Parallelism in the Front-End

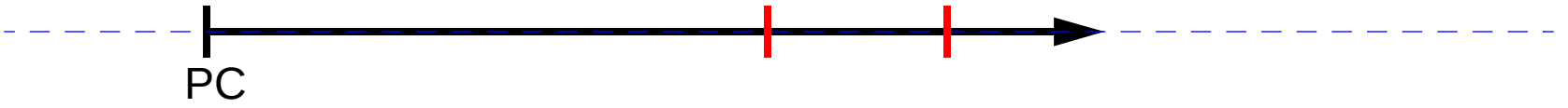
Paramjit Oberoi
Gurindar Sohi

University of Wisconsin–Madison

International Symposium on Computer Architecture

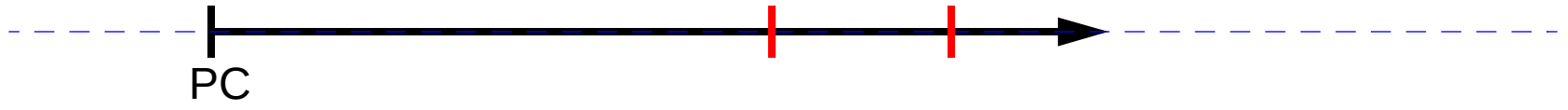
June 10, 2003

Sequential Front-Ends



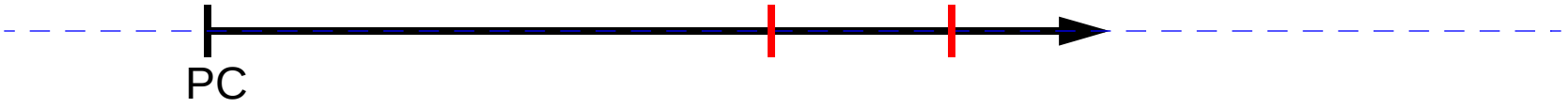
- **Sequential Fetch**
  - Fetch contiguous block of instructions starting at PC
  - Hard to fetch across discontinuities in the same cycle
    - Taken branches
    - Cache-line boundaries
- **Discontinuities are frequent**
- **Must fetch across discontinuities for high performance**

Fetching Across Discontinuities

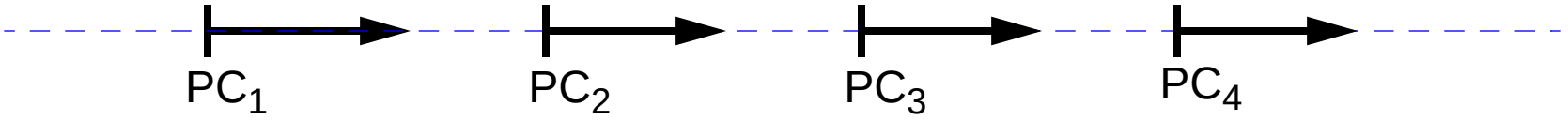


- **Code layout optimizations**
  - Effective, but limited
  - Programs have complex control flow
- **Trace Cache**
  - Significant space overhead
  - Poor performance on large instruction working sets
- **Collapsing Buffer**
  - Complex Hardware
  - Trace Cache performs better

Wide Fetch v/s Parallel Fetch

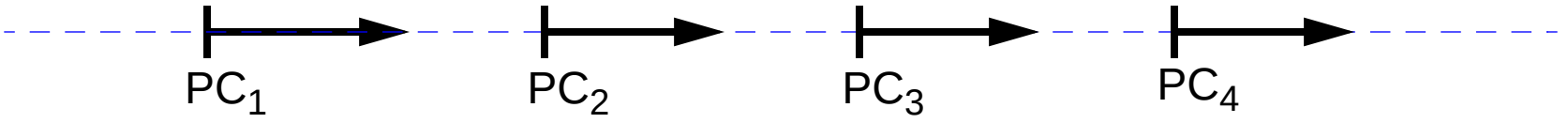


- **Wide Fetch**
  - Fetch long contiguous blocks of instructions
  - Hard to construct long contiguous blocks



- **Parallel Fetch**
  - Fetch multiple discontiguous blocks of instructions
  - Individual blocks can be small
  - More blocks => higher throughput

Parallel Fetch



- **Multiple PCs**
  - Each PC is the start of a fragment of the instruction stream
- **Multiple Sequencers**
  - Sequence through instructions in program order
  - Similar to a sequential fetch unit
  - Each sequencer is assigned a fragment
- **Multiple fragments are fetched in parallel**

