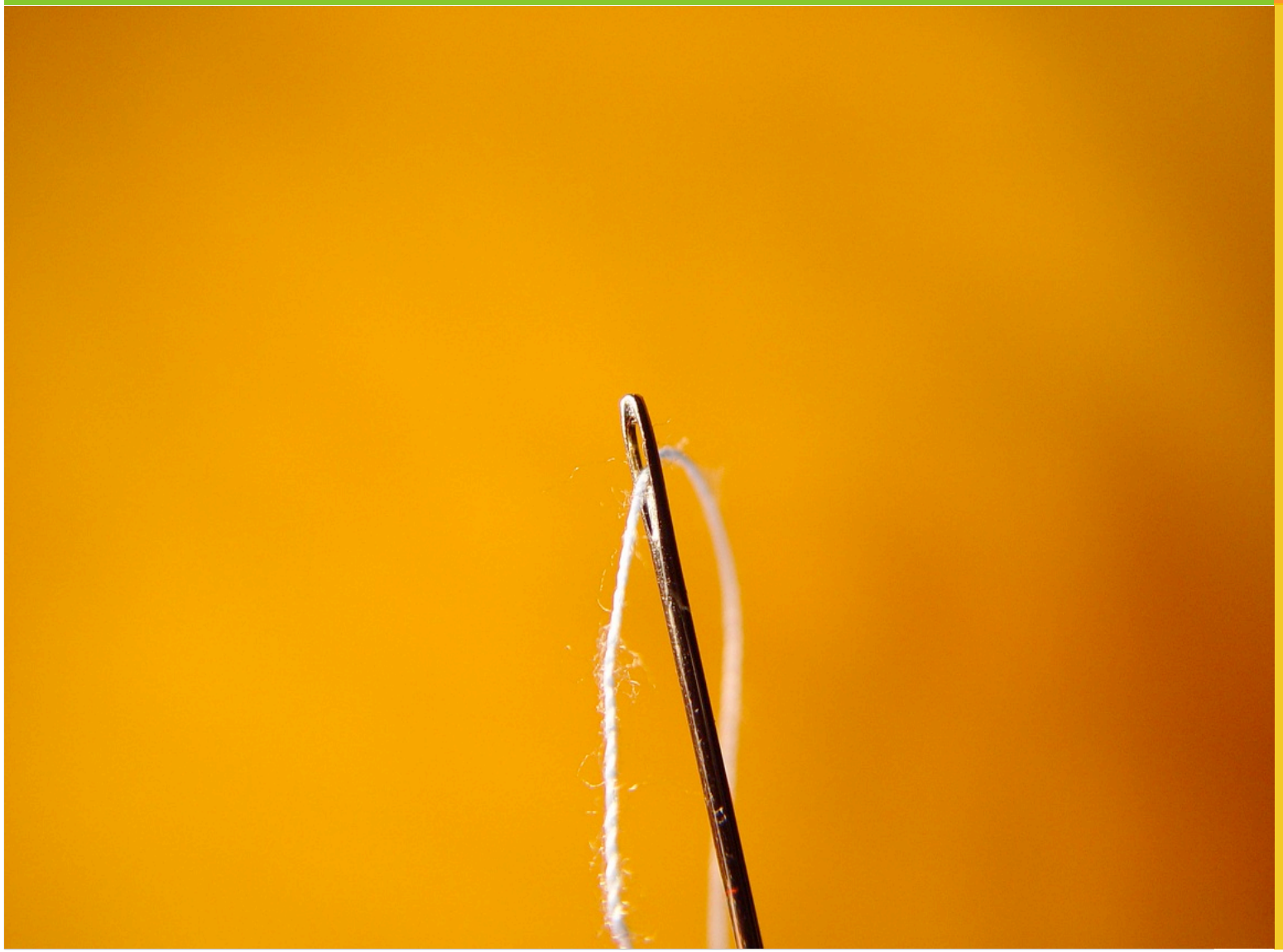
Using threads for Fault Tolerance

Pablo Montesinos

Today's Lecture

- Introduction
- Design Space
- Uniprocessors:
  - Transient fault detection
  - Transient fault recovery
- Multiprocessors:
  - Transient fault detection
  - Transient fault recovery

Introduction



Background

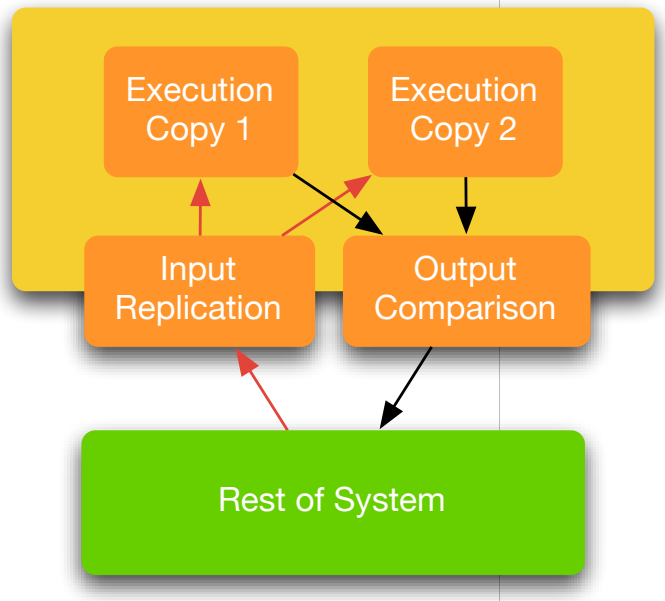
- Fault tolerant systems use redundancy to improve reliability:
  - Time redundancy: separate executions
  - Space redundancy: separate physical copies of resources
    - DMR/TMR
  - Data redundancy
    - ECC
    - Parity
- Examples:
  - IBM 390: duplicated pipelines, spare processors, ECC in memories...
  - HP NonStop: DMR/TMR processors, Parity/ECC in buses, memories...

