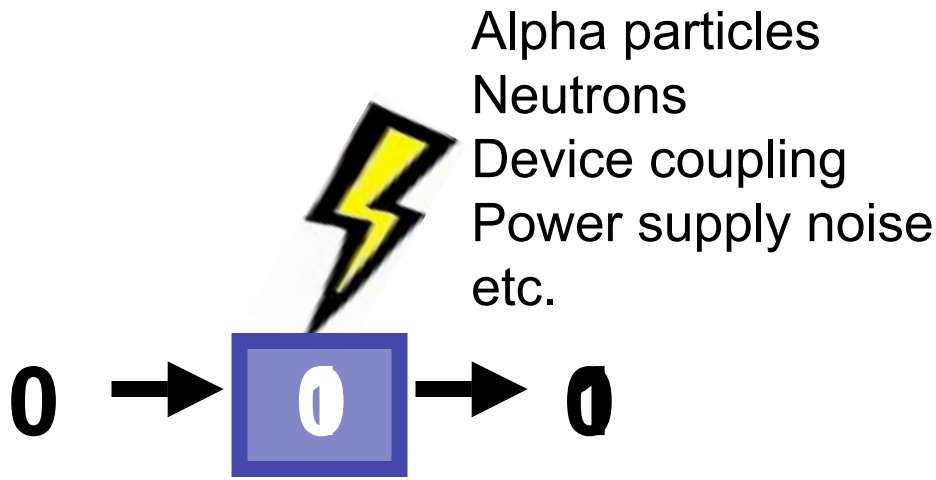
Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance

Outline

- Introduction and Motivation
- Software-centric Fault Detection
- Process-Level Redundancy
- Experimental Results
- Conclusion

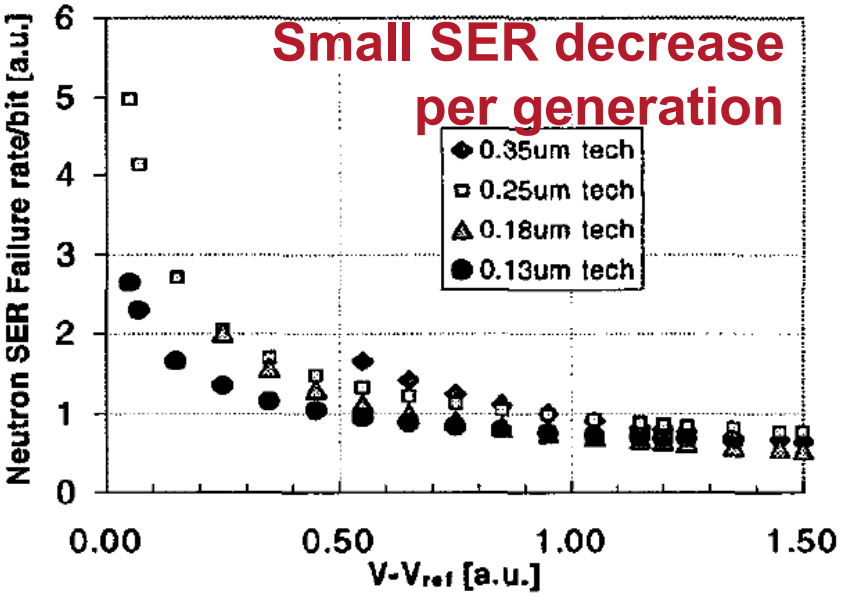
Transient Faults (Soft Errors)



Transient faults are already an issue!!

- **Sun Microsystems** [Baumann Rel. Notes 2002]
- **LANL ASC Q Supercomputer** [Michalak IEEE TDMR 2005]
- **…**

Predicted Soft Error Rates



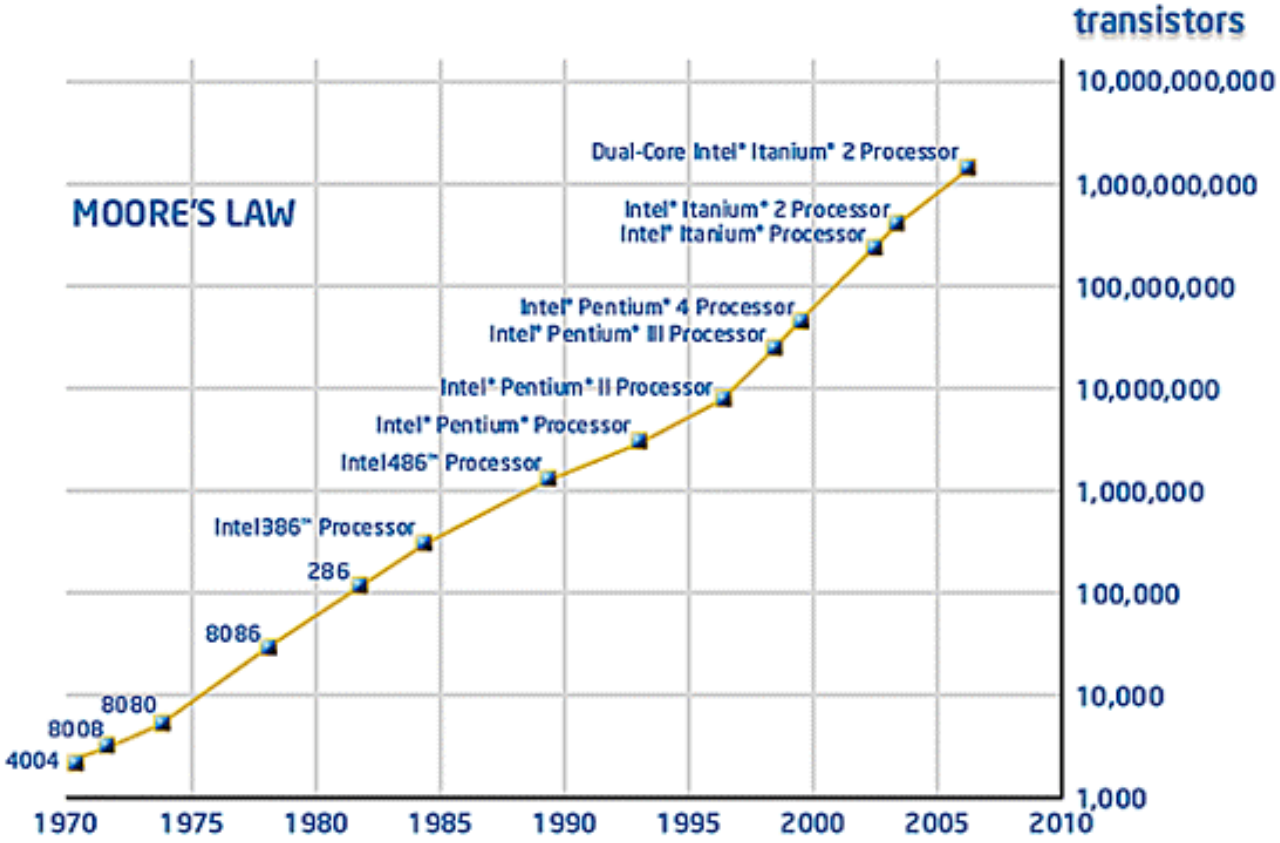
[Hareland VLSI 2001]

SER = **S**oft **E**rror **R**ate

"The neutron SER for a latch is likely to stay constant in the future process generations…"

[Karnik VLSI 2001]

Moore's Law Continues



[Source: www.intel.com/technology/mooreslaw]

Transient faults will be a significant issue in the design and execution of future microprocessors

Background

- One categorization: [Mukherjee HPCA 2005]
  I. Benign Fault
  II. Detected Unrecoverable Error (DUE)
    - False DUE- Detected fault would not have altered correctness
    - True DUE- Detected fault would have altered correctness
  III. Silent Data Corruption (SDC)

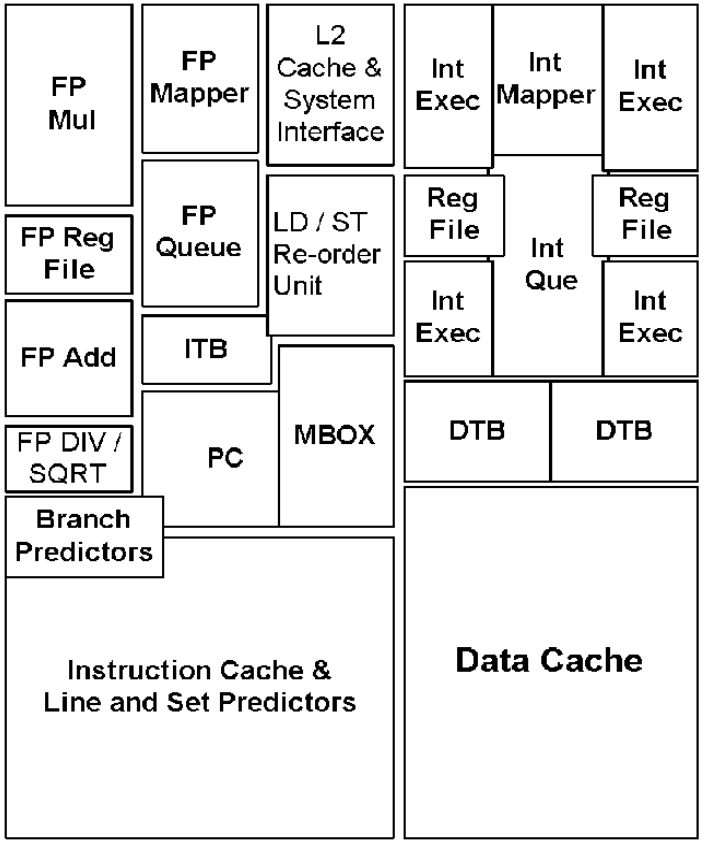
- Hardware Approaches
  – Specialized redundant hardware, redundant multi-threading
- Software Approaches
  – Compiler solutions: instruction duplication, control flow checking
  – Low-cost, flexible alternative but higher overhead

