DieHard: Memory Error Fault Tolerance in C and C++

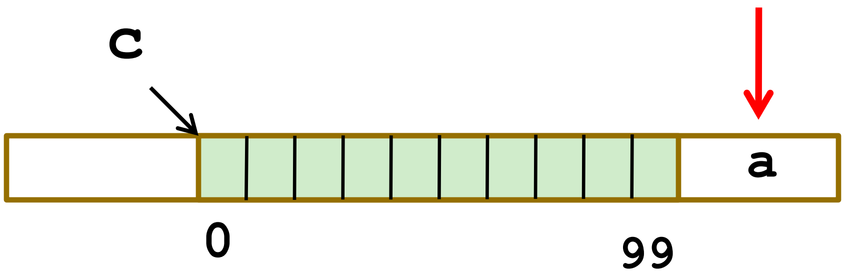
Ben Zorn
Microsoft Research

In collaboration with
Emery Berger and Gene Novark, Univ. of Massachusetts
Ted Hart, Microsoft Research

Focus on Heap Memory Errors

- Buffer overflow

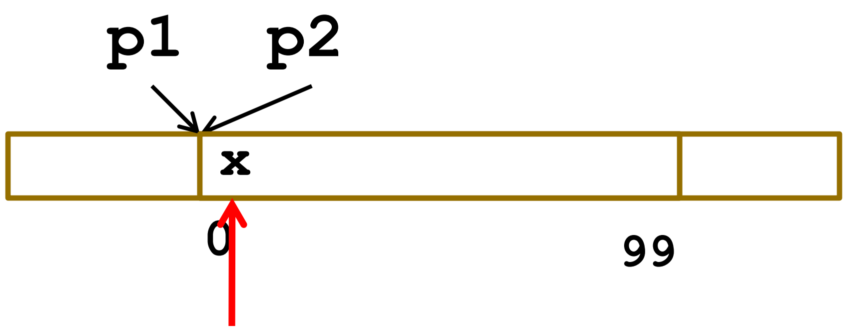
```
char *c = malloc(100);
c[101] = 'a';
```



- Dangling reference

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

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



Motivation

- Consider a shipped C program with a memory error (e.g., buffer overflow)
  - By language definition, “undefined”
  - In practice, assertions turned off – mostly works
    - I.e., data remains consistent
- What if you know it has executed an illegal operation?
  - Raise an exception?
  - Continue unsoundly (failure oblivious computing)
  - **Continue with well-defined semantics**

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

Research Themes

- Make existing programs more fault tolerant
  - Define semantics of programs with errors
  - Programs complete with correct result despite errors
- Go beyond all-or-nothing guarantees
  - Type checking, verification rarely a 100% solution
    - C#, Java both call to C/C++ libraries
  - Traditional engineering allows for errors by design
- Complement existing approaches
  - Static analysis has scalability limits
  - Managed code especially good for new projects
  - DART, Fuzz testing effective for generating illegal test cases

Approaches to Protecting Programs

- Unsound, *may* work or abort
  - Windows, GNU libc, etc.
- Unsound, *might* continue
  - *Failure oblivious* (keep going) [Rinard]
    - Invalid read => manufacture value
    - Illegal write => ignore
- Sound, *definitely* aborts (fail-safe, fail-fast)
  - CCured [Necula], others
- Sound and continues
  - **DieHard,** Rx, Boundless Memory Blocks, hardware fault tolerance

Outline

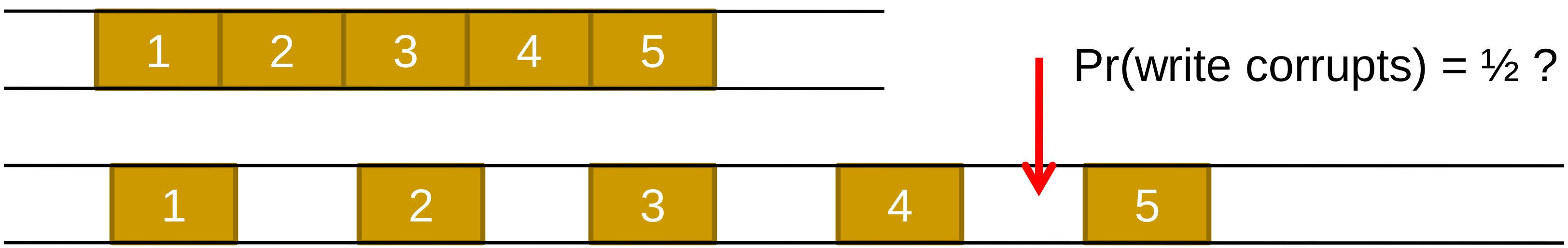
- Motivation
- DieHard
  - Collaboration with Emery Berger
  - Replacement for malloc/free heap allocation
  - No source changes, recompile, or patching, required
- Exterminator
  - Collaboration with Emery Berger, Gene Novark
  - Automatically corrects memory errors
  - Suitable for large scale deployment
- Conclusion

