Software Fault Tolerance

EECE 513: Design of Fault-tolerant Digital Systems

Learning Objectives

- Define Software Fault-tolerance and enumerate its challenges
- List three diversity-based techniques and evaluate their respective pros and cons
- Use robust data structures for structural integrity
- Use critical memory for semantic integrity

What is Software Fault Tolerance?

- Three alternative definitions

1. Management of faults originating from defects in design or implementation of software components
2. Management of hardware failures in software
3. Management of network failures

We will follow the classical definition (1) due to Avizienis in 1977

Motivation: Software Fault Tolerance

- Usual method of software reliability is fault avoidance using good software engineering methodologies
- Large and complex systems ⇒ fault avoidance not successful
  - Rule of thumb fault density in software is 10-50 per 1,000 lines of code for good software and 1-5 after intensive testing using automated tools
- Redundancy in software needed to detect, isolate, and recover from software failures
- Hardware fault tolerance easier to assess
- Software is difficult to prove correct

HARDWARE FAULTS	SOFTWARE FAULTS
1. Faults time-dependent	Faults time-invariant
2. Duplicate hardware detects	Duplicate software not effective
3. Random failure is main cause	Complexity is main cause

Challenges

- Improvements in software development methodologies reduce the incidence of faults, yielding fault avoidance
- Need for test and verification
- Formal verification techniques, such as proof of correctness, can be applied to only small programs
- Potential exists of faulty translation of user requirements
- Conventional testing is hit-or-miss. "Program testing can show the presence of bugs but never show their absence," - Dijkstra, 1972.
- There is a lack of good fault models for software defects

Features of software faults

- Mature software exhibits nearly constant failure rate
  - Bathtub curve for modeling entire lifetime from release to retirement
- Number of failures is correlated with
  - Execution time
  - Code density
  - Software timing,
  - Synchronization points

Approaches to Software Fault Tolerance

- **ROBUSTNESS**: The extent to which software continues to operate despite introduction of invalid inputs.

  Example: 1. Check input data

  =>ask for new input

  =>use default value and raise flag

  2. Self checking software

- **FAULT CONTAINMENT**: Faults in one module should not affect other modules.

  Example: Reasonable checks

  Watchdog timers

  Overflow/divide-by-zero detection

  Assertion checking

- **FAULT TOLERANCE**: Provides uninterrupted operation in presence of program fault through multiple implementations of a given function

Learning Objectives

- Define Software Fault-tolerance and enumerate its challenges
- List three diversity-based techniques and evaluate their respective pros and cons
- Use robust data structures for structural integrity
- Use critical memory for semantic integrity

Diversity

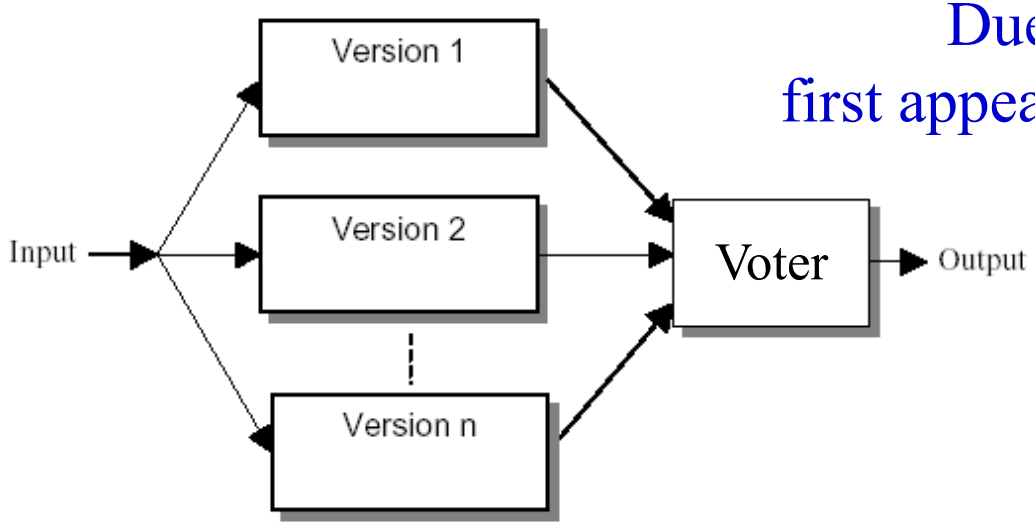
- Diversity as a technique for fault-tolerance goes back to the British Astronmer, Lord Maskelyne [Anh-2009]
  - Used two computers (human) to calculate lunar tables, when moon is at peak and its lowest point and compare the values
- Charles Babbage used Diversity in analytical engine

"When the formula to be computed is very complicated, it may be algebraically arranged for computation in two or more totally distinct ways, and two or more sets of cards may be made. If the same constants are now employed with each set, and if under these circumstances the results agree, we may be quite secure of the accuracy of them all."

Multi-Version Software Fault Tolerance

- Use of multiple versions (or “variants”) of a piece of software
- Different versions may execute in parallel or in sequence
- Rationale is that multiple versions will fail differently, i.e., for different inputs
- Versions are developed from common specifications
- Three main approaches
  - N-version Programming
  - Recovery Blocks
  - N Self-Checking Programming

N-Version Programming



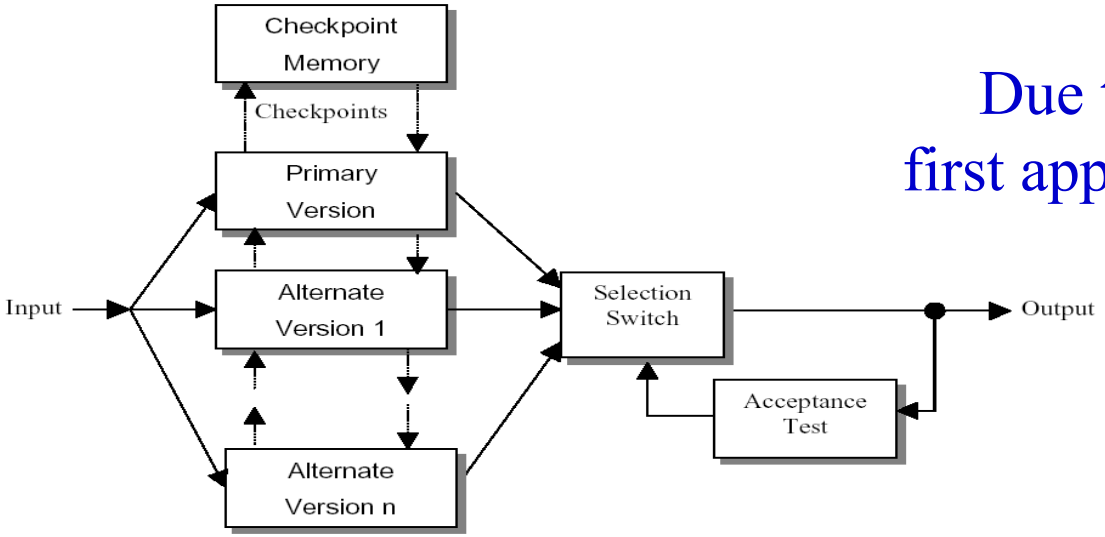
Due to Al Avizienis,
first appeared in CompSAC 1977

- All versions designed to satisfy same basic requirement
- Decision of output comparison based on voting
- Different teams build different versions to avoid correlated failures

Pros and Cons of NVP

- NVP relies on independence among the versions
  - But not always true in practice [Knight and Leveson'83]
- Why does this happen ?
  - People make same mistakes, e.g., incorrect treatment of boundary conditions
  - Some parts of a problem are more difficult than others - similarity in programmer's view of "difficult" regions
  - Specifications may themselves be incorrect/incomplete
- Note: This does not mean NVP is useless. Rather, it does not always mean that NVP will detect S/W faults. Its reliability is upper-bounded by independence.