Architectural Vulnerability Factor

- ACE—Required for Architecturally Correct Execution
- AVF—Architectural Vulnerability Factor
  - Likelihood that a transient error in a structure will lead to a computational error

$$AVF = \frac{\sum_{b \in B} t_b}{|B| \times \Delta t}$$

- B is the set of all bits in some structure
- t_b is the total time that bit b is ACE
- $\triangle t$ is the total time of the execution



Benefits of Selective Protection

- Software control provides selective protection
  - Hybrid and Software systems enable software control
- Compiler/user/runtime system can make different decisions for different code regions
  - Programs, functions, or individual instructions
- Regions have different levels of natural fault resistance
- Output corrupting faults have different severity

original jpegenc output

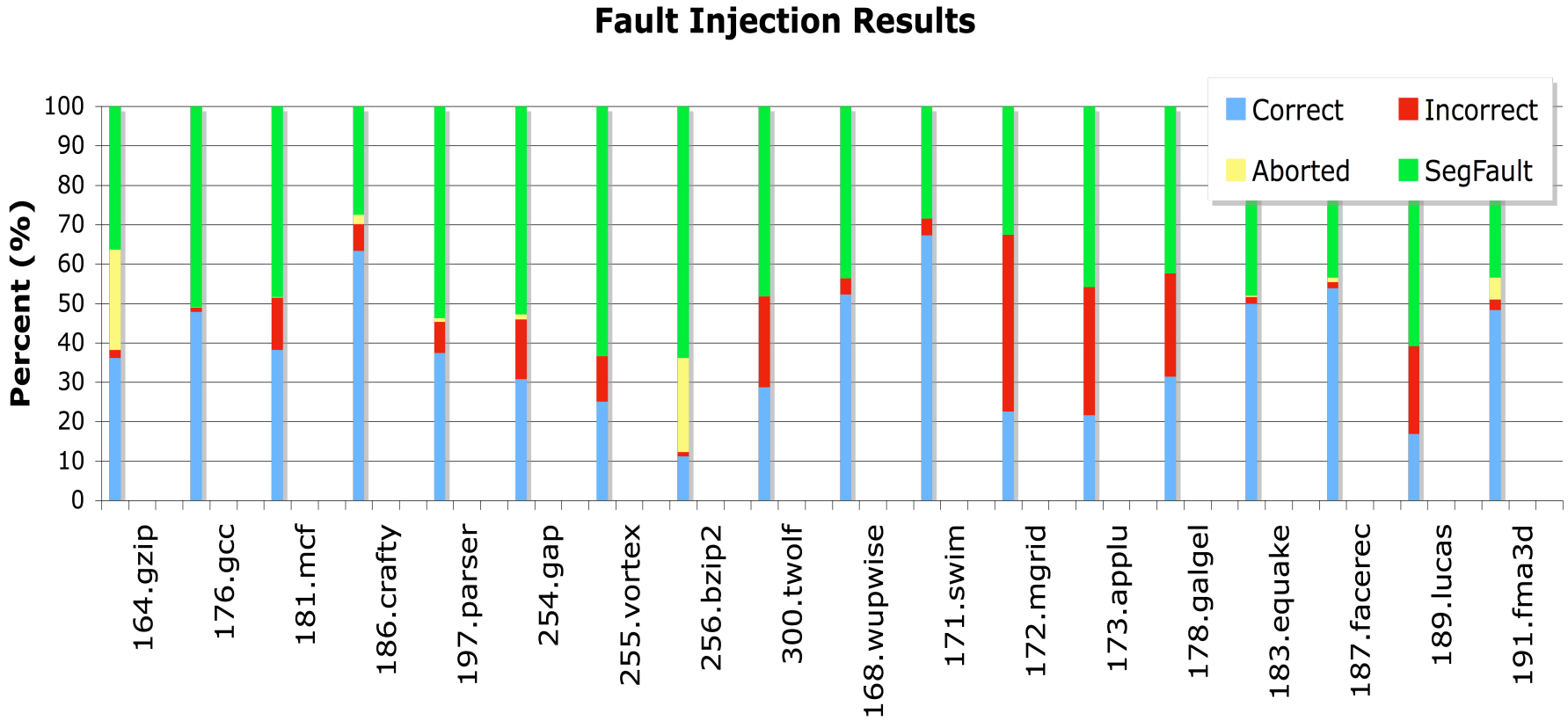
f a u l t y j p e g e n c o u t p u t

*faulty? jpegenc output**

- Selective protection can improve reliability

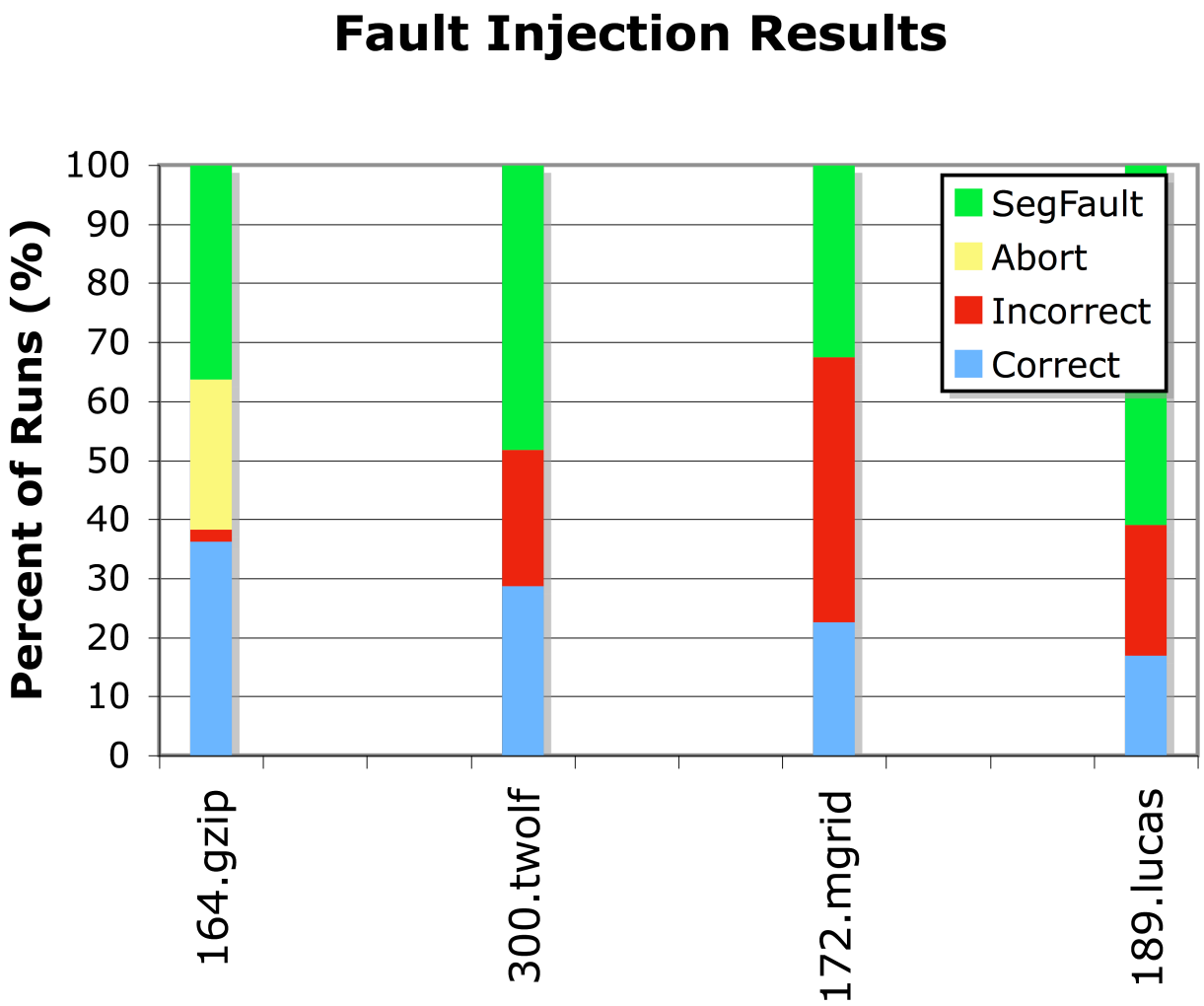
**Skadron (University of Virginia) Visual Vulnerability Spectrum*