Concept

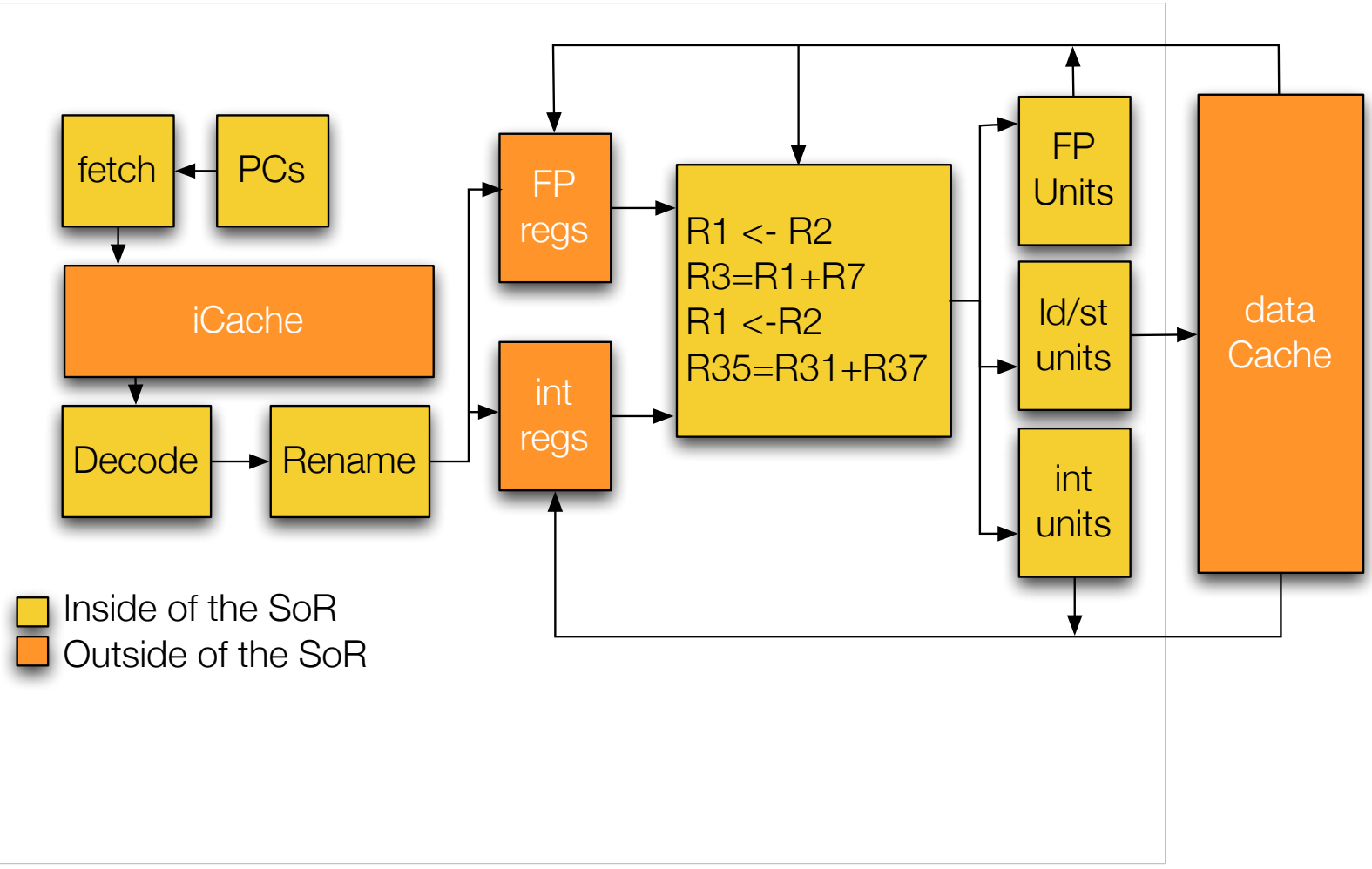
- SMT improves the performance of a processor by:
  - allowing independent threads to execute simultaneously
  - doing so in different functional units
- Redundant Multithreading (RMT):
  - leverages SMT's properties to allow fault detection for microprocessors
    - runs two copies of the same program as independent threads
    - compares their outputs and initiates recovery in case of mismatch
  - uses the sphere of replication to decide which components must be replicated
  - slack between threads provides time redundancy