DieHard: Probabilistic Memory Safety

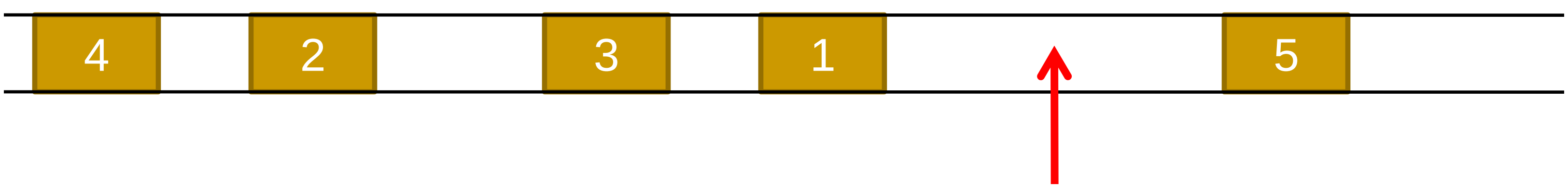
- Collaboration with Emery Berger
- Plug-compatible replacement for malloc/free in C lib
- We define "infinite heap semantics"
  - Programs execute as if each object allocated with unbounded memory
  - All frees ignored
- Approximating infinite heaps – 3 key ideas
  - Overprovisioning
  - Randomization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

