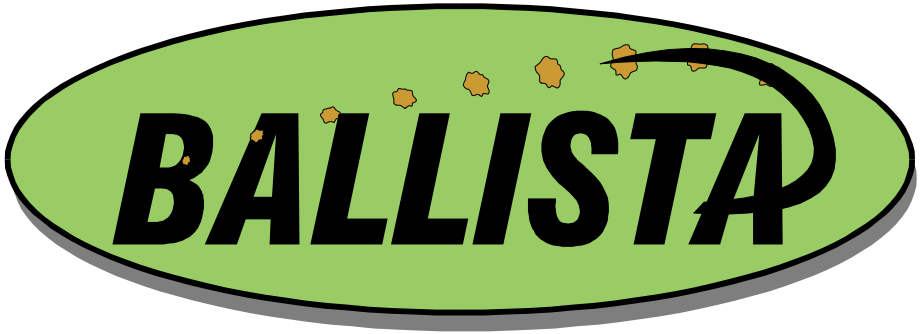
Software Robustness Testing

A Ballista Retrospective

Phil Koopman

koopman@cmu.edu

http://ballista.org



With contributions from:

Dan Siewiorek, Kobey DeVale

John DeVale, Kim Fernsler, Dave Guttendorf,
Nathan Kropp, Jiantao Pan, Charles Shelton, Ying Shi

Carnegie Mellon

Institute for Complex Engineered Systems

Electrical & Computer ENGINEERING

Overview

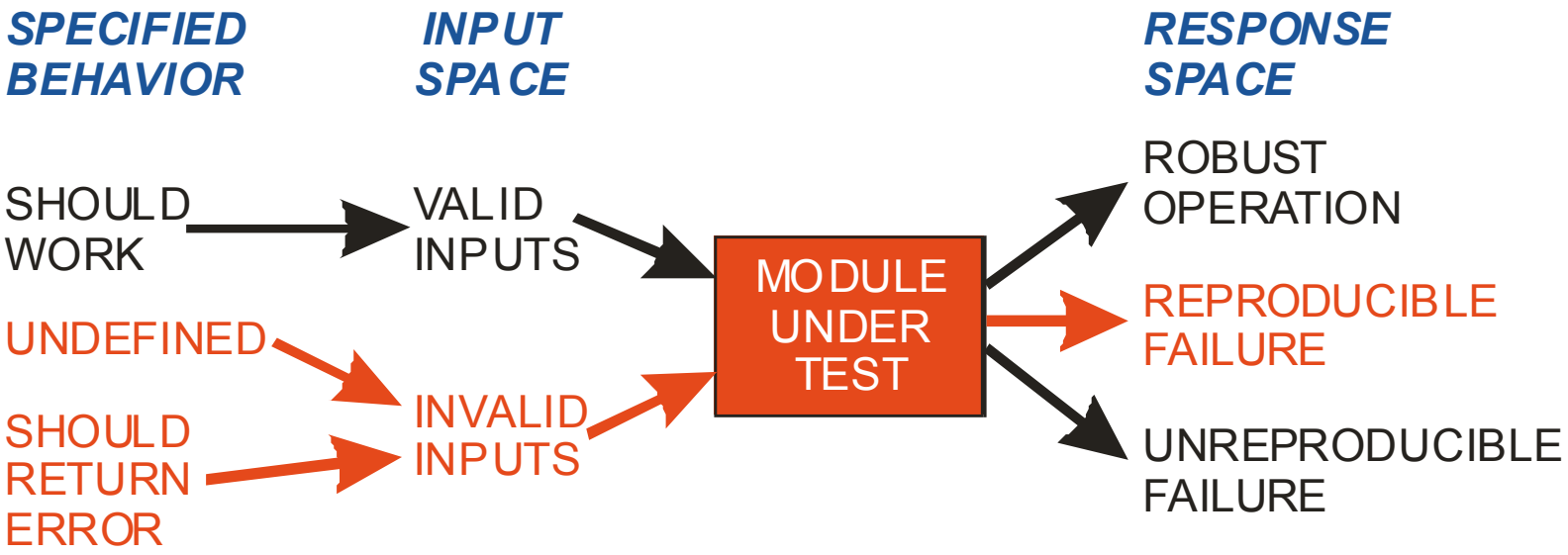
- **Introduction**
  - APIs aren't robust (and people act as if they don't want them to be robust!)

- **Top 4 Reasons people give for ignoring robustness improvement**
  - "My API is already robust, especially for easy problems" (it's probably *not*)
  - "Robustness is impractical" (it *is* practical)
  - "Robust code will be too slow" (it *need not be*)
  - "We already know how to do it, thank you very much" (*perhaps* they *don't*)

- **Conclusions**
  - The big future problem for "near-stationary" robustness isn't technology -- it is awareness & training

