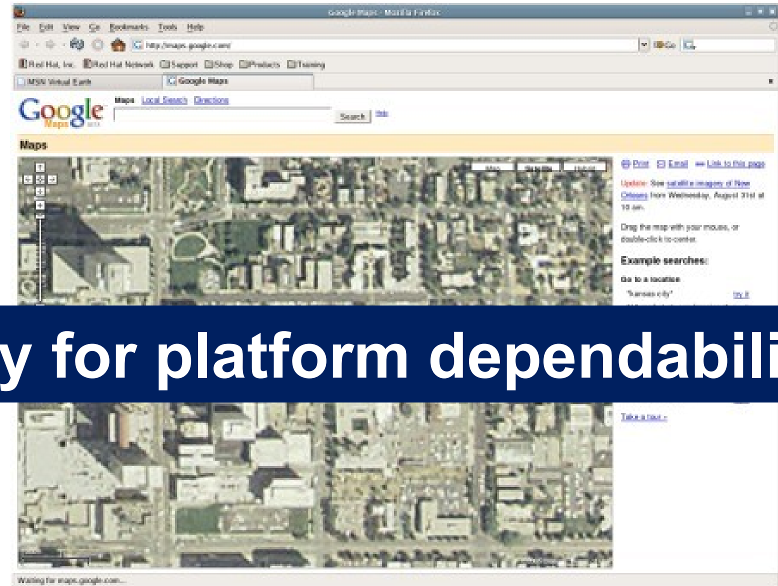
Samurai: Protecting Critical Data in Unsafe Languages

Karthik Pattabiraman, U. Illinois (Urbana Champaign)
Vinod Grover, NVIDIA Corp.
Ben Zorn, Microsoft Research

Motivation : Emerging Platforms



Fault-tolerance necessary for platform dependability

Smart-phones

Web-browsers

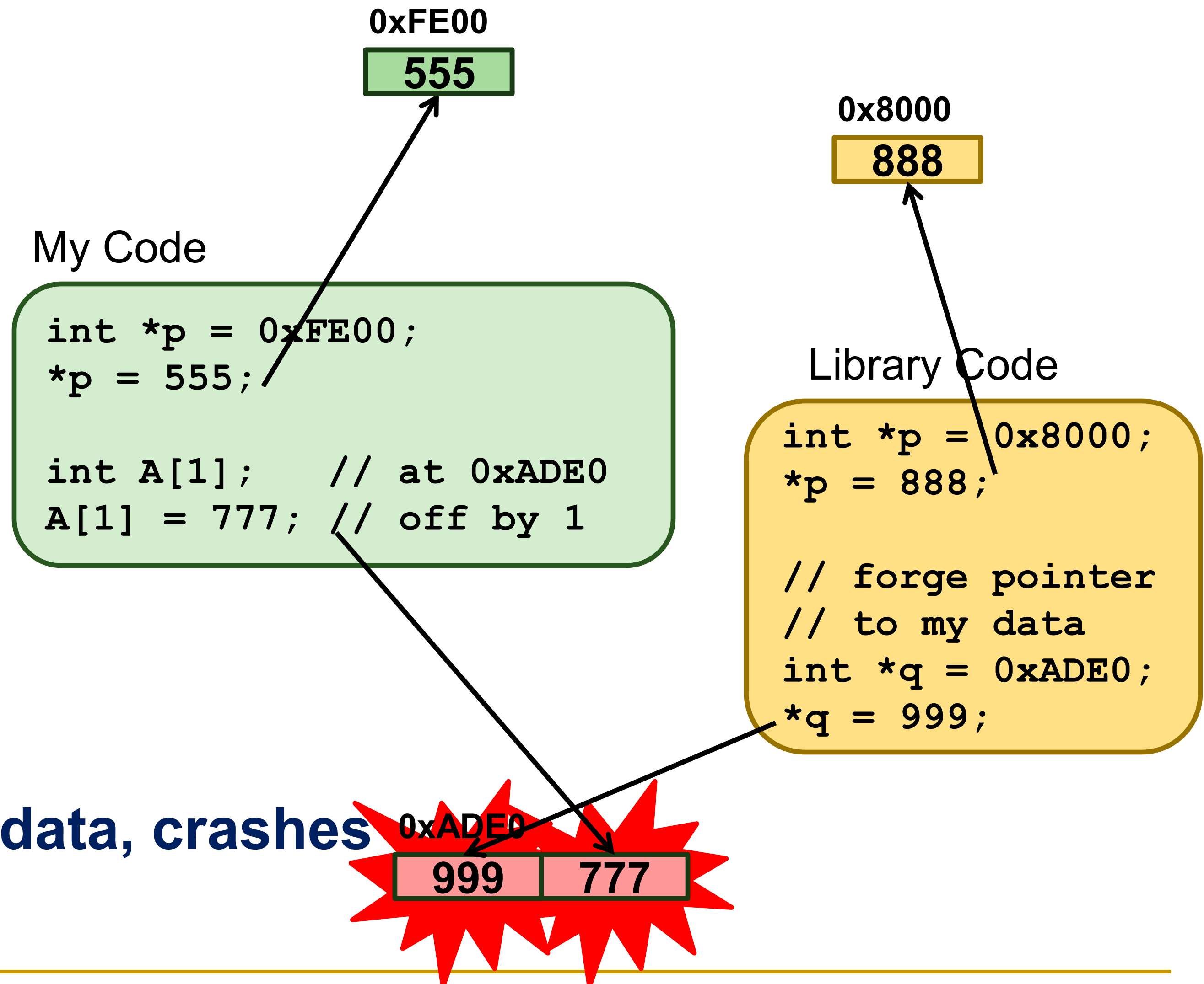
(1) Little or no isolation among components for functionality;
(2) Support for third-party components for extensibility

