18-742 Fall 2012
Parallel Computer Architecture
Lecture 13: Multithreading III

Prof. Onur Mutlu
Carnegie Mellon University
10/5/2012

New Review Assignments

- Due: Tuesday, October 9, 11:59pm.

- Sohi et al., "Multiscalar Processors," ISCA 1995.

- Due: Thursday, October 11, 11:59pm.

- Herlihy and Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," ISCA 1993.

- Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," MICRO 1999.

Last Lectures

- Caching in Multi-Core
- Cache and Memory Compression
- Efficient Caching

Today

- Wrap Up Multithreading
    - Other uses of multithreading

Other Uses of Multithreading

Now that We Have MT Hardware …

- … what else can we use it for?

- Redundant execution to tolerate soft (and hard?) errors

- Implicit parallelization: thread level speculation
  - Slipstream processors
  - Leader-follower architectures

- Helper threading
  - Prefetching
  - Branch prediction

- Exception handling

SMT for Transient Fault Detection

- Transient faults: Faults that persist for a “short” duration
  - Also called “soft errors”
- Caused by cosmic rays (e.g., neutrons)
- Leads to transient changes in wires and state (e.g., 0→1)

- Solution
  - no practical absorbent for cosmic rays
  - 1 fault per 1000 computers per year (estimated fault rate)
- Fault rate likely to increase in the feature
  - smaller feature size
  - reduced voltage
  - higher transistor count
  - reduced noise margin

Need for Low-Cost Transient Fault Tolerance

- The rate of transient faults is expected to increase significantly → Processors will need some form of fault tolerance.

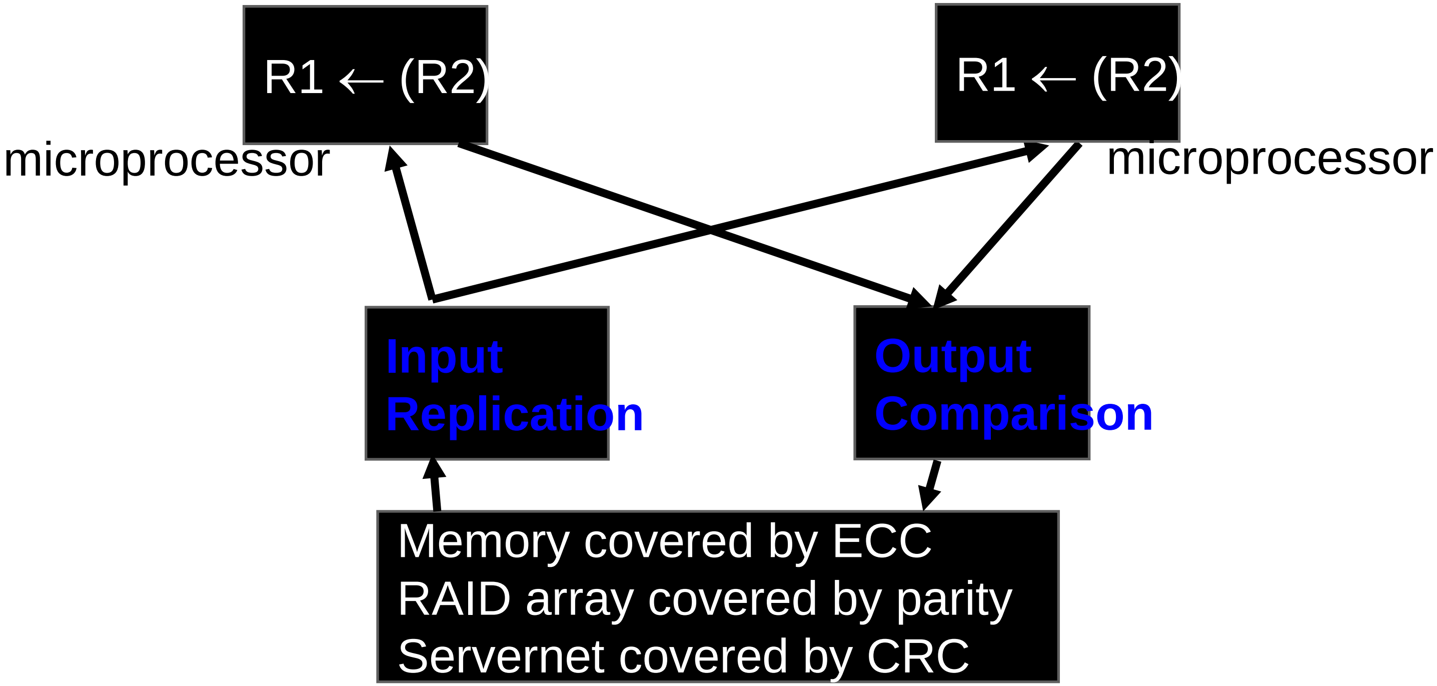
- However, different applications have different reliability requirements (e.g. server-apps vs. games) → Users who do not require high reliability may not want to pay the overhead.

- Fault tolerance mechanisms with low hardware cost are attractive because they allow the designs to be used for a wide variety of applications.

Traditional Mechanisms for Transient Fault Detection

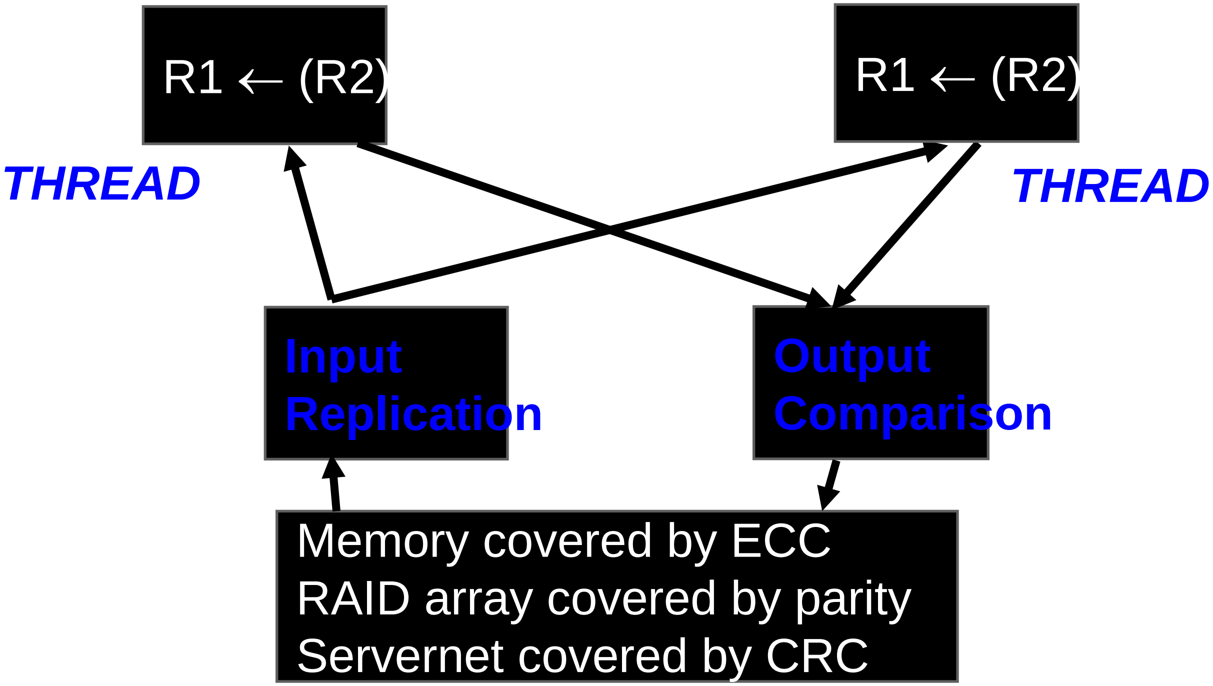
- Storage structures
  - Space redundancy via parity or ECC
  - Overhead of additional storage and operations can be high in time-critical paths
- Logic structures
  - Space redundancy: replicate and compare
  - Time redundancy: re-execute and compare
- Space redundancy has high hardware overhead.
- Time redundancy has low hardware overhead but high performance overhead.
- What additional benefit does space redundancy have?

Lockstepping (Tandem, Compaq Himalaya)



- Idea: Replicate the processor, compare the results of two processors before committing an instruction

Transient Fault Detection with SMT (SRT)



- Idea: Replicate the threads, compare outputs before committing an instruction
- Reinhardt and Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," ISCA 2000.
- Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," FTCS 1999.

Sim. Redundant Threading vs. Lockstepping

- SRT Advantages

  + No need to replicate the processor

  + Uses fine-grained idle FUs/cycles (due to dependencies, misses) to execute the same program redundantly on the same processor

  + Lower hardware cost, better hardware utilization

- Disadvantages

  - More contention between redundant threads → higher performance overhead (assuming unequal hardware)

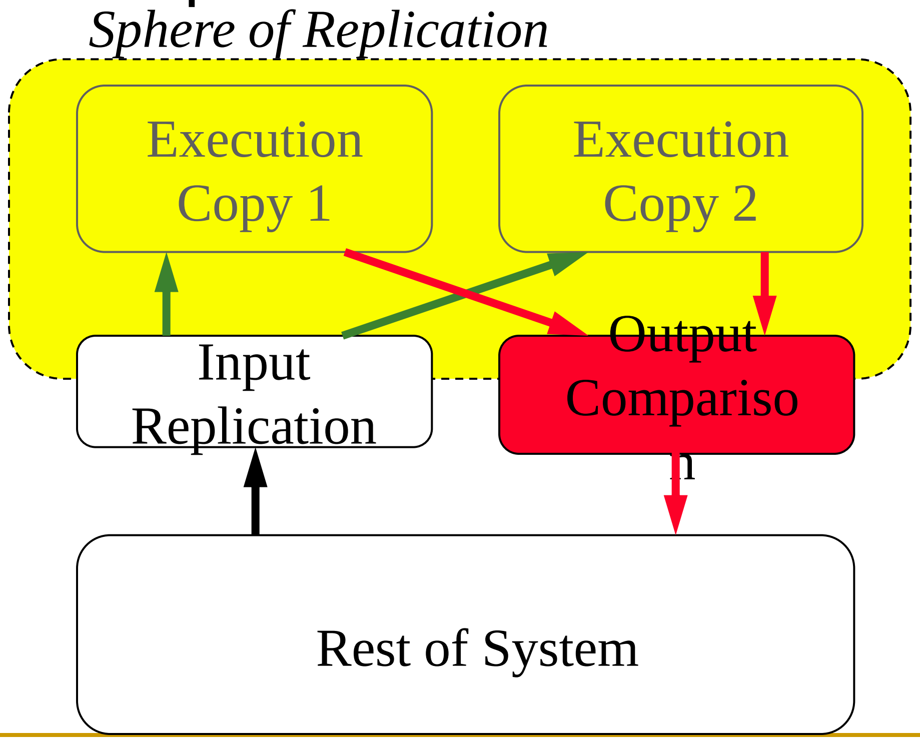
  - Requires changes to processor core for result comparison, value communication