Sphere of Replication

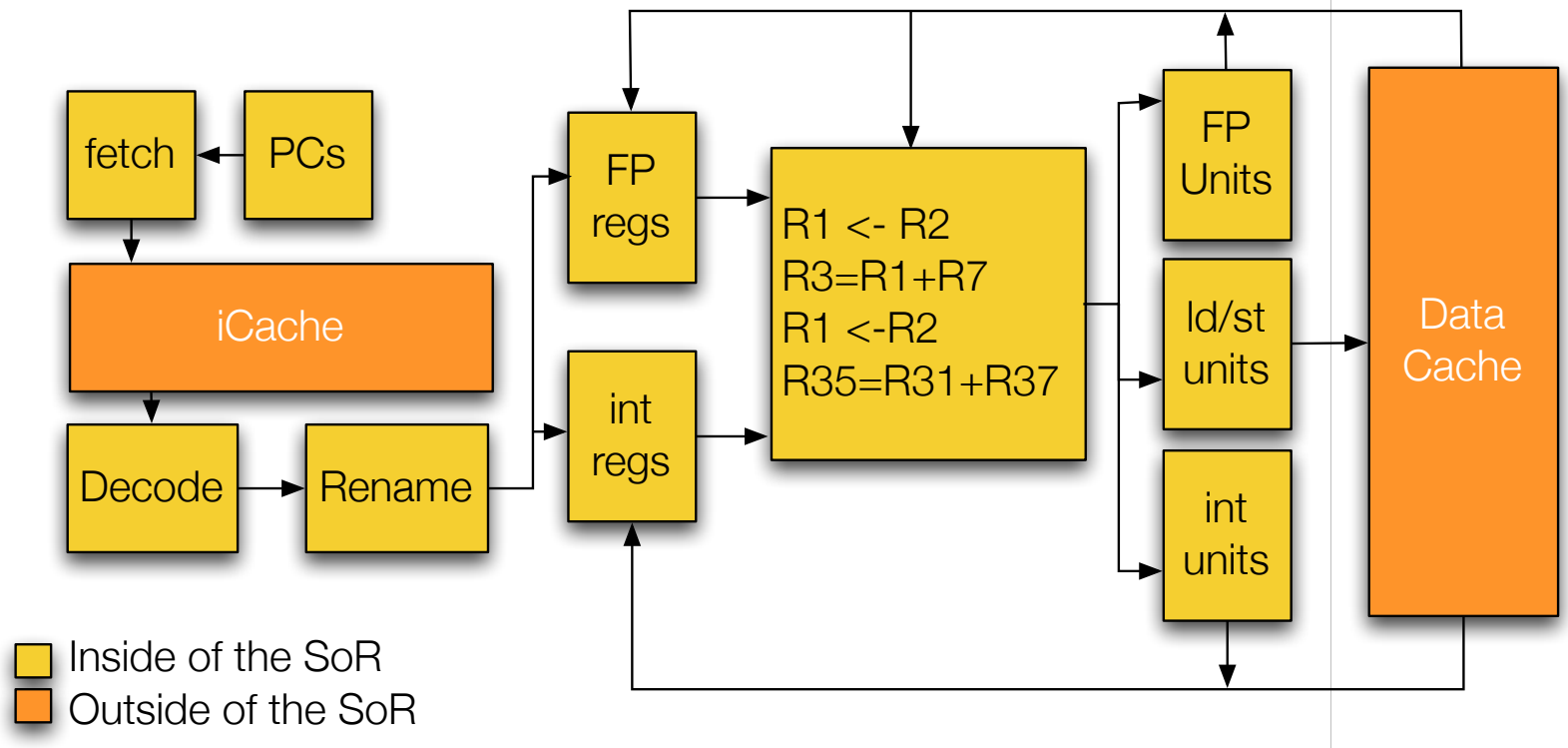
- Logical boundary of redundant execution within a system
  - Components within protected via redundant execution
  - Components outside must be protected via other means
- Its size matters:
  - Error detection latency
  - Stored-state size



Sphere of Replication



Sphere of Replication



Why is so attractive?

- Require less hardware than conventional hardware replication
  - Space Redundancy not critical: use Time and Data redundancy
  - Out-of-the-shelf processors provide most of the requirements
- Increased performance due to dynamic resource partition
  - HP might not agree on this, though

Which are the problems?

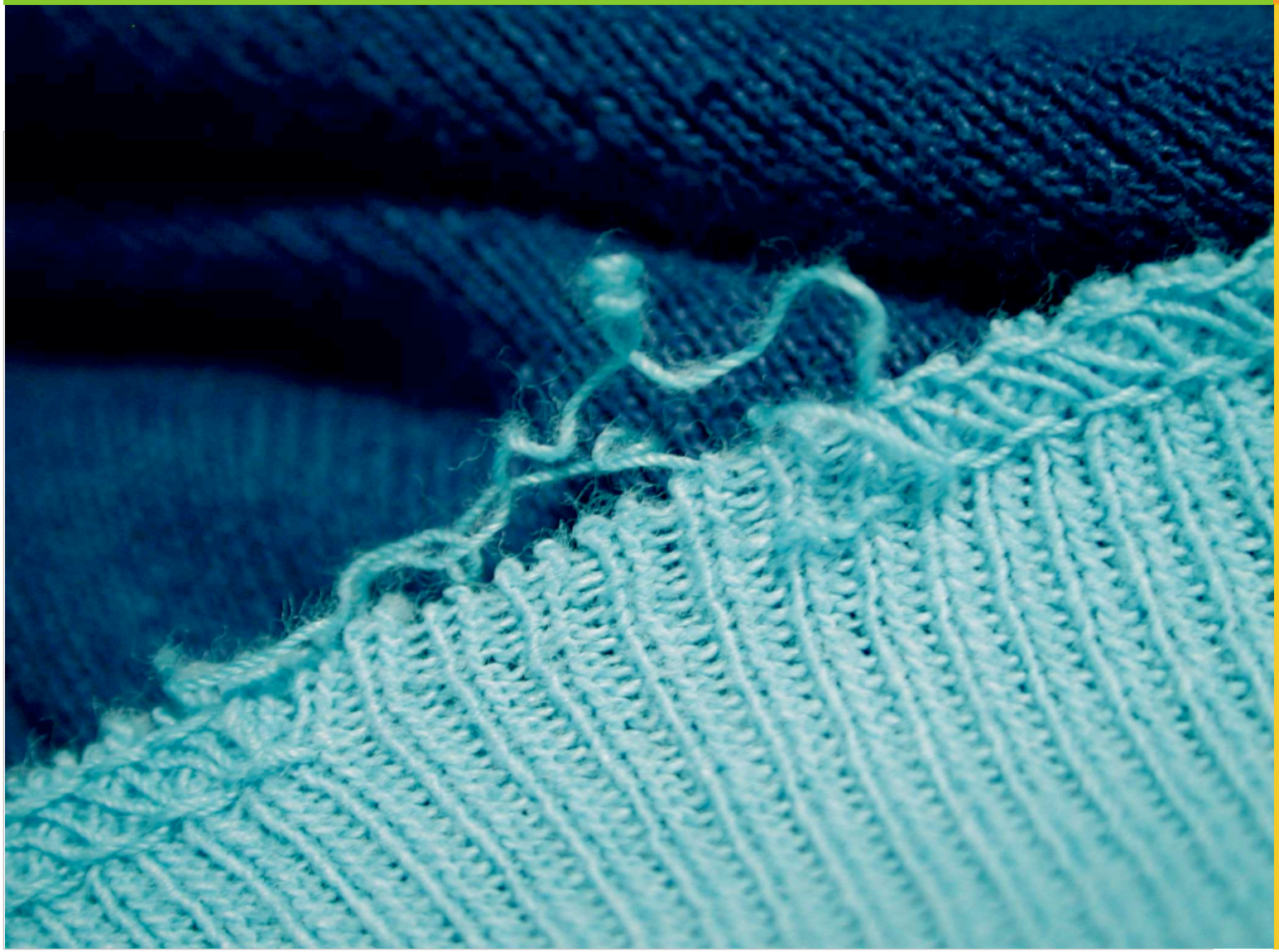
- Cycle-by-cycle output comparison and input replication useless:
  - Equivalent insts from different threads may execute in different cycles
  - Equivalent insts from different threads might execute in different order
- Precise scheduling of the threads crucial for optimal performance

