CATCHing: Content-Based Caching of Build Artifacts and Test Results

Yasuf Haidaryar, Marvin Petz, Philip Suskin,
Arshia Khademian, Ahmad Azkour

March 2022

Abstract

The typical workflow in software development involves frequent rebuilding of projects. In particular, rebuilding often occurs in order to execute tests that ensure the continuing validity of selected parts of the project in question. As a result, even changes pertaining only to a small part the codebase can result in rebuilding the entire project, leading to large rebuild times that disrupt the project workflow significantly.

In this paper, we offer the CATCHing approach as a solution to the aforementioned issues. CATCHing is comprised of two key components, one of which generates hashes based on the *AST* of given source files to represent their semantic state, and the other evaluating the last time of semantic change of a source file through comparison of hashes generated in the build process.

We evaluated the benefit that our implementation provides through rigorous testing of the invocations of the compiler, linker, and archiver while rebuilding various randomly generated C projects under CATCHing in comparison to commonly used build systems. The generator for the C projects in question was tested for the resemblance of its generated projects to real-world projects in order to increase the reliability of our test results.

The tests validate that under most circumstances, CATCHing saves a significant amount of linker and archiver calls.

1 Introduction

Depending on the size of a software project, rebuild times can cut significantly into the development process. In particular, changes that do not impact the majority of the project result in a full project rebuild. Taking into account that frequent rebuilding and testing is routine in most projects, rebuilding greatly impedes the efficient workflow of a developer. Our approach, *CATCHing*, solves this problem by generating a semantic-fingerprint [6] through hashes created by cHash [1], as these hashes uniquely represent a state of the AST (abstract syntax tree) of a source file and can thereby (through comparison with the previous

hash of a given source file) be used to identify a significant change. Using these two components allows us to rebuild targets if and only if the corresponding source files have encountered semantically significant changes.

1.1 Story

As a software developer, one writes scripts which constantly require adjustments and improvements. The applied changes require the project to be rebuilt, which takes time. The larger a project becomes, the more time it takes to rebuild it, even if the scope of the changes (the source files in which the changes were made, as well as all dependent source files and targets) stays the same. This stands in contrast to the goal of developers, where maximizing efficiency and thereby saving time whenever possible is crucial to maintain an effective workflow. Rebuilding projects is essentially equivalent to waiting and therefore wasted time, which needs to be reduced wherever possible.

A lot of the modern software projects rely on compiled languages, which all suffer under the same condition, namely that the projects have to be rebuilt for every slight or significant change made to its code. Our team, among many others, pose the question: is every recompilation necessary for the project to run properly? If not, how can redundant compilation be minimized?

Compiling certainly is not unnecessary if the changes done to the source code influence its functionality, but up to 97% of compiler invocations are redundant [1] and therefore result in lost time and resources.

This sets the tone for our goal and contribution: we effectively reduce the time spent for rebuilding by eliminating unnecessary compilations and linking steps. But how can we even distinguish relevant from irrelevant changes made to source code?

For this, we use the condensed state of a build artifact in the form of a *semantic fingerprint* to dynamically evaluate which build steps are necessary.