The Problem: A Dangerous Mix

Danger 1:
Flat, uniform address space

Danger 2:
Unsafe programming languages

Danger 3:
Unrestricted 3rd party code



Result: corrupt data, crashes security risks

Solution: Prevent or Detect Errors

- **Rewrite in a safe language (e.g. Java, C#, Cyclone)**
  - Considerable investment in programmer time
  - Performance-critical components may still be unsafe
- **Static techniques (e.g. LINT, SAL/ESP, SAFEcode)**
  - Require the entire code to be statically checked
  - May be subject to false-positives (wrong detections)
- **Dynamic techniques (CCured, CRED, etc.)**
  - All or nothing – issues with 3rd party code
  - High performance overheads

Alternative: **Tolerate Errors at Runtime**

- Failure oblivious computing [Rinard]
  - Data structure healing (unsound)
  - Boundless Memory Blocks
- Rx [Qin et al.] – checkpoint and restart
- DieHard [Berger et al.] – randomize, overprovision, replicate
- **Critical memory / Samurai**

Observations to Take Away…

- Memory is a weak abstraction
  - We can strengthen it
- Some data is more important
  - Programmers know it…
  - … and program accordingly
- No code stands alone
  - Solutions have to be compatible with / handle external code

Critical Memory

- Approach
  - Identify **critical program data**
  - Protect it with **isolation & replication**
- Goals:
  - **Harden** programs from both SW and HW errors
    - Unify existing ad hoc solutions
  - Enable **local reasoning** about memory state
    - Leverage powerful static analysis tools
  - Allow **selective, incremental hardening** of apps
  - Provide **compatibility** with existing libraries, apps

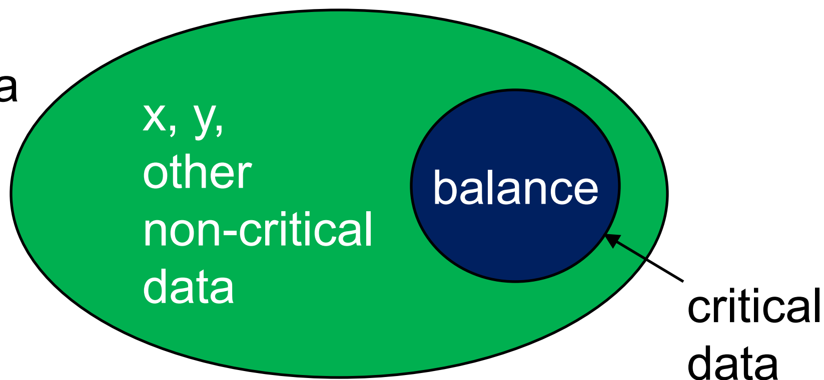
Outline

- Motivation
- Critical memory
- Samurai implementation
  - Software-only
  - Probabilistic guarantees
- Evaluation
- Conclusion