Results of Injecting Errors

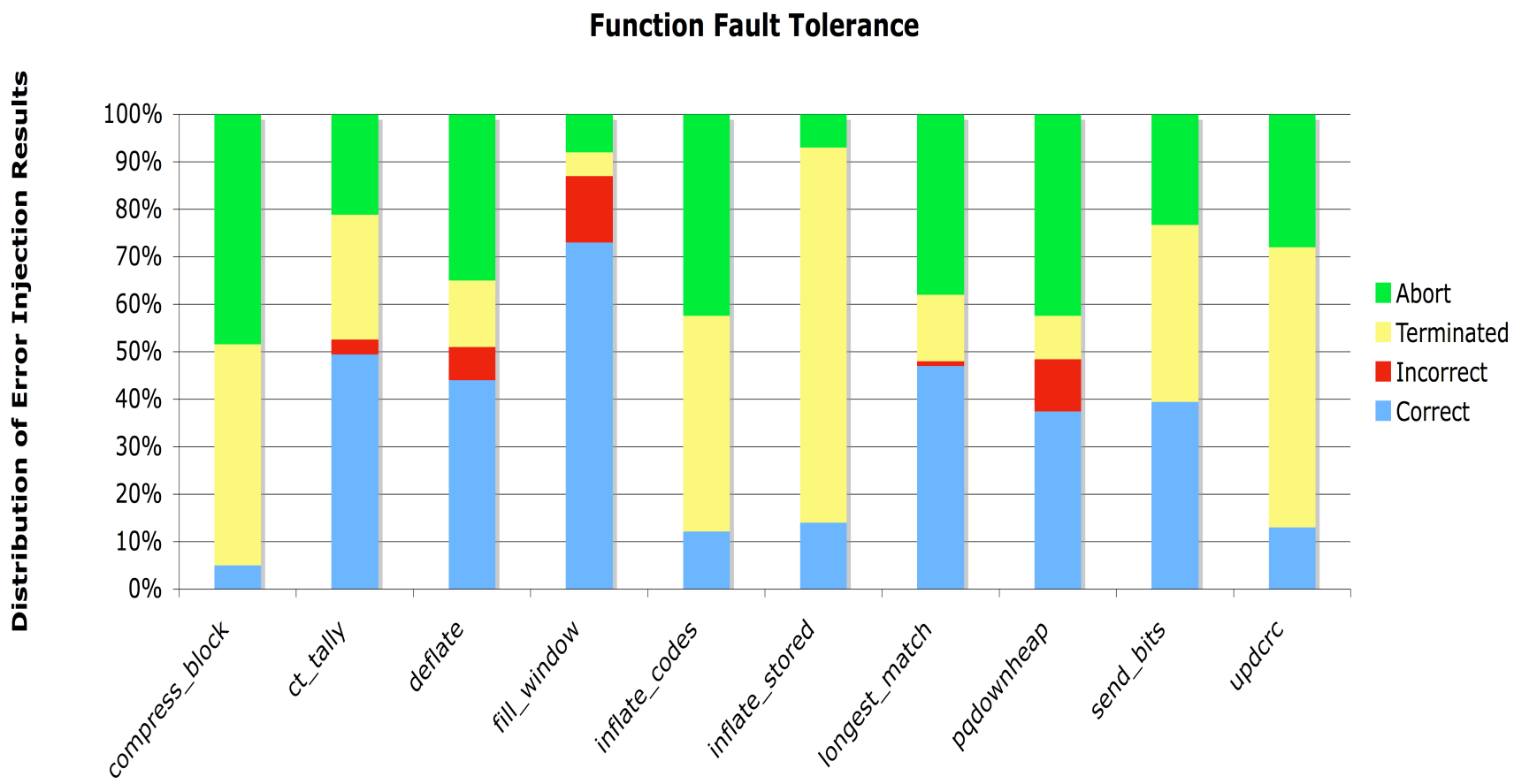


- Correct range: 25% to 60% (not impacted by error injection)
- Average correct execution 33%
- Application specific trends and behaviors

Application Specific Fault Injection Results

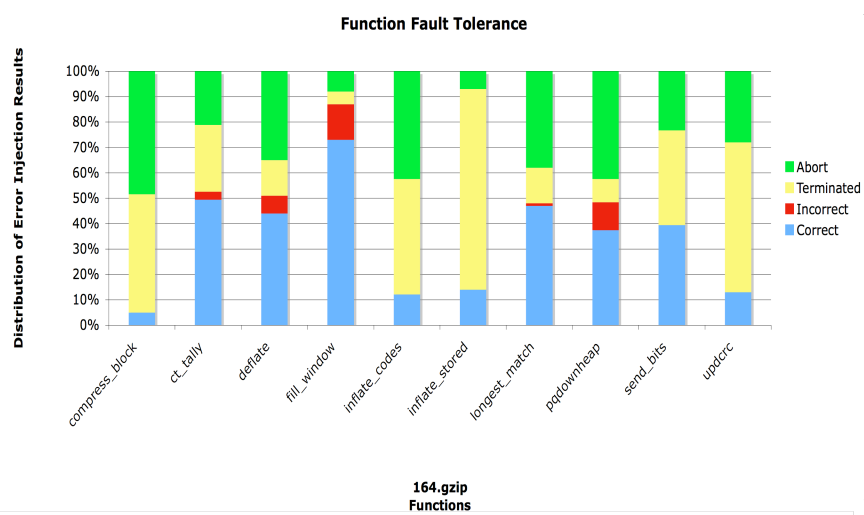


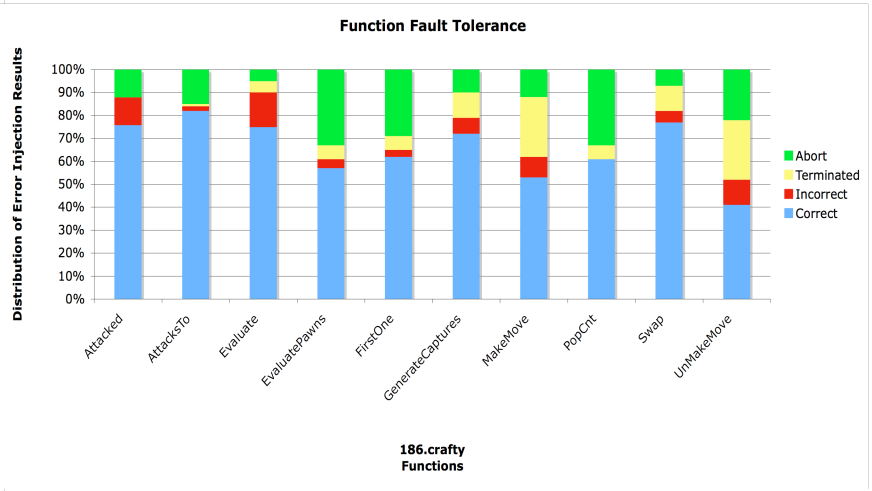
Function Analysis Experimental Results (164.gzip)

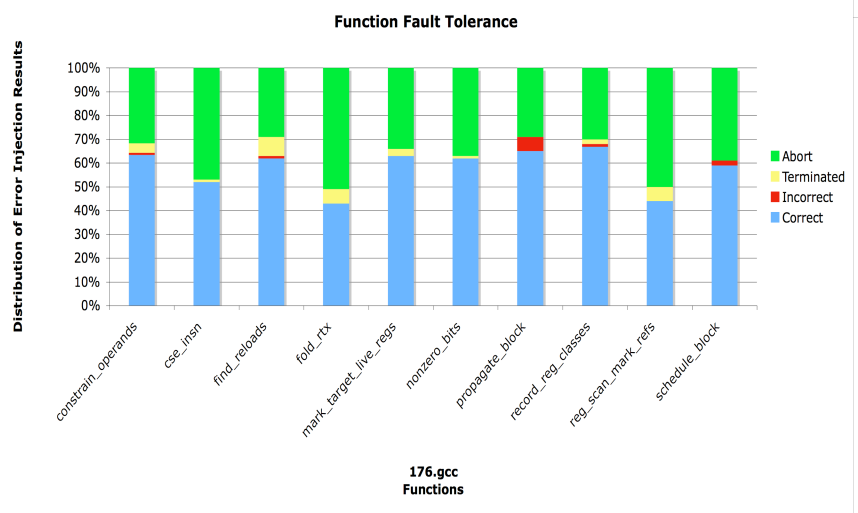


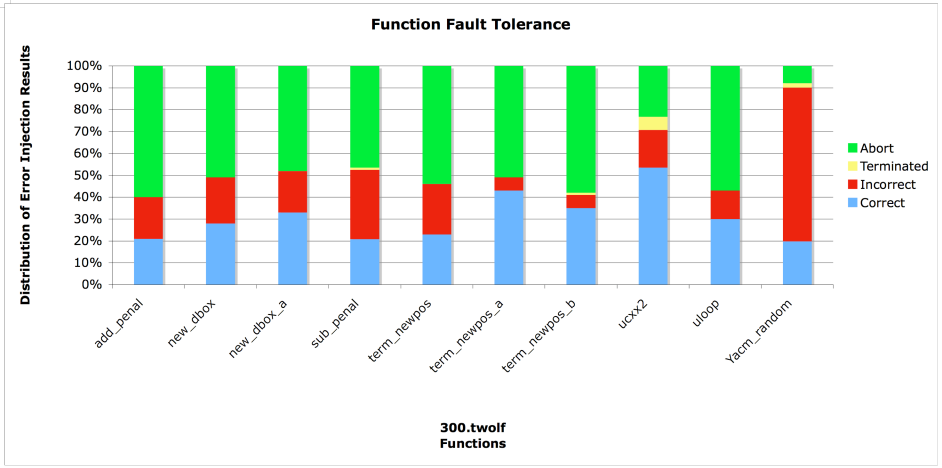
- Top executing function (by dynamic instruction count)
- Equal fault injections (1000) spread over each function's set of invocations