  - Must carefully fetch & schedule instructions from threads

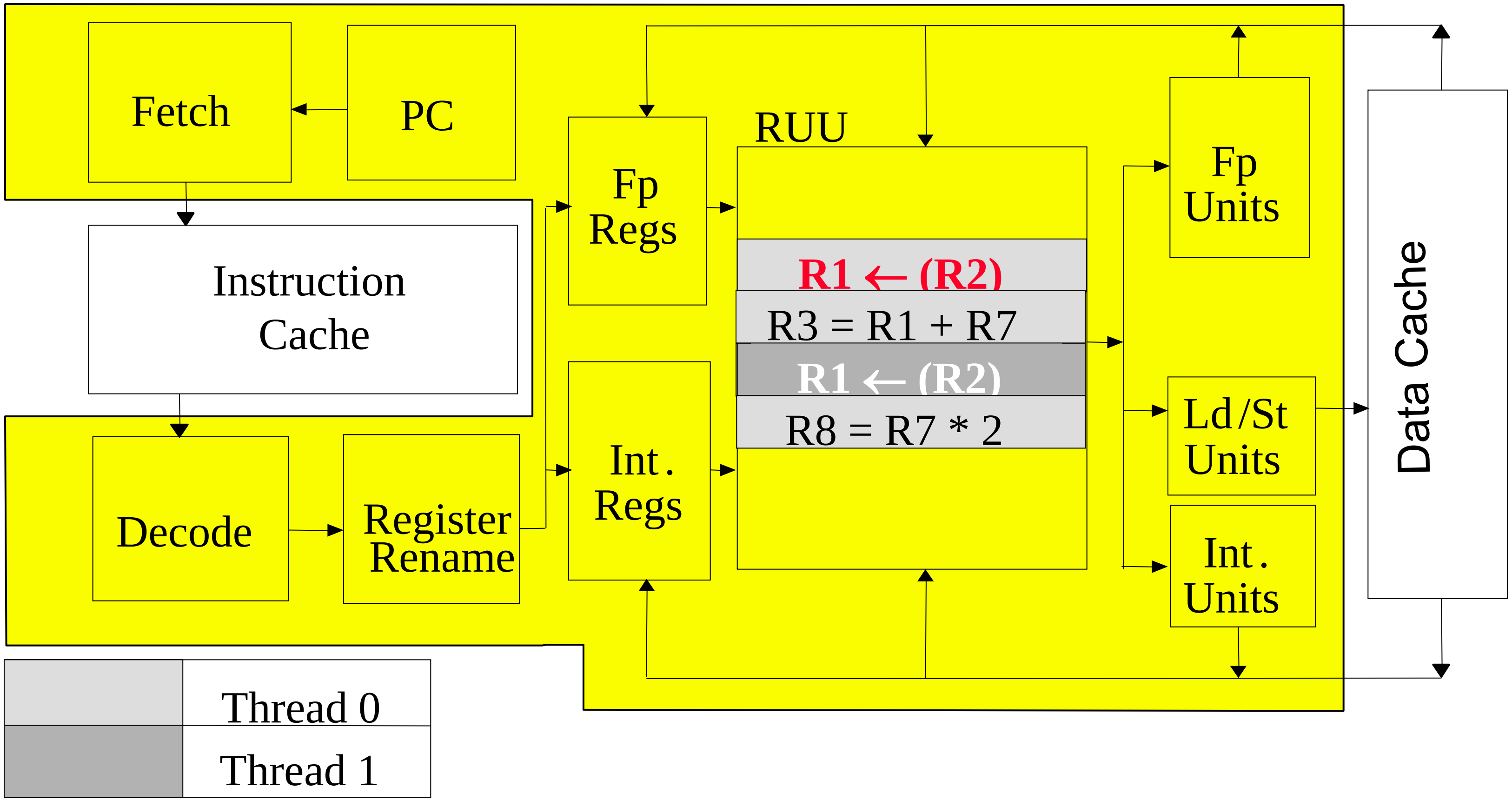
  - Cannot easily detect hard (permanent) faults

Sphere of Replication

- Logical boundary of redundant execution within a system
- Need to replicate input data from outside of sphere of replication to send to redundant threads
- Need to compare and validate output before sending it out of the sphere of replication



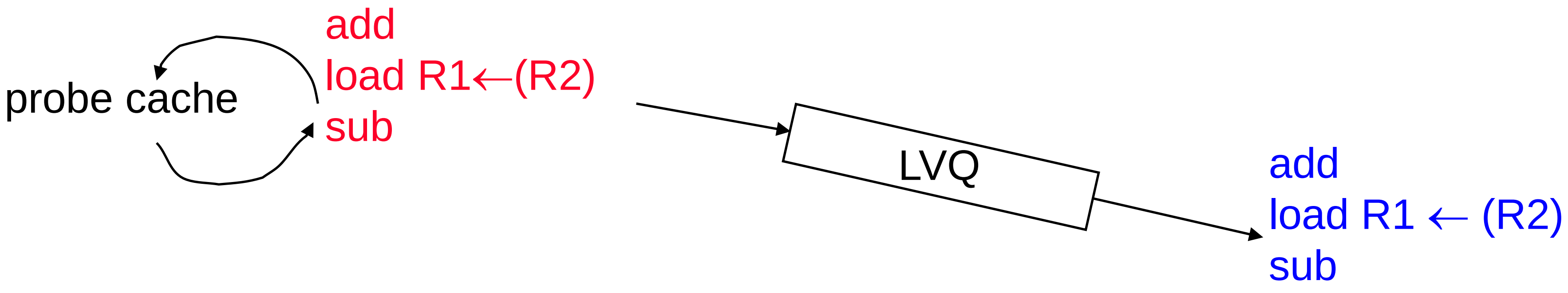
Sphere of Replication in SRT



Input Replication

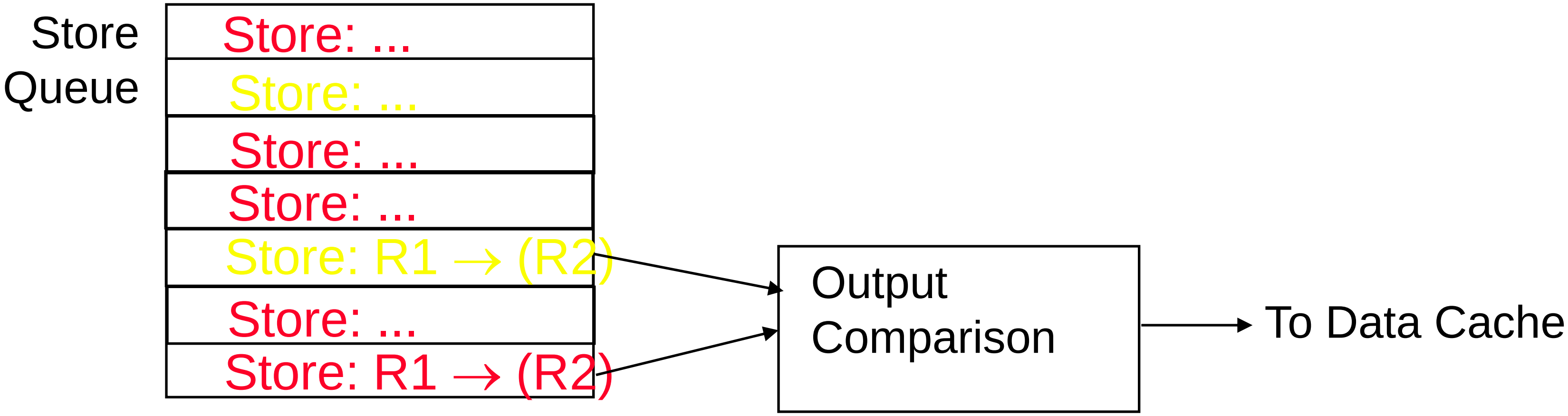
- How to get the load data for redundant threads
  - pair loads from redundant threads and access the cache when both are ready: too slow – threads fully synchronized
  - allow both loads to probe cache separately: false alarms with I/O or multiprocessors

- Load Value Queue (LVQ)
  - pre-designated leading & trailing threads



Output Comparison

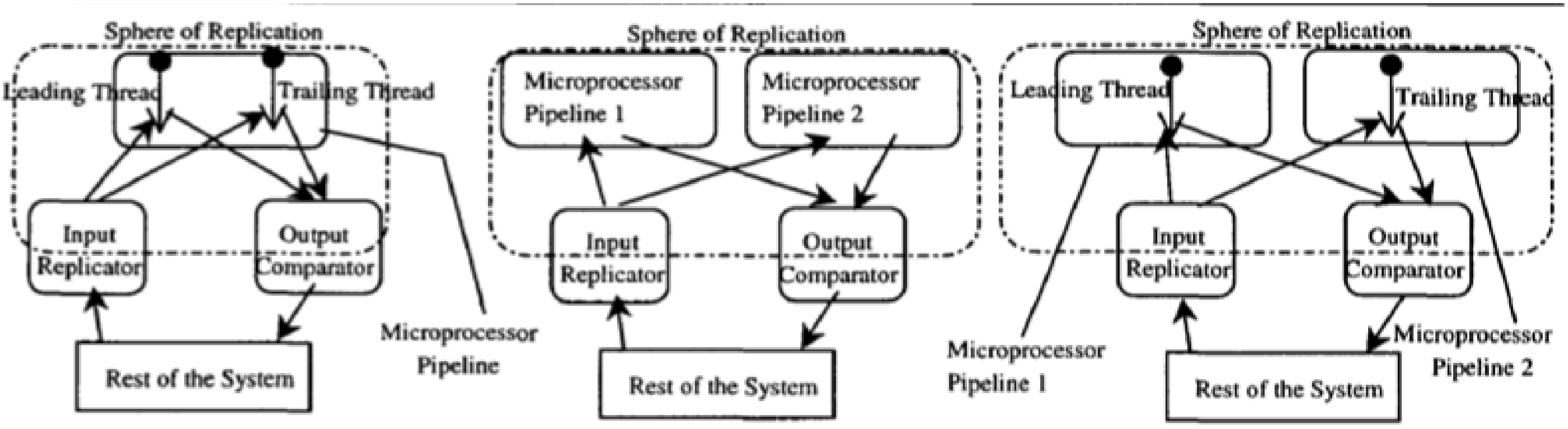
- <address, data> for stores from redundant threads
  - compare & validate at commit time



- How to handle cached vs. uncacheable loads
- Stores now need to live longer to wait for trailing thread
- Need to ensure matching trailing store can commit

SRT Performance Optimizations

- Many performance improvements possible by supplying results from the leading thread to the trailing thread: branch outcomes, instruction results, etc
- Mukherjee et al., "Detailed Design and Evaluation of Redundant Multithreading Alternatives," ISCA 2002.



