

# Computer Architecture: Multithreading (IV)

Prof. Onur Mutlu  
Carnegie Mellon University

# A Note on This Lecture

---

- These slides are partly from 18-742 Fall 2012, Parallel Computer Architecture, Lecture 15: Multithreading Wrap-Up
- Video of that lecture:
  - <http://www.youtube.com/watch?v=-hbmzIDe0sA&list=PL5PHm2jkkXmh4cDkC3s1VBB7-njlgIG5d&index=14>

# 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
-

# Why These Uses?

---

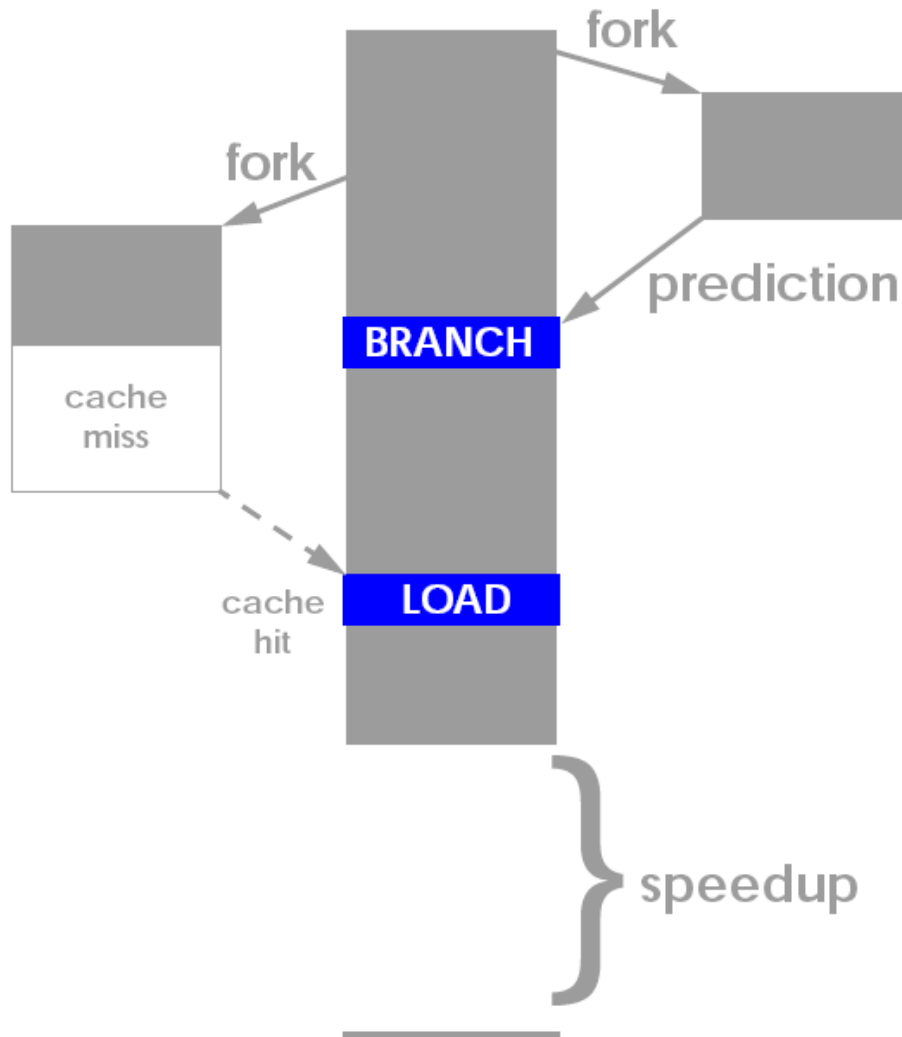
- What benefit of multithreading hardware enables them?
- Ability to communicate/synchronize with very low latency between threads
  - Enabled by proximity of threads in hardware
  - Multi-core has higher latency to achieve this

# 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)

# 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



# Slipstream Processors

---

- Goal: use multiple hardware contexts to speed up single thread execution (implicitly parallelize the program)
- Idea: Divide program execution into two threads:
  - Advanced thread executes a reduced instruction stream, speculatively
  - Redundant thread uses results, prefetches, predictions generated by advanced thread and ensures correctness
- Benefit: Execution time of the overall program reduces
- Core idea is similar to many thread-level speculation approaches, except with a reduced instruction stream
- Sundaramoorthy et al., “Slipstream Processors: Improving both Performance and Fault Tolerance,” ASPLOS 2000.

