
Tolerating and Correcting Memory Errors in C and C++

Ben Zorn
Microsoft Research

In collaboration with:

Emery Berger and Gene Novark, Umass - Amherst

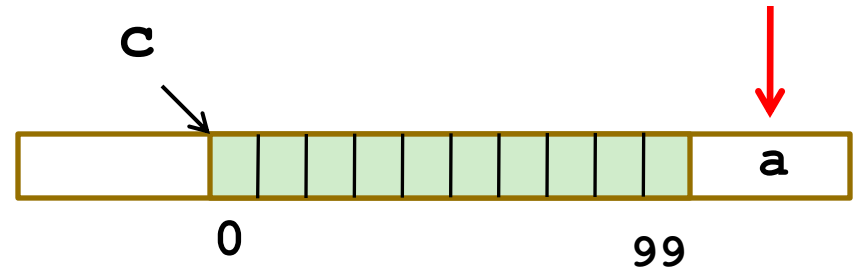
Karthik Pattabiraman, UIUC

Vinod Grover and Ted Hart, Microsoft Research

Focus on Heap Memory Errors

- Buffer overflow

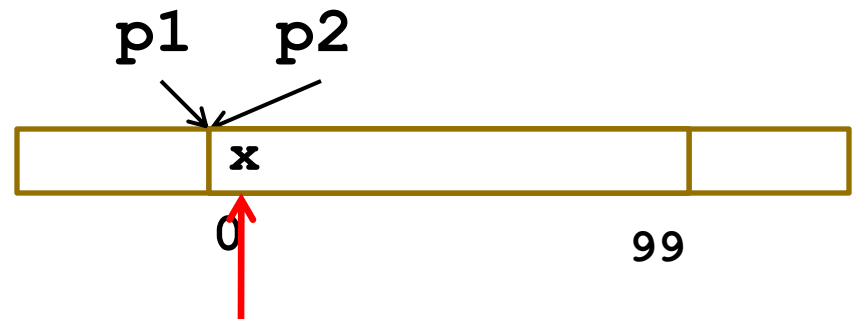
```
char *c = malloc(100);  
c[101] = 'a';
```



- Dangling reference

```
char *p1 = malloc(100);  
char *p2 = p1;
```

```
free(p1);  
p2[0] = 'x';
```



Approaches to Memory Corruptions

- Rewrite in a safe language
- Static analysis / safe subset of C or C++
 - SAFECODE [Adve], etc.
- Runtime detection, fail fast
 - Jones & Lin, CRED [Lam], CCured [Necula], etc.
 - Debugging possible, security advantage?
- Runtime toleration
 - Failure oblivious [Rinard] (unsound)
 - Rx, Boundless Memory Blocks, ECC memory
DieHard / Exterminator, Samurai

Fault Tolerance and Platforms

- Platforms necessary in computing ecosystem
 - Extensible frameworks provide lattice for 3rd parties
 - Tremendously successful business model
 - Examples numerous: Window, iPod, browser, etc.
- Platform power derives from extensibility
 - Tension between isolation for fault tolerance, integration for functionality
 - **Platform only as reliable as weakest plug-in**
 - Tolerating bad plug-ins necessary by design

Research Vision

- Increase robustness of installed code base
 - Potentially improve millions of lines of code
 - Minimize effort – ideally no source mods, no recompilation
- Reduce requirement to patch
 - Patches are expensive (detect, write, deploy)
 - Patches may introduce new errors
- Enable trading resources for robustness
 - E.g., more memory implies higher reliability

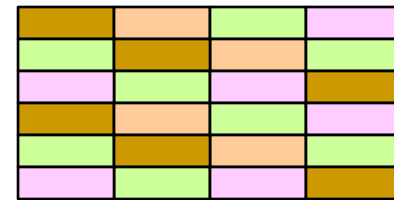
Outline

- Motivation
- Exterminator
 - Collaboration with Emery Berger, Gene Novark
 - Automatically corrects memory errors
 - Suitable for large scale deployment
- Critical Memory / Samurai
 - Collaboration with Karthik Pattabiraman, Vinod Grover
 - New memory semantics
 - Source changes to explicitly identify and protect critical data
- Conclusion

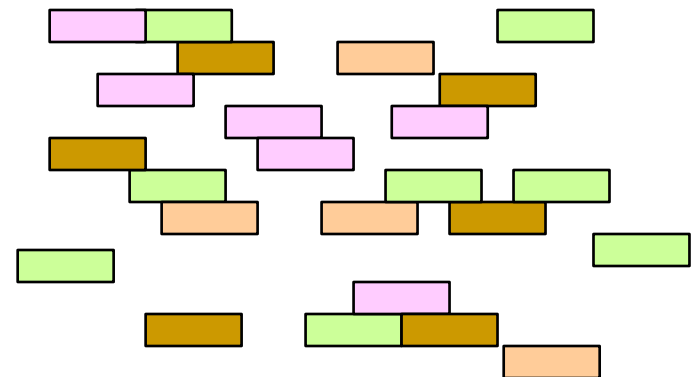
DieHard Allocator in a Nutshell

- Joint work with Emery Berger
- Existing heaps are packed tightly to minimize space
 - Tight packing increases likelihood of corruption
 - Predictable layout is easier for attacker to exploit
- We randomize and overprovision the heap
 - Expansion factor determines how much empty space
 - Semantics are identical
- Replication increases benefits
- Enables analytic reasoning

Normal Heap



DieHard Heap



DieHard in Practice

- **DieHard (non-replicated)**
 - Windows, Linux version implemented by Emery Berger
 - Try it right now! (<http://www.diehard-software.org/>)
 - Adaptive, automatically sizes heap
 - Mechanism automatically redirects malloc calls to DieHard DLL
- **Application: Firefox & Mozilla**
 - Known buffer overflow in version 1.7.3 crashes browser
- **Experience**
 - Usable in practice – no perceived slowdown
 - Roughly doubles memory consumption
 - 20.3 Mbytes vs. 44.3 Mbytes with DieHard

Caveats

- Primary focus is on protecting heap
 - Techniques applicable to stack data, but requires recompilation and format changes
- DieHard trades space, extra processors for memory safety
 - Not applicable to applications with large footprint
 - Applicability to server apps likely to increase
- DieHard requires non-deterministic behavior to be made deterministic (on input, `gettimeofday()`, etc.)
- DieHard is a brute force approach
 - Improvements possible (efficiency, safety, coverage, etc.)

Exterminator Motivation

- **DieHard limitations**
 - Tolerates errors probabilistically, doesn't fix them
 - Memory and CPU overhead
 - Provides no information about source of errors
- **“Ideal” solution addresses the limitations**
 - Program automatically detects and fixes memory errors
 - Corrected program has no memory, CPU overhead
 - Sources of errors are pinpointed, easier for human to fix
- **Exterminator = correcting allocator**
 - Joint work with Emery Berger, Gene Novark
 - Random allocation => isolates bugs while tolerating them

