

Code Optimizer: Removal of Dead Code

What is Code Optimization:

Code optimization is the process of improving the performance and efficiency of code without altering its output or behavior. It involves refining the code to make it faster, use less memory, or consume fewer resources. Common techniques include removing dead code (code that never affects the program), reducing redundant calculations, simplifying loops, folding constants at compile time, and minimizing memory usage. Code optimization can be done manually by developers through better algorithm choices or refactoring, and automatically by compilers using optimization flags. Tools such as Polyspace or static analyzers can also assist in identifying optimization opportunities. The goal is to produce cleaner, faster, and more resource-efficient programs while maintaining correctness.

Optimization Techniques:

- Dead Code Elimination
- Constant Folding
- Loop Optimization
- Function Inlining
- Common Subexpression Elimination
- Strength Reduction
- Memory Optimization
- Peephole Optimization
- Code Motion (Loop-Invariant Code Motion)
- Inline Expansion
- Instruction Scheduling
- Tail Call Optimization
- Register Allocation
- Unrolling Loops
- Branch Prediction Optimization

Static code refers to the original source code written by a programmer that does not change while the program is running. It is the fixed, human-readable code stored in files (like .py, .c, or .java) before compilation or execution. Static code is important because it can be analyzed without running the program a process known as **Static code analysis**. This analysis helps identify syntax errors, unused variables, type mismatches, security vulnerabilities, and violations of coding standards early in the development process. Since static code remains unchanged during runtime, it provides a reliable way to detect potential issues and improve code quality before deployment.

Ex:

```
int x = 10;

int y = 0;

int z = x/y; Division by zero Error

return 0;
```

- Analyzes the **source code without running it**
- Detects that b might be zero → possible **division by zero**
- Even if no inputs are given, it checks **all code paths**

Dynamic code analysis:

Dynamic code analysis is the process of analyzing a program, in order to observe its actual behavior during execution. Unlike static code analysis, which reviews the source code without execution, dynamic analysis checks how the program performs with real inputs, in real time. It can detect runtime errors like memory leaks, crashes, invalid pointer access, and performance bottlenecks that static analysis might miss. This method often involves tools like debuggers, profilers, test frameworks, or runtime analyzers such as Valgrind, AddressSanitizer, or Python's cProfile. Dynamic code analysis is especially useful for testing how software behaves under different conditions, ensuring reliability, security, and efficiency during execution.

Ex:

We can take the same example from static code but the will be compiled and the Error will be displayed in run time

- Actually executes the program
- With $y = 0$, this triggers a runtime crash
- Cannot detect the bug until the line is run

Difference between static and dynamic code analysis:

Feature	Static Code Analysis	Dynamic Code Analysis
When it's done	Before the program runs (at compile or code level)	While the program is running (runtime)
Code Execution	❌ Not executed	✅ Executed
Purpose	Detects potential bugs, vulnerabilities, coding issues	Finds actual runtime errors and performance issues
Input Data Needed	❌ Not required	✅ Required
Types of Issues Found	Syntax errors, dead code, null pointers, coding standard violations	Memory leaks, crashes, performance bottlenecks
Tools	Polyspace, Pylint, Cppcheck, SonarQube	Valgrind, AddressSanitizer, gprof, pytest
Speed	Faster analysis, works on all paths	Slower, depends on input coverage

What is Polyspace:

Key Features of Polyspace:

1. Static Code Analysis

- Analyzes the code **at compile time**, without needing to run it.

2. Finds Critical Errors

- Detects:
 - Null pointer dereferences
 - Division by zero
 - Array out-of-bounds
 - Unreachable code
 - Overflow and underflow

3. Formal Verification

- Uses mathematical techniques (like abstract interpretation) to **prove** if certain runtime errors can or cannot occur.

4. Coding Standard Compliance

- Supports checks for MISRA-C, MISRA-C++, CERT C, AUTOSAR, etc.

5. Clear Code Annotations

- Colors code lines to show status:
 - **Green** = proven safe
 - **Red** = proven faulty
 - **Gray** = not proven

6. Two Main Tools:

- **Polyspace Bug Finder**: Detects bugs, security flaws, and code quality issues.

- **Polyspace Code Prover:** Proves the absence of certain classes of runtime errors.

Polyspace Bug Finder:

Polyspace Bug Finder is a static analysis tool from **MathWorks** that automatically detects, and **code quality issues** in C and C++ source code **without executing it**. It's designed for **fast feedback during development**, making it ideal for early-stage debugging and continuous integration.

Example:

```
#include <stdio.h>

void display(int arr[], int size) {
    int i;
    int total;

    for (i = 0; i <= size; i++) { // Off-by-one error
        total += arr[i];        // Use of uninitialized variable
    }

    printf("Total: %d\n", total);
}

int main() {
    int nums[4] = {1, 2, 3, 4};
    display(nums, 4);
    return 0;
}
```

Polyspace Bug Finder Will Report:

total may be **used without init** - Uninitialized variable

i <= size may cause **out-of-bounds** - Array index bug

After Fixing:

```
#include <stdio.h>

void display(int arr[], int size) {
    int i;

    int total = 0; // Initialize variable

    for (i = 0; i < size; i++) { // Fix loop condition (i < size)
        total += arr[i];
    }

    printf("Total: %d\n", total);
}

int main() {
    int nums[4] = {1, 2, 3, 4};

    display(nums, 4);

    return 0;
}
```

Uninitialized variable - total initialized to 0

Off-by-one loop error - Changed i <= size to i < size

Polyspace Code Prover:

Polyspace Code Prover is a **formal verification** tool developed by MathWorks that **mathematically proves** the **absence of certain runtime errors** in C and C++ code

without executing it. It's widely used in **safety-critical systems** such as automotive, aerospace, and medical software.


Example:

```
#include <stdio.h>

void divide(int a, int b) {
    int result = a / b;
    printf("Result: %d\n", result);
}

int main() {
    divide(10, 0); // Passes zero as divisor
    return 0;
}
```

Polyspace Code Prover Analysis:

Possible division by zero -  Red - Proven to cause runtime failure

After Fixing:

```
#include <stdio.h>

void divide(int a, int b) {
    if (b != 0) {
        int result = a / b;
        printf("Result: %d\n", result);
    } else {
        printf("Error: Division by zero\n");
    }
}
```




```

}

int main() {
    divide(10, 0);
    return 0;
}




```

Runtime-safe (never divides by zero) -  Green

Safe conditional branch -  Green

Difference Between Bug Finder and Code Finder:

Feature	Bug Finder	Code Prover
	Detect likely bugs (fast static analysis)	Prove absence of runtime errors (formal verification)
Analysis Method	Pattern matching, heuristics, and static rules	Abstract interpretation (mathematical logic)
Speed	Fast	Slower (deep analysis)
Confidence Level	May have false positives	Sound – no false negatives (guaranteed proofs)
Understands Logic?	Limited	Yes – traces all execution paths logically
Test Cases Required	No	No
Example Bugs Found	Style violations, dead code, uninitialized variables	Division by zero, null pointer dereference, overflows

Target Use Case	Quick code quality check	Safety-critical systems (aerospace, automotive, etc.)
Output Format	List of issues with severity levels	Color-coded proof results:  Green,  Red,  Gray
Standards Supported	MISRA, CERT, AUTOSAR	Also supports those, with additional safety proofs