# Slipstreaming

---

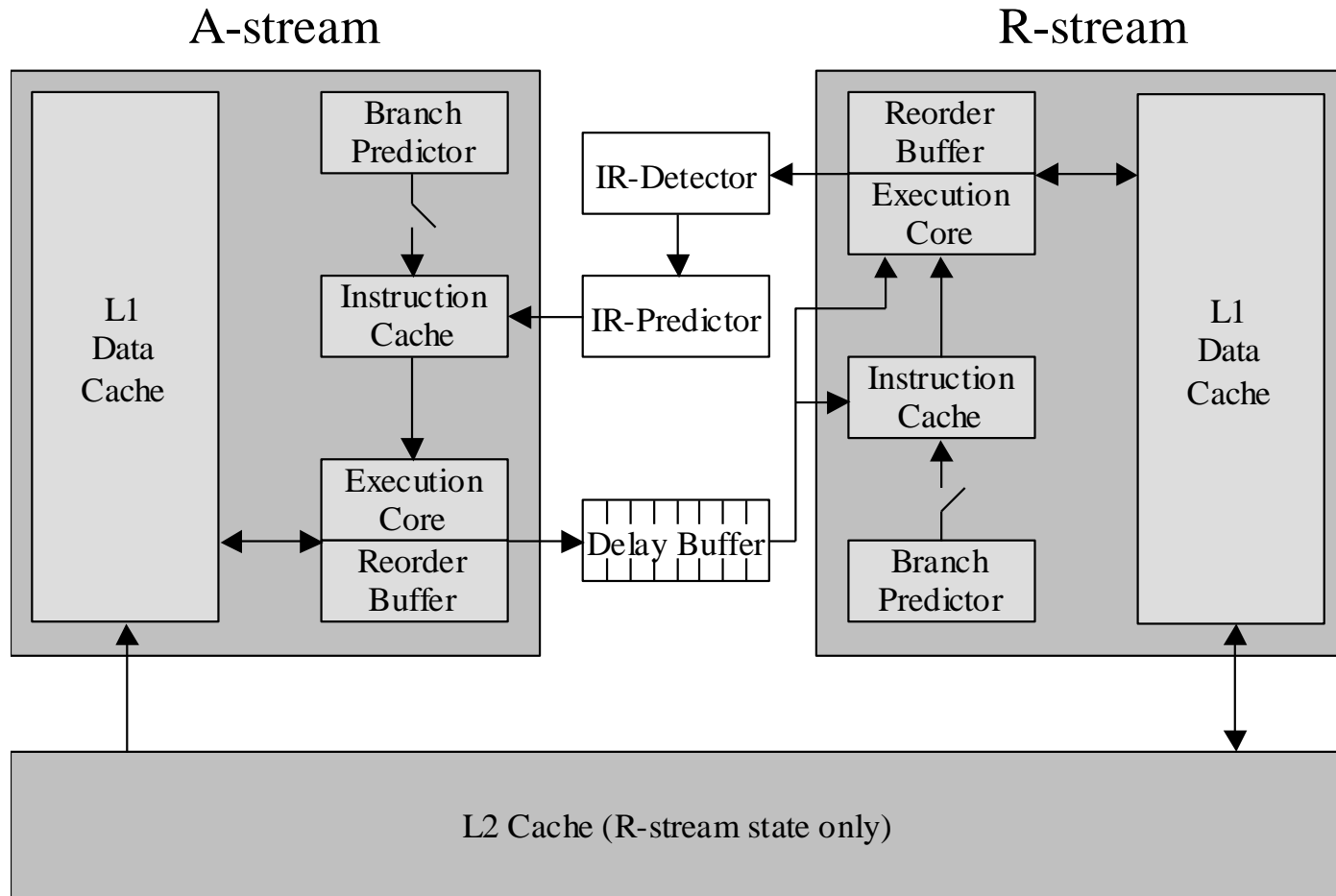
- “At speeds in excess of 190 m.p.h., high air pressure forms at the front of a race car and a partial vacuum forms behind it. This creates drag and limits the car’s top speed.
- A second car can position itself close behind the first (a process called *slipstreaming* or *drafting*). This fills the vacuum behind the lead car, reducing its drag. And the trailing car now has less wind resistance in front (and by some accounts, the vacuum behind the lead car actually helps pull the trailing car).
- As a result, both cars speed up by several m.p.h.: the two combined go faster than either can alone.”

# Slipstream Processors

---

- Detect and remove ineffectual instructions; run a shortened “effectual” version of the program (Advanced or **A-stream**) in one thread context
- Ensure correctness by running a complete version of the program (Redundant or **R-stream**) in another thread context
- Shortened A-stream runs fast; R-stream consumes near-perfect control and data flow outcomes from A-stream and finishes close behind
- Two streams together lead to faster execution (by helping each other) than a single one alone

# Slipstream Idea and Possible Hardware



# Instruction Removal in Slipstream

---

- IR detector
  - Monitors retired R-stream instructions
  - Detects ineffectual instructions and conveys them to the IR predictor
  - Ineffectual instruction examples:
    - dynamic instructions that repeatedly and predictably have no observable effect (e.g., unreferenced writes, non-modifying writes)
    - dynamic branches whose outcomes are consistently predicted correctly.
- IR predictor
  - Removes an instruction from A-stream after repeated indications from the IR detector
- A stream skips ineffectual instructions, executes everything else and inserts their results into delay buffer
- R stream executes all instructions but uses results from the delay buffer as predictions

# What if A-stream Deviates from Correct Execution?

---

- Why
  - A-stream deviates due to incorrect removal or stale data access in L1 data cache
- How to detect it?
  - Branch or value misprediction happens in R-stream (known as an IR misprediction)
- How to recover?
  - Restore A-stream register state: copy values from R-stream registers using delay buffer or shared-memory exception handler
  - Restore A-stream memory state: invalidate A-stream L1 data cache (or speculatively written blocks by A-stream)

