Dual-Core Execution: Building A Highly Scalable Single-Thread Instruction Window

Huiyang Zhou

School of Computer Science
University of Central Florida

New Challenges in Billion-Transistor Processor Era

- Performance
  - How to convert increasing number of transistors into performance improvement?
  - Chip Multi-processors for improved system throughput
- Issues
  - Contemporary CMPs rely on explicit thread-level parallelism
    - Lack of parallel tasks resulting in idle processors and underutilization of the system
    - Single-thread performance is not addressed

- ***Objective: utilize multi-core processors collaboratively to improve performance of single-thread applications***

Performance Bottlenecks of Single-Thread Applications

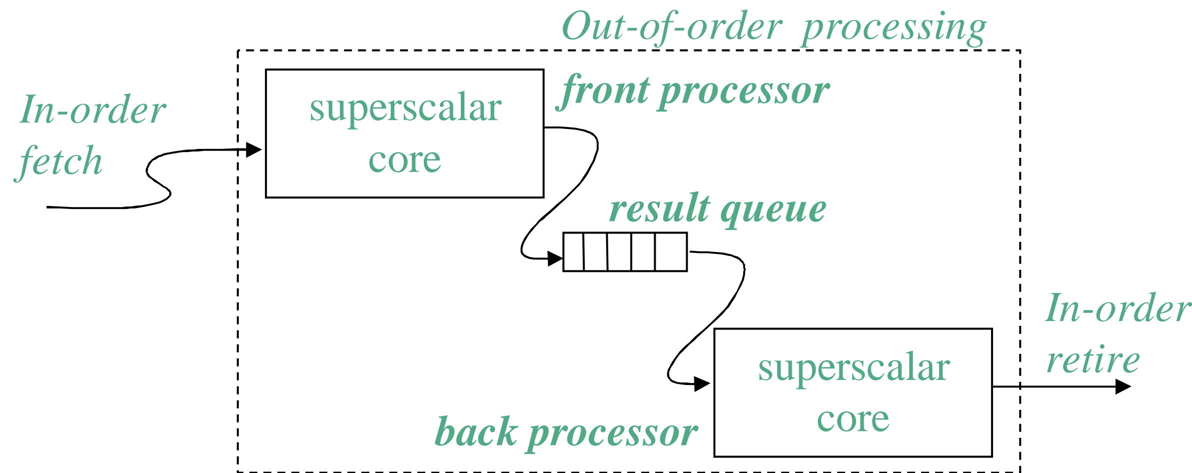
- Memory-wall problem
  - Promising solution: Out-of-order processing with very-large instruction windows [Akkary et.al.], [Cristal et.al.], [Lebeck et.al.], [Srinivasan, et.al.]
  - Often involves
    - Scalable design of critical resources including LSQ, issue queue, register file, etc.

- ***Our approach builds a very-large instruction window without the need for any large centralized structures.***

Outline

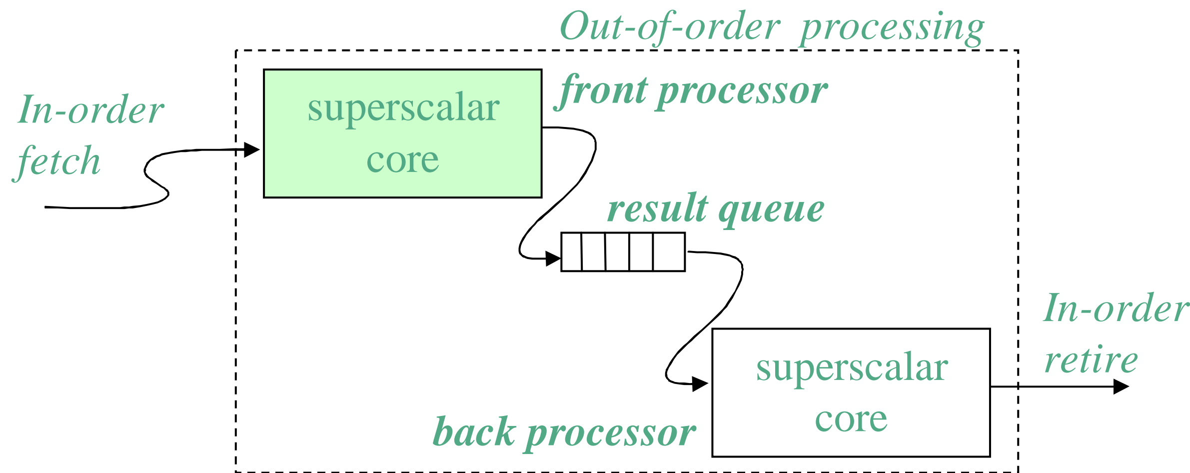
- Motivation
- Dual-core execution (DCE)
  - Overview
  - Comparison to Run-Ahead Execution
  - Architectural Design
- Experimental Results
- Related work
- DCE beyond performance
- Conclusion