Exterminator Components

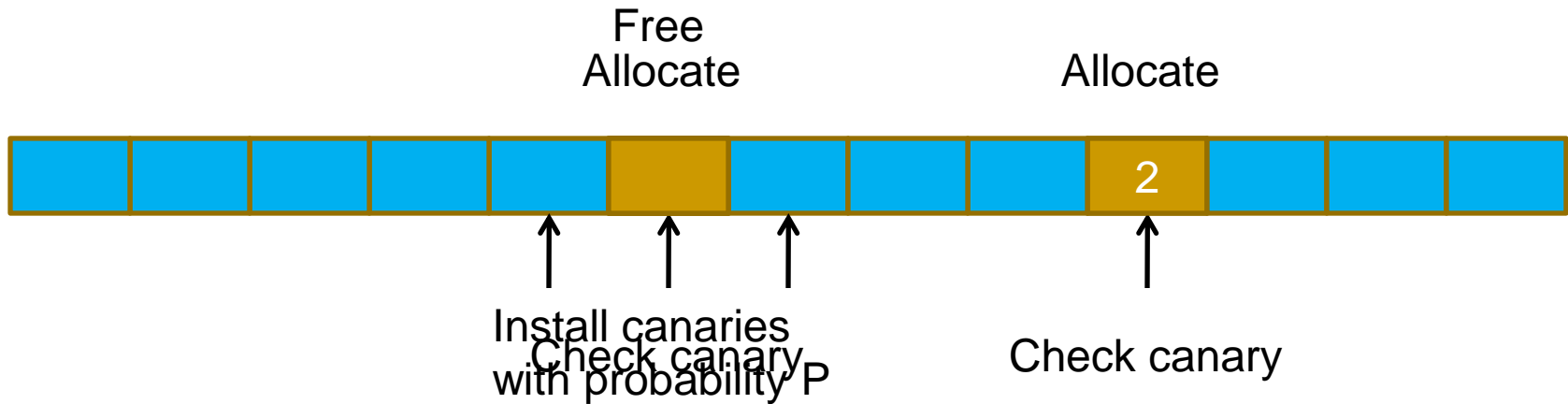
- Architecture of Exterminator dictated by solving specific problems
- How to detect heap corruptions effectively?
 - DieFast allocator
- How to isolate the cause of a heap corruption precisely?
 - Heap differencing algorithms
- How to automatically fix buggy C code without breaking it?
 - Correcting allocator + hot allocator patches

DieFast Allocator

- Randomized, over-provisioned heap
 - Canary = random bit pattern fixed at startup `100101011110`
 - Leverage extra free space by inserting canaries
- Inserting canaries
 - Initialization – all cells have canaries
 - On allocation – no new canaries
 - On free – put canary in the freed object with prob. P
 - Remember where canaries are (bitmap)
- Checking canaries
 - On allocation – check cell returned
 - On free – check adjacent cells

Installing and Checking Canaries

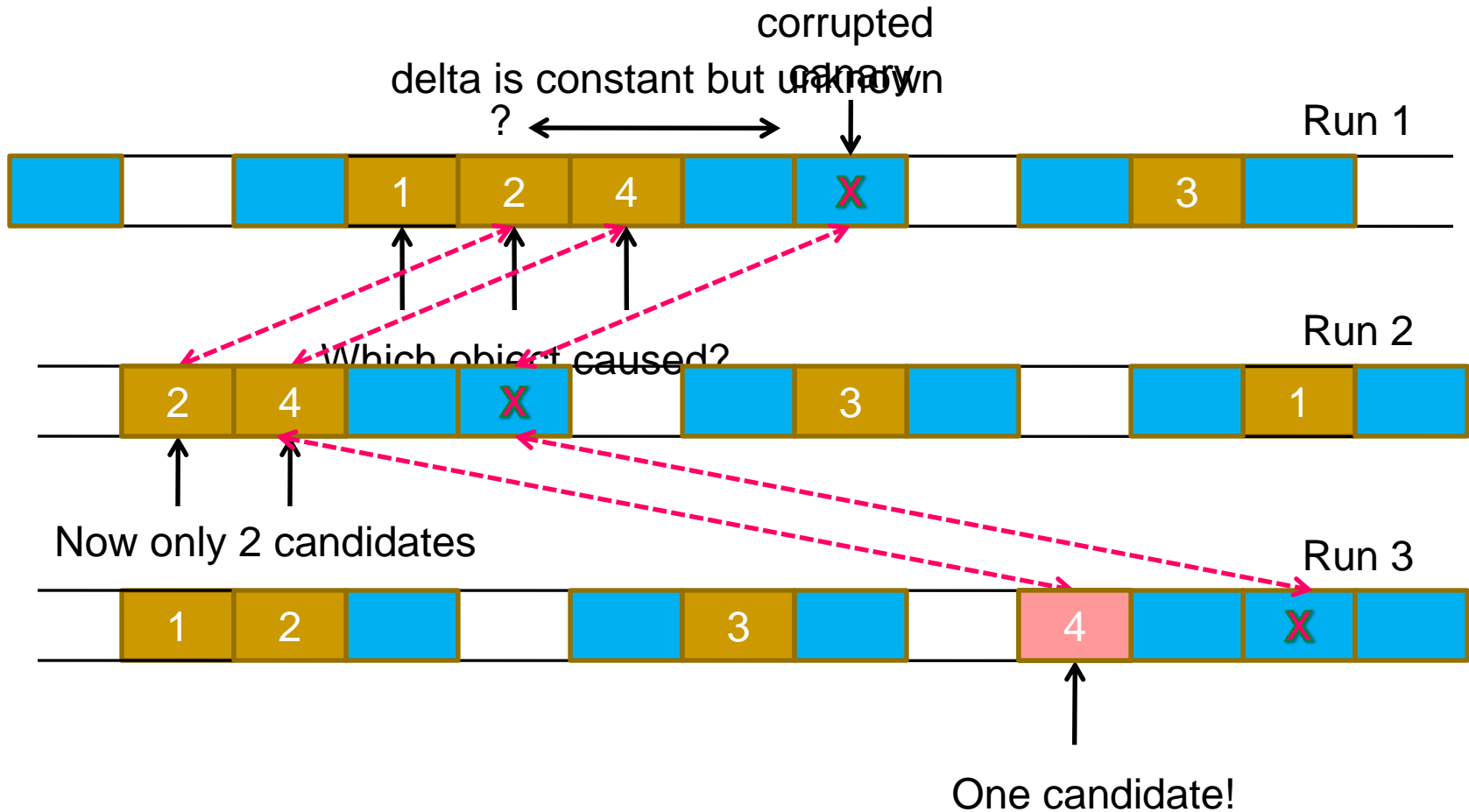
Initially, heap full of canaries



Heap Differencing

- Strategy
 - Run program multiple times with different randomized heaps
 - If detect canary corruption, dump contents of heap
 - Identify objects across runs using allocation order
- Key insight: Relation between corruption and object causing corruption is invariant across heaps
 - Detect invariant across random heaps
 - More heaps => higher confidence of invariant

Attributing Buffer Overflows



Precision increases exponentially with number of runs

Detecting Dangling Pointers (2 cases)

- Dangling pointer read/written (easy)
 - Invariant = canary in freed object X has same corruption in all runs
- Dangling pointer only read (harder)
 - Sketch of approach (paper explains details)
 - Only fill freed object X with canary with probability P
 - Requires multiple trials: $\approx \log_2(\text{number of callsites})$
 - Look for correlations, i.e., X filled with canary => crash
 - Establish conditional probabilities
 - Have: $P(\text{callsite X filled with canary} \mid \text{program crashes})$
 - Need: $P(\text{crash} \mid \text{filled with canary})$, guess “prior” to compute

