DieHard: Probabilistic Memory Safety for Unsafe Programming Languages

Emery Berger
University of Massachusetts Amherst

Ben Zorn
Microsoft Research

Problems with Unsafe Languages

- C, C++: pervasive apps, but langs. **memory unsafe**
- Numerous opportunities for security vulnerabilities, **errors**
  - Double **`free`**
  - Invalid **`free`**
  - Uninitialized reads
  - Dangling pointers
  - Buffer overflows (stack & **heap**)

Current Approaches

- **Unsound, *may* work or abort**
  - Windows, GNU libc, etc., *Rx* [Zhou]
- **Unsound, *will definitely* continue**
  - *Failure oblivious* [Rinard]
- **Sound, *definitely* aborts (fail-safe)**
  - *CCured* [Necula], *CRED* [Ruwase & Lam], *SAFECode* [Dhurjati, Kowshik & Adve]
    - Requires C source, programmer intervention
    - 30% to 20X slowdowns
  - Good for *debugging*, less for *deployment*

Probabilistic Memory Safety

DieHard: correct execution in face of errors
with high probability

- **Fully-randomized** memory manager
  - Increases odds of **benign** memory errors
  - Ensures different heaps across users
- **Replication**
  - Run multiple **replicas** simultaneously, vote on results
    - Detects crashing & non-crashing errors
- Trades space for increased reliability

Soundness for "Erroneous" Programs

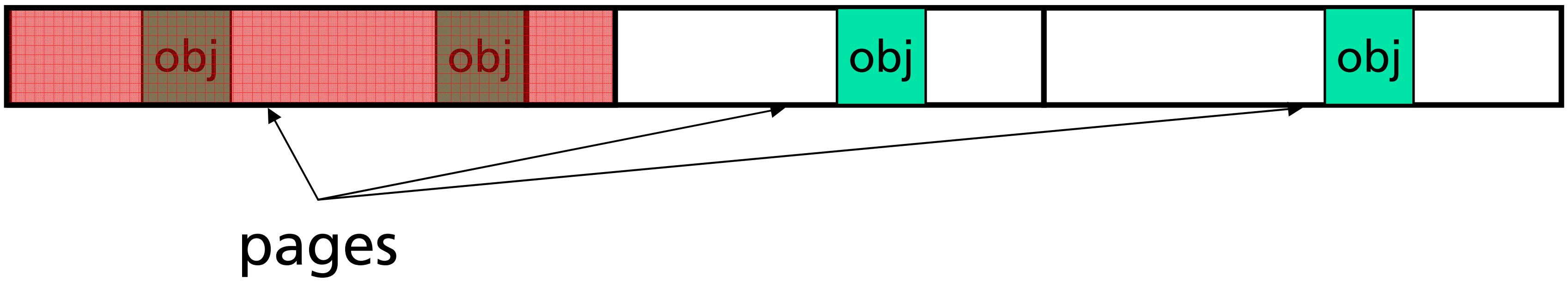
- Normally: memory errors $\Rightarrow \perp$ …
- Consider **infinite-heap** allocator:
  - All **new**s *fresh*;
    ignore **`delete`**
    - No dangling pointers, invalid frees, double frees
  - Every object **infinitely large**
    - No buffer overflows, data overwrites
- Transparent to correct program
- "Erroneous" programs **sound**

Approximating Infinite Heaps

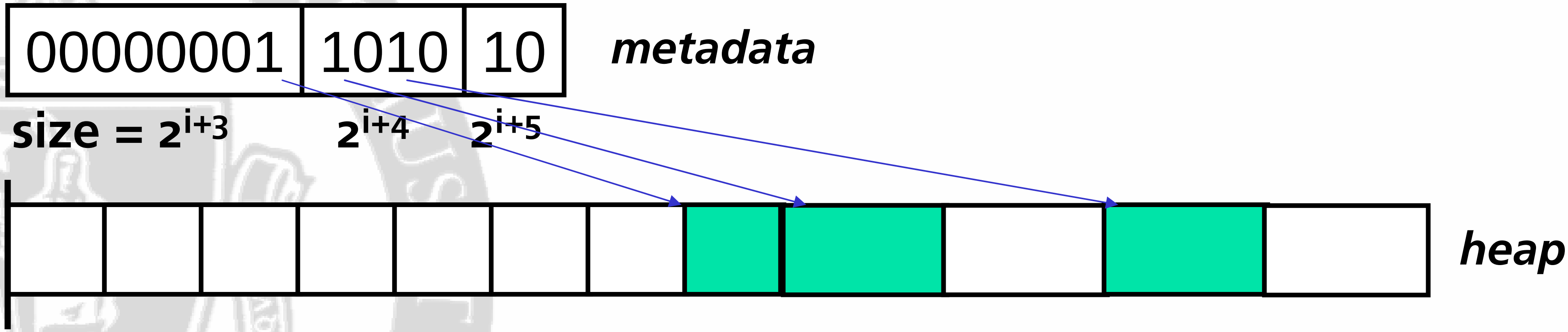
- Infinite ⇒ M-heaps: probabilistic soundness
- **Pad** allocations & **defer** deallocations
  - \+ Simple
  - – **No** protection from larger overflows
    - – pad = 8 bytes, overflow = 9 bytes…
  - – *Deterministic*: overflow crashes *everyone*
- Better: **randomize** heap
  - \+ Probabilistic protection against errors
    - \+ *Independent* across heaps
  - ? Efficient implementation…