Recovery Blocks



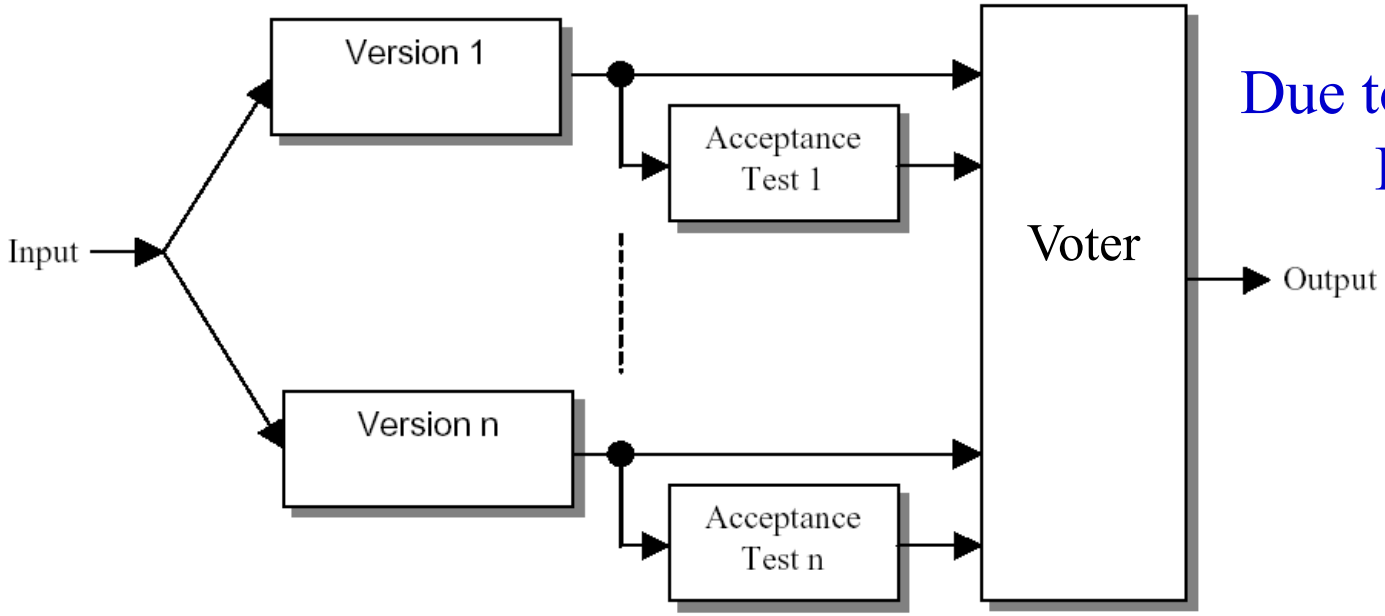
Due to Brian Randell, first appeared in ToSE 1975

- Checkpoint and restart approach
  - Try a version, if error detected through acceptance test, try a different version
  - Ordering of the different versions according to reliability
- Checkpoints needed to provide valid operational state for subsequent versions (hence, discard all updates made by a version)
- Acceptance test needs to be faster and simpler than actual code

Pros and Cons of RB

- Advantages
  - No performance or area overheads in the fault-free case, except the state saving overhead.
  - Allows gradual evolution of software components. Old versions can be replaced with new ones, and used as secondary.
  - Nice hierarchical design (structured approach)
- Disadvantages
  - Reliability depends on the coverage of the acceptance test. Acceptance test should be independent of the main version, but faster (e.g., range checks)
  - State saving mechanisms need to be employed.
  - Requires transaction-like semantics. Cannot always undo side-effects.

N Self-Checking Programming



Due to J. C. Laprie, FTCS 87

- Multiple software versions with structural variations of RB and NVP
- Use of separate acceptance tests for each version

Pros and Cons of NSCP

- Advantages
  - Combines advantages of NVP and RBs
  - Ensure that some errors are caught before the voting stage
  - Provides error containment
  - Almost no disruption in service due to faults
- Cons
  - Incurs more overhead than NVP and 'N' times the overhead of RB
  - Does not protect against errors in specifications
  - Extra effort to derive acceptance tests and write the N-versions

Similarity to H/W Fault-tolerance

- RB is equivalent to the stand-by sparing (of passive dynamic redundancy)
- NVP is equivalent to N-modular redundancy (static redundancy)
- NSCP is equivalent to active dynamic redundancy. A self-checking component results either from:
  - Association of an acceptance test to a version
  - Association of two variants with a comparison algorithm

Reliability Analysis of Multi-Version Approaches

Three postulates of software development [Sha-2000, IEEE Software]

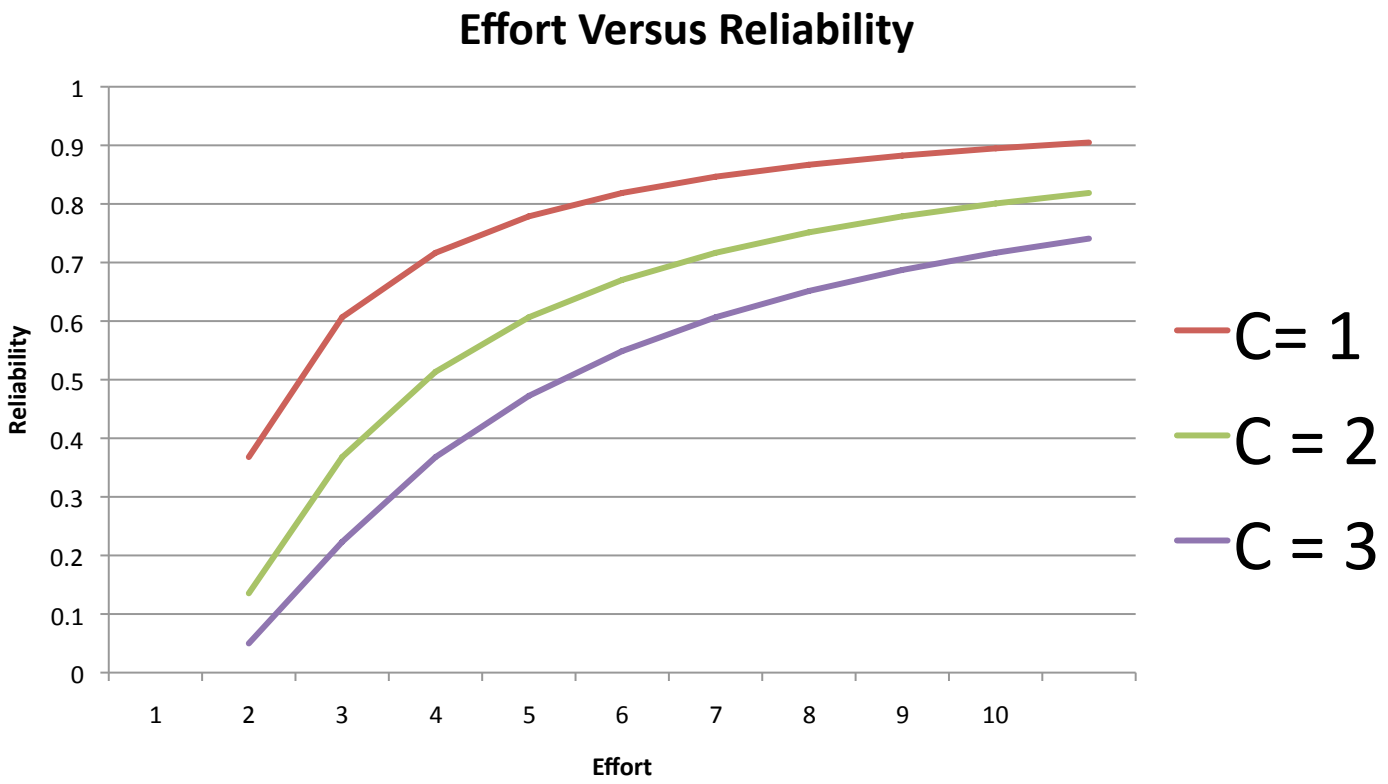
P1: Complexity Breeds Bugs: Everything else being equal, the more complex the software project is, the harder it is to make it reliable.

P2: All Bugs are Not Equal: You fix a bunch of obvious bugs quickly, but finding and fixing the last few bugs is much harder, if you can ever hunt them down.

P3: All Budgets are Finite: There is only a finite amount of effort (budget) that we can spend on any project. That is, if we go for n version diversity, we must divide the available effort n-way.

- We attempt to analyze the reliability of the three systems using methods from combinatorial modeling. We assume the following:
  - $R(t) = e^{-\lambda t}$
  - Failure rate $\lambda \propto 1/\text{Effort (E)}$
  - Failure rate $\lambda \propto$ Complexity (C)
  - Let $\lambda = kC / E$