Critical Memory: Idea

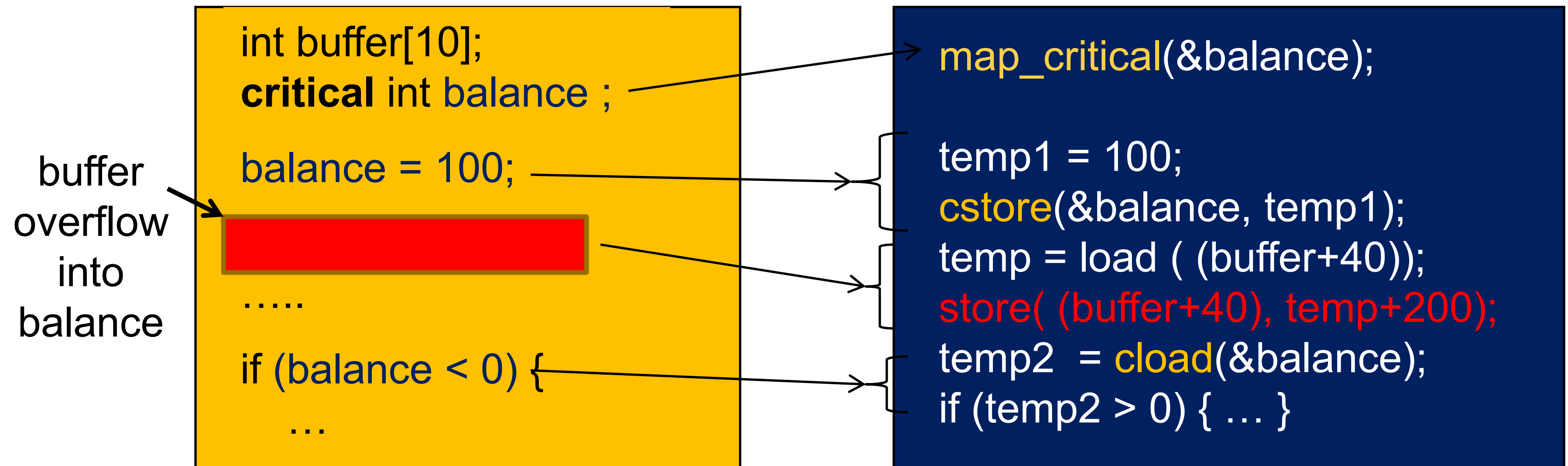
Code

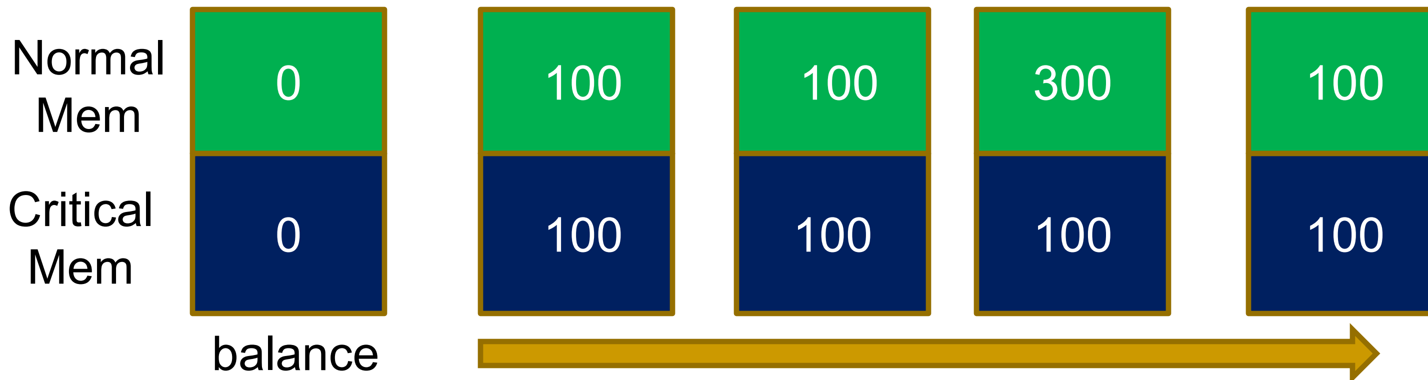
```
critical int balance;

balance += 100;
if (balance < 0) {
    chargeCredit();
} else {
    // use x, y, etc.
}
```



- Identify and mark some data as "critical
  - Type specifier like **const**
- Shadow critical data in parallel address space (critical memory)
- New operations on critical data
  - cload – read
  - cstore - write

Critical Memory: Example





Third-party Libraries/Untrusted Code

- Library code does not need to be critical memory aware
  - If library does not update critical data, no changes required
- If library needs to modify critical data
  - Allow normal stores to critical memory in library
  - Explicitly "promote" on return
- Copy-in, copy-out semantics

