Incremental Build of Linux Kernel Configurations

Georges Aaron Randrianaina

Incremental Build of Linux Kernel Configurations

RANDRIANAINA Georges Aaron
Univ Rennes, CNRS, Inria, IRISA - UMR 6074
F-35000 Rennes, France
georges-aaron.randrianaina@irisa.fr

Abstract

Building software is a crucial task to compile, test, and deploy software systems while continuously ensuring quality. The Linux Kernel is the most configurable and complex system with more than 15,000 features. To speed up the building of such a large configuration set, and in contrast to the common workflow relying on only building clean configurations, we propose to *incrementally* build them.

Initial results do not provide any optimal order to incrementally build configurations due to a high distance between them. However, we show it is possible to control the configurations generation process: reusing commonality can save up to 66% of build time compared to only clean builds.

Keywords: Linux Kernel, Highly Configurable System, Build Systems, Incremental Build

1 Introduction

The Linux kernel offers over 15,000 features making it the most configurable and complex system. Indeed, the Linux Kernel is highly configurable: users can either enable – as an integrated feature or an external loadable module – or disable features and combine them to create a *configuration* [3]. The set of all possible configurations is called *configuration space*.

Problem. If each of these 15,000 features could be enabled or disabled to obtain a valid configuration, the size of the configuration space would be 2^{15000}. Logical constraints among features reduce this number but the configuration space remains huge. Thus building Linux configurations is crucial to test, deploy and learn from its non-functional properties for optimization or tuning. As stated by Greg Kroah-Hartman, Linux maintainer, at FOSDEM'10, *"The Linux Kernel does not have a test suite[...] for hardware interaction [...] 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."* Building a Linux configuration running Make [9] takes on average nine minutes [1]. Therefore, building a large amount of configurations is time and CPU consuming (about a week to build only 1000 configurations). Even though the goal is not to cover the entire configuration space, we aim to explore the combination of most common options.

Approach. We propose an approach to investigate the *incremental build* of configurations: rather than independently building each configuration and cleaning the build artefacts between each build, a configuration can instead be incrementally built from an existing and already completed configuration build. The idea is to reuse artefacts of previous configurations builds, hence saving some computations, resources, and time. Behind this idea, the real question is to quantify how much and where we can gain or lose compared to a more conventional build (in addition to already existing techniques such as distributed build). 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 files. 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?

2 Incremental build

We present in this section our approaches that contribute to the incremental build of Linux configurations.

Ordering of random configurations. We address a first scenario of incremental build of configurations. Given a set of configurations (such as multiple Linux default configurations) we need to find an order to incrementally build them such that this outperforms the overall build time of only clean builds. To this end, we propose to incrementally build every pair of the set and if the incremental build outperforms clean build we then investigate the build artefact to understand the reasons why incremental build is effective and how to use it to order the set.

Configuration generation. In this scenario, starting from an initial configuration, we generate a set of configurations in which incremental build is guaranteed to outperform clean builds. To determine before compilation which files will be compiled if an option is enabled/disabled, we build a dependency graph (DG) between options according to their implementation at file level. If two options are implemented in the same file, an edge in the DG links them. This representation can over approximate the number of files to recompile when an option is modified in the configuration. Then the options that trigger fewer files to (re)compile are selected to be added/removed in the new configurations. We add them progressively to increase diversity by small increments, instead of building from scratch every configuration.

Incremental build correctness. An incremental build is correct if it is "identical" to a clean build. Here we suggest two criteria: the symbols and the size of the produced Linux binary `vmlinux`. If they are identical for both incremental and clean builds, the incremental build process is considered safe.

3 Preliminary results

Ordering of random configurations. We ran the experiment with two batches of 20 random configurations[1] of Linux. 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.*, [35]).

Configuration generation. We picked a random configuration and built it from scratch. Then we generated 9 configurations from it with the technique described in Section 2 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 4864 seconds, whereas total CPU time relying on one clean build of the initial configuration then incremental builds of the others was 1649 seconds, a gain of approximately 66% of build time.

Incremental build's correctness. For all our experiments, correctness holds according to our criteria.

