Good Enough Computer Systems: Reliability on the Cheap

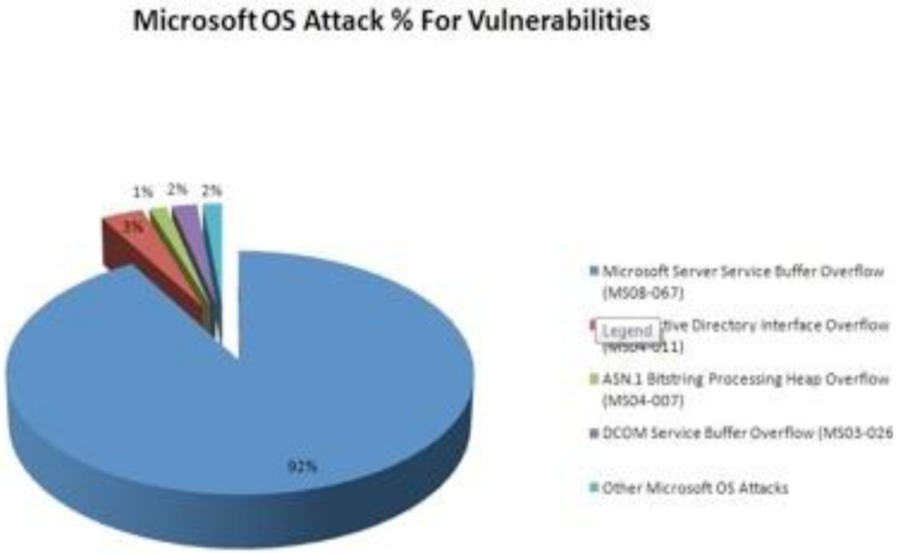
Karthik Pattabiraman

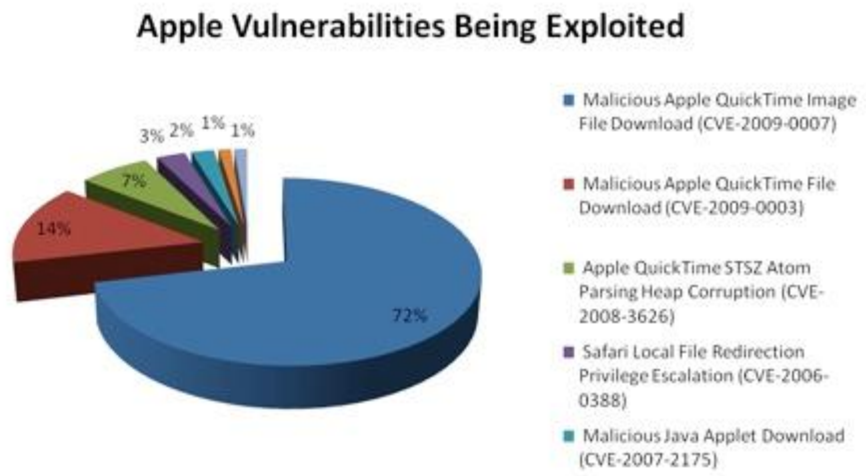
Electrical and Computer Engineering

Motivation: Memory Corruption

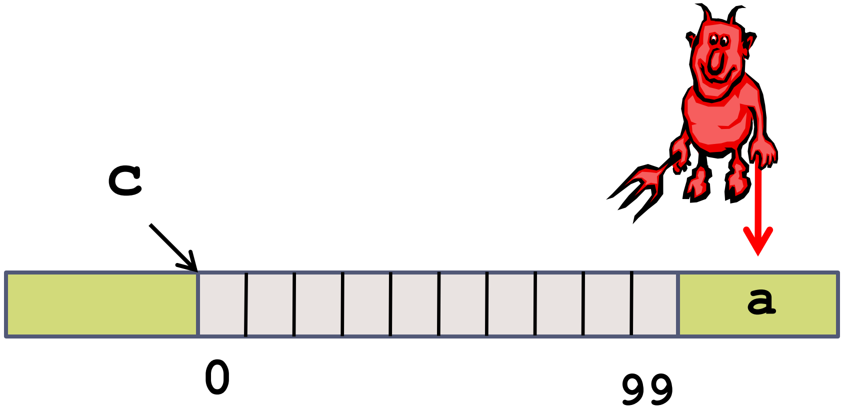
- Memory corruption errors are a leading cause of vulnerabilities in type-unsafe languages (C/C++)
  - C/C++ still among most used languages in real-world
  - Attackers continue to exploit mem. corruption errors

Source: sans.org (2009)

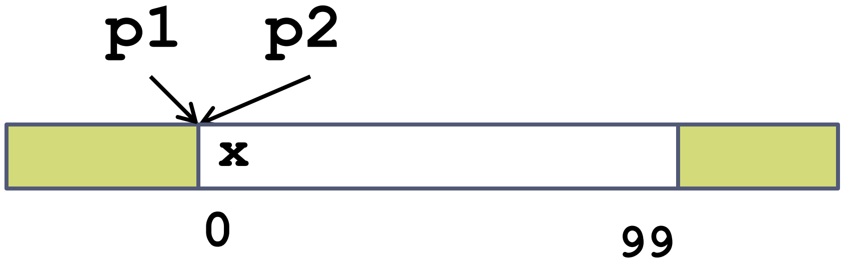




Background: Memory Corruption Errors



c[101] = ‘\n\’;



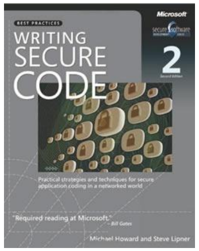
p1 = p2;

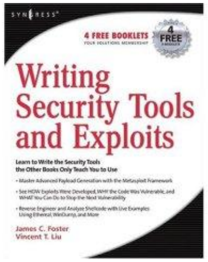
free(p1);

- 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 – no guarantees otherwise

Motivation: Hardware Memory Errors

- Memory elements are susceptible to soft-errors (cosmic ray strikes, alpha particles etc.)
- Variation in retention times among DRAM cells
  - Anywhere from a few milli-seconds to a few seconds

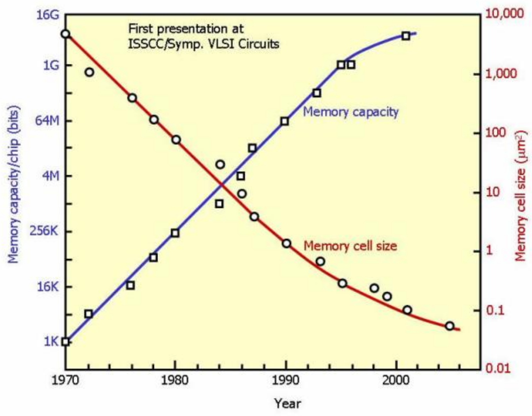


Figure 1

Figure from [Itoh’08]

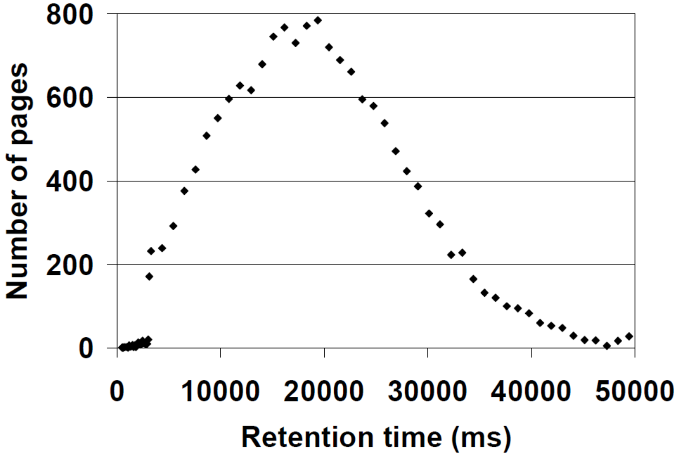


Figure from [Venkatesan’06]

Hardware Memory Errors: Solutions

- **Use of ECC memory**
  - Majority of commodity systems don't have ECC
  - Multi-bit errors and hard faults are becoming increasingly common [Li'07] [Schroeder'09]
- **Guard-band and over-provision for worst case**
  - Wastes power and leads to sub-optimal designs
    - Example: Set DRAM refresh times to 32-64 ms when idle, though only a small fraction of cells require such high rates

Average Worst-case

Take-away Observations/Goals

- **Need protection from both software memory corruption and hardware memory errors**
- **Must not require rewriting of code in safe languages or checking all memory writes**
- **Performance and energy overheads are important considerations for any technique**

How do we satisfy all three goals ?

