The Need for Speed: Increasing the Code Review Velocity
Kudrjavets, Gunnar

DOI:
10.33612/diss.781738808

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2023

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.
Chapter 6

The Unexplored Options to Increase Code Velocity

Abstract

Observational evidence from the software industry indicates that engineers frequently need to submit successive versions of code reviews to satisfy the CI system requirements. One reason is required cascading updates to the dead code to fix issues such as changes to the API signatures or other internal contracts. Another purpose is the necessary fixes to compiler warnings in various build configurations a particular project supports. An efficient solution is to ensure that a software project contains a minimal amount of dead code so that we can reduce the number of iterations in code reviews. Similarly, if there is no evidence that fixing compiler warnings is beneficial, we should either disable or ignore them. This chapter investigates the current state of quantifying the benefits of deleting dead code and fixing compiler warnings.
6.1 On Quantifying the Benefits of Dead Code Removal

6.1.1 Introduction

Dead code is code that is either needlessly executed or code that a system contains but never runs. Engineers consider the presence of dead code as an undesirable attribute of the code base [Neville-Neil, 2020]. Unnecessary code is a subcategory of technical debt and a “bad code smell” [Romano, Vendome, Scanniello, & Poshvyanyk, 2020]. It is also a cause of maintenance-related issues [Eder et al., 2012]. The presence of dead code is one of the categories in the Common Weakness Enumeration system [CWE—CWE-561: Dead Code (4.6), 2023].

Dead code is harmful to the effectiveness of code maintenance, but it impacts security even more.

The first automated attempt to remove dead code happens during the compilation. Compilers use techniques such as interprocedural optimization [Debray, Evans, Muth, & De Sutter, 2000] to remove dead code. Compiler optimizations are ineffective in optimizing code that an application loads during the run-time (e.g., using reflection). Approaches like static analysis can supplement compiler optimizations [Haas, Niedermayr, Roehm, & Apel, 2020]. Tools can efficiently determine if code is not called and optimize it away. These decisions require human intervention.

Automation cannot identify code that becomes unnecessary because of either changing requirements or obsolete features.

Organizations use various human-centric approaches to remove dead code systematically. They can establish virtual teams consisting of “janitors” [Morgensthaler, Gridnev, Sauciuc, & Bhansali, 2012] or designate a set of engineers as the “Code Cleanup Crew.” Organizations may also implement dedicated global engineering efforts that are either scheduled or continuous. Those engineering efforts are practical with the company’s “own” code. However, they fail to improve the code quality of third-party open-source software dependencies Companies also resort to a mix of material and social rewards. For example, engineers who delete at least $N$ LOC acquire a dedicated badge on an internal employee profile page, get a t-shirt, and become members of the “Dead Code Society.” The “Dead Code Society” was the name for an initiative at Facebook to delete dead code [Novati, 2023]. The movement later also extended to Pinterest [Parise, 2019].
6.1.2 Problem

Each organization or project has several competing priorities. Organizations can invest engineering effort into developing new features, fixing existing defects, or improving a product’s performance. Intuitively engineers know the tax dead code imposes on the development process. In *commercial software development*, organizations decide where to invest engineering resources by calculating the estimated return on the investment.

There are no straightforward approaches to estimating the value gained from deleting dead code.

Compared to other engineering activities, the benefits of which are well-known, calculating the estimated return on investment for dead code deletion presents a challenge.

The existing framework of “the bigger the number of deleted LOC, the better” is overly simplistic. This valuation scheme does not correctly quantify the benefits gained from deleting dead code.

Not all lines of code are equal.

The benefit from dead code deletion depends on a variety of contextual factors such as (a) abstraction level (e.g., kernel mode versus user mode), (b) performance cost (e.g., variable initialization versus creating a thread), and (c) exposure and intended usage (e.g., primary attack surface versus a rarely used feature).