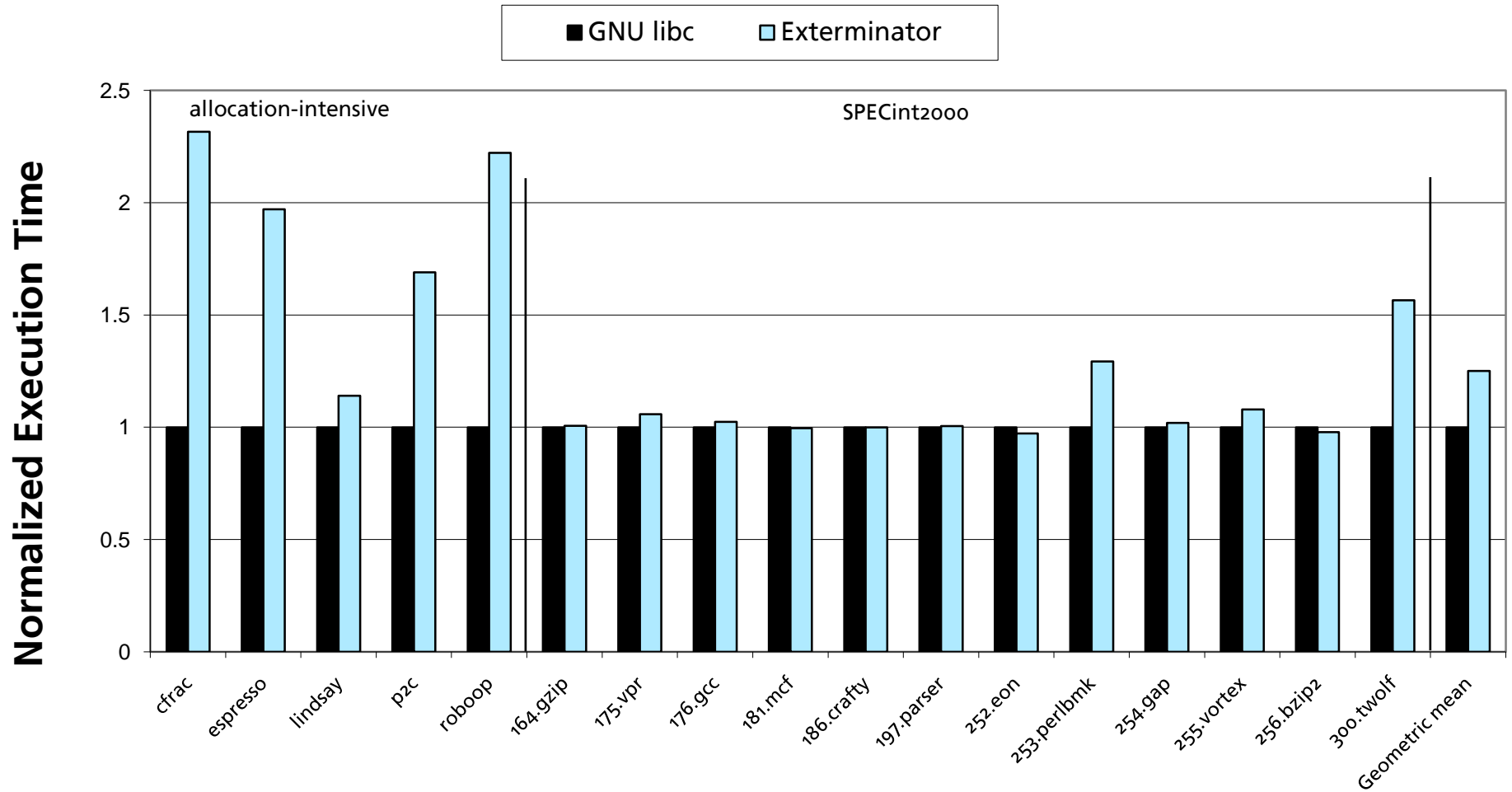
Correcting Allocator

- Group objects by allocation site
- Patch object groups at allocate/free time
- Associate patches with group
 - Buffer overrun => add padding to size request
 - `malloc(32)` becomes `malloc(32 + delta)`
 - Dangling pointer => defer free
 - `free(p)` becomes `defer_free(p, delta_allocations)`
 - Fixes preserve semantics, no new bugs created
- **Correcting allocation may != DieFast or DieHard**
 - Correction allocator can be space, CPU efficient
 - “Patches” created separately, installed on-the-fly

Deploying Exterminator

- Exterminator can be deployed in different modes
- Iterative – suitable for test environment
 - Different random heaps, identical inputs
 - Complements automatic methods that cause crashes
- Replicated mode
 - Suitable in a multi/many core environment
 - Like DieHard replication, except auto-corrects, hot patches
- Cumulative mode – partial or complete deployment
 - Aggregates results across different inputs
 - Enables automatic root cause analysis from Watson dumps
 - Suitable for wide deployment, perfect for beta release
 - Likely to catch many bugs not seen in testing lab

DieFast Overhead



Exterminator Effectiveness

- Squid web cache buffer overflow
 - Crashes glibc 2.8.0 malloc
 - 3 runs sufficient to isolate 6-byte overflow
- Mozilla 1.7.3 buffer overflow (recall demo)
 - Testing scenario - repeated load of buggy page
 - 23 runs to isolate overflow
 - Deployed scenario – bug happens in middle of different browsing sessions
 - 34 runs to isolate overflow

Outline

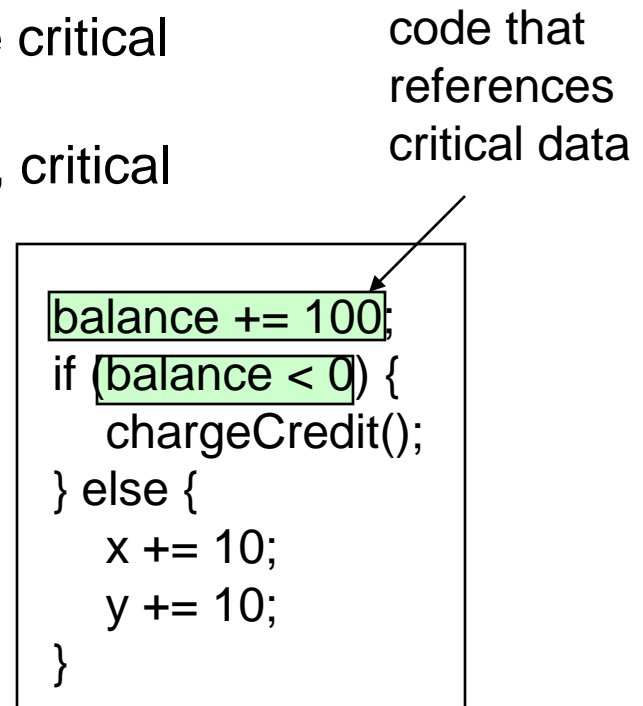
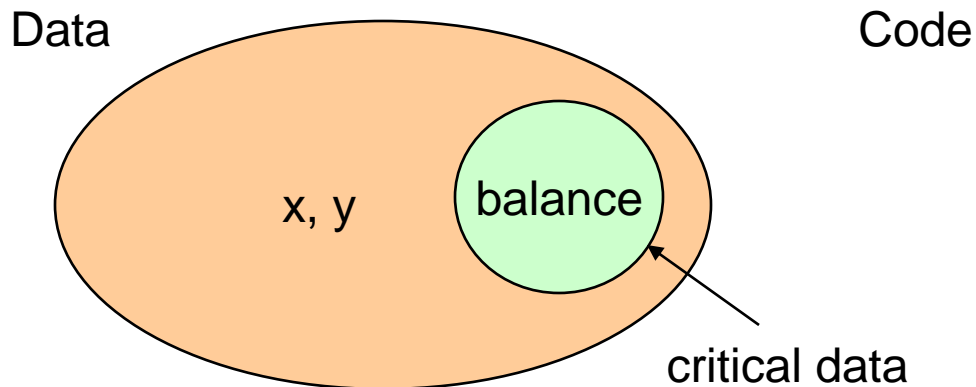
- Motivation
- Exterminator
 - Collaboration with Emery Berger, Gene Novark
 - Automatically corrects memory errors
 - Suitable for large scale deployment
- **Critical Memory / Samurai**
 - Collaboration with Karthik Pattabiraman, Vinod Grover
 - New memory semantics
 - Source changes to explicitly identify and protect critical data
- Conclusion