The “Good Enough” Revolution

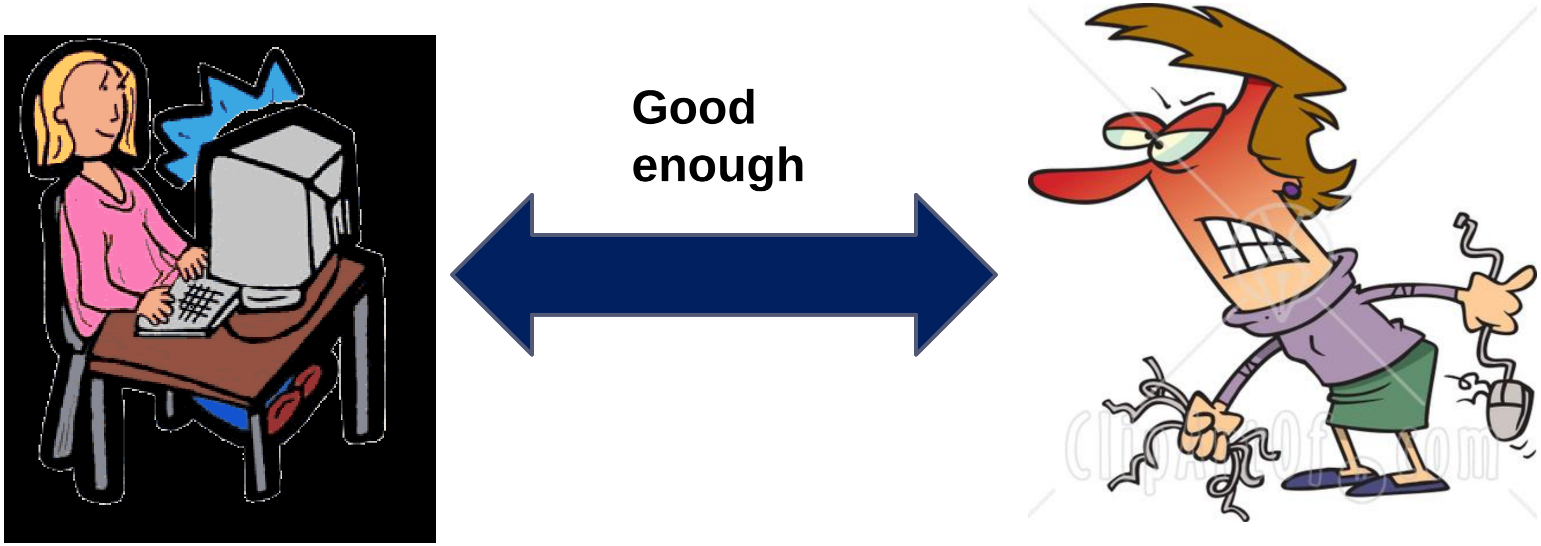
Source: WIRED Magazine (Sep 2009) – Robert Kapps
http://www.wired.com/gadgets/miscellaneous/magazine/17-09/ff_goodenough

People prefer “cheap and good-enough” over “costly and near-perfect”

Can we design computer systems with this principle ?

"Good Enough" Computer Systems

- **Just reliable enough to get the job done**
  - Do not provide the illusion of perfection to end user
  - But do not fail catastrophically or cause severe errors
  - Depends on the application and users



Good enough

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**

Outline

- Motivation and Overview

- **Samurai:** Protection of critical data from memory corruption errors in 3rd party modules [Eurosys'08]
  - In collaboration with Vinod Grover, Ben Zorn (MSR)

- **Flicker:** Protection of critical data from hardware errors introduced by power-saving features [TR'09]
  - In collaboration with Thomas Moscibroda, Ben Zorn (MSR) and Song Liu (Northwestern University)

- Future Directions and Conclusions

Outline

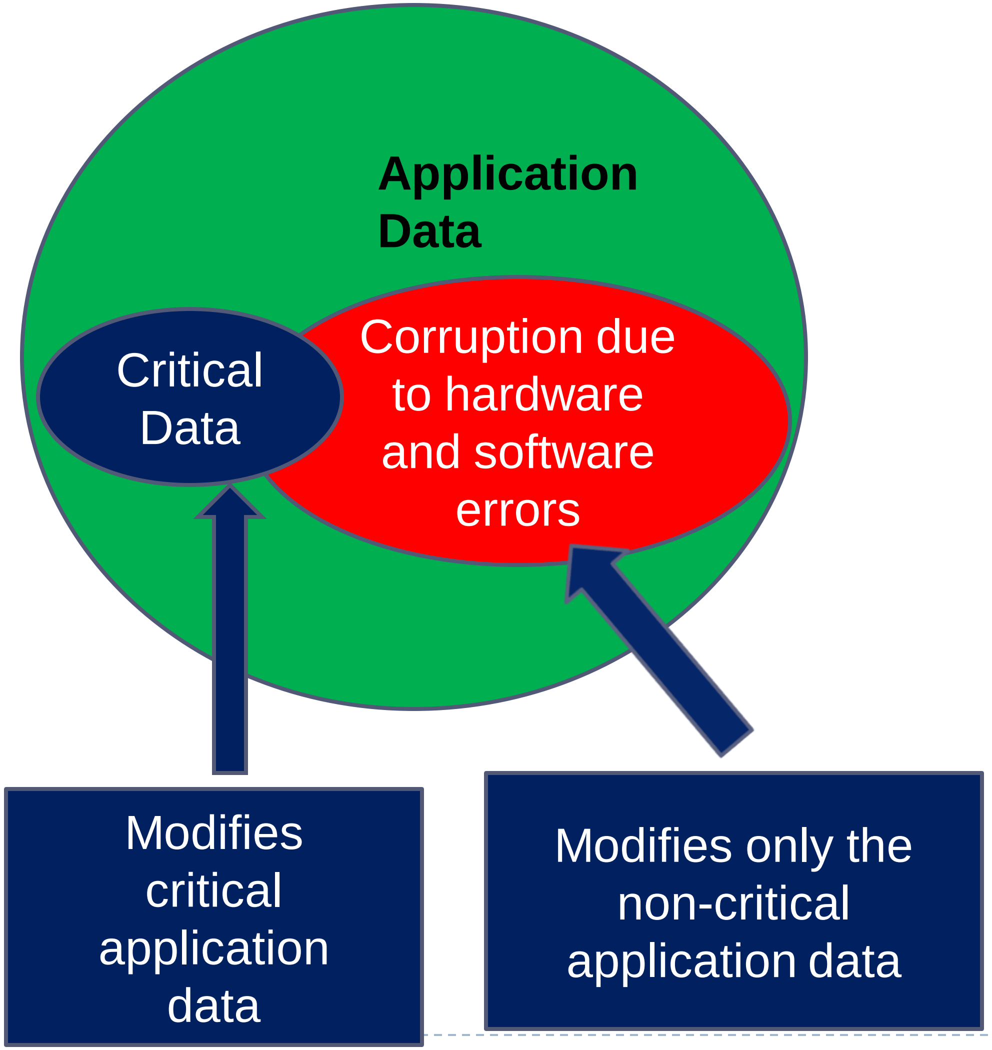
- Motivation and Overview

- **Samurai:** Protection of critical data from memory corruption errors in 3rd party modules [Eurosys'08]
  - In collaboration with Vinod Grover, Ben Zorn (MSR)

