SFU

# CMPT 450/750: Computer Architecture
# Fall 2022

## Pipeline Complexity & Memory Ordering

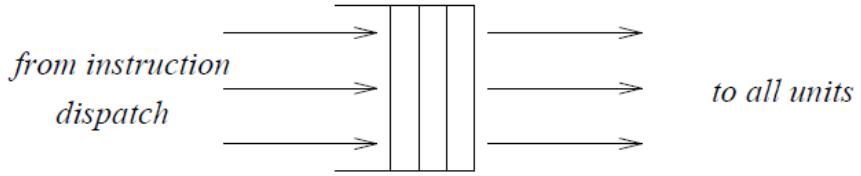
Alaa Alameldeen & Arrvindh Shriraman

# Dynamic Instruction Scheduling (Review)

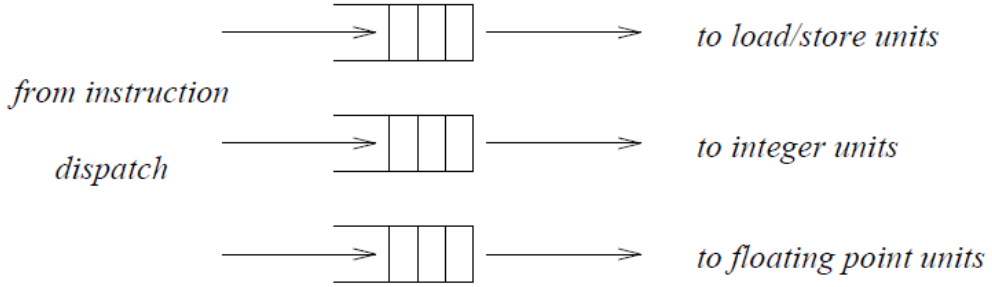
- **After instructions are fetched, decoded and renamed, they are placed in instruction buffers where they wait until issue**
- **An instruction can be issued when its input operands are ready, and there is a functional unit available**
- **All out-of-order issue methods must handle the same basic steps**
  - Identify all instructions that are ready to issue
  - Select among ready instructions to issue as many as possible
  - Issue the selected instructions, e.g., pass operands and other information to the functional units
  - Reclaim instruction window storage used by the now issued instructions

# Methods of Organizing Instruction Issue Buffers

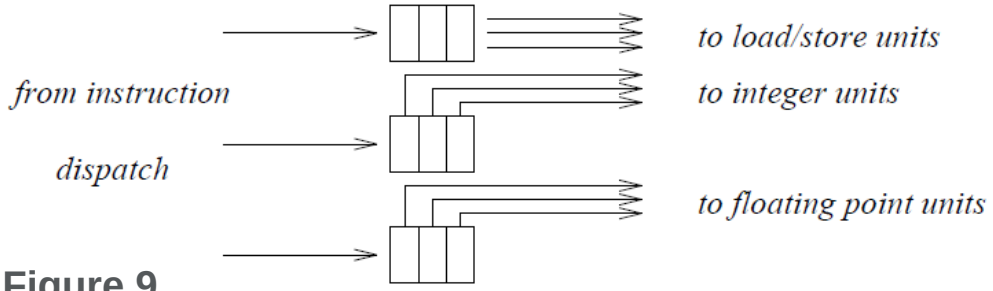
1. **Single shared queue**
   - ➢ Only for in-order issue
2. **Multiple queues, one per instruction type**
3. **Multiple reservation stations, one per instruction type**
4. **Single central reservation stations buffer**
   - ➢ Combines functions of RS and ROB. Allocation and removal in FIFO order.