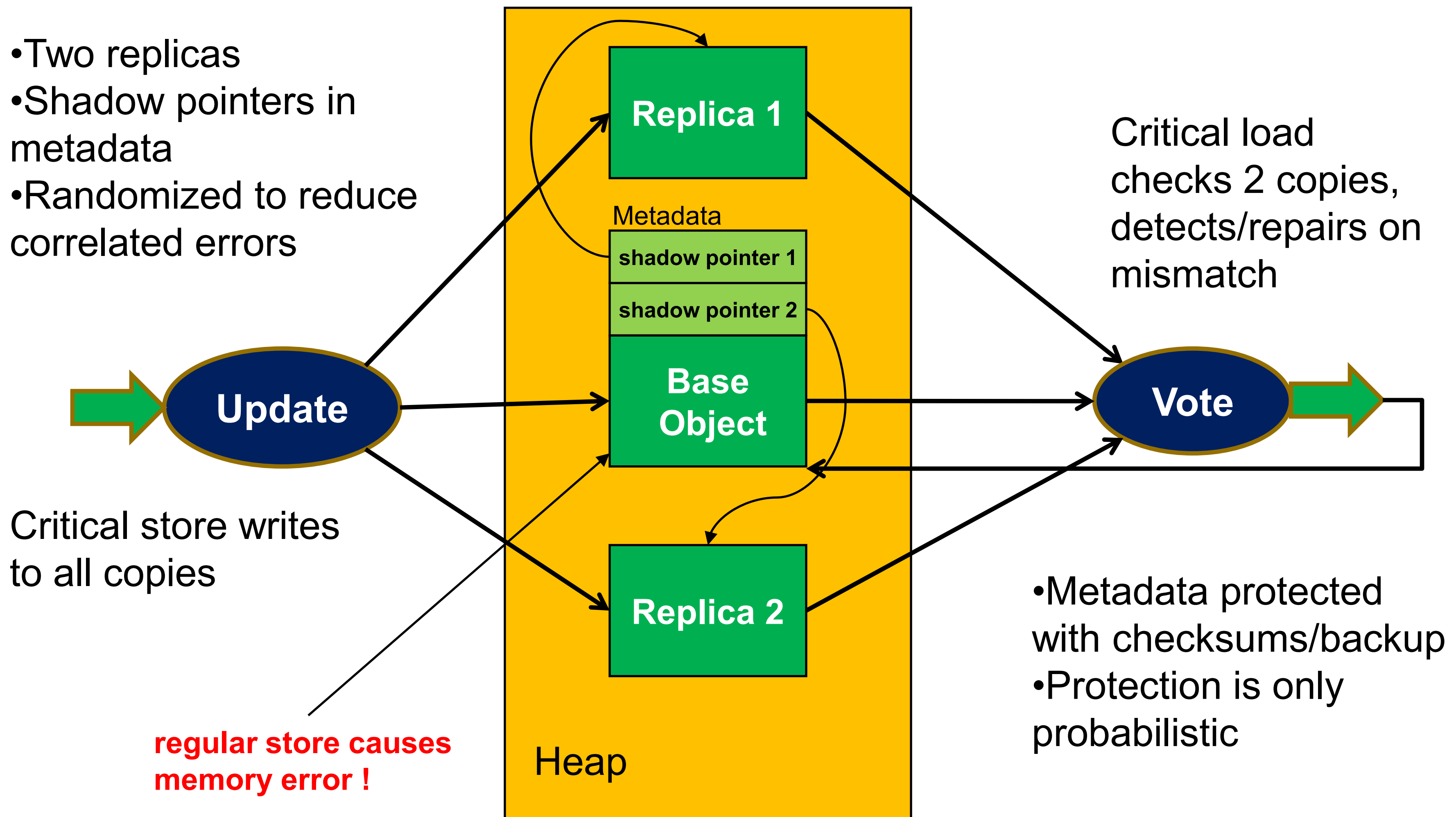
```
critical int balance = 100;
…
library_foo(&balance);
promote balance;
…
```

```
// arg is not critical int *
void library_foo(int *arg)
{
    *arg = 10000;
    return;
}
```

Samurai: SCM Prototype

- Software critical memory for heap objects
  - Critical objects allocated with crit_malloc, crit_free
- Approach
  - Replication – base copy + 2 shadow copies
  - Redundant metadata
    - Stored with base copy, copy in hash table
    - Checksum, size data for overflow detection
  - Robust allocator as foundation
    - DieHard unreplicated
    - Randomizes locations of shadow copies