# Slipstream Questions

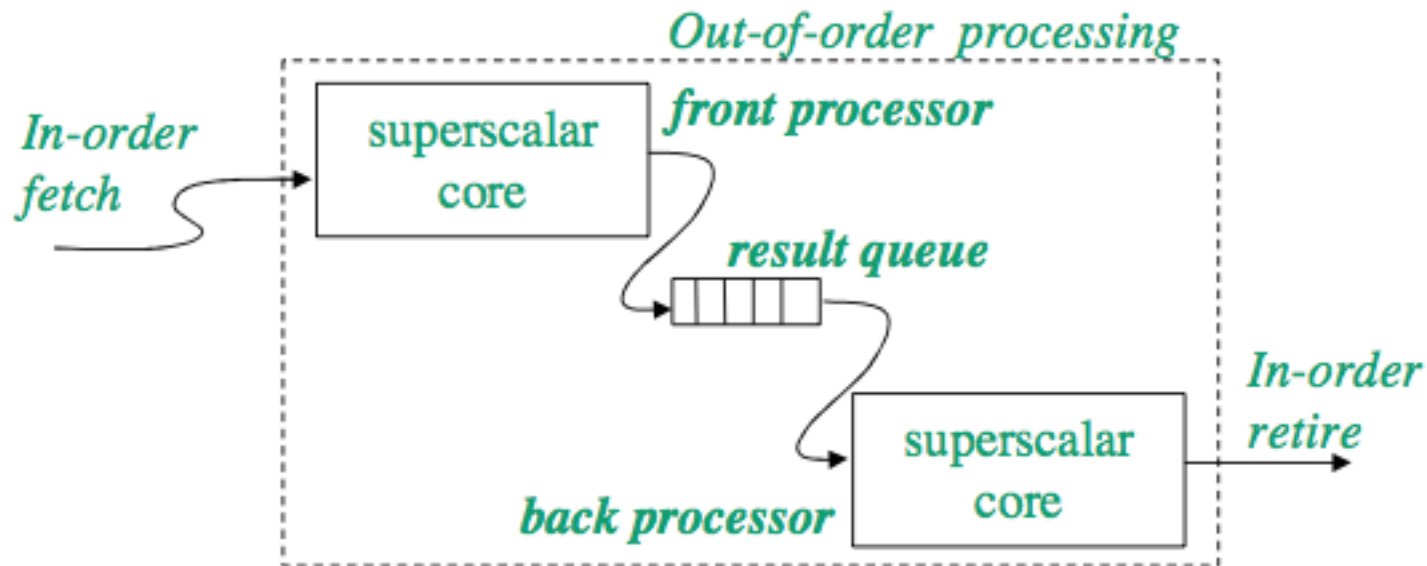
---

- How to construct the advanced thread
  - Original proposal:
    - Dynamically eliminate redundant instructions (silent stores, dynamically dead instructions)
    - Dynamically eliminate easy-to-predict branches
  - Other ways:
    - Dynamically ignore long-latency stalls
    - Static based on profiling
- How to speed up the redundant thread
  - Original proposal: Reuse instruction results (control and data flow outcomes from the A-stream)
  - Other ways: Only use branch results and prefetched data as predictions

# Dual Core Execution

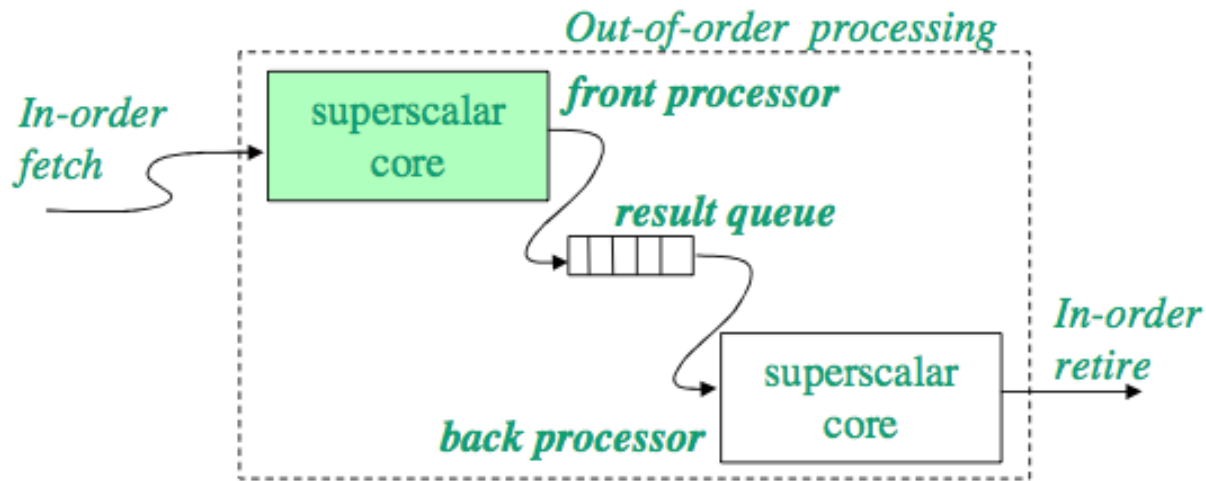
---

- Idea: One thread context speculatively runs ahead on load misses and prefetches data for another thread context
- Zhou, “Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window,” PACT 2005.