a) Single, shared queue




b) Multiple queues; one per instruction type

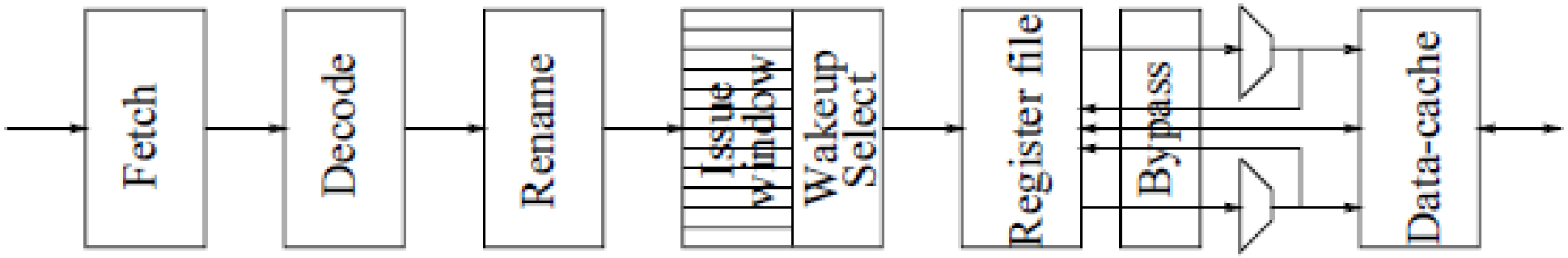



c) Multiple reservation stations; one per instruction type

**Smith & Sohi 1995, Figure 9**

# Tradeoffs in Pipeline Design

- **Tradeoff between complexity and speed**
  - ➢Pipeline consists of many stages with different functions




| FETCH | DECODE | RENAME | WAKEUP SELECT | REG READ | EXECUTE BYPASS | DCACHE ACCESS | REG WRITE COMMIT |
|---|---|---|---|---|---|---|---|

Palacharla et al., 1997, Figure 1

# Tradeoffs in Pipeline Design

- **Clock speed (i.e., Frequency) is determined by slowest, most complex stage**
  - Unless stage is split or work is redistributed to other stages
  - A good design is balanced: No single stage is the only bottleneck
- **To achieve high IPC, we need larger instruction window which increases complexity**
  - Remember execution time equation:

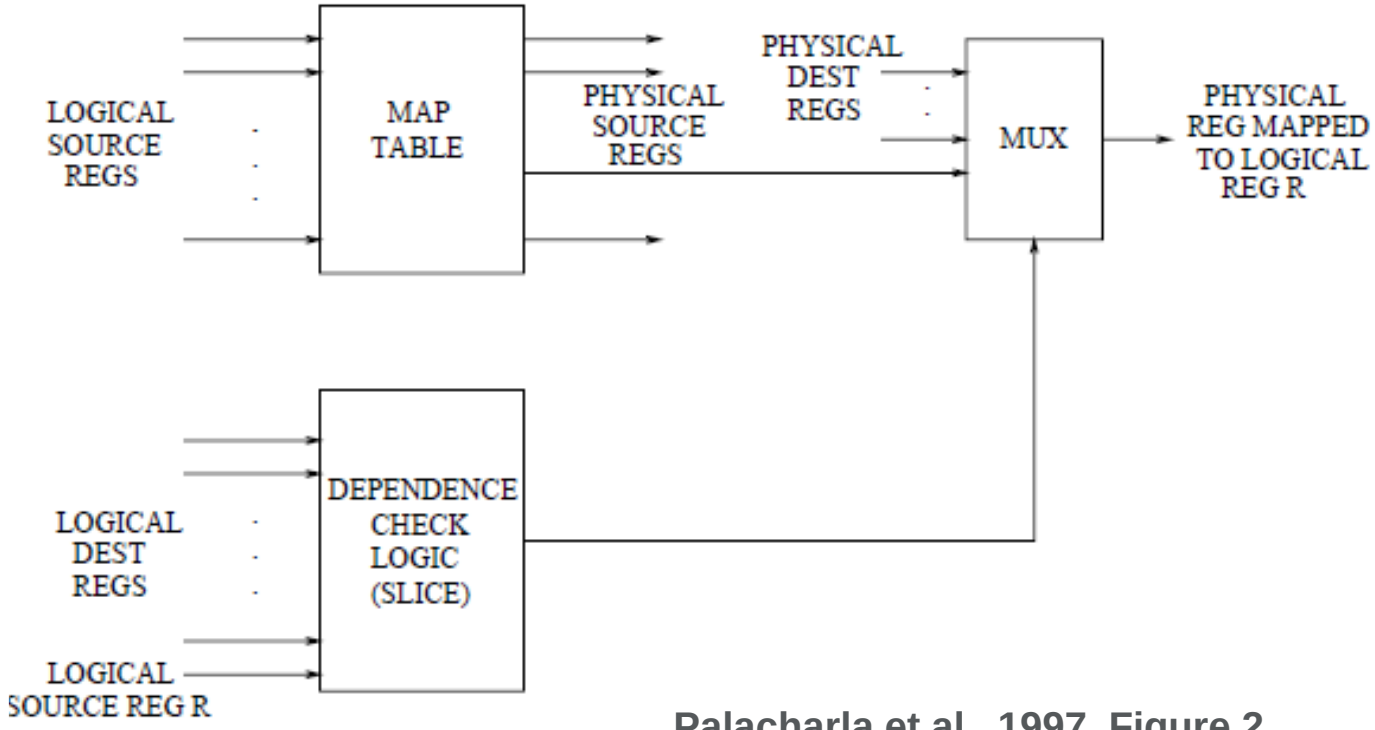
    Time = Instructions/Program x Cycles/Inst x Time/Cycle = Inst x CPI / Frequency
  - To reduce execution time, we need to decrease CPI and increase frequency
  - Decreasing CPI (i.e., increasing IPC) requires more complex wide superscalar processors that can fetch, decode, issue, execute and retire multiple instructions every cycle
  - Complexity increases work per pipe stage, which decreases clock speed

