

Compile-Time vs. Runtime Trade-offs in Systems Programming



Daniel Borgs Shiwei Cui Taylan Yildirim
Technical University of Applied Sciences Würzburg-Schweinfurt

Take-Home Message

The Safety-Performance Dilemma

Programs must ensure operations happen in valid order. A file must be opened before reading. Traditional solutions force a choice:

- **Safe code:** Check validity at runtime → slower execution
- **Fast code:** Skip checks → risk of crashes and security holes

Our Question: Can we get safety *without* paying the runtime cost?

Key Finding: Yes. Rust's type system catches invalid operations during compilation. The resulting program runs as fast as code with no safety checks—while preventing errors that would crash unsafe code.

Problem Statement

Why This Matters

Memory safety bugs dominate security vulnerabilities:

- 70% of Microsoft's security issues stem from memory safety problems
- The Heartbleed bug (2014) exposed millions of servers due to one missing bounds check
- The US government now recommends memory-safe languages

The Core Tension

Systems code (operating systems, databases, embedded devices) requires maximum performance. Safety checks consume CPU cycles. Developers face an uncomfortable choice: **safe or fast?**

State Management

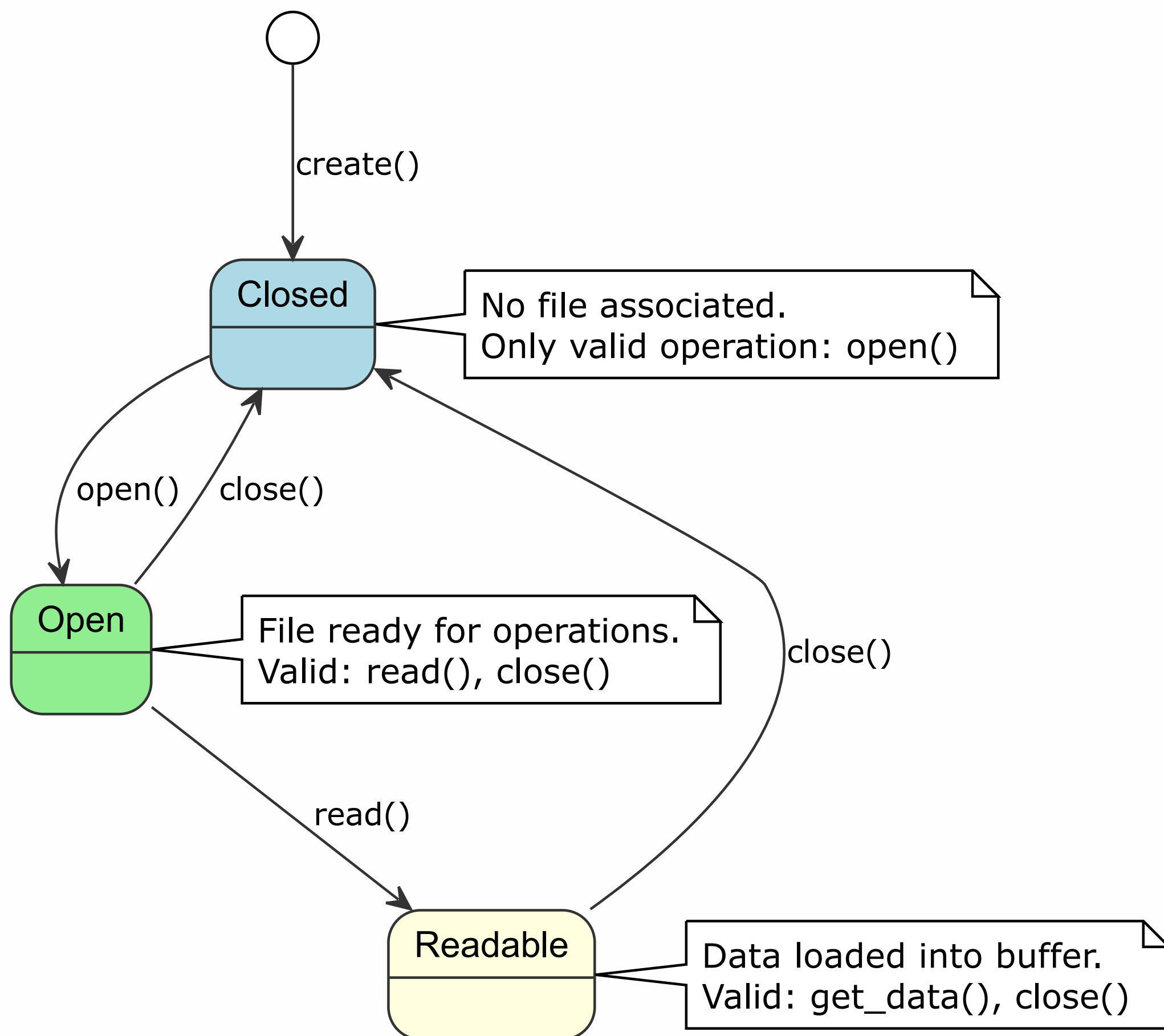
Many bugs arise from *state violations*: using an object incorrectly for its current state. Reading from a closed file handle causes undefined behavior—the program might crash, corrupt data, or create a security hole.