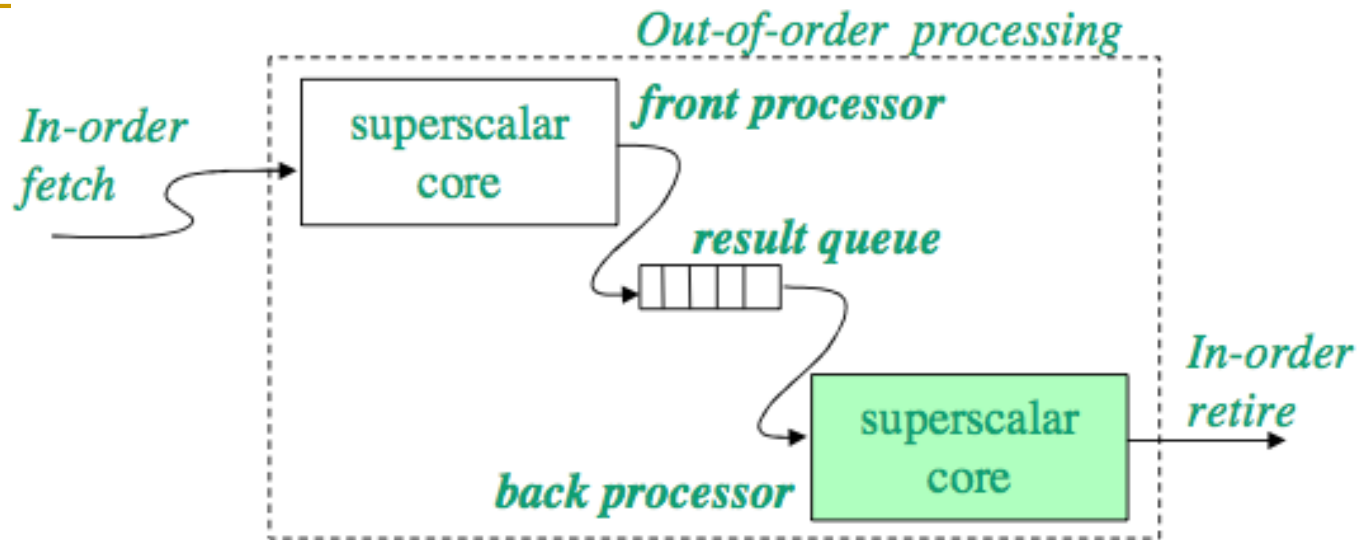


# Dual Core Execution: Front Processor



- The front processor runs faster by invalidating long-latency cache-missing loads, same as runahead execution
  - Load misses and their dependents are invalidated
  - Branch mispredictions dependent on cache misses cannot be resolved
- Highly accurate execution as independent operations are not affected
  - Accurate prefetches to warm up caches
  - Correctly resolved independent branch mispredictions