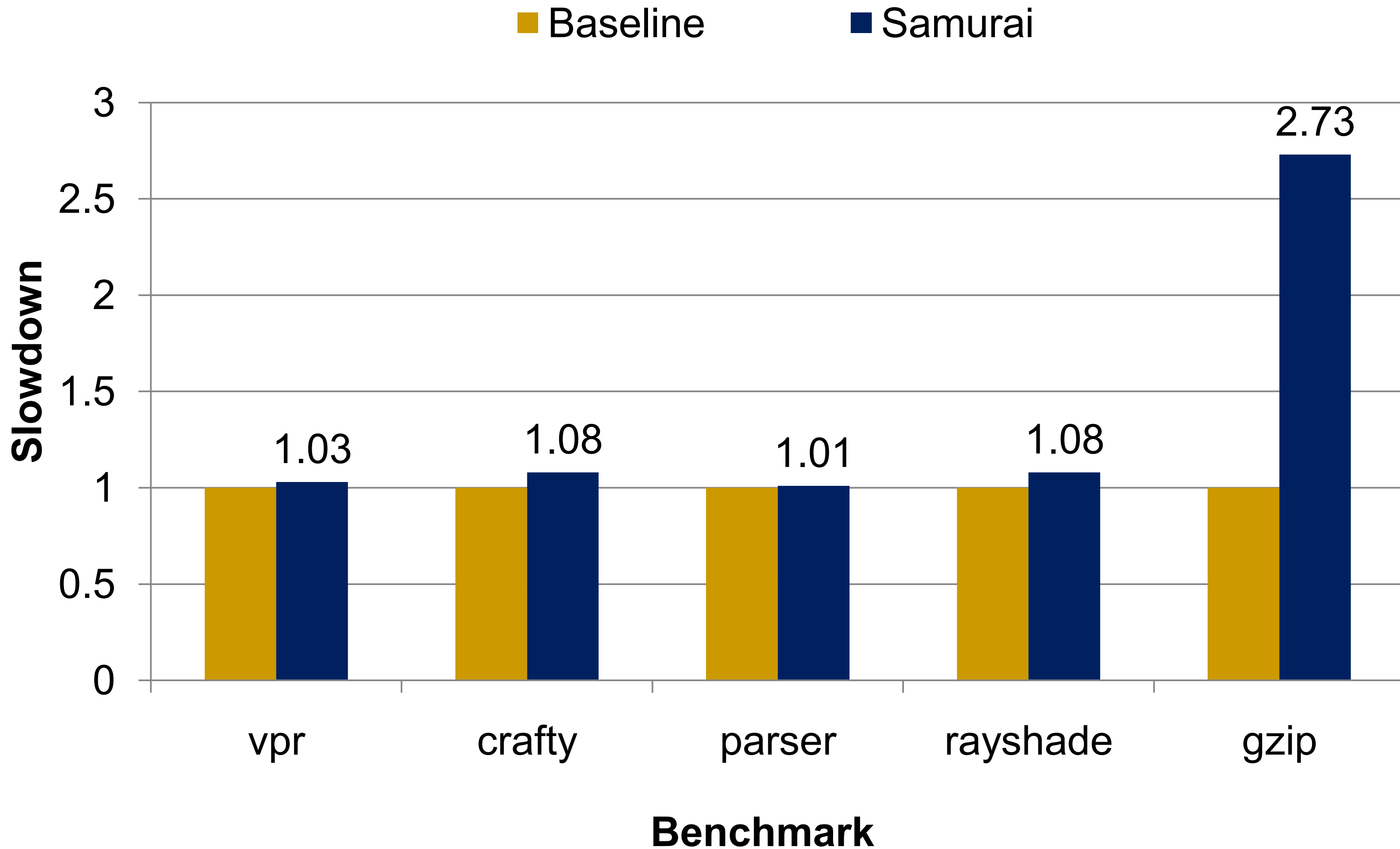
Samurai : Software Prototype



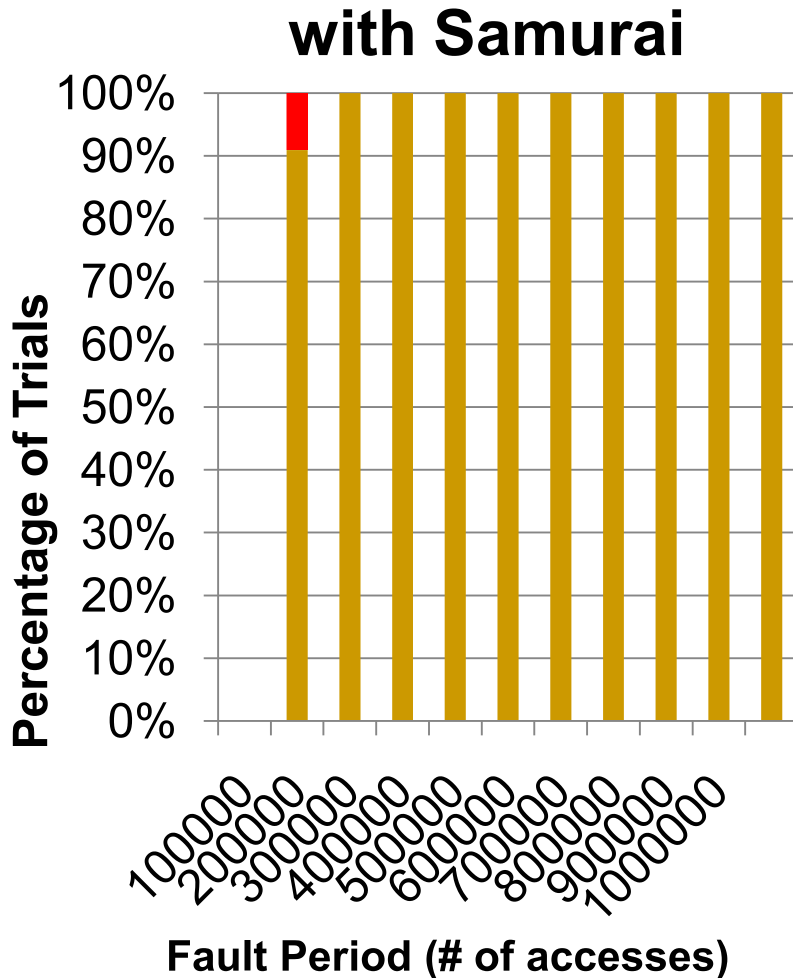
Samurai Experimental Results

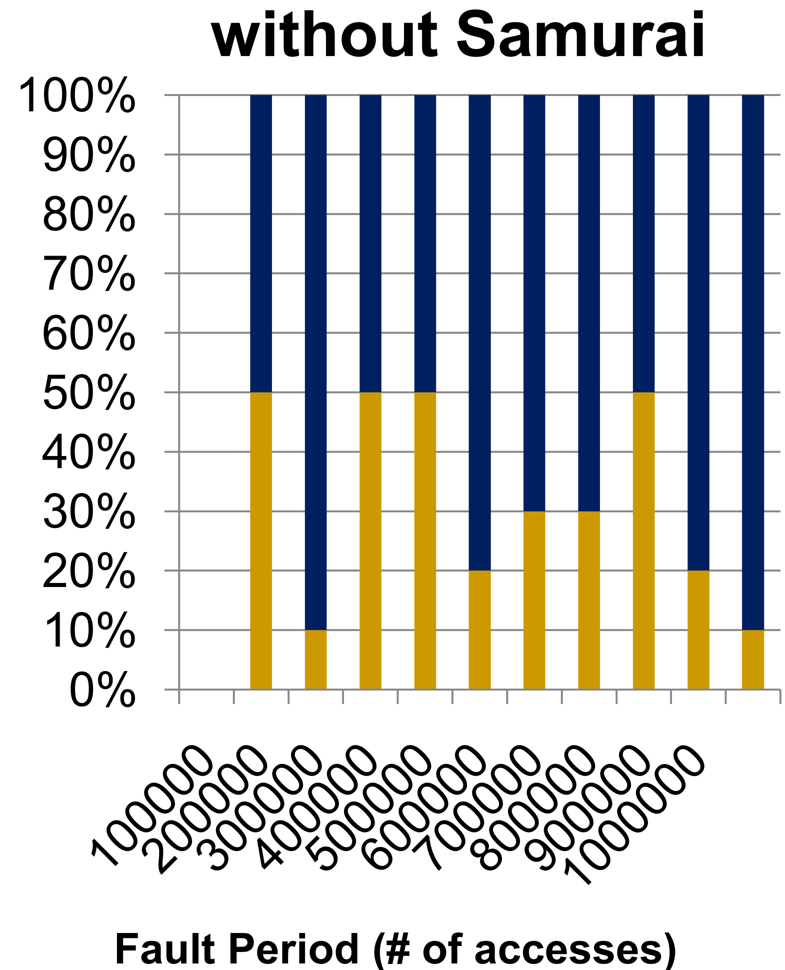
- Implementation
  - Automated Phoenix pass to instrument loads and stores
  - Runtime library for critical data allocation/de-allocation (C++)
- Protected critical data in 5 applications (mostly SPEC2k)
  - Chose data that is crucial for end-to-end correctness of program
  - Evaluation of performance overhead by instrumentation
  - Fault-injections into critical and non-critical data (for propagation)
- Protected critical data in libraries
  - **STL List Class**: Backbone of list structure (link pointers)
  - **Memory allocator**: Heap meta-data (object size + free list)