Critical Memory Motivation

- C/C++ programs vulnerable to memory errors
 - Software errors: buffer overflows, etc.
 - Hardware transient errors: bit flips, etc.
 - Increasingly a problem due to process shrinking, power
- Critical memory goals:
 - Harden programs from both SW and HW errors
 - Allow **local reasoning** about memory state
 - Allow **selective, incremental hardening** of apps
 - Provide **compatibility** with existing libraries, apps

Main Idea: Data-centric Robustness

- Critical memory
 - Some data is more important than other data
 - Selectively protect that data from corruption
- Examples
 - Account data, document contents are critical
// UI data is not
 - Game score information, player stats, critical
// rendering data structures are not



Critical Memory Semantics

- Conceptually, critical memory is parallel and independent of normal memory
- Critical memory requires special allocate/deallocate and read/write operations
 - `critical_store` (`cstore`) – only way to consistently update critical memory
 - `critical_load` (`cload`) – only way to consistently read critical memory
- Critical load/store have priority over normal load/store
- Normal loads still see the value of critical memory

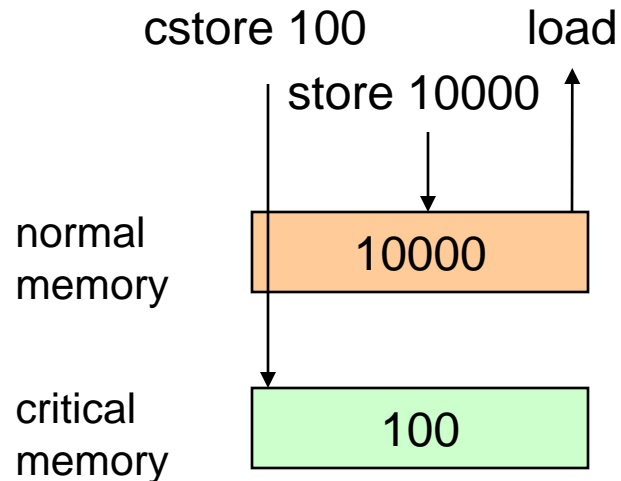
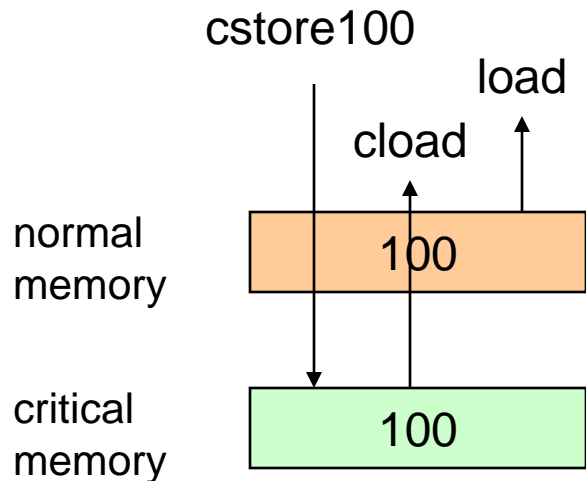
Example

cstore balance, 100
...
load balance returns 100
load balance returns 100

cstore balance, 100
store balance, 10000
(applications should not do this)

...
load balance returns 10000
(compatibility semantics)

load balance returns 100
(possibly triggers exception)



Critical Memory Deployment

- Define “critical” type specifier:
 - Like “const”
 - Easy to use, minimal source mods
- Allows local reasoning
 - External libraries, code cannot modify critical data
- Tolerates memory errors
 - Non-critical overflows cannot corrupt critical values
- Allows static analysis of program subset
 - Critical subset of program can be statically checked independently
- Additional checking on critical data possible

```
int x, y, buffer[10];
critical int balance = 100;

third_party_lib(&x, &y);
buffer[10] = 10000; // Bug!

// balance still == 100

if (balance < 0) {
    chargeCredit();
} else {
    x += 10;
    y += 10;
}
```