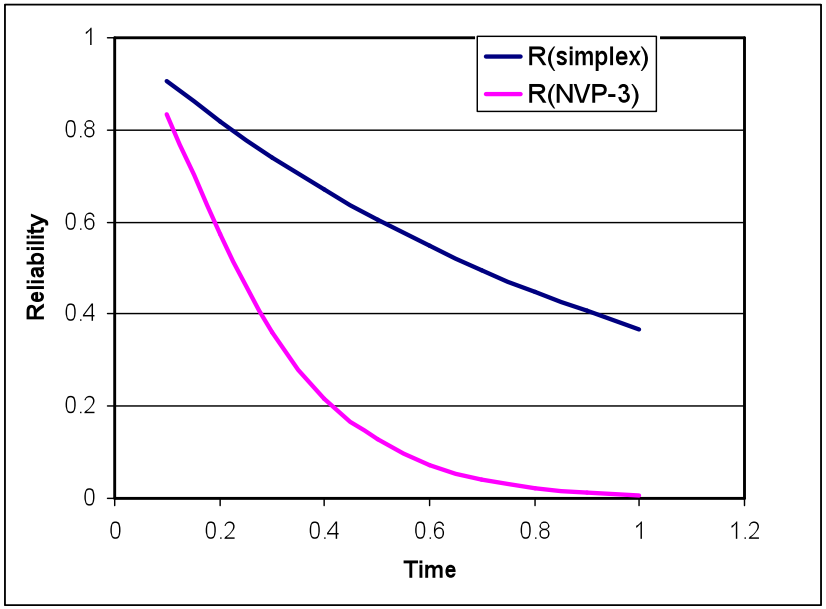
Reliability Vs Effort Vs Complexity



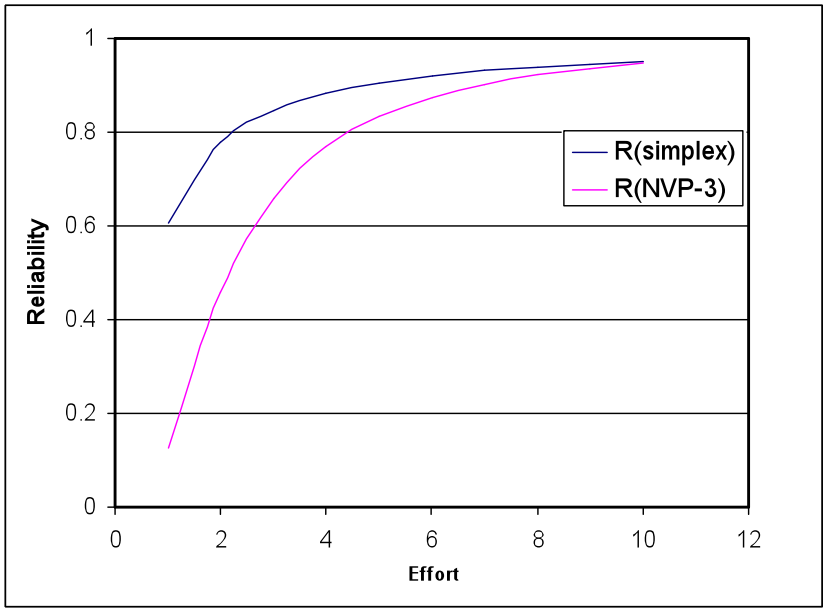
$$R_{simplex} = \exp(-kCt/E)$$

Reliability of NVP vs. single version

For Effort = 1



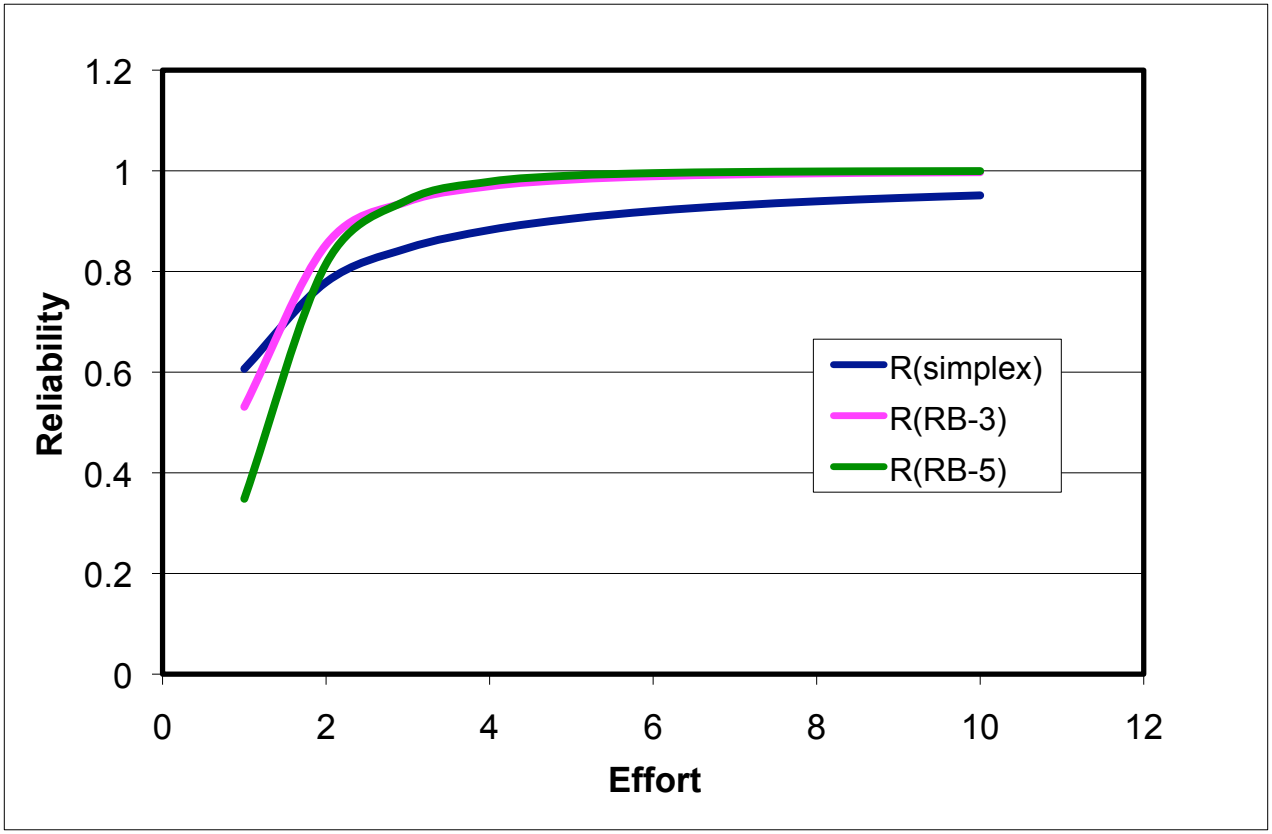
At time = 1



$$R_{simplex} = \exp(-kCt/E)$$

$$R_{NVP} = 3\exp(-6kCt/E) - 2\exp(-9kCt/E)$$

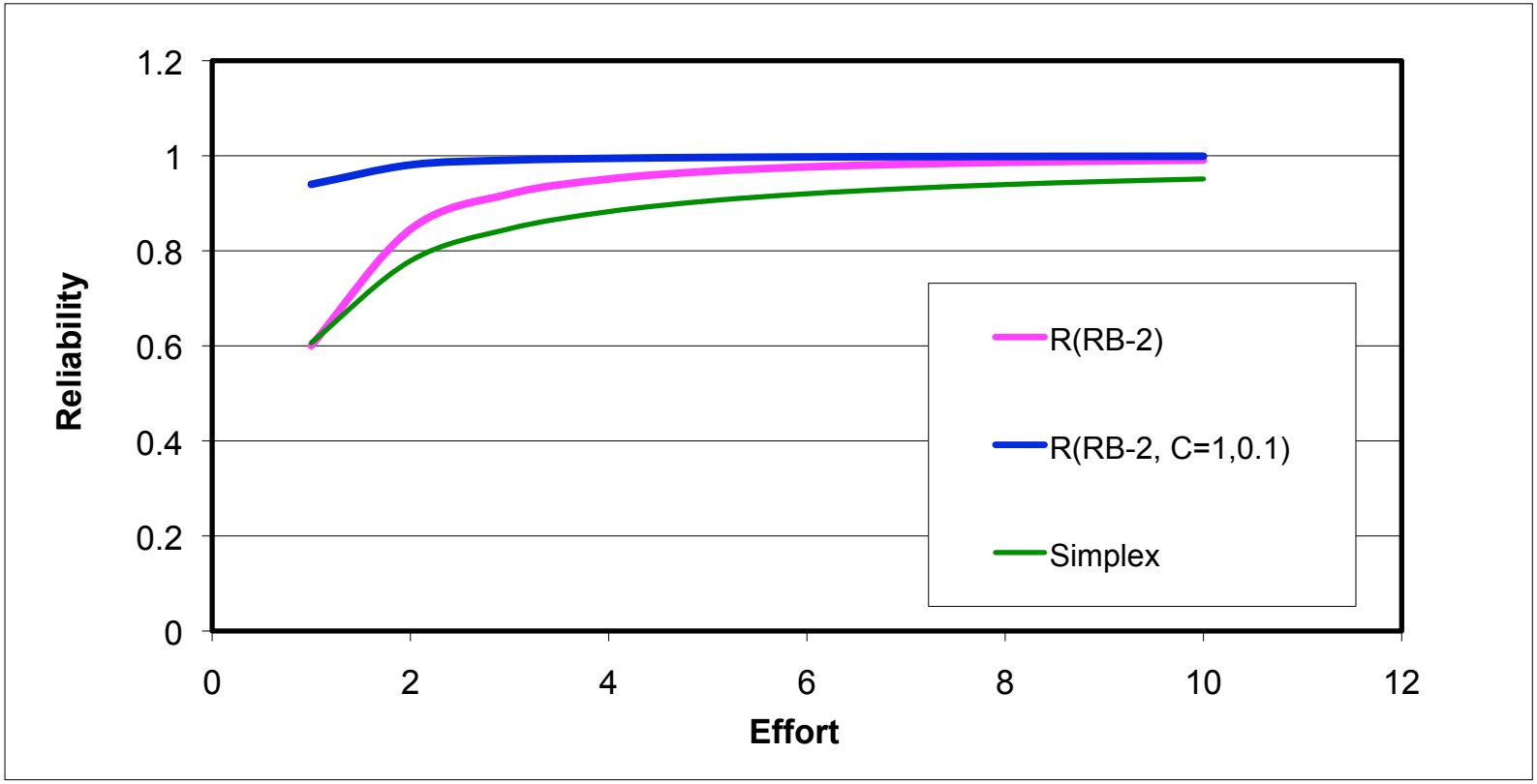
Reliability of RB vs. Simplex



$R_{simplex} = \exp(-kCt/E)$

$R_{RB} = 1-(1-\exp(-3kCt/E))^3$

Effort Vs. Complexity



Reducing complexity of alternatives drastically improves the availability. So using simpler alternates is good !