- **Flicker:** Protection of critical data from hardware errors introduced by power-saving features [TR'09]
  - In collaboration with Thomas Moscibroda, Ben Zorn (MSR) and Song Liu (Northwestern University)

- Future Directions and Conclusions

Samurai: Goals



Application Data

Critical Data

Corruption due to hardware and software errors

Modifies critical application data

Modifies only the non-critical application data

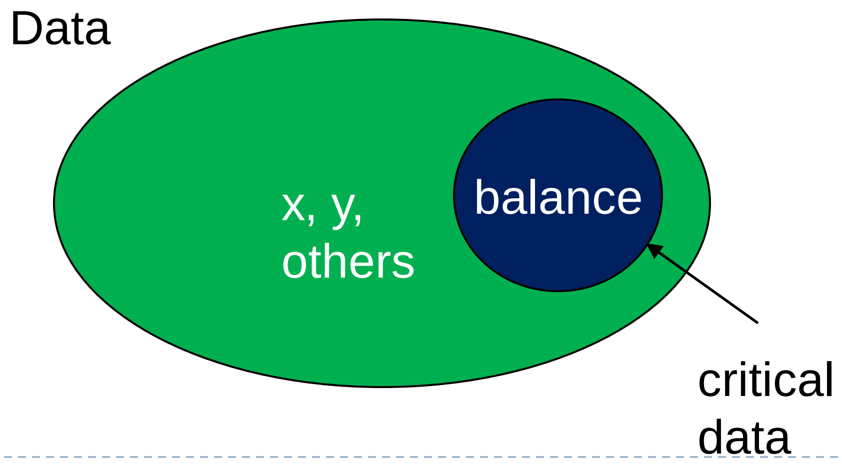
Critical data integrity should be preserved even if other data is corrupted

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

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

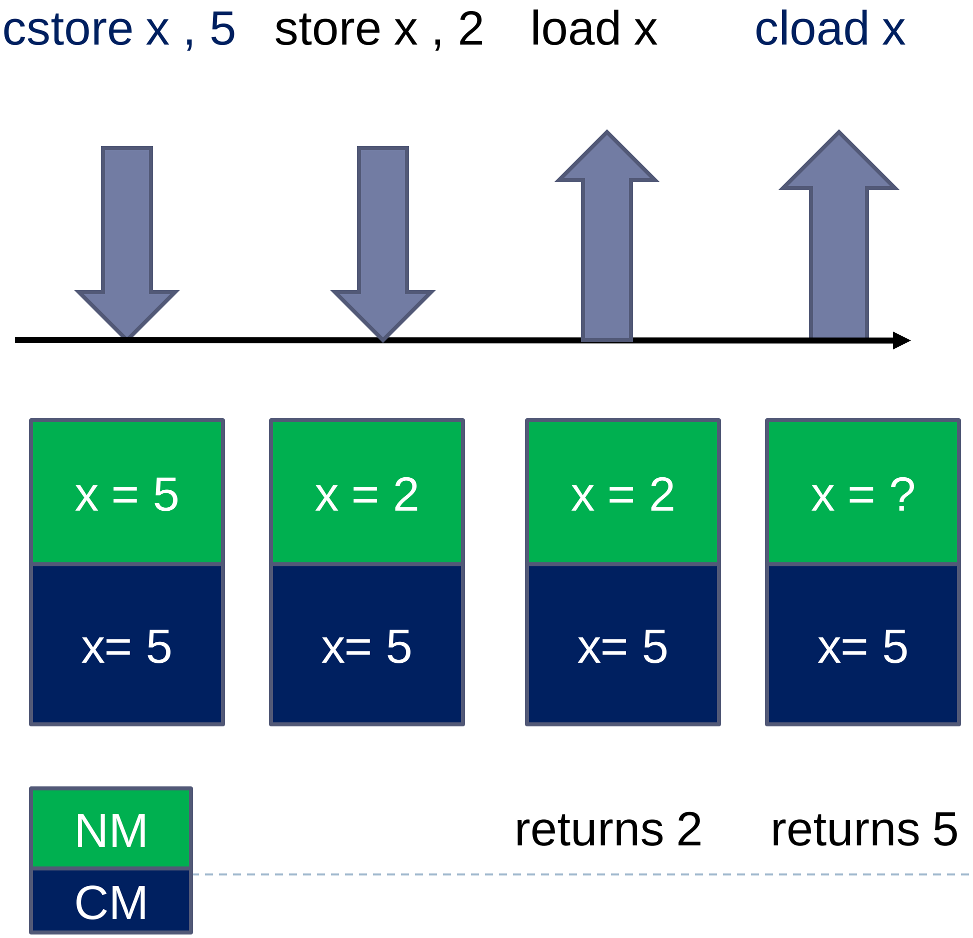
Samurai: Critical Memory Abstraction

```
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*)

Samurai : Critical Memory Model



cstore x , 5 store x , 2 load x cload x

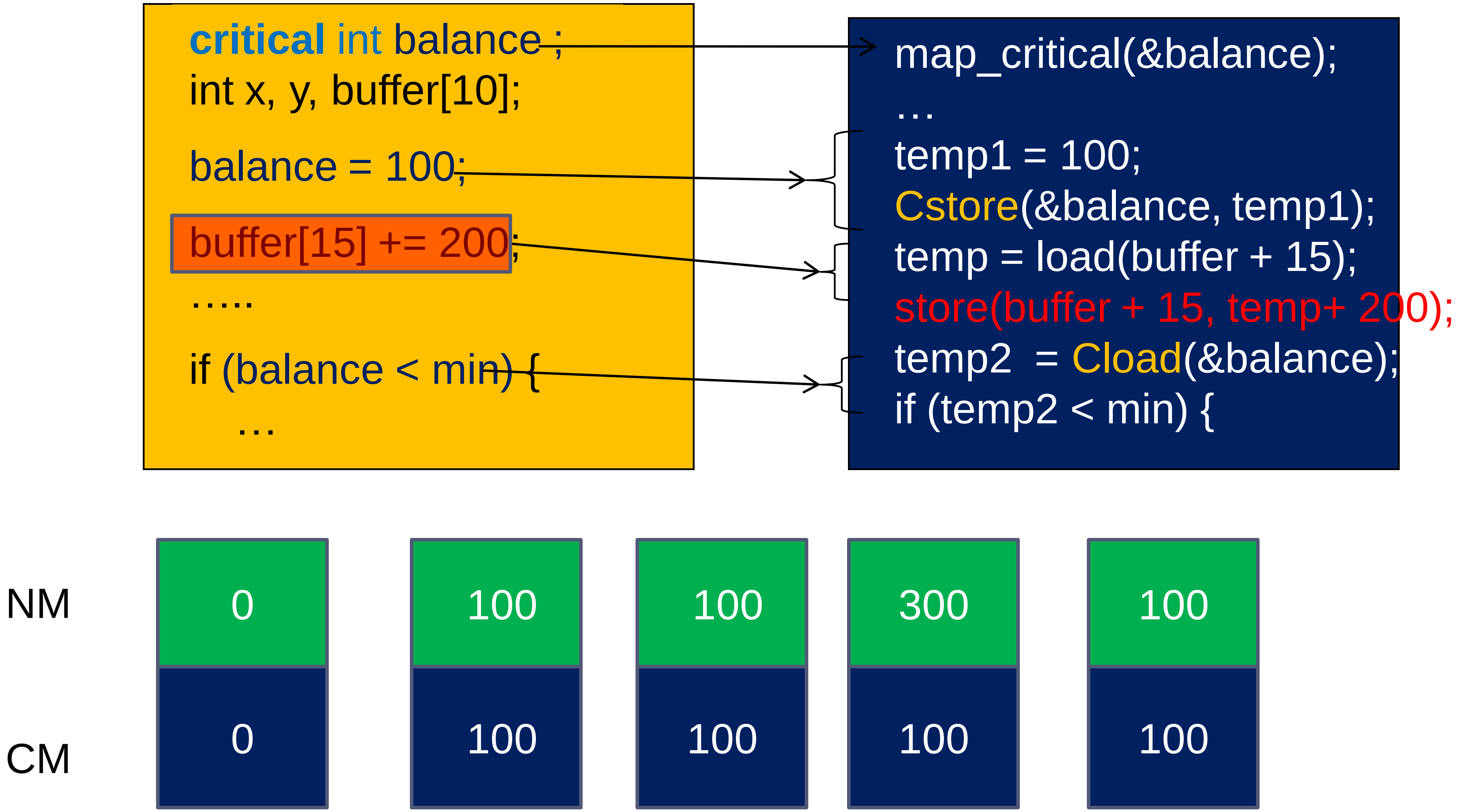
x = 5	x = 2	x = 2	x = ?
x= 5	x= 5	x= 5	x= 5
		returns 2	returns 5

NM

CM

- 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)

Samurai : Example