Carnegie Mellon

Ballista Software Testing Overview

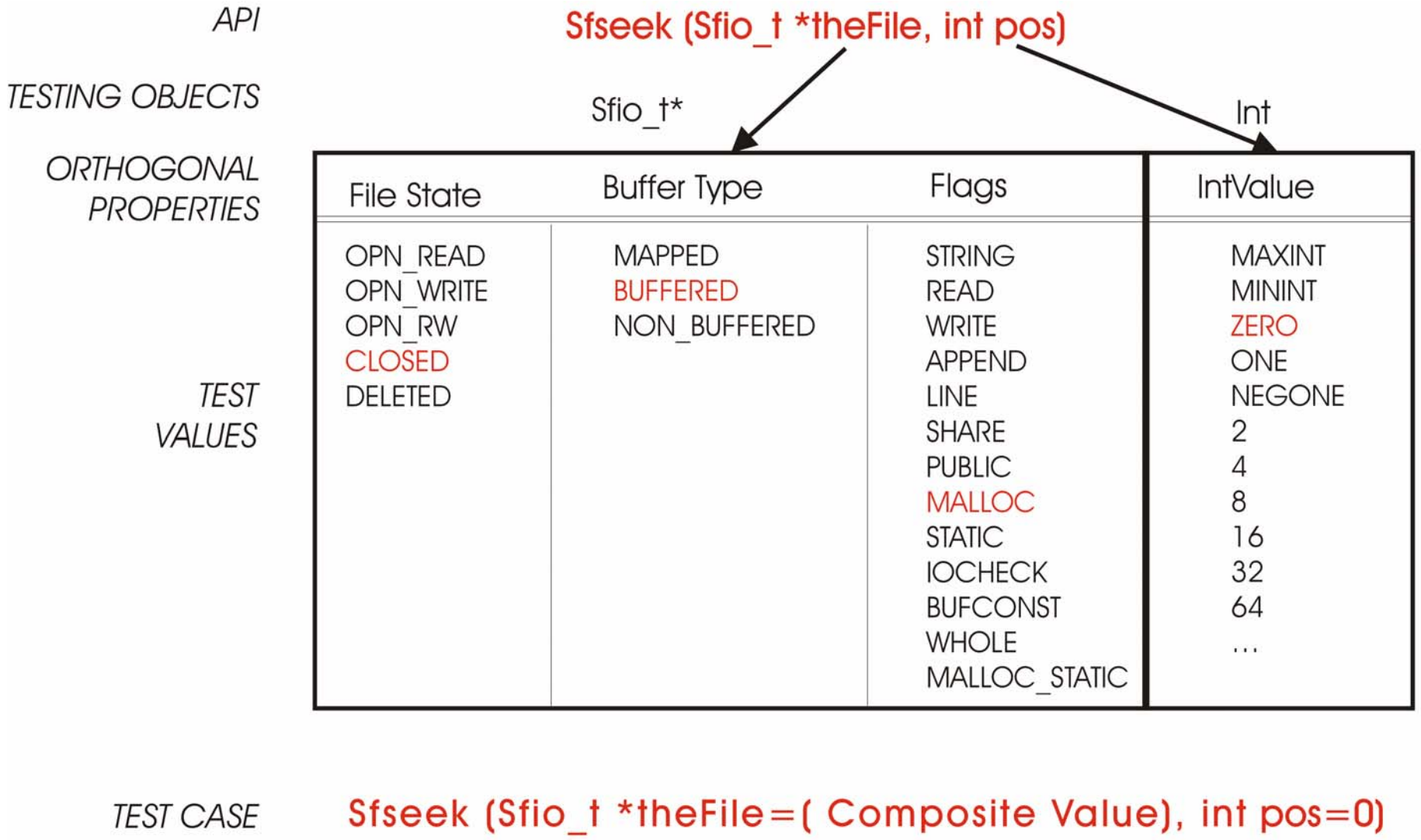


- **Abstracts testing to the API/Data type level**
  - Most test cases are exceptional
  - Test cases based on best-practice SW testing methodology

Carnegie Mellon

Ballista: Test Generation (fine grain testing)

- **Tests developed per data type/subtype; scalable via composition**

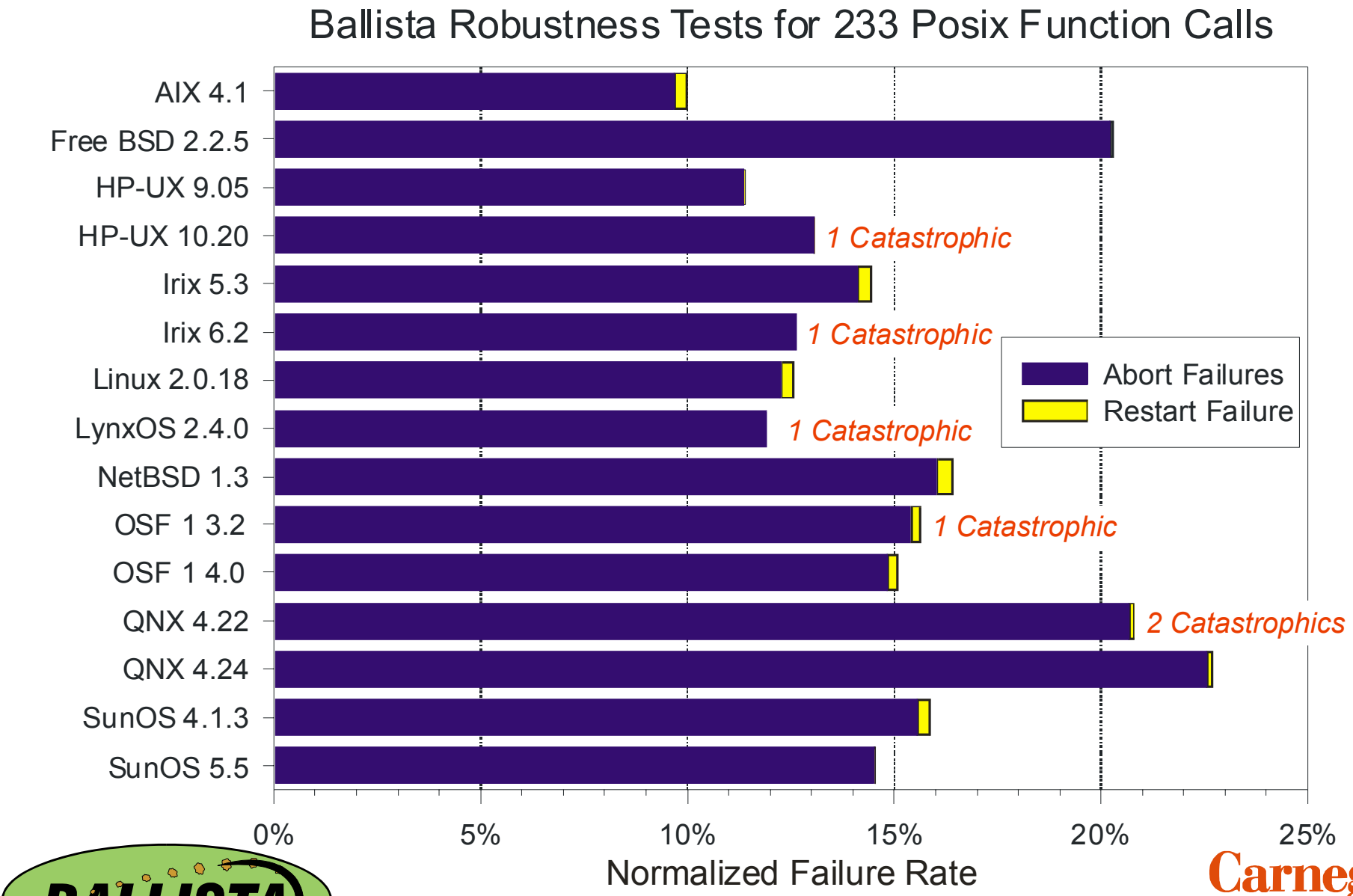


Initial Results: Most APIs Weren't Robust

- **Unix & Windows systems had poor robustness scores:**
  - 24% to 48% of intentionally exceptional Unix tests yielded non-robust results
  - Found simple "system killer" programs in Unix, Win 95/98/ME, and WinCE
- **Even critical systems were far from perfectly robust**
  - Safety critical operating systems
  - DoD HLA (where their stated goal was 0% robustness failures!)
- **Developer reactions varied, but were often extreme**
  - Organizations emphasizing field reliability often wanted 100% robustness
  - Organizations emphasizing development often said "core dumps are the Right Thing"
  - Some people didn't care
  - Some people sent hate mail