Performance Overhead



Fault Injection into Critical Memory (vpr)

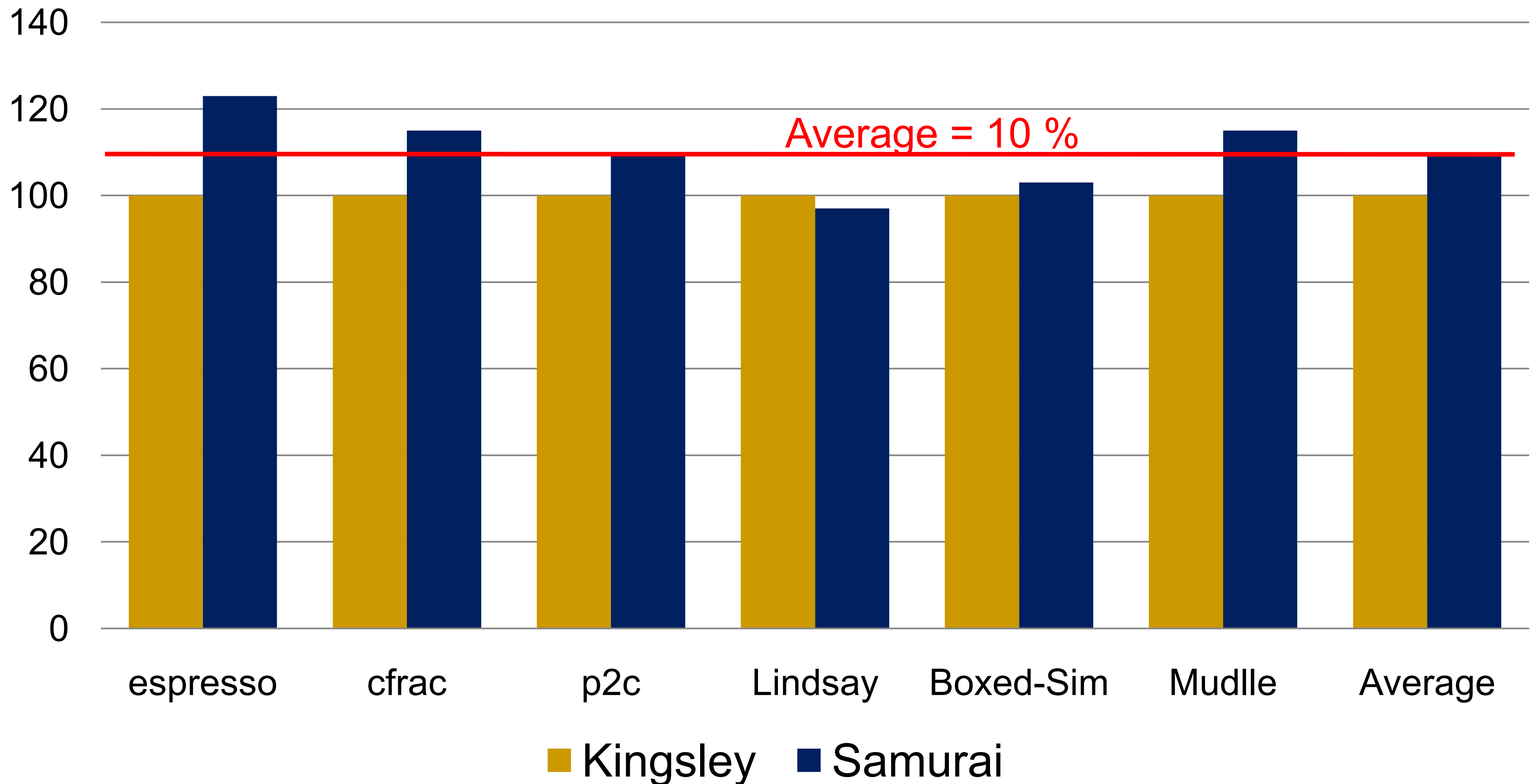




Samurai: STL Class + WebServer

- **STL List Class**
  - Modified memory allocator for class
  - Modified member functions *insert, erase*
  - Modified custom iterators for list objects
  - Added a new call-back function for direct modifications to list data
- **Webserver**
  - Used STL list class for maintaining client connection information
  - Made list critical – one thread/connection
  - Evaluated across multiple threads and connections
  - Max performance overhead = **9 %**