```
critical int balance ;
int x, y, buffer[10];

balance = 100;

buffer[15] += 200;
…..

if (balance < min) {
    …
```

```
map_critical(&balance);
…
temp1 = 100;
Cstore(&balance, temp1);
temp = load(buffer + 15);
store(buffer + 15, temp+ 200);
temp2  = Cload(&balance);
if (temp2 < min) {
```

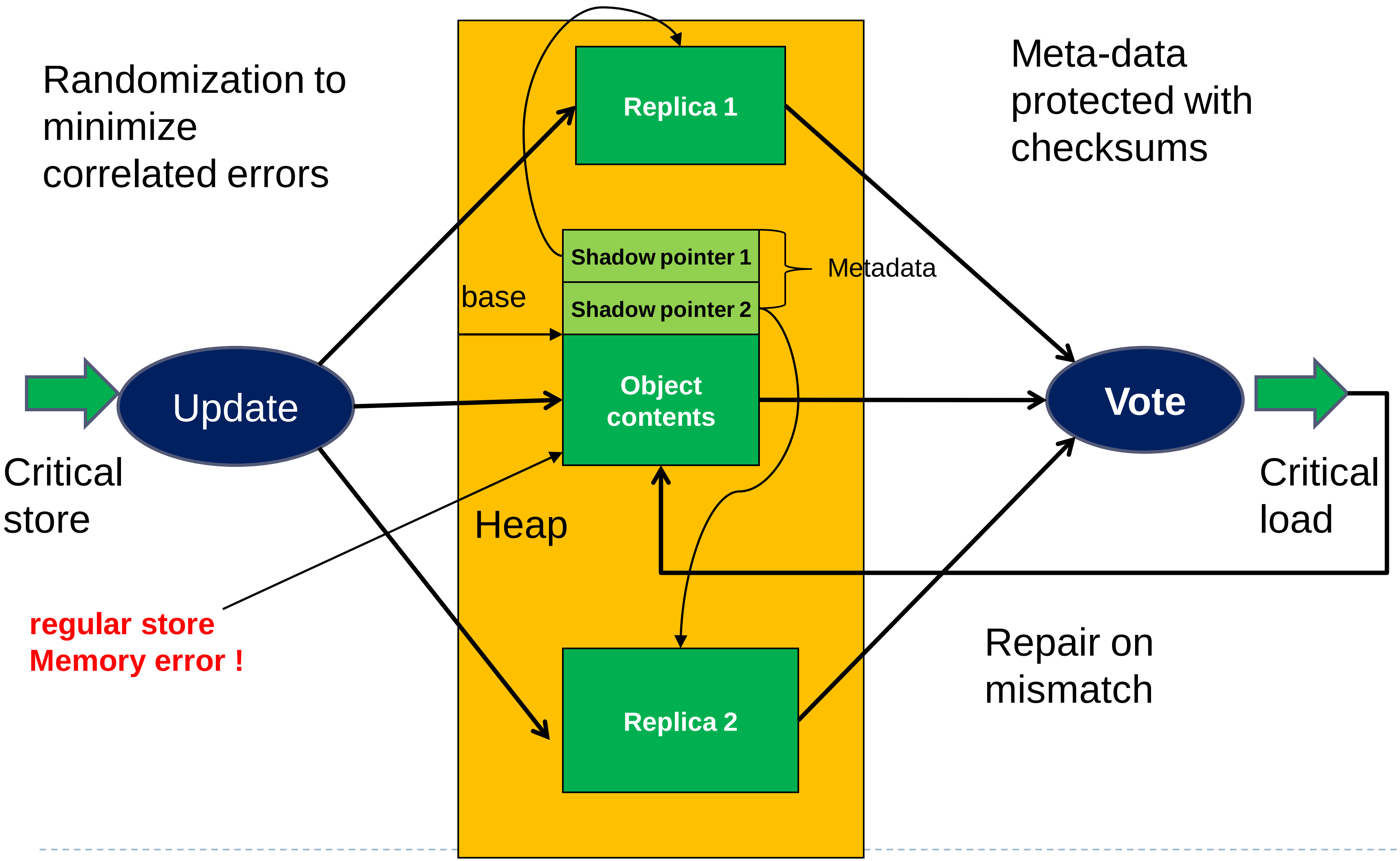
NM	0	100	100	300	100
CM	0	100	100	100	100

Critical Memory preserves its contents even under memory errors

Samurai : Implementation

Randomization to minimize correlated errors

Meta-data protected with checksums



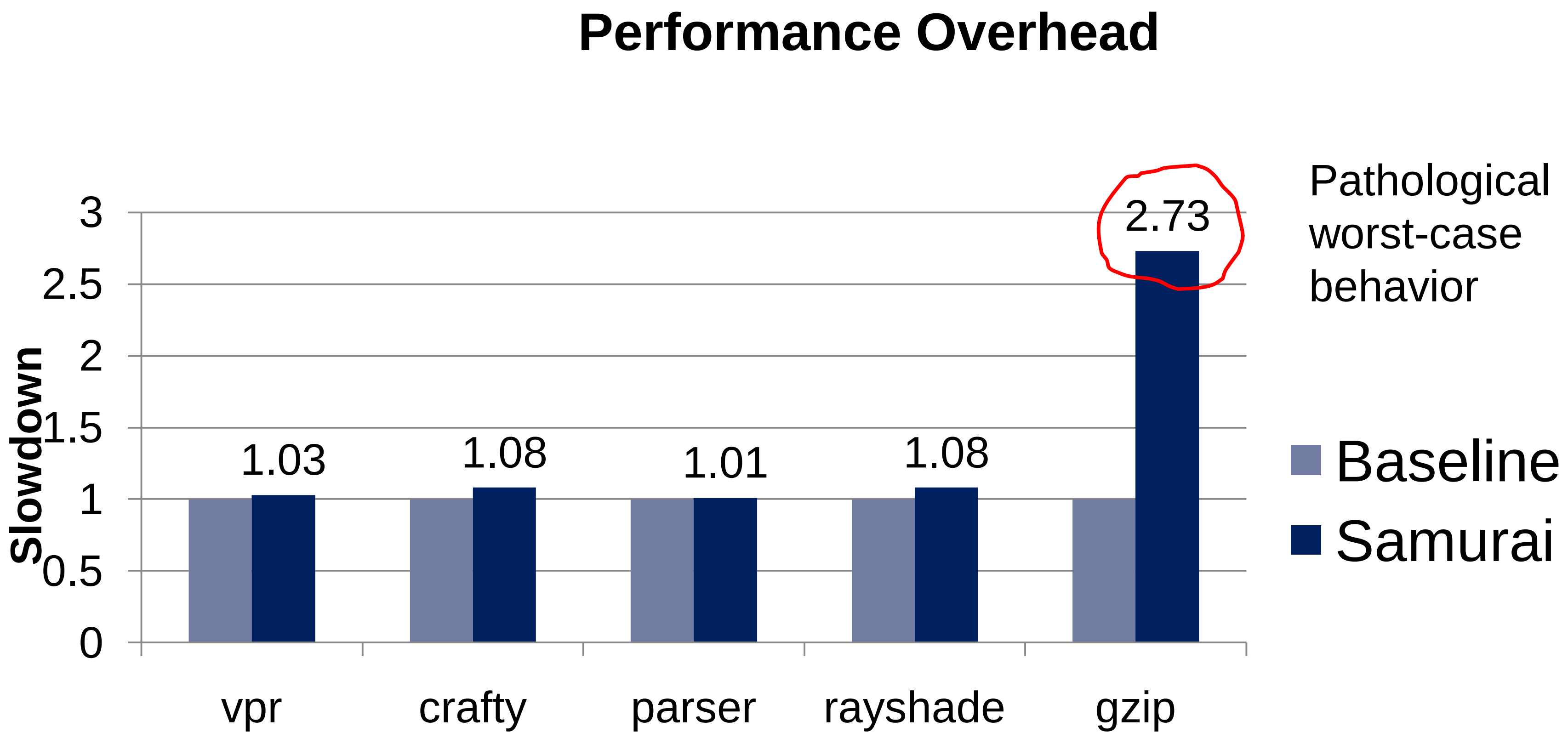
regular store
Memory error !

Repair on mismatch

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).

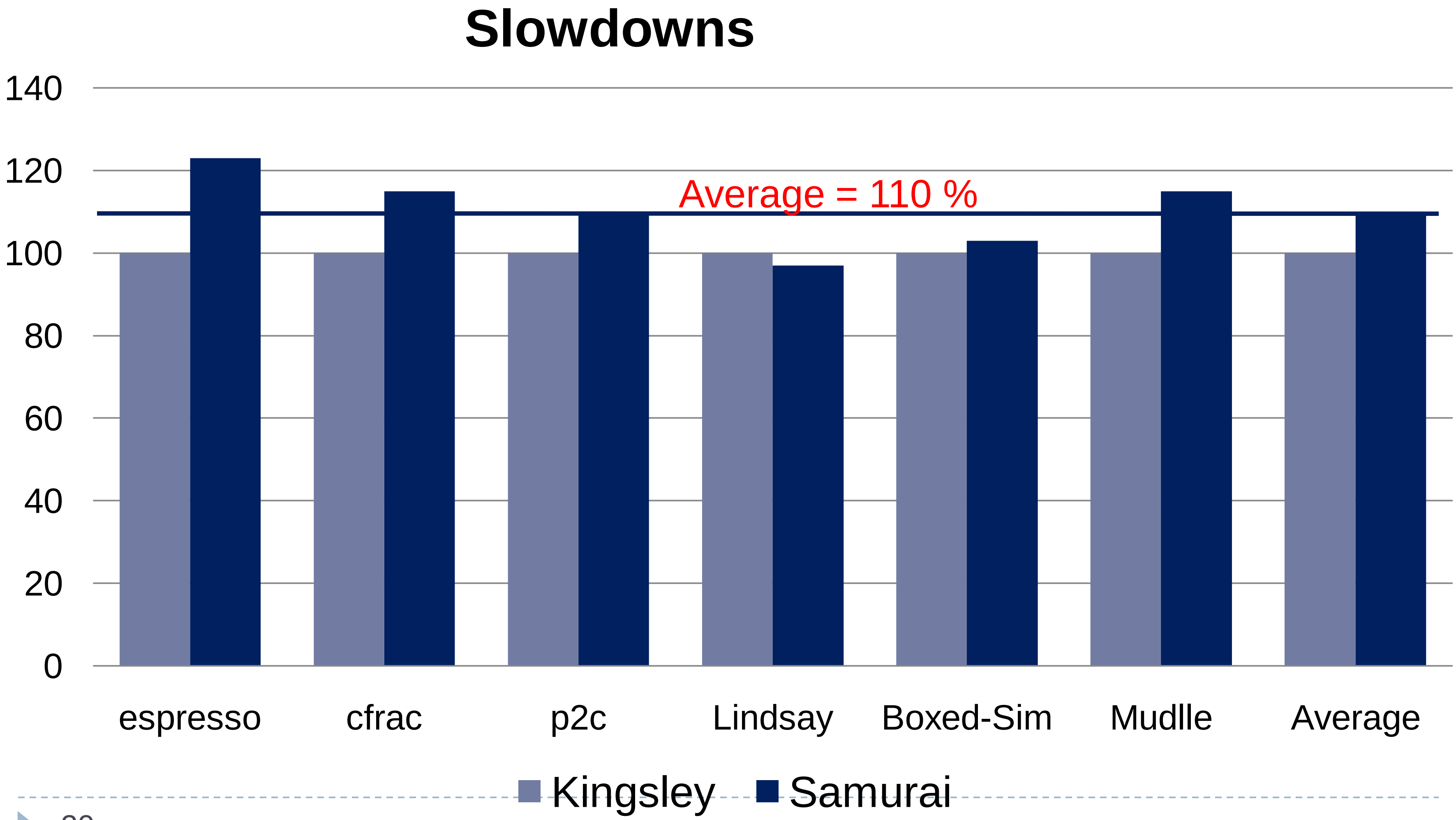
Samurai: Application Overheads



Pathological worst-case behavior

Overhead is less than 10% for all applications except gzip

Samurai: Memory Allocator Results

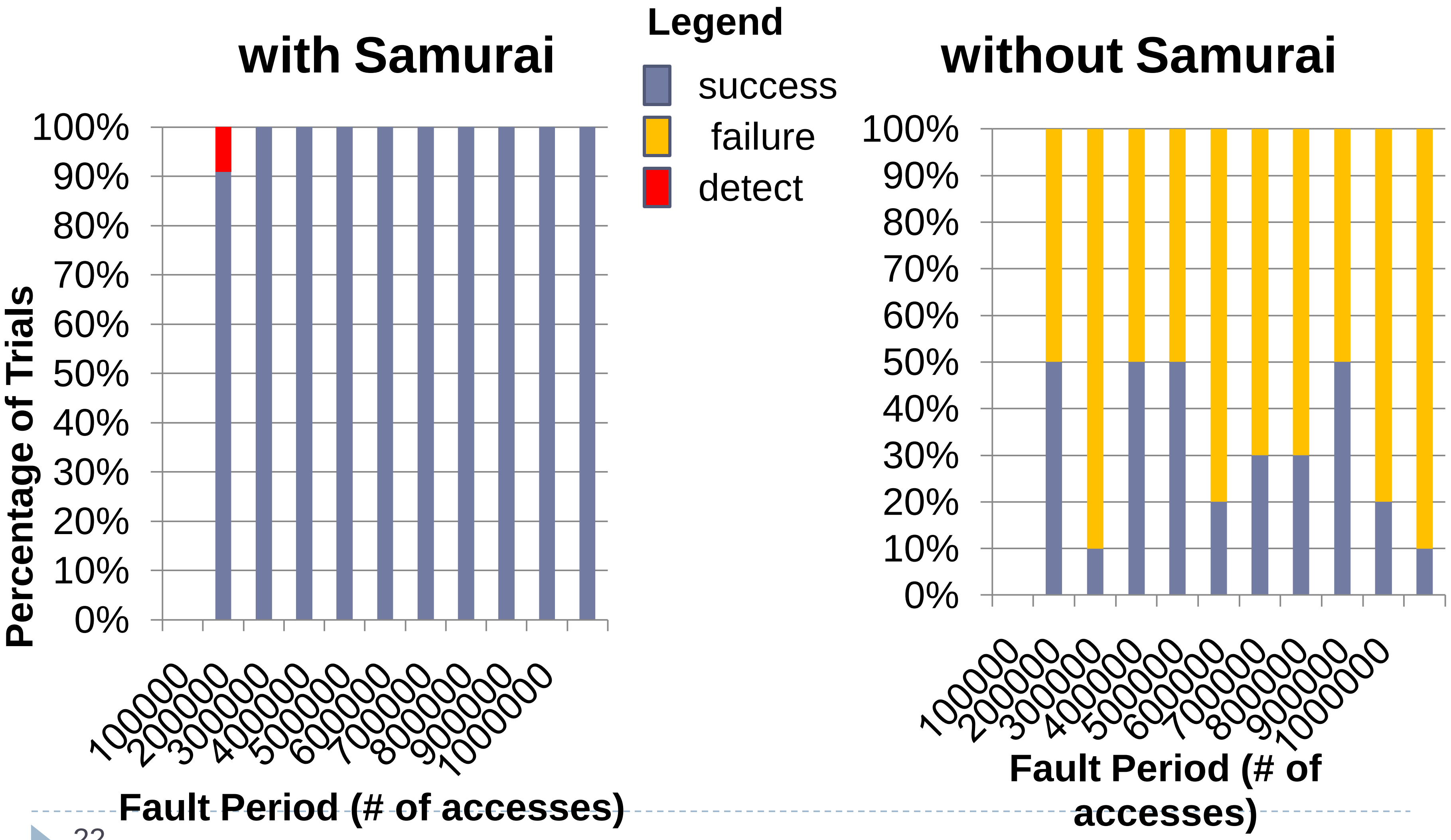


Samurai: STL Class and a WebServer

- **STL List Class**
  - Protected list backbone (pointers) and data
  - Modified memory allocator for class
  - Modified member functions *insert, erase*
  - Modified custom iterators for list objects

- **Webserver**
  - Used STL list class for maintaining client connection information