Carnegie Mellon

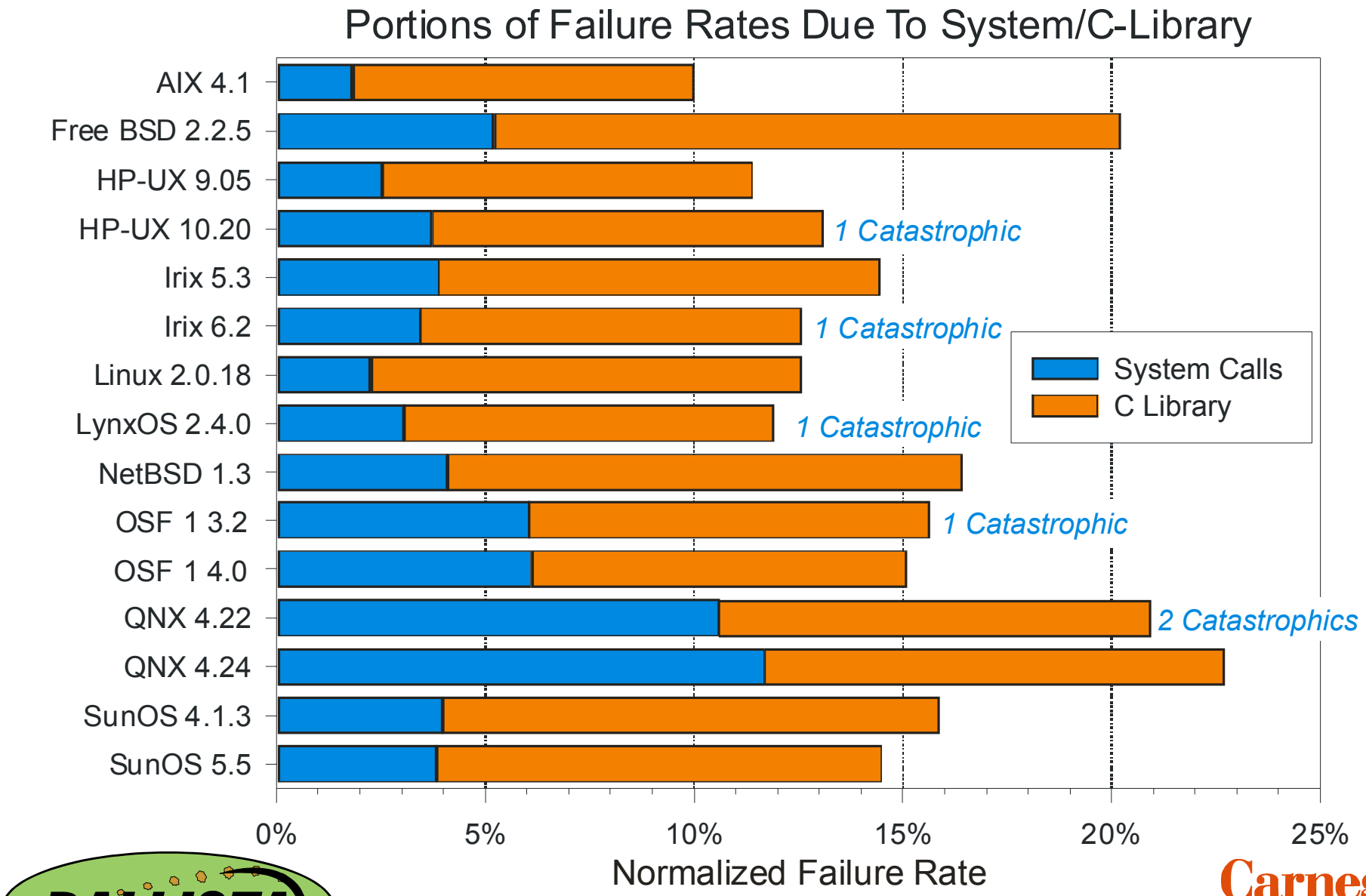
Comparing Fifteen POSIX Operating Systems



BALLISTA

Carnegie Mellon

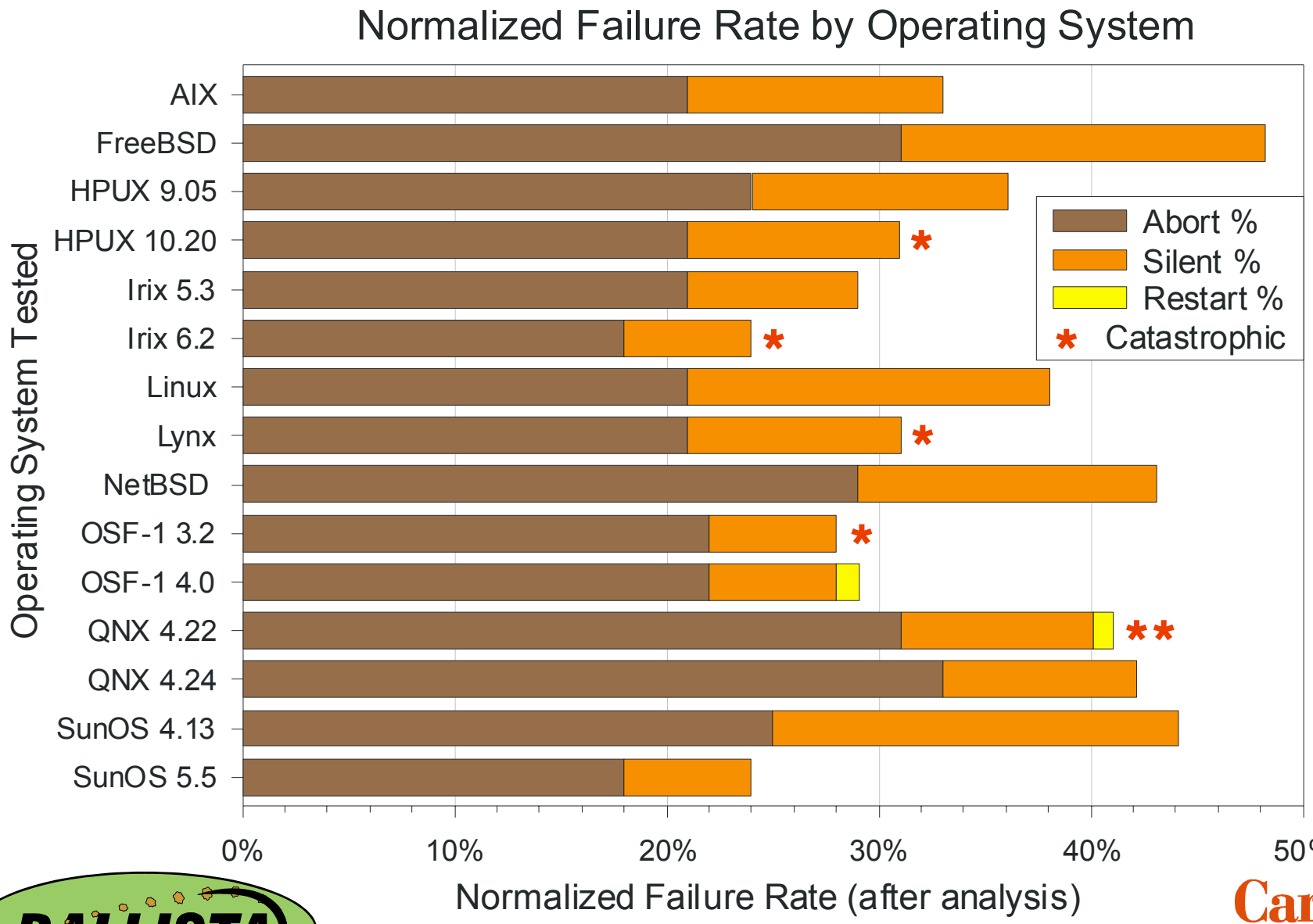
Robustness: C Library *vs.* System Calls