Background: What Is a State Machine?

Definition

A **state machine** defines: (1) states an object can be in, (2) valid transitions between states, and (3) operations permitted in each state.

File Handle State Machine



The Verification Problem

How do we ensure programs follow these rules?

Approach	When Checked	Cost
----------	--------------	------

The Typestate Pattern Explained

Core Idea

Instead of one `FileHandle` type with a state field, create separate types: `FileHandle<Closed>`, `FileHandle<Open>`, `FileHandle<Readable>`.

How It Prevents Errors

The `read()` method is defined *only* for `FileHandle<Open>`. Attempting to call `read()` on a `FileHandle<Closed>` produces a **compile-time error**—the method does not exist for that type.

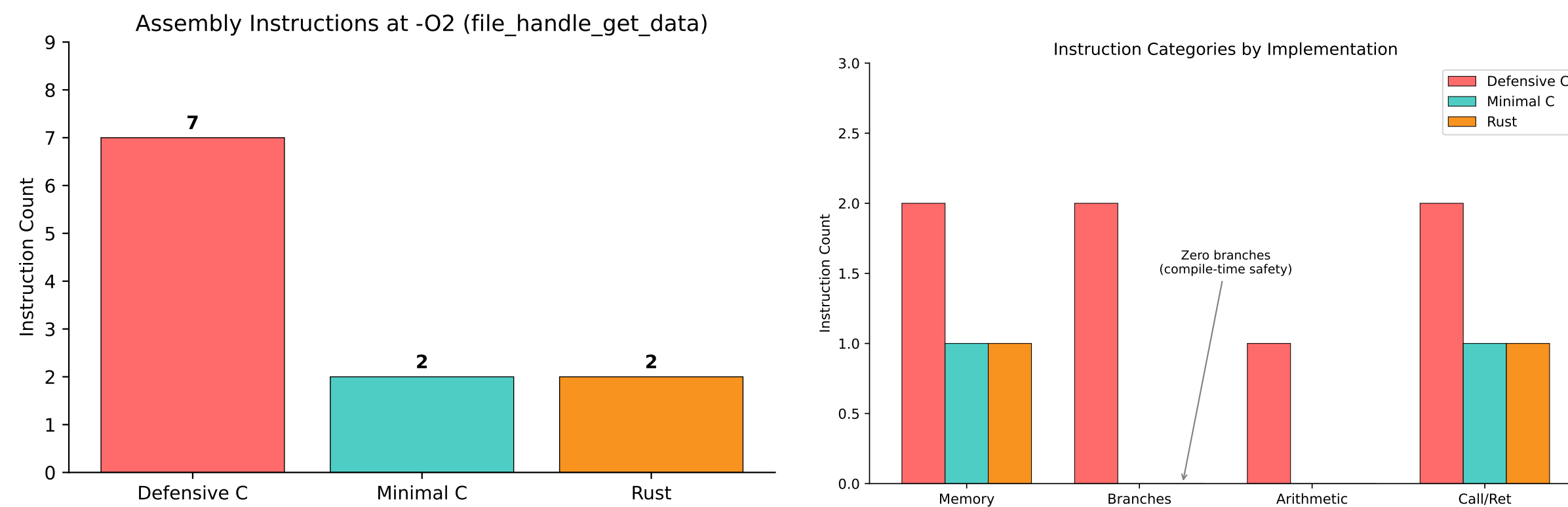
Why Zero Cost?

The state marker (`<Open>`, `<Closed>`) exists only for the compiler. It occupies **zero bytes** in memory and generates **zero instructions**. After compilation, the type information is erased—only the operations remain.