Samurai: Protecting Allocator Metadata



Related Work

- Robust Data Structures
  - Design [Bright et al.], [Kant et al.]
  - Specification [Dempsky et al.]
- Memory safety in unsafe languages
  - Static and dynamic methods (numerous, already mentioned)
  - Software fault isolation [Wahbe et al.]
- Recovery
  - Failure oblivious computing [Rinard et al.]
  - Rx [Qin et al.]
  - Sprite's recovery box [Baker and Sullivan]
  - SafeDrive extension recovery [Zhou et al.]

Summary

- **Critical Memory: abstract memory model**
  - Protect critical data in applications
  - Define special operations: critical load/store
  - Inter-operation with untrusted/trusted libraries
    - Compatible with existing code
    - Enables local reasoning
- **Samurai: heap-based software prototype**
  - Uses replication and forward error-correction
  - Tested on both applications and libraries
    - Performance overheads of less than 10% in many cases, including practical libraries

Questions?

Samurai: Protecting Critical Data in Unsafe Programming Languages

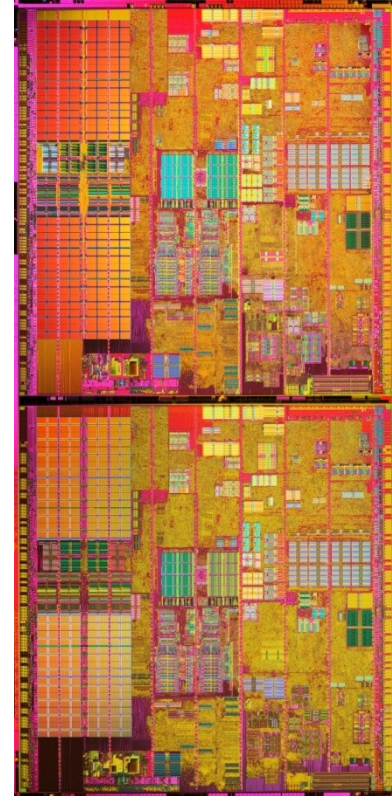
Karthik Pattabiraman, U. Illinois, U.C.
Vinod Grover, NVIDIA Corp.
Ben Zorn, Microsoft Research

Hardware Trends (1) Reliability

- Hardware transient faults are increasing
  - Even type-safe programs can be subverted in presence of HW errors
    - Academic demonstrations in Java, OCaml
  - Soft error workshop (SELSE) conclusions
    - Intel, AMD now more carefully measuring
    - "Not practical to protect everything"
    - Faults need to be handled at all levels from HW up the software stack
  - Measurement is difficult
    - How to determine soft HW error vs. software error?
    - Early measurement papers appearing

Hardware Trends (2) Multicore

- DRAM prices dropping
  - 2Gb, Dual Channel PC 6400 DDR2 800 MHz $85
- Multicore CPUs
  - **Quad-core** Intel Core 2 Quad, AMD Quad-core Opteron
  - **Eight core** Intel by 2008?
- *Challenge:*
  How should we use all this hardware?



Samurai: Applications/Critical Data

- **vpr**: Does place and route on FPGAs from netlist
  - Made routing-resource graph critical
- **crafty**: Plays a game of chess with the user
  - Made cache of previously-seen board positions critical
- **gzip**: Compress/Decompresses a file
  - Made Huffman decoding table critical
- **parser**: Checks syntactic correctness of English sentences
  - Made the dictionary data structures critical
- **rayshade**: Renders a scene file
  - Made the list of objects to be rendered critical

Samurai: Performance Measurement

- Measured baseline cost without Samurai (T_1)
- Instrumented all loads and stores in program
  - Measured cost of checking all loads, stores
    - *without Samurai* (T_2)
    - *with Samurai* (T_3)
  - Found fraction of critical loads and stores (*n*)
  - Estimated cost of checking critical loads and stores = $n * (T_2 - T_1)$
- Measured Samurai overhead as:

$$T_1 + n * (T_2 - T_1) + (T_3 - T_2)$$

Execution Characteristics

Benchmark	critical data	critical loads	critical stores
vpr	37%	0.009 %	0.000043 %
crafty	100%	0.25%	0.60 %
parser	0.2%	0.01%	0.00013 %
rayshade	0.7%	1.91%	0.000004 %
gzip	100%	12.8%	0.28 %

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)**