Design Space

Design Space

	Uniprocessor	Multiprocessor
base technology	SMT	CMP
Fault detection	AR-SMT (1999) SRT (2000), RMT(2002)	CRT (2002)
Fault detection and recovery	SRTR (2002)	CRTR (2003)

Uniprocessors: Fault Detection (SRT)

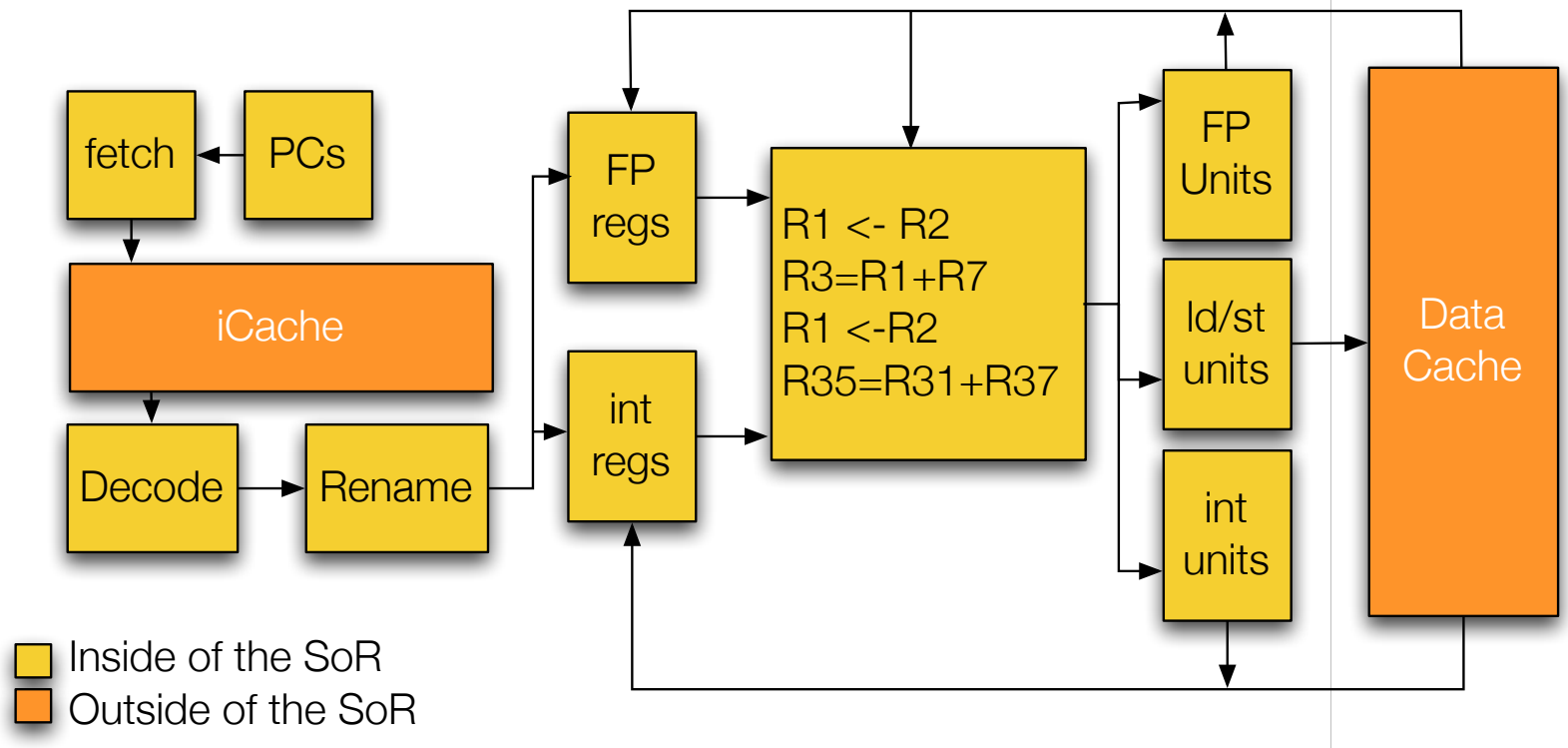


SRT

- **S**imultaneous and **R**edundantly **T**hreaded Processor
- *"Transient Fault Detection via Simultaneous Multithreading"*

  S.K.Reinrhardt and S.S Mukherjee, ISCA 2000

Processor Model



Input Replication

- Instructions:
  - No change needed
- Cached load data:
  - Active Load Address Buffer
  - Load Value Queue
- Uncached load data:
  - Synchronize when comparing addresses that leave the SoR
  - When data returns, replicate the value for the two threads
- External Interrupts:
  - Stall the leading thread
  - Deliver to the leading and record