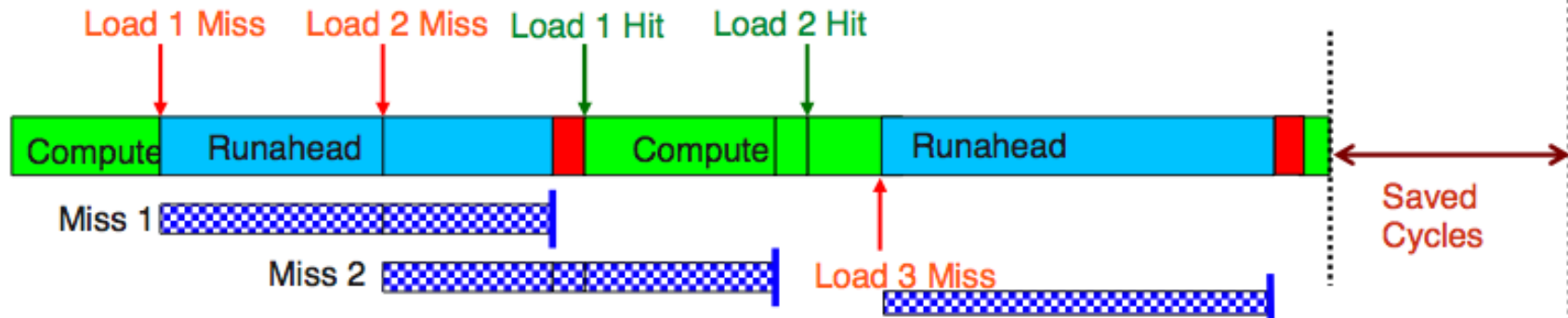
# Dual Core Execution: Back Processor



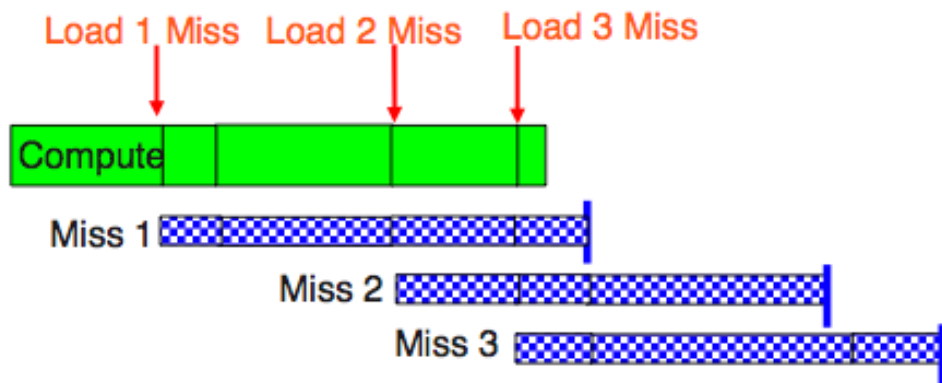
- Re-execution ensures correctness and provides precise program state
  - Resolve branch mispredictions dependent on long-latency cache misses
- Back processor makes faster progress with help from the front processor
  - Highly accurate instruction stream
  - Warmed up data caches

# Dual Core Execution

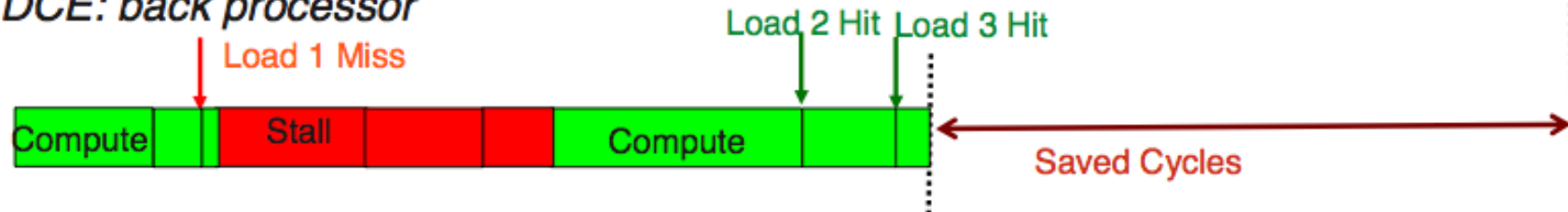
## Runahead:



## DCE: front processor



## DCE: back processor



# DCE Microarchitecture

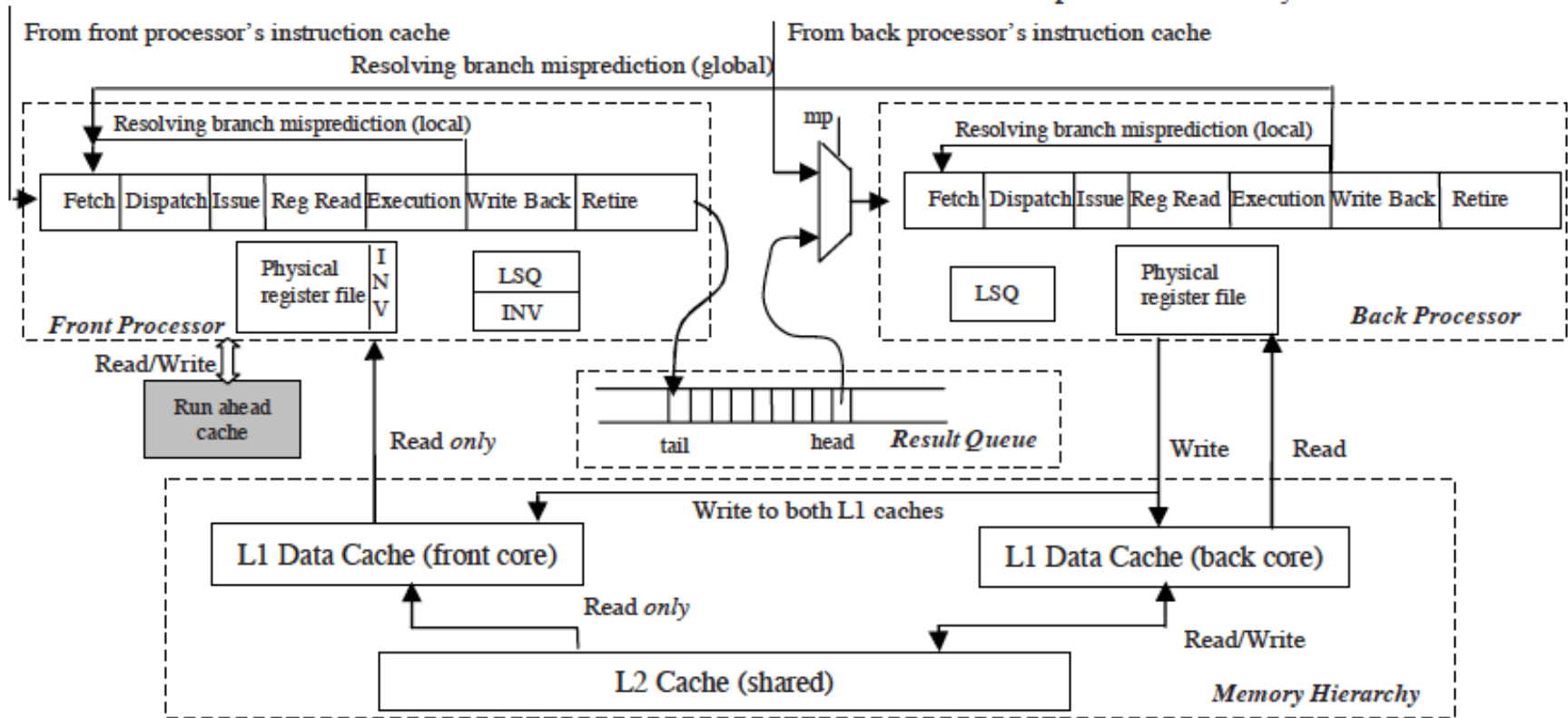


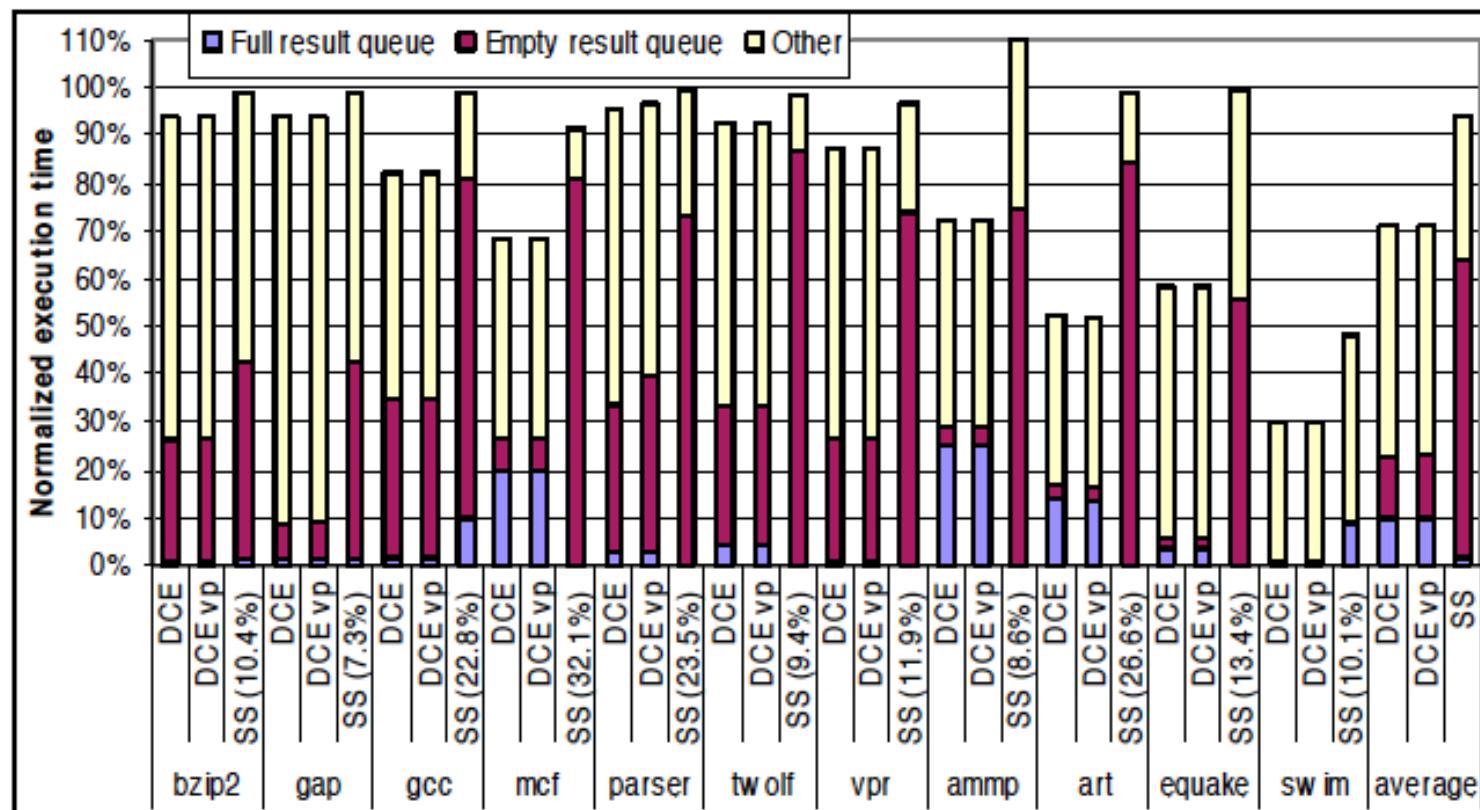
Figure 3. The design of DCE architecture.