Implementation Choices

- Conventional, freelist-based heaps
  - Hard to randomize, protect from errors
    - Double frees, heap corruption
- What about bitmaps? [Wilson90]
  – **Catastrophic** fragmentation
    - Each small object likely to occupy one page

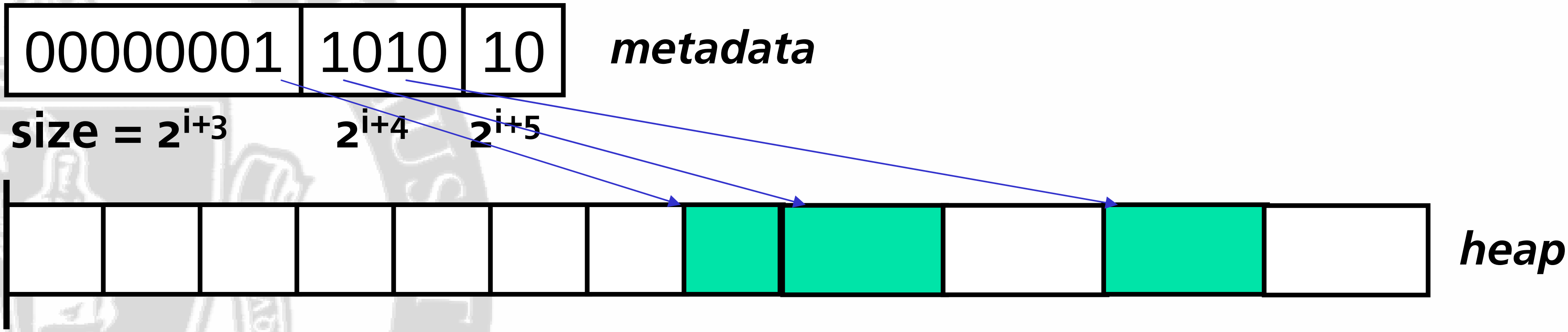


Randomized Heap Layout



- Bitmap-based, **segregated** size classes
  - Bit represents one **object** of given size
    - i.e., one bit = 2^{i+3} bytes, etc.
  - Prevents fragmentation

Randomized Allocation

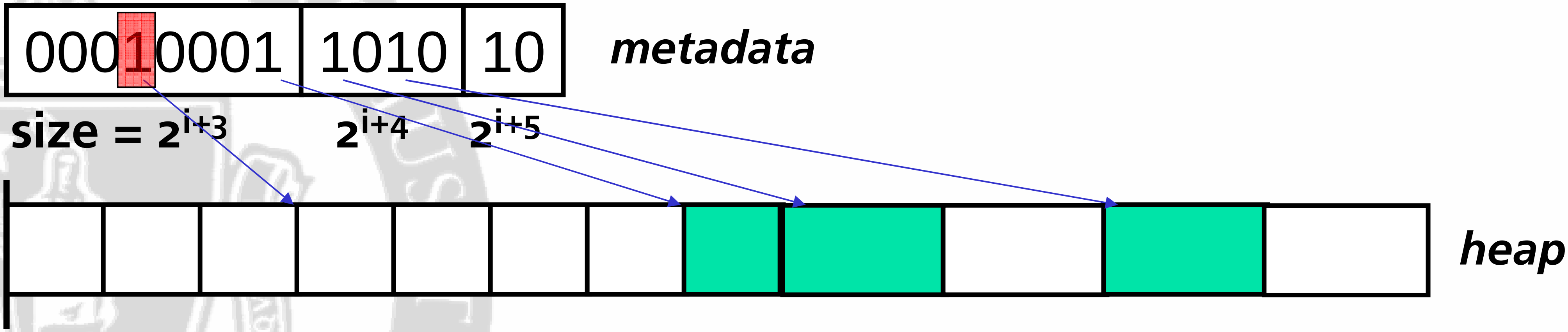


`malloc(8)`:

- compute size class = ceil($\log_2$ sz) – 3
- **randomly** probe bitmap for zero-bit (free)

- Fast: runtime O(1)
  - $M=2 \Rightarrow E[\#\text{ of probes}] \leq 2$

Randomized Allocation

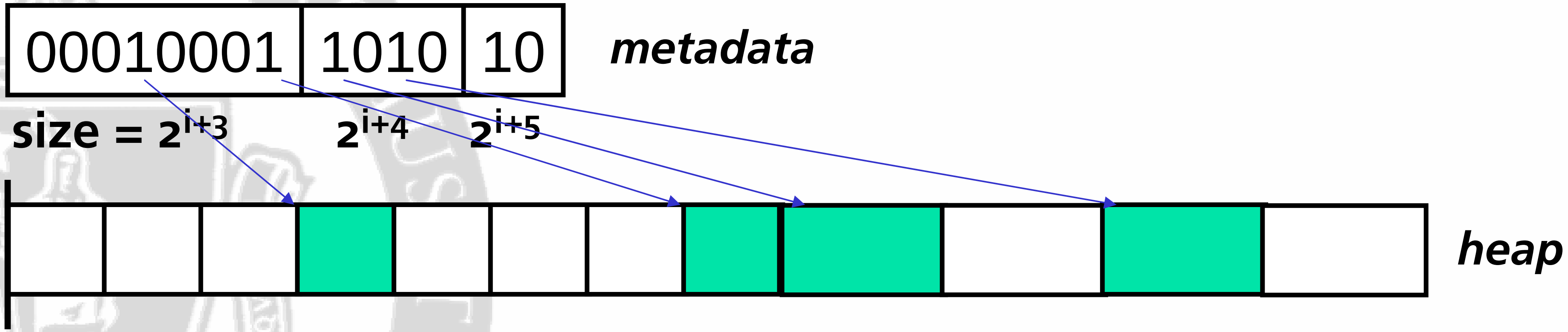


`malloc(8)`:

