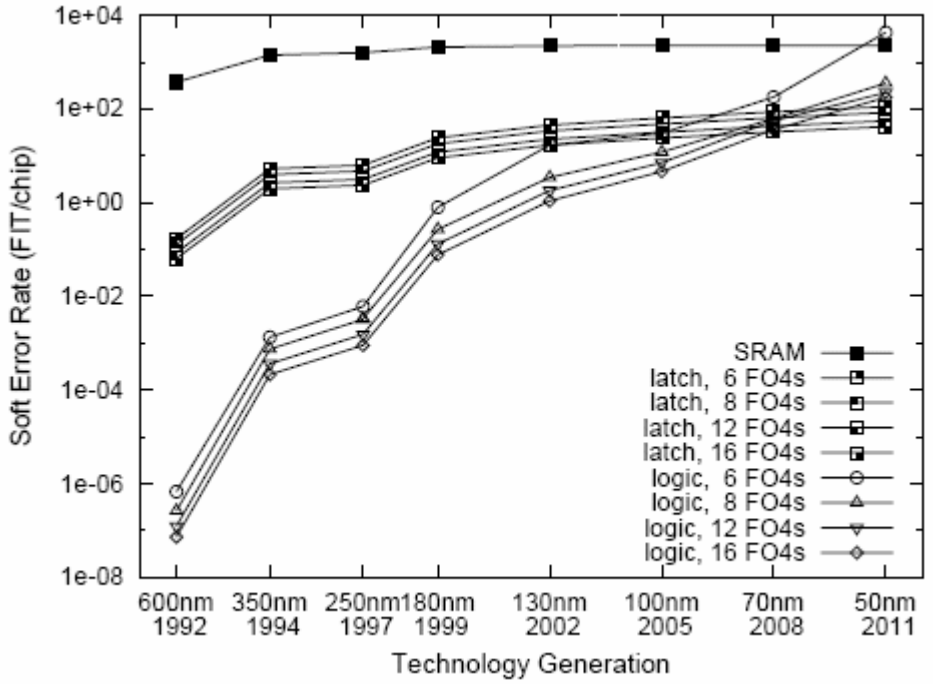
Efficient Transient-Fault Tolerance for Multithreaded Processors Using Dual-Thread Execution

Yi Ma Huiyang Zhou

Computer Science Department
University of Central Florida

Introduction

- Modern microprocessors are increasingly susceptible to transient faults.
  - Smaller transistors, higher density, lower supply voltage, etc.



SER/chip for SRAM/latches/logic [Shivakumar et.al.]

Introduction

- A promising approach is redundant execution utilizing multithreaded processors.
  - AR-SMT, SRT, SRTR, etc.
  - Shortcomings
    - Performance degradation
      - Delayed instruction commitment
      - Resource contention
    - Increased energy consumption
      - Dynamic energy due to redundant execution
      - Static energy due to increased execution time

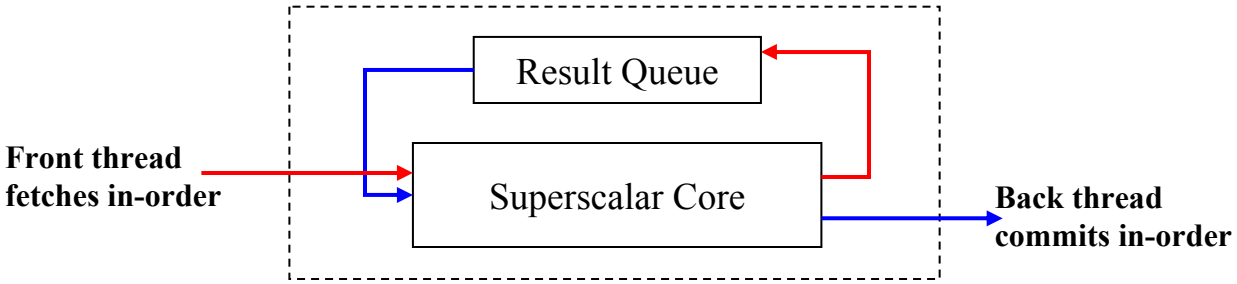
- The contribution of this paper:
  - Dual-Thread Execution: achieves both performance enhancement and transient-fault tolerance for multithreaded processors.

