
  
Karlsruhe Institute of Technology

# Reliable Computing I

## Lecture 10: Re-execution


**Instructor: Mehdi Tahoori**

INSTITUTE OF COMPUTER ENGINEERING (ITEC) – CHAIR FOR DEPENDABLE NANO COMPUTING (CDNC)



KIT – University of the State of Baden-Wuerttemberg and  
National Research Center of the Helmholtz Association

[www.kit.edu](http://www.kit.edu)

  
Karlsruhe Institute of Technology

## Today's Lecture

- Re-Execution techniques
  - RESO
  - Multithreading

(c) 2014, Mehdi Tahoori

Reliable Computing I: Lecture 10

2

## Re-Execution



- Replicate the actions on a module either
  - on the same module (temporal redundancy) or
  - on spare modules (temporal & spatial redundancy)
- Good for detecting and/or correcting transient faults
  - Transient error will only affect one execution
- Analogy from real life: calling to confirm a reservation
- Can implement this at many different levels
  - ALU
  - Thread context
  - Processor
  - System

## Re-Execution with Shifted Operands (RESO)



- Re-execute the same arithmetic operations, but with shifted operands (question: why shift?)
- Goal: detect errors in ALU
- Example: shift left by 2
  - Simplified example: we're ignoring wraparound

$$\begin{array}{r}
 0010 \\
 + 1001 \\
 \hline
 1010
 \end{array}
 \qquad
 \begin{array}{r}
 10XX \\
 + 01XX \\
 \hline
 11XX
 \end{array}$$

- By comparing output bit 0 of the first execution and output bit 2 of the shifted re-execution, we detect an error in the ALU, since they should be equal

## Re-Execution With a Twist



- After adding  $A + B = C$ , then compute  $C - B$ 
  - If we don't get  $A$ , there's a problem
- What new types of faults/errors does this detect?
- How general is this approach?
  - I.e., how many operations are reversible?
  - Can we extend this to higher-level operations (algorithms)?
- The devil is in the details (corner cases)
  - Overflow, underflow, divide by zero, etc.
- This type of execution checking is more frequently performed at the software level ... why?

## Re-Execution with Processes



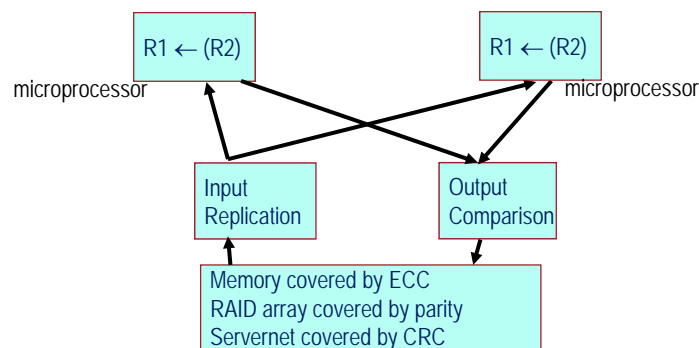
- Use redundant process to detect errors
- If we only have one single-threaded core, we must execute the two processes sequentially and then compare their results. If they differ, there's an error.
  - Problem: slowdown factor = 2
- In a multicore, we can execute copies of the same process simultaneously on 2 cores and have them periodically compare their results
  - Trend: even single chips contain multiple processors
  - Almost no slowdown, except for comparisons
  - Disadvantages: the opportunity cost and power/energy cost of not using that other core to perform non-redundant work
  - Is this an FER approach? (hint: what happens if an error occurs?)

## Re-Execution with Threads



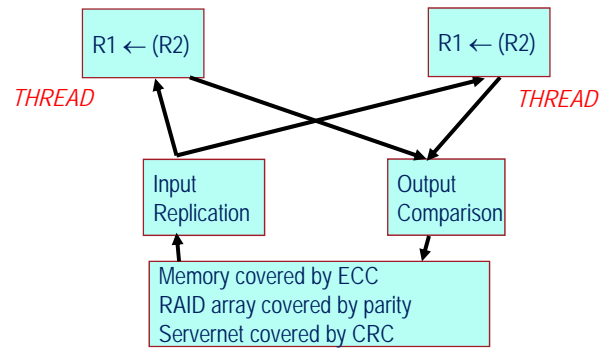
- Use redundant **threads** to detect/correct errors
  - A thread is like a process, except that multiple threads can share the same address space
- Many current microprocessors, like the Pentium4, are **multithreaded** (“hyperthreaded”, if you work for Intel)
  - Each processor can run multiple processes or multiple threads of the same process (i.e., it has multiple **thread contexts**)
- Can re-execute a program on multiple thread contexts, just like with multiple processors
  - Better performance than re-execution with multiple processors, since the comparison can be performed on-chip
  - Less opportunity cost to use extra thread context than extra processor