  - Multi-threaded
  - Evaluated across multiple threads and connections
  - Max performance overhead = 9 %

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
    - Corrects almost all memory corruption errors in critical data

Outline

- Motivation and Overview

- **Samurai:** Protection of critical data from memory corruption errors in 3rd party modules [Eurosys'08]
  - In collaboration with Vinod Grover, Ben Zorn (MSR)

- **Flicker:** Protection of critical data from hardware errors introduced by power-saving features [TR'09]
  - In collaboration with Thomas Moscibroda, Ben Zorn (MSR) and Song Liu (Northwestern University)

- Future Directions and Conclusions

Flicker: Smartphones

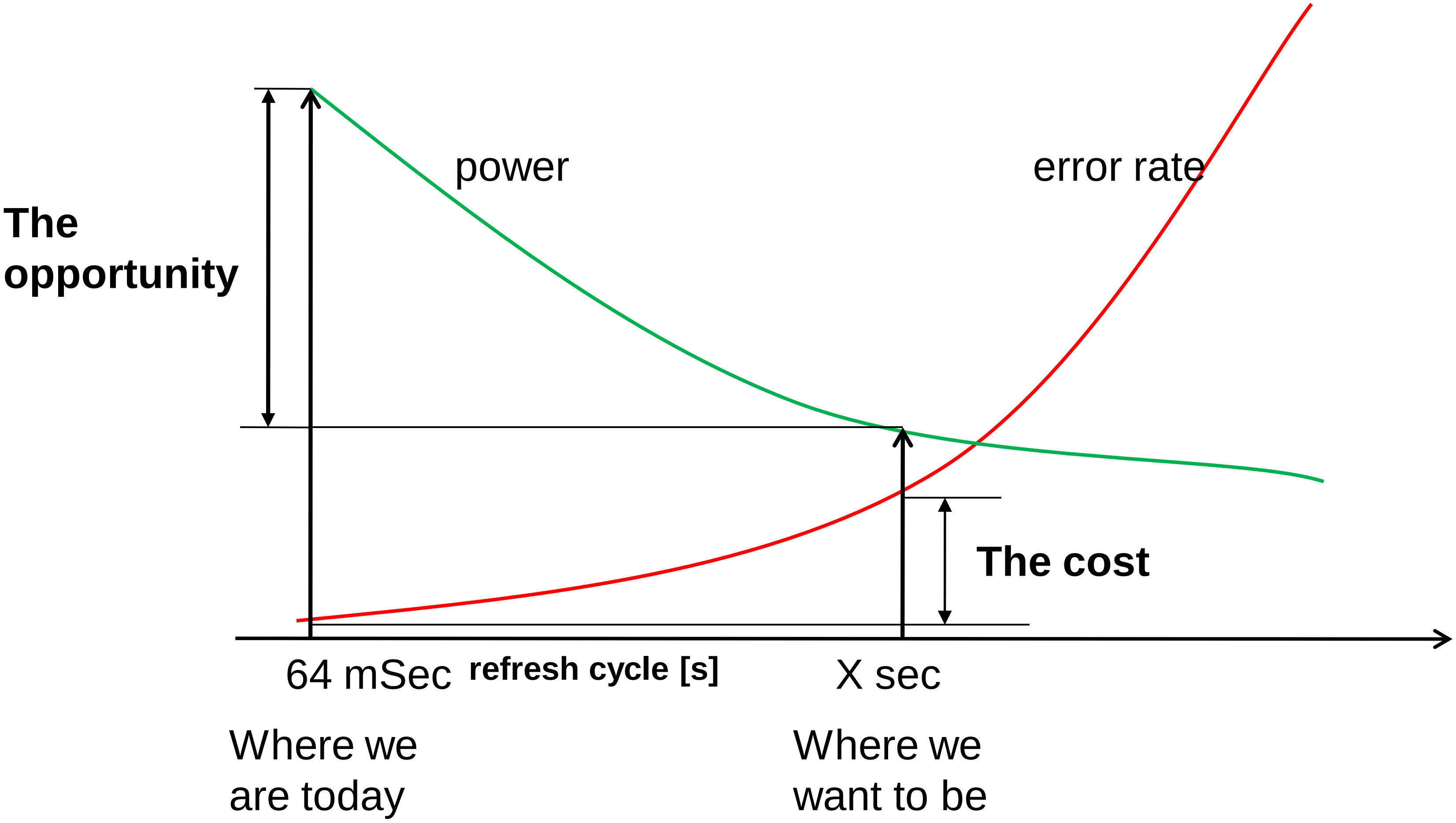
Smartphones becoming ubiquitous

Responsiveness is important

DRAM Memory consumes up to 30% of power

Can drain the battery even when idle

Flicker: DRAM Refresh

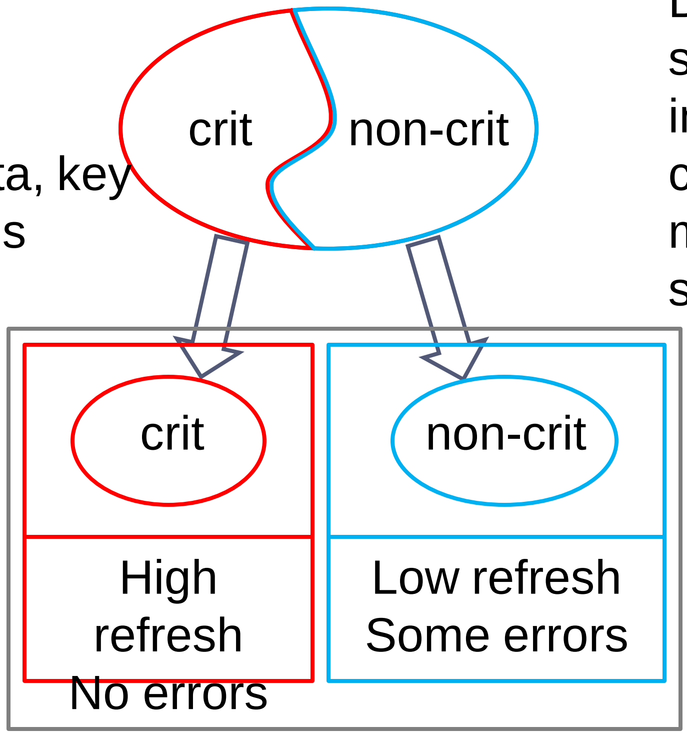


If software is able to tolerate errors, we can lower refresh rates to achieve considerable power savings

Flicker: Approach

- **Critical / non-critical data partitioning**

Important for application correctness e.g., meta-data, key data structures



crit

non-crit

crit

non-crit

High refresh
No errors

Low refresh