Outline

- Introduction
- Dual-Thread Execution (DTE)
  - Overview
  - Architecture
  - Exploiting Fetch Policies
- Experimental results
- Related Work
- Conclusion

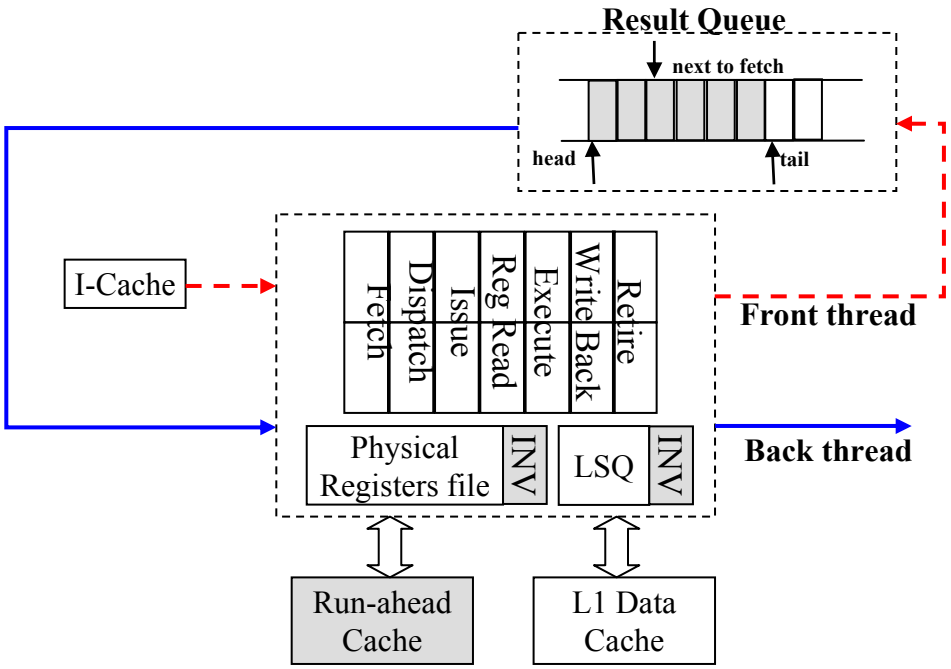
Dual-Thread Execution

- DTE is built on a Simultaneous Multithreaded (SMT) processor
  - Two threads: the front thread and the back thread
  - Instructions are executed speculatively by the front thread and re-executed by the back thread.