Dual-Core Execution



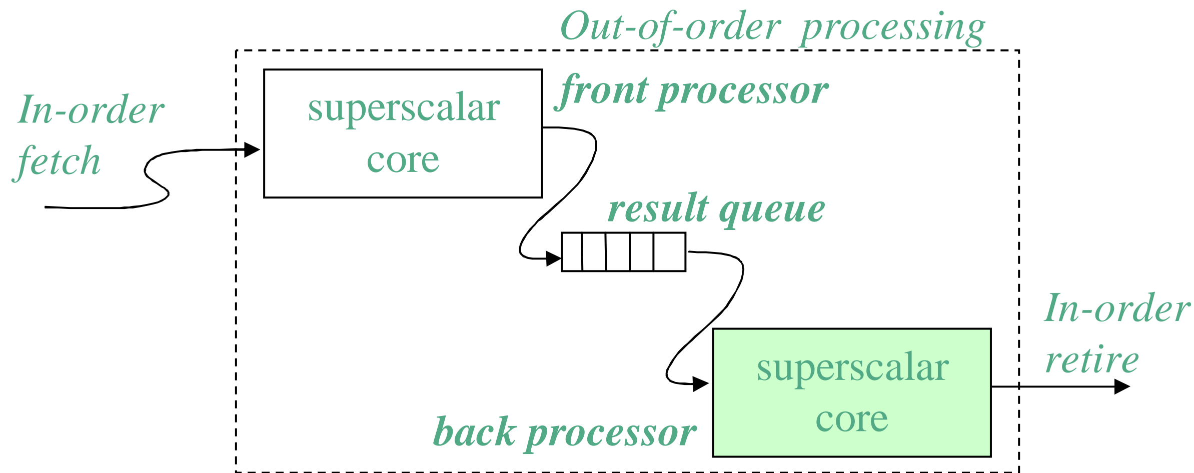
- DCE consists of two cores coupled with a queue
  - The front processor pre-processes the instructions in a fast yet very accurate way (no *correctness* requirement)
  - The retired instructions are kept in the result queue
  - The back processor re-executes the instruction stream

How Does DCE Work?



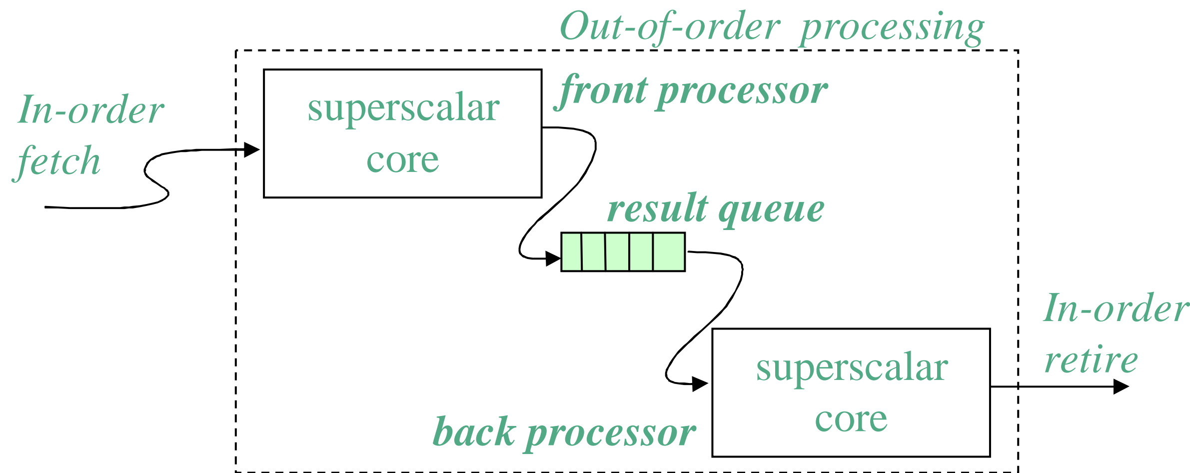
- The front processor runs faster by invalidating long-latency cache-missing loads, similar to run-ahead execution [Dundas and Mudge], [Mutlu et.al.]
  - *Speculative* execution results as load misses and their dependents are invalidated
    - Branch mispredictions dependent on cache misses are not resolved
  - *highly accurate* execution results as independent operations are not affected
    - Accurate prefetches to warm up caches
    - Correctly resolves independent branch mispredictions

