HAL
open science

Towards Incremental Build of Software Configurations

Georges Aaron Randrianaina, Djamel Eddine Khelladi, Olivier Zendra, Mathieu Acher

To cite this version:

Georges Aaron Randrianaina, Djamel Eddine Khelladi, Olivier Zendra, Mathieu Acher. Towards Incremental Build of Software Configurations. ICSE-NIER 2022 - 44th International Conference on Software Engineering – New Ideas and Emerging Results, May 2022, Pittsburgh, PA, United States. pp.1-5, 10.1145/3510455.3512792 . hal-03558479

HAL Id: hal-03558479

https://hal.science/hal-03558479

Submitted on 4 Feb 2022

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.

Towards Incremental Build of Software Configurations

Georges Aaron Randrianaina, Djamel Eddine Khelladi, Olivier Zendra
Univ Rennes, CNRS, Inria, IRISA - UMR 6074
F-35000 Rennes, France
{firstname[-name2].lastname}@irisa.fr

Mathieu Acher
Univ Rennes, CNRS, Inria, IRISA - UMR 6074
Institut Universitaire de France (IUF)
F-35000 Rennes, France
mathieu.acher@irisa.fr

ABSTRACT

Building software is a crucial task to compile, test, and deploy software systems while continuously ensuring quality. As software is more and more configurable, building multiple configurations is a pressing need, yet, costly and challenging to instrument. The common practice is to independently build (*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 surprisingly not been considered for software configurations. In this vision paper, we formulate the hypothesis that incremental build can reduce the cost of exploring the configuration space of software systems. We detail how we apply incremental build for two real-world application scenarios and conduct a preliminary evaluation on two case studies, namely `x264` and Linux Kernel. For `x264`, we found that one can incrementally build configurations in an order such that overall build time is reduced. Nevertheless, we could not find any optimal order with the Linux Kernel, due to a high distance between random configurations. Therefore, we show it is possible to control the process of generating configurations: we could reuse commonality and gain up to 66% of build time compared to only clean builds.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**.

KEYWORDS

Highly configurable system, Build system, Incremental build

1 INTRODUCTION

Building software is a crucial activity in every non-trivial software project. Different artifacts are assembled, compiled, tested, and eventually deployed, hopefully with success. The rise of continuous integration (CI) has amplified this trend with build services integrated into major code platforms (*e.g.*, Github, GitLab). Their interest is to continuously ensure the quality of a project, both in terms of functional and non-functional properties (*e.g.*, security, execution time). Techniques such as distributed builds and parallelization of build tasks have been developed to speed-up builds. Although widely adopted, building software is increasingly complex and expensive in terms of time and other resources [5, 9, 17, 27].

Software configurations add further complexity. 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. How the build is realized can also change *e.g.*, with different compiler flags. Developers and maintainers want to ensure that, throughout evolution, all or at least a subset of software configurations build well. It is no surprise that many organizations build several software configurations of their projects as part of their CI. For instance, initiatives like KernelCI or 0-day build thousands of default or random Linux configurations each day [2]. Another example is JHipster, a popular Web generator that builds dozens of configurations at each commit, involving different technologies (Docker, Maven, grunt, etc) [15].

The common practice is to independently build a subset of configurations *i.e.*, in a fresh environment. In this paper we propose and explore an approach, called *incremental build of configurations*: instead of starting from scratch and cleaning the build artefacts, a configuration can be (incrementally) built from an existing and already completed configuration build. While incremental build has been considered for software evolution with small code edits, it has surprisingly not been considered for software configurations. The idea is to reuse artefacts of previous configurations builds and thus save some computations, hence resources, including time. Behind this idea, the real question is to quantify how much and where it can gain or lose compared to a more conventional build, in addition of already existing techniques such as distributed build. Moreover, this approach has risks: an incremental build might not work or might be incorrect compared to a conventional build, for example the build system might forget to recompile some necessary artefacts. Another unknown remains about the strategy to schedule the incremental build of configurations. Given a set of configurations, numerous possible orderings exist, possibly with different effects on correctness and overall build cost (e.g., CPU time). Does incremental build pay off whatever the ordering and the "distance" among configurations? Is it worth finding an optimal ordering? Our goal in this vision paper is to bridge the worlds of software build and configurable software. In particular, to push for an in-depth exploration of the above hypothesis and address, to the best of our knowledge, new open unaddressed questions: **(RQ1, efficiency)** Does incremental build outperform clean build? **(RQ2, correctness)** Is the result of incremental build the same as a clean build? **(RQ3, optimality)** Is there an order of configurations that brings an optimal (overall) incremental build time?