4 Future work

A priori knowledge for ordering configurations. As shown in Section 3, the brute force strategy of picking random configurations does not bring benefits for Linux because of high differences among these random configurations. We plan to develop techniques to infer knowledge directly from the build system [7, 31], so as to deduce similarities among possible configurations and determine an *a priori* order to incrementally build them.

Automatic generation of configurations. We plan to improve our configurations mutations to add/remove more options (beyond 183). We aim to automatize the whole process to explore the configuration space while effectively using incremental build. We plan to investigate how to bring the technique to continuous integration platforms, improving our builds with distribution and even more caching.

5 Related work

Large scale build infrastructures for Linux [13, 17, 22] exist that build hundreds of Linux configurations per day. Nevertheless, they only perform clean builds.

Many works exist about build systems [4, 5, 8, 9, 12, 20, 24, 25, 28, 33] but focus on code changes through the evolution of software (*e.g.*, commits) rather than configurations. Maudoux *et al.* [23] show that incremental build could speed up builds of continuous integration (CI), and Gallaba *et al.* [11] 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.

The Variability and Software Product Lines (SPL) community develops numerous methods and techniques to manage a family of variants (or products). Formal methods and program analysis can identify some classes of configuration defects [6, 32], leading to variability-aware testing approaches (*e.g.*, [10, 15, 16, 18, 19, 21, 26, 27, 29, 34]). Static analysis and notably type-checking have been used to find bugs in configurable software and can scale to very large code bases such as the Linux Kernel [15, 16, 34]. 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 [2, 14, 30, 32]; incremental build brings new challenges. We are unaware of studies that consider or apply incremental build of configurations.

6 Conclusion

We presented our approach that consists in bridging incremental build with software configurations. Results of preliminary experiments over the Linux Kernel show that incremental build can reduce the cost of building (*e.g.*, a gain of 66% when controlling the generation of configurations). We intend to test this approach on broader software projects, beyond Linux. Yet several challenges remain, related to the diversity of configurations and the ordering of incremental builds. Many software projects face the problem of building multiple configurations and we encourage the software engineering community to more widely explore the potential of incremental build (*e.g.*, combination with parallel or distributed build).

7 Acknowledgments

I would like to thank my PhD advisors: Mathieu Acher, Djamel Eddine Khelladi and Olivier Zendra.

[1] Generated using `randconfig`

References

[1] M. Acher, H. Martin, J. Alves Pereira, A. Blouin, D. Eddine Khelladi, and J.-M. Jézéquel, *Learning From Thousands of Build Failures of Linux Kernel Configurations*, technical report, Inria ; IRISA, June 2019.

[2] J. Alves Pereira, M. Acher, H. Martin, and J.-M. Jézéquel, *Sampling effect on performance prediction of configurable systems: A case study*, in Proceedings of the ACM/SPEC International Conference on Performance Engineering, 2020, pp. 277–288.

[3] S. Apel, D. Batory, C. Kstner, and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation*, Springer Publishing Company, Incorporated, 2013.

[4] Bazel, *A fast, scalable, multi-language and extensible build system.*

[5] Q. Cao, R. Wen, and S. McIntosh, *Forecasting the duration of incremental build jobs*, in 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2017, pp. 524–528.

[6] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin, *Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking*, IEEE Transactions on Software Engineering, 39 (2013), pp. 1069–1089.

[7] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann, *A robust approach for variability extraction from the linux build system*, in Proceedings of the 16th International Software Product Line Conference - Volume 1, SPLC '12, New York, NY, USA, 2012, Association for Computing Machinery, p. 21–30.

[8] S. Erdweg, M. Lichter, and M. Weiel, *A sound and optimal incremental build system with dynamic dependencies*, ACM Sigplan Notices, 50 (2015), pp. 89–106.

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

[10] S. Fischer, R. Ramler, C. Klammer, and R. Rabiser, *Testing of highly configurable cyber-physical systems – a multiple case study*, in 15th International Working Conference on Variability Modelling of Software-Intensive Systems, VaMoS'21, New York, NY, USA, 2021, Association for Computing Machinery.

[11] K. Gallaba, Y. Junqueira, J. Ewart, and S. Mcintosh, *Accelerating continuous integration by caching environments and inferring dependencies*, IEEE Transactions on Software Engineering, (2020), pp. 1–1.