BALLISTA

Carnegie Mellon

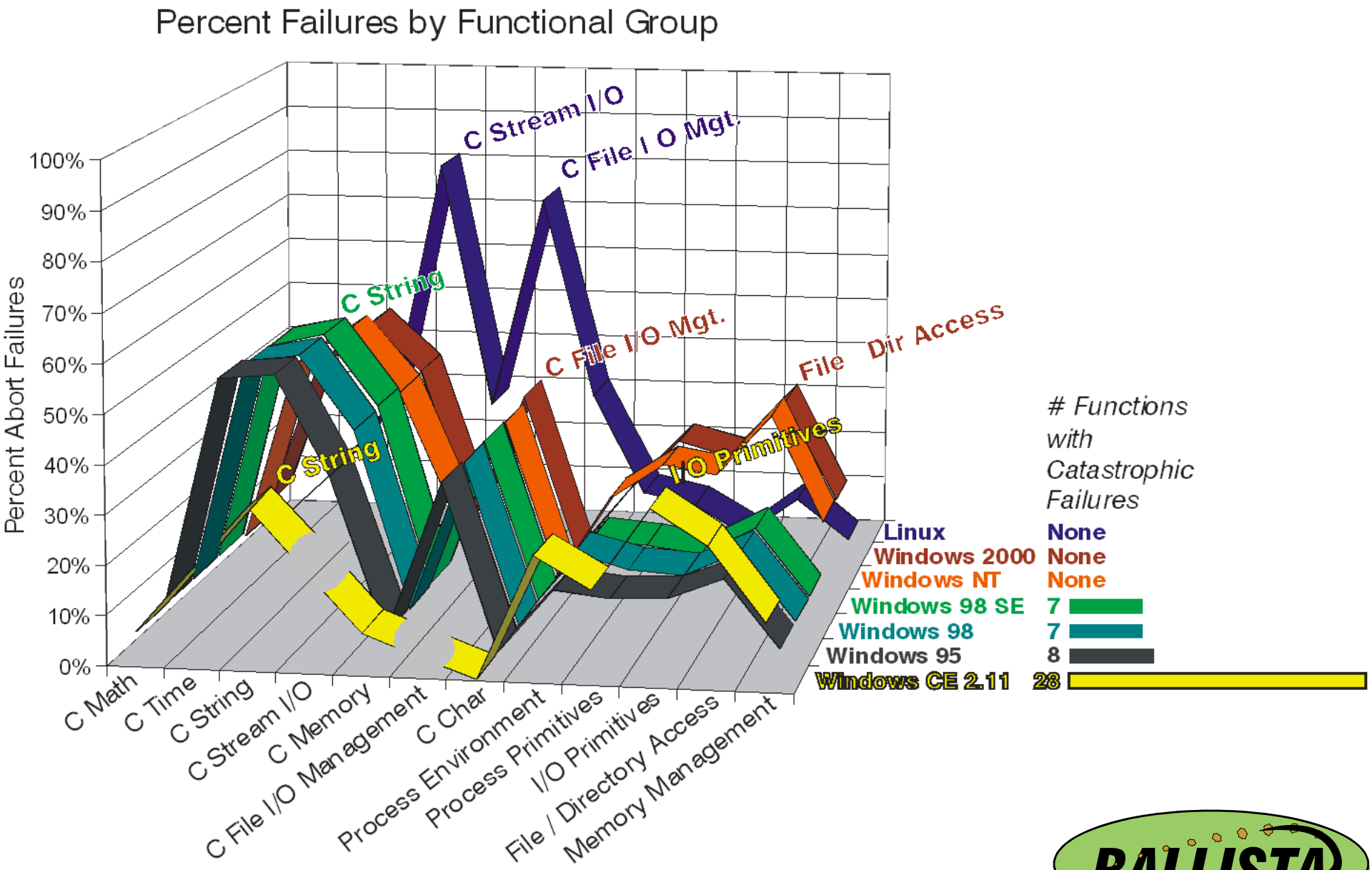
Estimated N-Version Comparison Results



BALLISTA

Carnegie Mellon

Failure Rates by Function Group



BALLISTA

Even Those Who Cared Didn't Get It Right

- **OS Vendors didn't accomplish their stated objectives** (e.g.,):
  - IBM/AIX wanted few Aborts, but had 21% Aborts on POSIX tests
  - FreeBSD said they would always Abort on exception (that's the Right Thing) but had more Silent (unreported) exceptions than AIX!
  - Vendors who said their results would improve dramatically on the next release were usually wrong

- **Safe Fast I/O (SFIO) library**
  - Ballista found that it wasn't as safe as the authors thought
    - Missed: valid file checks; modes vs. permissions; buffer size/accessibility

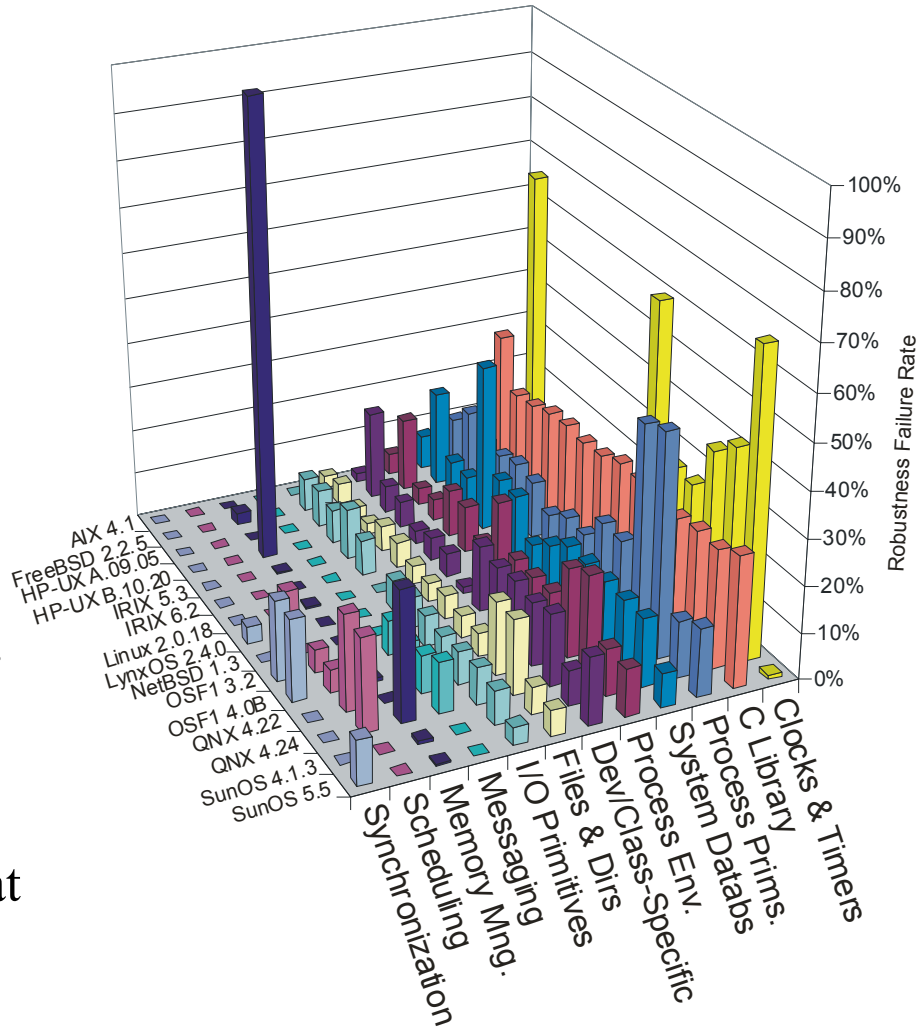
- **Do people understand what is going on?**
  - We found four widely held misconceptions that prevented improvement in code robustness