Overprovisioning, Randomization

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



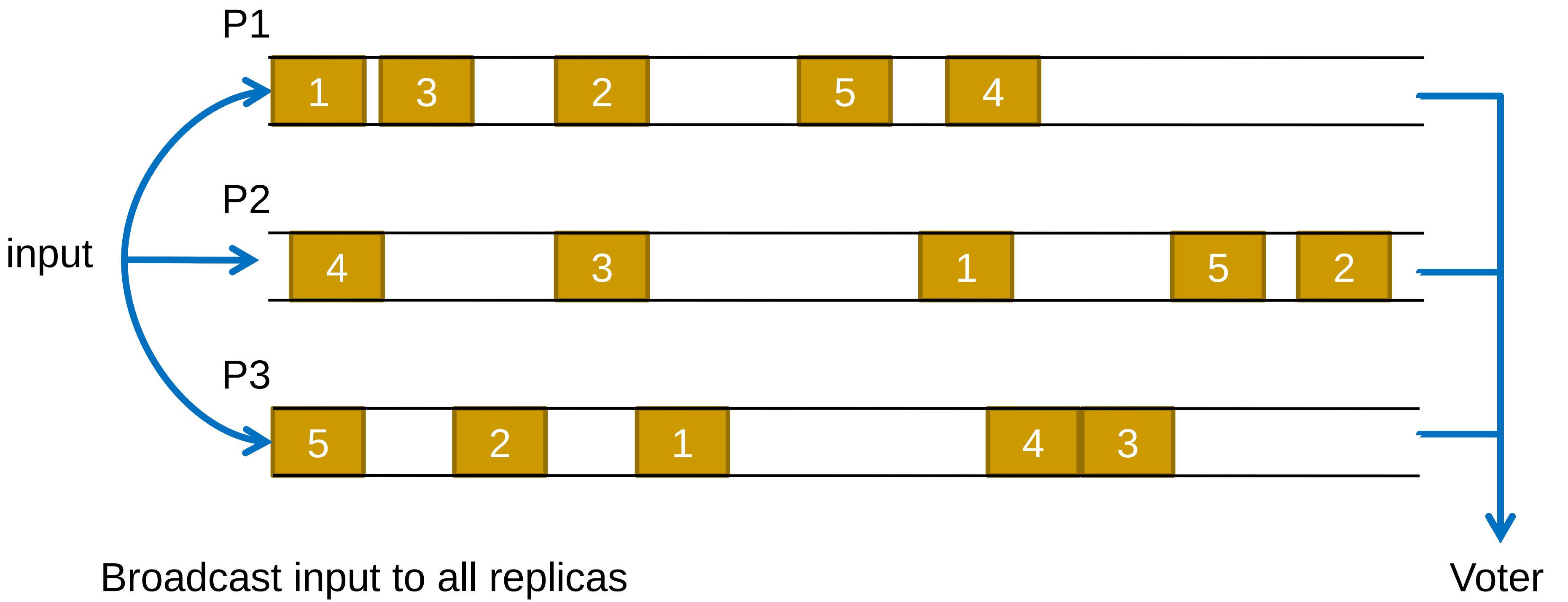
Randomize object placement



Pr(write corrupts) = ½ !

Replication (optional)

Replicate process with different randomization seeds



Broadcast input to all replicas

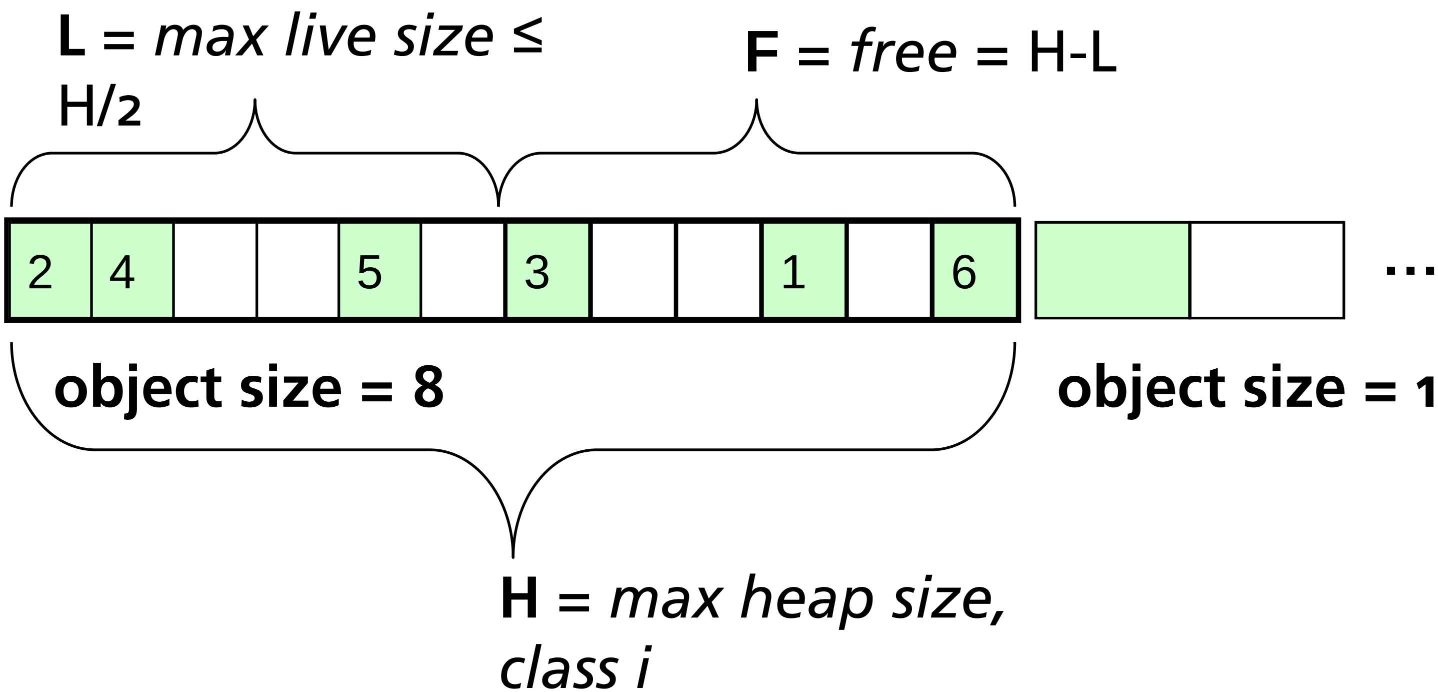
Compare outputs of replicas, kill when replica disagrees

DieHard Implementation Details

- Multiply allocated memory by factor of M
- Allocation
  - Segregate objects by size (log2), 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