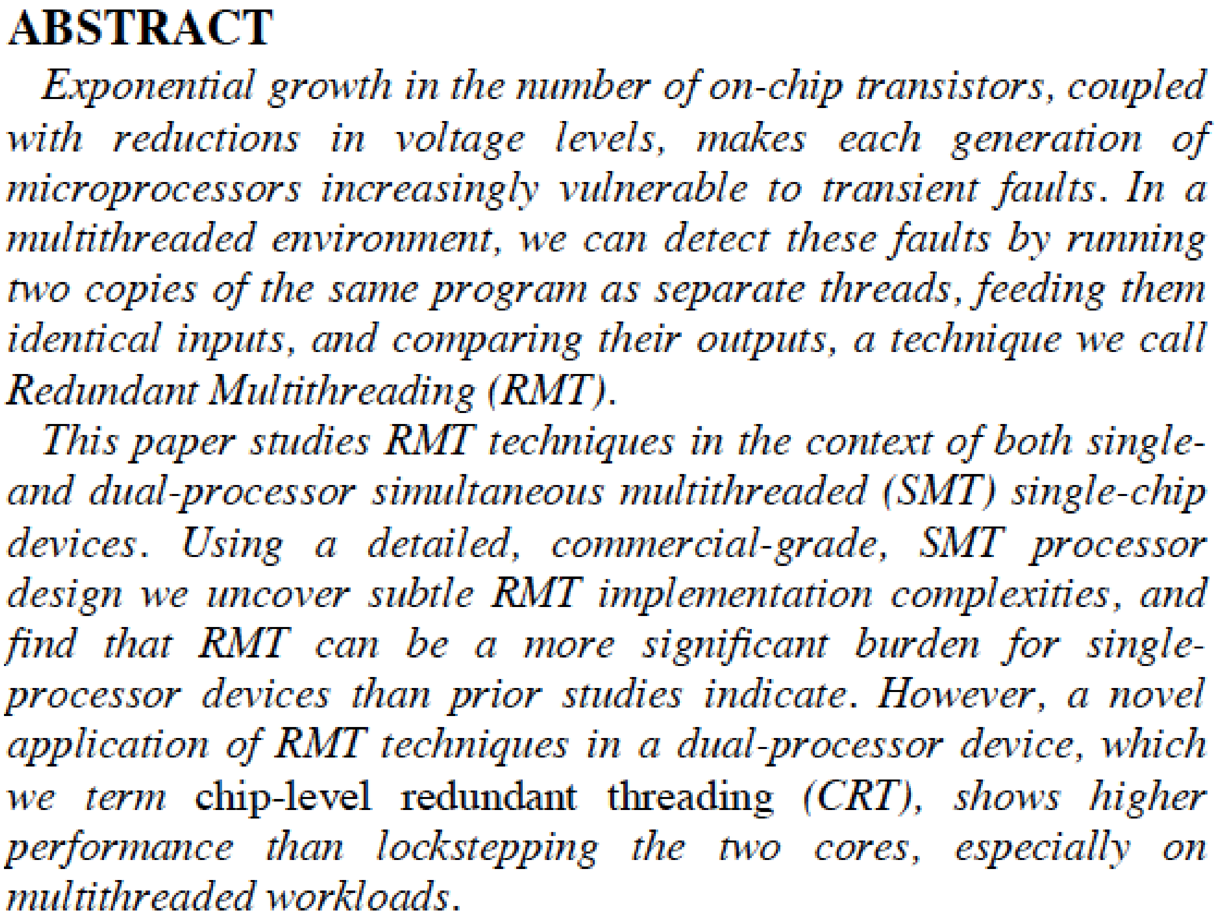
(a) Simultaneous and Redundantly Threaded Processor (SRT)

(b) Lockstepped Microprocessor Pipelines

(c) Chip-Level Redundantly Threaded Processor (CRT)

Recommended Reading

- Mukherjee et al., "Detailed Design and Evaluation of Redundant Multithreading Alternatives," ISCA 2002.

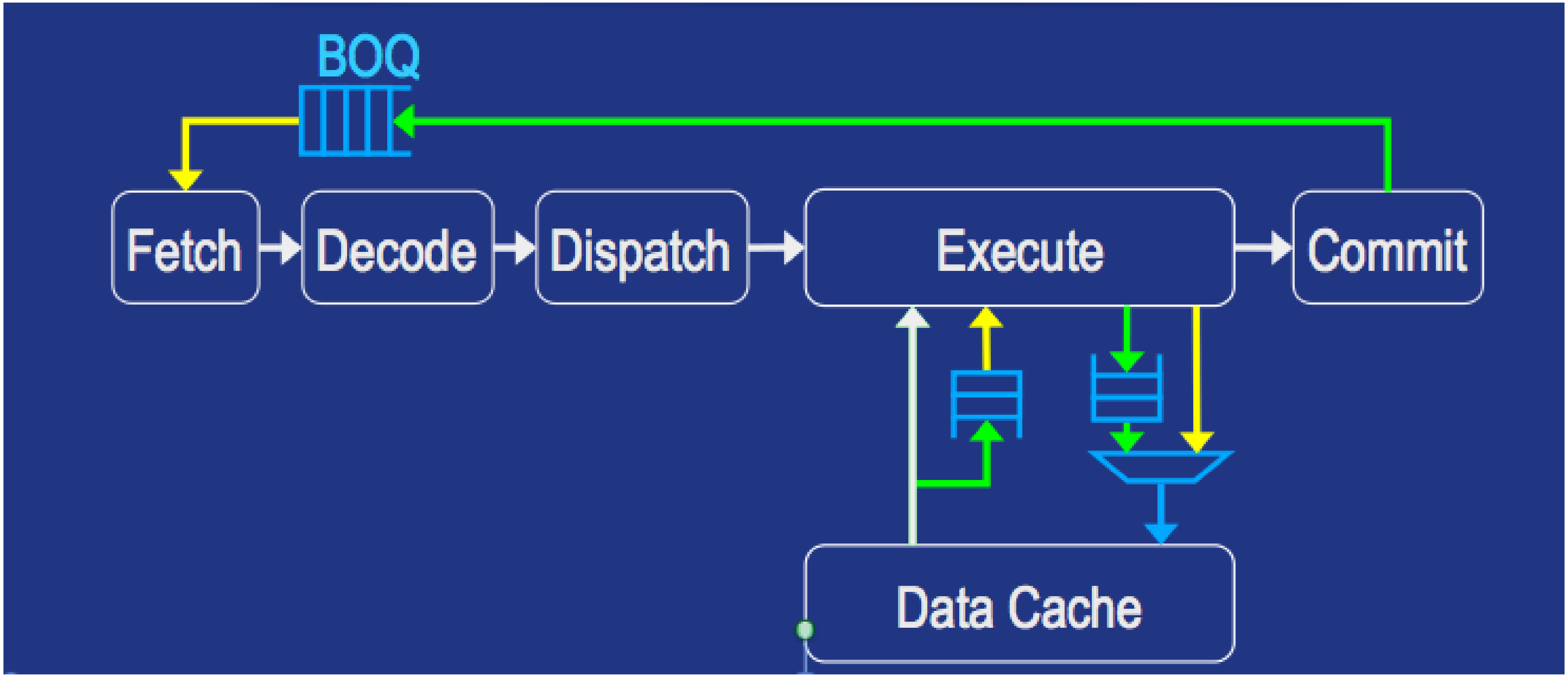


ABSTRACT

Exponential growth in the number of on-chip transistors, coupled with reductions in voltage levels, makes each generation of microprocessors increasingly vulnerable to transient faults. In a multithreaded environment, we can detect these faults by running two copies of the same program as separate threads, feeding them identical inputs, and comparing their outputs, a technique we call Redundant Multithreading (RMT).

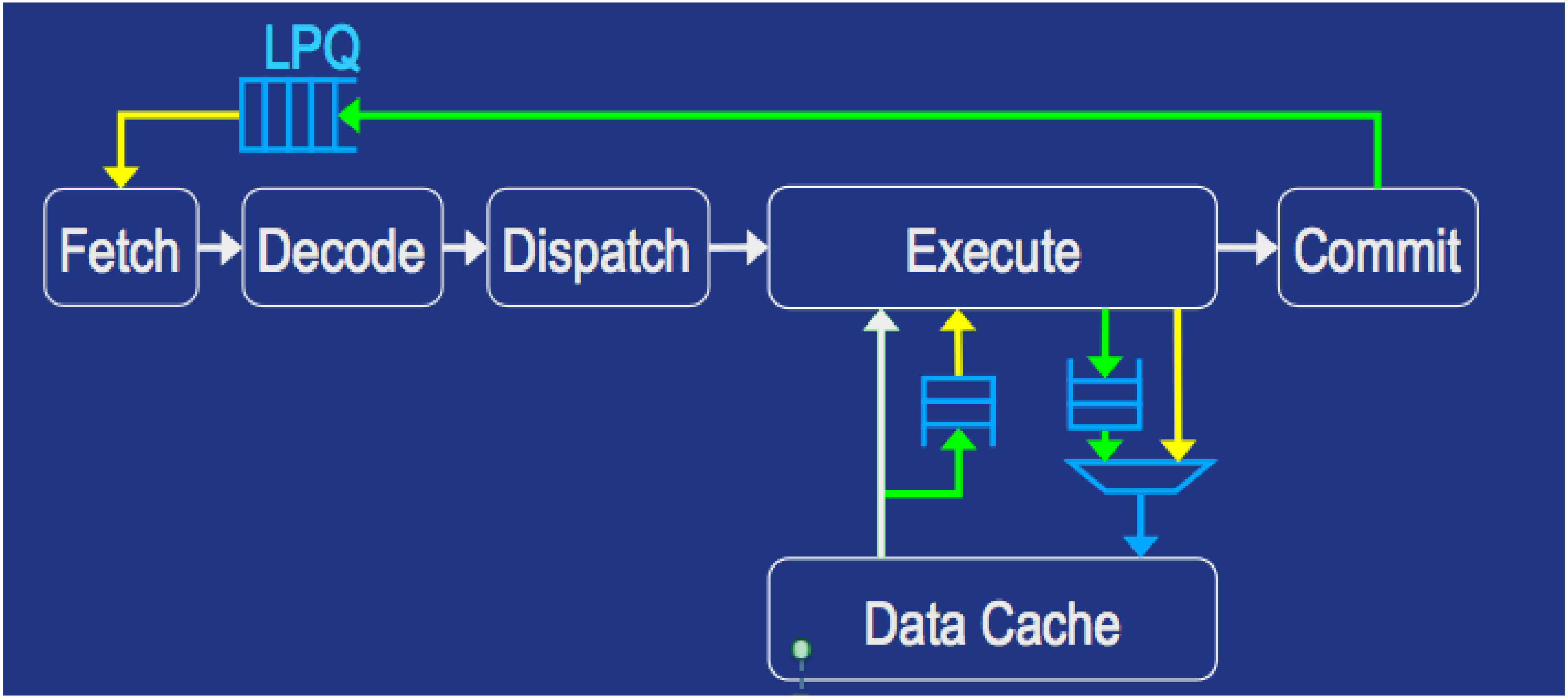
This paper studies RMT techniques in the context of both single- and dual-processor simultaneous multithreaded (SMT) single-chip devices. Using a detailed, commercial-grade, SMT processor design we uncover subtle RMT implementation complexities, and find that RMT can be a more significant burden for single-processor devices than prior studies indicate. However, a novel application of RMT techniques in a dual-processor device, which we term chip-level redundant threading *(CRT), shows higher performance than lockstepping the two cores, especially on multithreaded workloads.*