Third-party Libraries/Untrusted Code

- Library code does not need to be critical memory aware
 - If library does not modify 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;  
...
```

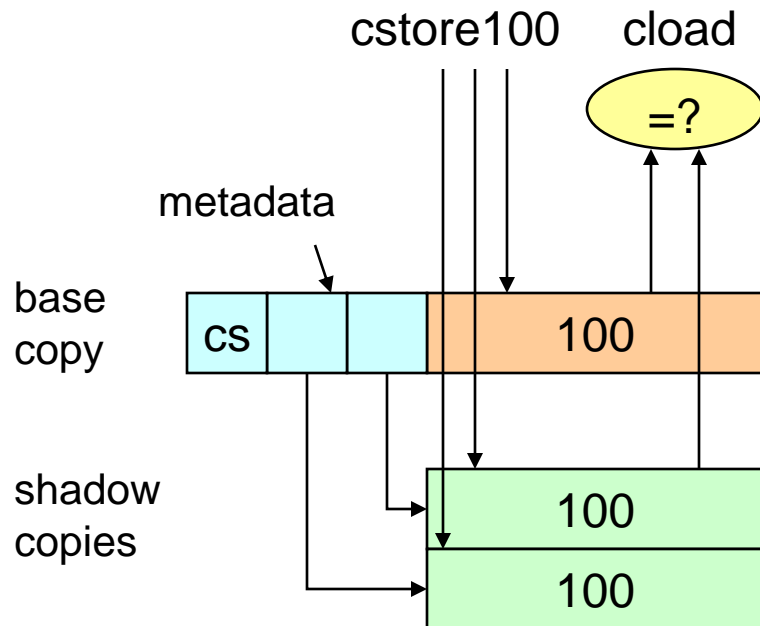
```
// arg is not critical int *  
void library_foo(int *arg)  
{  
    *arg = 10000;  
    return;  
}
```

Samurai: SCM Prototype

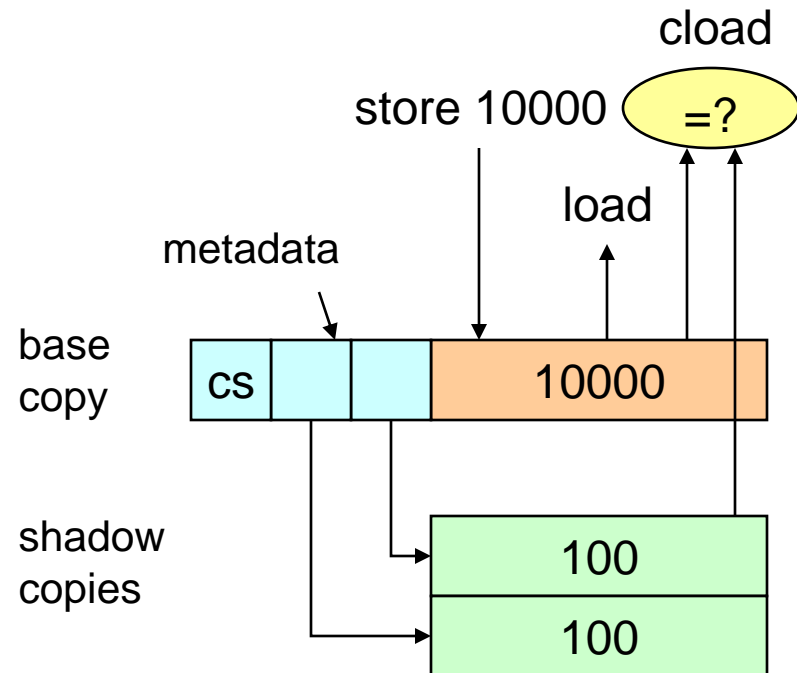
- Software critical memory for heap objects
 - Critical objects allocated with `crit_malloc`, `crit_free`
- Approach
 - Replication – base copy + 2 shadow copies
 - Redundant metadata
 - Stored with base copy, copy in hash table
 - Checksum, size data for overflow detection
 - Robust allocator as foundation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

Samurai Implementation

cstore balance, 100
...
cloud balance returns 100
load balance returns 100



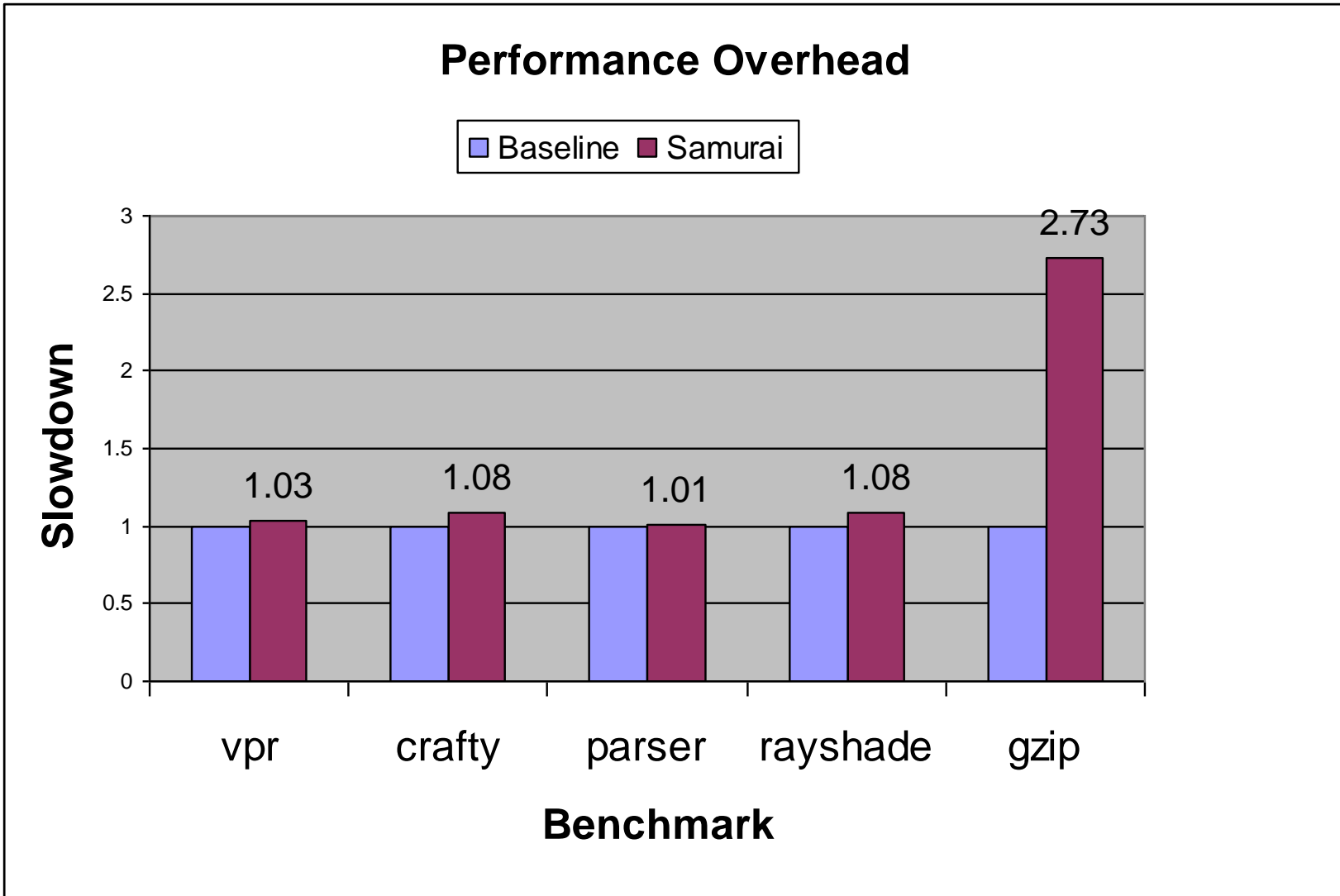
cstore balance, 100
store balance, 10000...
load balance returns 10000
cloud balance returns 100



Samurai Experimental Results

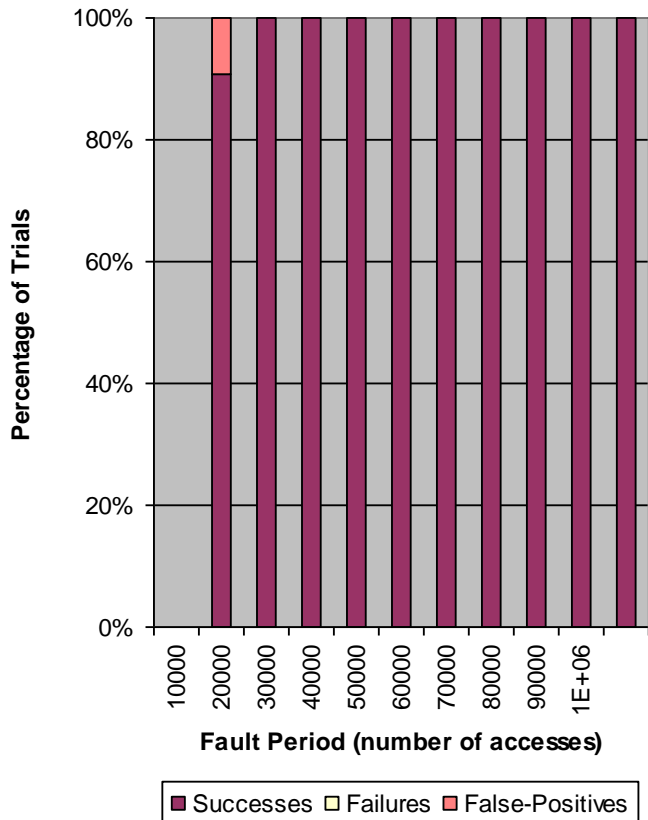
- Prototype implementation of critical memory
 - Fault-tolerant runtime system for C/C++
 - Applied to heap objects
 - Automated Phoenix compiler pass
- Identified critical data for five SPECint applications
 - Low overheads for most applications (less than 10%)
- Conducted fault-injection experiments
 - Fault tolerance significantly improved over based code
 - Low probability of fault-propagation from non-critical data to critical data for most applications
 - No new assertions or consistency checks added