Active Load Address Buffer (ALAB)

- Delays a cache block's replacement or invalidation after the retirement of the trailing load

tag	counter	pending-invalidate bit

- When a cache block is about to be replaced:
  - The ALAB is searched for an entry matching the block's address
  - **If** counter != 0 **then:**
    - Do not replace nor invalidate until trailing thread is done
    - Set the pending-invalidate bit
  - **Else** replace - invalidate
- It can deadlock

Load Value Queue (LVQ)

- Alternate mechanism to ALAB (also simpler)
- One thread is designed as leading and the other as trailing
- Originally: simple, ECC-protected FIFO
  - When loads commit, store address and value in the LVQ
  - Trailing thread execute loads in program order
- Optimization:
  - Allow out-of-order load issue from the trailer thread

Output comparison

- Stores:
  - Check address and data of committed stores before leaving SoR
  - Bottleneck if store queue is shared
  - Separate per-thread store queues boost performance
- Cached load addresses:
  - No need to check: they don’t modify the architectural state of the machine
  - Errors will be detected via other checks before they change the state
- Uncached load addresses:
  - They usually have side effects in I/O devices outside SoR
  - Stall in execute stage until the corresponding instruction in the other thread

Wanna more performance?

- Slack fetch:
  - Maintains a constant slack of instructions between threads
  - Benefit?
- Branch Misprediction Queue:
  - Sends the outcomes of the committed branch outcomes to the trailing thread
  - Can be tricky to implement on a real processor
- Preferencial Space Redundancy:
  - Tries to execute instructions in different FUs

Uniprocessors: Fault Detection and Recovery (SRTR)

SRTR

- **S**imultaneous and **R**edundantly **T**hreaded Processor with **R**ecovery
- *"Transient-Fault Recovery Using Simultaneous Multithreading"*

  T.N. Vijaykumar et al. ISCA 2002

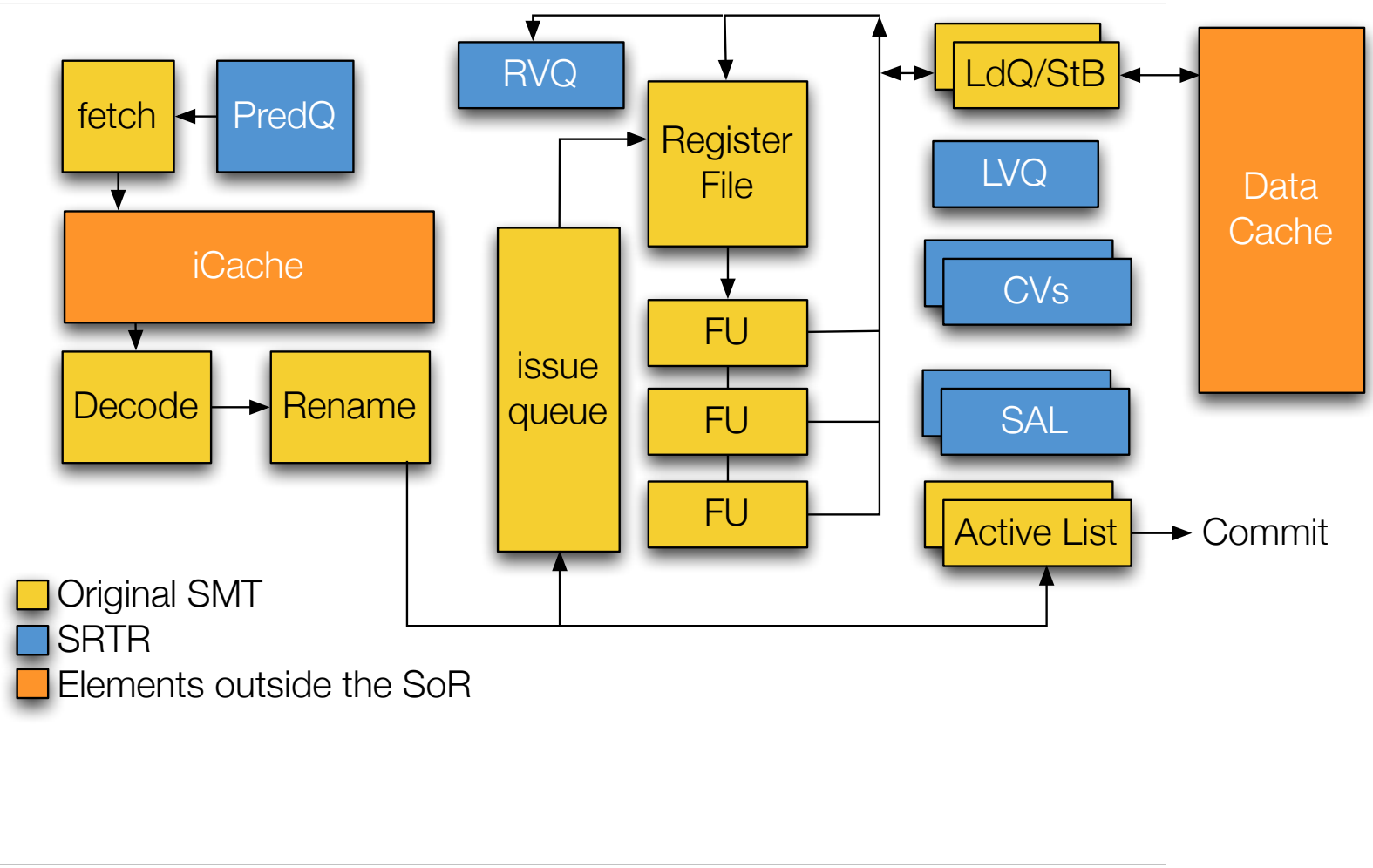
What's wrong with STR?