How Does DCE Work? (cont.)



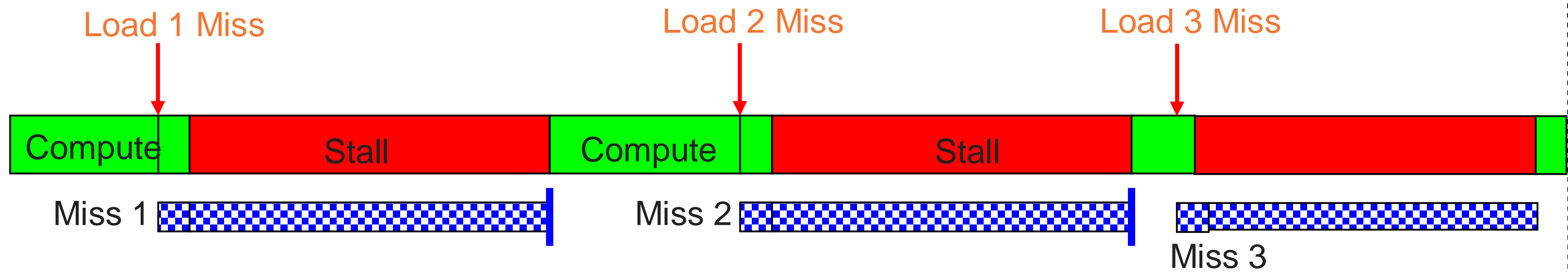
- Re-execution ensures the correctness and provides precise program state.
  - Resolve branch mispredictions dependent on long-latency cache misses
- The back processor makes faster progress with the help from the front processor.
  - Highly accurate instruction stream
  - Warmed up data caches

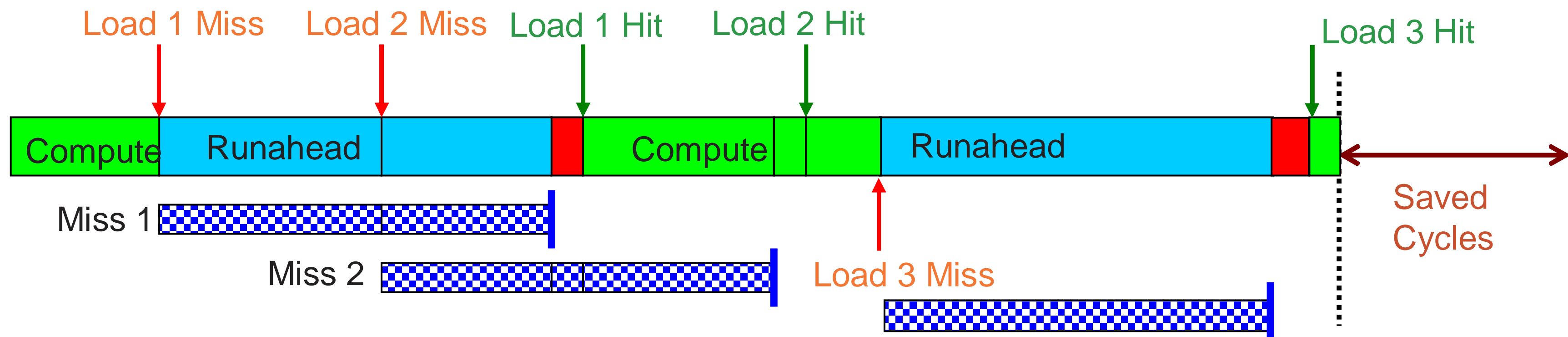
How Does DCE Work? (cont.)