Some errors

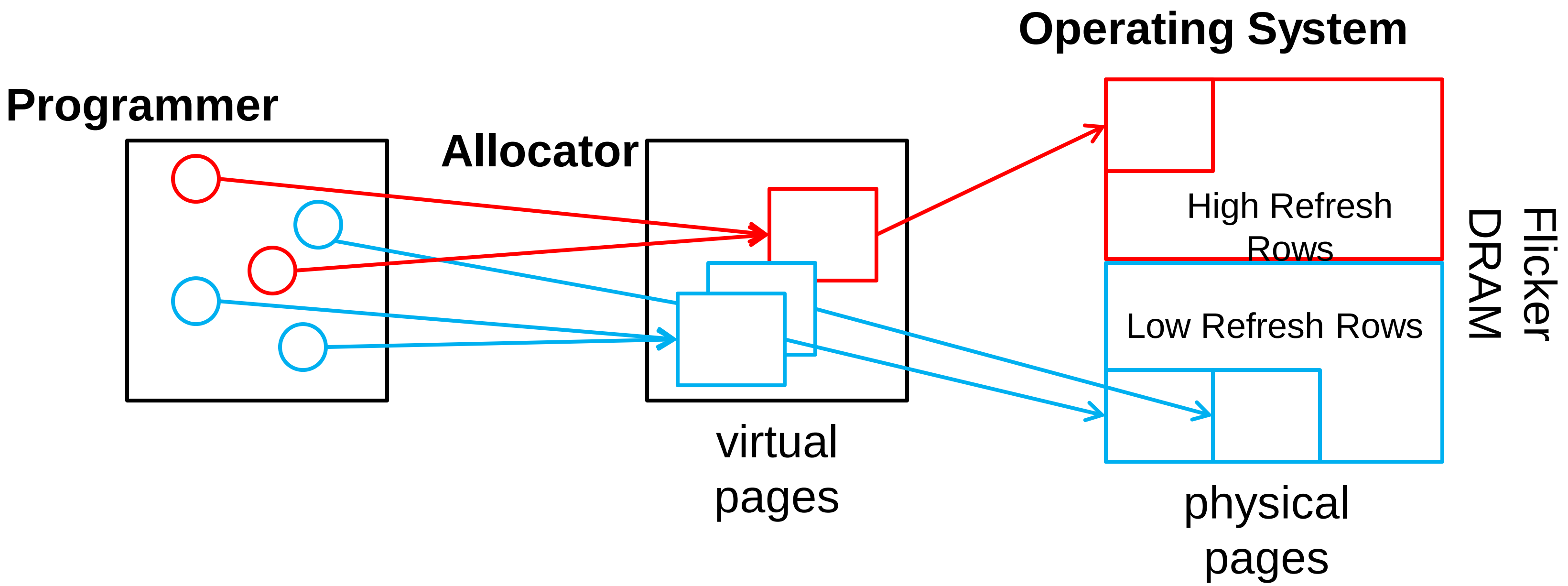
Does not substantially impact app correctness e.g., multimedia data, soft state

Flicker DRAM

Mobile applications have substantial amounts of non-critical data that can be easily identified by application

Flicker: Software Implementation

Minor changes to the memory allocator and the OS (memory manager)



○ critical object

○ non-critical object

🟥 critical page

☐ non-critical page

Flicker: Summary

- **First software technique to intentionally lower hardware reliability for energy savings**
  - Minimal changes to hardware – based on PASR mode
  - No modifications required for legacy applications

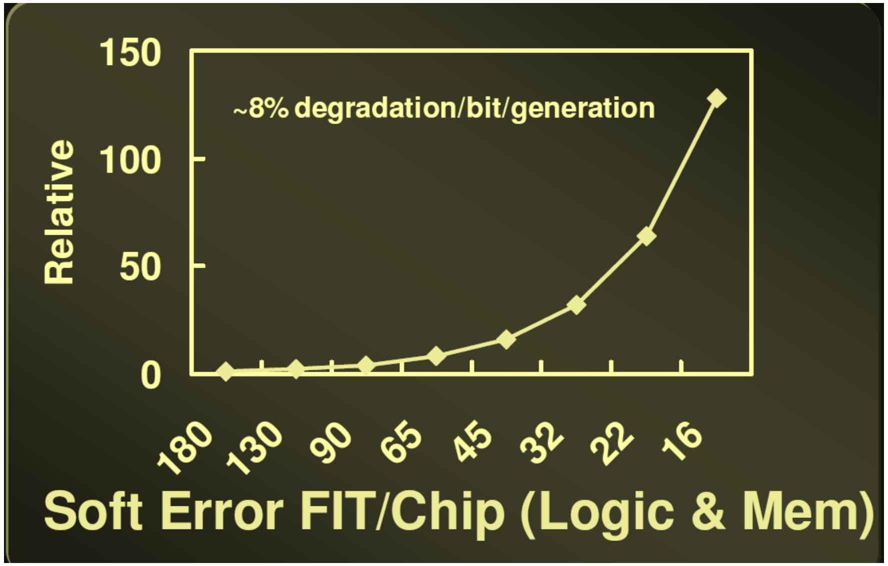
- **Reduced overall DRAM power by 20-25% with negligible loss of performance (< 1 %) and reliability across five application classes**
  - Took less than a day to partition each application
  - No crashes reported even at 1 second refresh rate
  - Minor degradation in output quality of two applications
    - Discernible to human eye only if image is zoomed by 5X

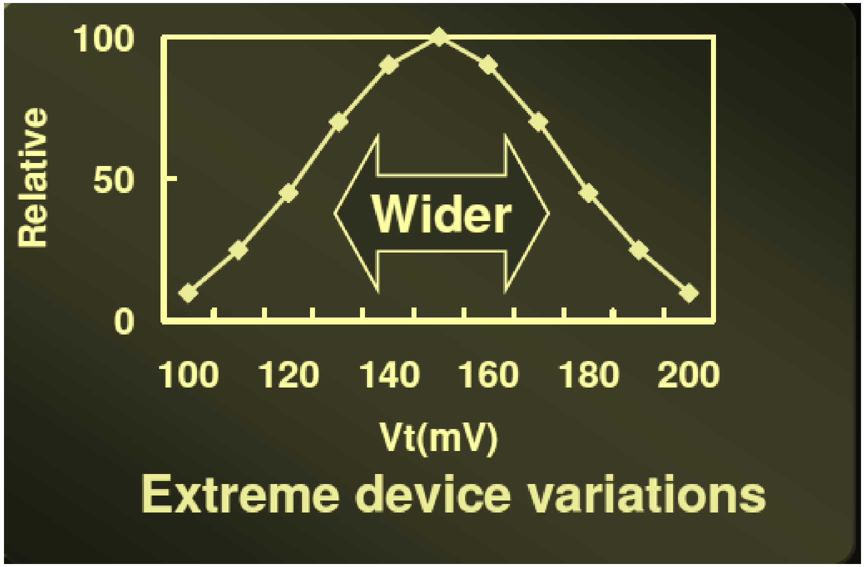
Outline

- Motivation and Overview
- **Samurai:** Protection of critical data from memory errors in 3rd party modules [Eurosys'08]
  - In collaboration with Vinod Grover, Ben Zorn (MSR)