6.1.3 Challenges to the research community

The research community can help practitioners by

- conducting studies on systems that use a systematic approach to delete dead code (e.g., an effort to “prune” code in OpenBSD [Unangst, 2015]) to investigate if other system characteristics such as performance, quality, or the perceived cleanliness of code base change as a result

- providing means to compose guidance about the order in which to focus on different system components (e.g., based on abstraction level, code coverage, or attack surface)

- defining methods to quantify the benefits of deleting dead code that are more precise than the number of LOC deleted.
6.2 On Quantifying the Benefits of Fixing Compiler Warnings

6.2.1 Opportunity and motivation

One of the earliest stages of software development during which bugs can be detected is when new code changes are compiled. A compiler can flag potential issues found in the code. The cost of fixing a software defect increases significantly during the later phases of the development cycle [Tassey, 2002]. It is optimal to correct problems as early as possible because even minor bugs can result in catastrophic consequences [Zhivich & Cunningham, 2009]. Classical memory safety related bugs, common to programming languages such as C and C++ (e.g., a double-free), are the reasons behind approximately 70% of security updates Microsoft issues each year [Matt, 2019]. For Microsoft, the cost of fixing a bug resulting in a security bulletin is approximately $100,000 [Howard & Leblanc, 2002, p. 11]. Modern compilers (e.g., Clang, GCC, and MSVC) can detect typical programming mistakes such as integer overflows or underflows, out-of-bounds errors, memory management problems, etc., either during the compilation phase or via using runtime sanitizers. Acting on compiler warnings enables engineers to fix defects early and prevent the cascading set of failures the bugs would have otherwise caused. Despite the potential benefits of heeding compiler warnings, industry attitudes are mixed. For Linux kernel development, it took thirty years to start treating warnings as errors [Torvalds, 2021]. Google takes a somewhat contrarian approach by aiming to never issue compiler warnings because they find that developers ignore them [Winters et al., 2020, p. 427]. The warnings are either enabled as errors or never shown in the compiler output.

In our experience, demonstrating the value of fixing compiler warnings or changing organizational culture to treat compiler warnings as a first-class defect prevention tool is challenging. Often, changes in attitude and engineering processes only take place after damaging events (e.g., critical services becoming inaccessible, irrecoverable data loss, zero-day exploits) have already manifested.

6.2.2 Existing evidence and guidance

Existing research and empirical data about the benefits of fixing compiler warnings is minimal. Most of the data comes from grey literature related to writing secure code or anecdotal knowledge passed down from experienced practitioners.
Microsoft practices recommend using the highest level of warnings to inspect code for potential security vulnerabilities and compile “cleanly” without any errors or warnings [Howard, 2006; Howard & Lipner, 2006]. The downside of not fixing the warnings is articulated in a case study on a large code base where integer-related warnings had been disabled. Analysis reveals that about 20% of the hidden warnings contain potentially exploitable conditions [Howard & Leblanc, 2007]. A post-mortem analysis from Facebook finds that enabling all compiler warnings as errors reveals issues such as memory leaks, infinite recursion, and catastrophic bugs where a compiler would “optimize” away critical functions [Kumar, 2019]. Reducing the attack surface and finding opportunities to clean up code during the maintenance phase is another suggested application for utilizing compiler warnings [Pearse & Oman, 1995].

We can find only one paper investigating the correlation between compiler warnings and defects [Moser, Russo, & Succi, 2007]. The study finds experimental evidence that “[a] large number of compiler warnings of a source file is an indicator that the file contains also an above-average number of defects.” The conclusion is based on a limited amount of data and uses a version of GCC from 2006. Given the advances in compiler technology during the last 16 years and the size of industrial code bases (e.g., in 2017 the Windows code base contained 3.5 million files) we need additional studies utilizing the latest versions of compiler toolsets and larger projects [Harry, 2017]. The remaining discoverable research related to compiler warnings is focused on their correctness, readability, and validity [Barik, Ford, Murphy-Hill, & Parnin, 2018; Sun, Le, & Su, 2016].

6.2.3 Observations from industry

The trend we observe is that an engineer’s experience and seniority are directly related to his or her attitude towards fixing warnings. The more experience with the cost and consequences of basic programming errors the engineers have, the more appreciative they are of ensuring the correctness of the code as early as possible.