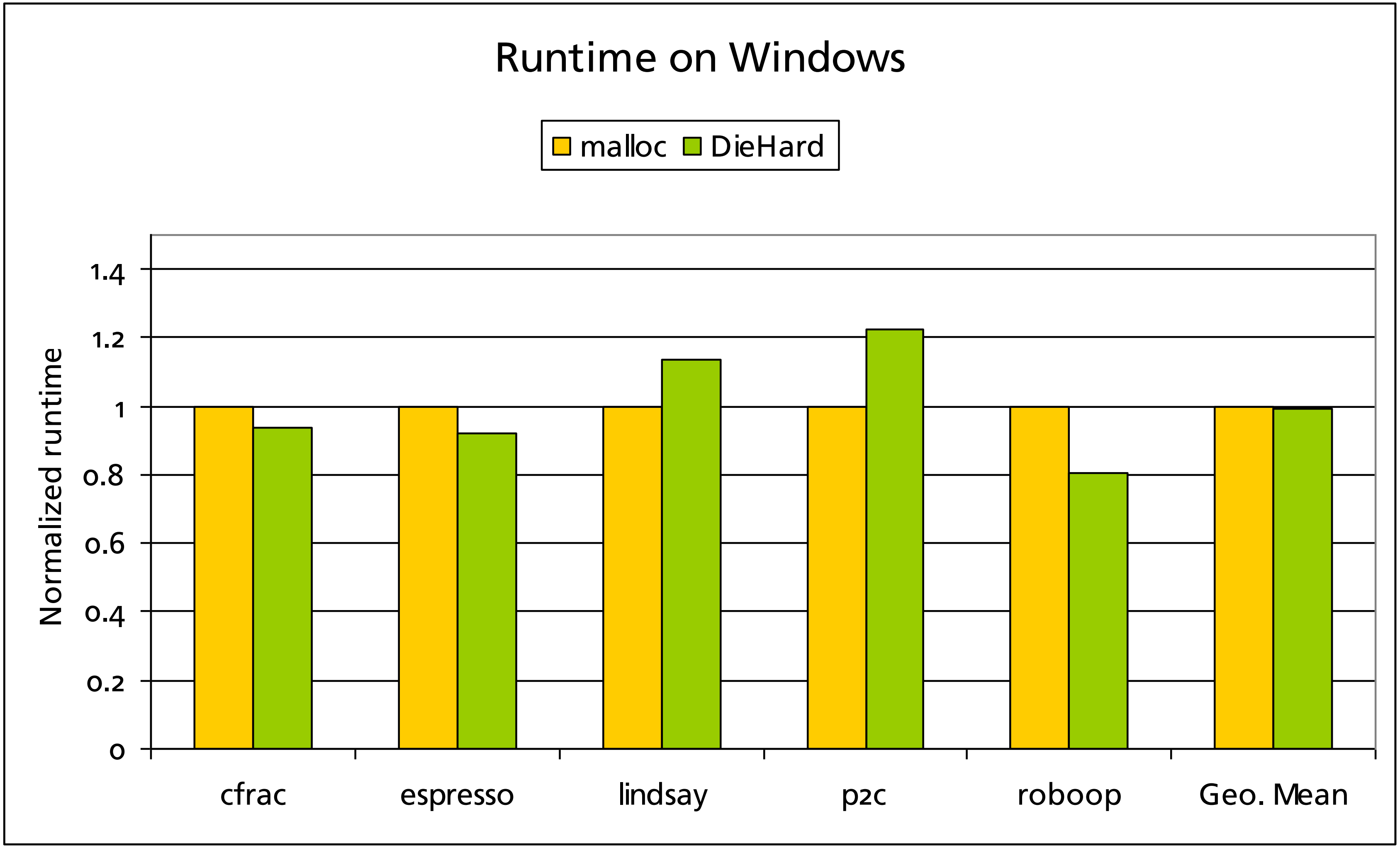
- Static strategy pre-allocates size classes
- Adaptive strategy grows each size class incrementally