Branch Outcome Queue



Line Prediction Queue

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



Handling of Permanent Faults via SRT

- SRT uses time redundancy
  - Is this enough for detecting permanent faults?
  - Can SRT detect some permanent faults? How?

- Can we incorporate explicit space redundancy into SRT?

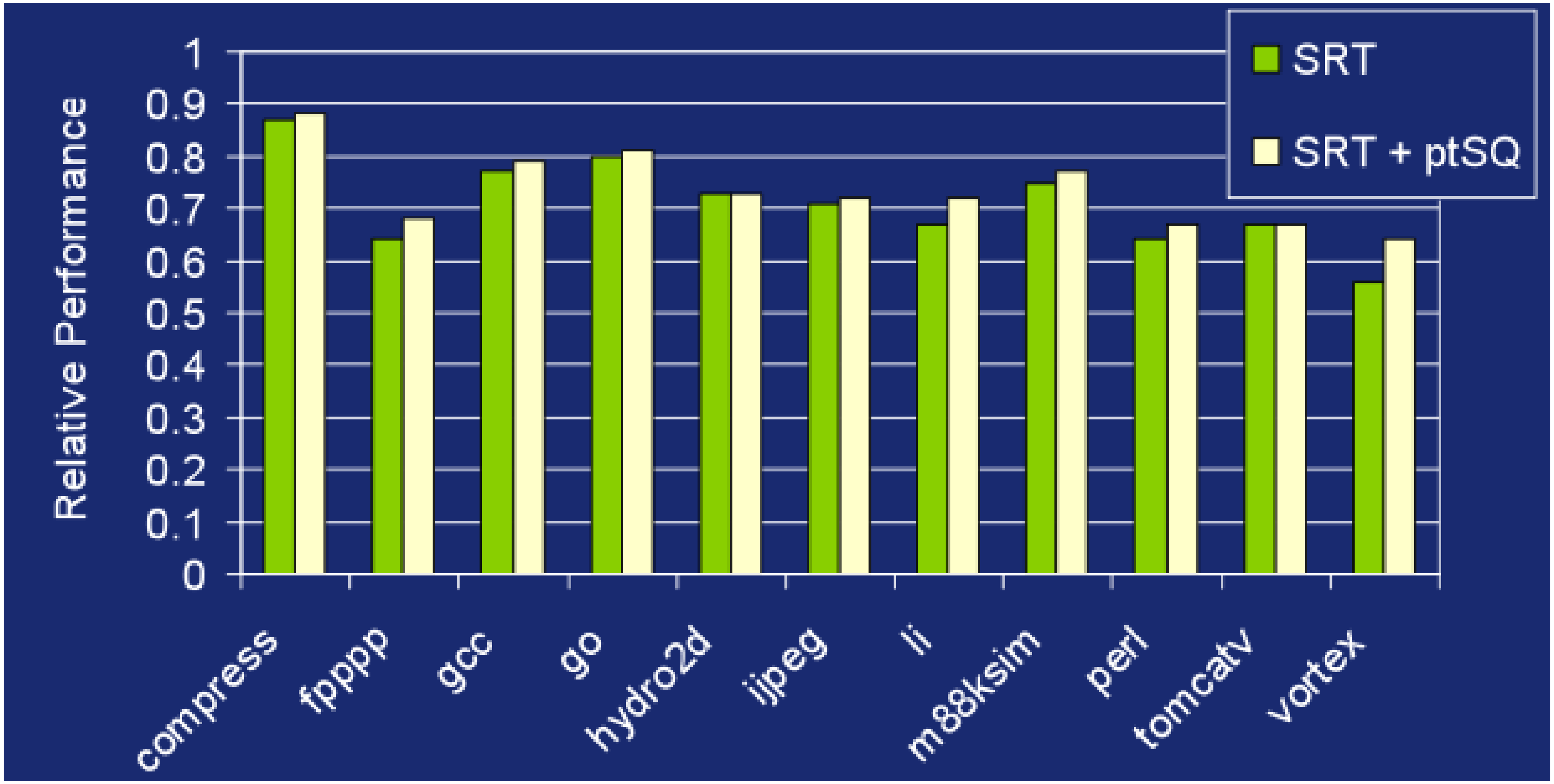
- Idea: Execute the same instruction on different resources in an SMT engine
  - Send instructions from different threads to different execution units (when possible)

SRT Evaluation

- SPEC CPU95, 15M instrs/thread
  - Constrained by simulation environment
  - → 120M instrs for 4 redundant thread pairs

- Eight-issue, four-context SMT CPU
  - Based on Alpha 21464
  - 128-entry instruction queue
  - 64-entry load and store queues
    - Default: statically partitioned among active threads
  - 22-stage pipeline
  - 64KB 2-way assoc. L1 caches
  - 3 MB 8-way assoc L2

Performance Overhead of SRT



- Performance degradation = 30% (and unavailable thread context)
- Per-thread store queue improves performance by 4%

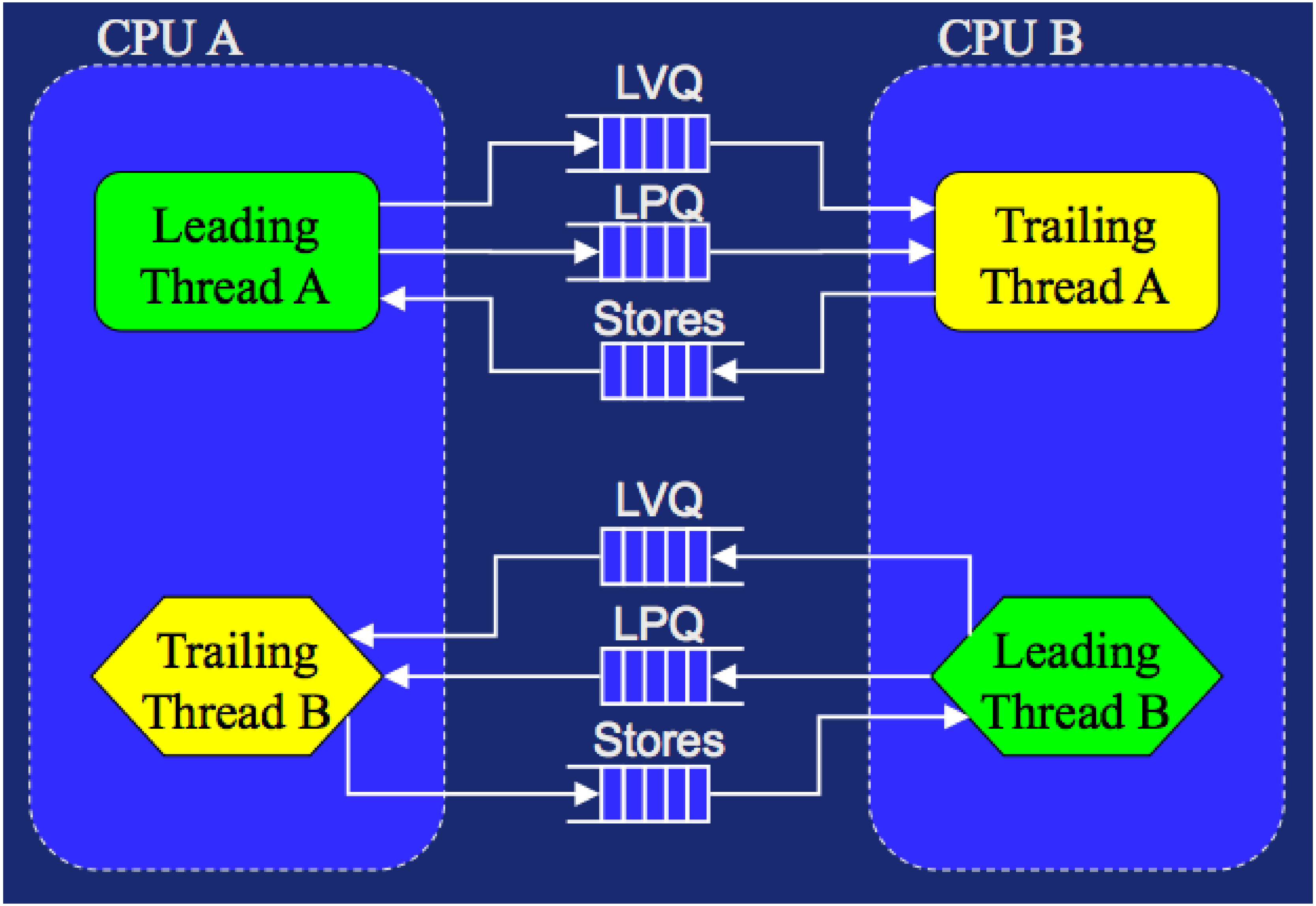
Chip Level Redundant Threading

- SRT typically more efficient than splitting one processor into two half-size cores
- What if you already have two cores?

- Conceptually easy to run these in lock-step
  - Benefit: full physical redundancy
  - Costs:
    - Latency through centralized checker logic
    - Overheads (e.g., branch mispredictions) incurred twice

- We can get both time redundancy and space redundancy if we have multiple SMT cores
  - SRT for CMPs

Chip Level Redundant Threading



Some Other Approaches to Transient Fault Tolerance

- Austin, “DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design,” MICRO 1999.
- Qureshi et al., “Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors,” DSN 2005.