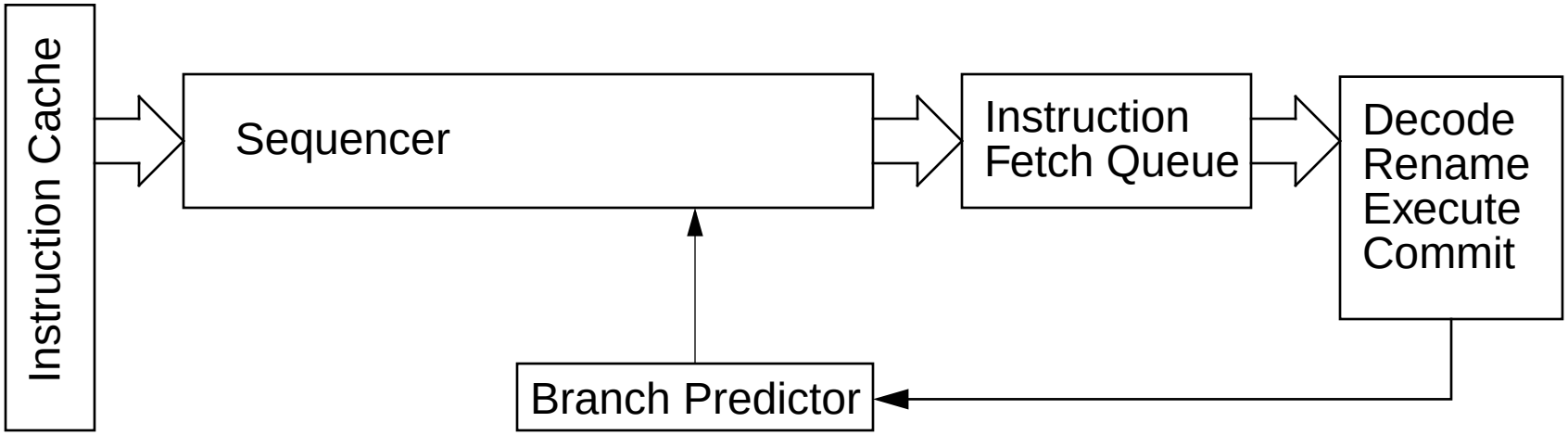
Benefits of Parallel Fetch

- **Throughput not limited by individual sequencers**
  - Add hardware to increase throughput
- **Latency Tolerance**
  - One stall does not block fetch completely
  - Stalls are overlapped with fetch of other instructions
- **Flexibility**
  - Fine-grain allocation of fetch bandwidth to threads
  - May ease implementation of other techniques
    - Dual-path execution, Speculative threads
- **Replicated Hardware**
  - Easier design and verification

Outline

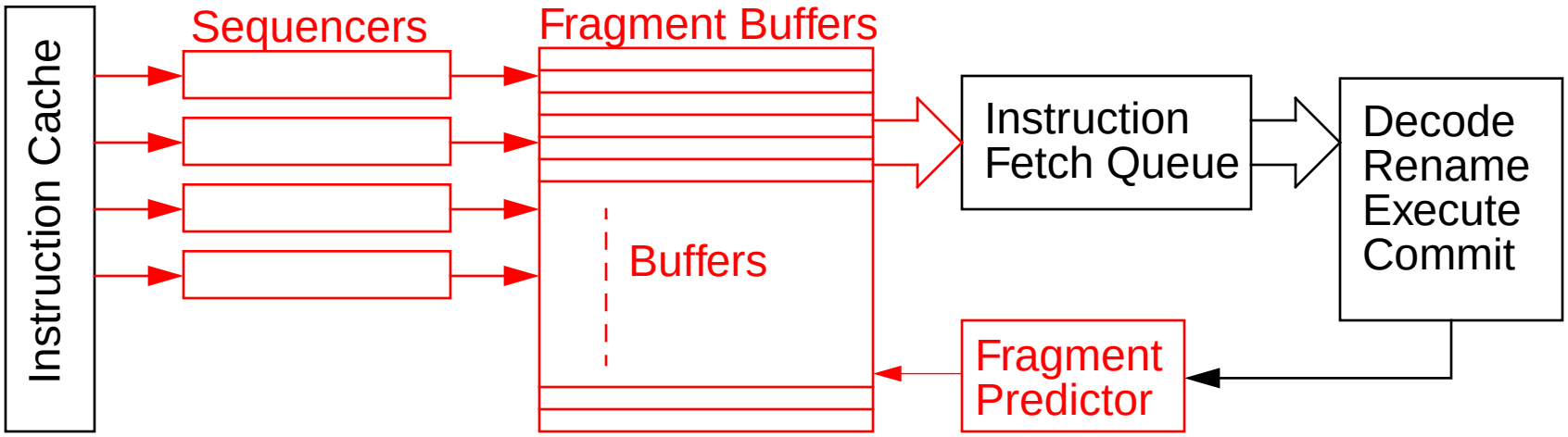
- Introduction
- **Parallel Fetch**
  - **Design Overview**
  - **Fragment Selection & Prediction**
- Parallel Renaming
- Simulation Results
- Related Work
- Conclusions