Randomness enables Analytic Reasoning Example: Buffer Overflows

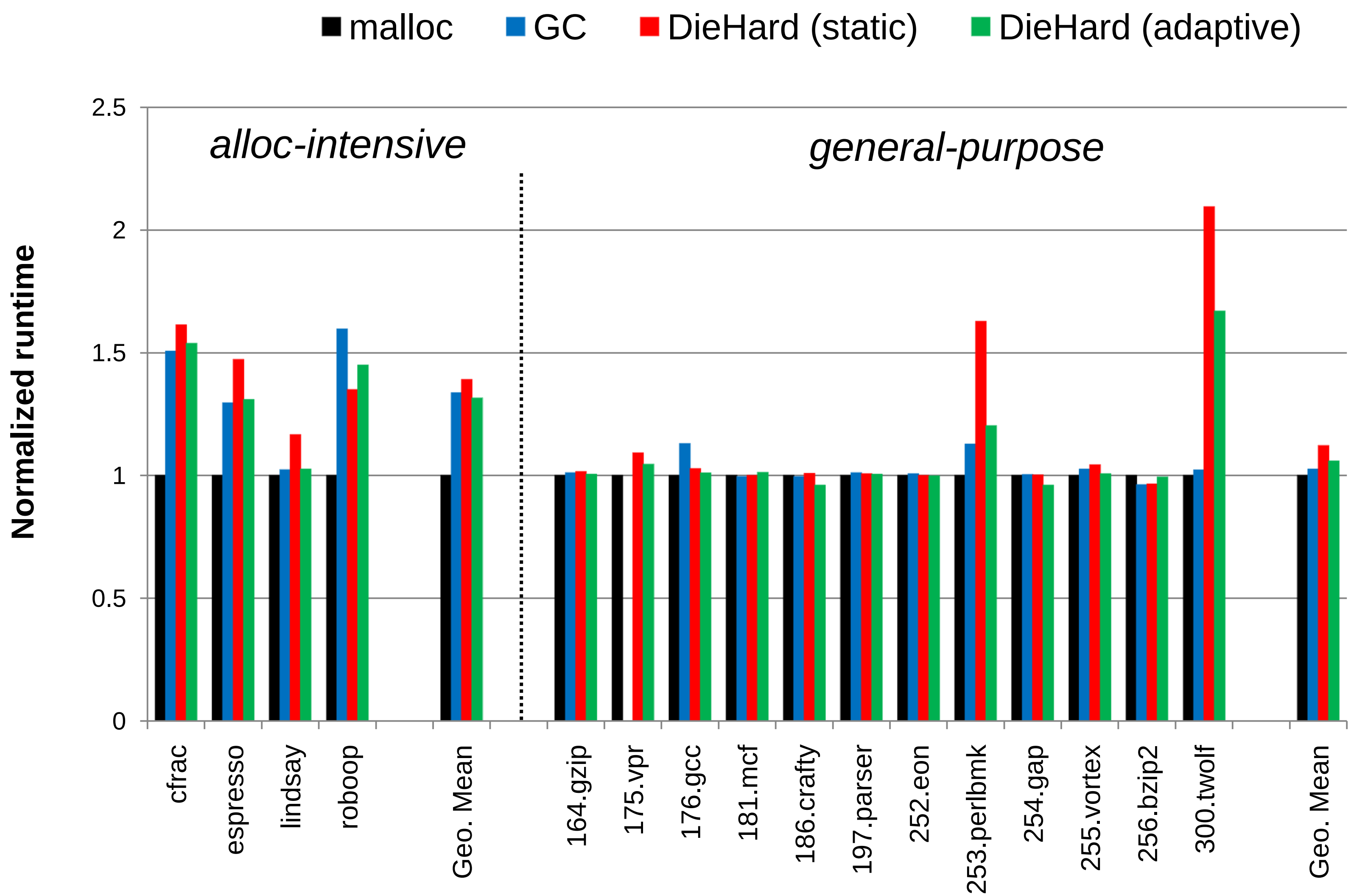
$$\text{Pr(Mask Buffer Overflow)} = 1 - \left[1 - \left(\frac{F}{H}\right)^{Obj}\right]^{k}$$

- *k* = # of replicas, *Obj* = size of overflow
- With no replication, Obj = 1, heap no more than 1/8 full:
  Pr(Mask buffer overflow), = 87.5%
- 3 replicas: Pr(*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…

DieHard Demo

- DieHard (non-replicated)
  - Windows, Linux version implemented by Emery Berger
  - Available: http://www.diehard-software.org/
  - Adaptive, automatically sizes heap
  - Detours-like mechanism to automatically redirect malloc/free calls to DieHard DLL
- Application: Mozilla, version 1.7.3
  - Known buffer overflow crashes browser
- Takeaways
  - 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.)