- The result queue keeps a large number of in-flight instructions, which do not reserve any centralized structures
  - Highly scalable instruction window

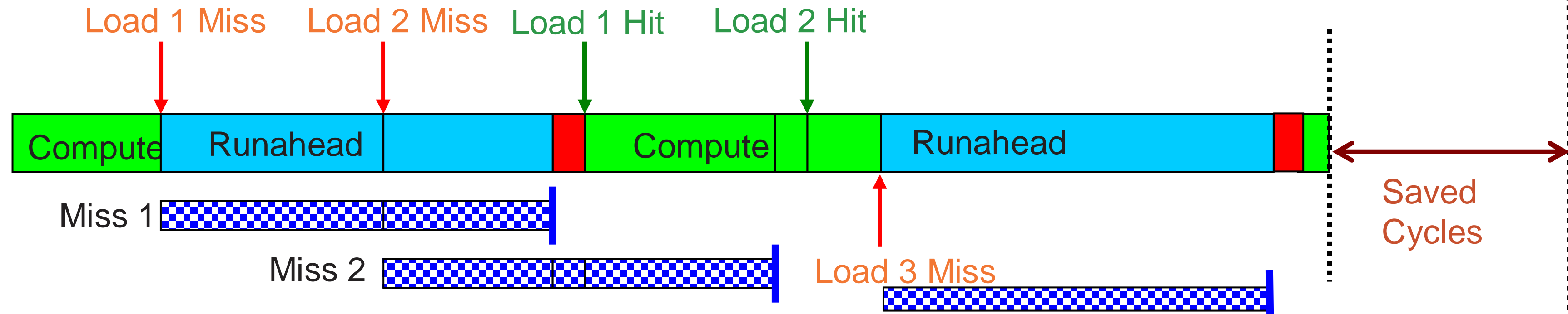
Example

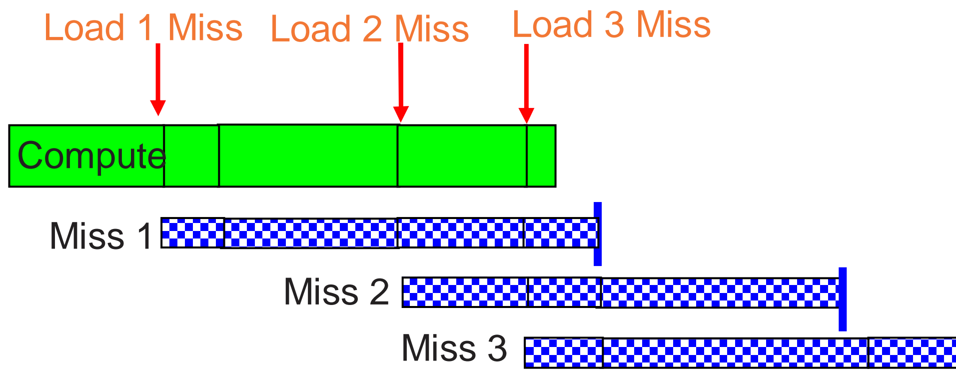


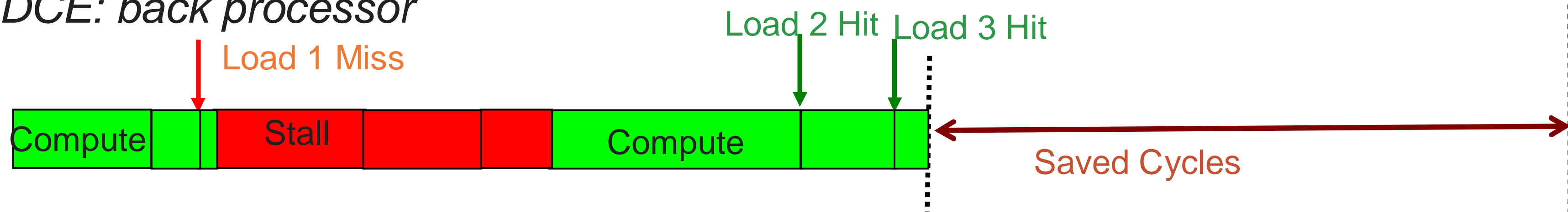


From [Mutlu et.al. ISCA05]