- compute size class = $\lceil \log_2 sz \rceil - 3$
- **randomly** probe bitmap for zero-bit (free)

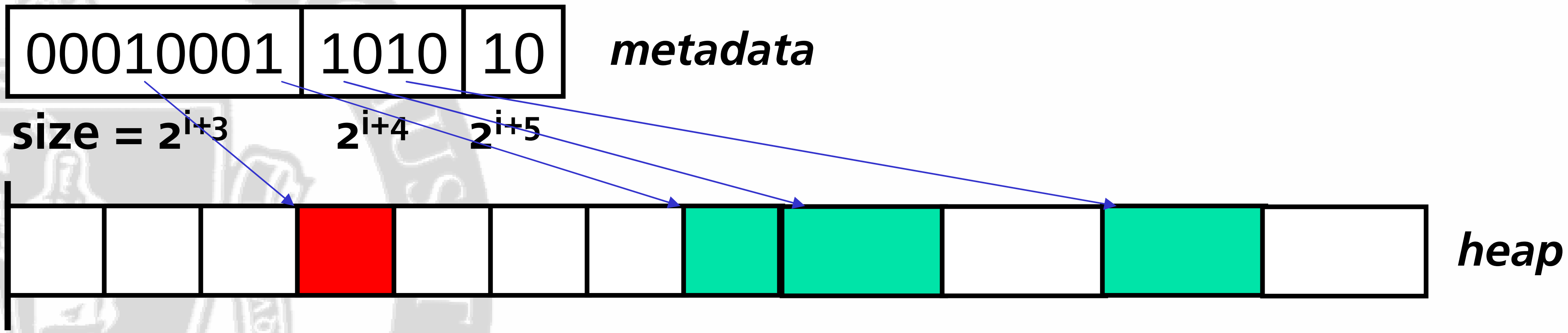
- Fast: runtime O(1)
  - $M=2 \Rightarrow E[\#\text{ of probes}] \leq 2$

Randomized Deallocation



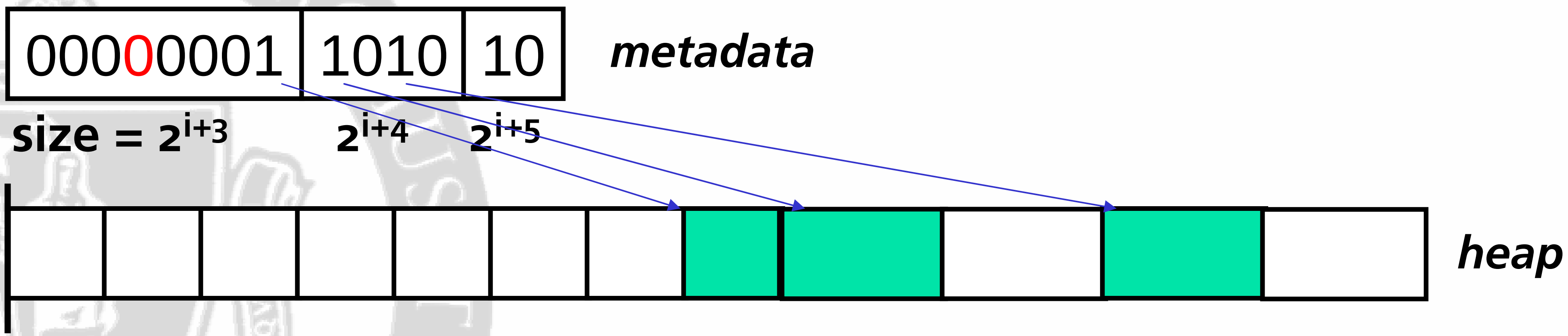
- `free(ptr)`:
  - Ensure object *valid* – aligned to right address
  - Ensure *allocated* – bit set
  - Resets bit
- Prevents invalid frees, double frees

Randomized Deallocation



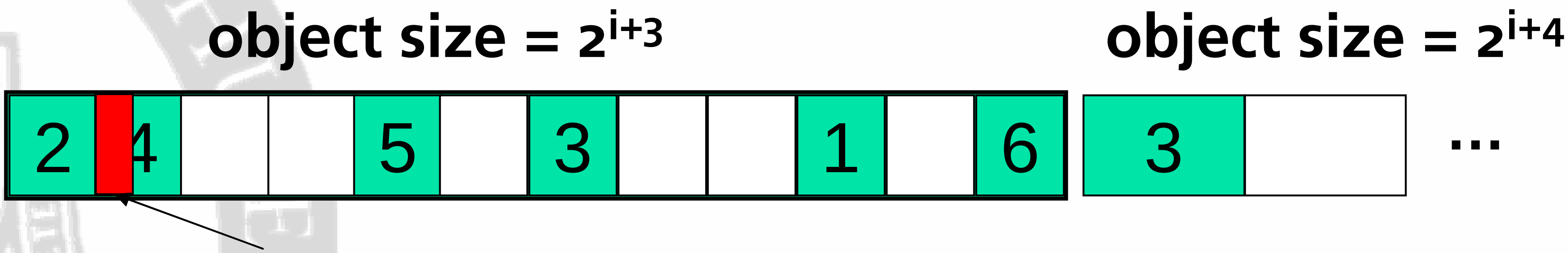
- `free(ptr)`:
  - Ensure object *valid* – aligned to right address
  - Ensure *allocated* – bit set
  - Resets bit
- Prevents invalid frees, double frees