# Which Structures Have High Complexity?

- **Some structures that get more complex with larger issue width or larger instruction window**
  - Register Rename Logic
    - Translates logical registers to physical registers
  - Wakeup Logic
    - Wakes up instructions waiting for their source operands
  - Selection Logic
    - Selects instructions for execution from pool of ready instructions
  - Bypass Logic
    - Bypass operand values from instructions that finished execution but not write back to subsequent instructions

# Register Rename Logic

- **Translates logical to physical registers by accessing a map table indexed by logical register**
- **Map table is multi-ported**
  - Multiple instructions (each with one or more register operands) need renaming every cycle
- **Dependence check logic needed**
  - Detects whether logical register being renamed is written by an earlier instruction in the current group being renamed
  - Sets up output MUXes to select appropriate physical registers



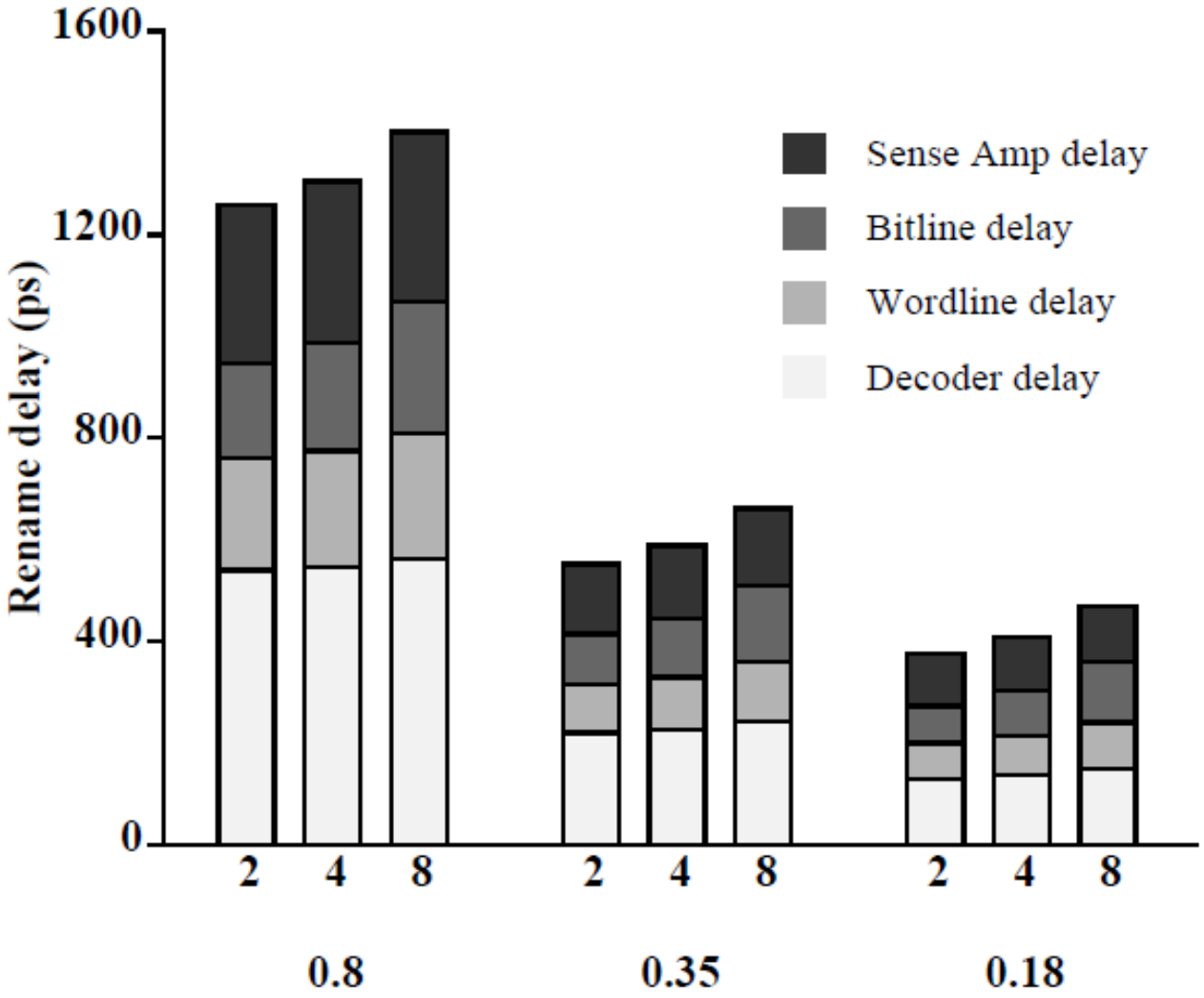