Function Analysis Experimental Results



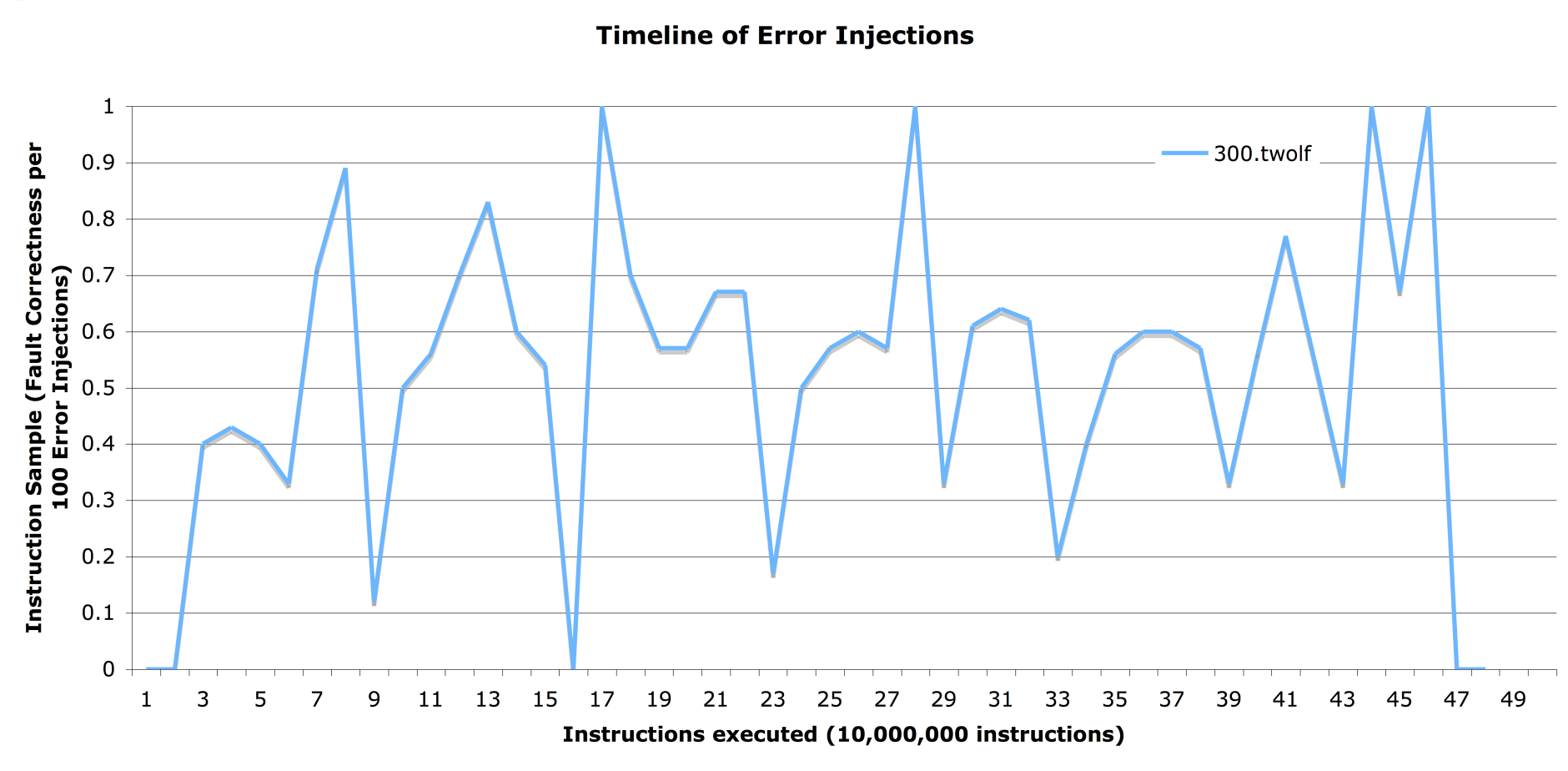






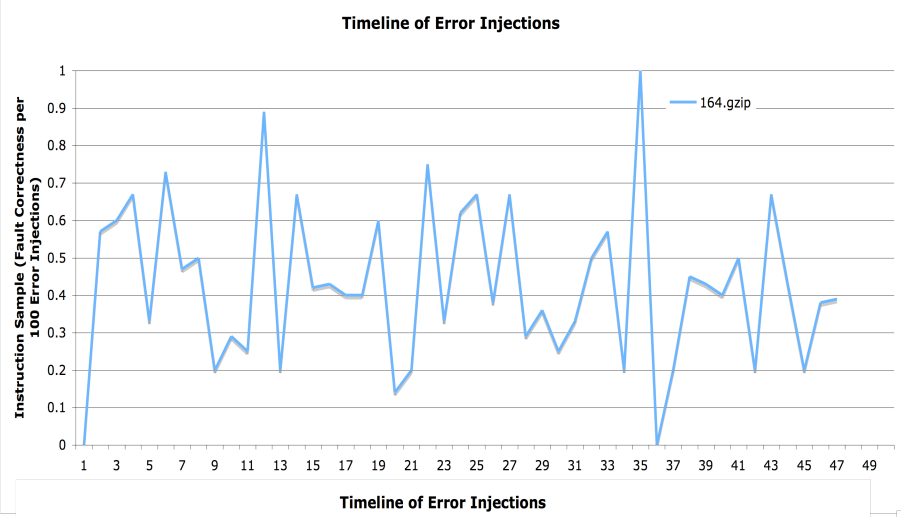
- Per-function (top 10 function executed per application)
- Compiler optimization can change 5-10% of CORRECT category
- Currently looking into correlation between compilation/optimization and transient fault tolerant nature of code

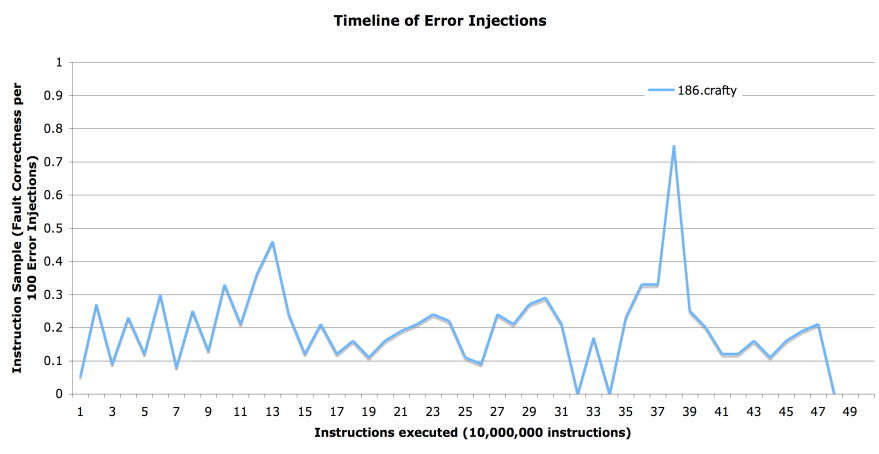
Fault Timeline Experimental Results

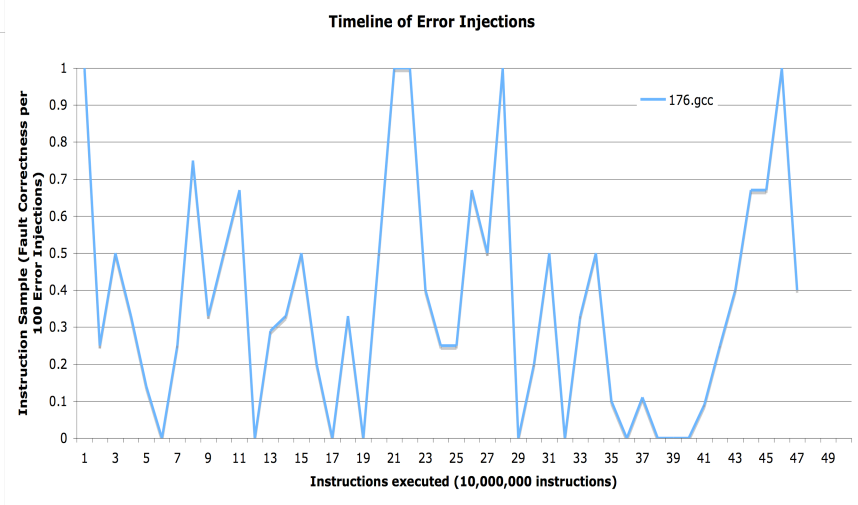


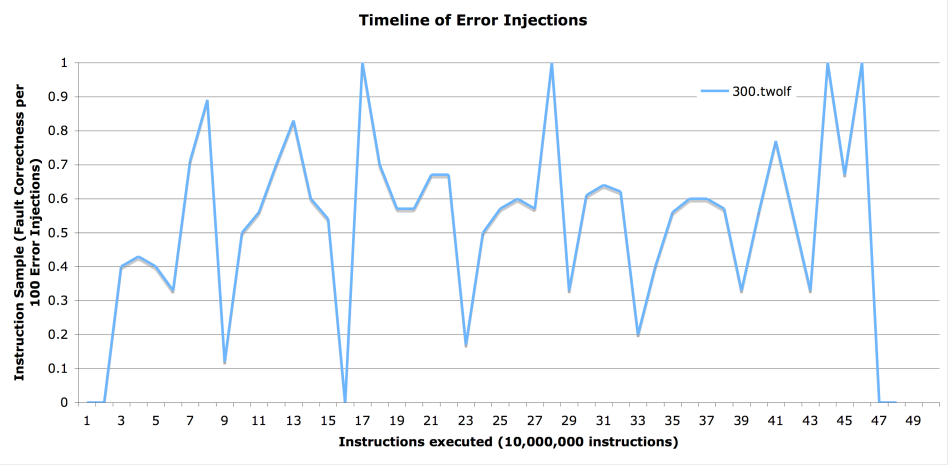
- Error injections into equal time segments
- Percentage of injections resulting in CORRECT execution