Randomized Deallocation



- `free(ptr)`:
  - Ensure object *valid* – aligned to right address
  - Ensure *allocated* – bit set
  - Resets bit
- Prevents invalid frees, double frees

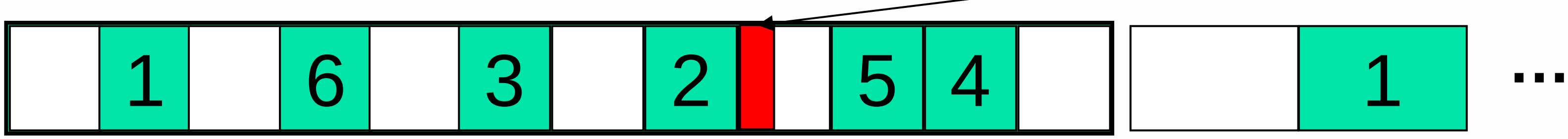
Randomized Heaps & Reliability



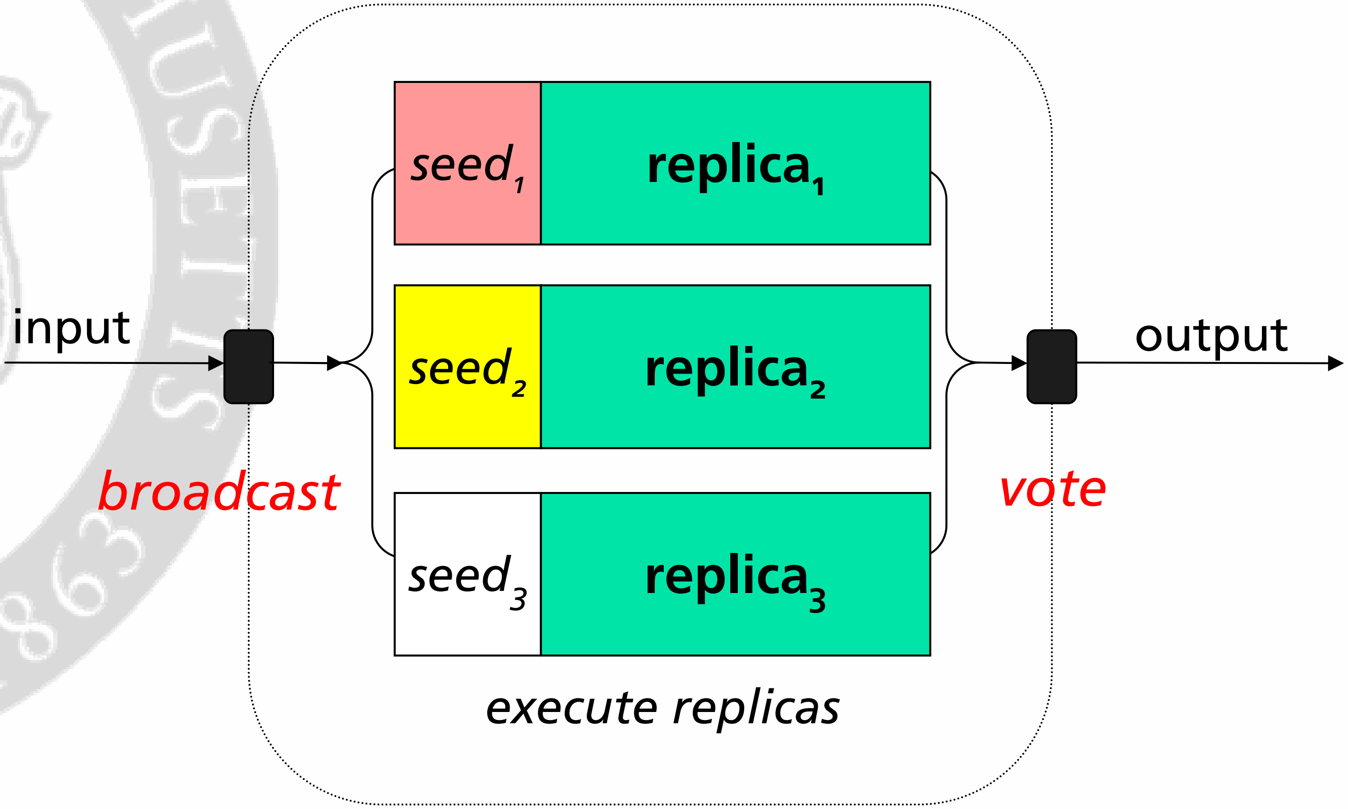
My Mozilla: "malignant" overflow

- Objects randomly spread across heap
- Different run = different heap
  - Errors across heaps *independent*

Your Mozilla: "benign" overflow



DieHard software architecture



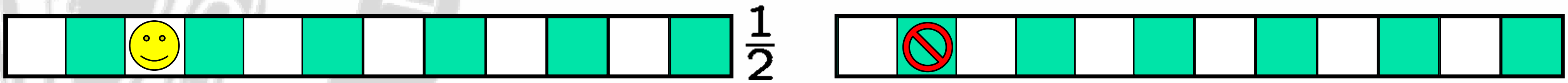
- Each replica has different allocator
- “Output equivalent” – kill failed replicas