Diversity: Summary

- Simplicity often yields higher benefits than diversity and its associated complexity
- Given a choice between increasing effort for a single component and building > 2 diverse components with less effort, prefer the former
- Recovery blocks with simpler alternates do offer benefits over Simplex architecture

Learning Objectives

- Define Software Fault-tolerance and enumerate its challenges
- List three diversity-based techniques and evaluate their respective pros and cons
- Use robust data structures for structural integrity
- Use critical memory for semantic integrity

Robust Data Structures: Goals

- The goal is to find storage structures that are robust in the face of errors and failures
- What do we want to preserve?

Semantic integrity - the data is not corrupted

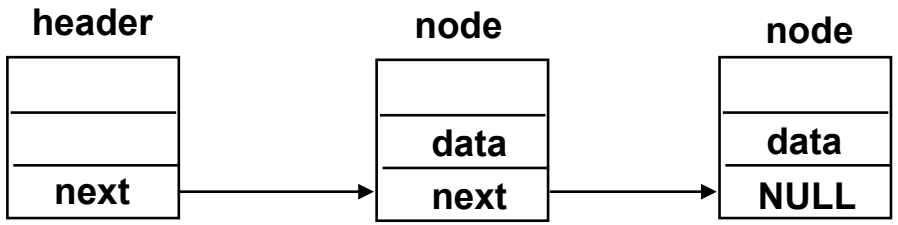
Structural integrity - the correct data representation is preserved

Robust Data Structure: Definition

- A robust data structure contains *redundant data* which allow *erroneous changes* to be detected, and corrected
  - a change is defined as an elementary (e.g., as single word) modification to the encoded form (e.g., data structure representation in memory) of a data structure instance
  - structural redundancy
    - a stored count of the numbers of nodes in a structure instance
    - identifier fields
    - additional pointers

Example: Linked Lists

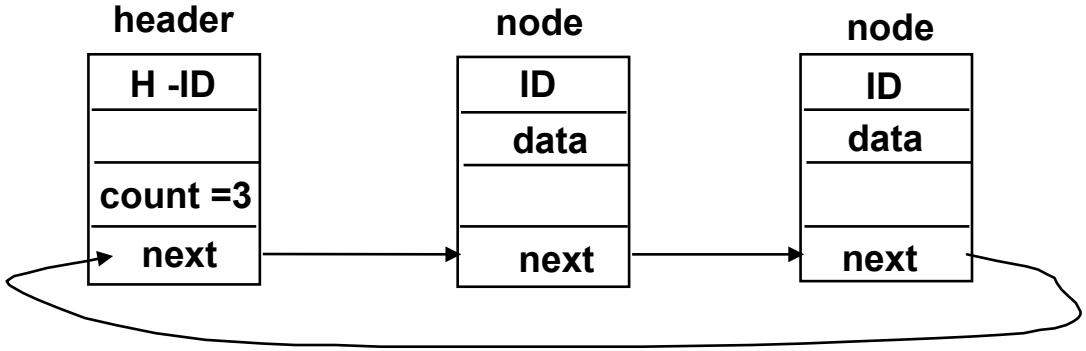
- Non-robust data structure: No redundant information to detect/recover from pointer errors



0-detectable and 0-correctable
changing one pointer to NULL can reduce any list to empty list

Example: Robust List

- Additions for improving robustness
  - an identifier field to each node
  - replace the NULL pointer in the last node by pointer to the header of the list
  - stores a count of the number of nodes



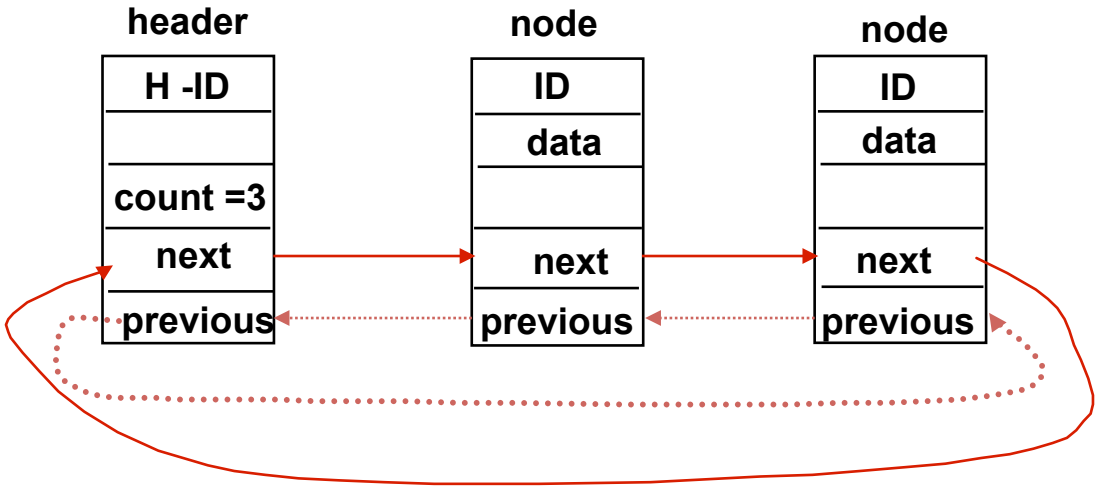
1-detectable and 0-correctable

•change to the count can be detected by comparing it against the number of nodes found by following pointers

•change to the pointer may be detected by a mismatch in count number or the new pointer points to a foreign node (which cannot have a valid identifier)

Example: Robust Double Linked List

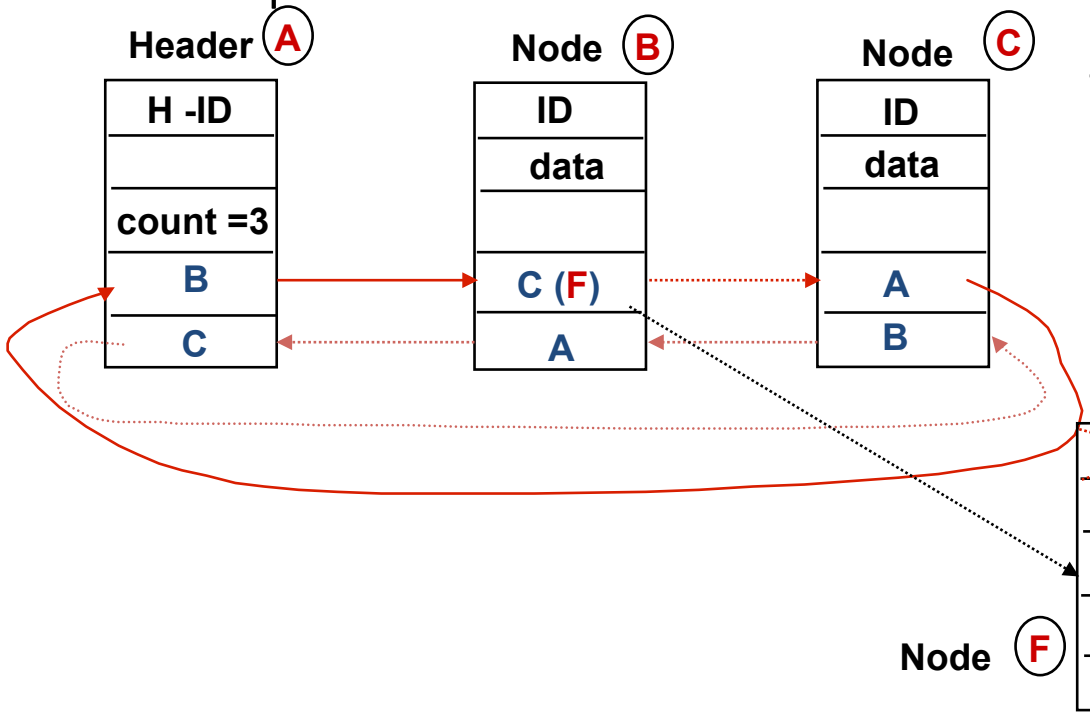
- Additions for improving robustness: Make it a double linked list