Outline

- Motivation
- DieHard
  - Collaboration with Emery Berger
  - Replacement for malloc/free heap allocation
  - No source changes, recompile, or patching, required
- Exterminator
  - Collaboration with Emery Berger, Gene Novark
  - Automatically corrects memory errors
  - Suitable for large scale deployment
- Conclusion

Exterminator Motivation

- DieHard limitations
  - Tolerates errors probabilistically, doesn't fix them
  - Memory and CPU overhead
  - Provides no information about source of errors
  - Note – DieHard still extremely useful
- "Ideal" 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 instead of tolerating them

Exterminator Components

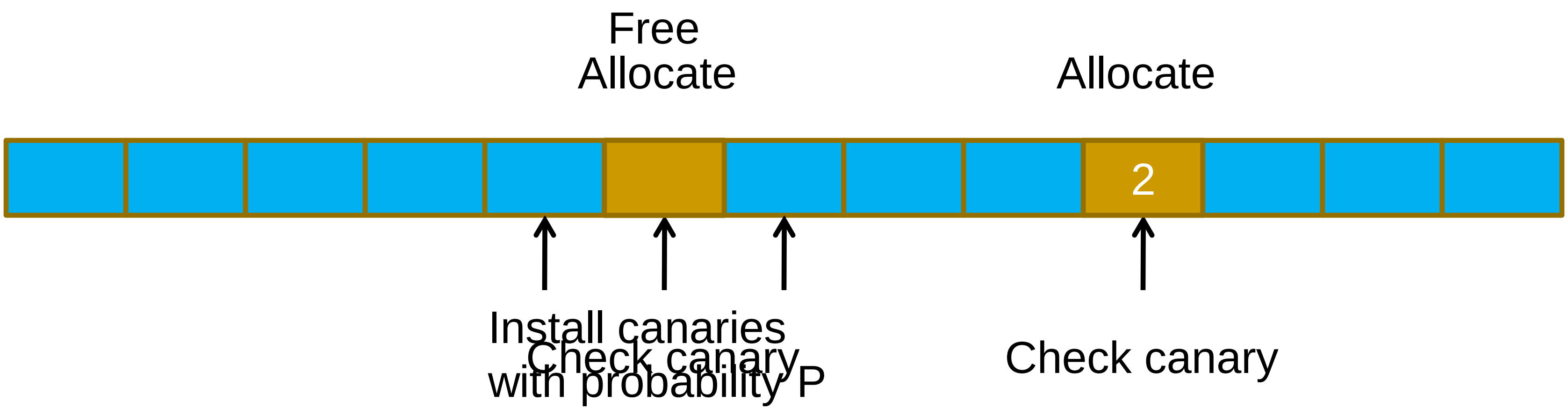
- Architecture of Exterminator dictated by solving specific problems
- How to detect heap corruptions effectively?
  - DieFast allocator
- How to isolate the cause of a heap corruption precisely?
  - Heap differencing algorithms
- How to automatically fix buggy C code without breaking it?
  - Correcting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