Performance Results

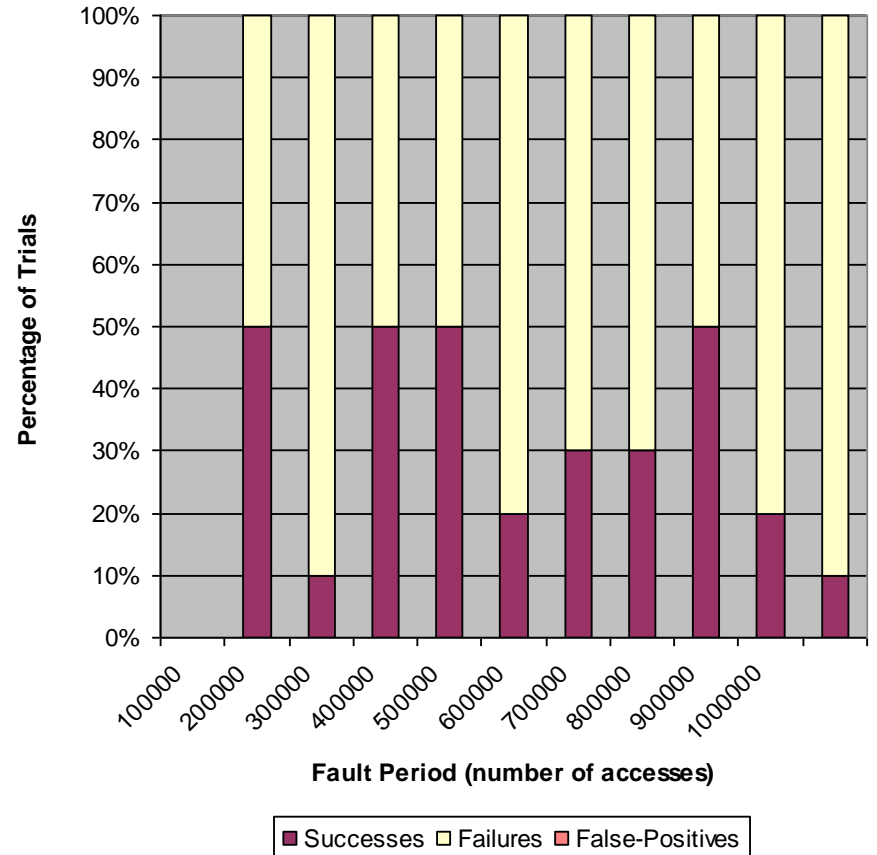


Fault Injection into Critical Data (vpr)

Fault Injections into vpr (with Samurai)



Fault Injections into vpr (without Samurai)



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) |

Experience with CM for Components

- Hardened two libraries with critical memory
 - CM-STL – hardened List Class data and metadata
 - CM-malloc – hardened allocator metadata
- Effort
 - Relatively small changes to existing code
- Performance impact
 - CM-STL in Web Server thread list => 10% slower
 - CM-malloc in cfrac benchmark => 22% slower

Conclusion

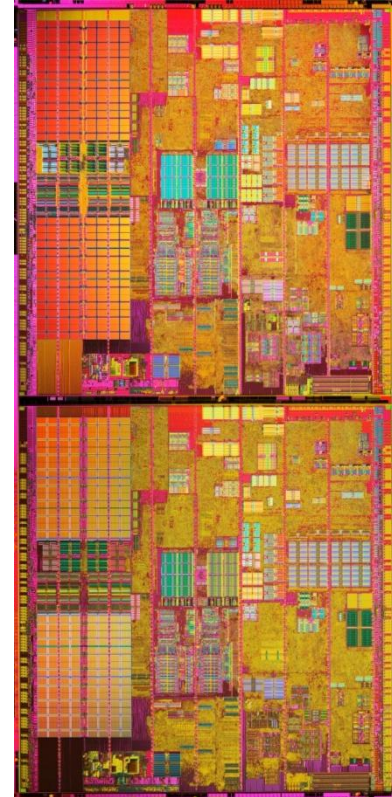
- Programs written in C / C++ can execute safely and correctly despite memory errors
- Research vision
 - Improve existing code without source modifications
 - Reduce human generated patches required
 - Increase reliability, security by order of magnitude
- Current projects
 - **DieHard / Exterminator**: automatically detect and correct memory errors (with high probability)
 - **Critical Memory / Samurai**: enable local reasoning, allow selective hardening, compatibility
 - **ToleRace**: replication to hide data races

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?



Additional Information

■ Web sites:

- Ben Zorn: <http://research.microsoft.com/~zorn>
- DieHard: <http://www.diehard-software.org/>
- Exterminator: <http://www.cs.umass.edu/~gnovark/>

■ Publications

- Emery D. Berger and Benjamin G. Zorn, "**DieHard: Probabilistic Memory Safety for Unsafe Languages**", *PLDI'06*.