[12] M. A. Hammer, J. Dunfield, K. Headley, N. Labich, J. S. Foster, M. Hicks, and D. Van Horn, *Incremental computation with names*, Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, (2015).

[13] Intel, *0-day, an automated linux kernel test service*, Online; accessed 2022.

[14] P. Jamshidi, M. Velez, C. Kästner, and N. Siegmund, *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, 2018, pp. 71–82.

[15] C. Kastner and S. Apel, *Type-checking software product lines-a formal approach*, in 2008 23rd IEEE/ACM International Conference on Automated Software Engineering - ASE '08, IEEE, 2008, pp. 258–267.

[16] A. Kenner, C. Kästner, S. Haase, and T. Leich, *Typechef: Toward type checking #ifdef variability in c*, in Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development, FOSD '10, New York, NY, USA, 2010, ACM, pp. 25–32.

[17] KernelCI, *a community-led test system focused on the upstream linux kernel*, 2021.

[18] C. H. P. Kim, D. S. Batory, and S. Khurshid, *Reducing combinatorics in testing product lines*, in Proceedings of the tenth international conference on Aspect-oriented software development, AOSD '11, ACM, 2011, pp. 57–68.

[19] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim, *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, 2013, pp. 257–267.

[20] G. Konat, S. Erdweg, and E. Visser, *Scalable incremental building with dynamic task dependencies*, in 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2018, pp. 76–86.

[21] J. A. P. Lima, W. D. F. Mendonça, S. R. Vergilio, and W. K. G. Assunção, *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, R. E. Lopez-Herrejon, ed., ACM, 2020, pp. 31:1–31:11.

[22] H. Martin, M. Acher, J. A. Pereira, L. Lesoil, J.-M. Jézéquel, and D. E. Khelladi, *Transfer learning across variants and versions: The case of linux kernel size*, IEEE Transactions on Software Engineering, (2021), pp. 1–17.

[23] G. Maudoux and K. Mens, *Bringing incremental builds to continuous integration*, in Proc. 10th Seminar Series Advanced Techniques & Tools for Software Evolution, 2017, pp. 1–6.

[24] ———, *Correct, efficient, and tailored: The future of build systems*, IEEE Software, 35 (2018), pp. 32–37.

[25] N. Mitchell, *Shake before building*, ACM SIGPLAN Notices, 47 (2012), p. 55.

[26] H. V. Nguyen, C. Kästner, and T. N. Nguyen, *Exploring variability-aware execution for testing plugin-based web applications*, in Proceedings of the 36th International Conference on Software Engineering - ICSE '14, ACM, 2014, pp. 907–918.

[27] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, *Using symbolic evaluation to understand behavior in configurable software systems*, in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1 of ICSE '10, ACM Press, 2010, p. 445.

[28] R. W. Schwanke and G. E. Kaiser, *Smarter recompilation*, ACM Trans. Program. Lang. Syst., 10 (1988), p. 627–632.

[29] J. Shi, M. B. Cohen, and M. B. Dwyer, *Integration Testing of Software Product Lines Using Compositional Symbolic Execution*, in Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering, vol. 7212 of LNCS, Springer, 2012, pp. 270–284.

[30] S. Souto, M. d'Amorim, and R. Gheyi, *Balancing soundness and efficiency for practical testing of configurable systems*, in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 632–642.

[31] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, *Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem*, in Proceedings of the Sixth Conference on Computer Systems, EuroSys '11, New York, NY, USA, 2011, Association for Computing Machinery, p. 47–60.

[32] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, *A classification and survey of analysis strategies for software product lines*, ACM Computing Surveys, 47 (2014), pp. 6:1–6:45.

[33] W. F. Tichy, *Smart recompilation*, ACM Trans. Program. Lang. Syst., 8 (1986), p. 273–291.

[34] A. von Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel, *Variability-aware static analysis at scale: An empirical study*, ACM Trans. Softw. Eng. Methodol., 27 (2018), pp. 18:1–18:33.

[35] Y. Zhang, Y. Jiang, C. Xu, X. Ma, and P. Yu, *Abc: Accelerated building of c/c++ projects*, 2015 Asia-Pacific Software Engineering Conference (APSEC), (2015), pp. 182–189.