2-detectable and 1-correctable
the data structure has two independent, disjoint sets of pointers, each of which may be used to reconstruct the entire list

Error Correcting in Double-Linked List

- *Scan the list in the forward direction* until an identifier field error or forward/backward pointer mismatch is detected
- When this happens *scan the list in the reverse direction* until a similar error is detected
- Repair the data structure



The forward scan detects a mismatch in Node B and sets
$Local\_Ptr_B$ = B (local node's pointer)
$Next\_Ptr_B$ = F (pointer to the next node)
The reverse scan detects a mismatch in Node C and sets
$Local\_Ptr_C$ = C (local node's pointer)
$Back\_Ptr_C$ = B (pointer to the previous node)

Correction
($Local\_Ptr_B == Back\_Ptr_C$) ⇒
$Next\_Ptr_B := Local\_Ptr_C$
i.e., ($Next\_Ptr_B$ = C)

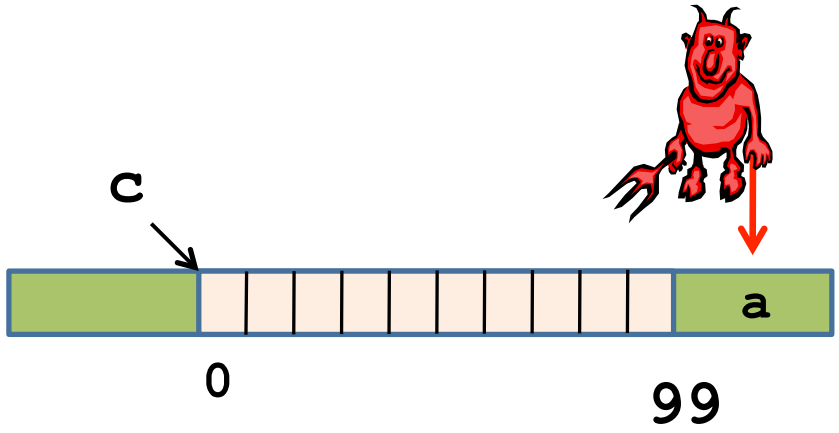
Robust Data Structures: Summary

- Advantages
  - Incurs much lower overheads than full duplication
  - Can detect both S/W and H/W errors that corrupt DS
  - Independent of programming language/compiler
- Limitations
  - Not transparent to the application
  - Best in tolerating errors which corrupt the structure of the data (not the semantics)
  - Increased complexity of the update routines may make them error prone – error propagation

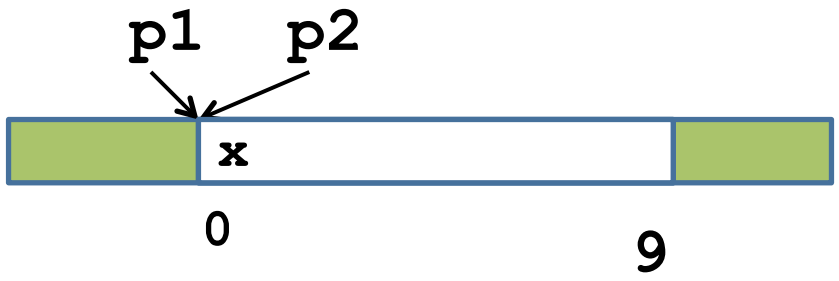
Learning Objectives

- Define Software Fault-tolerance and enumerate its challenges
- List three diversity-based techniques and evaluate their respective pros and cons
- Use robust data structures for structural integrity
- Use critical memory for semantic integrity

Background: Memory Corruption




```
c[101] = '\n';
```

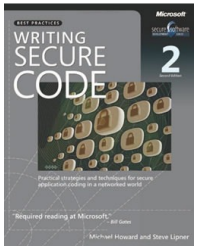


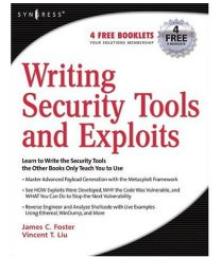

```
p1 = p2;
free(p1);
```

- **Buffer-overflows**
  - Stack and Heap buffers
  - Can corrupt both control and non-control data
- **Dangling Pointers**
  - Use after free
  - Aliased with used memory

Memory Corruption Errors : "Solutions"

- **Write code using secure programming practices**
  - Requires tremendous programmer effort
  - Loading of unsafe libraries and plugins





- **Statically check code for memory corruption errors**
  - False-positives, requires manual inspection to understand
  - Developers often reluctant to fix non-exploitable bugs

- **Dynamically check all memory writes**
  - Prohibitive overheads in practice (60 to 100%)
  - "All or nothing" technique – requires libraries' source code

Take-away Observations/Goals

- Protection from all classes of memory errors
- Must not require rewriting of code in safe languages or checking all library code
- Overhead must be configurable by the programmer – depending on application

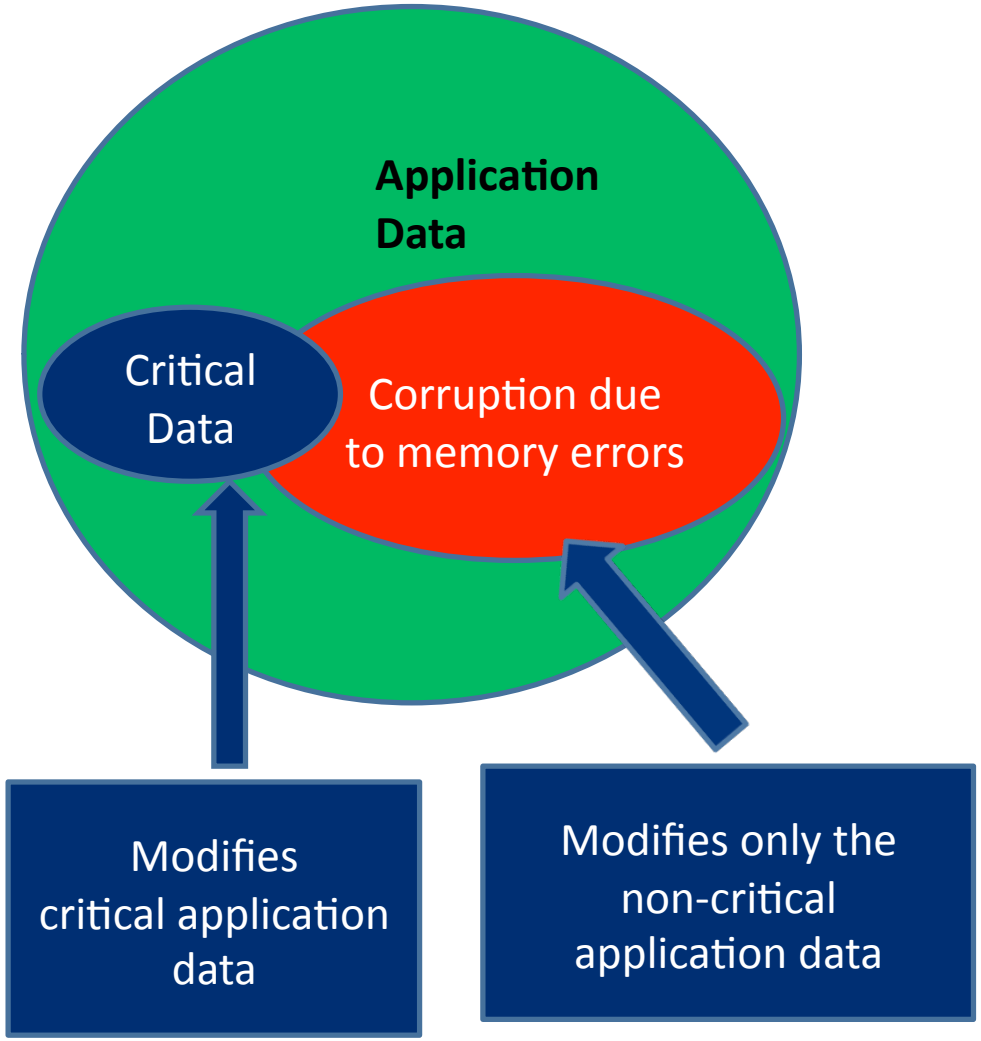
How do we satisfy all three goals ?