## Fault Detection via Lockstepping (HP Himalaya)



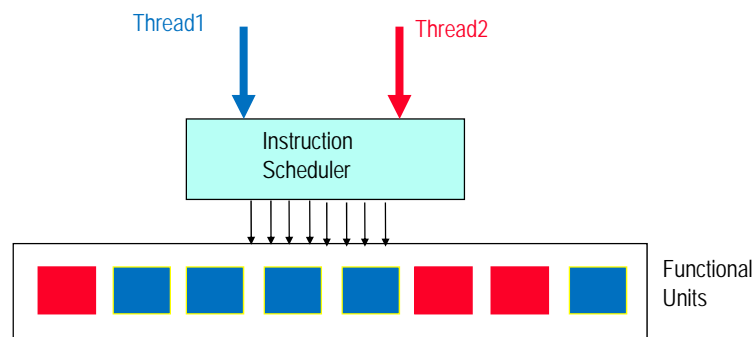
*Replicated Microprocessors + Cycle-by-Cycle Lockstepping*

## Fault Detection via Simultaneous Multithreading



*Threads* ?  
*Replicated Microprocessors + Cycle-by-Cycle Lockstepping*

## Simultaneous Multithreading (SMT)



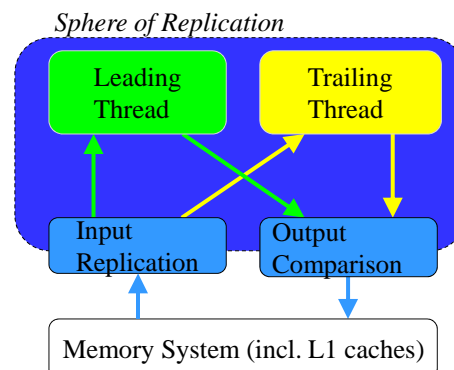
Example: Alpha 21464, Intel Northwood

## Redundant Multithreading (RMT)

■ RMT = Multithreading + Fault Detection (& Recovery)

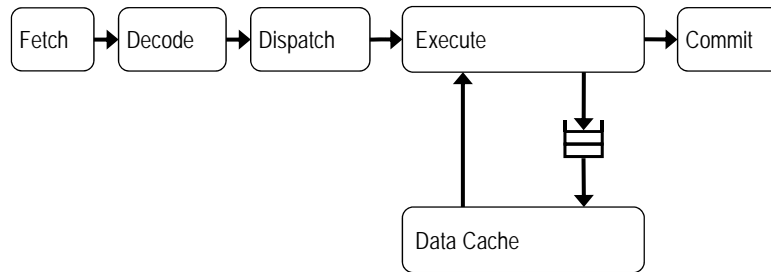
	Multithreading (MT)	Redundant Multithreading (RMT)
Multithreaded Uniprocessor	Simultaneous Multithreading (SMT)	Simultaneous & Redundant Threading (SRT)
Chip Multiprocessor (CMP)	Multiple Threads running on CMP	Chip-Level Redundant Threading (CRT)

## Sphere of Replication



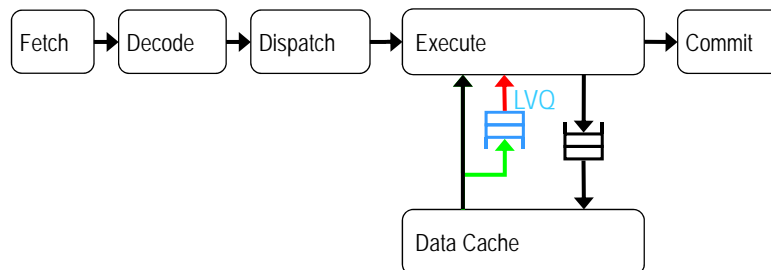
- Two copies of each architecturally visible thread
  - Co-scheduled on SMT core
- Compare results: signal fault if different