- A leading non-store instruction may commit before is checked
  - Relies on the trailing thread to trigger the detection
  - Recovery might not be possible

SRTR's basic principles

- SRTR must not allow any leading instruction to commit before checking
- SRTR exploits the time between the completion and commit time of leading instruction and checks the results as soon as the trailing completes
  - In SPEC95, complete to commit takes about 29 cycles
  - This short slack has some implications:
    - Leading thread provides branch predictions
    - The StB, LVQ and BOQ need to handle mispredictions

SRTR´s additions to SMT



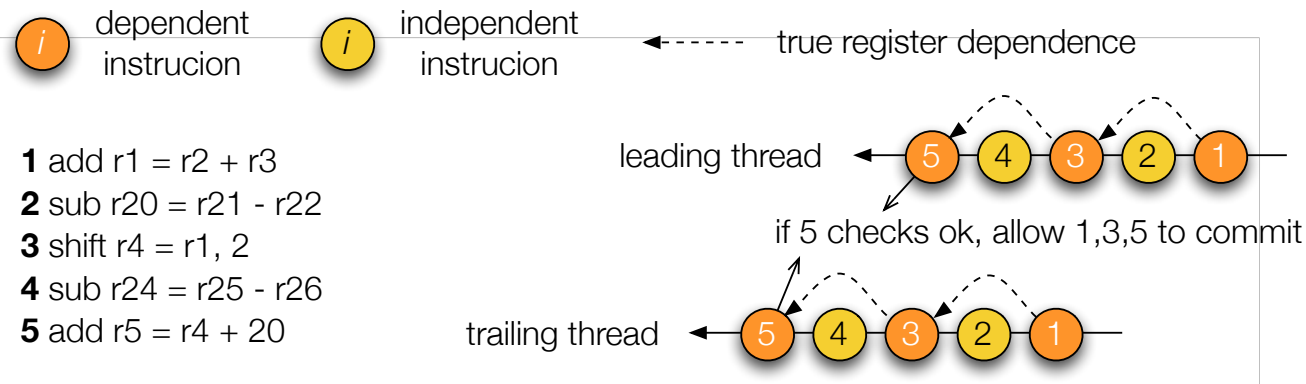
Active Lists (ALs) and the LVQ

- Leading and trailing instructions occupy the same positions in their ALs
  - May enter their AL and become ready to commit them at different times
- The LVQ has to be modified to allow speculative loads
  - The Shadow Active List holds pointers to LVQ entries
  - A trailing load might issue before the leading load
  - Branches place the LVQ tail pointer in the SAL
    - The LVQ's tail pointer points to the LVQ has to be rolled back in a misprediction

Checking Instructions

- SRTR performs checking when the trailing instruction completes
- The Register Value Queue (RVQ) stores register values for checking
  - Avoids pressure on the register file (no extra accesses for checking)
  - RVQ entries are allocated when instruction enter the AL
- If check succeeds, the entries in Commit Vectors are set to *checked-ok*
- If check fails, the entries in Commit Vector are set to *failed*
  - Rollback done when entries in head of AL

DBCE concept



1 add r1 = r2 + r3
2 sub r20 = r21 - r22
3 shift r4 = r1, 2
4 sub r24 = r25 - r26
5 add r5 = r4 + 20

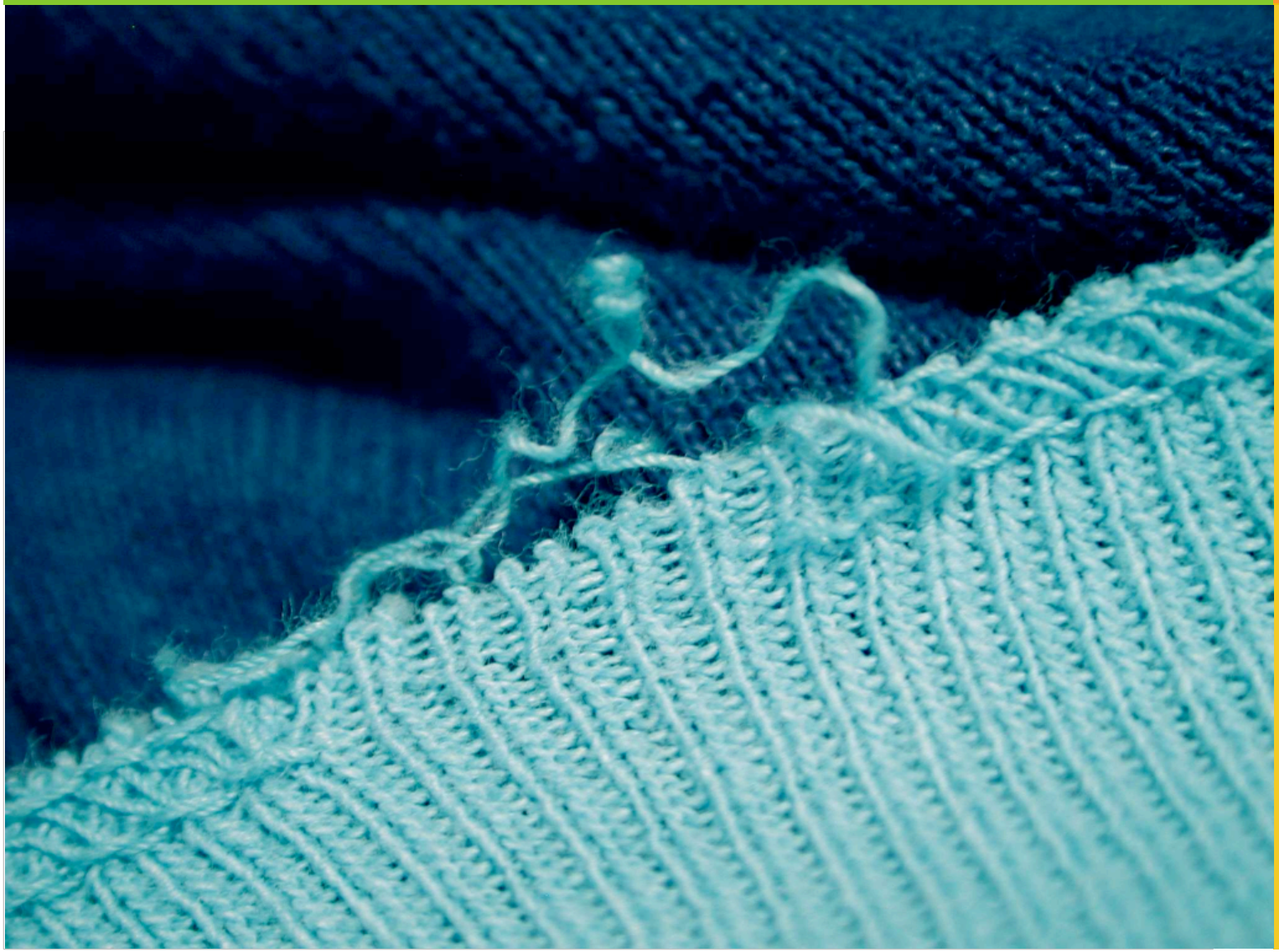
reg	AL	chain tag	flag
101	1	1	1
120	2	2	0
104	3	1	1
124	4	4	0
105	5	1	0