To investigate these questions, we detail how we apply incremental build for two real-world application scenarios (1) ordering over a fixed set of configurations; (2) generation of configurations in such a way the incremental build is efficient. We conduct a preliminary evaluation on two case studies, namely `x264` and Linux Kernel. For `x264`, we found that an optimal ordering of random software configurations exists and reduces overall build time. A caveat is that the correctness of the incremental build is sometimes not ensured for certain pairs of `x264` configurations and thus poses some challenges. In contrast to `x264`, we could not find any optimal order with the Linux Kernel due to a high distance between

configurations. Nonetheless, we show it is possible to control the process of generating configurations: we could reuse commonality and gain up to 66% of build time compared to only clean builds. In addition, we ensure that the generated configurations of Linux change as much as possible.

2 WHY INCREMENTAL BUILD?

This section introduces our vision, before presenting two real-world scenarios of incremental build of configurations

2.1 Incremental build of configurations

When testing a configurable system, developers often build a set of configurations. The common workflow is to build each of them from scratch. A build typically has a cost, such as CPU time, energy consumption, billed when done in a paid cloud service, etc. In the rest of the paper, we take as an example the cost function of CPU time. Figure 1a illustrates three configurations c_1, c_2 and c_3 of the same configurable software system, and their build from scratch times (respectively 10, 15 and 20 minutes).

The black node can be seen as an initial state, or a clean directory to launch the build task from. The edge from the initial state to the configuration represents the build task and the time it takes. Building these three configurations, to check if they compile or for testing, takes 45 minutes as shown on Figure 1a[1].

However, configurations of the same software system can share similarities: build rules, source files and even object files. Our hypothesis is that these similarities could lead to sharing: the files produced during the build of a configuration can be reused by other configurations built after. This is the main point of incremental build: avoiding redundant builds and processing only the required build rules to be efficient. Figure 1b shows a possible order to build the three configurations presented above: first build c_1 which takes 10 minutes; then compile c_2 and c_3 without going back to the initial state (e.g. without cleaning the directory) but reusing the compiled files from c_1's build. This reuse reduces build times for c_2 and c_3 from 15 and 20 min to 10 min. In the end, three configurations were built in 30 minutes instead of 45.

Therefore, an incremental build is intended to be efficient compared to a classic clean build. This efficiency could be measured by CPU load or energy consumed during the build.

However, incremental build may not always be recommended, due to correctness issues for some systems. Indeed, reusing files built for other configurations could introduce improper files into the configuration being built. Hence, the result of the incremental build could be different compared to a clean build for the same configuration. Hence, the *correctness* of incremental build of configurations must be ensured. To do so, the produced output of a clean build and an incremental build of the same configuration should verify an equivalence, according to a correctness criterion. Such a criterion can be two binaries having the same size or same number of symbols and the same name for each one of these symbols. It also can be a bit by bit comparison or a binary diff program which compares the binaries control flow graph.

[1]These 3 builds can be built in parallel (*e.g.*, using distributed resources).

2.2 [Scenario 1 (S1)] Improving overall build of a fixed set of configurations

When testing highly configurable systems, one usually has a set of configurations to build. This set contains either default standard configurations that represent the frequent configurations or a mix of random configurations. Instead of being built from scratch they could be built incrementally as in Figure 1b, benefiting from their similarities. Nevertheless, not all configurations share similarities. Some can be different up to the point that the build system cannot reuse any artefact from a previous build.

To detect beforehand if two configurations have some files in common that their builds could reuse, knowledge on the system is necessary. This knowledge is used to order the (incremental) build of the configurations in a way such that each incremental build performed uses as much materials from previous builds as possible. The previous build artefacts could be either the build directory or a dedicated cache. However, the goal remains the same: maximising the reuse of materials produced by previous builds and improving overall build time compared to only clean builds. With our set of configurations $\{c_1, c_2, c_3\}$, we can consider the minimum possible build time is 30 minutes (Figure 1b). This best build time is obtained because c_1 is built first, then c_2 and c_3 both benefiting from some c_1 artefacts. Knowledge of the system helps computing a heuristic to automatically find this order of builds, maximise artefacts reuse, and improve total build time. Starting with configuration c_2 then c_1 and c_3, the impact on total build time might be different.

