On Making Transactional Applications Resilient to Data Corruption Faults

Mohamed Mohamedin, Roberto Palmieri, and Binoy Ravindran
Virginia Tech
USA
{mohamedin,robertop,binoy}@vt.edu

NCA'14
Transient Faults Problem

- Faults
  - Permanent
  - Transient

- Transient faults change the behavior of an application and may or may not crash the application.
  - Software bugs
  - Hardware errors (e.g., soft errors)

- Transient faults can cause data corruption
  - Wrong results
  - Data loss
  - Propagation of corruption

- Major outage of Amazon S3 service
  - 7 hour outage to understand and fix the problem
What are Soft-errors?

- Transient faults that may happen anytime during application execution
- Caused by physical phenomena (e.g., cosmic particle strikes, electric noise)
- E.g., Soft-error can cause a single bit in a CPU register to flip which may cause a transient fault.
Do soft-errors represent a problem?

- Soft-errors are:
  - Random: Can occur anytime
  - Undetectable: No hardware interrupt is triggered
  - Corrupting: Can silently corrupt program data or crash the program
Soft-errors in multicore architectures

- Soft-errors rate is growing in the current and emerging multicore architectures
  - Smaller transistors (e.g., Intel Haswell uses 22nm)
  - More components on same chip (e.g., more cores)

#### Graphs

- **Soft-error failure-in-time of a chip**
- **SER as a function of the number of chips**
Soft-errors effect

CPU mathematical operation

101011101011
+ 100010101001
= 1001110010100
Soft-errors effect

CPU mathematical operation

\[
\begin{align*}
101011101011 & \text{ + } 100010101001 = 1001110010100
\end{align*}
\]

But if a soft-error happened

\[
\begin{align*}
101011101011 & \text{ + } 100010101001 \text{ (\text{\textcolor{red}{\textasteriskcentered}})} = 1001110010100
\end{align*}
\]
Soft-errors effect

CPU mathematical operation

\[
\begin{align*}
101011101011 + 100010101001 &= 1001110010100
\end{align*}
\]

But if a soft-error happened

\[
\begin{align*}
101011101011 + 000010101001 &= 1001110010100
\end{align*}
\]
Soft-errors effect

CPU mathematical operation

\[
\begin{align*}
101011101011 + 100010101001 &= 1001110010100 \\
100010101001 + 000010101001 &= 0101110010100
\end{align*}
\]

But if a soft-error happened

\[
\begin{align*}
101011101011 + 000010101001 &= 0101110010100
\end{align*}
\]
Soft-errors effect

Memory store
e.g., mov [address], ax

Address
10001

ax
110101

Memory
110101
Soft-errors effect

Memory store  
e.g., mov [address], ax

But if a soft-error happened
Soft-errors effect

Memory store
e.g., mov [address], ax

But if a soft-error happened

Address 10001

ax 110101

Memory

110101

Address 11001

ax 110101

Memory

000001
Soft-errors effect

Memory store
e.g., mov [address], ax

But if a soft-error happened

Overwrite
Soft-errors effect

Memory store
e.g., mov [address], ax

But if a soft-error happened

Overwrite
Wrong Value

Address: 10001
ax: 110101
Memory: 110101

Address: 11001
ax: 110101
Memory: 000001

Address: 10001
ax: 110101
Memory: 110101

Address: 11001
ax: 110101
Memory: 110101
Soft-errors effect

Memory store

- e.g., mov [address], ax

But if a soft-error happened

Address 000001

Memory

Wrong Value 000001

Memory Protection

Exception

Address 11001

Memory

ax 110101

110101

ax 110101

110101
How to tolerate transient faults?

- Restart the application!
  - It may not crash!
  - Not suitable for critical business applications
    - We need to maintain availability/reliability constraints

- Checkpointing
  - Depends on error-detection accuracy
  - How many check points to keep
  - Time to restore a check point

- Encoding
  - Overhead & limited

- Assertions/invariants
  - Not accurate
How to tolerate transient faults? (2)

- Replication
  - Permanent faults or Byzantine faults
  - Designed for distributed system
  - Several sources of overhead
    - Wrapping a request into network message
    - Totally ordering these messages
    - In-order execution
  - Computational resources are partitioned into replicas