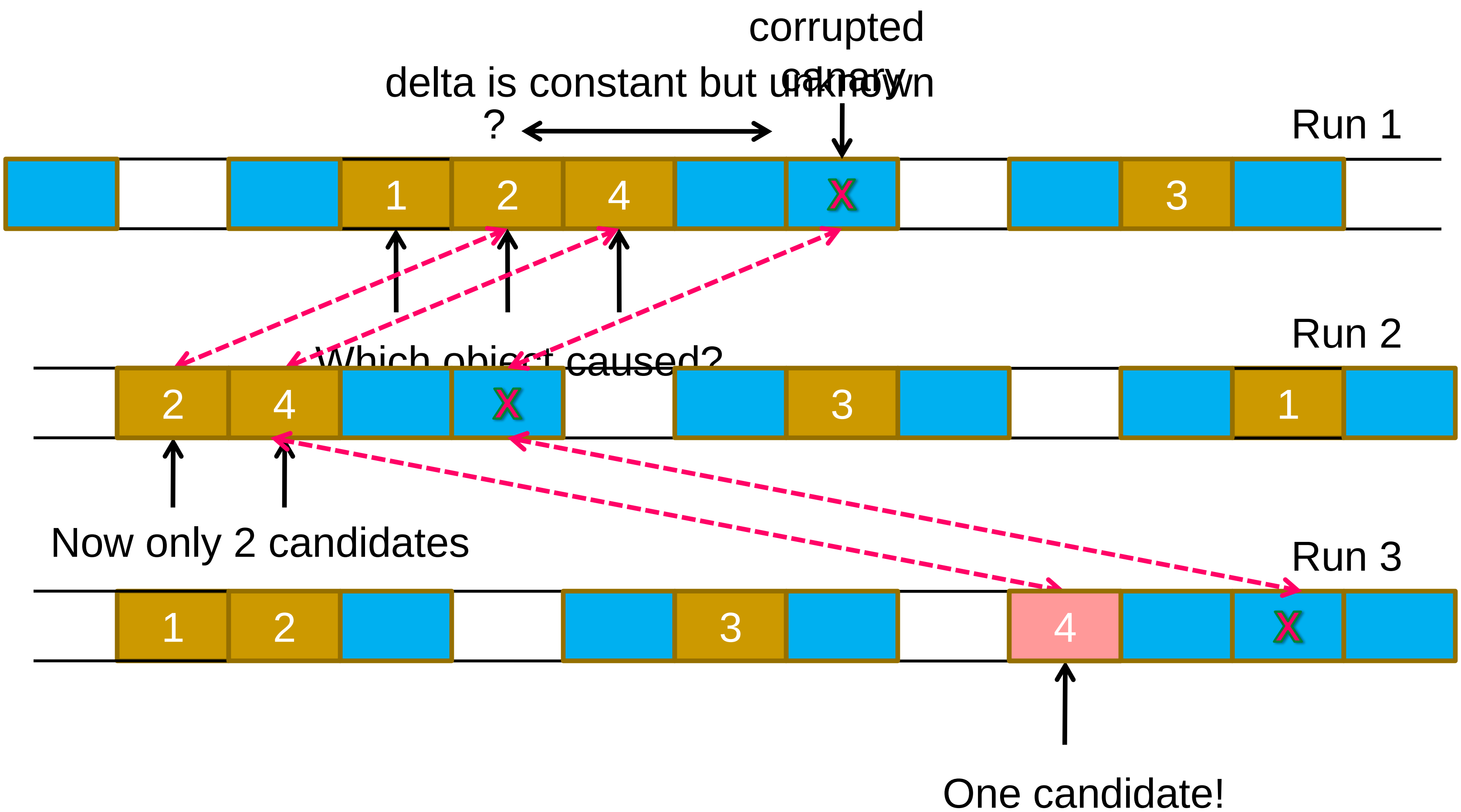
Initially, heap full of canaries



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 <u>all</u> 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: ≈ $\log_2$(number of callsites)
    - Look for correlations, i.e., X filled with canary => crash
    - Establish conditional probabilities
      - Have: P(callsite X filled with canary | program crashes)
      - Need: P(crash | filled with canary), guess “prior” to compute