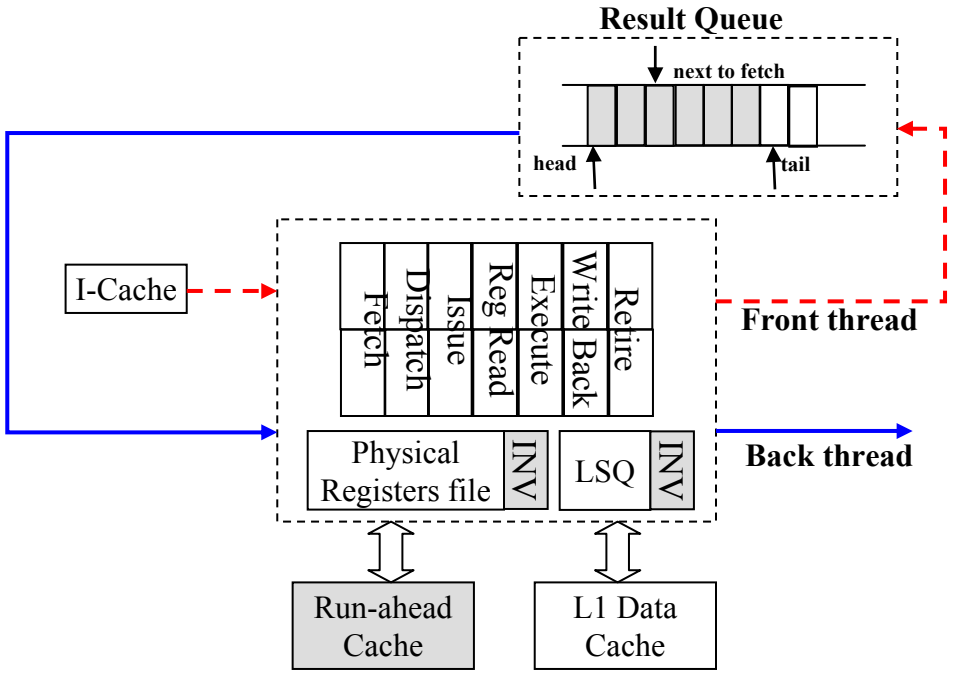
- Resource sharing is critical to DTE's overall performance.
  - Explore effective fetch policies for DTE.

Architecture



- Front thread
  - Fetches instructions from the I-cache.
  - Executes instructions normally except for long-latency (L2 miss) loads.
    - Invalidates long-latency loads and their dependant instructions by setting the INV flag. (The INV flag is propagated.)
  - Writes store values into the *run-ahead cache* instead of the D-cache when retiring.
  - Forwards the retired instructions with their results to the *result queue (FIFO)*.

Architecture (cont.)



- Back thread
  - Fetches instructions from the result queue.
    - Instructions invalidated by the front thread are fetched twice to achieve full redundancy coverage.
  - Performs redundancy check.
    - Compares with the front thread results for valid instructions.
    - Compares with the redundant copy.

Architecture (cont.)

- When a discrepancy is detected
  - Soft error
  - Misspeculation from the front thread

- Rewind both threads to the currently committed states.
  - Squash everything in the back thread, the result queue and the front thread.
  - Invalidate the run-ahead cache.
  - Copy the back thread's architectural states to the front thread.
  - Resume execution.

How does DTE improve performance?

- The front thread runs on virtually ideal L2 by invalidating long-latency cache-missing loads.
- The cache misses in the front thread become very useful pre-fetches for the back thread.
  - It reduces cache misses and enables more computation overlapping in the back thread.
- Front thread resolves all the branches that are independent on the invalidated instructions.
  - It provides back thread highly accurate control flow.

How does DTE achieve transient-fault tolerance?

- Every instruction is redundantly executed.
- The redundant results are checked before committing to ECC protected architectural states.
- Any discrepancy due to soft errors can be transparently repaired.

Outline

- Introduction
- Dual-Thread Execution (DTE)
  - Overview
  - Architecture
  - Fetch policies for DTE
- Experimental results
- Related Work
- Conclusion

Fetch Policies for DTE

- ROUND-ROBIN (RR) policy
  + Fairness
  - Fails to consider the resource requirement for each thread.
- ICOUNT policy
  + Good for high ILP threads
  - Favors the front thread in DTE.
- SLACK policy
  + Speeds up the trailing thread in SRT and SRTR.
  - Favors the front thread in DTE.

Fetch Policies for DTE (cont.)

- Back-First (BF) policy
  + Favors the back thread.
  - Limits the fast progress of the front thread.

- Queue-Occupancy (QO) policy
  - When the occupancy is less than 50%, it favors the front thread, otherwise it favors the back thread.
  + Allocates resources effectively to both threads.