2 System Model

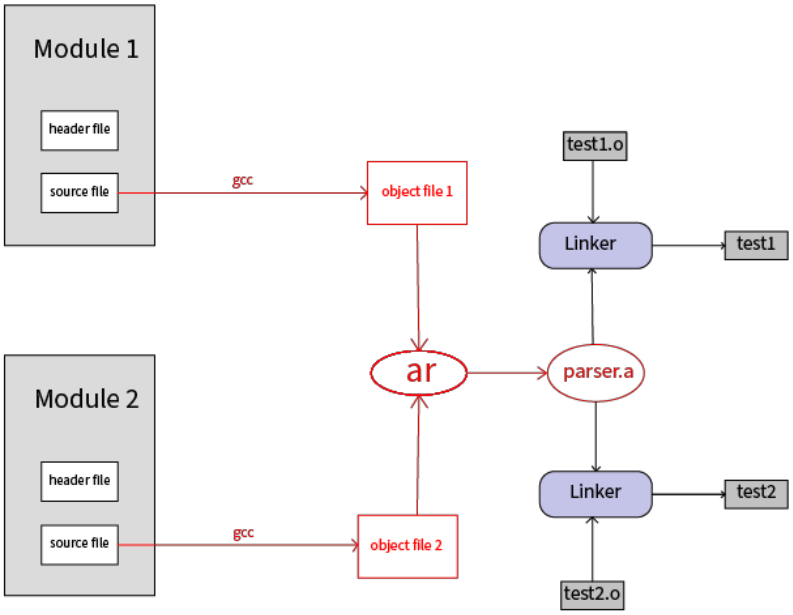


Figure 1: Build Process

When describing our approach to speed up the building process of a project, we have to define some terms. A *build step* describes either the action of compiling a source file to an object file, adding object files to a static library or linking object files to create an executable.

In order to organize these build steps, we need a *build system*. Its primary function is to build or rebuild a software project with the simple press of a button. It has to run the necessary build steps in the correct order to create valid binaries, which includes rebuilding objects files that depend files that have been changed.

If a source file contains functions, structures or variables from other files, it becomes dependent on those files and has to include them. In that sense, static libraries are also dependent on the object files they include, as changing those will also change the library. Depending on the overall structure, this may include a significant part of the entire project which is why we want to avoid redundant rebuilds whenever possible with our solution.

In addition to this, the build system outputs additional files, like reports or log files, which may be insightful for developers. These files are called *build artifacts*.

3 The CATCHing Approach

In general, a build system has to execute a build step, which deterministically produces the same output if started with the same inputs, only if (a) the destination artifact does not exist, or (b) an input artifact has *actually* changed. However, the timestamp-based (re-)compilation regime, which was introduced by `make` [8], is still the state of the art: if any input artifact (e.g., source file) has a newer UNIX timestamp than the destination artifact (object file), `make` invokes the build step to (re-)create the destination. Hence, `make` over-approximates the re-execute decision and often executes steps that will yield the same results.

With CATCHing, we provide a much stricter, and therefore faster, build methodology that uses *semantic fingerprints* for its re-execute decision. In the following, we will introduce semantic fingerprints at the example of cHash [1] and describe how we make use of these fingerprints in a fingerprint-aware variant of `make`, which we call Fpr-Make [6].

3.1 cHash

cHash is an extension of the Clang compiler and arguably the most important building block of CATCHing. During the build process, cHash creates a hash of every source file in the project. It is unique, because it focuses on the semantics of the source code: instead of creating a hash on the syntactic level, cHash creates a hash over the Abstract Syntax Tree (AST) of every source file. Therefore, the hash is unchanged if the AST is not affected (e.g., a comment is added or a struct is declared but not used). Ideally, the hash will only change if the build target changes after the build step. The hash is generated during compilation. If the hash did not change, we can abort the build step before linking starts, saving the most time consuming phase of the build process.

3.2 Fpr-Make

Fpr-Make is a modified version of the commonly known and used `make` [8], implementing three extended attributes to the files generated during the build process. The *fpr-hash* is evaluated during compilation by a pre-determined tool with the capability of generating hashes from files, such as, in our case, cHash. The *fpr-time* is defined by the "true" last modification time of a file, which is set for a given target after compilation of its source if and only if the hash generated for the source differs from the hash it corresponded to before compilation. The final extended attribute, *fpp*, is a boolean variable which signifies when the interjected extended attribute modification is finished.