Fault Timeline Experimental Results



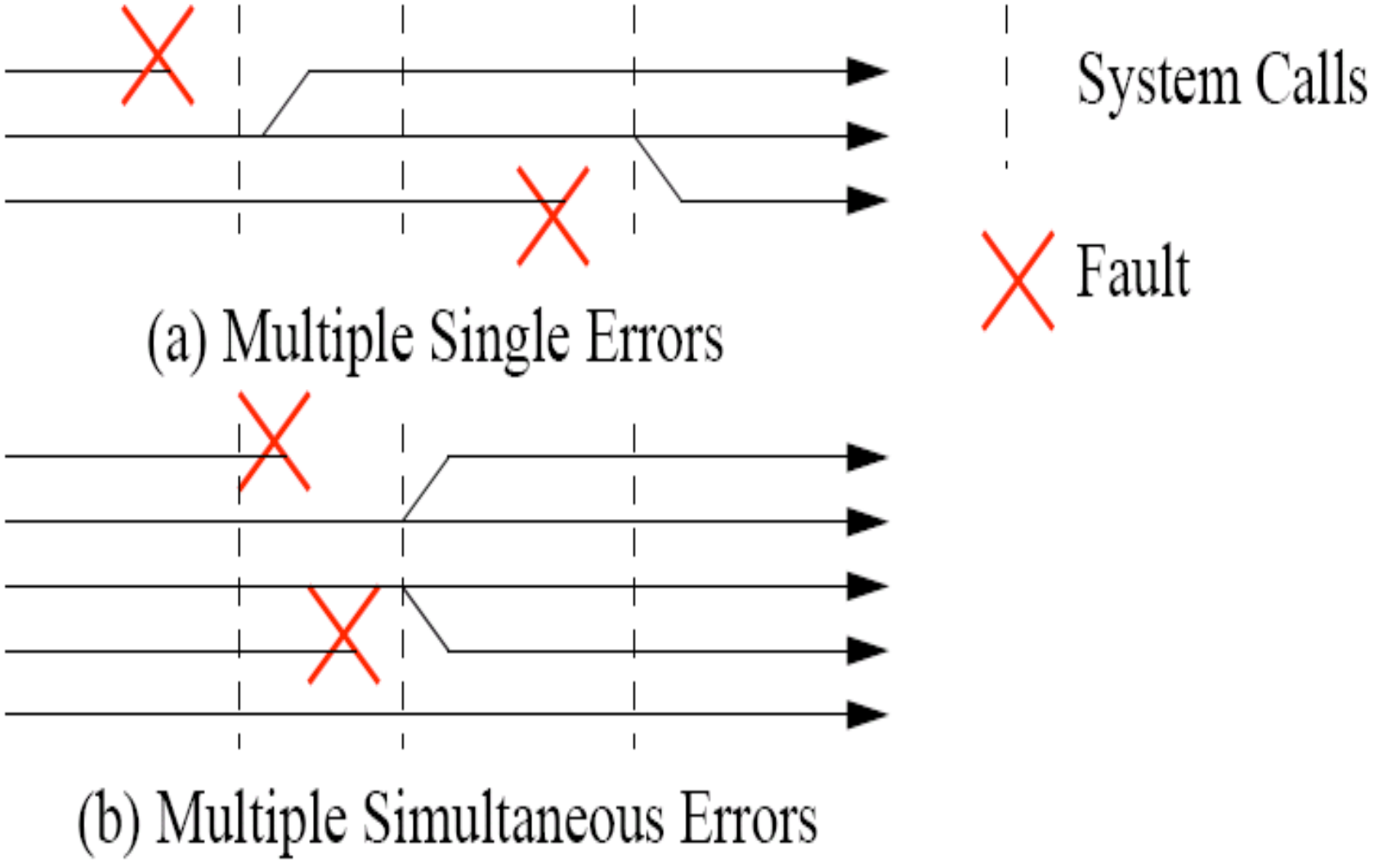






- Analysis of fault susceptibility over time
- Injection of errors in equal time segments of applications

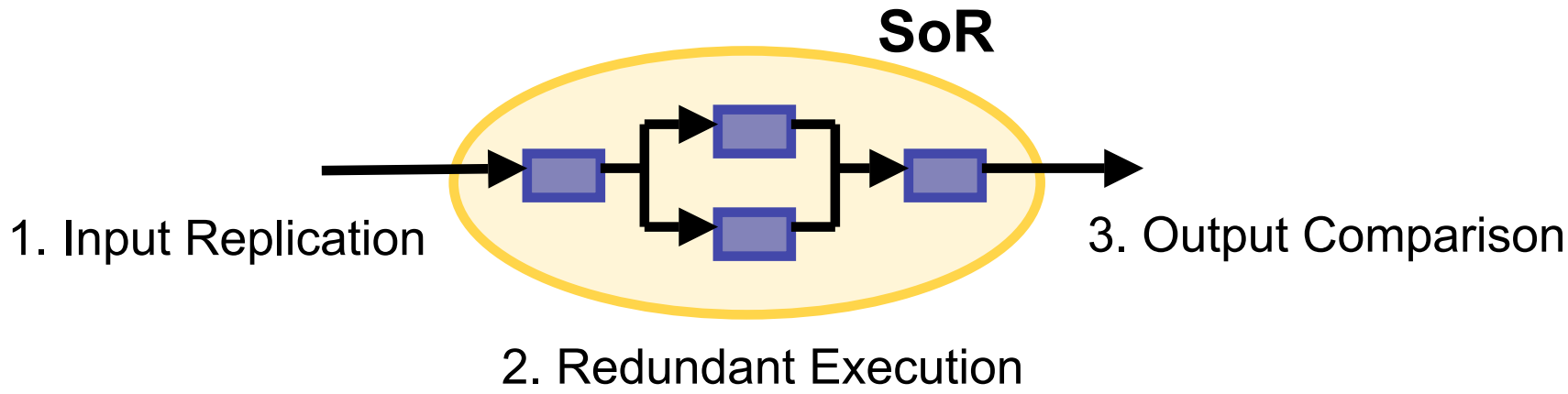
Process-level Redundancy



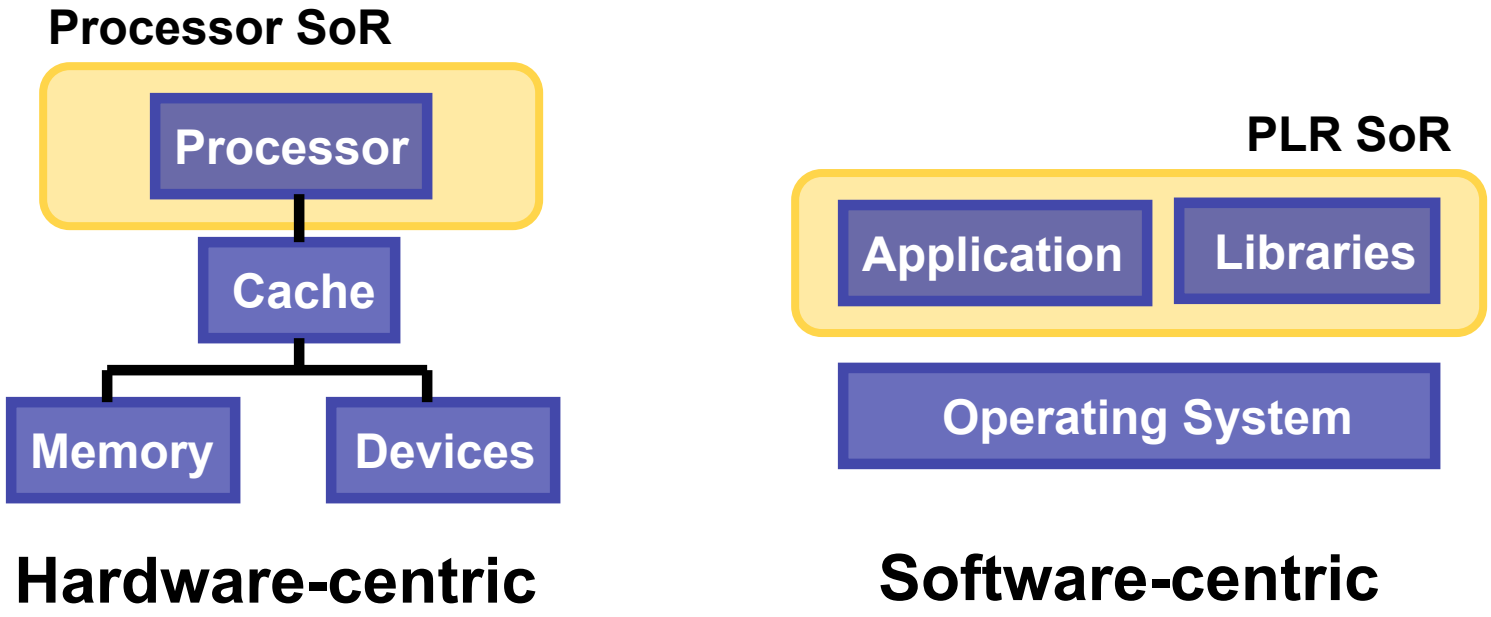
Goal

Use software to leverage available hardware parallelism for low-overhead transient fault tolerance.