Results

- Analytical results (pictures!)
  - Buffer overflows
  - Dangling pointer errors
  - Uninitialized reads
- Empirical results
  - Runtime overhead
  - Error avoidance
    - Injected faults & actual applications

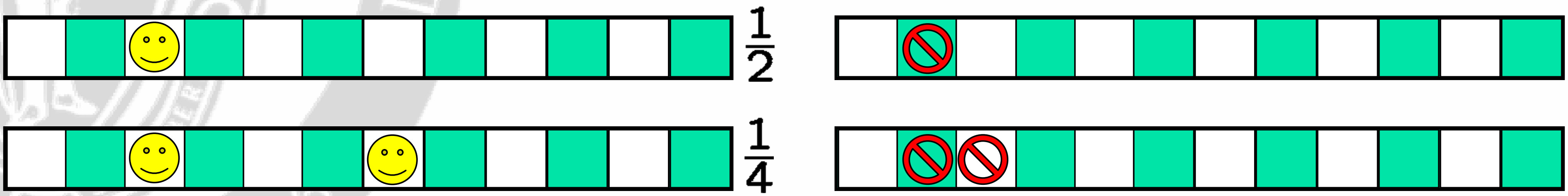
Analytical Results: Buffer Overflows

- Model overflow as write of live data
  - Heap half full (max occupancy)



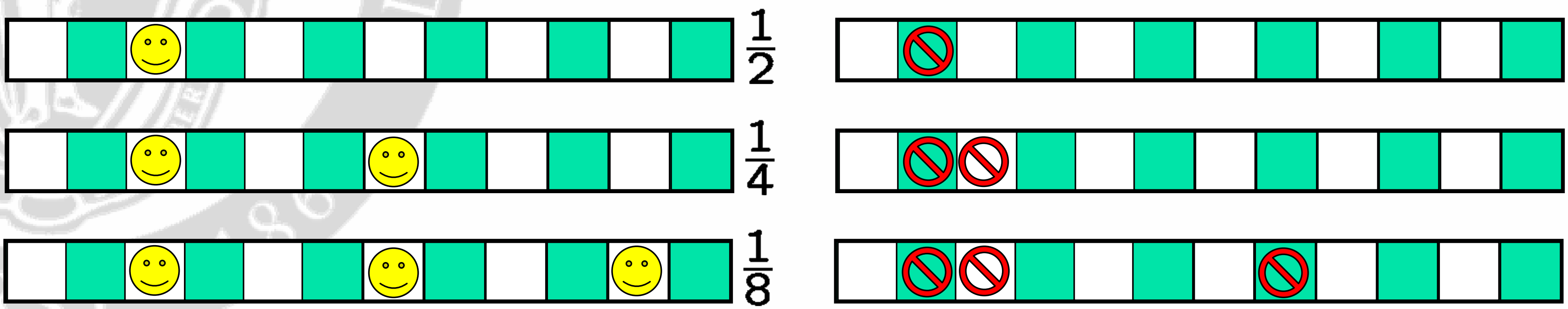
Analytical Results: Buffer Overflows

- Model overflow as write of live data
  - Heap half full (max occupancy)



Analytical Results: Buffer Overflows

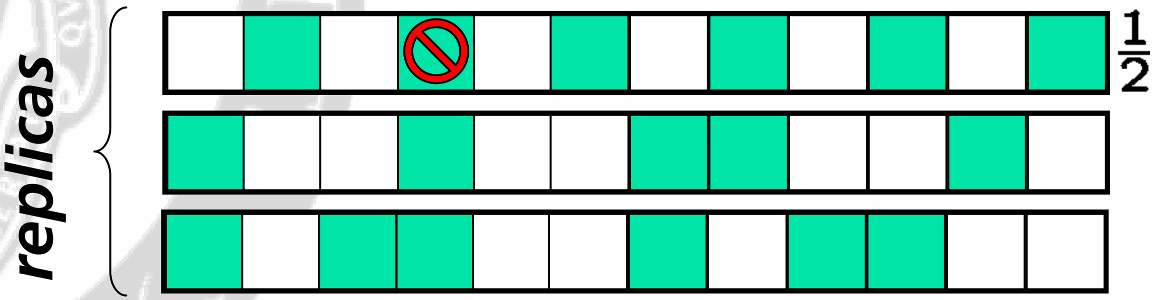
- Model overflow as write of live data
  - Heap half full (max occupancy)



