Related work

Dead code, or inactive sections that no longer add to a program's functioning, is a common problem in the field of software development. The inefficiencies caused by dead code become increasingly noticeable as codebases get larger, which reduces performance and limits development agility. In order to clarify complexity and provide creative solutions, this paper focuses on the crucial issue of detecting and eliminating dead code. Our effort adds to the larger conversation on software efficiency in addition to addressing the demand for lighter, efficient code. We provide methods for methodically locating and eliminating dead code using a combination of static and dynamic analysis, opening the door to more effective and maintainable software systems.

Dead code is frequently inserted into source code by malicious code writers in order to thwart reverse engineering research. When opaque predicates are combined with dead code, there aren't many ways to find it. This work presents a program slicing based approach that combines static and dynamic slicing analysis for dead code discovery. Additionally, creating a mechanism for detecting dead code within the LLVM compiler architecture. The suggested approach is utilized to identify the dead code that has been added to certain benchmark programs. The suggested method offers a high detection rate, according to the testing data.

The solution combines manual iteration with tool help to find and eliminate unnecessary classes at the class level. Using static analysis,creating a class dependency network based on known entry points, such as primary methods. By detecting transitively reachable classes to identify definitely used classes, and manage runtime-loaded classes via reflection technologies. The use of a profile to collect runtime data aids in the refinement of the study. Discover classes loaded by reflection by comparing statically inaccessible classes to executed classes. The iterative technique eliminates potentially unnecessary classes until runtime-executed and statically inaccessible classes no longer overlap. Please keep in mind that this method does not discover classes that are referenced in code but are not used (a more in-depth study is required). ConQAT was used to implement the solution in Java.

Web applications are typically constructed using HTML, CSS, and JavaScript, often incorporating third-party libraries and frameworks for enhanced productivity. However, this adoption can introduce JavaScript dead code—code implementing unused features—leading to detrimental effects on app loading time and resource usage. Our study introduces Lacuna, an approach for automatically identifying and eliminating JavaScript dead code in web apps. Lacuna supports both static and dynamic analyses, offering flexibility across coding styles and JavaScript constructs. Through empirical experiments on 30 mobile web apps, we applied Lacuna with varying optimization levels, assessing run-time overhead in terms of energy consumption, performance, network usage, and resource utilization. Results reveal that removing JavaScript dead code positively impacts mobile web app loading times and significantly reduces network transfer sizes.

Reference

[https://ieeexplore.ieee.org/document/8250352]

[https://teamscale.com/hubfs/26978363/Publications/2014-dead-code-detection-on-class-level.pdf]

[https://www.researchgate.net/publication/370276301_JavaScript_Dead_Code_Identification_Elimination_and_Empirical_Assessment]