Sphere of Replication (SoR)



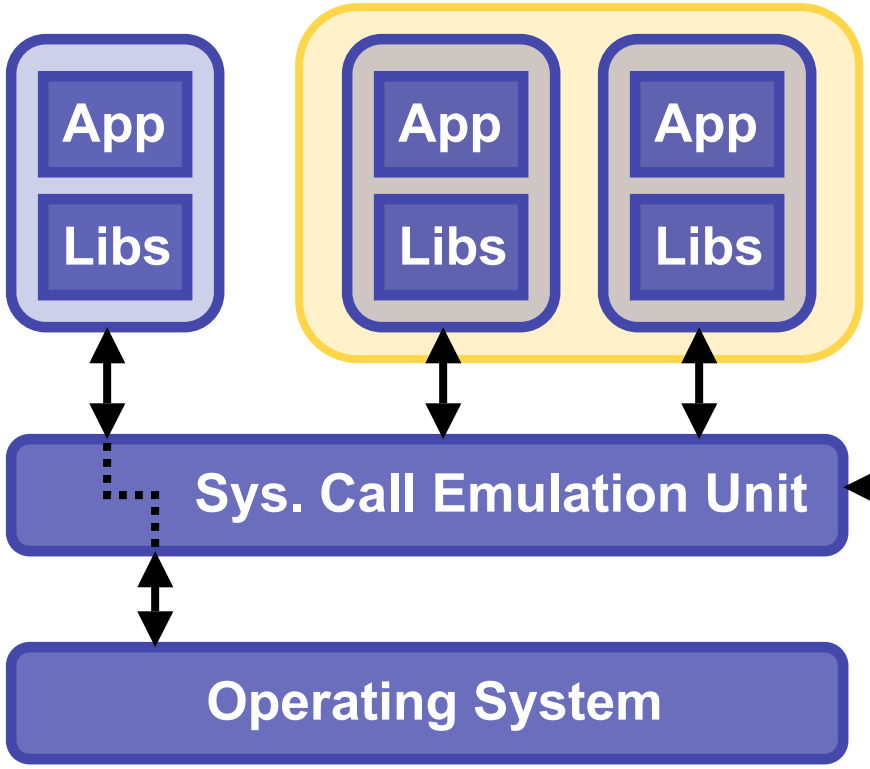
Software-centric Fault Detection



- Most previous approaches are hardware-centric
  - Even compiler approaches (e.g. EDDI, SWIFT)
- Software-centric able to leverage strengths of a software approach
  - Correctness is defined by software output
  - Ability to see larger scope effect of a fault
  - Ignore benign faults

Process-Level Redundancy (PLR)

Master Process
- only process allowed to perform system I/O



Slave Processes
- identical address space, file descriptors, etc.
- not allowed to perform system I/O

Watchdog Alarm
- occasionally a process will hang

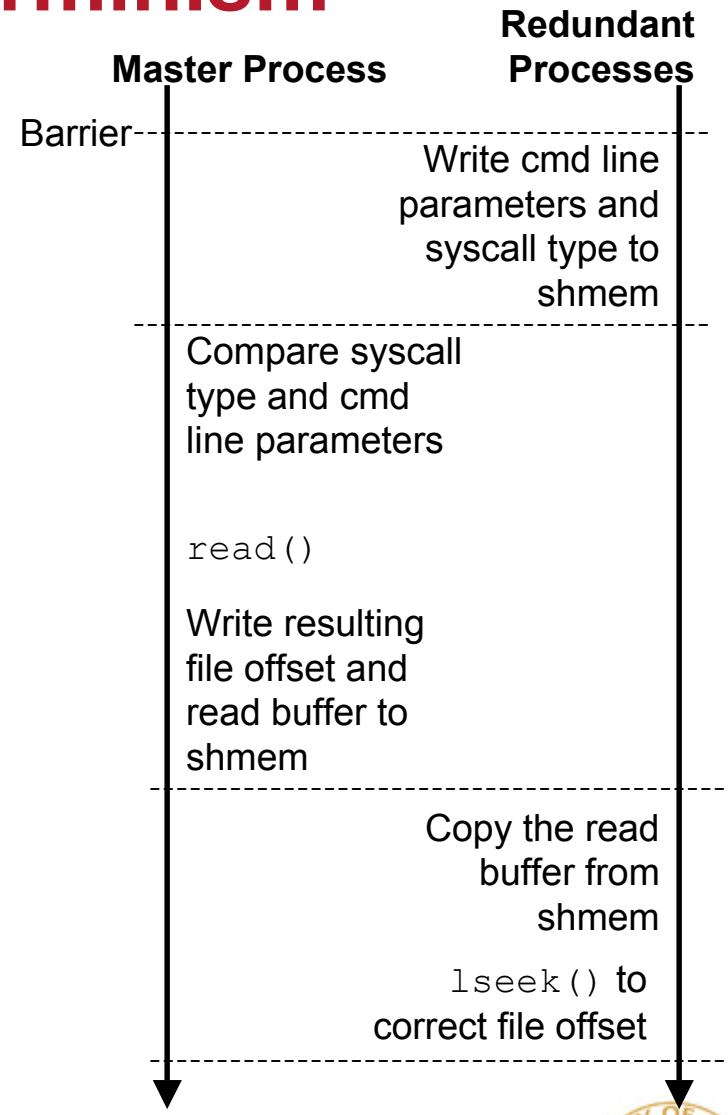
System Call Emulation Unit (SCEU)
- Enforces SoR with input replication and output comparison
- System call emulation for determinism
- Detects and recovers from transient faults

Enforcing SoR

- Input Replication
  - All read events: `read()`, `gettimeofday()`, `getrusage()`, etc.
  - Return value from all system calls
- Output Comparison
  - All write events: `write()`, `msync()`, etc.
  - System call parameters

Maintaining Determinism

- Master process executes system call
- Redundant processes emulate it
  - Ignore some: `rename()`, `unlink()`
  - Execute similar/altered system call
    - Identical address space: `mmap()`
    - Process-specific data: `open()`, `lseek()`
- Challenges
  - Shared memory
  - Asynchronous signals
  - Multi-threading



Example of handling a `read()` system call

Maintaining Determinism

- Master process executes system call
- Slave processes emulate it
  - Ignore some: `rename()`, `unlink()`
  - Execute similar/altered system call
    - Identical address space: `mmap()`
    - Process-specific data: `open()`, `lseek()`

- Challenges we do not handle yet
  - Shared memory
  - Asynchronous signals
  - Multi-threading

Fault Detection/Recovery

Type of Error	Detection Mechanism	Recovery Mechanism
Output Mismatch	Detected as a mismatch of compare buffers on an output comparison	Use majority vote ensure correct data exists, kill incorrect process, and `fork()` to create a new one
Program Failure	System call emulation unit registers signal handlers for SIGSEGV, SIGIOT, etc.	Re-create the dead process by forking one of existing processes
Timeout	Watchdog alarm times out	Determine the missing process and fork() to create a new one

- PLR supports detection/recovery from multiple faults by increasing number of redundant processes and scaling the majority vote logic