DBQ entries showing only leading instructions

not check		last-instr	
L	T	L	T
1	1	5	5
0	0	2	2
1	1	5	5
0	0	4	4
0	0	5	5

Check Table

Multiprocessors: Fault Detection (CRT)

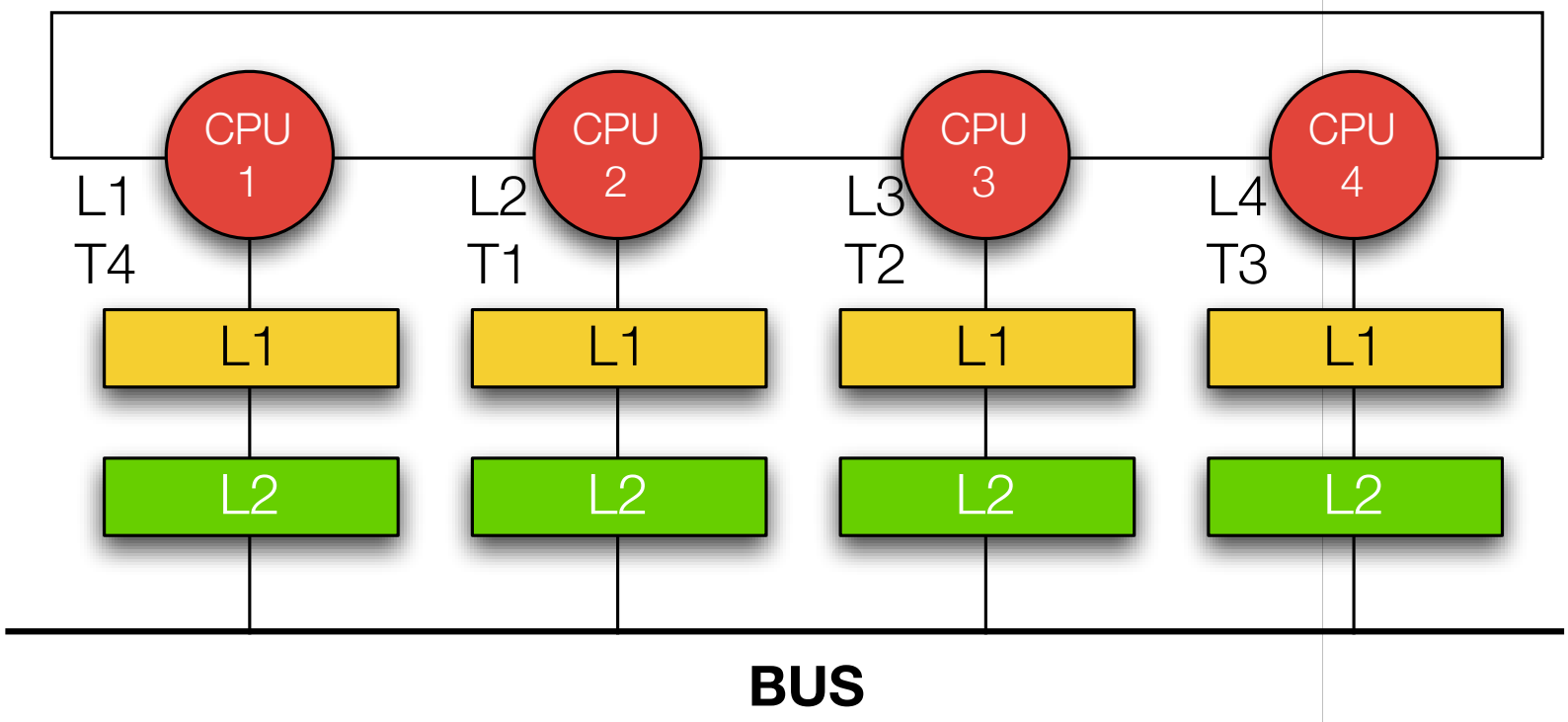


CRT

- **C**hip-level **R**edundantly **T**hreaded processor
- *"Detailed Design and Evaluation of Redundant Multithreading Alternatives" .*