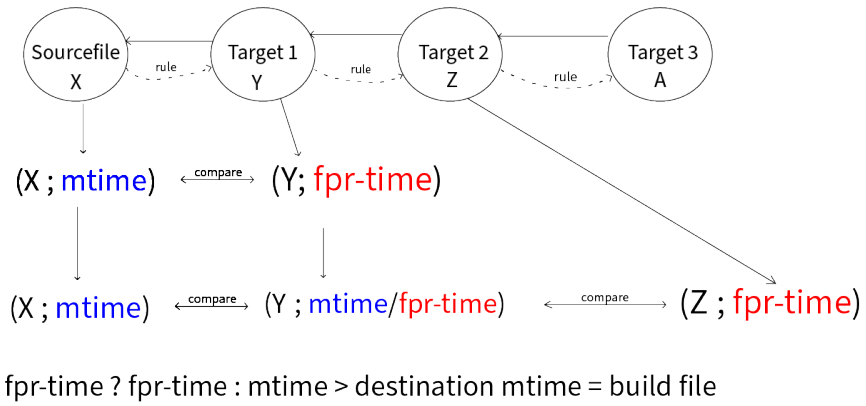


Figure 2: Semantic-Fingerprint-Make

As explained in Figure 2, if the source file already has an associated fpr-time, Fpr-Make will compare it with the *mtime* (last modification time) of the destination file. If the source file has no fpr-time, it will compare the mtime of the source file with the mtime of the destination file. As such, Fpr-Make only (re-)builds targets if their mtime is older than the fpr-time/mtime of the source file.

3.3 Implementation

Porting cHash to use its generated hash during the execution of Fingerprint-Make relies on modifying the extended attributes of the source file(s) compiled by Clang with the accompanying cHash plugin, and using the Clang and cHash binding for the build rules of all source files, as well as Clang with an additional md5 hash generation for the build rules of all other files. These rules are written into a Makefile that is executed by Fpr-Make. The method for executing a rule for the compilation of a given source file is shown in Figure 3.

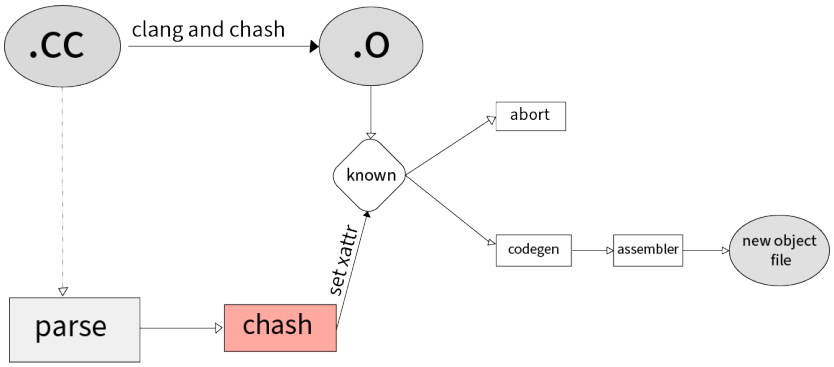


Figure 3: Compilation with Clang and cHash

3.4 Validating our implementation

To improve, test, and finalize the core of our project (merging cHash and Fingerprint-Make), we required a method to test our findings on each of the respective projects with the help of a trivial C project. One could create such a kind of *skeleton project* by hand, but this idea was not practical and, more importantly, not scalable. Instead, we developed a generator for this class of projects, which created project structures with varying levels of complexity on which to test the effects of cHash and Fingerprint-Make. Specifically, we wanted to get accustomed to identifying when and why the hash generated by cHash changed, based on the change in source code (i.e., adding a comment, adding trivial code, making true semantic changes), as well as investigating the capabilities of avoiding unnecessary recompilation of modules by using Fingerprint-Make on a dummy generated Makefile.

4 Evaluation

We wanted to not just create a static example project for validation and testing, but instead mimic the workflow of a real software project with frequent changes and rebuilds. Therefore, we created a generator that creates a random dependency graph and a corresponding mock project in C. By dependency graph, we mean a description of the project structure based on which functions (and thereby indirectly which modules and libraries) depend on each other. If function A calls function B, then A is dependent on B. The project generator takes a JSON file representing the dependency graph of the project and outputs source files, header files and a makefile.

Combining these programs, we are able to create multiple projects with arbitrarily many functions, modules and static libraries. The structure has to feature at least one library containing at least one module, which is a unity of a source file and a header file. The source files feature primitive functions and global variables. These functions may call other functions from other source files creating dependencies between modules from the same library or other libraries.

Furthermore, our generator creates executables calling the functions randomly, just like a developer would. Since real software projects are often tested each time they're rebuilt, we also implemented an option to generate executables mimicking these kinds of tests.