$$P(\text{No Overflow}) \geq \left(\frac{1}{2}\right)^N$$

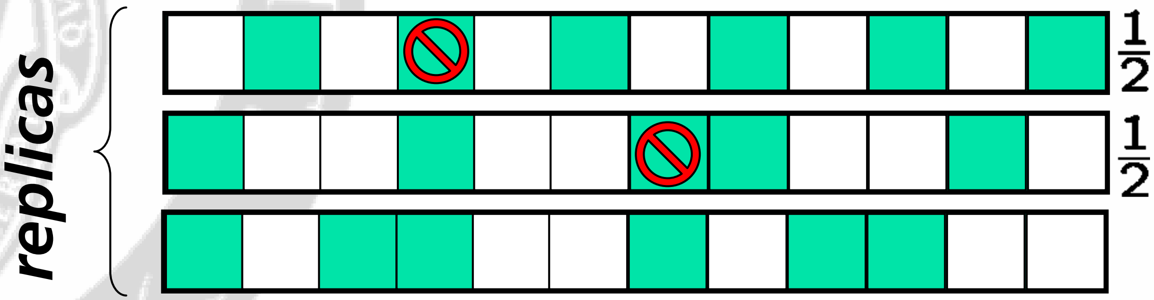
Analytical Results: Buffer Overflows

- **Replicas:** Increase odds of avoiding overflow in *at least one* replica



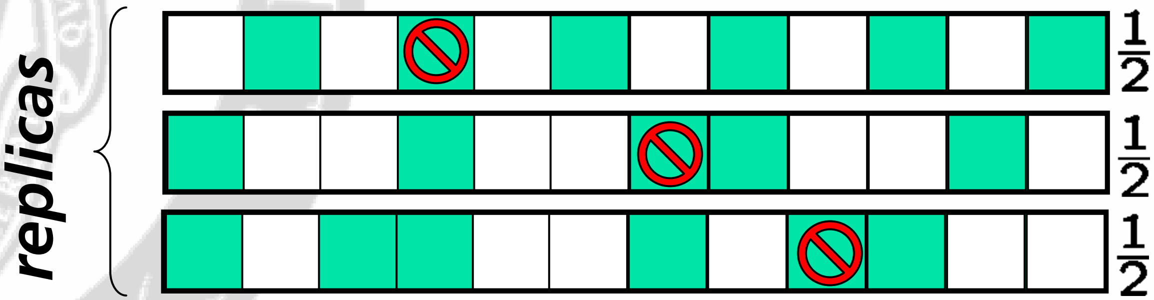
Analytical Results: Buffer Overflows

- **Replicas:** Increase odds of avoiding overflow in *at least one* replica



Analytical Results: Buffer Overflows

- **Replicas:** Increase odds of avoiding overflow in *at least one* replica

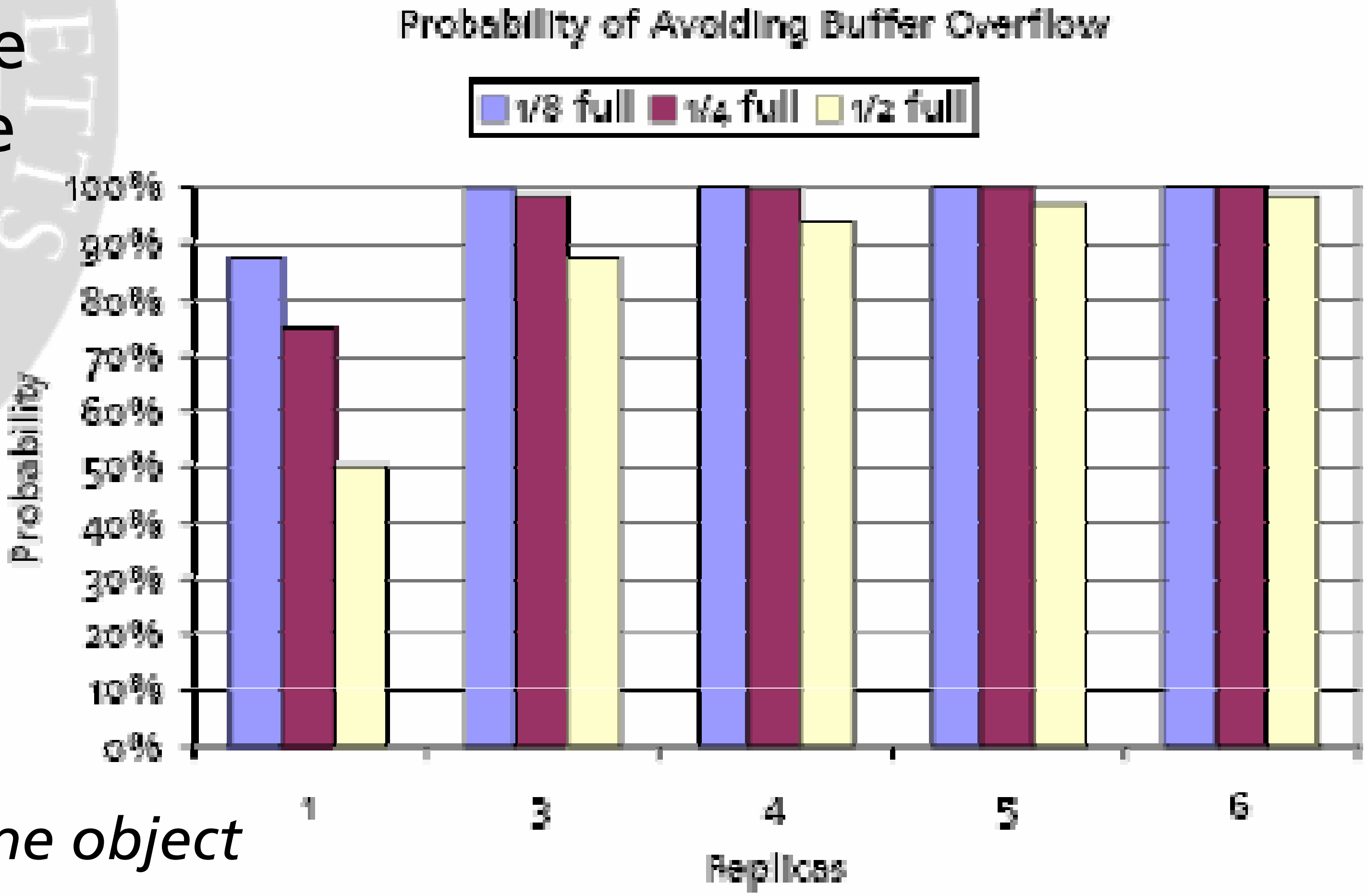


- P(Overflow in **all** replicas) = $(1/2)^3 = 1/8$
- P(No overflow in $\geq$ 1 replica) = $1-(1/2)^3 = 7/8$

