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)

- Alpha particles
- Neutrons
- Device coupling
- Power supply noise
- etc.

Transient faults are already an issue!!

- Sun Microsystems [Baumann Rel. Notes 2002]
- LANL ASC Q Supercomputer [Michalak IEEE TDMR 2005]
- ...
Predicted Soft Error Rates

"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

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

- \( B \) is the set of all bits in some structure
- \( t_b \) is the total time that bit \( b \) is ACE
- \( \Delta t \) 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

- Selective protection can improve reliability
Results of Injecting Errors

Fault Injection Results

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

Results

Fault Injection Results

<table>
<thead>
<tr>
<th>164.gzip</th>
<th>300.twolf</th>
<th>172.mgrid</th>
<th>189.lucas</th>
</tr>
</thead>
<tbody>
<tr>
<td>SegFault</td>
<td>Abort</td>
<td>Incorrect</td>
<td>Correct</td>
</tr>
</tbody>
</table>
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

DRACO Architecture Research Group. DSN, Edinburgh UK, 06.25.2007
Fault Timeline Experimental Results

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

DRACO Architecture Research Group. DSN, Edinburgh UK, 06.25.2007
Process-level Redundancy

(a) Multiple Single Errors

(b) Multiple Simultaneous Errors

System Calls
Fault
Goal

Use software to leverage available hardware parallelism for low-overhead transient fault tolerance.
Sphere of Replication (SoR)

1. Input Replication
2. Redundant Execution
3. Output Comparison
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

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: \texttt{rename()}, \texttt{unlink()}
  - Execute similar/altered system call
    - Identical address space: \texttt{mmap()}
    - Process-specific data: \texttt{open()}, \texttt{lseek()}

- Challenges
  - Shared memory
  - Asynchronous signals
  - Multi-threading

Example of handling a \texttt{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

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

- 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

Percent of Runs (%)

164.gzip 300.twolf 172.mgrid 189.lucas

Segments Faulted
Abort
Incorrect
Correct
Fault Injection Results w/ PLR

Fault Injection Results w/ PLR

Percent of Runs (%)

Correct | Incorrect | Abort | SegFault | Mismatch | SigHandler

164.gzip | 300.twolf | 172.mgrid | 189.lucas

DRACO Architecture Research Group. DSN, Edinburgh UK, 06.25.2007
PLR Performance

As a comparison: SWIFT is \(0.4\times\) slowdown for detection and \(2\times\) 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

![Graph showing soft error rates for different technology generations for SRAM, latches, and logic gates.]

**SER = Soft Error Rate**

Small SER decrease per generation

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

[Hareland VLSI 2001]

[Shivakumar DSN 2002]
Overhead

L2 Cache Misses/Sec vs. Overhead

- PLR2
- PLR3

% Overhead

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

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

Bytes Written per Second

DRACO Architecture Research Group