Run-Ahead Execution vs. DCE



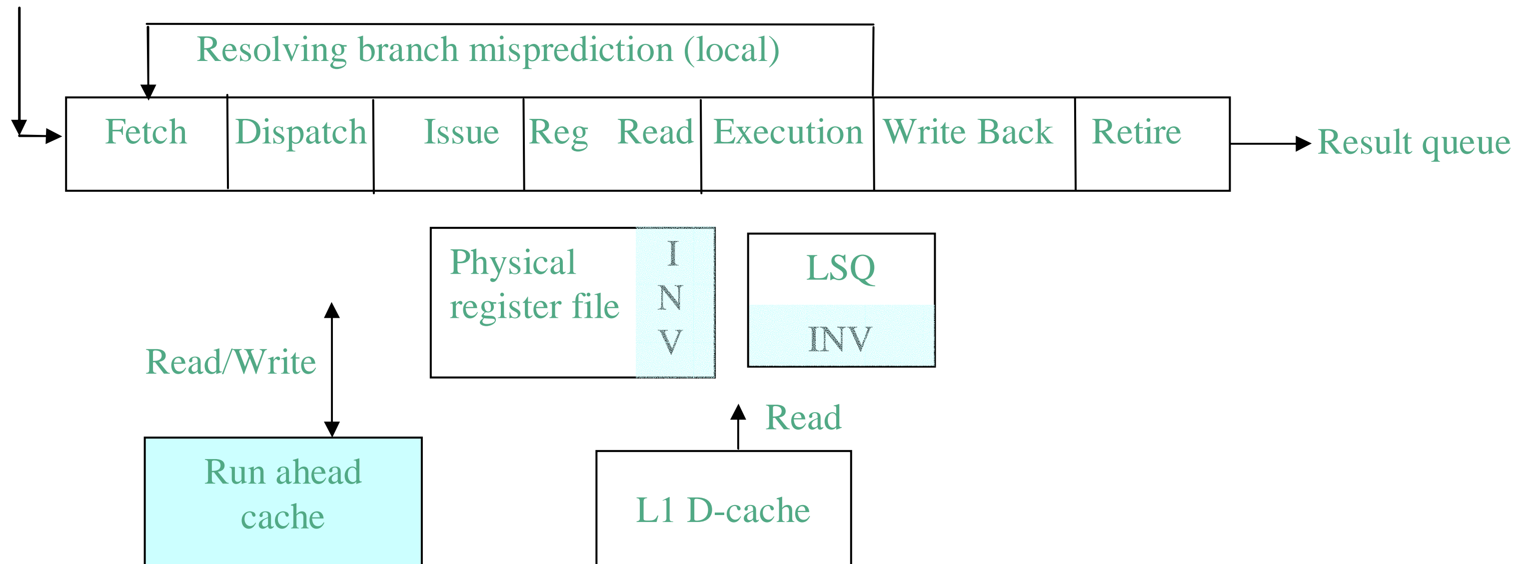




Outline

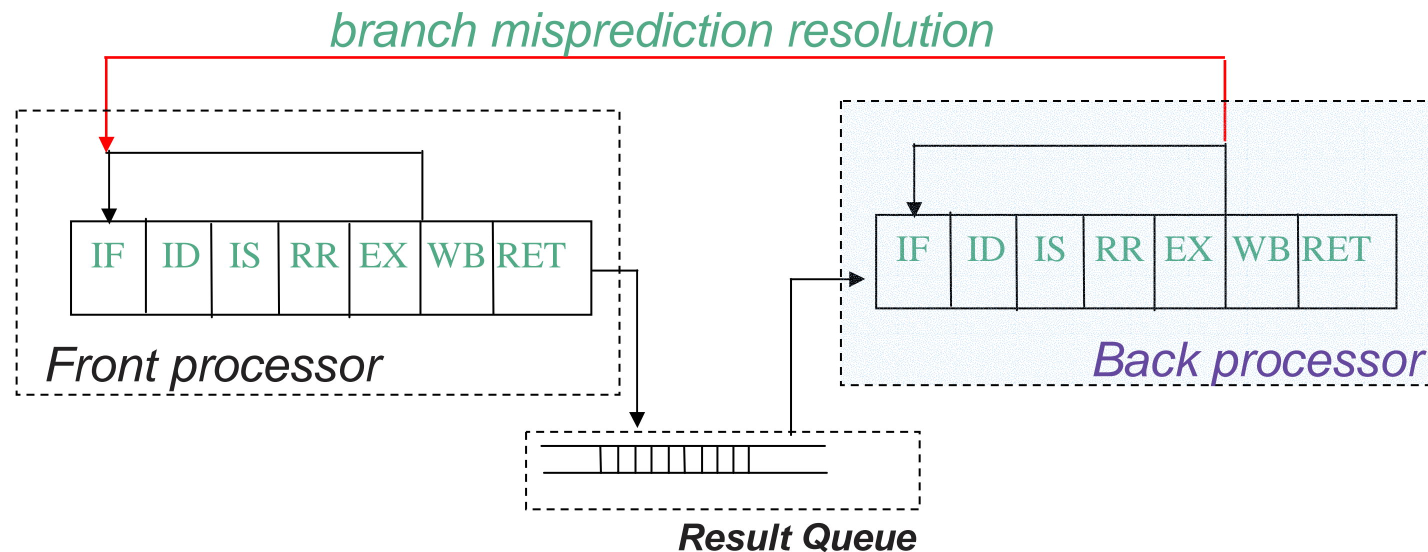
- Motivation
- Dual-core execution (DCE)
  - Overview
  - Comparison to Run Ahead Execution
  - Architectural Design
- Experimental Results
- Related work
- DCE beyond performance
- Conclusion