One of the main challenges in this scenario is how to construct this knowledge (e.g., a heuristic) so as to find an optimal ordering of incremental builds for a given set of configurations.

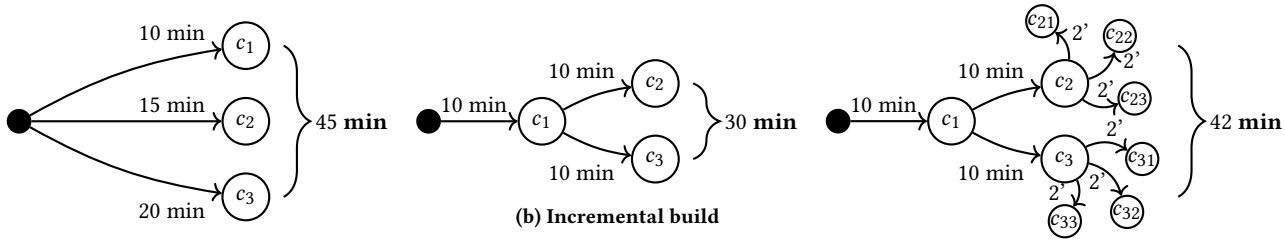
2.3 [Scenario 2 (S2)] Beyond building fixed configurations: exploring a larger space

In addition to building and improving total build time for a fixed set of configurations, we propose to generate configurations to explore and test a larger configuration space. Indeed, one of the best way to test a configurable software is to build *variants* of it[2].

The more configurable a software system is, the larger its configuration space, hence the more configurations have to be tested. We want to have high diversity of configurations in order to cover a configuration space as large as possible. In this scenario, for all picked configurations, we try to generate variant configurations "around them" in order to improve also what we call the *local diversity* in the configuration space. This way, given a starting configuration, close configurations can be generated such that the incremental build of the variants always outperform clean builds. We present this scenario in Figure 1c. We first build configurations such as c_1, c_2, c_3. Then, from e.g., c_2 we derive other configurations c_{21}, c_{22},... Then, we incrementally build these configurations from c_2. To accelerate even more this configuration space exploration, we can execute the incremental build of each generated configurations in parallel.

This technique is thus not only about mutating a configuration or generating new ones from a seed. We must ensure that the incremental build of the generated configurations from the original

[2] *"The Linux Kernel does not have a test suite.[...] The best thing you can ever do for us is: you just build the Kernel and tell us if you have a problem. That is our QA cycle."*. Greg Kroah-Hartman, at FOSDEM 2010.



(a) Clean, traditional build

(b) Incremental build

(c) Local diversity with incremental build

Figure 1: Incremental build scenarios

outperform clean builds and remains correct. For instance, we can control that the build of each generated configuration in Figure 1c takes 2 minutes. Instead of 3 clean builds in 45 minutes (Figure 1a), we can build 9 configurations in 42 minutes. Our main goal is to have a proper strategy to mutate the configurations in order to get correct incremental builds that outperform clean builds and can diversify enough to explore the configuration space.

3 PRELIMINARY RESULTS

We present in this section our preliminary results on experimenting incremental build of configurations over `x264` (scenario S1) and Linux (scenarios S1 and S2).

3.1 [S1] Ordering configurations of x264

`x264` is a software system for video encoding written in C. It has about 115.243 lines of code and 50 possible compile-time options (or features). We built two batches with 20 random generated configurations each. In this way, we perform $2 * 20$ clean builds and $2*(20*19) = 760$ incremental builds, i.e., 760 pairs of configurations $(c1, c2)$ where c1 is built with a clean build, and c2 incrementally build after c1. Thus, a total of 800 builds. To generate the random configurations, we use the random product generator in the FeatureIDE framework [33].

For the first batch, 7 out of 20 cases of incremental build were slower than clean build. For these cases, we recommend a simple clean build. We also had one case only where the binary produced by incremental build was not consistent with the binary supposed to be produced by a clean build: it did not have the same binary size. Nevertheless, with a combination of correct and most efficient pairs of incremental builds, we could still deduce an optimal order. Total CPU time using only clean builds was 747,99 seconds, whereas total CPU time relying on incremental builds in the optimal order plus clean build (for incorrect incremental builds) was 747,26 seconds, representing a non-significant gain of 0.10% of build time.