Approach : Critical Data Protection

- **Observation:** Some application data is much more important than other data – **Critical Data**
  - **Examples:** Bank account information, game player data, document information in word-processor
  - Identified by programmer based on appln. semantics

- **Goal: Selectively protect only the critical data**
  - Many applications are inherently tolerant of errors
  - Degraded outputs are acceptable as long as it does not corrupt the critical data or cause massive failures
  - **Provide "good enough" reliability at low cost**

Critical Data Protection: Advantages



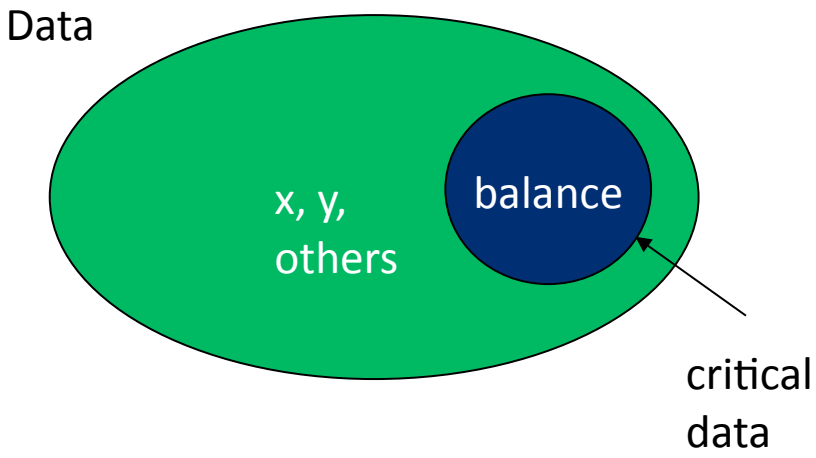
Critical data integrity should be preserved even if other data is corrupted

Apply incrementally to legacy systems, based on protection required and acceptable performance overhead

Should not need the entire application's source code – only the part that modifies the critical data

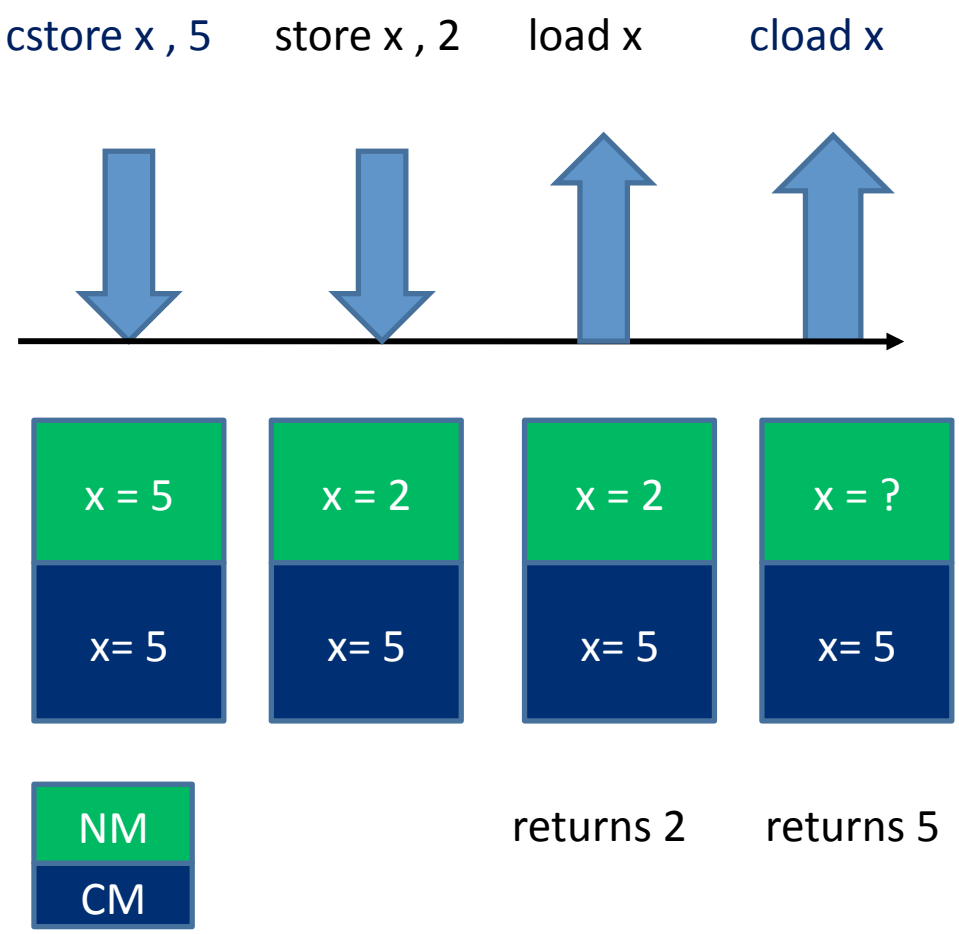
Critical Memory: Abstraction

```
critical int balance;
int  x, y;
balance = 100;
if (balance<min) {
   chargeCredit();
} else {
   x += 10;
   y += 10;
}
```



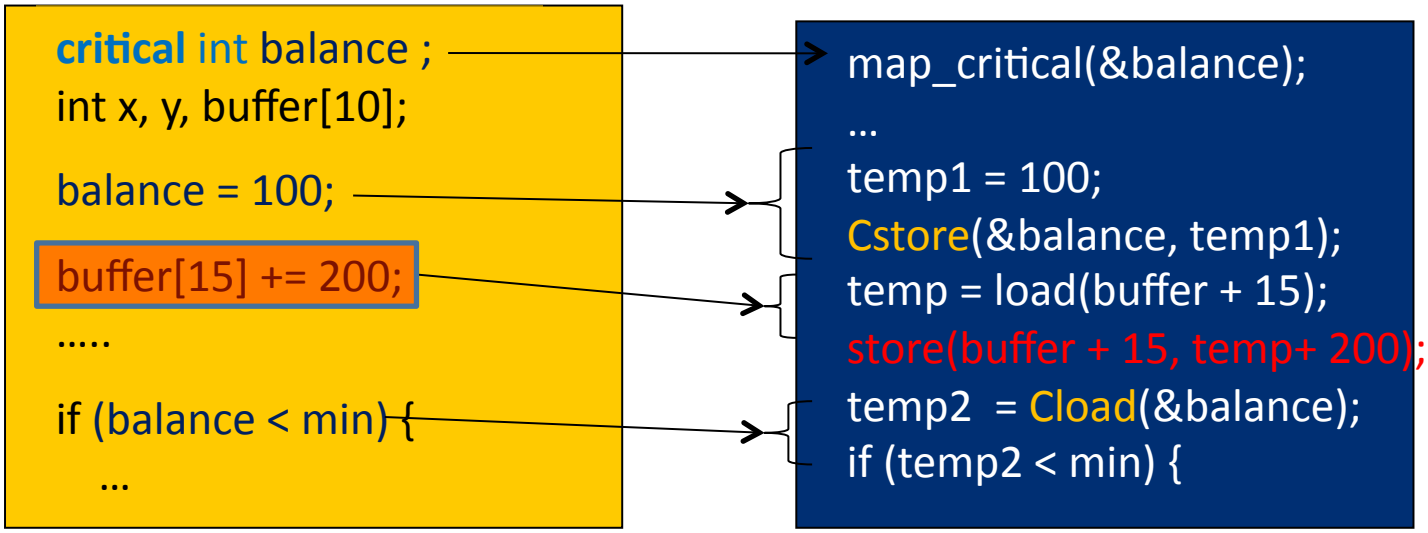
- Critical Memory: Abstract memory model
  - Protect and reason about critical data consistency
- Need to mark critical data (similar to **const**)
- Identify where CM is
  - Read from (*cload*)
  - Written to (*cstore*)

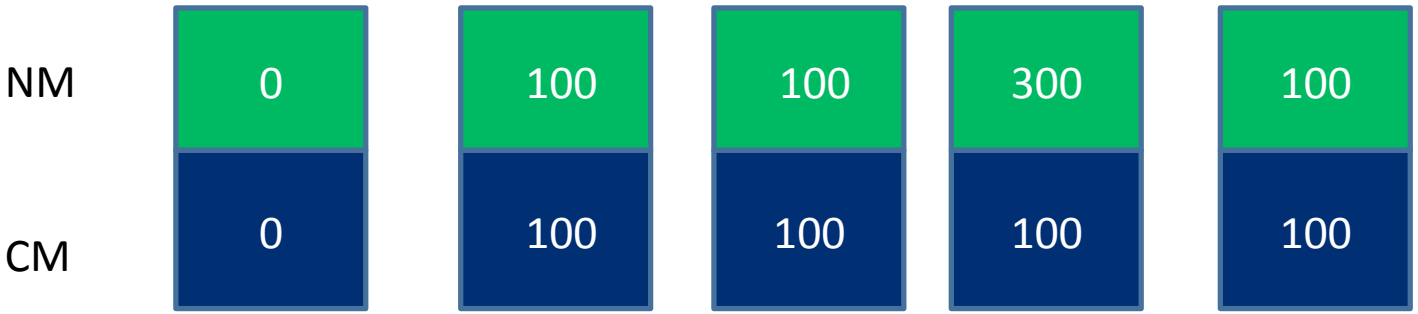
Critical Memory: Model



- Critical store writes to both NM and CM locations
- Normal stores write to NM
- Normal loads read from NM
- Critical load returns CM value
  - Can correct value in NM
  - Can trap on mismatch (debug mode)