## Basic Pipeline



Both leading & trailing threads would go through this pipeline

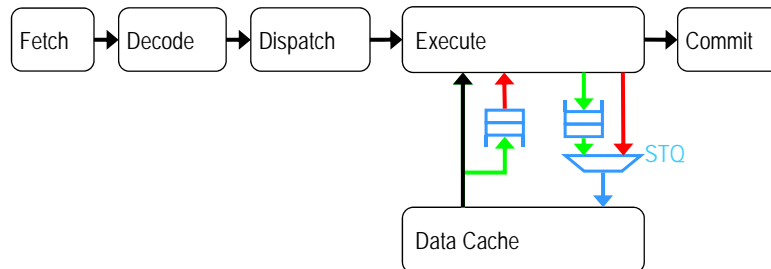
## Load Value Queue (LVQ)



### ■ Load Value Queue (LVQ)

- Keep threads on same path despite I/O or MP writes
- Out-of-order load issue possible

## Store Queue Comparator (STQ)



### ■ Store Queue Comparator

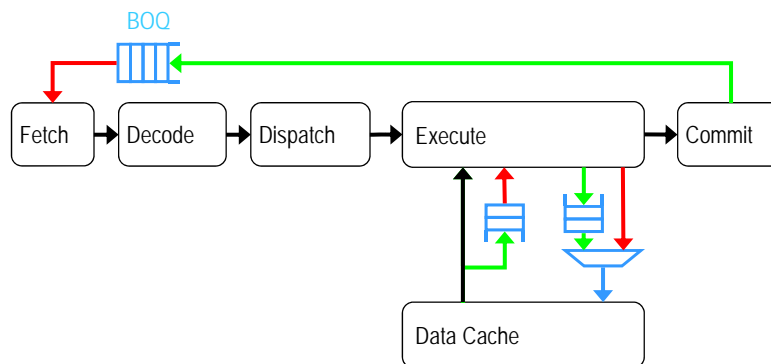
- Compares outputs to data cache
- Catch faults before propagating to rest of system

(c) 2014, Mehdi Tahoori

Reliable Computing I: Lecture 10

15

## Branch Outcome Queue (BOQ)



### ■ Branch Outcome Queue

- Forward leading-thread branch targets to trailing fetch
- 100% prediction accuracy in absence of faults

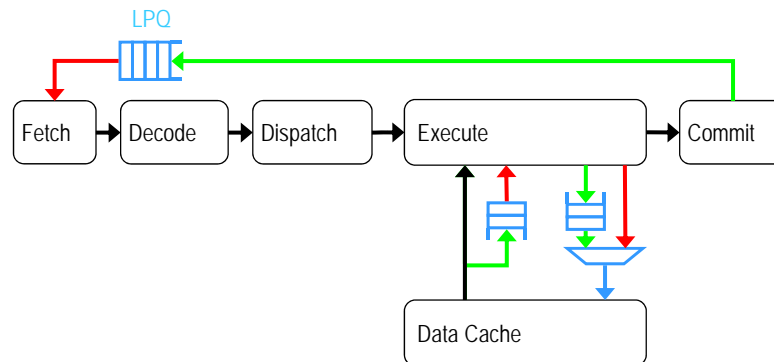
(c) 2014, Mehdi Tahoori

Reliable Computing I: Lecture 10

16



## Line Prediction Queue (LPQ)



### ■ Line Prediction Queue

- Alpha 21464 fetches chunks using line predictions
- Chunk = contiguous block of 8 instructions

(c) 2014, Mehdi Tahoori

Reliable Computing I: Lecture 10

17

## SRT Performance



- One logical thread → two hardware contexts
  - Performance degradation = 30%
  - Per-thread store queue buys extra 4%
- Two logical threads → four hardware contexts
  - Average slowdown increases to 40%
  - Only 32% with per-thread store queues

(c) 2014, Mehdi Tahoori

Reliable Computing I: Lecture 10

18

## Chip-Level Redundant Threading



- SRT typically more efficient than splitting one processor into two half-size CPUs
- What if you already have two CPUs?
  - Multicore processors
- Conceptually easy to run these in lock-step
  - Benefit: full physical redundancy
  - Costs:
    - Latency through centralized checker logic
    - Overheads (misspeculation etc.) incurred twice
- CRT combines best of SRT & lockstepping
  - requires multithreaded CMP cores
- With per-thread store queues, ~13% improvement over lockstepping with 8-cycle checker latency

## Chip-Level Redundant Threading