Architectural Design of DCE: Front Processor



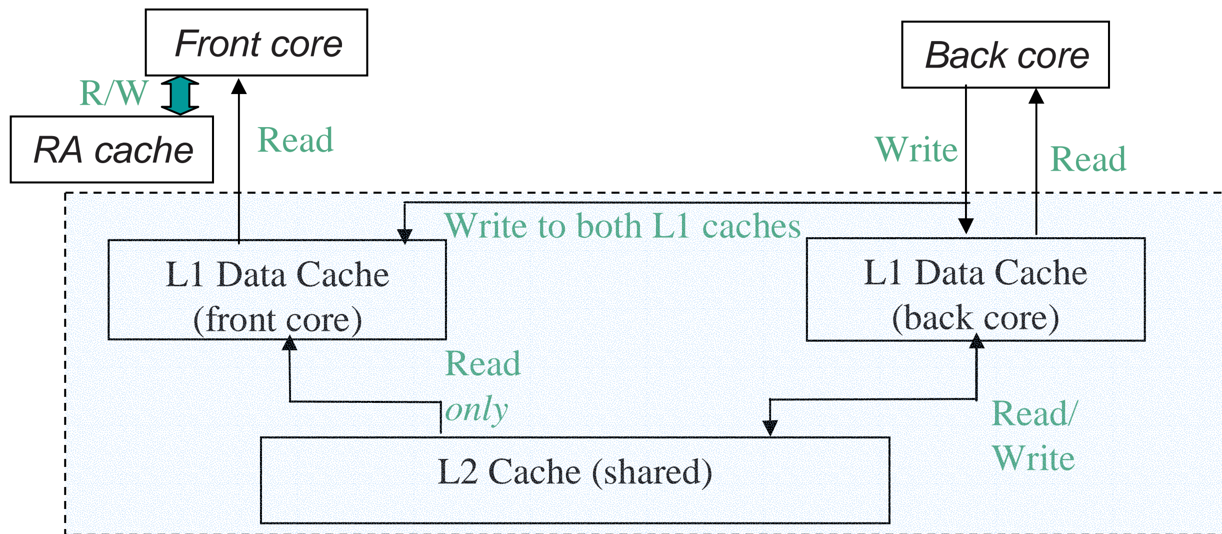
- Changes
  - Invalidation and INV propagation
  - Store instructions don't write to D-caches when being commited
  - Run-ahead cache for speculative stores
  - Similar to run-ahead execution except no checkpointing and mode transition

Architectural Design of DCE: Back Processor



- Fetches instructions from the result queue
- Branch targets computed by the front processor become the corresponding predictions
- Branch misprediction recovery:
  - Squash instructions from the front, back processors and result queue
  - Copy the architectural state (register value and PC) from the back to front processor

Architectural Design of DCE: Memory Hierarchy



- Separate L1 D-caches and shared L2 cache
- The front processor does not commit stores to its L1 D-cache
- The back processor updates both L1 D-caches