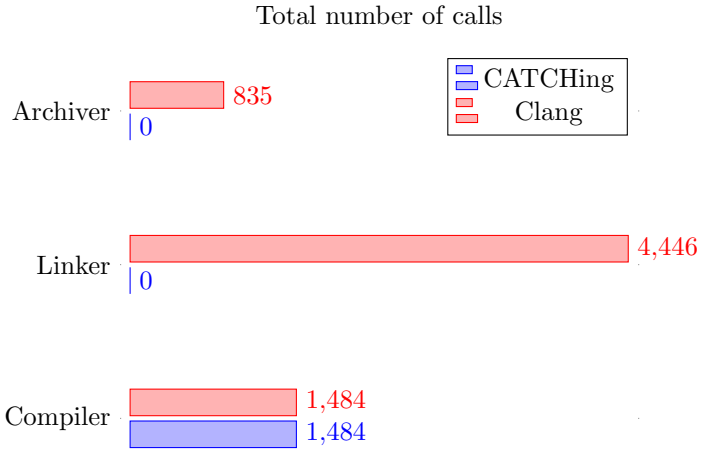
With our mock project set up, we created an additional layer of software allowing us to introduce changes randomly and rebuilding the project after a given number of changes. Our software either introduces comments to source files, which do not change the final executable, "soft changes", which consist of structs being added to header files but not used in any source files (this way, we make a real change in code which has no effect on the AST), and "hard changes", which change the behavior of the executables and require rebuilding parts of the project.

We also implemented a feature in which batches of random changes are se-

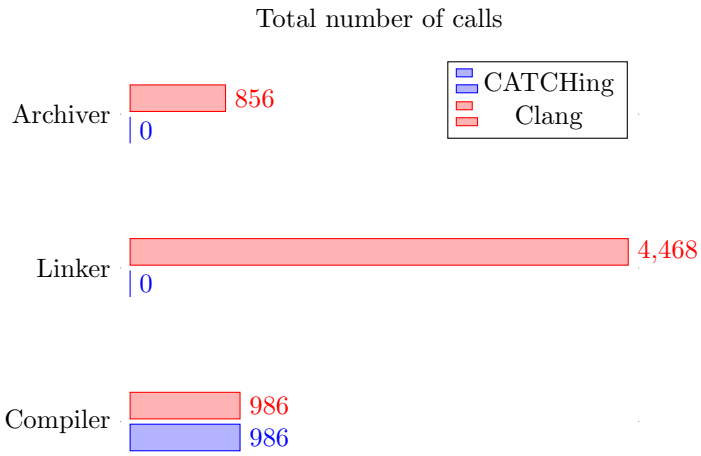
quentially applied to two copies of a randomly generated project. After each application of changes, the projects are rebuilt with Clang and CATCHing respectively.

With this, our method for evaluation was ready to be applied. We generated multiple projects of varying sizes and introduced the different types of changes to them to observe CATCHing's improvement.

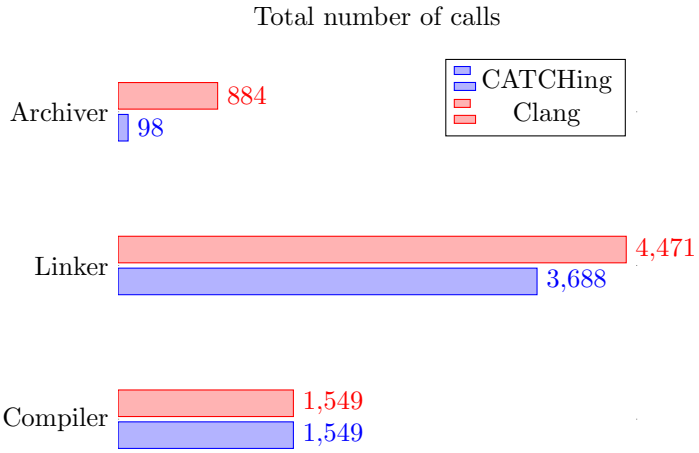
We started with a larger project of 200 functions, 15 modules, 10 libraries and 45 binaries and measured how often the project has to recompile, link or archive any files. Using this, we determined how often redundant build steps were detected and avoided. This can be observed in the following results:



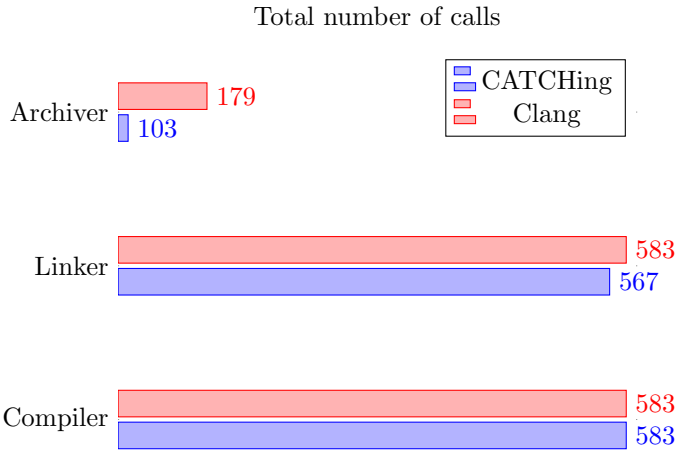
Comments are reliably detected and the build step was always aborted after compilation across multiple test runs. In the above run, we sequentially introduced 100 comments to the project and rebuilt it after every change.



Soft changes were also reliably detected. As long as changes do not alter the abstract syntax tree, no rebuilds are necessary. Once again, we introduced 100 changes.



Although hard changes require rebuilds by definition, we were still able to reduce the linker calls by up to 17%.



We also observed that for larger projects with more dependencies, the system has the largest benefit with respect to hard changes. A smaller project with 20 functions, 5 libraries and 8 binaries, like the one above, only yielded slight improvements.

4.1 Random Dependencies

We set it as our goal to not just create hollow projects, but projects with true structure and complexity that were comparable (or, at the very least, as realistic as possible) to genuine C projects in regards to their call graph structure. This involved consideration of parameters such as number of incoming dependencies, number of outgoing dependencies, number of functions per module, and avoiding/minimizing cyclic dependencies. A function dependency graph (call graph) alone is, in general, not capable of modelling an entire C project. Our next step was to create *clusters* (equivalent to modules) around *nodes* (equivalent to functions) in the dependency graph and *cluster groups* (equivalent to libraries) around said clusters, to reach the full extent of the C project architecture.

The randomly generated call graph is generated in the form of a directed acyclic graph (DAG) derived from an adjacency matrix, which is generated by setting $2 \cdot \sqrt{n}$ elements of the lower diagonal of a $n \times n$ matrix (where n is the number of functions to generate) to 1, while setting all other elements to 0. Following this, cycles are added randomly with a respective probability of cycle creation (bernoulli) and cycle extension (binomial) for each edge. After this, modules are generated around exclusive sets of functions, just as libraries are generated around exclusive sets of modules. Figure 4 depicts an exemplary dependency graph generated by the random dependencies project:

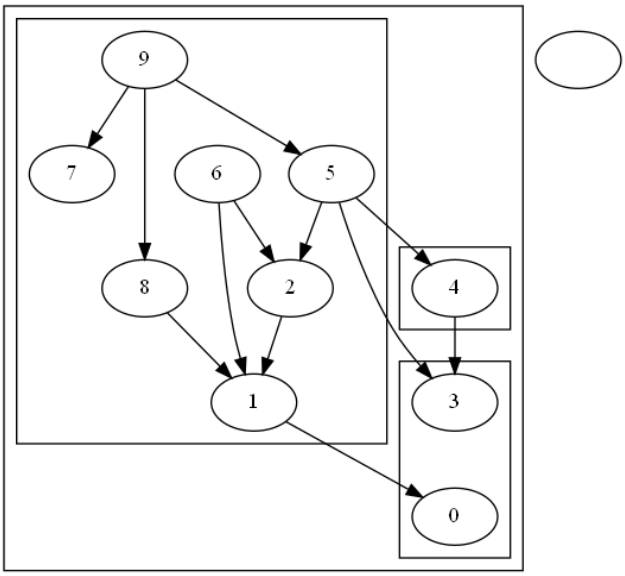
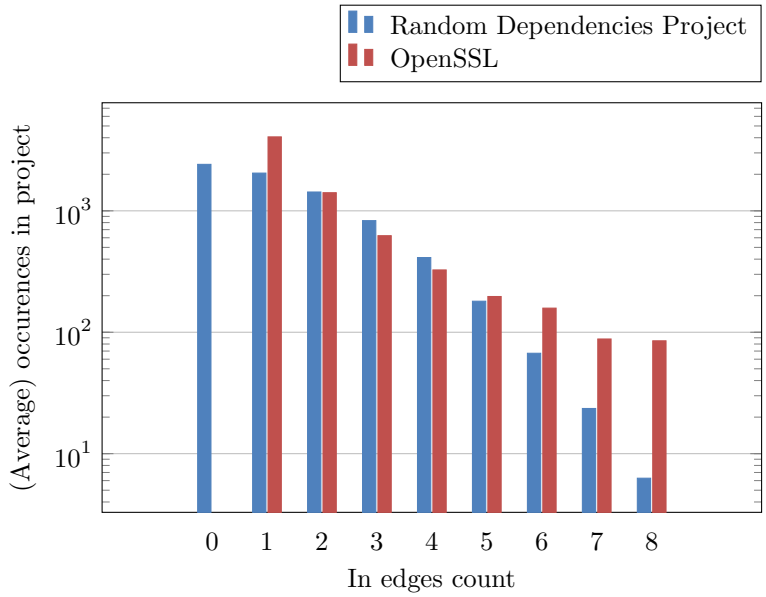
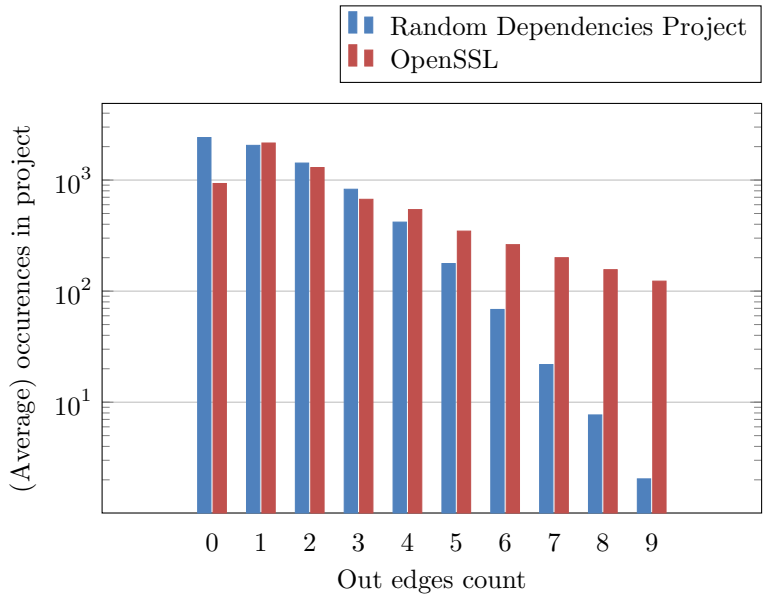


Figure 4: randomly generated dependency strcture with 10 functions, 3 modules, and 1 library

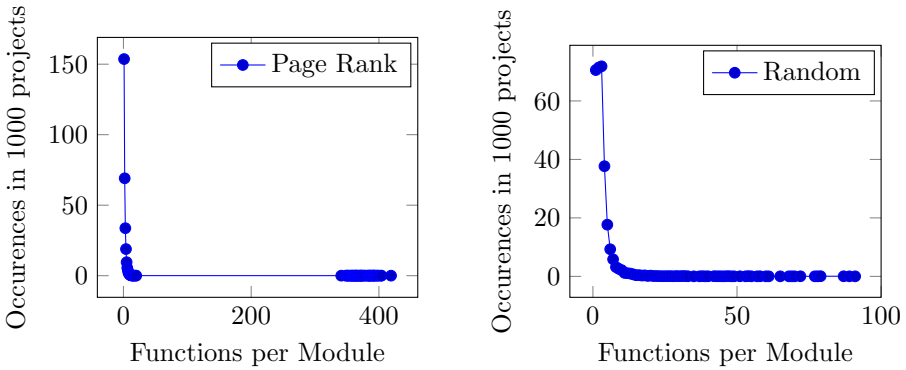
To evaluate the "realisticness" of graphs generated by this method, we compared the number of in and out edges per node (function) with that of an existing project, namely OpenSSL [9] 1.1.1m:





We also had to make considerations toward the number of functions per module. The two methods we adopted to group functions into modules were cluster generation via PageRank as well as purely random grouping. To compare

the two, we evaluated the average cluster sizes per project under PageRank and random grouping across 1000 projects with 7411 functions (the number of functions used in the previous test):



5 Discussion of our results

5.1 Random Dependencies

Upon further analysis of the in/out nodes, we recognize one major difference, namely that the OpenSSL project doesn't contain any functions with 0 in edges (this can be imagined as a function that isn't called by any function), whereas the random dependecies project does. This is because of the fact that the OpenSSL data by principle only contains functions which are also called, whereas the random dependencies project is meant to be a project with the necessary source code to generate libraries which most definitely can contain surface level functions that are intended to be solely called externally. Aside from this observation, the two graphs appear to be generally comparable and lead to a satisfactory generation of functions and function dependencies.

Concerning the cluster generation, a major difference is observable between PageRank and random generation. However, considering we didn't make use of weighted edges when connecting nodes, since function dependencies can be generally viewed as "weightless", it is no surprise that PageRank yielded unusual results. To be precise, PageRank was much more polarized in its cluster sizes, generally generating 1 large cluster alongside many miniscule ones, including multiple single-node clusters, equivalent to a single-function module (which is entirely possible, but not particularly desirable/realistic in large quantities within one project). Therefore, we conclude that random clustering results in more realistic project structures.

5.2 CATCHing

Our evaluation of compiler, linker, and archiver invocations has validated that CATCHing can reliably detect changes that do not affect the abstract syntax tree. Changes in the form of comments or declarations of variables will not trigger recompilation. Even semantic changes to the code will result in less redundant builds, especially when applied to a large project with many dependencies, displaying the potential savings in time. In combination, this is promising, as smaller changes/improvements in large projects will not be met with a large time penalty in comparison to ordinary, unoptimized build systems.

6 Conclusion

All things considered, we present CATCHing with as a feasible method to effectively reduce rebuild times and thereby greatly improve the overall software development experience. Although results may vary among projects and software environments, we conclude that the time overhead that arises from generating and comparing semantic fingerprints is largely outweighed by the time won by minimizing redundant build steps through the use of CATCHing, as long as the project in question is not exceedingly small and the changes made to the source code don't affect an overwhelming majority of the entire project. In comparison to state-of-the-art build systems, CATCHing is thereby shown to yield a consistent benefit.

References

[1] Christian Dietrich, Valentin Rothberg, Ludwig Füracker, Andreas Ziegler and Daniel Lohmann. "CHash: Detection of Redunant Compilations via AST Hashing", *Friedrich-Alexander Universität Erlangen-Nürnberg & Leibniz Universität Hannover.*

[2] Anonymous Authors. "TASTING: Reuse Test-Case Execution by Global AST Hashing". *EuroSys '22, April, 2022, Rennes, France.*

[3] Gregg Rothermel, Roland J. Untch, Chenguyun Chu and Mary Jean Harrold, 1999. Test Case Prioritization: An Empirical Study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM ´99).* IEEE Computer Society, USA.

[4] H. K. N. Leung and L. White. 1989. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance -1989. 60-69.* https://doi.org/10.1109/ICSM.1989.65194.

[5] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. *IEEE Trans. Softw. Eng, 22,8* (Aug. 19696).

[6] Tom Kaiser. 08.2017. In *Vermeindung redundanter neutübersetzungen durch die weitergabe von semantischen Fingerabdrücken* Universitiy of Leibniz Hannover. (Aug. 2017).

[7] Farhad Ahmed Sagar. sept. 2016. In *Cryptographic Hashing Functions - MD5. https://cs.indstate.edu/ fsagar/doc/paper.pdf.* College of Art and Science. USA.(sept.2016).

[8] 'GNU Make Manual - GNU Project - Free Software Foundation'. Accessed 15 March 2022. https://www.gnu.org/software/make/manual/.

[9] Welcome to the OpenSSL Project. C. 2013. Reprint, OpenSSL, 2022. https://github.com/openssl/openssl.