Carnegie Mellon

#1: "Ballista will never find anything (important)"

1. **"Robustness doesn't matter"**
   - HP-UX gained a system-killer in the upgrade from Version 9 to 10
     - In newly re-written memory management functions…
       … which had a 100% failure rate under Ballista testing
   - So, robustness seems to matter!

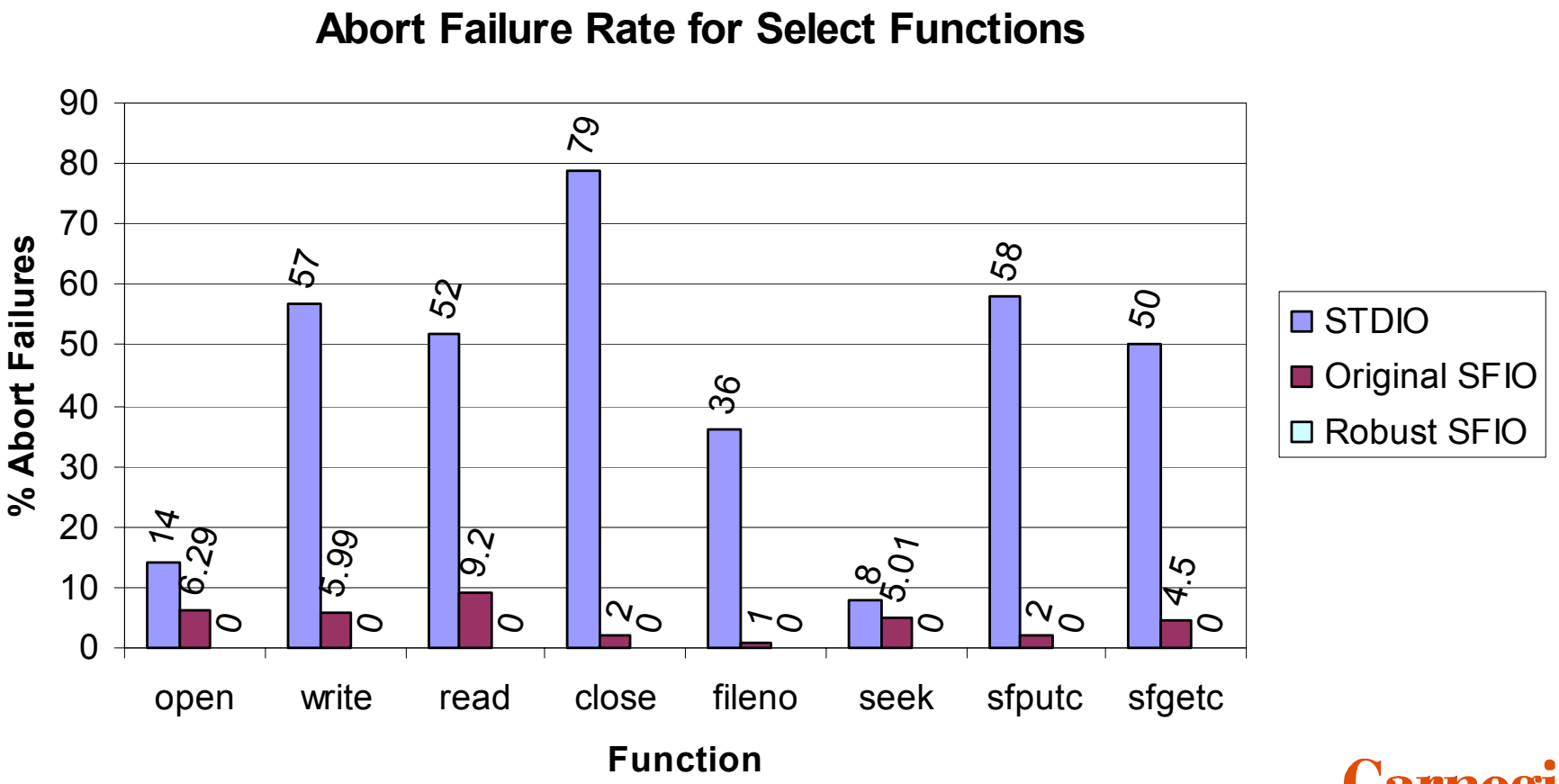
2. **"The problems you're looking for are too trivial -- we don't make those kinds of mistakes"**
   - HLA had a handful of functions that were very non-robust
   - SFIO even missed some "easy" checks
   - See Unix data to the right…



Carnegie Mellon

#2: “100% robustness is impractical”