Outline

- Motivation
- Dual-core execution (DCE)
  - Overview
  - Comparison to Run Ahead Execution
  - Architectural Design
- Experimental Results
- Related work
- DCE beyond performance
- Conclusion

Methodology

- MIPS R10000 style superscalar processor
- 4-way issue, 128-entry ROB, 64-entry issue queue, 64-entry LSQ.
- 32 kB 2-way L1 caches, 1024 kB 8-way L2 cache, L2 miss latency: 220 cycles
- Branch predictor: 64k-entry G-share; 32k-entry BTB
- Stride-based stream-buffer hardware prefetcher
- 1024-entry result queue, 4 kB 4-way run-ahead cache
- Latency for copying architectural register values from back to front processor: 16 or 64 cycles

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

Latency Hiding using DCE

Latency Hiding using DCE

RA mode | unstalled execution | stall with full ROB (other) | stall with full ROB (due to misses) | empty ROB

Normalized execution time

120% 110% 100% 90% 80% 70% 60% 50% 40% 30% 20% 10% 0%

base | DCE wo rc | DCE | DCE_64 | RA

bzip2 | gap | gcc | mcf | parser | twolf | vpr | ammp | art | equake | swim | average

On average, DCE achieves 41% speedup

Non-uniform Branch Handling

Br mispred. per 1K retired insns

28
26
24
22
20
18
16
14
12
10
8
6
4
2
0

bzip2
gap
gcc
mcf
parser
twolf
vpr
ammp
art
equake
swim
Average

base
DCE front proc. (w o rc)
DCE back proc. (w o rc)
DCE front proc.
DCE back proc.

On average, 0.65 mispredictions per 1K insn. in the back processor

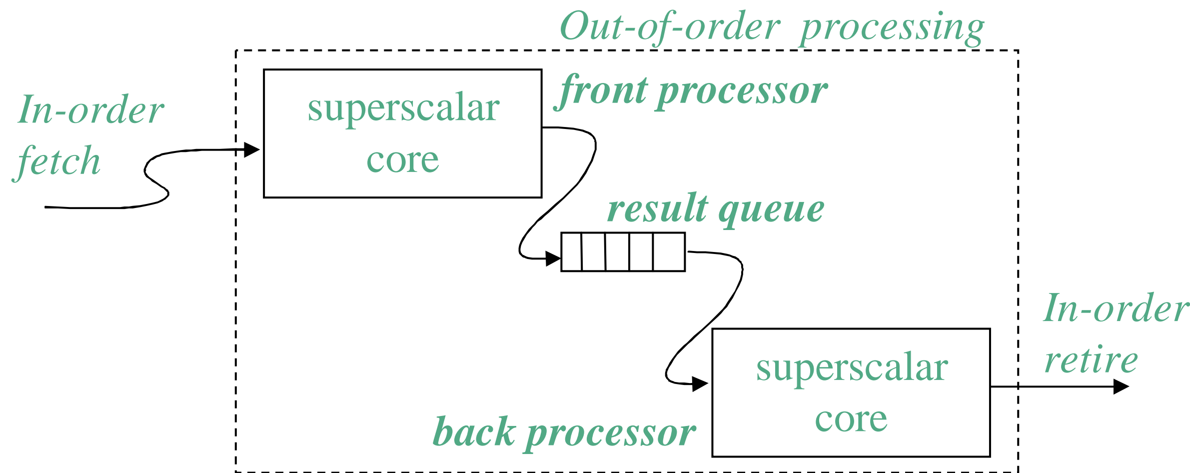
Related Work

- Run-ahead execution [Dundas and Mudge], [Mutlu et.al.]
- Large instruction window processors [Akkary et.al.], [Cristal et.al.], [Lebeck et.al.], [Srinivasan, et.al.]
  - DCE eliminates the need for scalable designs for critical centralized resources
  - DCE has somewhat lower performance potential as instructions in the result queue can not be issued.
- Slipstream processors [Sundaramoorthy et.al.] [Purser et. al.]
  - Similar high level architecture
  - Fundamentally different ways to achieve performance improvement
  - Higher performance with much less hardware overhead (see the paper)
- Flea-flicker two-pass pipelining [Barnes et. al.]
  - Re-execution in the back processor is the key to relieve the front processor of *correctness* constraints and enable it to run further ahead with much less complexity overhead.

