kernelci.org/

[8] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. 2007. Design recovery and maintenance of build systems. In *Proceedings of the 23rd International Conference on Software Maintenance (ICSM)*, Ladan Tahvildari and Gerardo Canfora (Eds.). IEEE Computer Society, Paris, France, 114–123.

[9] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. 2020. Sampling effect on performance prediction of configurable systems: A case study. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 277–288. https://doi.org/10.1145/3358960.3379137

[10] Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. *ACM SIGPLAN Notices* 47, 1 (jan 2012), 165.

[11] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. 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* (Buenos Aires, Argentina) *(MSR '17)*. IEEE Press, Piscataway, NJ, USA, 356–367.

[12] Qi Cao, Ruiyin Wen, and Shane McIntosh. 2017. Forecasting the duration of incremental build jobs. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 524–528. https://doi.org/10.1109/ICSME.2017.34

[13] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering* 39, 8 (aug 2013), 1069–1089.

[14] Máté Cserép and Anett Fekete. 2020. Integration of Incremental Build Systems Into Software Comprehension Tools.. In *ICAI*. 85–93. http://ceur-ws.org/Vol-2650/paper10.pdf

[15] Jack Edge. 2020. The costs of continuous integration. https://lwn.net/Articles/813767/

[16] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015. A sound and optimal incremental build system with dynamic dependencies. *ACM Sigplan Notices* 50, 10 (2015), 89–106.

[17] Stuart I. Feldman. 1979. Make — a program for maintaining computer programs. *Software: Practice and Experience* 9, 4 (1979), 255–265. https://doi.org/10.1002/spe.4380090402 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380090402

[18] Stefan Fischer, Rudolf Ramler, Claus Klammer, and Rick Rabiser. 2021. Testing of Highly Configurable Cyber-Physical Systems – A Multiple Case Study. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems* (Krems, Austria) *(VaMoS'21)*. Association for Computing Machinery, New York, NY, USA, Article 19, 10 pages. https://doi.org/10.1145/3442391.3442411

[19] Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. 1986. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica* 6, 2 (1986), 109–122.

[20] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 301–311.

[21] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2019. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empir. Softw. Eng.* 24, 2 (2019), 674–717. https://doi.org/10.1007/s10664-018-9635-4

[22] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. 2015. Incremental computation with names. *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Oct 2015). https://doi.org/10.1145/2814270.2814305

[23] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 197–207.

[24] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 426–437.

[25] Christian Kastner and Sven Apel. 2008. Type-checking software product lines-a formal approach. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - ASE '08*. IEEE, 258–267.

[26] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development* (Eindhoven, The Netherlands) *(FOSD '10)*. ACM, New York, NY, USA, 25–32.

[27] Chang Hwan Peter Kim, Don S Batory, and Sarfraz Khurshid. 2011. Reducing combinatorics in testing product lines. In *Proceedings of the tenth international conference on Aspect-oriented software development (AOSD '11)*. ACM, 57–68.

[28] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. 2013. SPLat: lightweight dynamic analysis for reducing combinatorics in testing configurable systems - ESEC/FSE '13. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 257–267.

[29] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. 2018. Scalable incremental building with dynamic task dependencies. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 76–86. https://doi.org/10.1145/3238147.3238196

[30] Gabriël Konat, Roelof Sol, Sebastian Erdweg, and Eelco Visser. [n.d.]. Precise, Efficient, and Expressive Incremental Build Scripts with PIE. ([n. d.]).

[31] Carlene Lebeuf, Elena Voyloshnikova, Kim Herzig, and Margaret-Anne Storey. 2018. Understanding, debugging, and optimizing distributed software builds: A design study. In *2018 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 496–507.

[32] Jackson A. Prado Lima, Willian Douglas Ferrari Mendonça, Silvia R. Vergilio, and Wesley K. G. Assunção. 2020. Learning-based prioritization of test cases in continuous integration of highly-configurable software. In *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, Roberto Erick Lopez-Herrejon (Ed.). ACM, 31:1–31:11. https://doi.org/10.1145/3382025.3414967

[33] Guillaume Maudoux and Kim Mens. 2017. Bringing incremental builds to continuous integration. In *Proc. 10th Seminar Series Advanced Techniques & Tools for Software Evolution*. 1–6.

[34] Guillaume Maudoux and Kim Mens. 2018. Correct, efficient, and tailored: The future of build systems. *IEEE Software* 35, 2 (2018), 32–37.

[35] Guillaume Maudoux and Kim Mens. 2019. Lessons and Pitfalls in Building Firefox with Tup.. In *SATToSE*.

[36] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering software variability with FeatureIDE*. Springer.

[37] Neil Mitchell. 2012. Shake before building. *ACM SIGPLAN Notices* 47 (10 2012), 55. https://doi.org/10.1145/2398856.2364538

[38] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build systems à la carte. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–29. https://doi.org/10.1145/3236774

[39] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2020. Finding Faster Configurations Using FLASH. *IEEE Trans. Software Eng.* 46, 7 (2020), 794–811. https://doi.org/10.1109/TSE.2018.2870895

[40] Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. 2014. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering - ICSE '14*. ACM, 907–918.

[41] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S Foster, and Adam Porter. 2010. Using symbolic evaluation to understand behavior in configurable software systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10, Vol. 1)*. ACM Press, 445.

[42] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 342–352.

[43] Robert W. Schwanke and Gail E. Kaiser. 1988. Smarter Recompilation. *ACM Trans. Program. Lang. Syst.* 10, 4 (Oct. 1988), 627–632. https://doi.org/10.1145/48022.214505

[44] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' Build Errors: A Case Study (at Google). In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 724–734.

[45] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. 2012. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (LNCS, Vol. 7212)*. Springer, 270–284.

[46] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM, 284–294.

[47] Klaas-Jan Stol and Brian Fitzgerald. 2018. The ABC of software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 3 (2018), 1–51.

[48] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (2014), 6:1–6:45.

[49] Walter F. Tichy. 1986. Smart Recompilation. *ACM Trans. Program. Lang. Syst.* 8, 3 (June 1986), 273–291. https://doi.org/10.1145/5956.5959

[50] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (2018), 18:1–18:33. https://doi.org/10.1145/3280986

[51] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The impact of continuous integration on other software development practices: a large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 60–71.