- **Flicker:** Protection of critical data from hardware errors introduced by power-saving features [TR'09]
  - In collaboration with Thomas Moscibroda, Ben Zorn (MSR) and Song Liu (Northwestern University)
- Future Directions and Conclusions

Future Work: Processor Errors

- Errors are becoming more common in processors
  - Soft Errors and manufacturing variations (timing errors)
  - Processors experience wear-outs and thermal hotspots

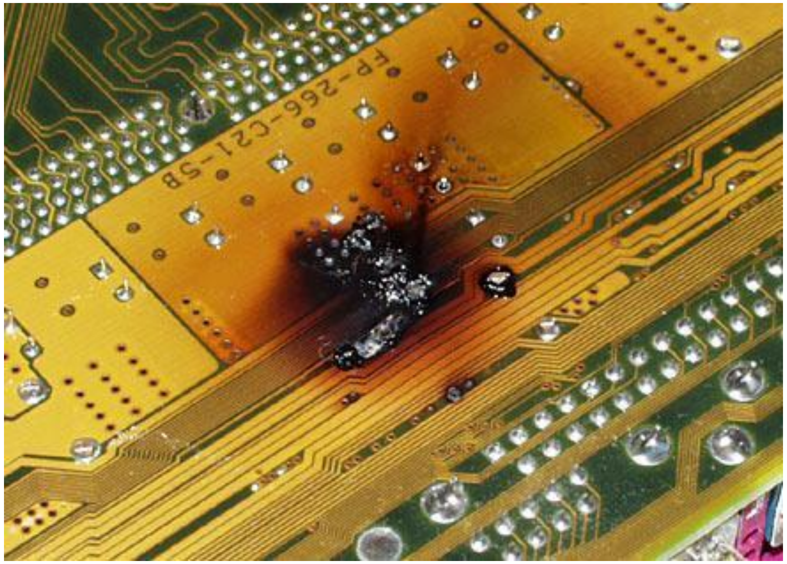




Source: Shekar Borkar (Intel) - Stanford talk in 2005

Future Work: Traditional Solutions

- Duplication is the most commonly-used solution to mask h/w errors (e.g., IBM Mainframe z-series)
- However, duplication consumes large amounts of power – not desirable in commodity systems



Ongoing Directions – this project

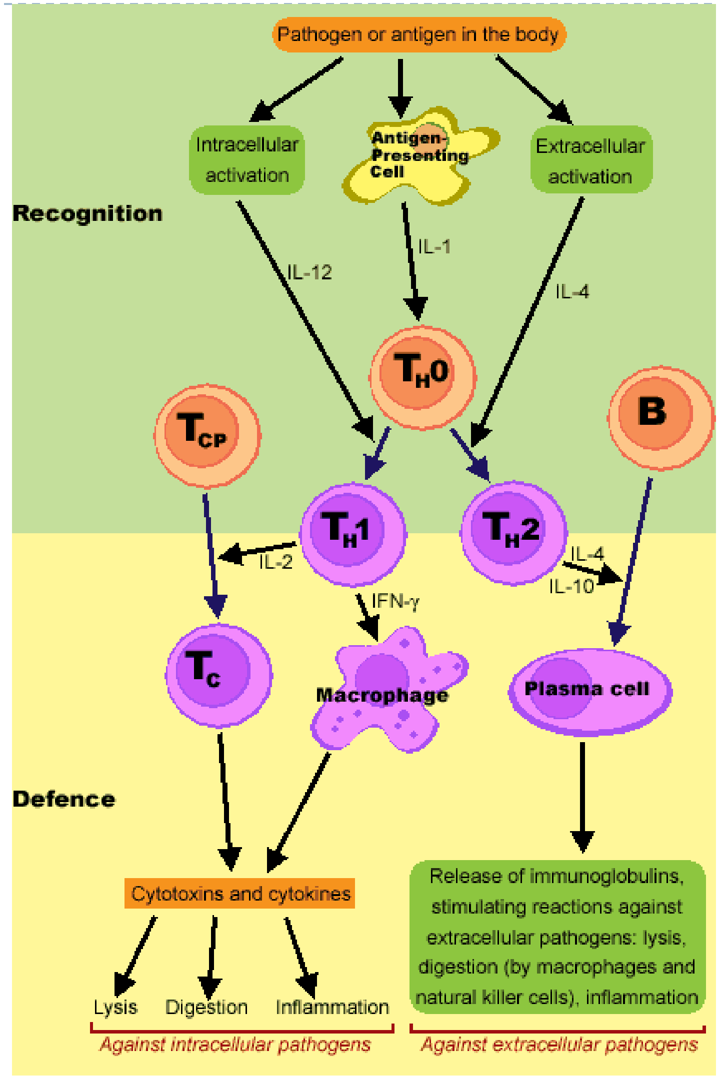
- **Exposing computational (processor) errors to the software and handling the errors in software**
  - Identification of critical code segments and variables
  - Compiler techniques to insert checks into programs
  - Runtime systems to initiate diagnostic and recovery actions

- **Formal methods to reason about the effects of hardware errors on software programs**
  - Model-checking to reason about error propagation in programs
  - Type-systems to ensure correctness of protection mechanisms

- **Developing probabilistic notions of program correctness at the algorithmic level (similar to big O)**

Vision: Software as an Immune system

- Engineering of software systems that anticipate and handle errors in both hardware and in (other) software
  - Minimal intervention from programmers
  - First detect and diagnose the source of the errors
  - Then defend against the detected errors by taking appropriate actions



Release of immunoglobulins, stimulating reactions against extracellular pathogens: lysis, digestion (by macrophages and natural killer cells), inflammation

Source: mcld.co.uk

Conclusions

- **Software systems should provide “good enough” reliability in the face of errors**

- **Protect critical data in applications with low performance and resource overheads**
    - **Samurai** – to protect critical data from memory corruption errors in third-party modules (using selective replication)
    - **Flicker** – to protect critical data from hardware errors introduced by highly-aggressive power saving features (using data partitioning)
    - **Future Work**: Focus on computational errors and how software can be built to work around such errors