Critical Memory: Example





Critical Memory preserves its contents even under memory errors

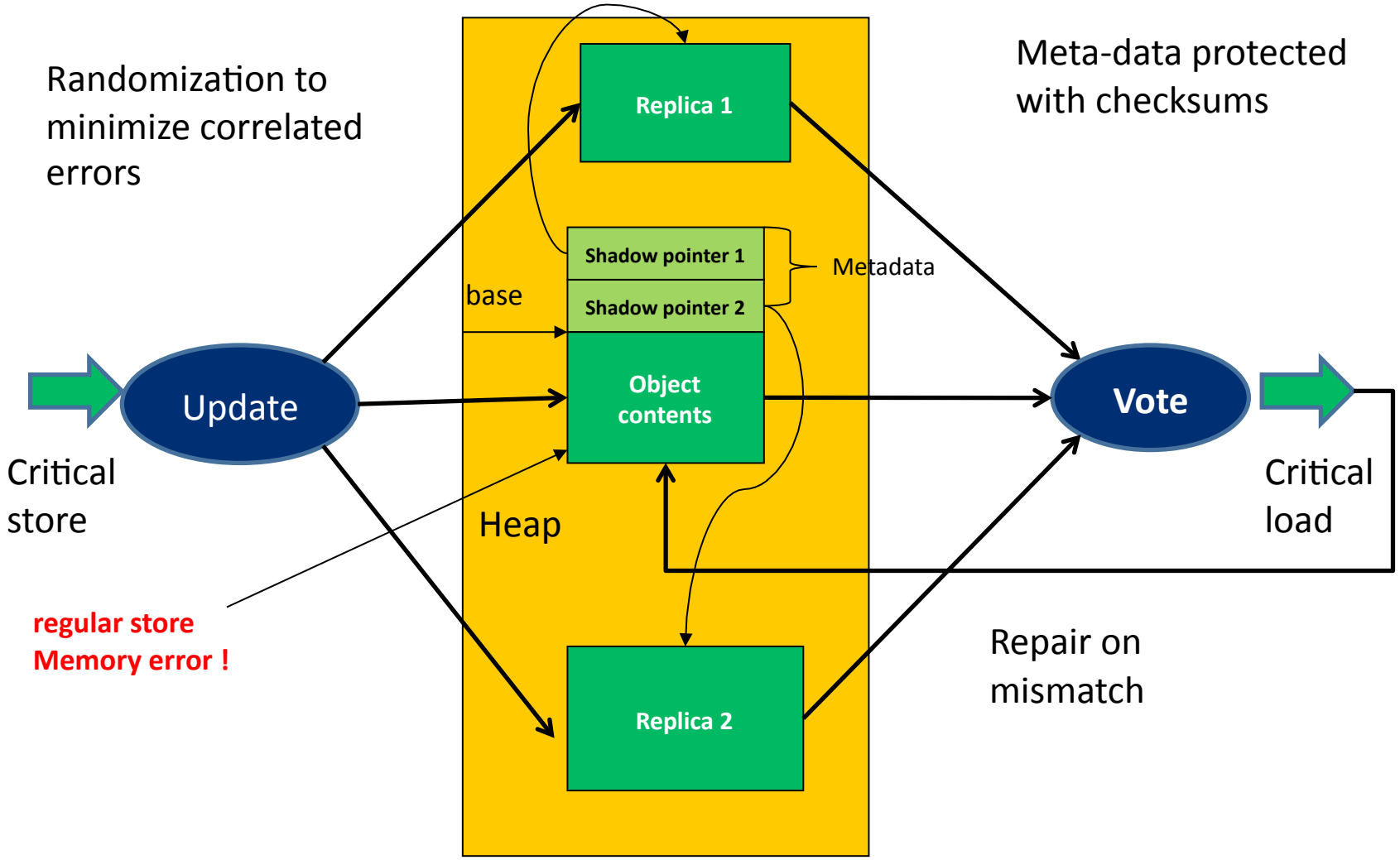
Critical Memory: Library Functions

```
critical int balance ;
int x, y, buffer[10];
balance = 100;
libFunction(&x, &y);
// Check value of balance
promote(&balance);
```

```
libFunction(int* xp, int* yp)
{
        *xp = balance;
        *yp = 2  * balance;
        balance = *xp + *yp;
}
```