# Dual Core Execution vs. Slipstream

---

- Dual-core execution does not
  - remove dead instructions
  - reuse instruction register results
  - uses the “leading” hardware context solely for prefetching and branch prediction
  
- + Easier to implement, smaller hardware cost and complexity
- “Leading thread” cannot run ahead as much as in slipstream when there are no cache misses
- Not reusing results in the “trailing thread” can reduce overall performance benefit

# Some Results



**Figure 6. Normalized execution time of DCE, DCE with value prediction (DCE vp), and slipstreaming processors (SS).**

# Thread Level Speculation

---

- Speculative multithreading, dynamic multithreading, etc...
- Idea: Divide a single instruction stream (speculatively) into multiple threads at compile time or run-time
  - Execute speculative threads in multiple hardware contexts
  - Merge results into a single stream
- Hardware/software checks if any true dependencies are violated and ensures sequential semantics
- Threads can be assumed to be independent
- Value/branch prediction can be used to break dependencies between threads
- Entire code needs to be correctly executed to verify such predictions

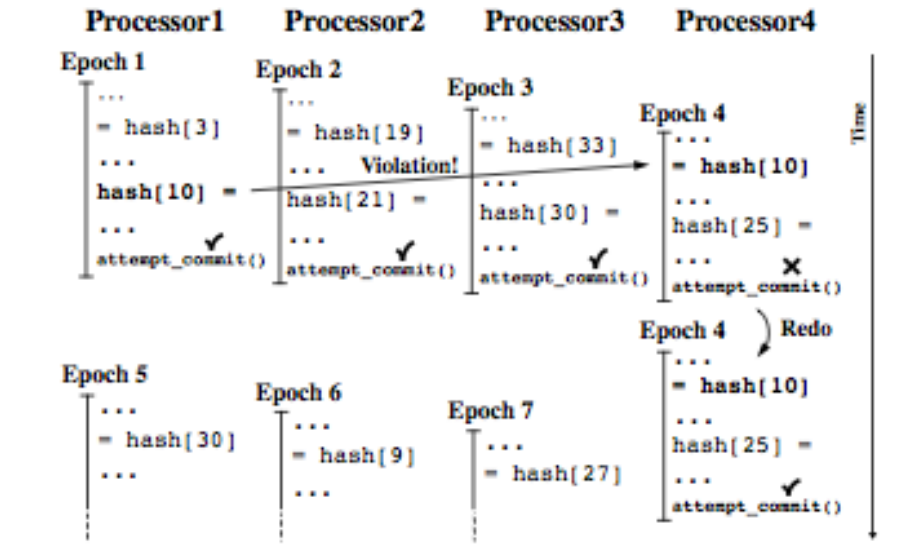
# Thread Level Speculation Example

- Colohan et al., “A Scalable Approach to Thread-Level Speculation,” ISCA 2000.

(a) Example psuedo-code

```
while(continue_condition) {  
    ...  
    x = hash[index1];  
    ...  
    hash[index2] = y;  
    ...  
}
```