DIVA

- Idea: Have a "functional checker" unit that checks the correctness of the computation done in the "main processor"

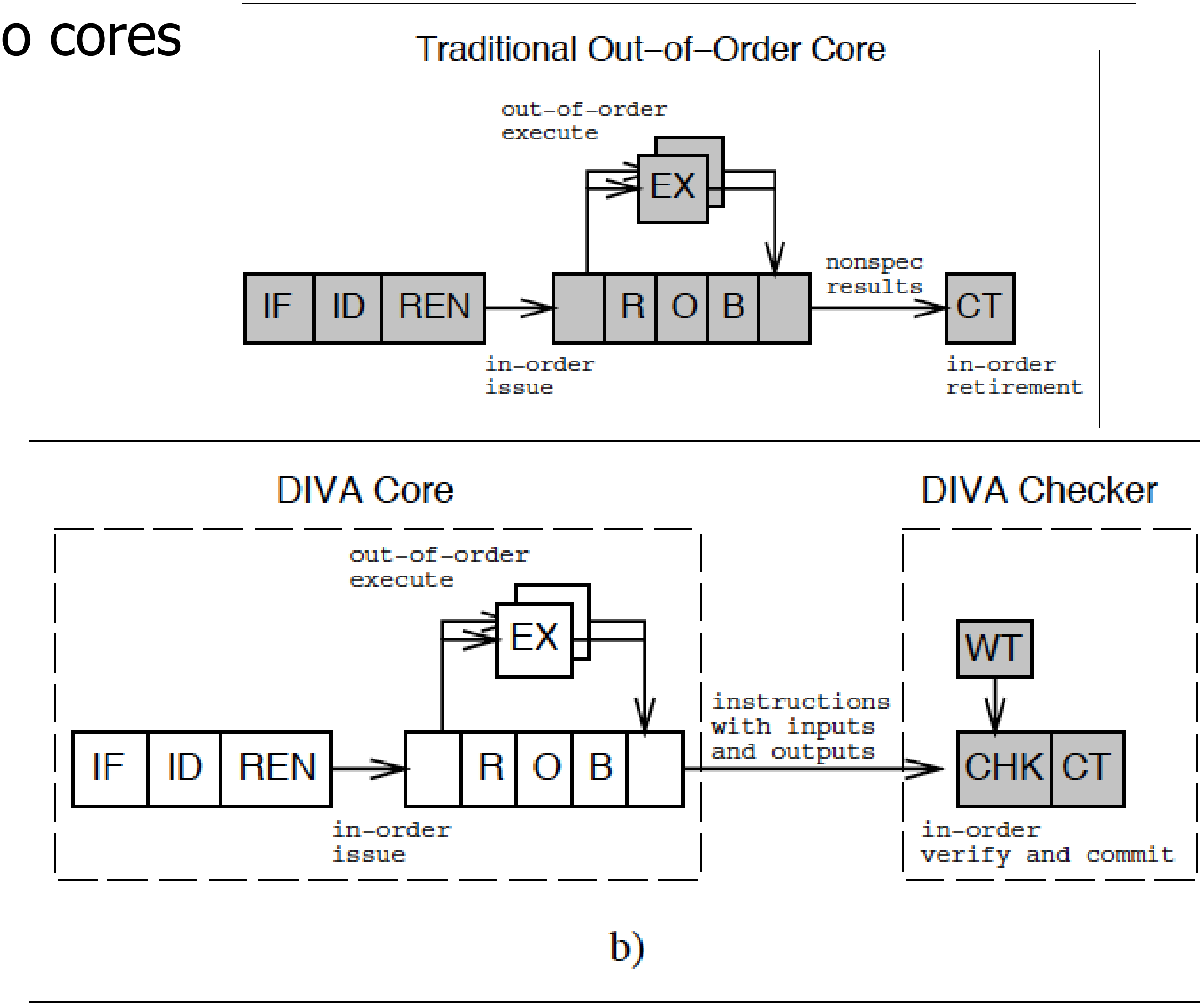
- Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," MICRO 1999.

- Benefit: Main processor can be prone to faults or sometimes incorrect (yet very fast)

- How can checker keep up with the main processor?
  - Verification of different instructions can be performed in parallel (if an older one is incorrect all later instructions will be flushed anyway)

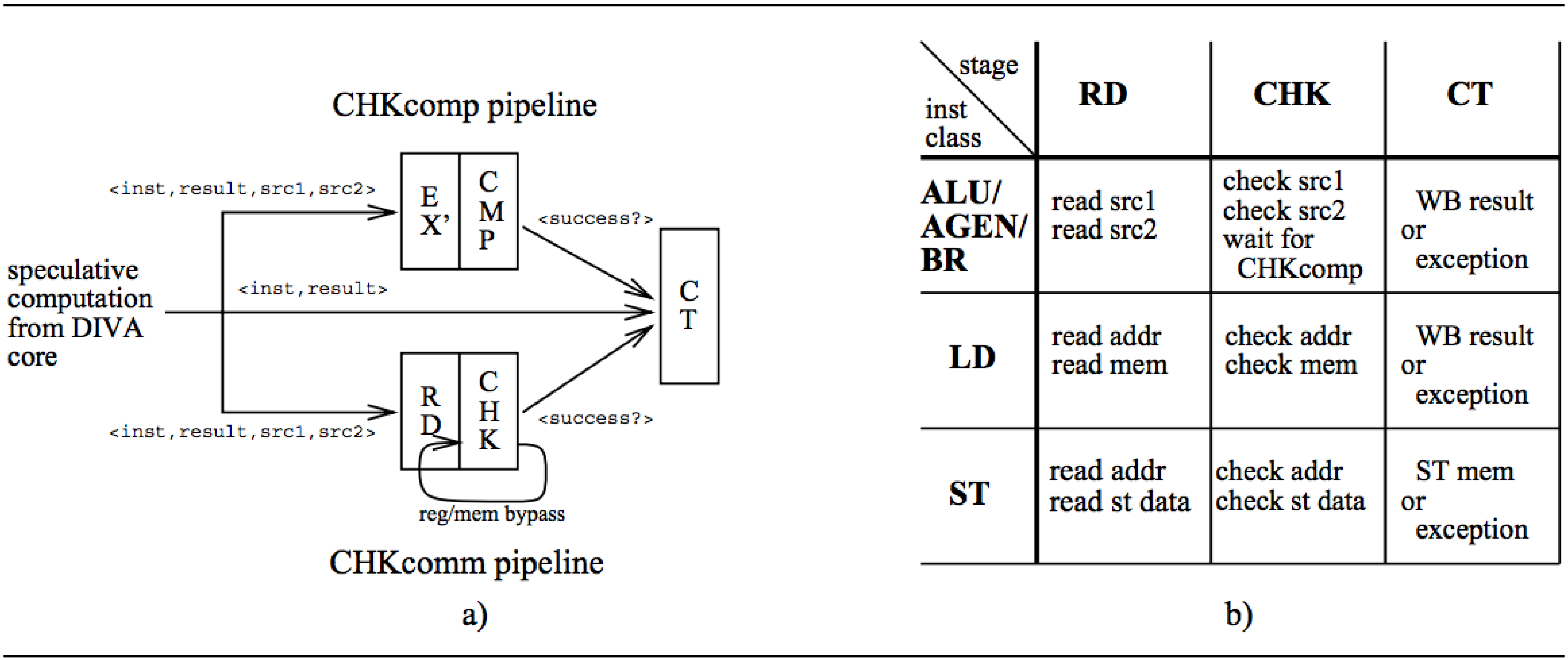
DIVA (Austin, MICRO 1999)

- Two cores



b)

DIVA Checker for One Instruction



stage inst class	RD	CHK	CT
ALU/ AGEN/ BR	read src1 read src2	check src1 check src2 wait for CHKcomp	WB result or exception
LD	read addr read mem	check addr check mem	WB result or exception
ST	read addr read st data	check addr check st data	ST mem or exception

Figure 2. A Dynamic Implementation Verification Architecture (DIVA). Figure a) illustrates the DIVA architecture and its interface to the core processor. Figure b) details the DIVA CHKcomm pipeline operation for each instruction class.

A Self-Tuned System using DIVA

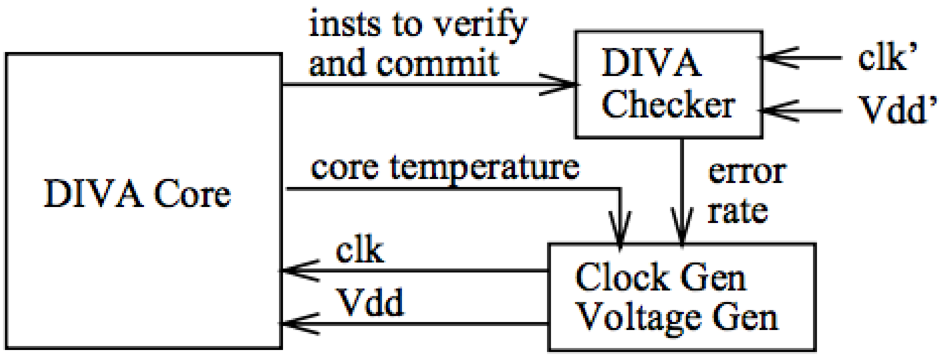


Figure 8. A Self-Tuned System.

In a *self-tuned system* [19], clock frequency and voltage levels are tuned to the system operating environment, *e.g.*, temperature. The approach minimizes timing and voltage margins which can improve performance and reduce power consumption. Using the DIVA checker, a self-tuned system could be constructed by introducing a voltage and frequency control system into the processor, as shown in Figure 8. The control system decreases voltage and/or increases frequency while monitoring system temperature and error rates until the desired system performance-power characteristics are attained. If the control system over steps the bounds of correct operation in the core, the DIVA checker will correct the error, reset the core processor, and notify the control system. To ensure correct operation of the DIVA checker, it is sourced by a fixed voltage and frequency that ensures reliable operation under all operating conditions.

DIVA Discussion

- Upsides?

- Downsides?

Some Other Approaches to Transient Fault Tolerance

- Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," MICRO 1999.

- Qureshi et al., "Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors," DSN 2005.