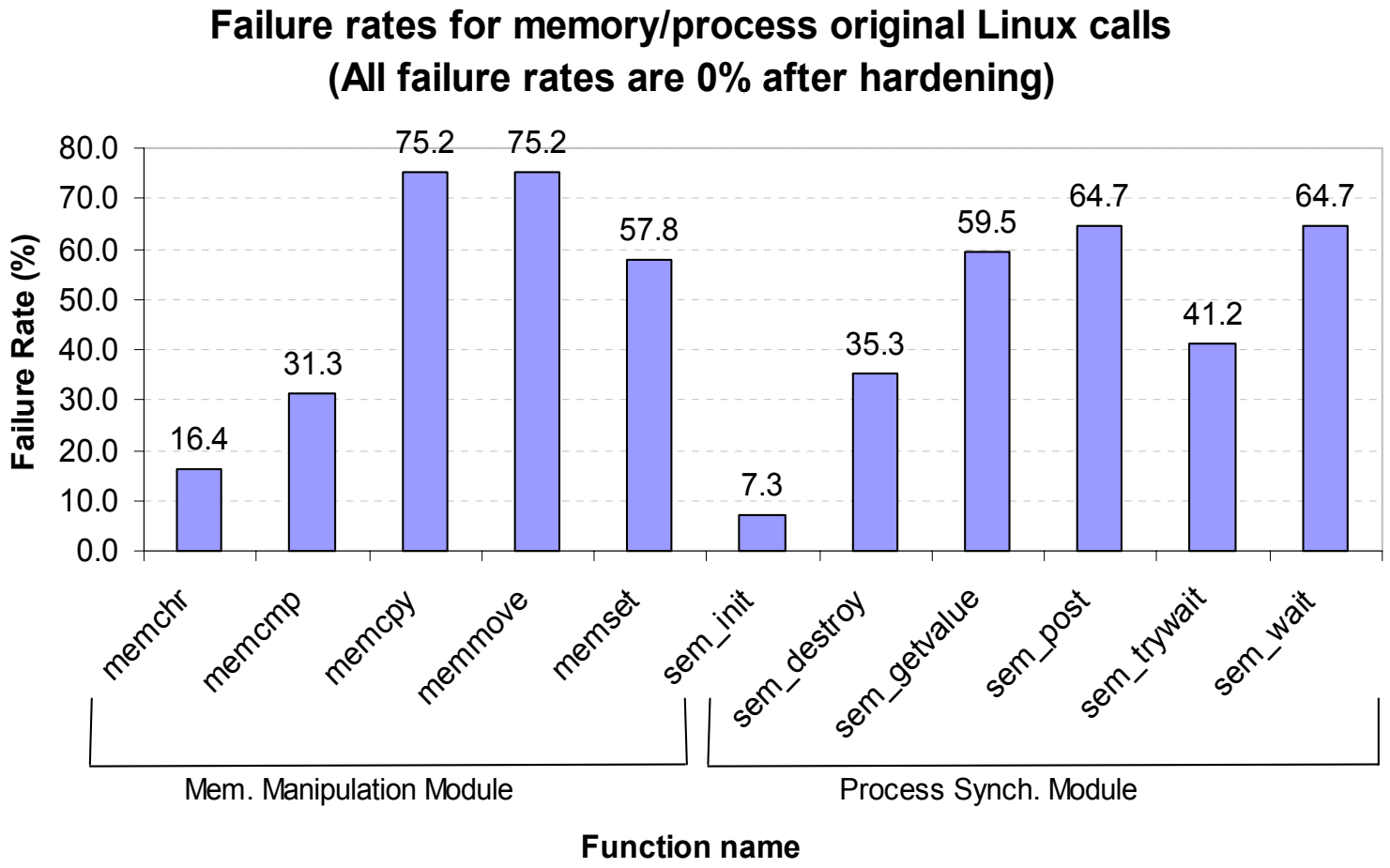
- **The use of a metric – in our case Ballista – allowed us to remove all detectable robustness failures from SFIO and other API subsets**
  - (Our initial SFIO results weren’t entirely zero; but now they are)



Carnegie Mellon

Can Even Be Done With "Ordinary" API

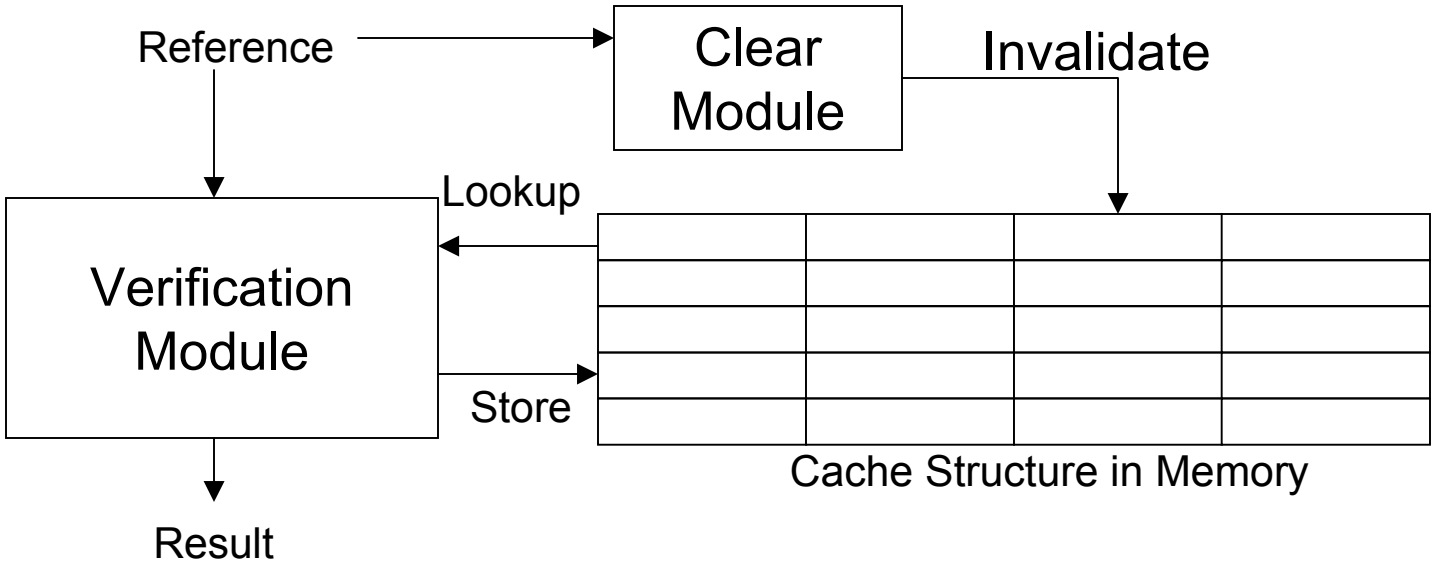
- **Memory & semaphore robustness improved for Linux**
  - Robustness hardening yielded 0% failure rate on standard POSIX calls below



Carnegie Mellon

#3: "It will be too slow"

- **Solved via caching validity checks**
  - Completely software-implemented cache for checking validity