Analytical Results: Buffer Overflows

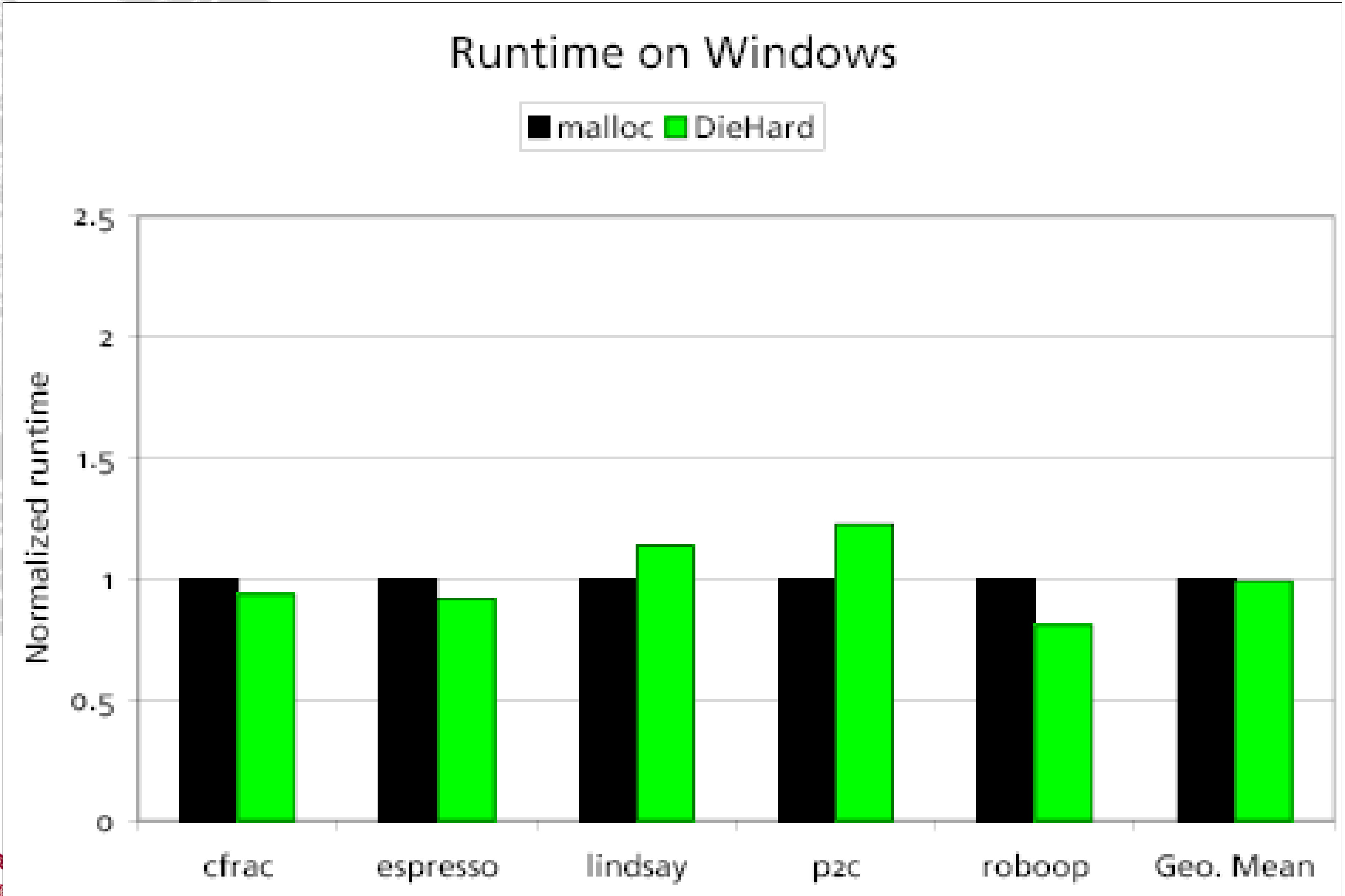
$$P(\text{No Overflow Error}) = 1 - \left[1 - \left(\frac{F}{H}\right)^N\right]^k$$

- *F* = free space
- *H* = heap size
- *N* = # objects worth of overflow
- *k* = replicas