Microarchitecture Based Introspection

- Idea: Use cache miss stall cycles to redundantly execute the program instructions

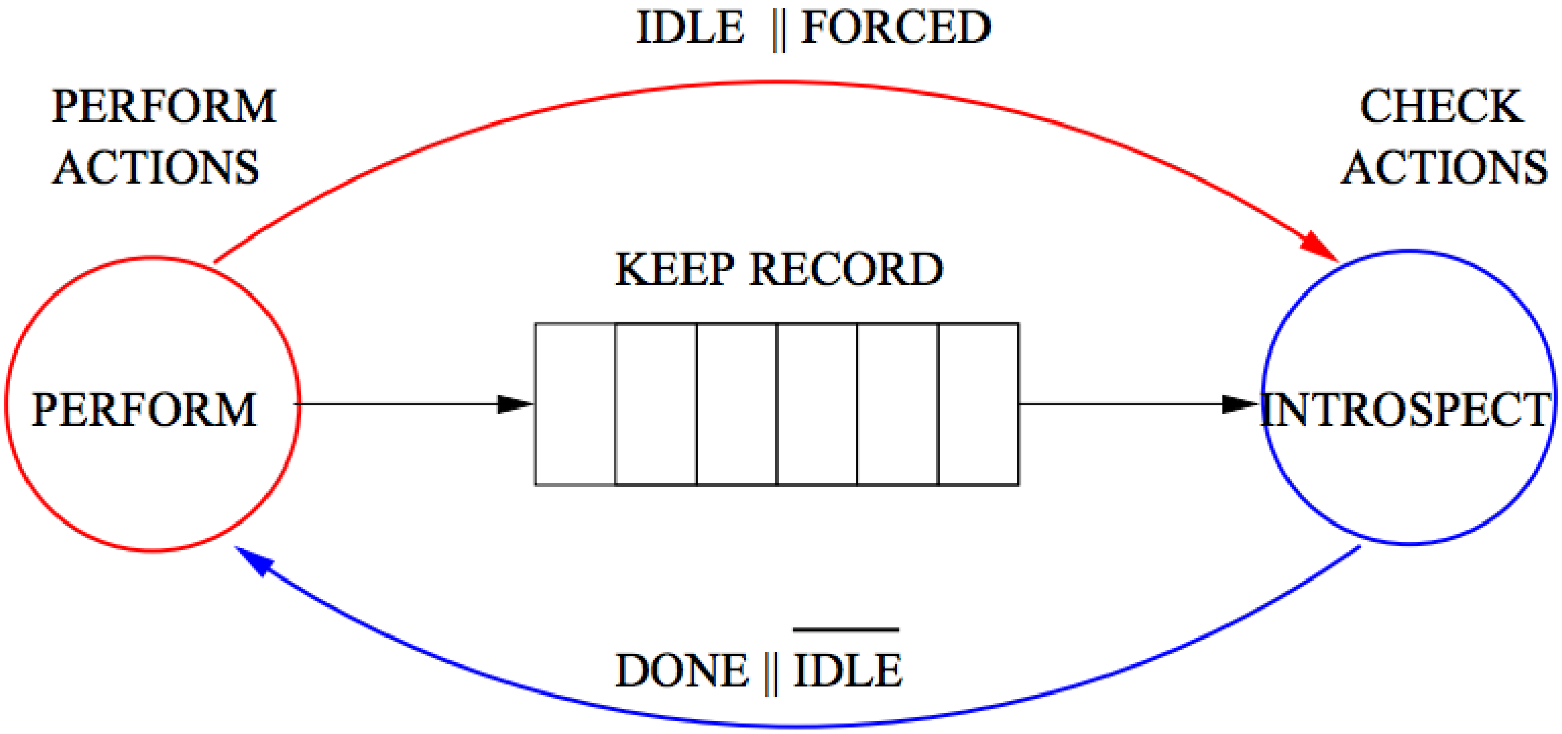
- Qureshi et al., "Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors," DSN 2005.

- Benefit: Redundant execution does not have high performance overhead (when there are stall cycles)

- Downside: What if there are no/few stall cycles?

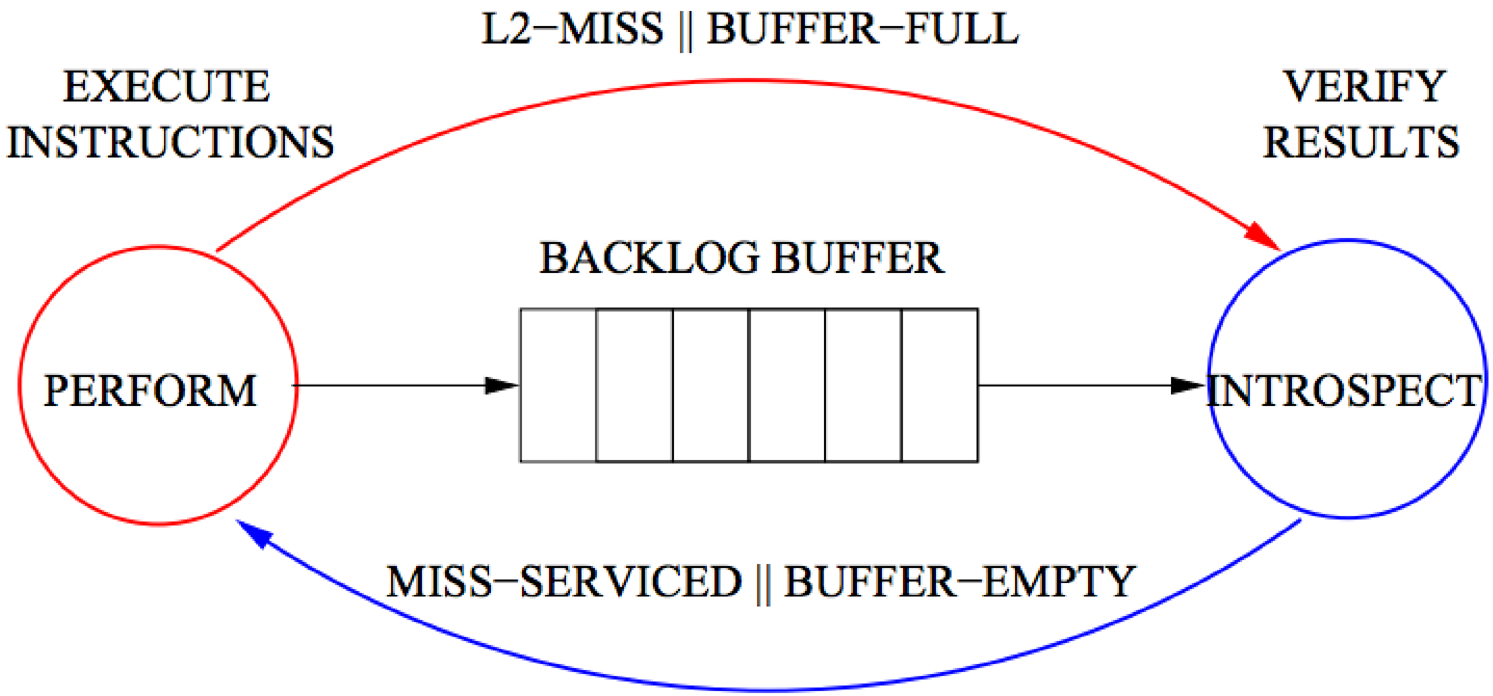
Introspection

An example from the human domain:



MBI (Qureshi+, DSN 2005)

Extending it to the microarchitecture domain:



Microarchitecture-Based Introspection (MBI)

MBI Microarchitecture

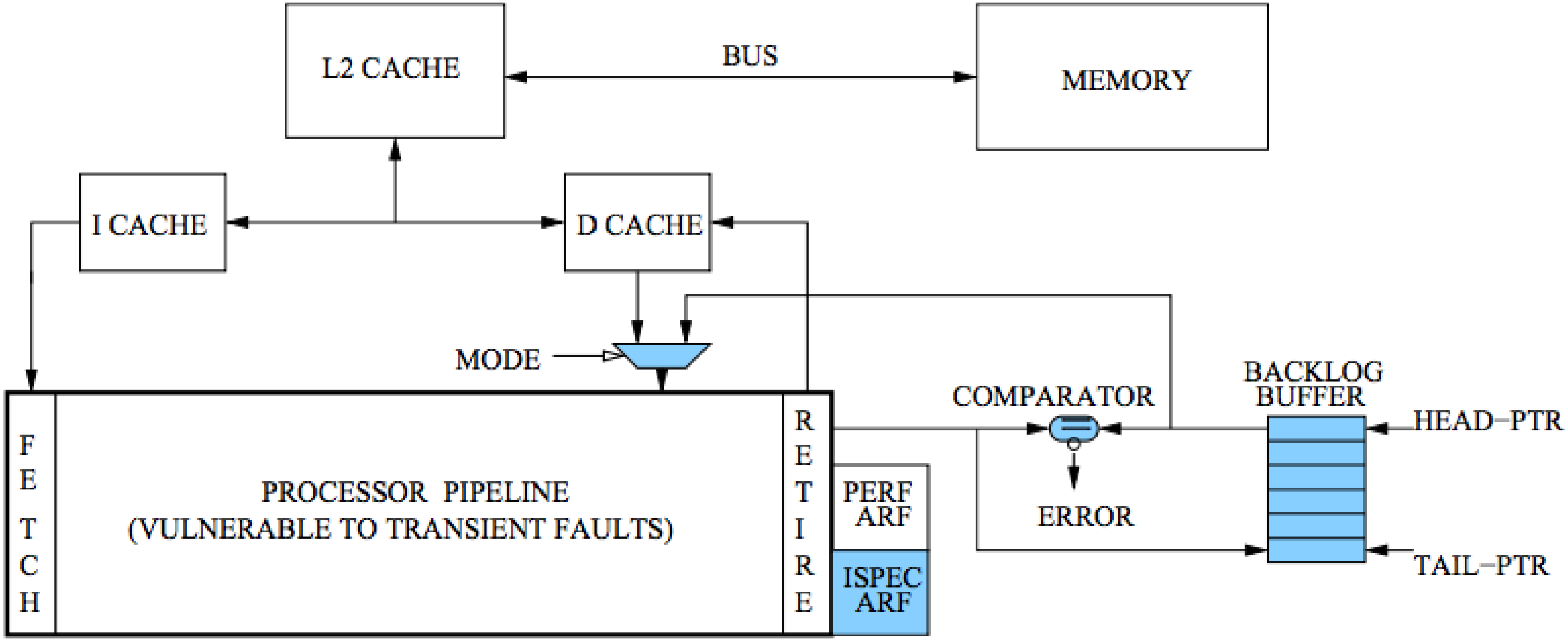


Figure 3. Microarchitecture support for MBI.