Sequential Fetch Unit



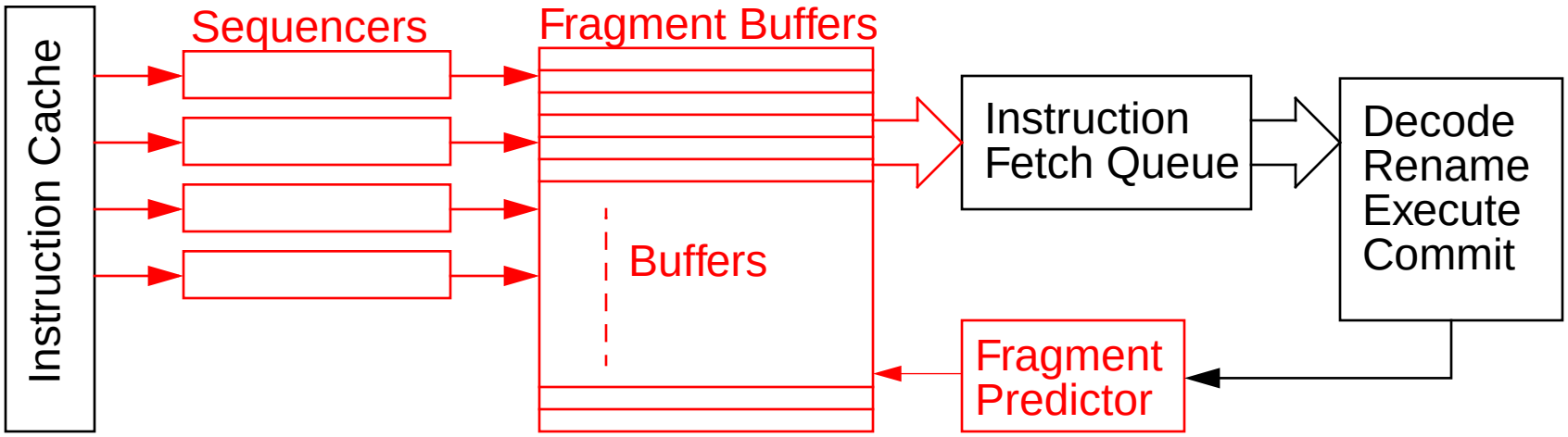
- Single sequencer
- Instruction fetch queue buffers instructions
- Branch predictor predicts individual branches

Parallel Fetch Unit



- Fragment predictor predicts future fragments
  - Each fragment is assigned a fragment buffer
  - Sequencers fetch multiple fragments in parallel
- Instructions from oldest fragment are placed in the IFQ
  - Instructions enter the IFQ in program order
  - No changes to rest of the pipeline

Fragments

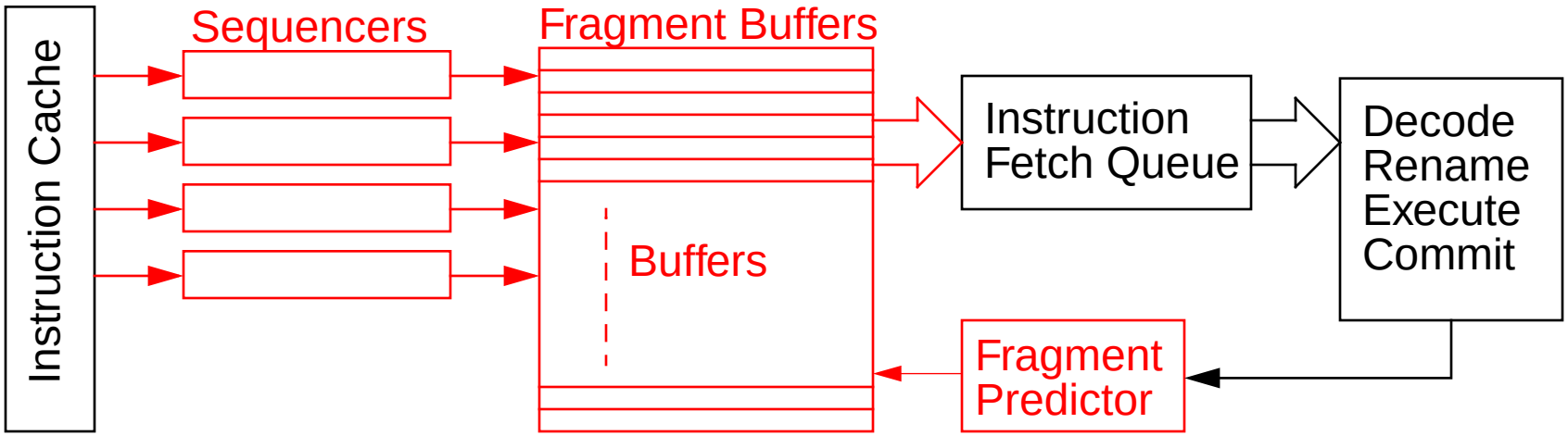


- **Fragments**
  - Defn: A contiguous portion of the dynamic instruction stream
  - The entire stream is obtained by concatenating all fragments
  - The division can be completely arbitrary
    - Unlike traces, tasks, streams, ...
- Identical to **Traces** in this paper
  - Simplifies evaluation

Fragment Selection and Prediction

- **Selection**
  - Similar to previous techniques (*tasks*, *traces*)
  - More details in paper
- **Prediction**
  - Path-based prediction [Jacobson et al., MICRO 1997]
  - 95% prediction accuracy
- **Characteristics**
  - Maximum size: 16 instructions
  - Average size:10-12 instructions
  - Typical working set sizes are small (< 500 fragments)