Outline

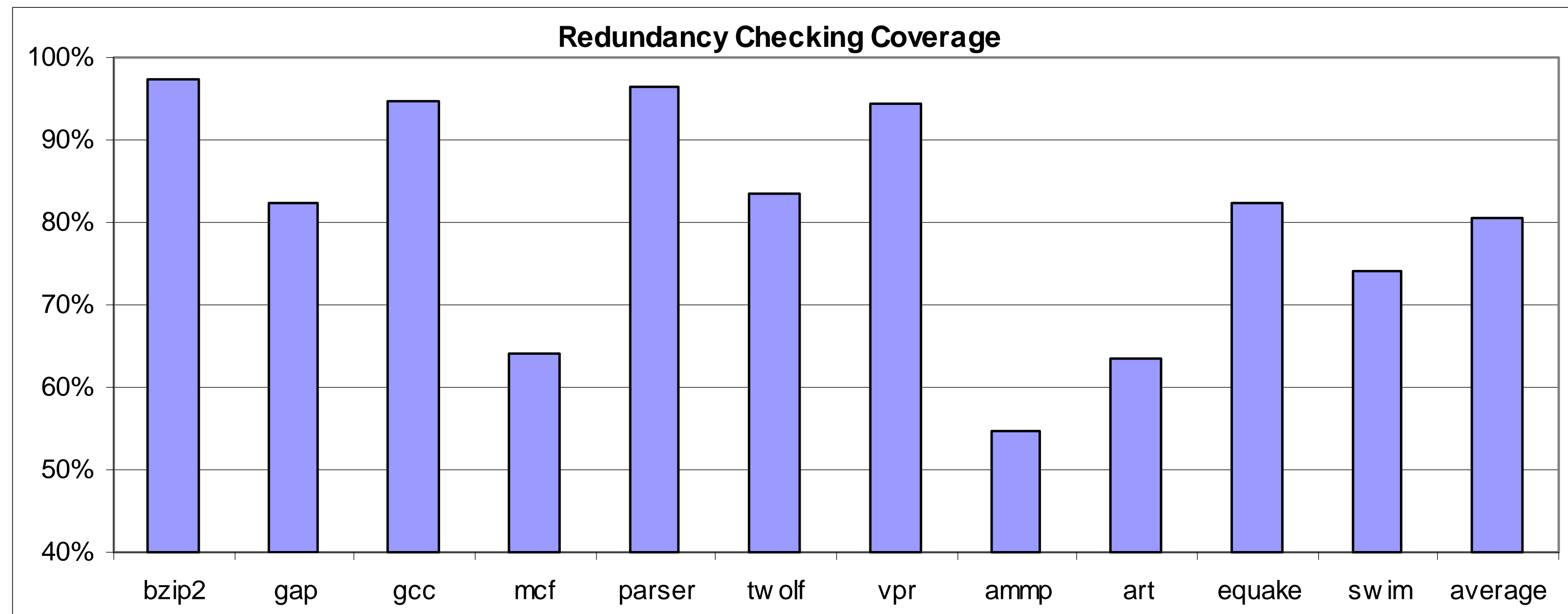
- Motivation
- Dual-core execution (DCE)
  - Overview
  - Comparison to Run Ahead Execution
  - Architectural Design
- Experimental Results
- Related work
- DCE beyond performance
- Conclusion

DCE beyond Performance



- Exploiting the redundant execution in DCE to achieve transient-fault tolerance.
  - Result queue carries the *results* (if not invalid) from the front processor
  - Compared to the back processor results to detect transient faults
  - Discrepancy => fault recovery using the existing branch misprediction recovery
- Achieves both performance enhancement and fault tolerance simultaneously [Zhou, CAL]

DCE beyond Performance



- High redundancy coverage (81%) with no observable performance loss compared to original DCE
- Slight modification to the back processor to enable full redundancy coverage and still achieves significant performance improvement (22.5%)

Conclusion & Future Work

- Dual-core execution
  - Builds upon two-way CMPs
  - Relative simple cores
  - Small hardware changes
  - Significant speedups from tolerating memory access latency
  - Efficient ways to improve transient-fault tolerance

- Future work
  - Improving energy/power efficiency
    - No need to re-execute every instruction if don't care for reliability

Thank you and Questions?