Performance Impact of MBI

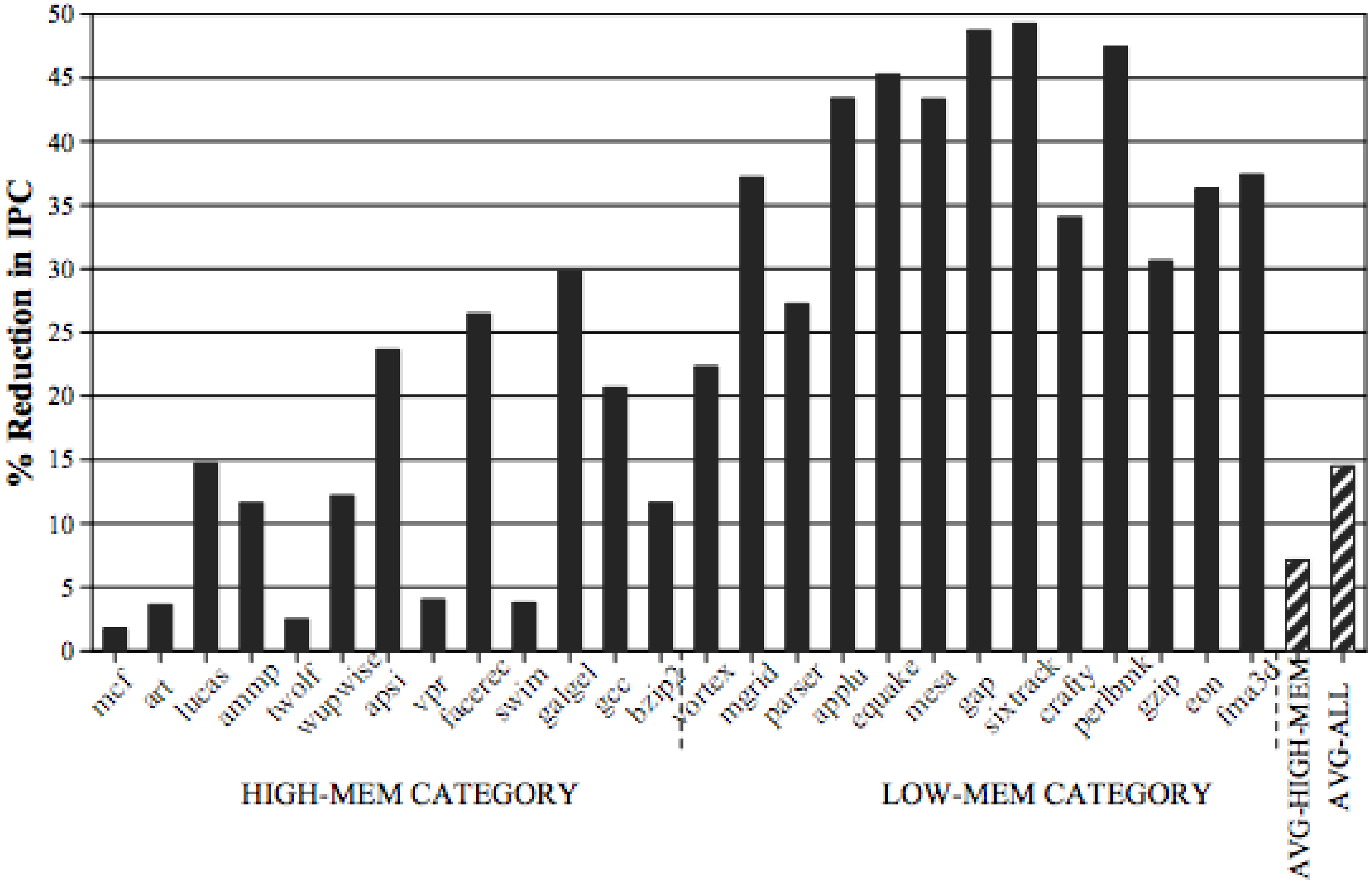


Figure 4. IPC reduction due to the MBI mechanism.

Food for Thought

- Do you need to check that the result of **every** instruction is correct?

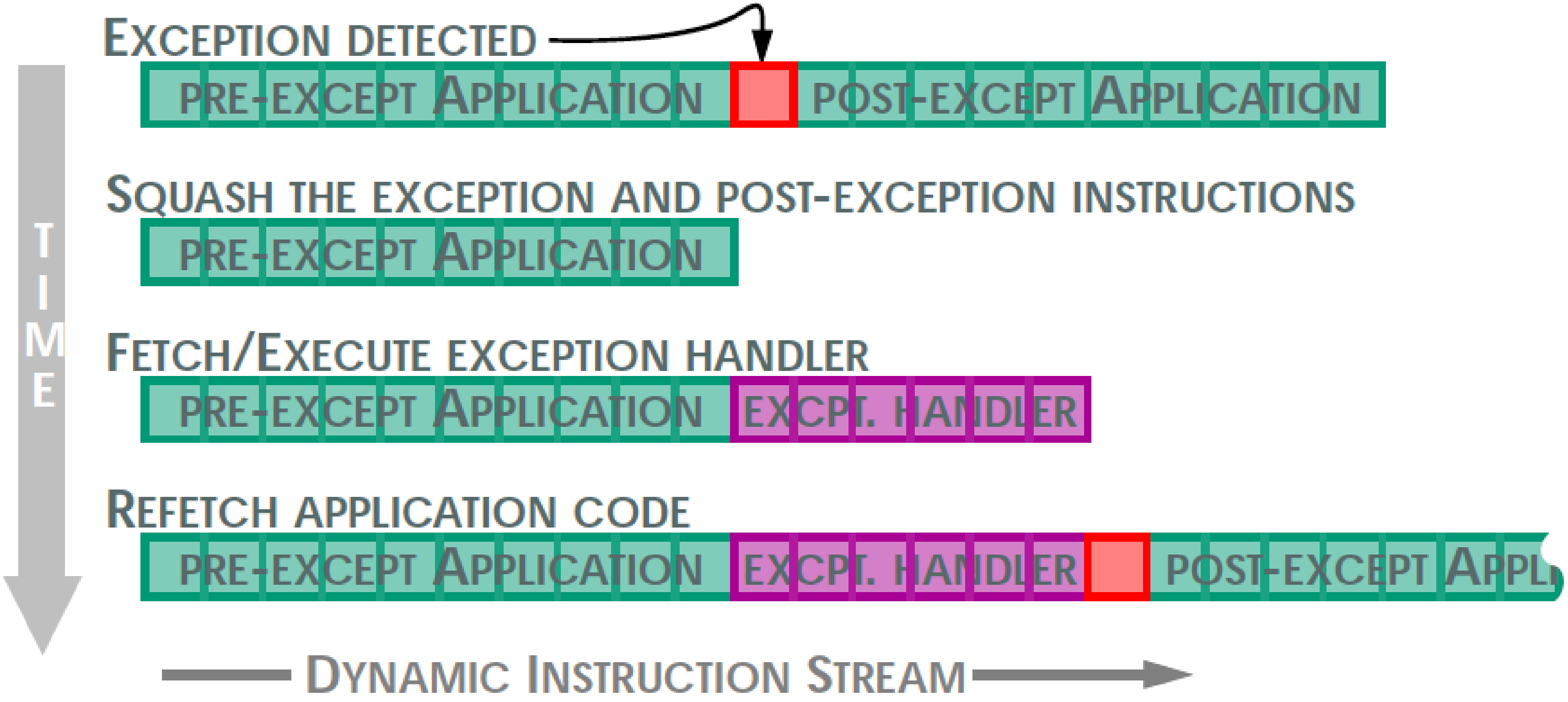
- Do you need to check that the result of **any** instruction is correct?

- What do you really need to check for to ensure correct operation?
  - Soft errors?
  - Hard errors?

Other Uses of Multithreading

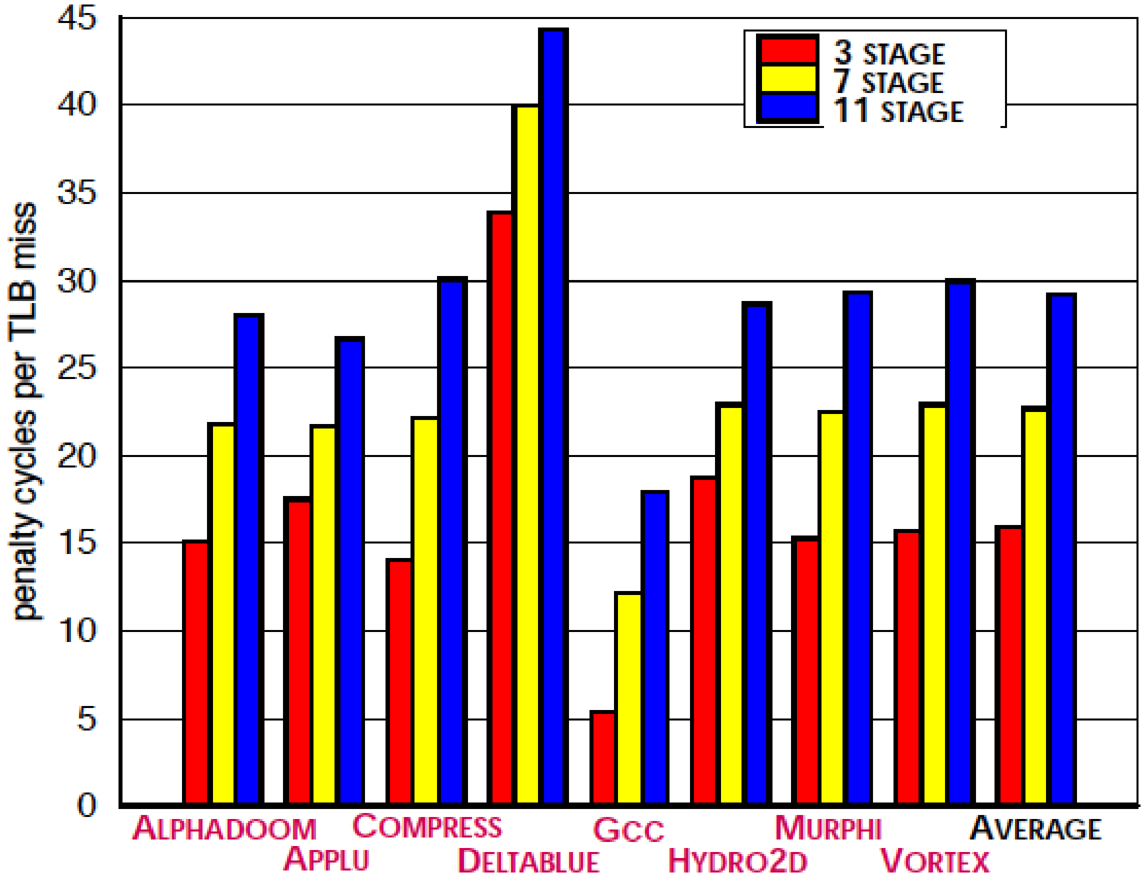
MT for Exception Handling

- Exceptions cause overhead (especially if handled in software)
- Some exceptions are recoverable from (TLB miss, unaligned access, emulated instructions)
- Pipe flushes due to exceptions reduce thread performance