Performance Intuition

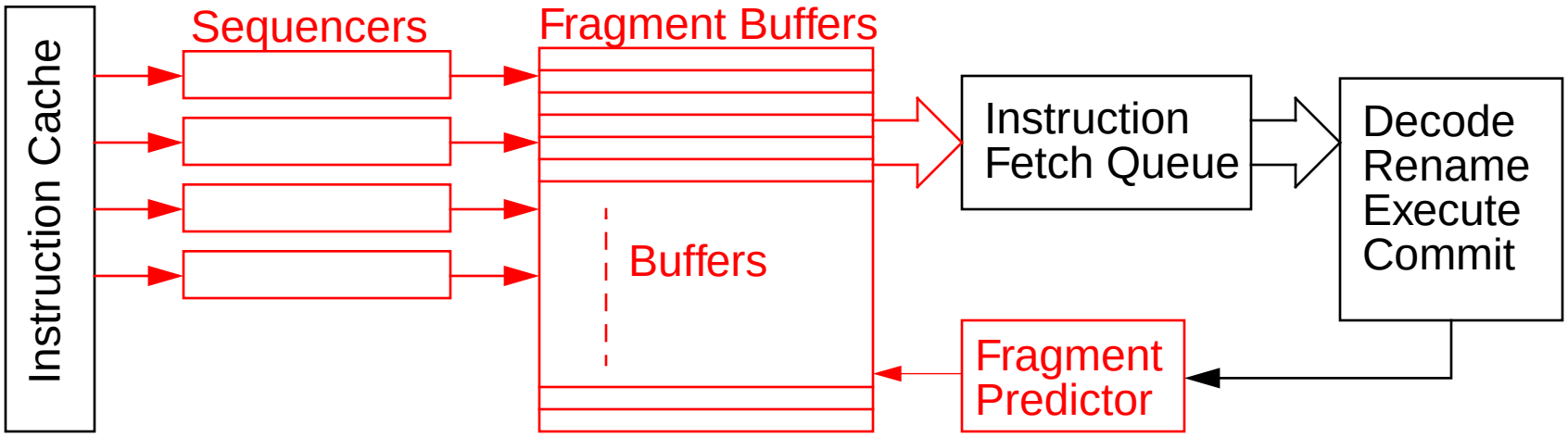


- Sequencers fetch fragments far ahead of rest of the pipeline
- Steady State: Just-in-time fragment construction
  - 84% fragments are constructed in advance
  - Compare: 87% traces hit in the trace cache
- Performance of Trace Cache
  - Storage efficiency of Instruction Cache

Outline

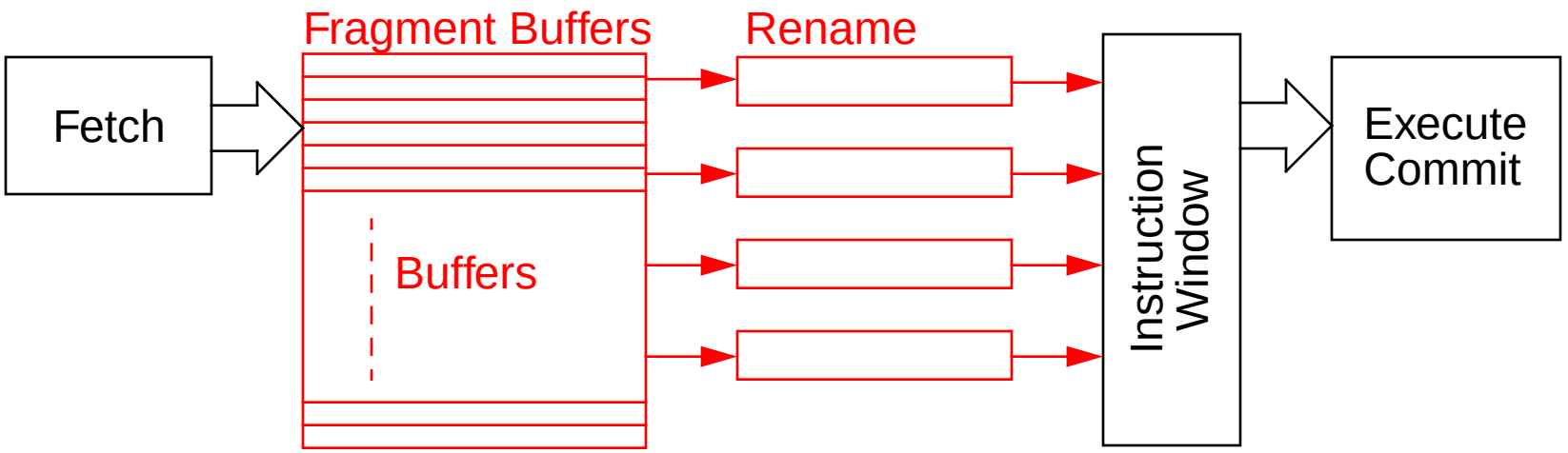
- Introduction
- Parallel Fetch
- **Parallel Renaming**
  - **Motivation**
  - **Design Overview**
- Simulation Results
- Related Work
- Conclusions