  S.S Mukherjee, M. Kontz, S.K. Reinhardt ISCA 2002

A High Level View of CRT



CRT's Basic Principles

- Replicates register values but not memory values
- The leading thread commits stores only after checking
  - Memory is guaranteed to be correct
  - Other instructions commit without checking
- The leading thread sends committed values for:
  - branch outcomes
  - load/store values
  - store addresses
- Remember: the slack is big
- Uncached accesses are performed non-speculatively

Multiprocessors: Fault Detection and Recovery (CRTR)

CRTR

- **C**hip-level **R**edundantly **T**hreaded Processor with **R**ecovery
- *"Transient-Fault Recovery for Chip Multiprocessors"*

  M. Gooma et al. ISCA 2003

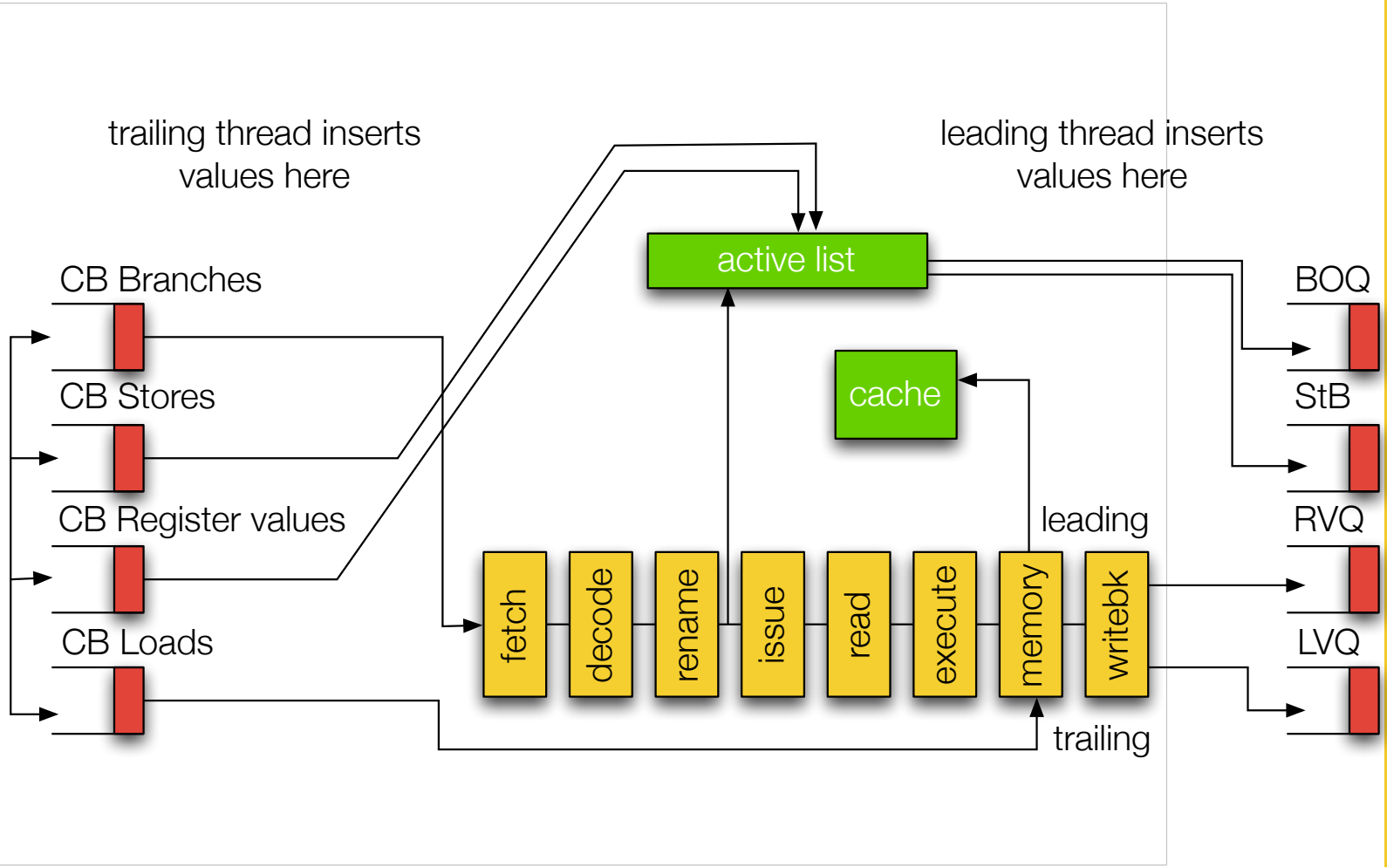
CRTR's Basic Principles

- Asymmetric commit
  - Trailing instructions do NOT commit without checking
  - The register state of the trailing thread is used for recovery
  - Hides interprocessor latency
- Like CRT, the leading thread sends committed values for:
  - Branch outcomes
  - Load/store values
  - Store addresses
  - **Register values**
- Stores are executed only after checking
  - Memory guaranteed to be correct

Sending Leading Values

- Sending values at commit (in program order) eliminates the need for the values to be cleared up in mispredictions
  - We could avoid having a RVQ
  - But committed instructions don't have values at commit, so need to access the register file in both threads
  - Which increases the pressure on the register files
- CRTR places values at writeback in the RVQs
  - The leading thread RVQ entries are sent to the trailing thread
  - Values are read from it at commit

A CRTR CPU


trailing thread inserts values here
leading thread inserts values here
active list
CB Branches
CB Stores
CB Register values
CB Loads
cache
BOQ
StB
RVQ
LVQ
leading
trailing
fetch
decode
rename
issue
read
execute
memory
writebk

Recovery

- The trailing thread preserves the faulting instruction PC so that execution can restart from that instruction
- The exception handler saves the trailing register state and PC to the CMP shared memory
- The leading CPU launches a restoring thread that copies it to the processor.
- The restoring copies the new state of the leading thread to the memory and compare both