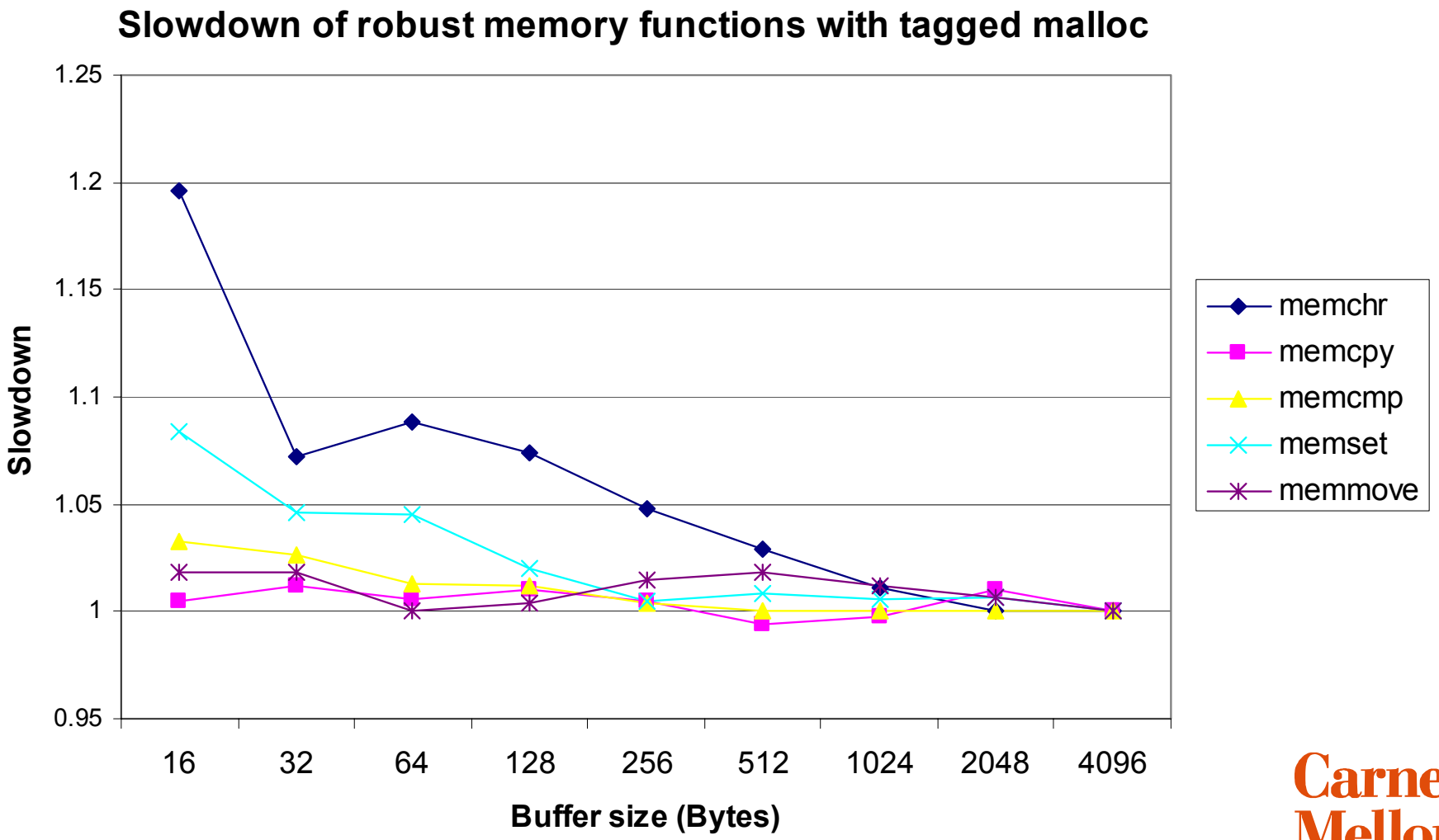


  - Check validity once, remember result
    - Invalidate validity check when necessary

Carnegie Mellon

Caching Speeds Up Validity Tests

- **Worst-case of tight loops doing nothing but "mem" calls is still fast**
  - L2 Cache misses would dilute effects of checking overhead further



Carnegie Mellon

Future MicroArchitectures Will Help

- **Exception & validity check branches are highly predictable**
  - Compiler can structure code to assume validity/no exceptions
  - Compiler can give hints to branch predictor
  - Branch predictor will quickly figure out the "valid" path even with no hints
  - Predicated execution can predicate on "unexceptional" case

- **Exception checks can execute in parallel with critical path**
  - Superscalar units seem able to execute checks & functions concurrently
  - Out of order execution lets checks wait for idle cycles

- **The future brings more speculation; more concurrency**
  - Exception checking is an easy target for these techniques
  - Robustness is cheap and getting cheaper (if done with a view to architecture)

Carnegie Mellon

#4: "We Did That On Purpose"

- **Variant: "Nobody could reasonably do better"**
  - Despite the experiences with POSIX, HLA & SFIO, this one persisted
  - So, we tried an experiment in self-evaluating robustness

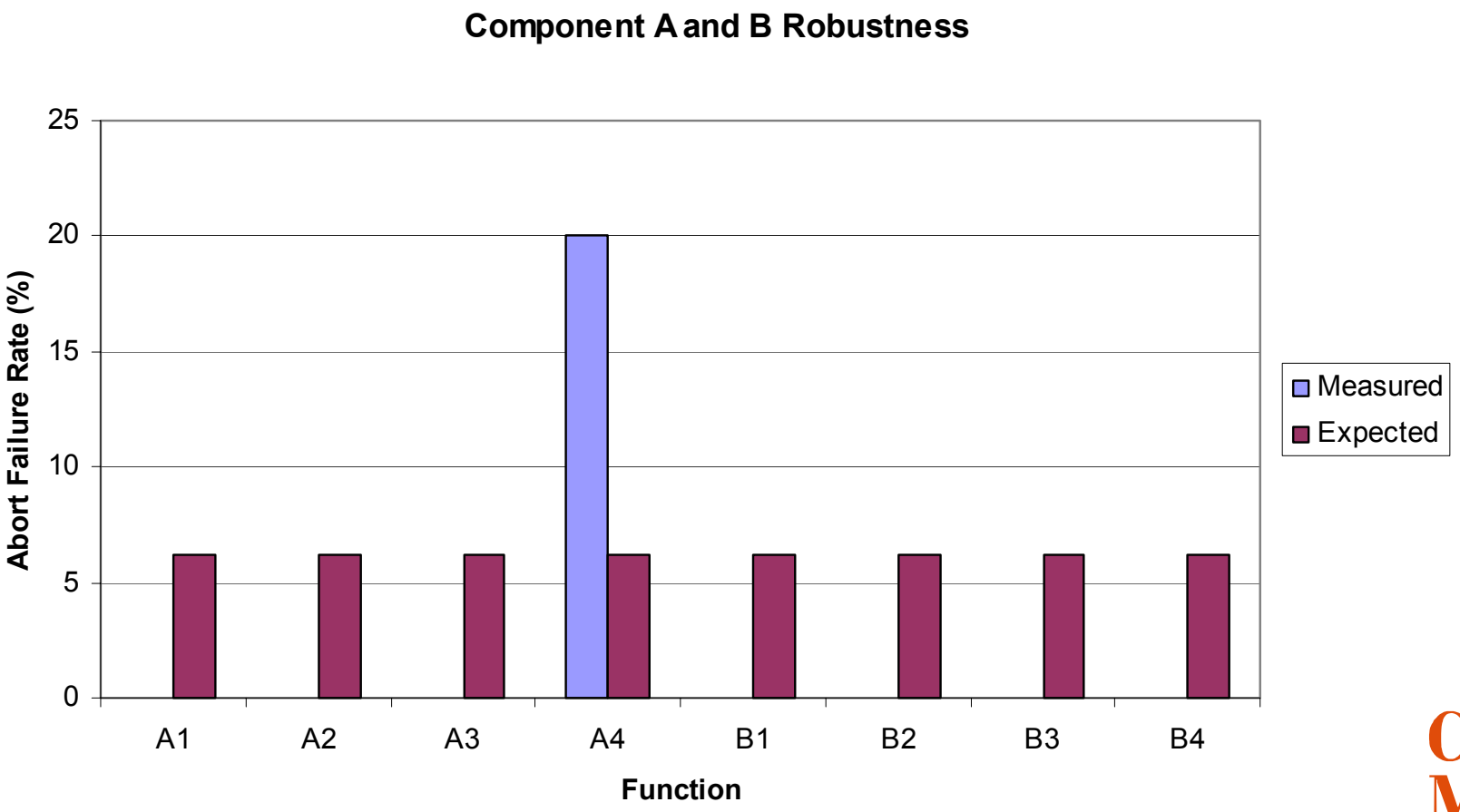
- **Three experienced commercial development teams**
  - Components written in Java
  - Each team self-rated the robustness of their component per Maxion's "CHILDREN" mnemonic-based technique
  - We then Ballista tested their (pre-report) components for robustness