From a technical point of view, we rarely observe projects treating warnings as errors and triggering build breaks as a result. Turning on all possible warnings is mainly done by engineers developing compilers themselves. Very few projects in industry have a zero-tolerance policy towards the presence of compiler warnings. A rare example is safety-critical code, e.g., software developed by NASA [Holzmann, 2006]. We have not been able to find any public data regarding standards related to compiler warnings in other companies producing safety-critical software, e.g., Airbus, Boeing, Tesla, etc.
A variety of reasons contribute to compiler warnings either not being fixed or deprioritized. The main reason is the lack of empirical evidence to show either correlation or causal relationship between fixing compiler warnings and decrease in defect density. Another key reason is the cost of adapting stricter compiler warning levels to legacy code. Techniques such as treating warnings as errors are time-consuming to implement unless projects established this policy from the very beginning. We cannot discount the impact on an engineer’s career as well. The lack of external motivation to fix the compiler warnings is often caused by the fact that preemptively fixing compiler warnings does not get rewarded as well as the post hoc activity associated with debugging and bug fixing. The repetitive nature of fixing the compiler warnings is another factor making long-term code quality improvement initiatives unpopular. The number of warnings to be analyzed may reach into hundreds, thousands or even tens of thousands depending on the size of the code base. A key reason related to engineers not willing to fix compiler warnings is distrust in the validity of the warnings due to past experiences with false positives. This belief can be summarized as “if warnings would indicate real problems, then they would be errors instead”.

### 6.2.4 Can Mining Data About Compiler Warnings Help?

**Background and motivation**

Compiler warnings that indicate potential problems in the code are a byproduct of a daily “edit-compile-debug” development cycle. If warnings are present, an engineer must decide whether to fix, ignore, or suppress the warnings. Industry practices for commercial and open-source projects range from a zero-tolerance policy towards warnings [Holzmann, 2006; Nethercote, 2017; The Apache Software Foundation, 2021] to stating that fixing warnings introduces more defects [SQLite, 2022].

Ignoring critical compiler warnings can cause defects to propagate into a production environment. Those defects, in turn, can trigger issues such as reduced availability of services, memory corruption, and zero-day vulnerabilities. We need empirical data to develop sound, evidence-based policies to determine if and in what order to address compiler warnings.

Google’s experience with applying static analysis at a large scale indicates that it is hard to motivate developers to fix potential issues without a clear incentive [Sadowski, Aftandilian, Eagle, Miller-Cushon, & Jaspan, 2018]. In our industry experience, we make concurring observations regarding prioritizing fixing compiler
warnings. Lack of evidence that can provide the incentive makes it challenging to determine the priority of this work. Outside of the warning level ranking provided by the compiler, we are not aware of any other prioritization techniques to determine what warnings should be fixed first. Our industry experience indicates that some warnings are more beneficial than others. Benefit in this context means that warnings detect serious defects, have less false positives, and engineers trust and understand the diagnostic reports [Barik et al., 2017; Becker et al., 2019]. We anecdotally observe that the order in which engineers fix the warnings does not always match the severity level a compiler assigns to them. To determine the types of warnings that engineers either continuously ignore, fix immediately, or delay fixing, we propose that researchers mine the existing build logs and investigate this topic further.

Availability of data

CI is a practice of continuously integrating new code changes into a shared code base [Fowler, 2006b]. The CI systems are widely used in industry by major companies such as Google [Memon et al., 2017] and Meta [Distefano, Fähndrich, Logozzo, & O’Hearn, 2019]. The adoption of CI systems for open-source software is growing as well [Hilton et al., 2016]. Several popular open-source projects such as FreeBSD [Hsu, 2017] or Mozilla [Lampel, Just, Apel, & Zeller, 2021] have embraced the CI systems as the primary code delivery and validation vehicle.

The CI process results in a large number of build artifacts. In 2017, Google executed approximately 800,000 builds on a daily basis [Memon et al., 2017]. The build logs contain compiler output that includes the list of warnings that were generated during the compilation process. Typically, each CI build tracks also what new code changes are included in that build. Popular dynamic and static analysis frameworks such as CodeChecker enable inspecting the differences between two builds [Márton & Krupp, 2020]. In our industry experience, the resulting build logs persist anywhere from weeks to months. As a result, a variety of data from both industry and open-source is available for analysis.

6.2.5 Future research directions

We recommend that researchers partner with practitioners working on open- and closed-source software to focus on following topics:
1. **Explore the current state.** What are the default warning levels, attitudes and sets of beliefs toward fixing the warnings? Are they influenced by software’s technical abstraction level?

2. **Investigate the relationship (or lack thereof) between compiler warnings and defects, team productivity, and product risk.**

3. **Establish baseline metrics related to compiler warnings.** For example, warnings per file, per KLOC, change in the ratio of warnings with the application of stricter levels of compilation, number of suppressed warnings per KLOC?

4. **Rank warning categories according to their precision and recall.** Propose a recommended set of warnings per compiler.