The IFQ Bottleneck



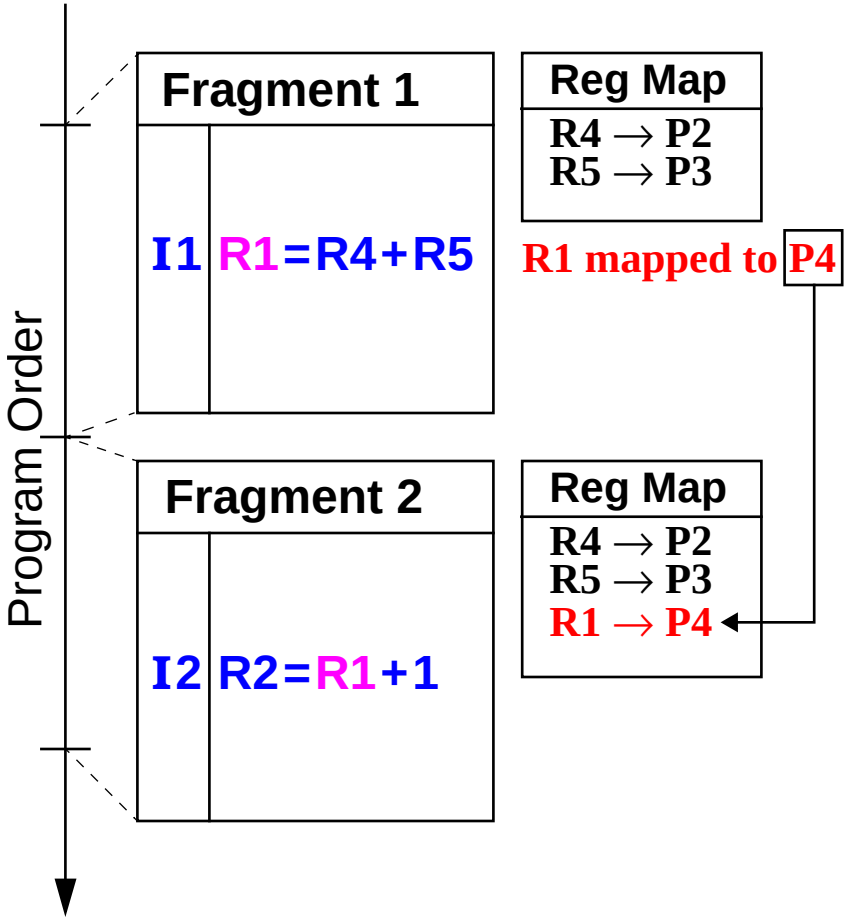
- **Instructions are renamed in-order**
  - Delay in constructing a fragment affects all future fragments
  - Example: Branch misprediction, Cache miss
- **Rename rate is low**, even though fetch rate is high
  - Serialization exposes latencies

Parallel Renaming



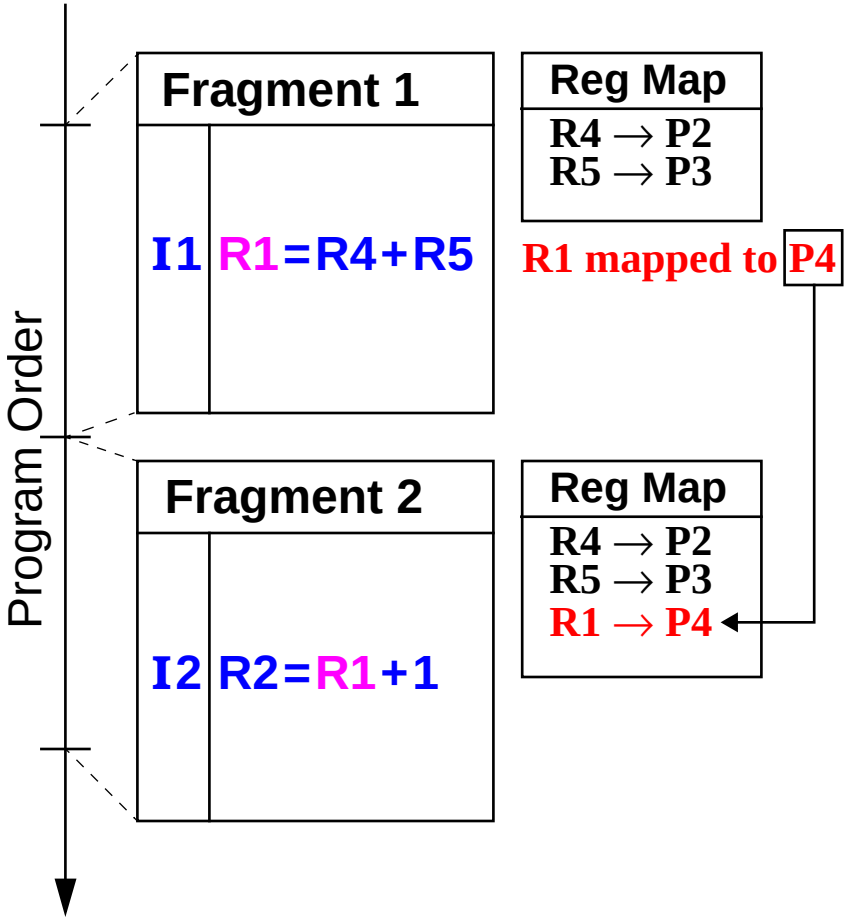
- **Rename multiple fragments simultaneously**
  - Replicate rename units
  - All renamers operate in parallel
- **Must rename instructions out-of-order**
  - How?

Out-of-Order Renaming—Problem



- I1 writes to R1
  - **Create** mapping R1 → P4
- I2 reads R1
  - **Use** mapping R1 → P4
- If I2 is renamed before I1?