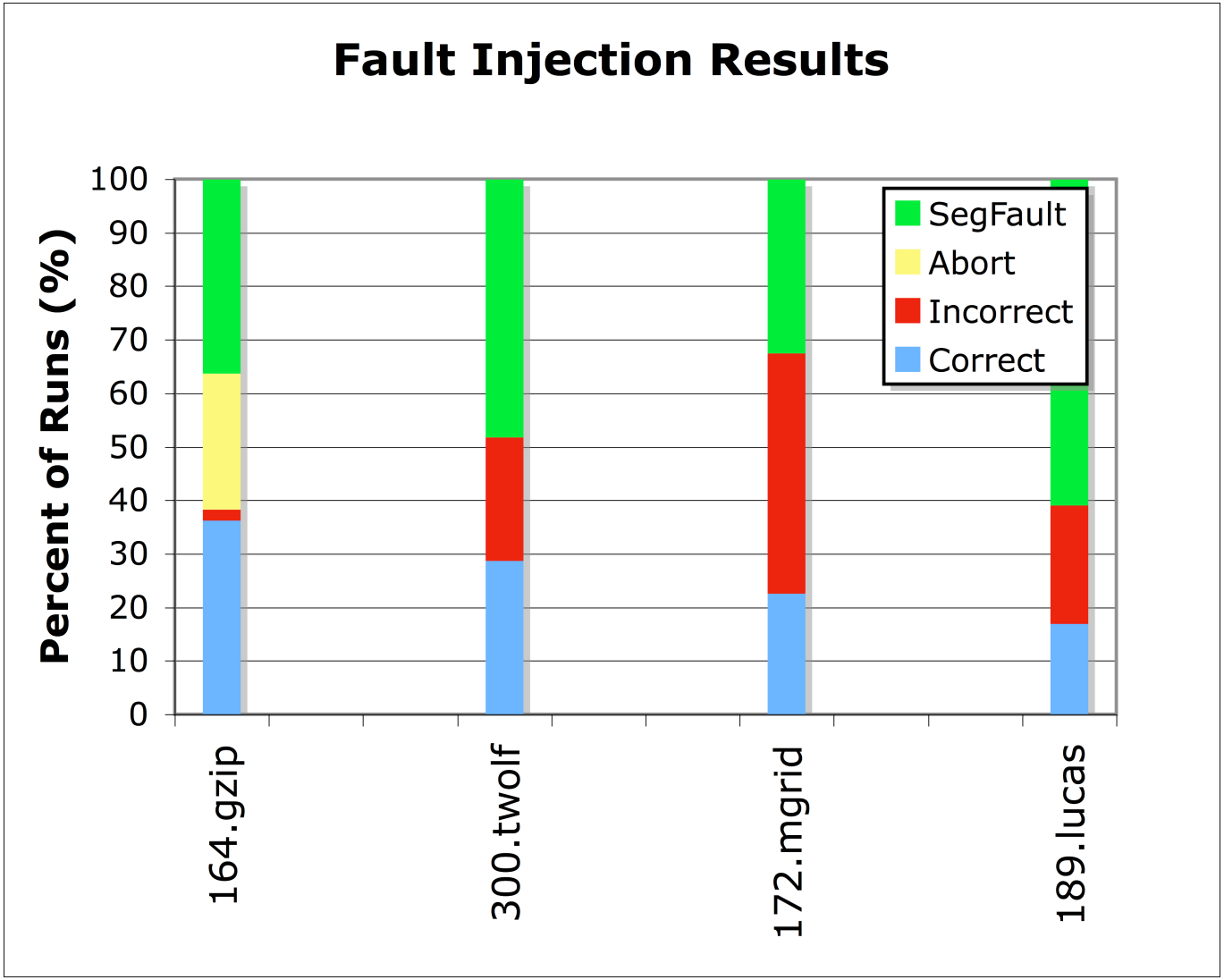
Windows of Vulnerability

- Fault during PLR execution
- Fault during execution of operating system

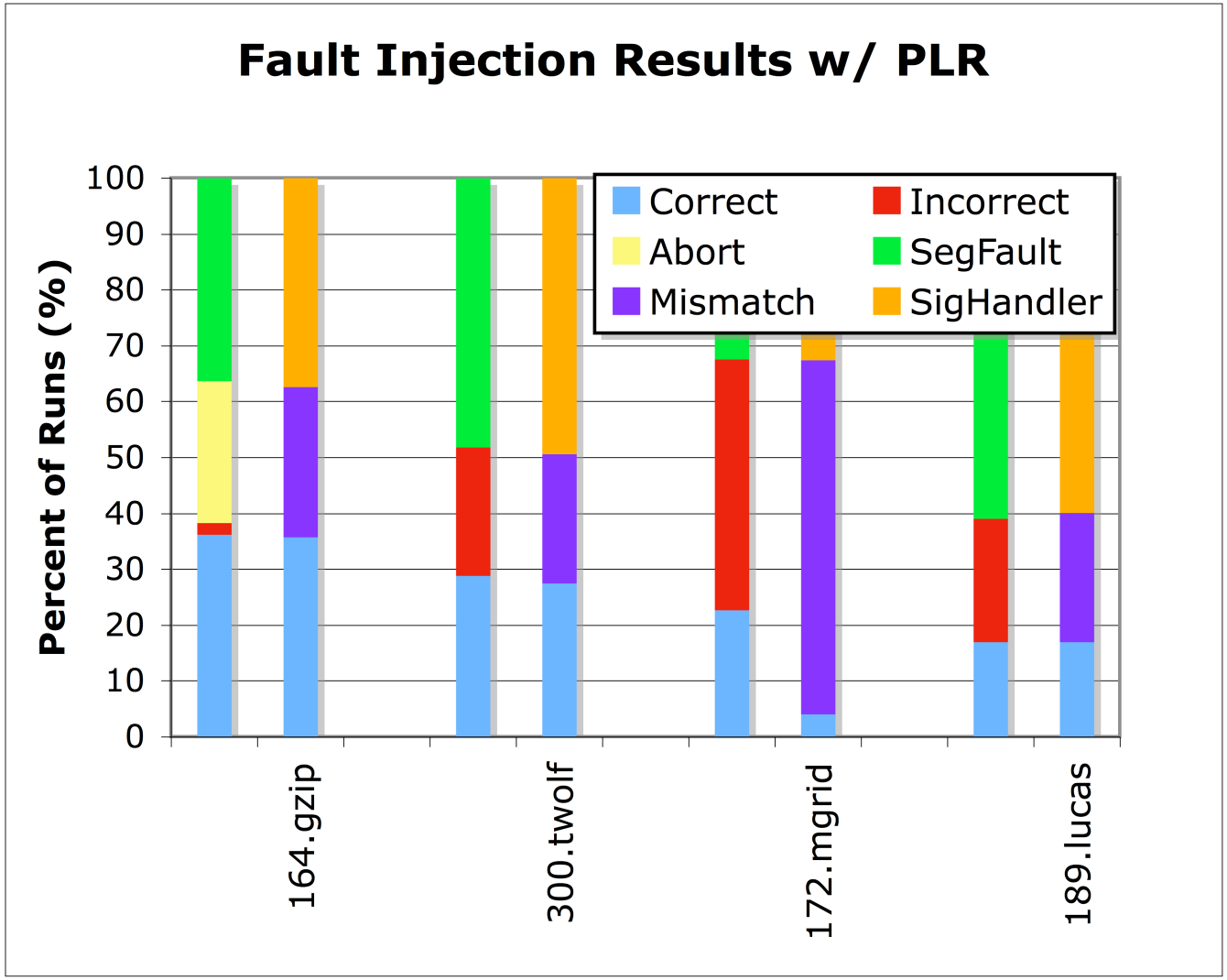
Experimental Methodology

- Set of *SPEC2000* benchmarks
- Prototype developed with Intel Pin dynamic binary instrumentation tool
  - Use Pin Probes API to intercept system calls
- Register Fault Injection (*SPEC2000* test inputs)
  - 1000 random test cases per benchmark generated from an instruction profile
    - Test case: a specific bit in a source/dest register in a particular instruction invocation
  - Insert fault with Pin `IARG_RETURN_REGS` instruction instrumentation
  - `specdiff` in SPEC2000 harness determines output correctness
- PLR Performance (*SPEC2000* ref inputs)
  - 4-way SMP, 3.00Ghz Intel Xeon MP 4096KB L3 cache, 6GB memory
  - Red Hat Enterprise Linux AS release 4

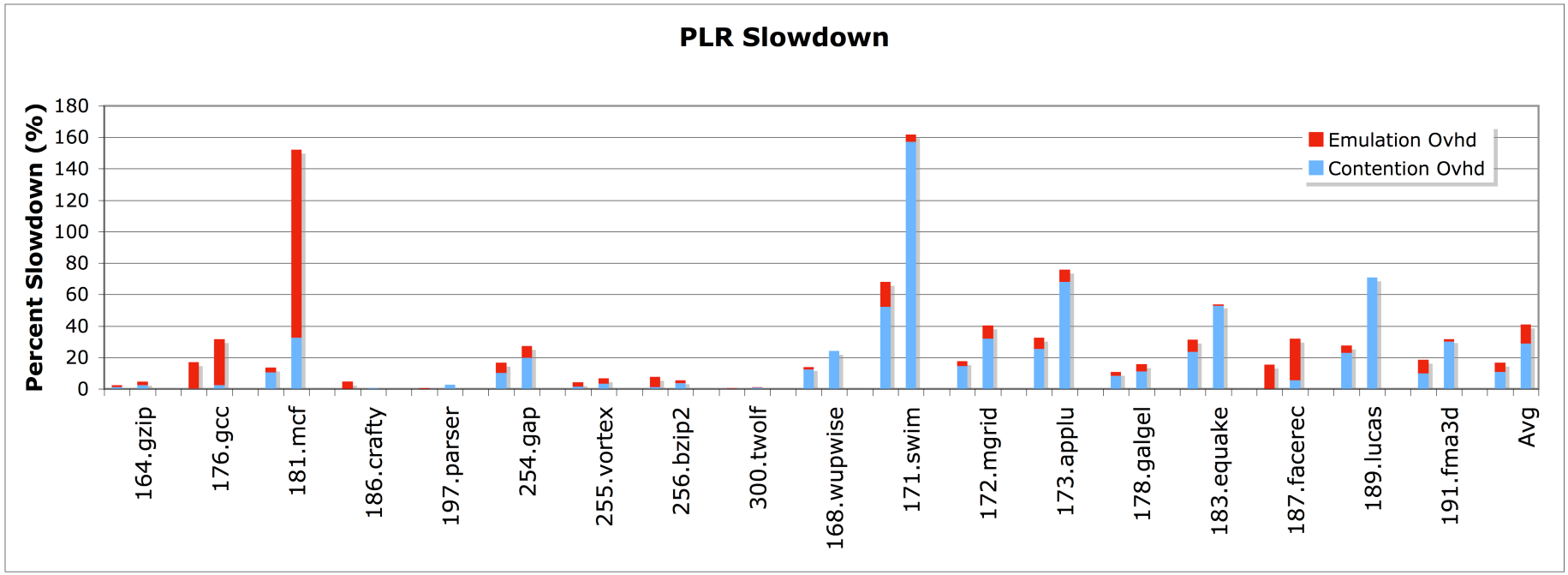
Fault Injection Results



Fault Injection Results w/ PLR



PLR Performance



- As a comparison: SWIFT is .4x slowdown for detection and 2x slowdown for detection+recovery

- **Contention Overhead**: Overhead of running multiple processes using shared resources (caches, bus, etc)
- **Emulation Overhead**: Overhead of PLR synchronization, shared memory transfers, etc.