For the second batch, for each of these 20 configurations, we found an incremental build that outperforms its clean build. We further checked their correctness w.r.t. the criteria on the binaries and found that incremental build was always correct. After that, based on the most efficient pairs of incremental builds, i.e., the fastest observed incremental build, we could deduce an optimal order. The total CPU time using only clean builds was 754.92 seconds, whereas the total time of the incremental build in the optimal order was 666.12 seconds, representing a gain of 11.6% of build time.

> It is thus possible to order a set of configurations such that incremental build combined with clean builds gives a better performance than only clean builds.

3.2 [S1] Ordering configurations of Linux

In order to extend our results to a bigger software system, we replicated it on Linux. Linux is a kernel of operating system written in C. Linux is considered as the most complex configurable software system with over 15.000 features and about 28 millions of lines of code. We generated two batches of 20 configurations each and conducted the same experiment as `x264`. The configurations were generated using the Linux Kernel command line utility `randconfig`, which generates random configurations. Though the correctness holds, we could not find any pair of configurations in which incremental build was faster than clean build. After investigation, we found out that there was too much difference between two randomly generated configurations, with up to a thousand options added or removed. In addition, the strategies of the build system (Make) can also fail to reuse files and do redundant builds (see, e.g., [43]).

> It is thus not straightforward to find an optimal incremental build ordering for a given set of random configurations due to high difference between them.

3.3 [S2] Local diversity of Linux configurations

As ordering a fixed set of configurations seems to be difficult to find with Linux, we decided for another approach. Instead of picking all the configurations randomly, we extract knowledge from the source code to generate new configurations.

The Linux Kernel build system is composed of multiple makefiles at each directory level. Obviously, these makefiles contain build rules which compile and link files together. By enabling or disabling an option, the modification impacts the addition or not of build rules, hence, the compilation of some files and their dependency.

To determine before compilation which files will be compiled if an option is enabled (or disabled), we built a *dependency graph (DG)* between options according to their implementation at the file level. If two options are implemented in the same file, we link them with an edge in the DG. With this representation, we could over approximate the amount of files to recompile when an option is modified from the configuration. Then, we sort options according to the amount of files they would recompile if they were enabled or disabled. With this information, we can deduce which options can

be progressively added to increase diversity by small increments, instead of building from scratch every configuration.

We picked a random configuration and built it from scratch. Then we generated 9 configurations from it with the technique described before using the DG. The number of different options between each generated configuration and the original varies from 7 to 183. Total CPU time using only clean builds was 4.864,26 seconds, whereas total CPU time relying on one clean build of the initial configuration then incremental builds of the others was 1.648,87 seconds, representing a gain of approximately 66% of build time.

Thus, the generation of configurations to be built incrementally can be guided with knowledge from the system in order to outperform clean build.

4 RELATED WORK

Build systems. Many works exist on (incremental) build systems [1, 6, 10, 11, 16, 25, 31, 34, 37, 41] but they focus on code changes through the evolution of software (e.g. commits) rather than configurations. Cserep *et al.* [8] introduce how to detect only the necessary files to build with incremental parsing of the codebase. Maudoux *et al.* [30] show that incremental build could help speed up builds of continuous integration (CI), and Gallaba *et al.* [13] that caching can accelerate CI. An open issue is to adapt these techniques over distant software configurations that may have very different impacts on the files to build. Several empirical studies on build systems have been performed (*e.g.*, [17, 18, 26, 27, 31, 32, 44]). Beller *et al.* [5] 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 software configurations rather than environments.

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 or execute variants and are subject to intensive research. For instance, building variants is a necessary step before deriving performance prediction models [4, 14, 19, 20, 28]. Formal methods and program analysis can identify some classes of configuration defects [7, 40], leading to variability-aware testing approaches (e.g., [12, 21–24, 29, 35, 36, 38, 42]). 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 [21, 22, 42]. 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 (possibly distant and diverse) configurations with an unexplored approach – incremental build. Sampling configurations is subject to intensive research [3, 20, 39, 40]: incremental build brings new challenges (see also next section). There are several empirical studies about the build of SPLs and configurable systems. For instance, Halin *et al.* [15] report on the endeavour to build all possible configurations of the JHipster configurable software system. We are unaware of studies that consider or apply incremental build of configurations.