Results: Assembly Comparison

Methodology

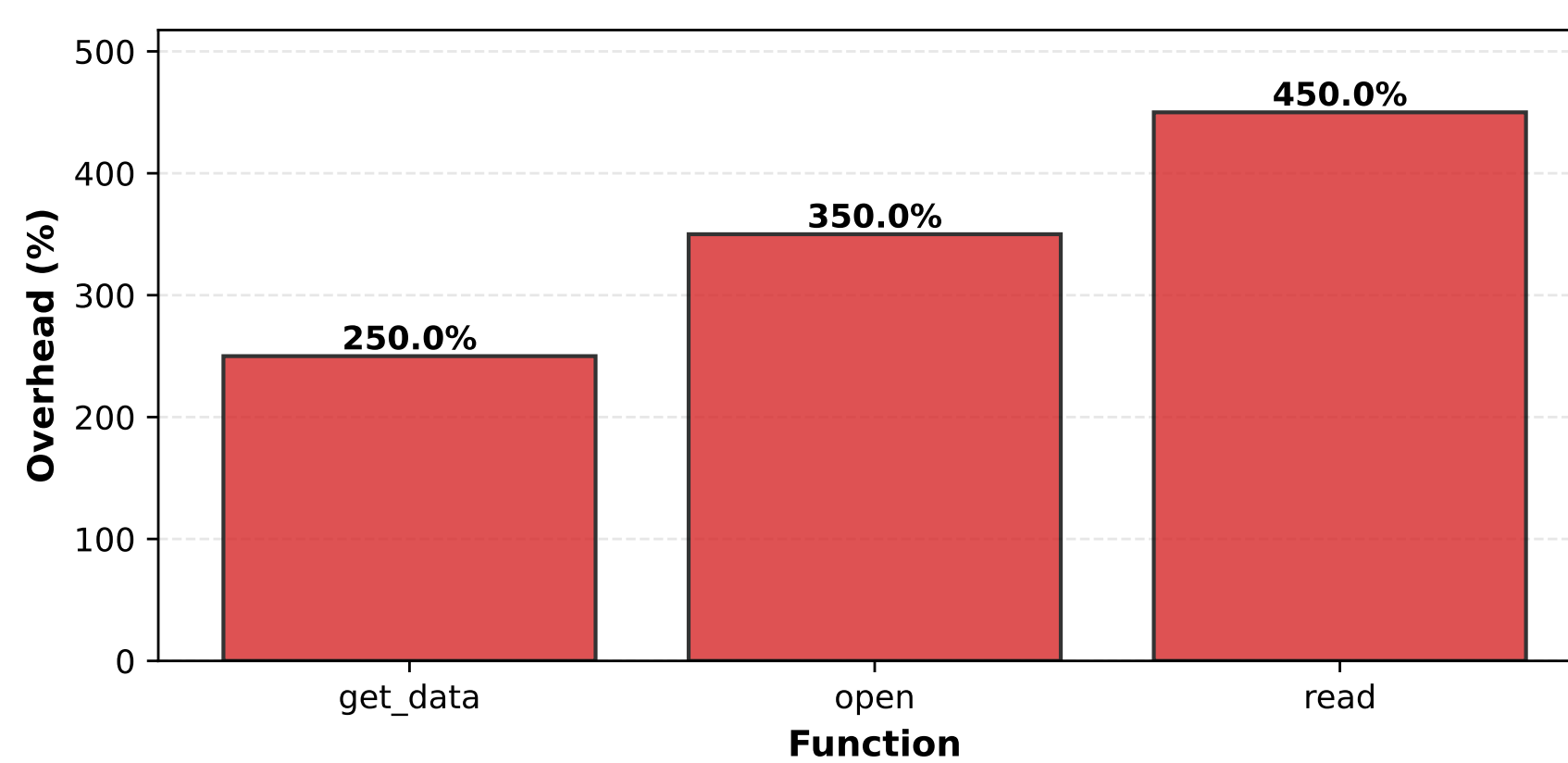
Compiled all three versions at `-O2` optimization, analyzed generated ARM64 assembly, counted total instructions and conditional branches.



Key Observations

- **Rust matches Minimal C exactly:** 2 instructions, 0 branches
- **Defensive C has 3.5× overhead:** Extra instructions for state validation
- **Critical:** Rust has **zero conditional branches**—the compiler *proved* they were unnecessary

Runtime Overhead: Defensive C vs Rust Typestate (Higher = More Overhead)



Trade-off Thinking in Computer Science

A Unifying Principle

The compile-time vs. runtime trade-off appears throughout CS. Doing work *earlier* often saves work *later*.

Trade-off Spectrum: When Does the Work Happen?