MT for Exception Handling

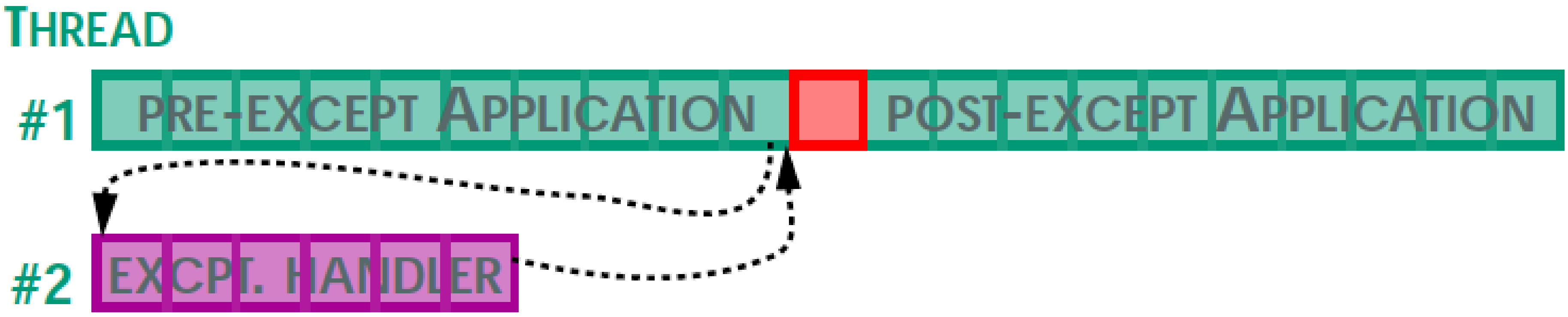
- Cost of software TLB miss handling
- Zilles et al., "The use of multithreading for exception handling," MICRO 1999.



MT for Exception Handling

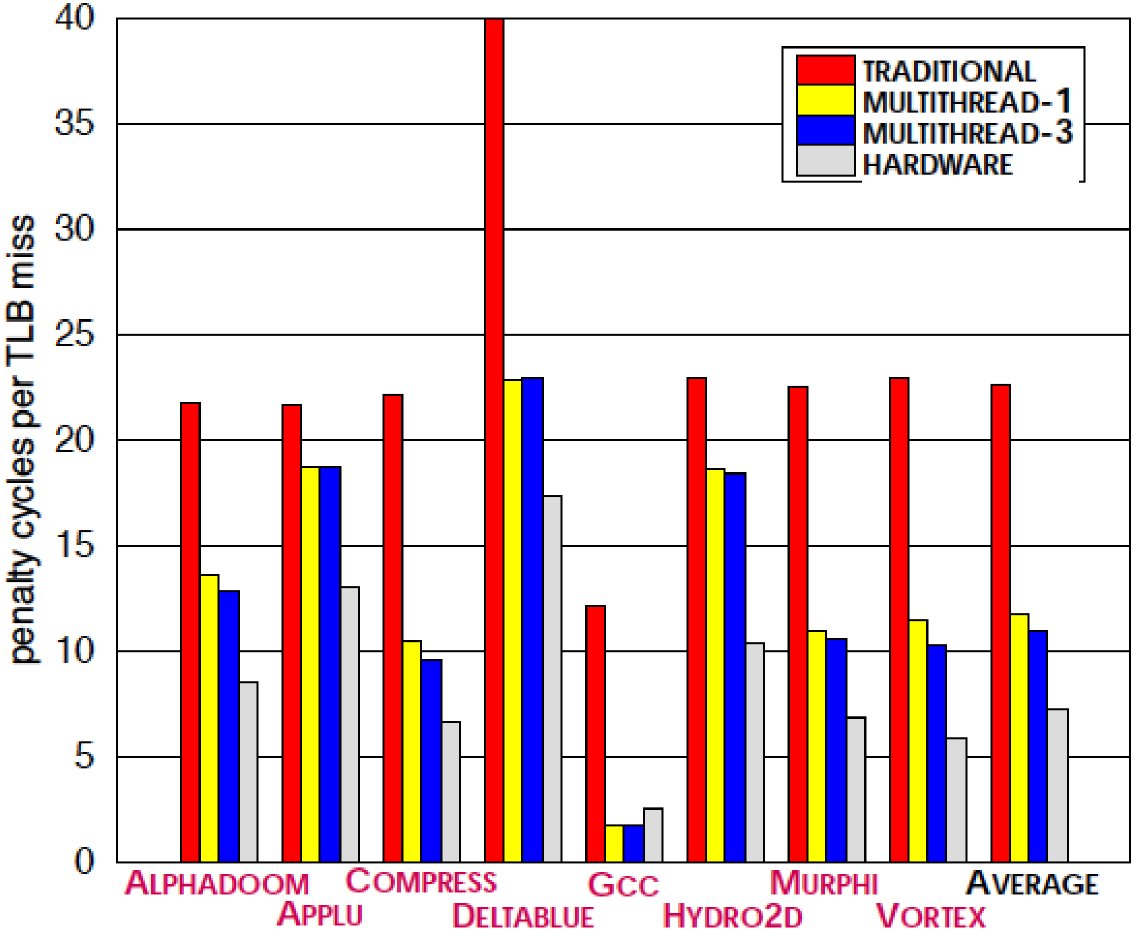
- Observation:
  - The same application instructions are executed in the same order INDEPENDENT of the exception handler's execution
  - The data dependences between the thread and exception handler are minimal

- Idea: Execute the exception handler in a separate thread context; ensure appearance of sequential execution



MT for Exception Handling

- Better than pure software, not as good as pure hardware handling



Why These Uses?

- What benefit of multithreading hardware enables them?

- Ability to communicate/synchronize with very low latency between threads
  - Enabled by proximity of threads in hardware
  - Multi-core has higher latency to achieve this

Helper Threading for Prefetching

- Idea: Pre-execute a piece of the (pruned) program solely for prefetching data
  - Only need to distill pieces that lead to cache misses

- Speculative thread: Pre-executed program piece can be considered a "thread"

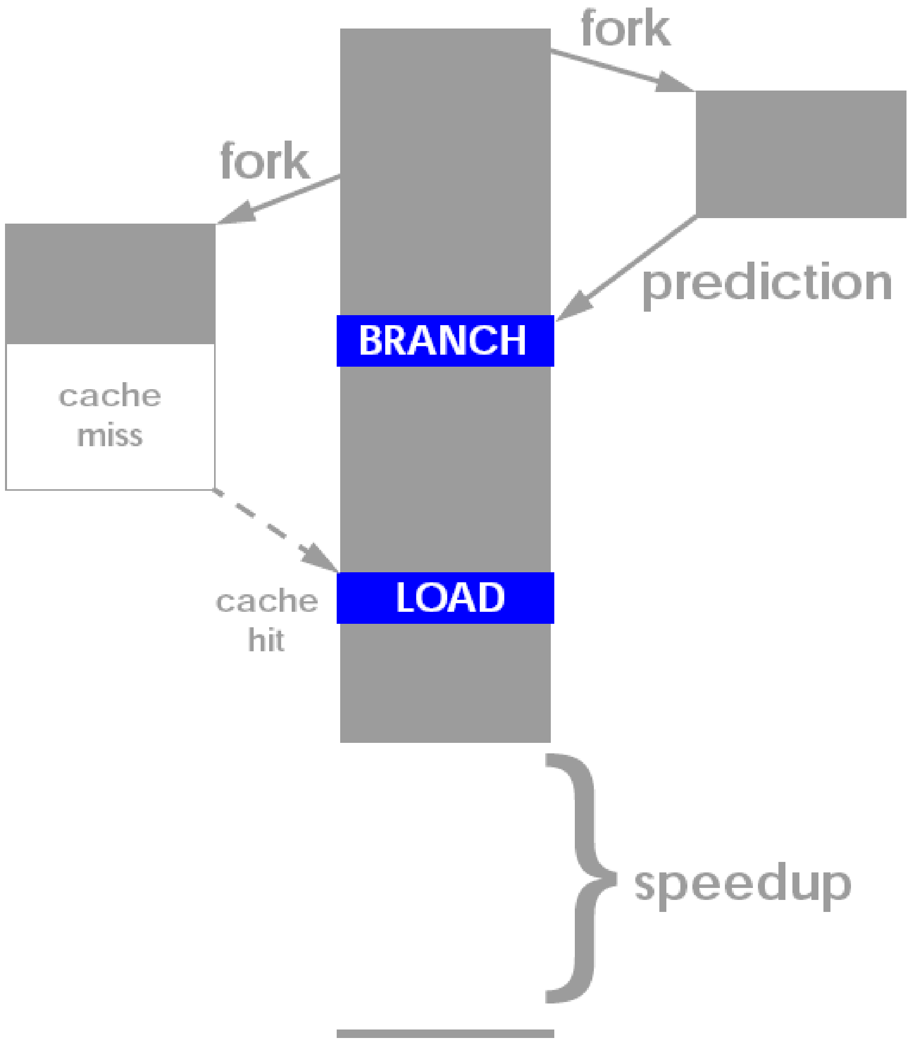
- Speculative thread can be executed
- On a separate processor/core
- On a separate hardware thread context
- On the same thread context in idle cycles (during cache misses)

Helper Threading for Prefetching

- How to construct the speculative thread:
  - Software based pruning and “spawn” instructions
  - Hardware based pruning and “spawn” instructions
  - Use the original program (no construction), but
    - Execute it faster without stalling and correctness constraints

- Speculative thread
  - Needs to discover misses before the main program
    - Avoid waiting/stalling and/or compute less
  - To get ahead, uses
    - Branch prediction, value prediction, only address generation computation

Generalized Thread-Based Pre-Execution



- Dubois and Song, "Assisted Execution," USC Tech Report 1998.
- Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," ISCA 1999.
- Zilles and Sohi, "Execution-based Prediction Using Speculative Slices", ISCA 2001.

Thread-Based Pre-Execution Issues

- Where to execute the precomputation thread?

  1. Separate core (least contention with main thread)
  2. Separate thread context on the same core (more contention)
  3. Same core, same context
     - When the main thread is stalled

- When to spawn the precomputation thread?

  1. Insert spawn instructions well before the “problem” load
     - How far ahead?
       - Too early: prefetch might not be needed
       - Too late: prefetch might not be timely
  2. When the main thread is stalled

- When to terminate the precomputation thread?

  1. With pre-inserted CANCEL instructions
  2. Based on effectiveness/contention feedback