Outline

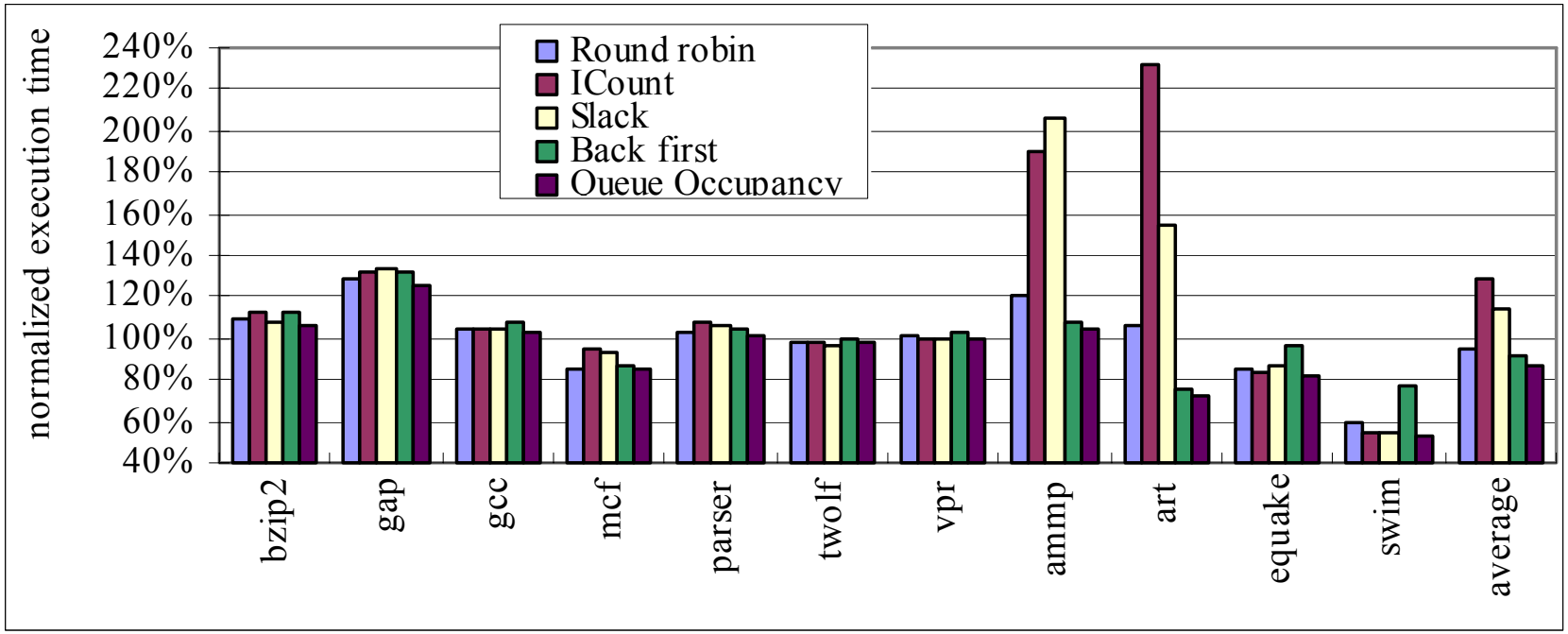
- Introduction
- Dual-Thread Execution (DTE)
  - Overview
  - Architecture
  - Fetch policies for DTE
- Experimental results
- Related Work
- Conclusion

Methodology

- Processor settings
  - MIPS R10000 style superscalar processor supporting SMT
  - 8-way issue, 128-entry ROB, 128-entry issue queue, 128-entry LSQ
  - 32 kB 2-way L1 caches, 1024 kB 8-way L2 cache, L2 miss latency: 300 cycles
  - Branch predictor: 64k-entry G-share; 32k-entry BTB
  - Stride-based stream-buffer hardware prefetcher
  - 512-entry result queue, 4 kB 4-way run-ahead cache
  - Latency for copying architectural register values from back to front thread: 8 cycles