- Karthik Pattabiraman, Vinod Grover, and Benjamin G. Zorn, "**Samurai - Protecting Critical Data in Unsafe Languages**", Microsoft Research, MSR-TR-2006-127, September 2006.
- Gene Novark, Emery D. Berger and Benjamin G. Zorn, "**Exterminator: Correcting Memory Errors with High Probability**", *PLDI'07*.

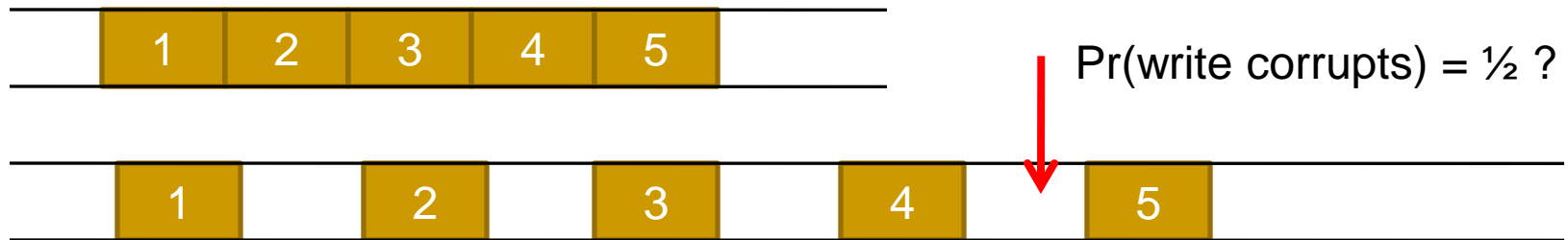
Backup Slides

DieHard: Probabilistic Memory Safety

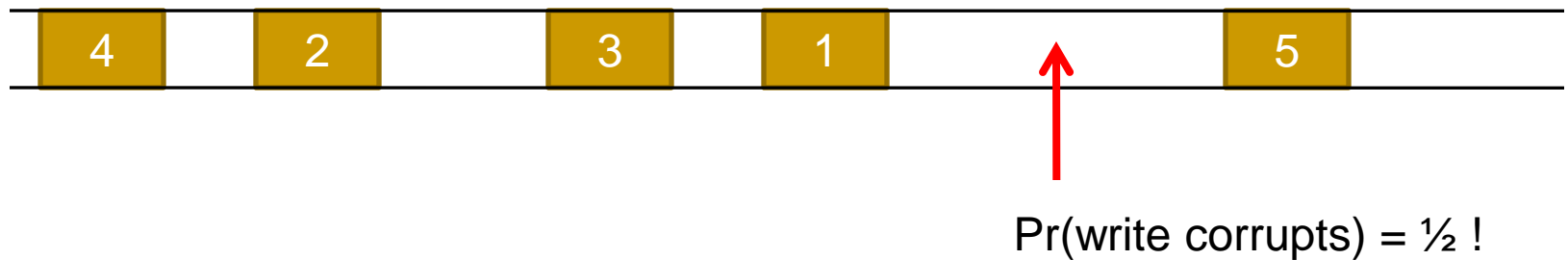
- Collaboration with Emery Berger
- Plug-compatible replacement for malloc/free in C lib
- We define “infinite heap semantics”
 - Programs execute as if each object allocated with unbounded memory
 - All frees ignored
- Approximating infinite heaps – 3 key ideas
 - Overprovisioning
 - Randomization
 - Replication
- Allows analytic reasoning about safety

Overprovisioning, Randomization

Expand size requests by a factor of M (e.g., $M=2$)

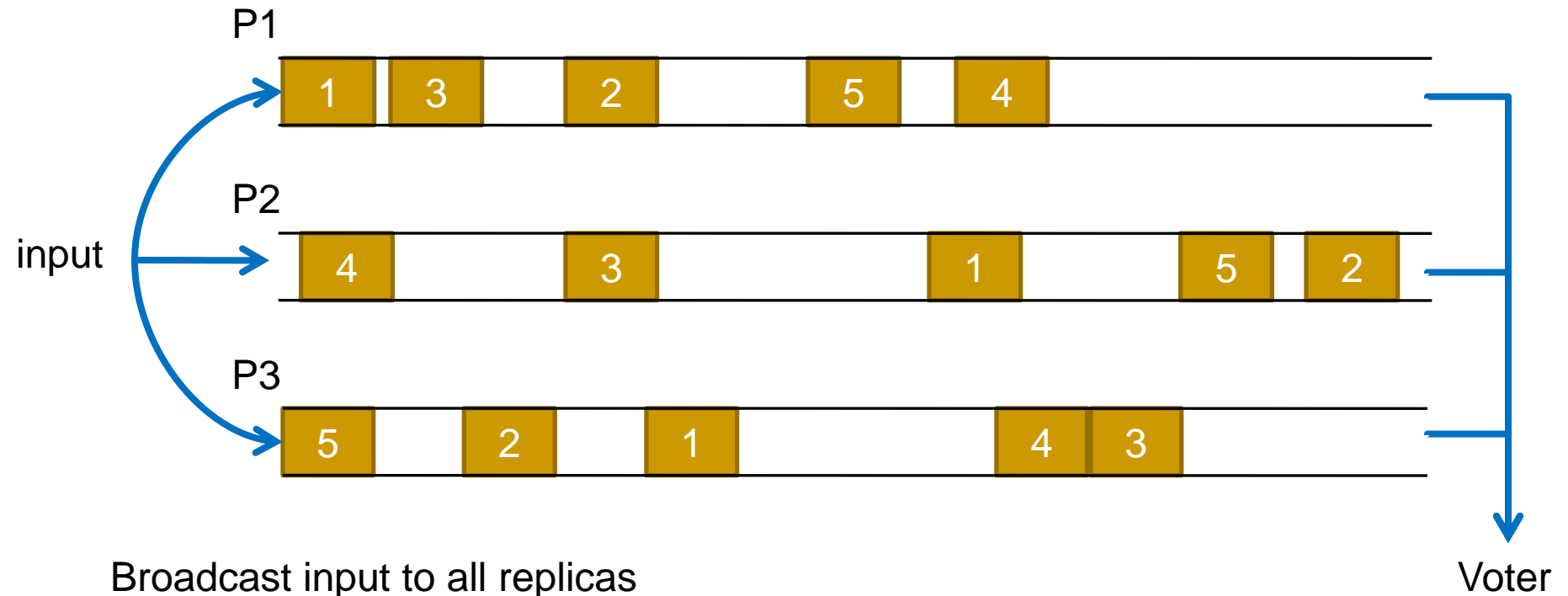


Randomize object placement



Replication (optional)

Replicate process with different randomization seeds

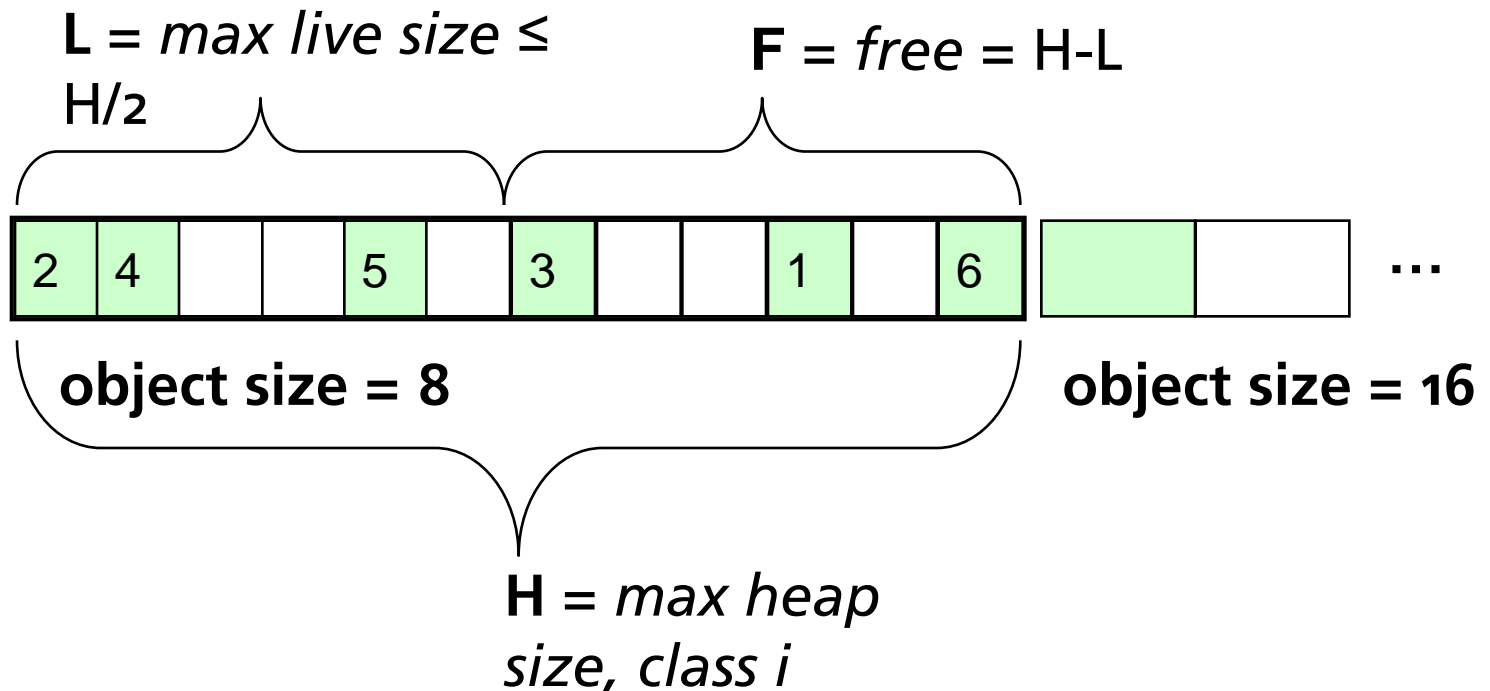


DieHard Implementation Details

- Multiply allocated memory by factor of M
- Allocation
 - Segregate objects by size (\log_2), bitmap allocator
 - Within size class, place objects randomly in address space
 - Randomly re-probe if conflicts (expansion limits probing)
 - Separate metadata from user data
 - Fill objects with random values – for detecting uninit reads
- Deallocation
 - Expansion factor => frees deferred
 - Extra checks for illegal free

Over-provisioned, Randomized Heap

Segregated size classes



- Static strategy pre-allocates size classes