Out-of-Order Renaming—Solution



- **Speculate**
  - Predict Frag 1 writes to R1
  - **Create** mapping R1 → P4
- **Rename**
  - **Use** predicted mapping when renaming I2
  - Use pre-allocated register when renaming I1

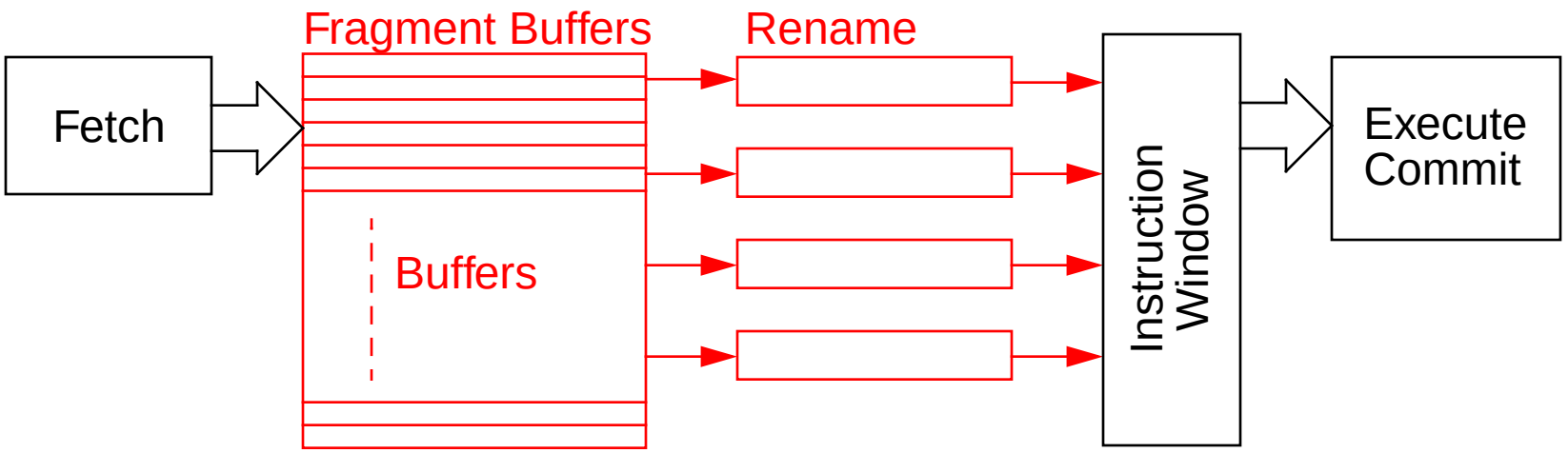
Out-of-Order Renaming—Details

- **Three predictions for each fragment**
  - Length of the fragment
  - Registers written by the fragment (live-outs)
  - Instructions corresponding to live-out values
- **Stage 1:** Serialized, but fast
  - Allocate ROB slots
  - Allocate physical registers to all live-outs
  - Forward new mappings to future fragments
- **Stage 2:** Multiple fragments in parallel
  - Rename instructions as usual
  - Use pre-allocated physical registers for live-outs
  - Insert into ROB
- Similar to Skipper [Cher et al., MICRO 2001]

Live-out Prediction

- **Live-outs are predictable**
  - Fragments are fixed sequences of instructions
    (in the absence of self-modifying code)
  - Record live-outs when a fragment is seen
  - Use recorded values as predictions
- **Predictor**
  - 4K entries, 2-way set associative
  - 98% accuracy
- **Size: 42KB**
  - 84 bits per entry
    Registers: 64, Instructions: 16, Tag: 4
  - Size can be reduced by using more complex encoding
    Most fragments have only 4–6 live-outs

Mispredictions



- **Detecting Mispredictions**
  - After a fragment is renamed, check predicted live-outs
  - Check fragment length
    - Overprediction OK
- **Recovery**
  - Squash all future fragments
  - Selective re-execution can also be used

Outline

- Introduction
- Parallel Fetch
- Parallel Renaming
- **Simulation Results**
  - **Front-End Throughput**
  - **Performance**
  - **Sensitivity to Cache Size**
- Related Work
- Conclusions