5. **Conduct case studies about projects having zero-tolerance policy towards warnings.** Is the approach cost-effective outside the scope of safety-critical software?

6. **Evaluate the economics (e.g., negative impact) of fixing warnings.** SQLite development team finds that “[m]ore bugs have been introduced into SQLite while trying to get it to compile without warnings than have been found by static analysis” [SQLite, 2022].

7. **Variation between programming languages.** How similar are or should be warnings for low-level (e.g., C), functional (e.g., OCaml), or scripting (e.g., Ruby) languages?

The repository of CI logs can help researchers find answers to several questions such as:

1. What warnings have the longest and shortest lifespan? Does the order of fixing those warnings correspond with compiler’s classification scheme and suggestions?

2. Are warnings from some compilers fixed faster than others (e.g., Clang versus GCC)? What about different programming languages or static analysis tools?

3. Is there a relationship between various characteristics such as abstraction level, amount of code churn, programming language, or seniority of engineers and how fast the warnings are fixed?

4. What is the temporal change direction in the ratio of warnings to source lines of code? Does the ratio depend on different project characteristics?
Answering these questions will help engineers prioritize fixing compiler warnings, suppress the ineffective warnings, and provide feedback to compiler developers. Combing the data mined from CI logs with the information from the defect tracking database and source control system will provide even stronger evidence to substantiate any claims.

6.3 Pragmatic evaluation strategies

Evaluating the effectiveness of fixing the compiler warnings and the benefits of deleting dead code is complex and potentially time-consuming. Developer folklore considers both practices beneficial for increasing the code base’s quality. The amount of empirical evidence in software engineering research that supports these beliefs is limited. In this section, we propose several approaches that can help the industry to gather the information that helps to evaluate the benefit of both practices.

6.3.1 Compiler warnings

An ideal way to determine if a compiler warning is valid will be to ask engineers to keep track of each time a compiler issues a warning and record the resulting action. An engineer can compile their code changes from tens to hundreds of times before submitting them for a code review. Asking practicing engineers to record this information manually will likely result in low participation because of the overhead.

A potential approach to automate the process is instrumenting the projects’ build environment, keeping track of the build warnings, and using a set of heuristics to determine the usefulness of warnings. A reasonable assumption is that if an engineer works on a fixed set of code changes, warnings appear, and the engineer fixes them during the next build cycle, then the warnings are valid.

In the industrial setting, where the code must support multiple platforms and build types, it is typical to use CI/CD systems that execute tens to hundreds of builds to cover each configuration. Mining the data from all these builds logs about what warnings engineers fix will help to determine which ones engineers consider helpful.

The evaluation needs to consider the individual project’s policy. For example, a policy can require an engineer to fix a specific category of warnings regardless of their perceived validity. A suitable candidate project will be without mandatory policy and engineers who can decide which warnings to fix.
6.3.2 Dead code

A desired experimental design to verify the benefit of deleting code is a randomized controlled trial (RCT) [J. M. Kendall, 2003]. The possibility of conducting that trial with open-source software is low. Mandating that two or more groups adhere to either systematically deleting dead code or not is unrealistic. The open-source developers have no incentive to follow any external guidance. In addition, this approach will require all the comparison groups to collect a set of metrics that quantify various project characteristics (e.g., the number of post-release defects). Based on the author’s observations, most open-source projects lack the infrastructure to collect and store these metrics.

A more realistic approach will be using an industrial setting with products undergoing multiple release cycles. A more extensive software system like Microsoft Office consists of several isolated products with decades of history. Using an RCT design or incentivizing the deletion of dead code for several releases can provide additional insight. It will be challenging to eliminate all the confounding variables, such as the complexity of features, team composition, or competing priorities. However, historical data (e.g., defect counts, performance issues, security patches) coupled with the interviews and surveys with engineers with in-depth knowledge of the project can indicate if “things are improving.”

A significant problem for measuring the effectiveness of fixing compiler warnings or removing dead code is the time it takes for effects to manifest. A notable example of removing dead code is an open-source software project called LibreSSL that forked from the OpenSSL project. The OpenBSD developers initiated the LibreSSL project after the Heartbleed vulnerability in the OpenSSL [Unangst, 2014]. In this case, the most objective measurement of the effectiveness is the presence of CVEs (Common Vulnerabilities and Exposures) over a longer time, typically years.