- Hardware
  - High end systems
  - Expensive
Our Goal

- Develop a low-intrusive technique that has:
  - Good performance
  - Less synchronization
  - Less bandwidth
  - Guarantee both safety and liveness
Target Applications

- Transactional applications
  - Very important class of applications
  - Based on transactions
    - Manage the program state using transactions
  - Examples:
    - Banking systems
    - Automatic teller systems
    - Stock trading
    - Application web servers
  - Need to be protected
    - Requires safety and liveness
Proposed Solution (SoftX)

- Speculative execution of the same transaction on different cores in parallel
  - Compare outcomes using a dedicated *committer threads*
    - Low synchronization overhead
  - Without partitioning computational resources
    - Cores are reused
    - Single copy of the memory
  - Without ordering transactions
- Implemented using Software Transactional Memory abstraction
- SoftX inherits both the checkpointing and replication advantages
SoftX Overview
Assumptions

- Data in memory is not replicated
  - We rely on memory error detection and correction (e.g., ECC)
- Only committer threads can write to shared data
  - Speculative threads have read-only access to shared data
- Committer threads keep an undo log
  - Can recover from an error during write operations
- Works on a single machine
  - Cannot tolerate a machine crash or HW permanent error
  - Other techniques can be used in parallel
    - E.g., Asynchronous checkpointing to a stable storage
- No non-deterministic operations (e.g., random, getTime) inside a transaction
SoftX Design

- STM + Resilient to transient faults
SoftX Design (2)

- Starting a transaction by forking a group of threads

Diagram:

Application

Thread

Forking

Group
Speculative threads must observe the same initial state
  - Committer threads pause all commit operations
Each speculative thread executes the transaction independently.
At commit time, send to committer threads and wait for their decision.
Committer threads cooperatively decide if the transaction has no conflicts/errors.
Committer Threads

- A group of threads responsible for detecting faults (voter) and conflicts between transactions.
- Its main purpose is to reduce synchronization overhead and maintain fault tolerance.
  - Reduces cache misses and invalidation
- They can also tolerate faults during commit procedure.
- Independently:
  - Validate each read-set
  - Compare write-sets
- Majority are valid and match → commit, otherwise, restart
- Committer threads decisions also must match
- One thread do write back and others confirm write is correct
  - Undo log is used to recover in case of a fault
Speculative threads

- Act as a group
  - An abort in one thread, trigger an abort for the entire group
- Number of threads is related to degree of resiliency
  - 2 threads: detect a fault but cannot recover. The transaction must restart
  - $2f + 1$ threads: Can recover up to $f$ faults without restarting the transaction
Applicability

- We don't target deterministic software bugs
  - The same behavior on all replicas
  - Diversity?
- We target HW transient faults, random SW bugs
Evaluation

- SoftX is implemented in C++ in RSTM library
  - Bus-based (x86): 48-core AMD Opteron machine
  - Message-passing: 36-core Tilera TILE-Gx co-processor

- Competitors:
  - Non transient fault tolerant STM: NOrec
  - State Machine-Like Transactional Replication (SMR)
  - Byzantine Fault Tolerant system (PBFT)

- Benchmarks:
  - Bank
  - List
  - TPC-C
Evaluation: x86

- TPC-C
Evaluation: x86 (2)

- List

Graph showing the performance of different protocols (NOrec, SMR, PBFT, SoftX) in terms of transactions per second (tx/sec) against the number of application threads/groups. The graph indicates how the performance scales with increasing thread counts.
Evaluation: Tilera

- TPC-C

![Graph showing TPC-C performance for different application threads/groups. The graph plots K tx/sec against Application Threads/Groups. The lines represent NOrec, SMR, PBFT, and SoftX, respectively. The graph shows an increasing trend as the number of application threads/groups increases.](image-url)
Evaluation Summary

- SoftX overhead is reasonable
  - High contention (e.g., List)
  - Long transactions (e.g., TPC-C)
- Message-passing reduces synchronization and communication overhead
  - SoftX has the lowest overhead compared to SMR and PBFT
- SoftX performs better than replication-based approaches
  - Requires less data transfer between system components
Conclusion

- SoftX adds fault tolerance to concurrency control protocols
  - Reasonable overhead
  - Better than optimized SMR
  - Suitable for both shared bus and message passing architectures
Questions?

Research project’s web-site: www.hyflow.org