(b) Execution using thread-level speculation





# TLS Conflict Detection Example

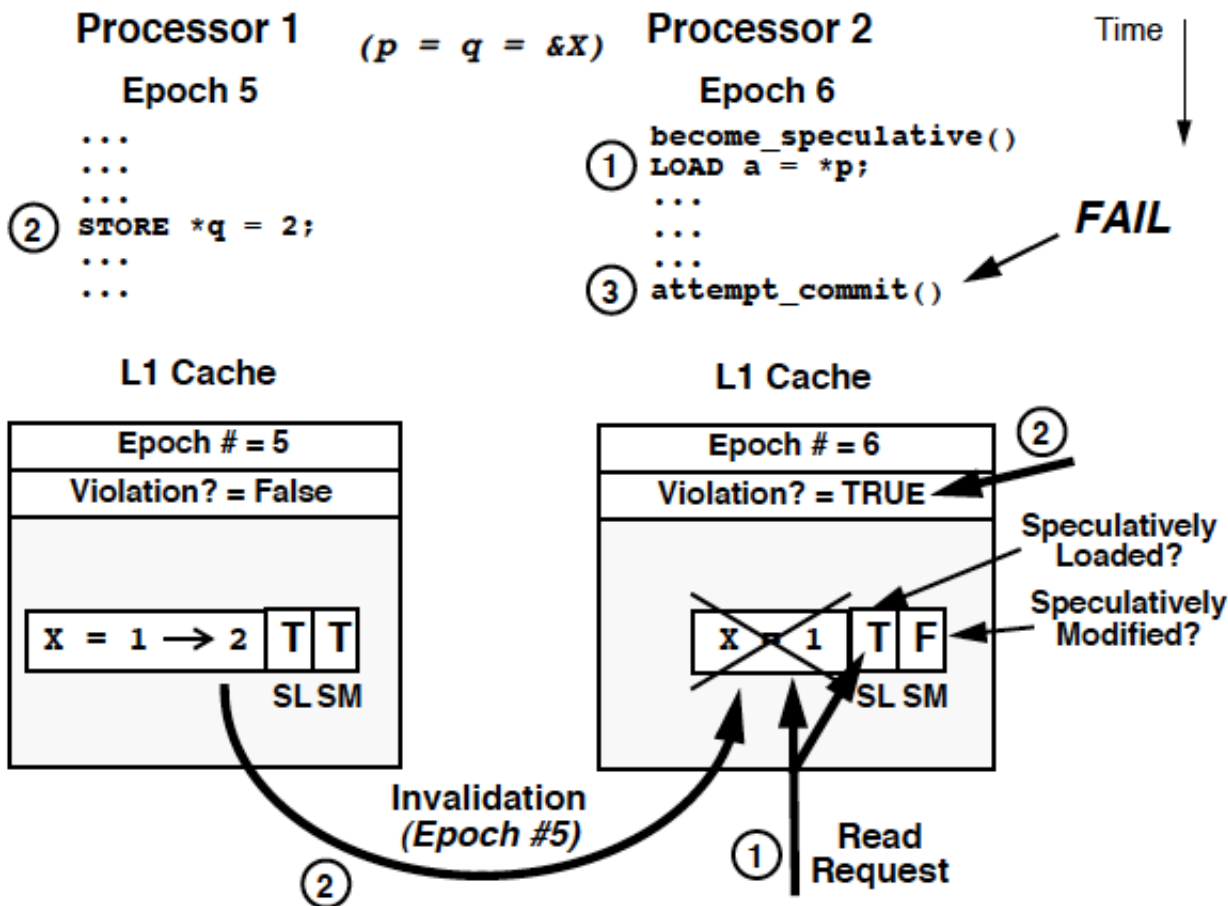


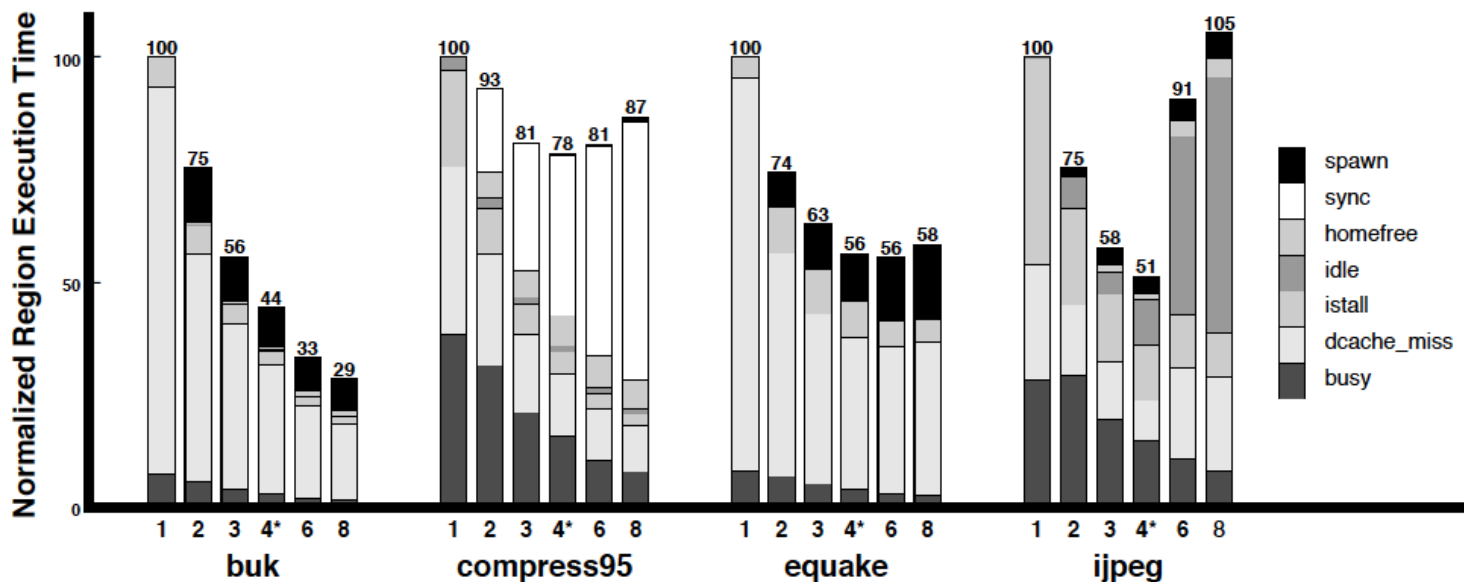
Figure 2. Using cache coherence to detect a RAW dependence violation.

# Some Sample Results [Colohan+ ISCA 2000]

**Table 3. Performance impact of TLS on our baseline architecture (a four-processor single-chip multiprocessor).**

Application	Overall Region Speedup	Parallel Coverage	Program Speedup
buk	2.26	56.6%	1.46
compress95	1.27	47.3%	1.12
equake	1.77	39.3%	1.21
jpeg	1.94	22.1%	1.08

**(a) Execution Time**



# Other MT Issues

---

- How to select threads to co-schedule on the same processor?
  - Which threads/phases go well together?
  - This issue exists in multi-core as well
- How to provide performance isolation (or predictable performance) between threads?
  - This issue exists in multi-core as well
- How to manage shared resources among threads
  - Pipeline, window, registers
  - Caches and the rest of the memory system
  - This issue exists in multi-core as well