Critical Memory allows local reasoning about critical data

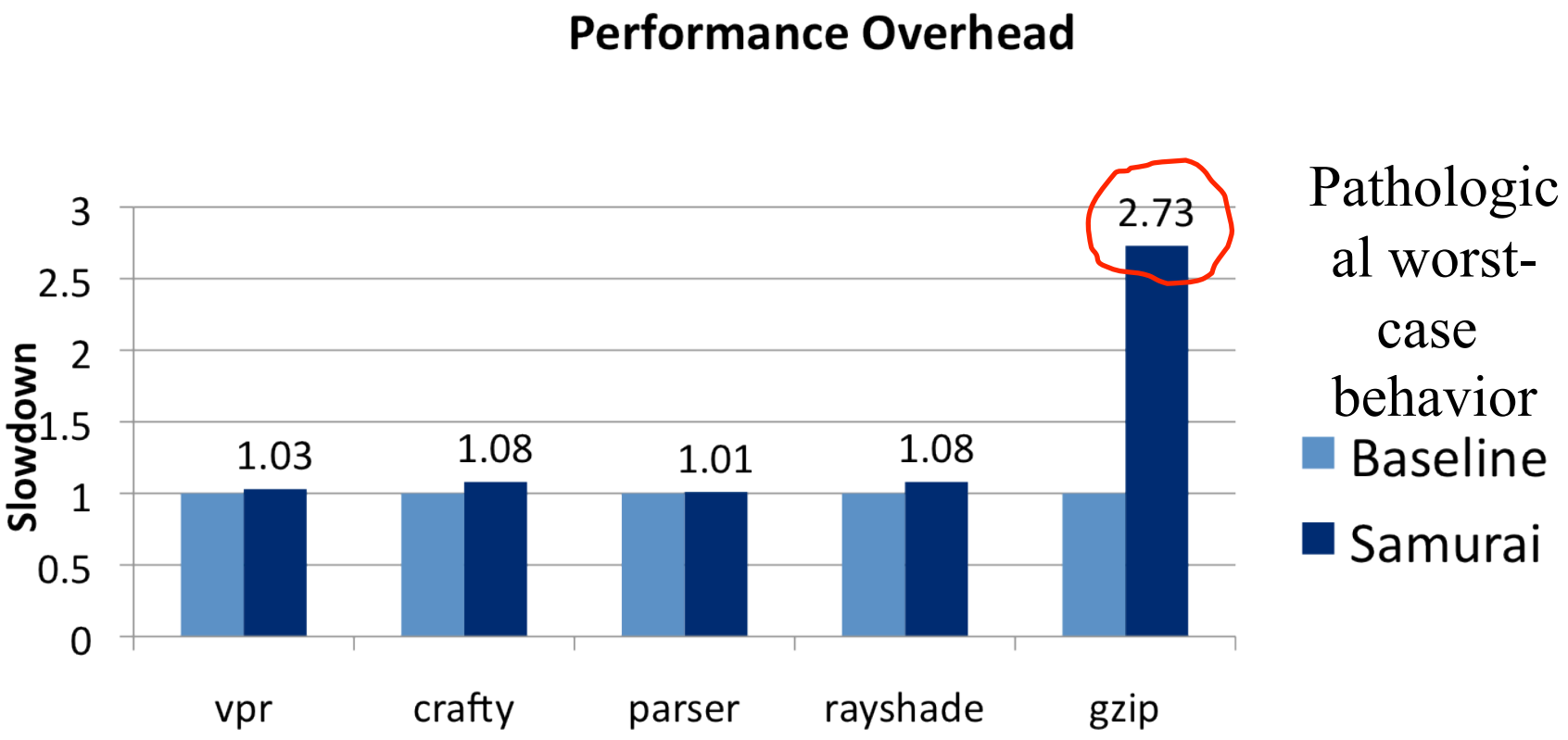
Samurai : Implementation



Samurai: Experimental Setup

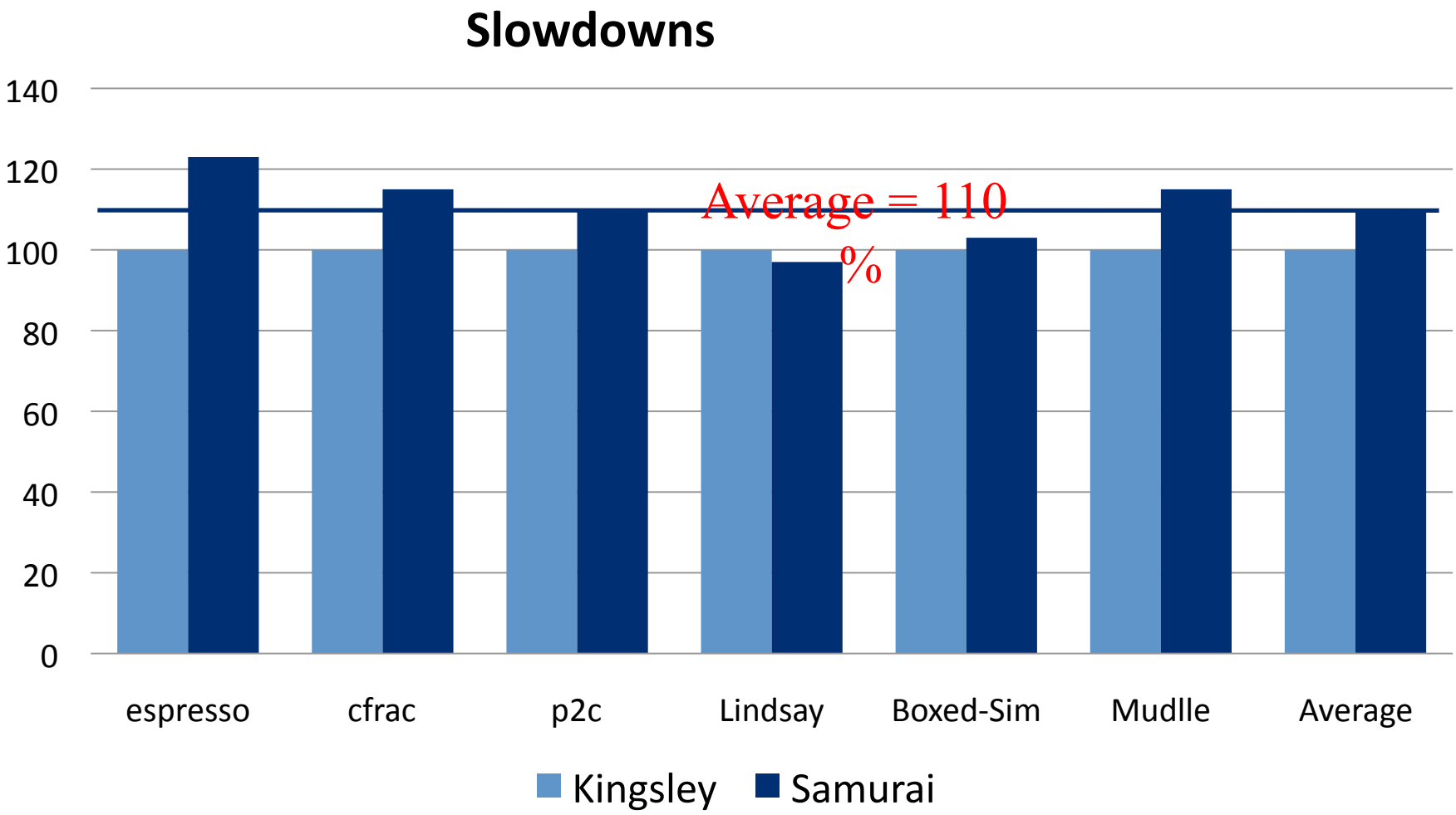
- **Implementation**
  - Automated compiler pass to instrument critical loads and stores
  - Runtime library for critical data allocation/de-allocation (C++)
- **Protected critical data in 5 applications (SPEC2k)**
  - Chose data that is crucial for end-to-end correctness of program
  - Evaluation of performance overhead by direct measurements
  - Fault-injections into critical data to evaluate their resilience
- **Also Protected critical data in libraries**
  - **STL List Class**: Backbone of list structure. Used in web server.
  - **Memory allocator**: Heap meta-data (object size + free list).

Samurai: Application Overheads



Overhead is less than 10% for all applications except gzip

Samurai: Memory Allocator Results



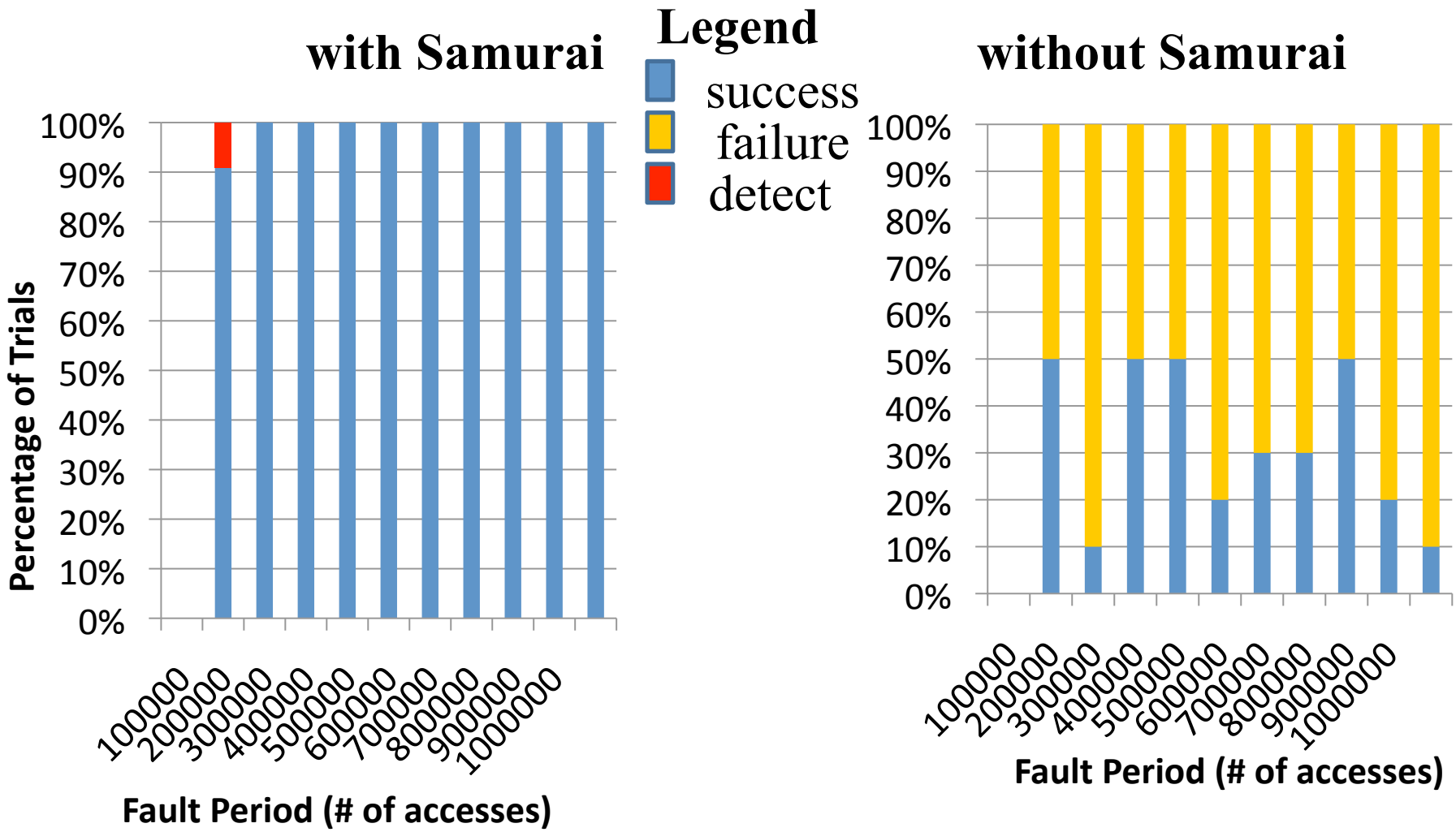
Samurai: Fault Injection Methodology

- **Injections into critical data**
  - Corrupted objects on Samurai heap, one at a time
  - Injected more faults into more populated heap regions (Weighted fault-injection policy)
  - Outcome: success, failure, false-positive
- **Injections into non-critical data**
  - Measure propagation to critical data
  - Corrupted results of random store instructions
  - Compared memory traces of verified stores
  - Outcomes: control error, data error, pointer error

Fault Injection into Non-Critical Data

App	Number of Trials	Control Errors	Data Errors	Pointer Errors	Assertion Violations	Total Errors
vpr	550 (199)	0	203 (0)	1 (0)	2 (2)	**203 (0)**
crafty	55 (18)	12 (7)	9 (3)	4 (3)	0	**25 (13)**
parser	500 (380)	0	3 (1)	0	0	**3 (1)**
rayshade	500 (68)	0	5 (1)	0	1 (1)	**5 (1)**
gzip	500 (239)	0	1 (1)	2 (2)	157 (157)	**3 (3)**

Fault Injection into Critical Data



Samurai/Critical Memory: Summary

- **Critical Memory: Abstract Memory Model**
  - Reason about critical data in applications
  - Define special operations: critical loads/stores
  - Inter-operation with un-trusted third-party code

- **Samurai: Software Prototype of CM**
  - Uses replication and forward error-correction
  - Demonstrated on both applications and libraries
    - Performance overheads of 10 % or less in most cases

Learning Objectives

- Define Software Fault-tolerance and enumerate its challenges
- List three diversity-based techniques and evaluate their respective pros and cons
- Use robust data structures for structural integrity
- Use critical memory for semantic integrity