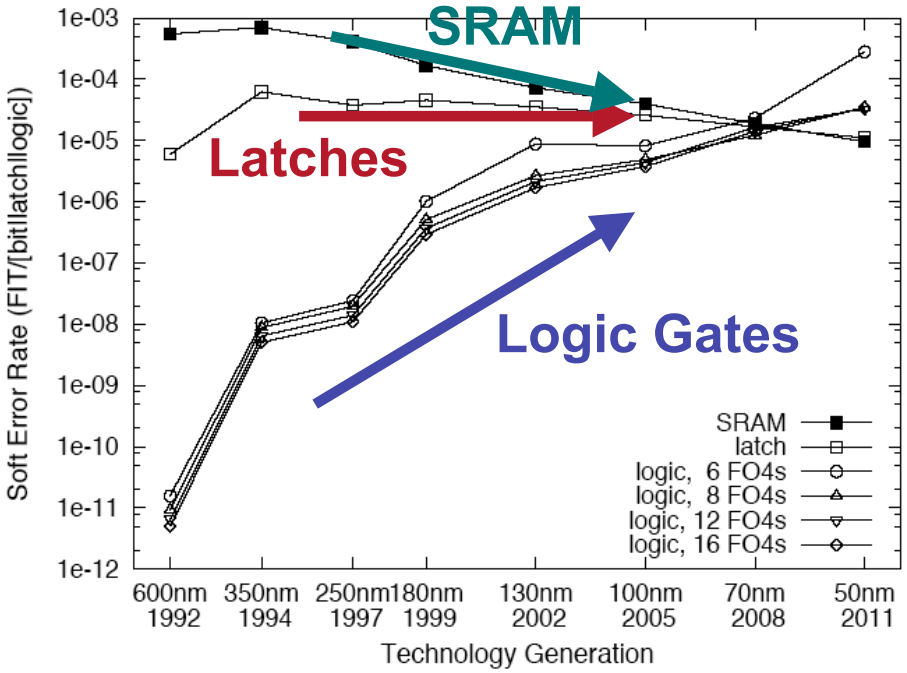
Conclusion

- Present a software-implemented transient fault tolerance technique to utilize general-purpose hardware with multiple cores
- Differentiate between hardware-centric and software-centric fault detection models
  - Show how software-centric can be effective in ignoring benign faults
- Prototype PLR system runs on a 4-way SMP machine with 16.9% overhead for detection and 41.1% overhead with recovery

Questions?

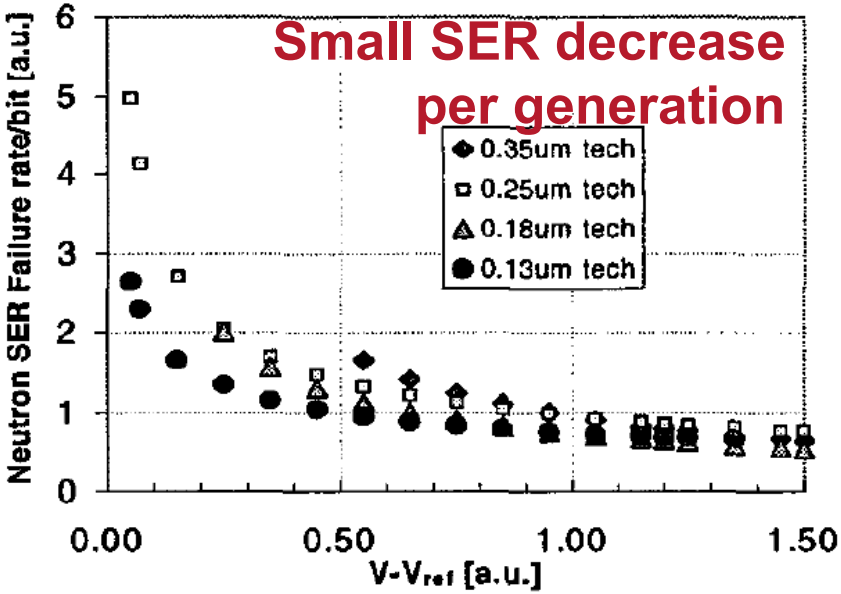
Extra Slides

Predicted Soft Error Rates



[Shivakumar DSN 2002]

SER = Soft Error Rate

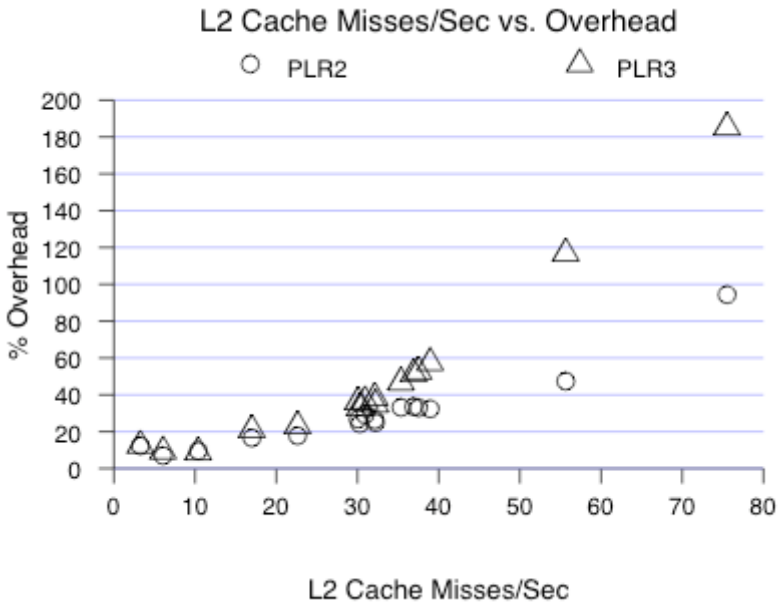


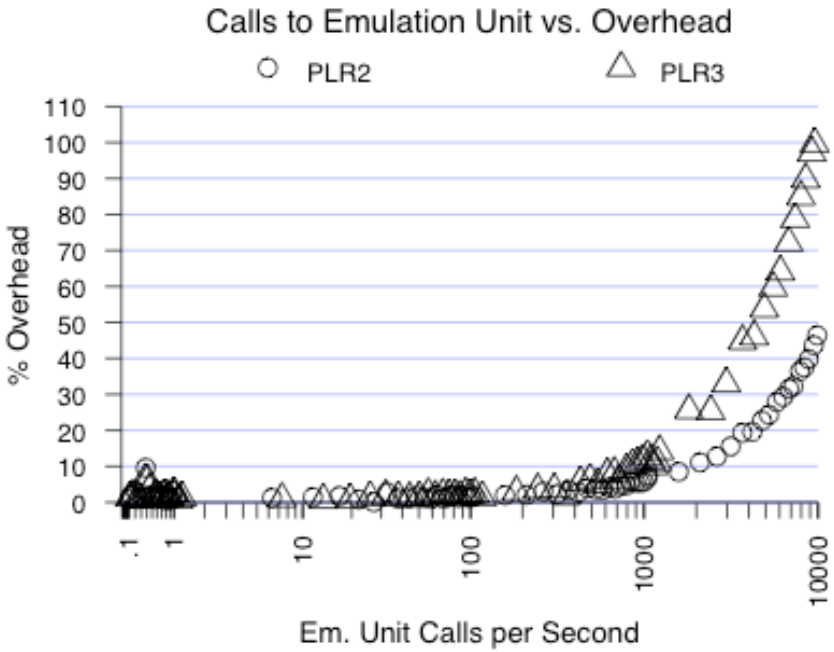
[Hareland VLSI 2001]

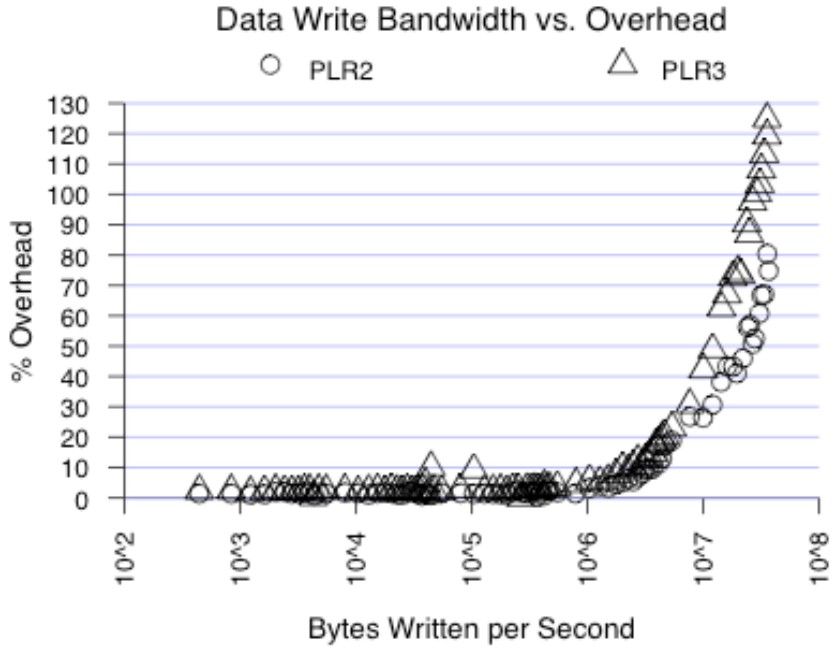
"The neutron SER for a latch is likely to stay constant in the future process generations…"

[Karnik VLSI 2001]

Overhead


L2 Cache Misses/Sec vs. Overhead
PLR2
PLR3
% Overhead
0
20
40
60
80
100
120
140
160
180
200
0
10
20
30
40
50
60
70
80
L2 Cache Misses/Sec


Calls to Emulation Unit vs. Overhead
PLR2
PLR3
% Overhead
0
10
20
30
40
50
60
70
80
90
100
110
.1
1
10
100
1000
10000
Em. Unit Calls per Second


Data Write Bandwidth vs. Overhead
PLR2
PLR3
% Overhead
0
10
20
30
40
50
60
70
80
90
100
110
120
130
10^2
10^3
10^4
10^5
10^6
10^7
10^8
Bytes Written per Second