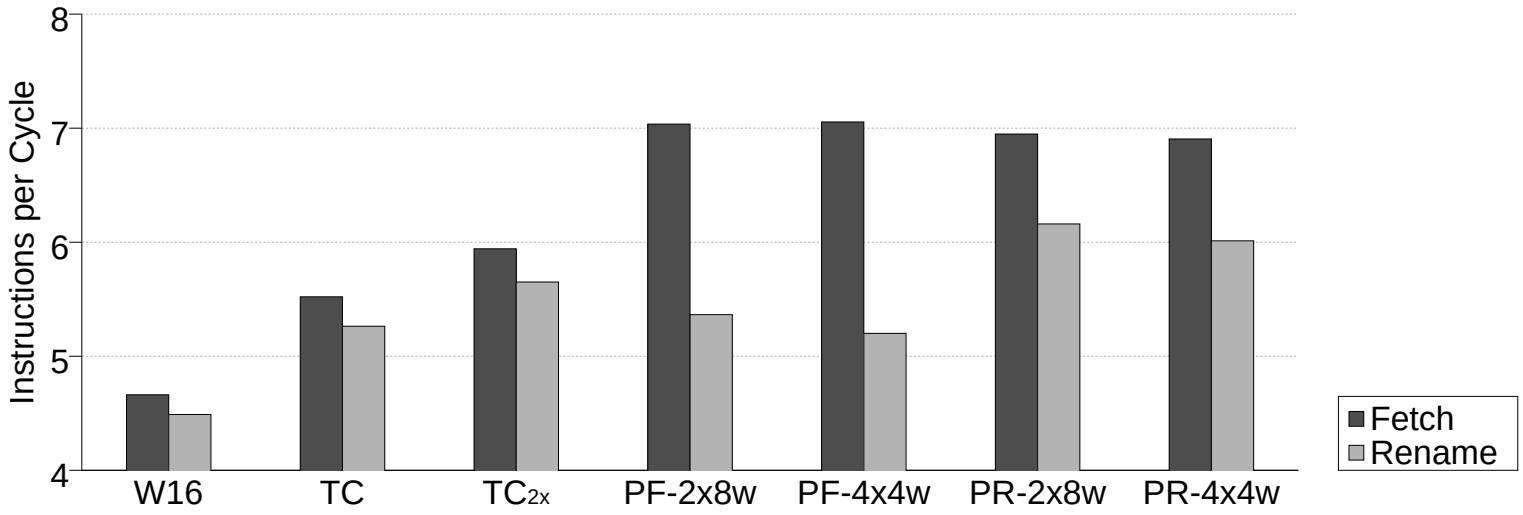
Simulated Configurations

- 16-wide processor, 256 entry window, 64KB L1, 1MB L2
  - Fetch unit is a bottleneck
- **W16** — Sequential, 16-wide
  - 64KB instruction cache
  - stop at taken branches and cache-line boundaries
- **TC** — Trace Cache
  - 2-way set associative
  - Space split equally between TC and I-Cache
    - Performs better than allocating all space to TC
    - 32KB TC + 32KB I-cache
- **TC_{2x}** — Trace Cache with 2x cache size
  - 64KB TC + 64KB I-cache

Simulated Configurations (contd.)

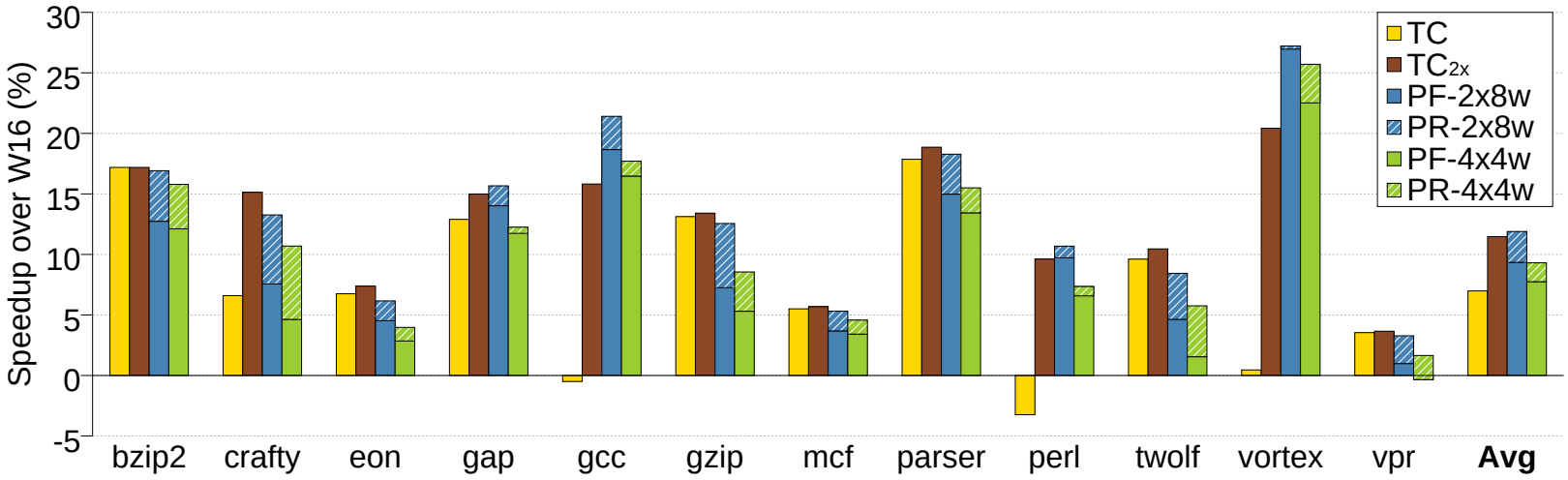
- **Parallel Front-End**
  - 16 fragment buffers (1KB)
  - Sequencers are identical to W16 (except their width)
- **PF** — Parallel Fetch
  - PF-2x8w: 2 sequencers, 8-wide each
  - PF-4x4w: 4 sequencers, 4-wide each
- **PR** — Parallel Rename
  - PR-2x8w: 2 sequencers, 2 renamers, 8-wide each
  - PR-4x4w: 4 sequencers, 4 renamers, 4-wide each
- **Branch Predictor**
  - All mechanisms—W16, TC, PF/PR—use a trace predictor
  - TC & PF/PR have identical trace/fragment selection