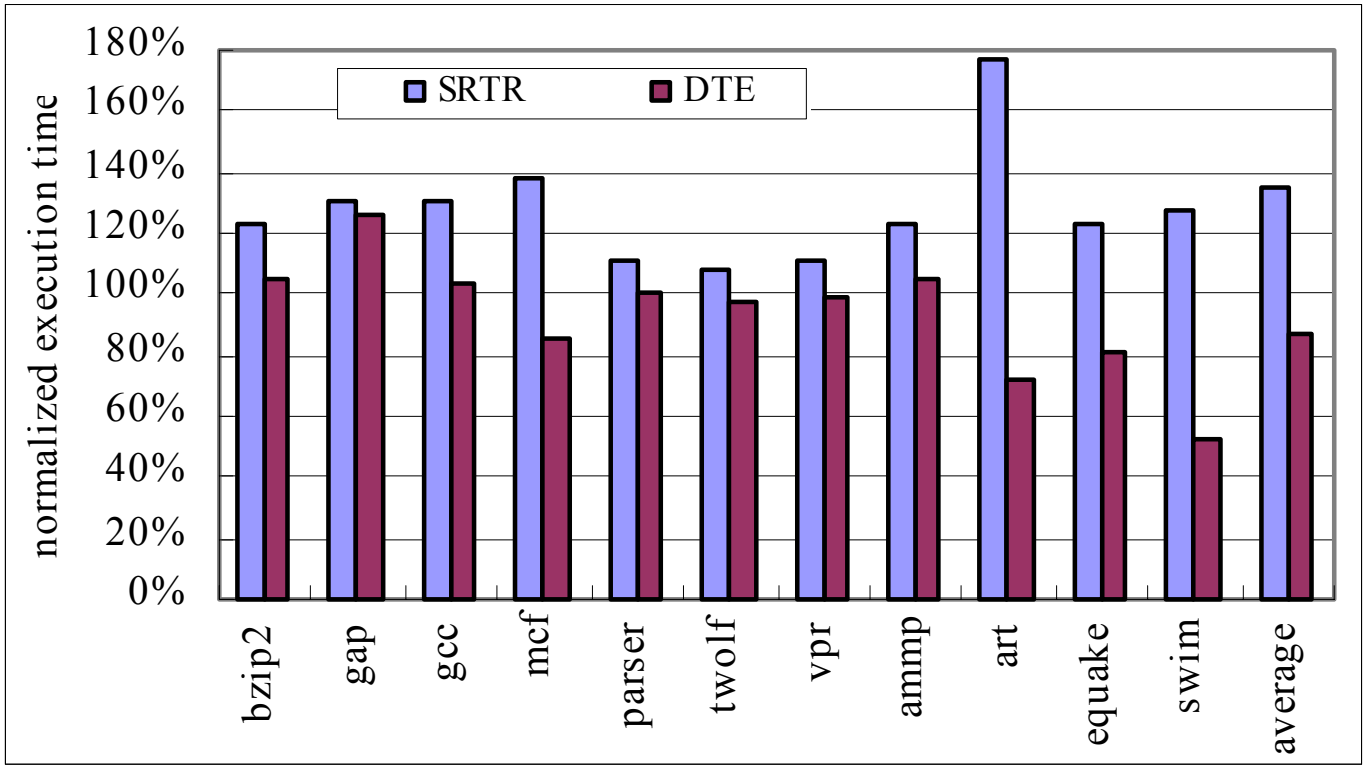
- Benchmarks
  - Memory-intensive spec2000 benchmarks (>40% speedup with perfect L2) and two computation-intensive benchmarks, bzip2 and gap.