  - Metric: did the teams accurately predict where their robustness vulnerabilities would be?
    - They didn't have to be perfectly robust
    - They all felt they would understand the robustness tradeoffs they'd made

Carnegie Mellon

Self Report Results: Teams 1 and 2

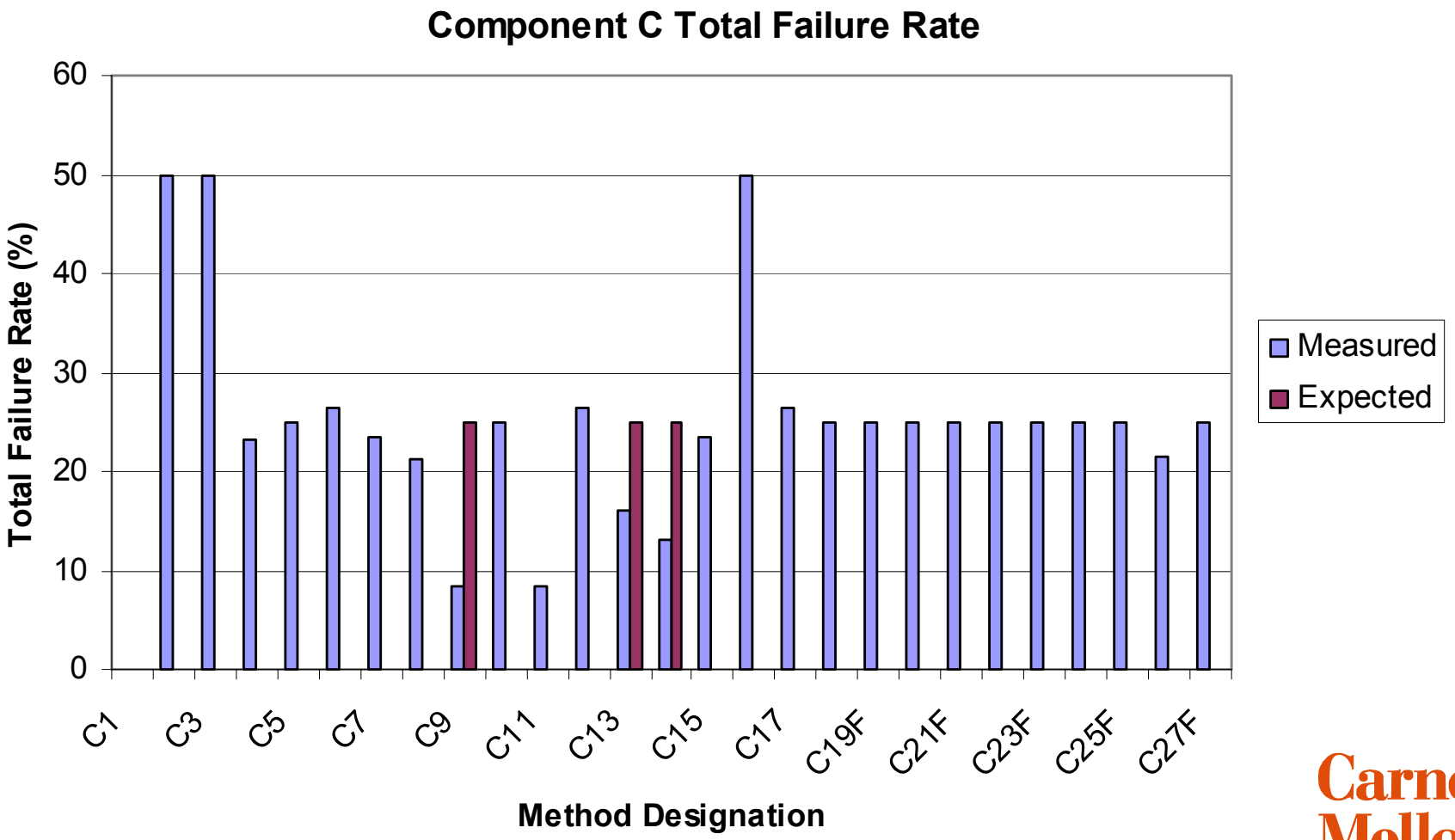
- **They were close in their prediction**
  - Didn't account for some language safety features (divide by zero)
  - Forgot about, or assumed language would protect them against NULL in A4



Carnegie Mellon

Self Report Data: Team 3

- **Did not predict several failure modes**
  - Probably could benefit from additional training/tools



Carnegie Mellon

Conclusions: Ballista Project In Perspective

- **General testing & wrapping approach for Ballista**
  - Simple tests are effective(!)
    - Scalable for both testing and hardening
  - Robustness tests & wrappers can be abstracted to the data type level
    - Single validation fragment per type – *i.e.* checkSem(), checkFP()…
- **Wrappers are fast (under 5% penalty) and usually 100% effective**
  - Successful check results can be cached to exploit locality
    - Typical case is an index lookup, test and jump for checking cache hit
    - Typical case can execute nearly "for free" in modern hardware
  - After this point, it is time to worry about resource leaks, device drivers, *etc.*
- **But, technical solution alone is not sufficient**
  - Case study of self-report data
    - Some developers unable to predict code response to exceptions
  - Training/tools needed to bridge gap
    - Even seasoned developers need a QA tool to keep them honest
    - Stand-alone Ballista tests for Unix under GPL; Windows XP soon

Carnegie Mellon

Future Research Challenges In The Large

- **Quantifying "software aging" effects**
  - Simple, methodical tests for resource leaks
    - Single-threaded, multi-threaded, distributed all have different issues
    - One problem is multi-thread contention for non-reentrant resources
      - » e.g., exception handling data structures without semaphore protection
  - Measurement & warning systems for need for SW rejuvenation
    - Much previous work in predictive models
    - Can we create an on-line monitor to advise it is time to reboot?

- **Understanding robustness tradeoffs from developer point of view**
  - Tools to provide predictable tradeoff of effort vs. robustness
    - QA techniques to ensure that desired goal is reached
    - Ability to specify robustness level clearly, even if "perfection" is not desired
  - Continued research in enabling ordinary developers to write robust code
  - Need to address different needs for development *vs.* deployment
    - Developers want heavy-weight notification of unexpected exceptions
    - In the field, may want a more benign reaction to exceptions

Carnegie Mellon

BALLISTA