Fetch and Rename Rate



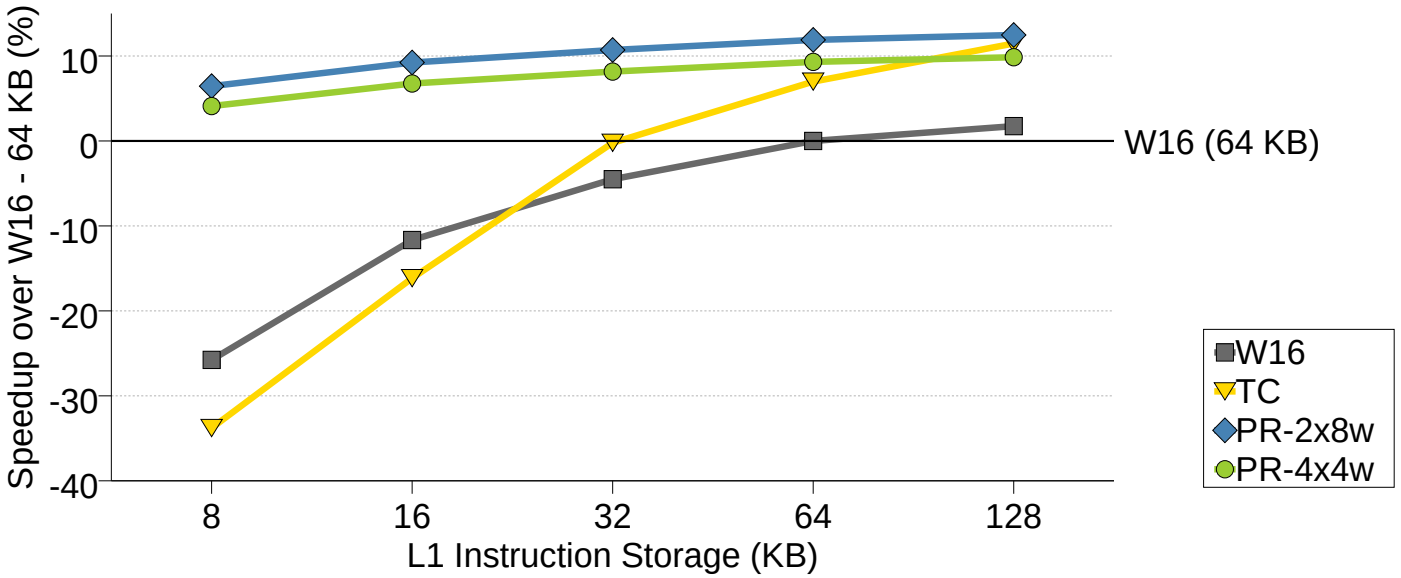
- **High Fetch Rate**
  - 20% more than TC, 49% more that W16
- **PF has low rename rate**
  - Despite a high fetch rate
- **PR increases rename rate by 13%**
  - Still, large gap between fetch and rename rate

Performance



- **9–12% better than W16 on average**
- **PR-2x8w 5% better than TC on average**
  - Equivalent to TC_{2x} with only half the space
  - 10–20% better than TC on large programs (`crafty`, `gcc`, `perl`, `vortex`)
- **PR increases performance by 0–6%**

Sensitivity to Cache Size



- Increase cache misses by reducing cache size
  - W16 and TC - performance deteriorates rapidly
  - PF & PR performs robustly across a range of sizes
- 40–60% faster on average for small cache sizes
  - Parallelism hides L1 miss latency

Outline

- Introduction
- Parallel Fetch
- Parallel Renaming
- Simulation Results
  - Front-End Throughput
  - Performance
  - Sensitivity to Cache Size
- **Related Work**
- **Conclusions**

Related Work

- **Wide Fetch**
  - Code Relayout [many schemes]
  - Trace Cache [Peleg, many others]
  - Collapsing Buffer [Conte, et al.]
- **Trace Selection and Prediction**
  - Multiscalar, Other Speculative MT architectures
  - Trace Cache [Rotenberg et al., Patel et al.]
  - Path-based Prediction [Jacobson el al.]
- **Renaming**
  - Out-of-Order Renaming [Stark et al.]
  - Skipper [Cher et al.]
- **Alpha 21464**
  - Fine grained allocation of fetch resources to threads

Conclusions

- **High bandwidth fetch**
  - Wide fetch is difficult due to discontinuities
  - Parallel fetch is an alternative
- **Parallel Fetch and Rename**
  - Higher throughput than a Trace Cache
  - Robust performance across a wide range of cache sizes
- **Better fit for the future**
  - High performance
  - Replicated hardware
  - Larger programs, Smaller low-latency caches

Simulation Parameters

Width	Fetch, decode and commit at most 16 instructions per cycle
Functional Units	16 integer ALUs, 4 integer multipliers, 4 floating point ALUs, 1 floating point multiplier, 4 load/store units
In-flight Instructions	256 entry instruction window 128 entry load/store queue
L1 Caches (Insn & Data)	64K, 2-way set-associative, 1 cycle access time, 64b blocks
L2 Cache (Unified)	1M, 4-way set-associative, 10 cycle access time, 128 byte blocks
Memory	100 cycle access time
Trace Predictor	64K entry primary table 16K entry secondary table D=9, O=4, L=7, C=9