- Adaptive strategy grows each size class incrementally

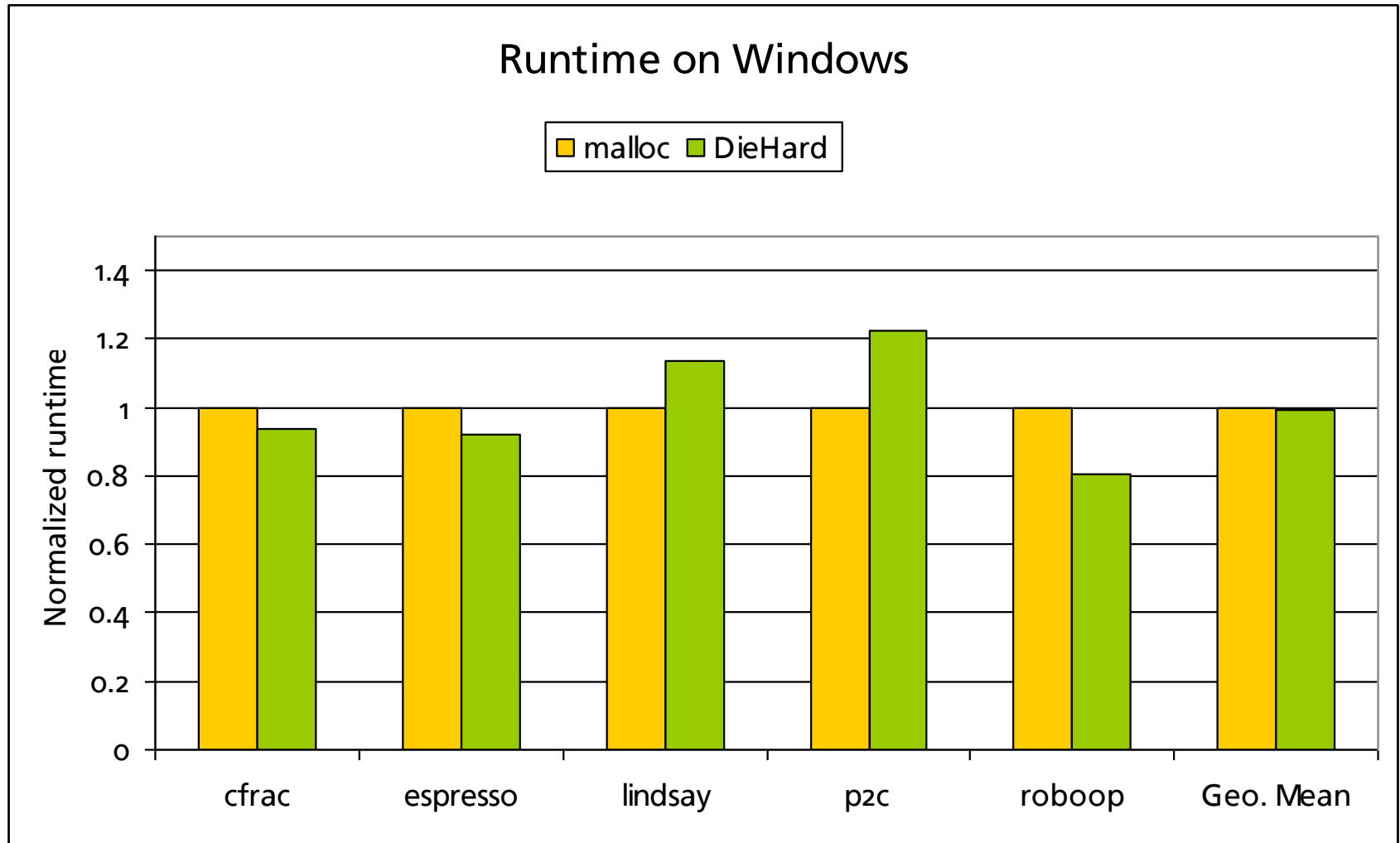
Randomness enables Analytic Reasoning

Example: Buffer Overflows

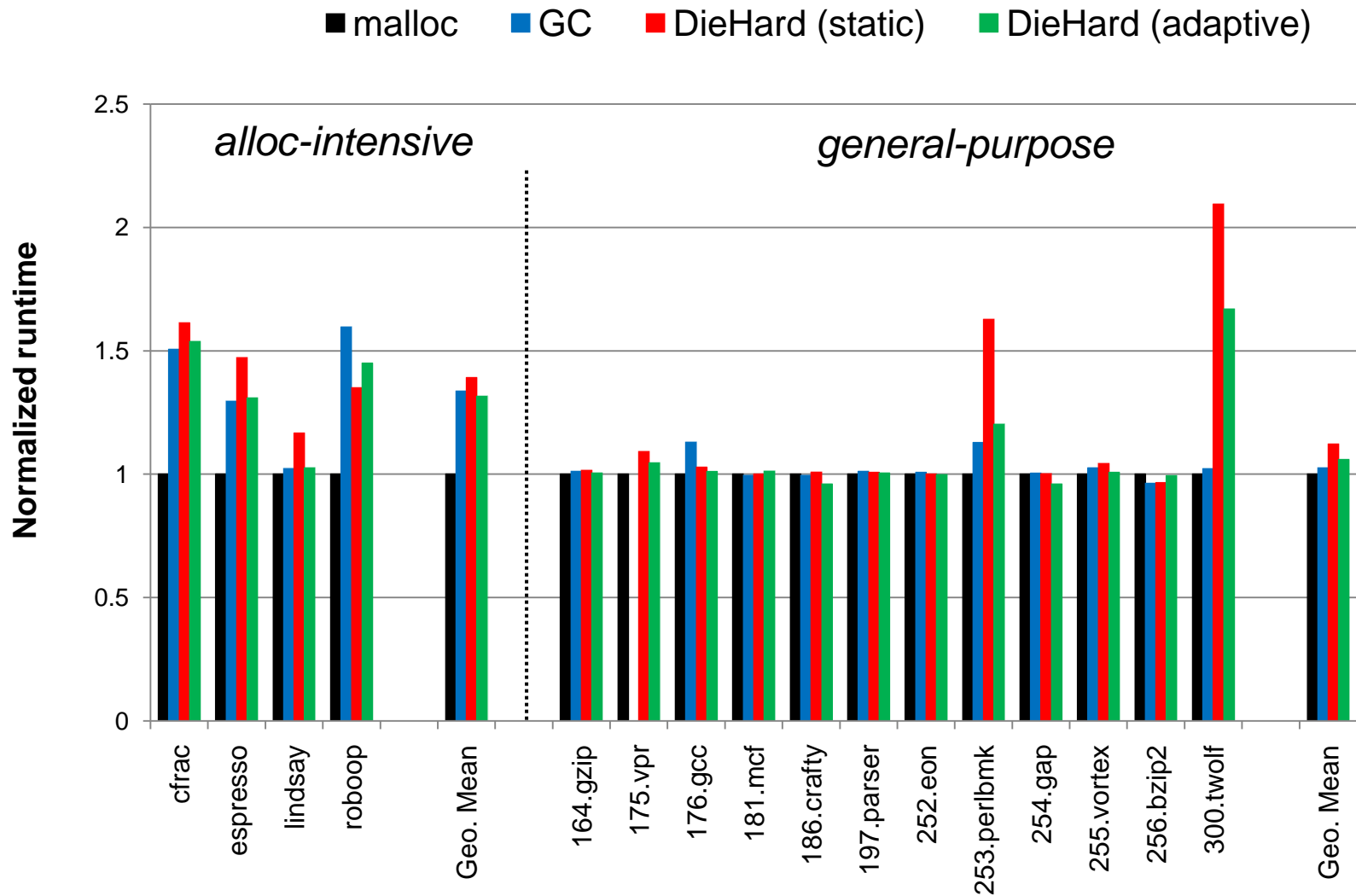
$$\Pr(\text{Mask Buffer Overflow}) = 1 - \left[1 - \left(\frac{F}{H} \right)^{\text{Obj}} \right]^k$$

- $k = \#$ of replicas, $\text{Obj} =$ size of overflow
- With no replication, $\text{Obj} = 1$, heap no more than 1/8 full:
 $\Pr(\text{Mask buffer overflow}), = 87.5\%$
- 3 replicas: $\Pr(\text{ibid}) = 99.8\%$

DieHard CPU Performance (no replication)



DieHard CPU Performance (Linux)



Correctness Results

- Tolerates high rate of synthetically injected errors in SPEC programs
- Detected two previously unreported benign bugs (197.parser and espresso)
- Successfully hides buffer overflow error in Squid web cache server (v 2.3s5)
- But don't take my word for it...

Experiments / Benchmarks

- 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 based on a dictionary
 - Made the dictionary data structures critical
- rayshade: Renders a scene file
 - Made the list of objects to be rendered critical

Related Work

- Conservative GC (Boehm / Demers / Weiser)
 - Time-space tradeoff (typically >3X)
 - Provably avoids certain errors
- Safe-C compilers
 - Jones & Kelley, Necula, Lam, Rinard, Adve, ...
 - Often built on BDW GC
 - Up to 10X performance hit
- N-version programming
 - Replicas truly statistically independent
- Address space randomization (as in Vista)
- Failure-oblivious computing [Rinard]
 - Hope that program will continue after memory error with no untoward effects