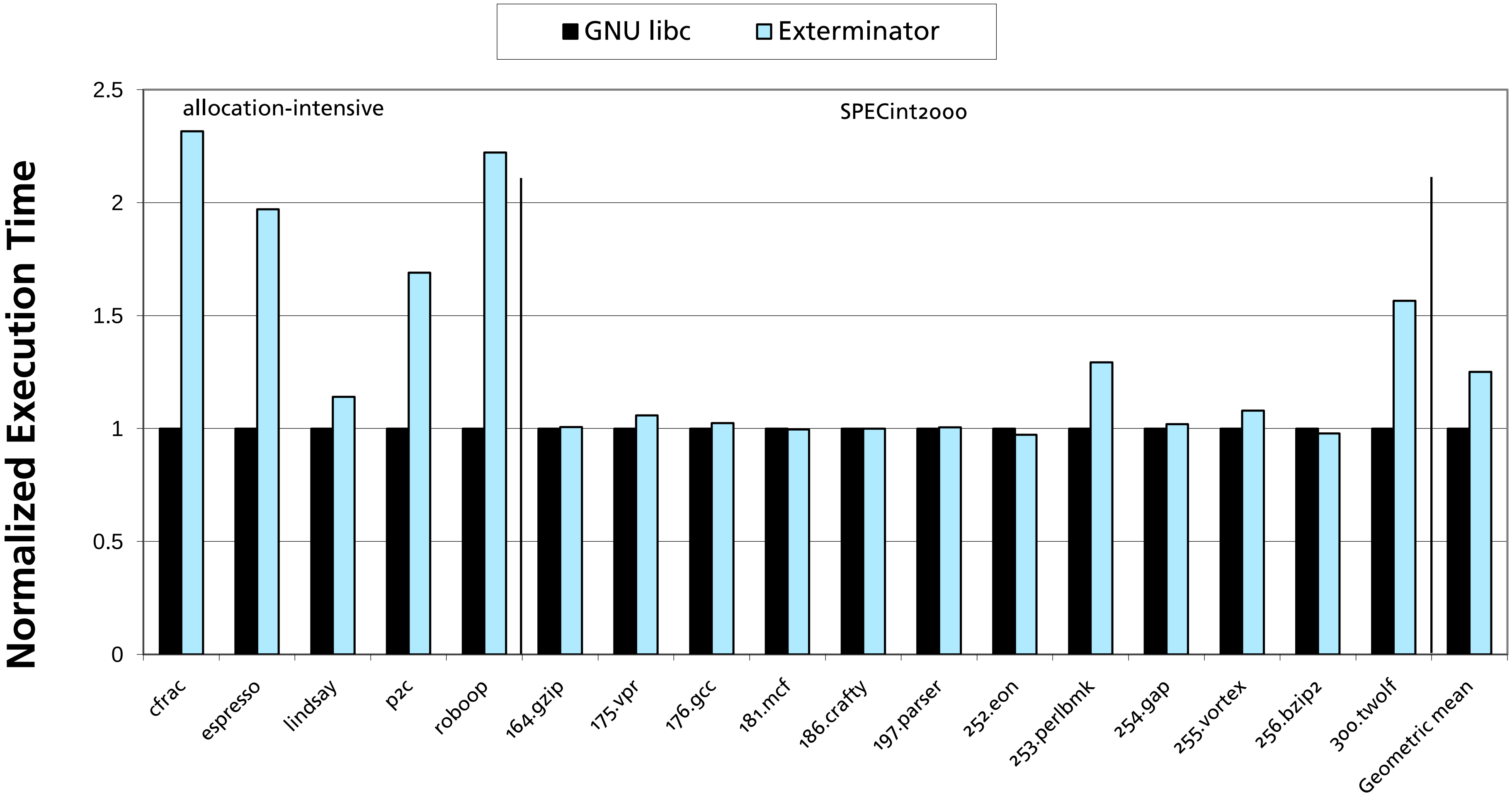
Correcting Allocator

- 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

Comparison with Existing Approaches

- Static analysis, annotations
  - Finds individual bugs, developer still has to fix
  - High cost developing, testing, deploying patches
  - DieHard reduces threat of <u>all memory errors</u>
- Testing, OCA / Watson dumps
  - Finds crashes, developer still has find root cause
- Type-safe languages (C#, etc.)
  - Large installed based of C, C++
  - Managed runtimes, libraries have lots of C, C++
  - Also has a memory cost

Conclusion

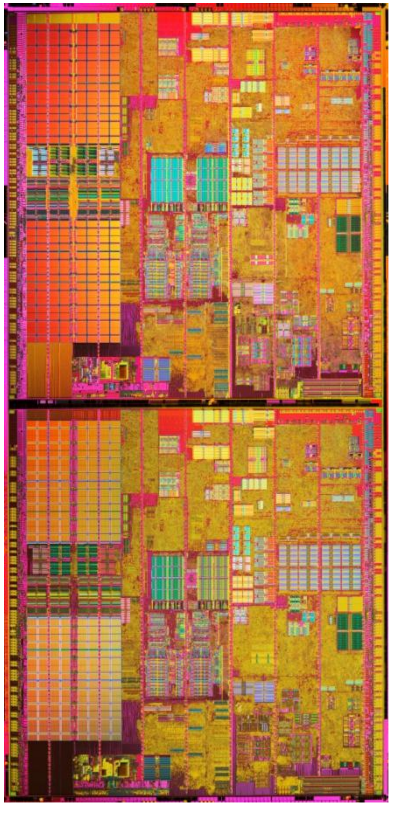
- Programs written in C / C++ can execute safely and correctly despite memory errors
- Research vision
  - Improve existing code without source modifications
  - Reduce human generated patches required
  - Increase reliability, security by order of magnitude
- Current projects and results
  - DieHard: overprovisioning + randomization + replicas = probabilistic memory safety
  - Exterminator: automatically detect and correct memory errors (with high probability)
  - Demonstrated success on real applications

Hardware Trends

- 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

Power to Spare

- 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? http://www.hardwaresecrets.com/news/709
- *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.*
  - Gene Novark, Emery D. Berger and Benjamin G. Zorn, "**Exterminator: Correcting Memory Errors with High Probability**", *PLDI'07.*

Backup Slides

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