5 FUTURE WORK PLANS

A priori knowledge for ordering and generating configurations. As shown in Section 3, the brute force strategy of picking random configurations may not work in all cases. It is effective for `x264`. However, it does not bring benefits for Linux because of high differences among its random configurations. We plan to develop techniques to infer information about the system either from the source code analysis or directly from the build system, so as to deduce similarities among possible configurations and determine an *a priori* order to incrementally build them. In addition to configuration ordering, we plan to use inferred knowledge from the system to generate valid configurations while providing high diversity.

Correctness of build systems. The challenge is to obtain a binary produced from incremental build that is consistent with a binary obtained from a clean build. Section 3.1 reports on issues with some pairs of configurations. We aim to test existing build systems (*e.g.*, Make, Maven, Bazel, Pluto, etc.) correctness while performing incremental build over configurations instead of code additions such as commits. The correction of build systems is a notoriously difficult problem and configurations add further complexity.

Multi-objective problem and tradeoffs. There are several criteria: (1) the cost of building; (2) the correctness of the build; (3) the diversity of the targeted configurations. We fall into a multi-objective optimisation problem. A conservative approach that would slightly modify one option at a time when incrementally building has good chances of reducing the cost while being correct. Yet the diversity of configurations and their distances would be weak, which has limited interest for testing diverse configurations or learning over the configuration space [4]. Otherwise, a more aggressive approach is to pick random configurations with high diversity. Yet, as shown in Section 3.2, incremental build can be as costly as a traditional build. A research direction is to apply multi-objective algorithms or find a tradeoff suited for the specific needs of the software project.

6 CONCLUSION

We presented a novel problem and a vision that consists in bridging incremental build with software configurations. Results of preliminary experiments over `x264` and Linux Kernel show that incremental build can reduce the cost of building (*e.g.*, a gain of 66% over Linux when controlling the generation of configurations). Yet there are several challenges ahead related to the correctness of build systems, the diversity of configurations, and the ordering of the incremental builds. Many software projects are concerned with the problem of building multiple configurations into their CI; we encourage the software engineering community to more widely explore the potential of incremental build.

REFERENCES

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

[2] 2021. KernelCI. https://kernelci.org/

[3] 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

[4] Juliana Alves Pereira, Hugo Martin, Mathieu Acher, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2021. Learning Software Configuration Spaces: A Systematic Literature Review. *Journal of Systems and Software* (Aug. 2021). https://doi.org/10.1016/j.jss.2021.111044

[5] 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.

[6] 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

[7] 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.

[8] 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

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

[10] 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.

[11] Stuart I. Feldman. 1979. Make — a program for maintaining computer programs. *Software: Practice and Experience* 9, 4 (1979), 255–265.

[12] 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

[13] Keheliya Gallaba, Yves Junqueira, John Ewart, and Shane Mcintosh. 2020. Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. *IEEE Transactions on Software Engineering* (2020), 1–1.

[14] 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.

[15] 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

[16] 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).

[17] 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.

[18] 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.

[19] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer learning for performance modeling of configurable systems: an exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 497–508.

[20] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to sample: exploiting similarities across environments to learn performance models for configurable systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 71–82.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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

[26] Julia Lawall and Gilles Muller. 2017. JMake: Dependable Compilation for Kernel Janitors. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 357–366.

[27] 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.

[28] Luc Lesoil, Mathieu Acher, Xhevahire Tërnava, Arnaud Blouin, and Jean-Marc Jézéquel. 2021. The interplay of compile-time and run-time options for performance prediction. In *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A*, Mohammad Mousavi and Pierre-Yves Schobbens (Eds.). ACM, 100–111.

[29] 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*, Roberto Erick Lopez-Herrejon (Ed.). ACM, 31:1–31:11.

[30] 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.

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

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

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

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

[35] 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.

[36] 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.

[37] Robert W. Schwanke and Gail E. Kaiser. 1988. Smarter Recompilation. *ACM Trans. Program. Lang. Syst.* 10, 4 (Oct. 1988), 627–632.

[38] 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.

[39] Sabrina Souto, Marcelo d'Amorim, and Rohit Gheyi. 2017. Balancing soundness and efficiency for practical testing of configurable systems. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 632–642.

[40] 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.

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

[42] 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.

[43] Y. Zhang, Y. Jiang, Chang Xu, X. Ma, and Ping Yu. 2015. ABC: Accelerated Building of C/C++ Projects. *2015 Asia-Pacific Software Engineering Conference (APSEC)* (2015), 182–189.

[44] 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.