Different Fetch Policies



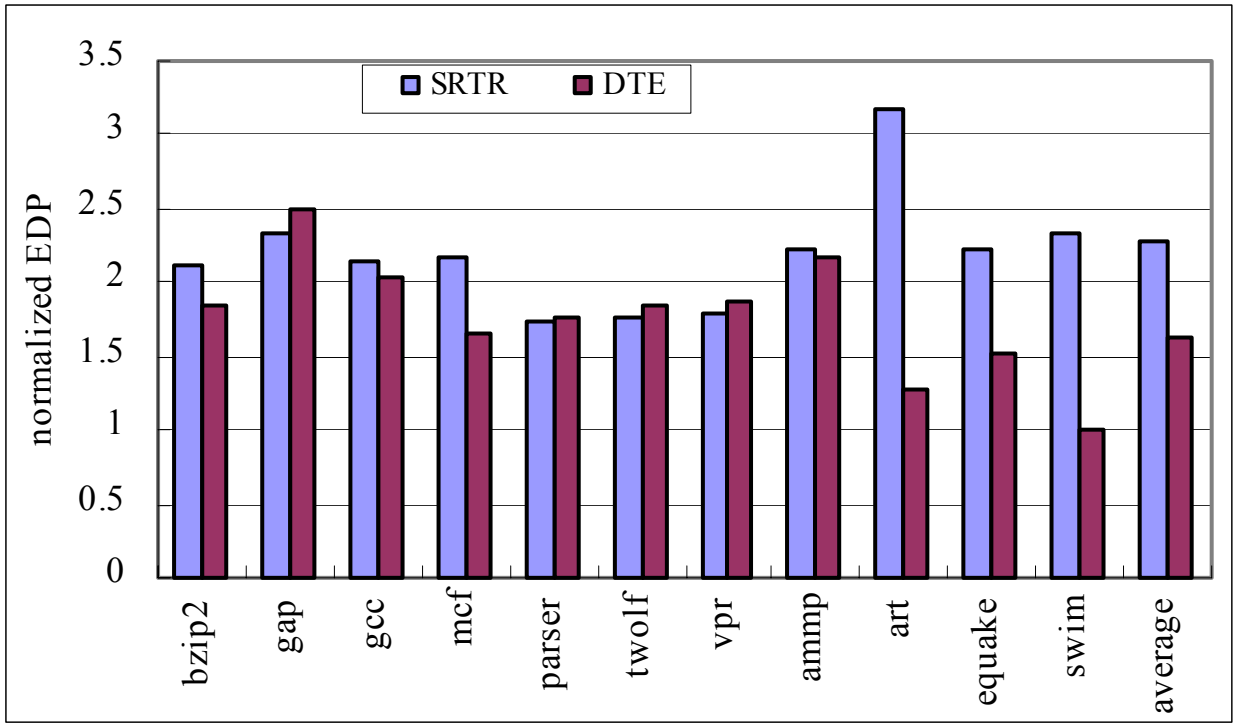
Queue-Occupancy fetch policy works best for DTE.

Performance Impact of DTE



On average, DTE achieves 15.5% speedup.

Energy Efficiency of DTE



On average, DTE reports much higher energy efficiency than SRTR (1.63 vs. 2.29).

Related Work

- SRT [Reinhadt and Mukherjee], SRTR [Vijaykumar et.al.]
- AR-SMT [Rotenberg]
  - Similar high-lever architecture (delay buffer vs. result queue)
  - The A-stream executes the program non-speculatively.
  - The R-stream validates the results from the A-stream.
- DIVA [Austin]
  - Uses a separate simple in-order checker to verify the out-of-order execution of the main thread.
- Dual-Core Execution [Zhou]
  - DCE builds on two processor cores on a single chip.
  - The two cores work cooperatively to improve the performance of single-thread.
  - DTE is derived from DCE.

Summary

- Dual-Thread Execution builds upon SMT processors.
- The front thread and the back thread execute instruction stream collaboratively to provide efficient transient-fault tolerance.
- Works best with the Queue-Occupancy fetch policy.
- SMT-based design to achieve both high reliability and performance improvement.

Thank you and Questions?

Computer Science Department
University of Central Florida