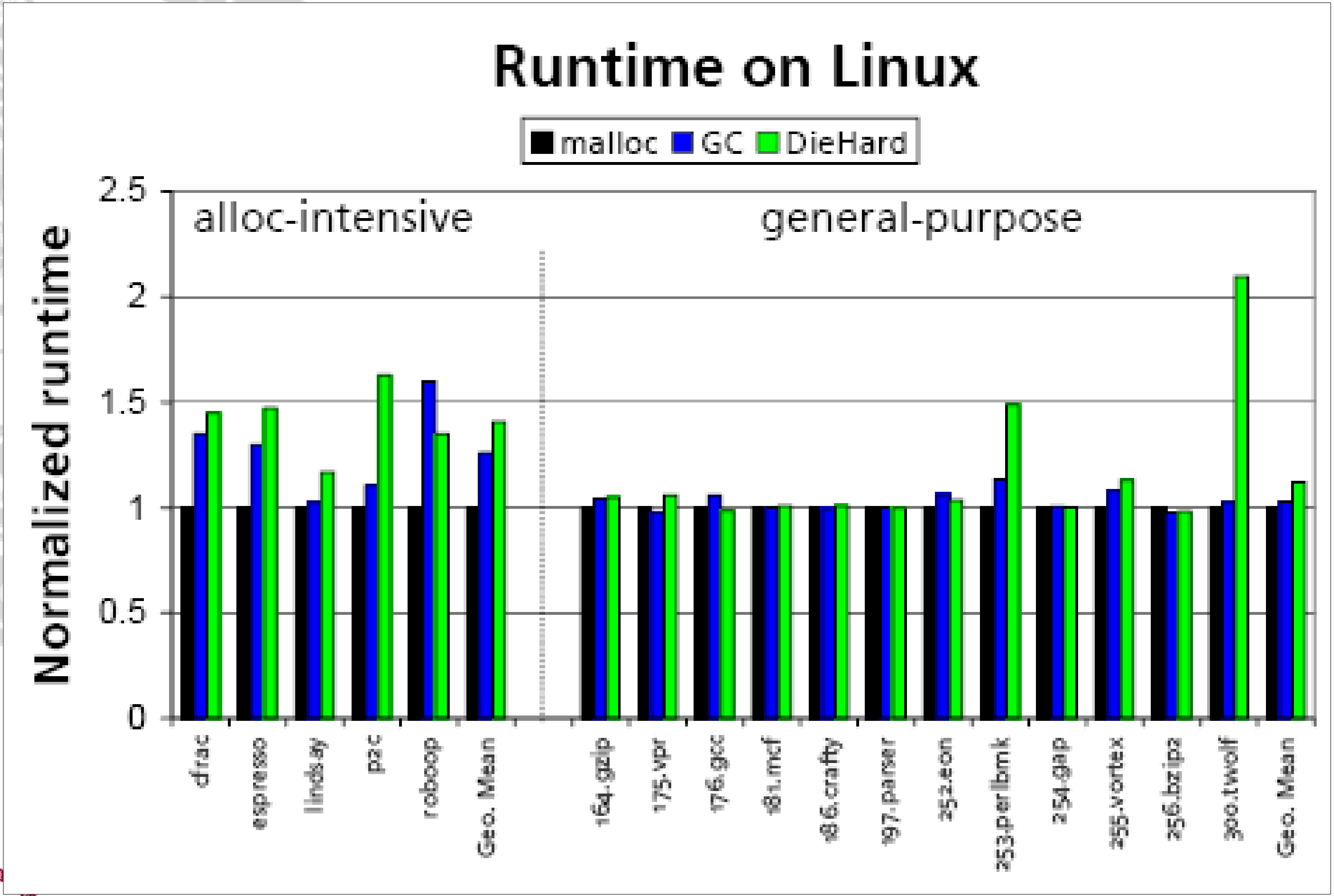


- *Overflow one object*

Empirical Results: Runtime



Empirical Results: Runtime

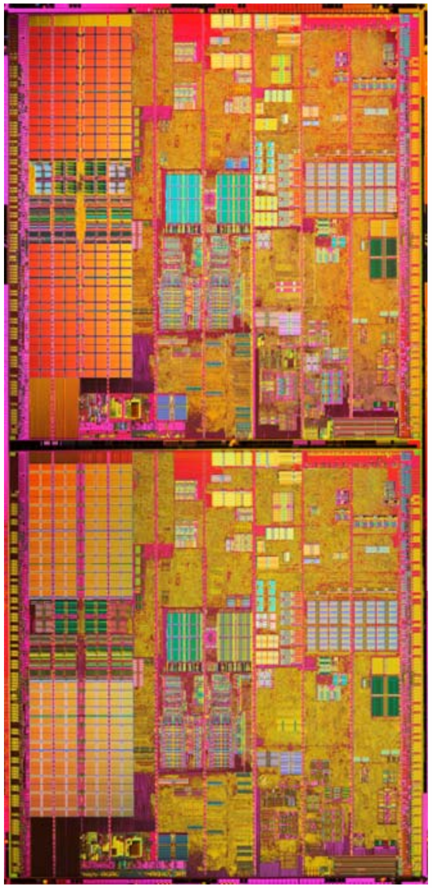


Empirical Results: Error Avoidance

- **Injected faults:**
  - ***Dangling pointers*** (@50%, 10 allocations)
    - **glibc**: *crashes*; **DieHard**: *9/10 correct*
  - ***Overflows*** (@1%, 4 bytes over) –
    - **glibc**: *crashes 9/10, inf loop*; **DieHard**: *10/10 correct*
- **Real faults:**
  - Avoids Squid web cache overflow
    - *Crashes BDW & glibc*
  - Avoids dangling pointer error in Mozilla
    - *DoS in glibc & Windows*

Conclusion

- Randomization + replicas = **probabilistic memory safety**
  - Improves over today (0%)
  - Useful point between absolute soundness (fail-stop) and unsound
- **Trades hardware resources (RAM,CPU) for reliability**
  - Hardware trends
    - Larger memories, multi-core CPUs
  - Follows in footsteps of ECC memory, RAID



DieHard software

http://www.cs.umass.edu/~emery/diehard

- *Linux, Solaris (stand-alone & replicated)*
- *Windows (stand-alone only)*