**Palacharla et al., 1997, Figure 2**

# Register Rename Logic Implementations

- **RAM (Random Access Memory) Scheme (e.g., MIPS R10000)**
  - Logical register used to index table, corresponding entry contains current physical register mapping
  - Number of entries = number of logical registers
- **CAM (Content-Addressable Memory) Scheme (e.g., DEC 21264)**
  - Number of entries = number of physical registers
  - Each entry contains current logical register mapped to this physical register, and a valid bit
  - Renaming done by CAM matching on logical register field
- **Dependence check logic done in parallel and has lower latency**
  - Only map access is on the critical path

# Rename Delay vs. Issue Width

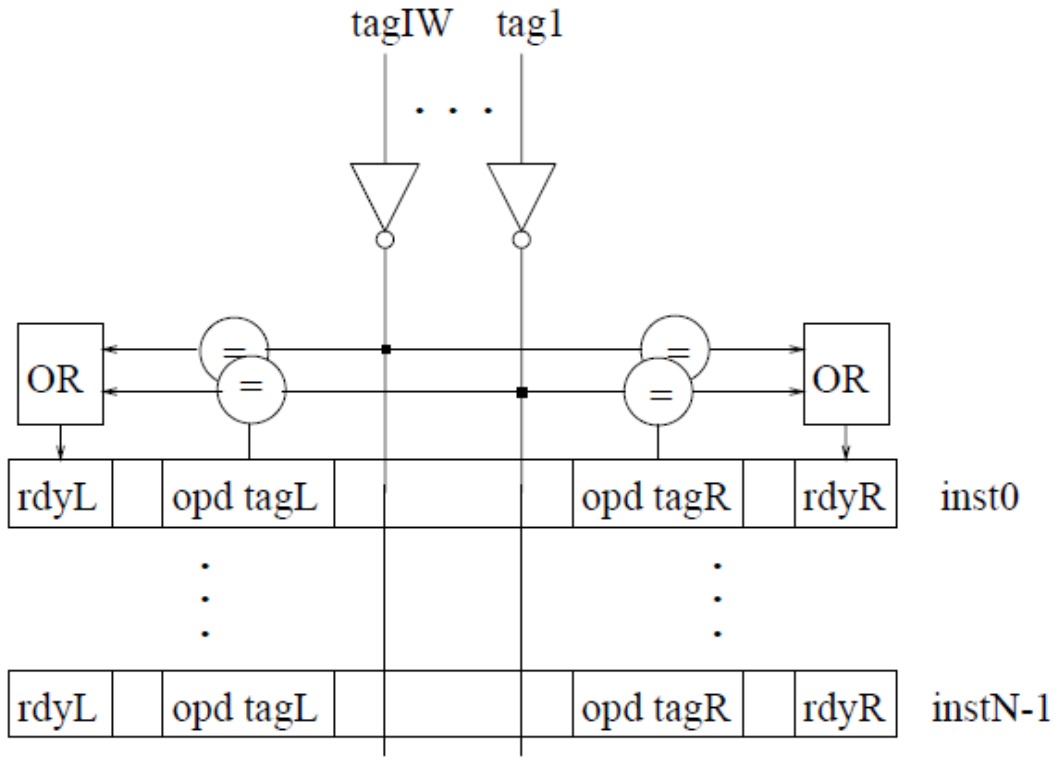
- **RAM scheme acts like a random access memory:**
  - Address decoders drive wordlines
  - Accessed lines discharge on bitlines
  - Bitline changes are sensed by a sense amplifier which produces output
- **Components of Rename delay (RAM):**
  1. Decoder delay
  2. Wordline delay
  3. Bitline delay
  4. Sense amplifier delay
- **First three components are linear with issue width**




Palacharla et al., 1997, Figure 3

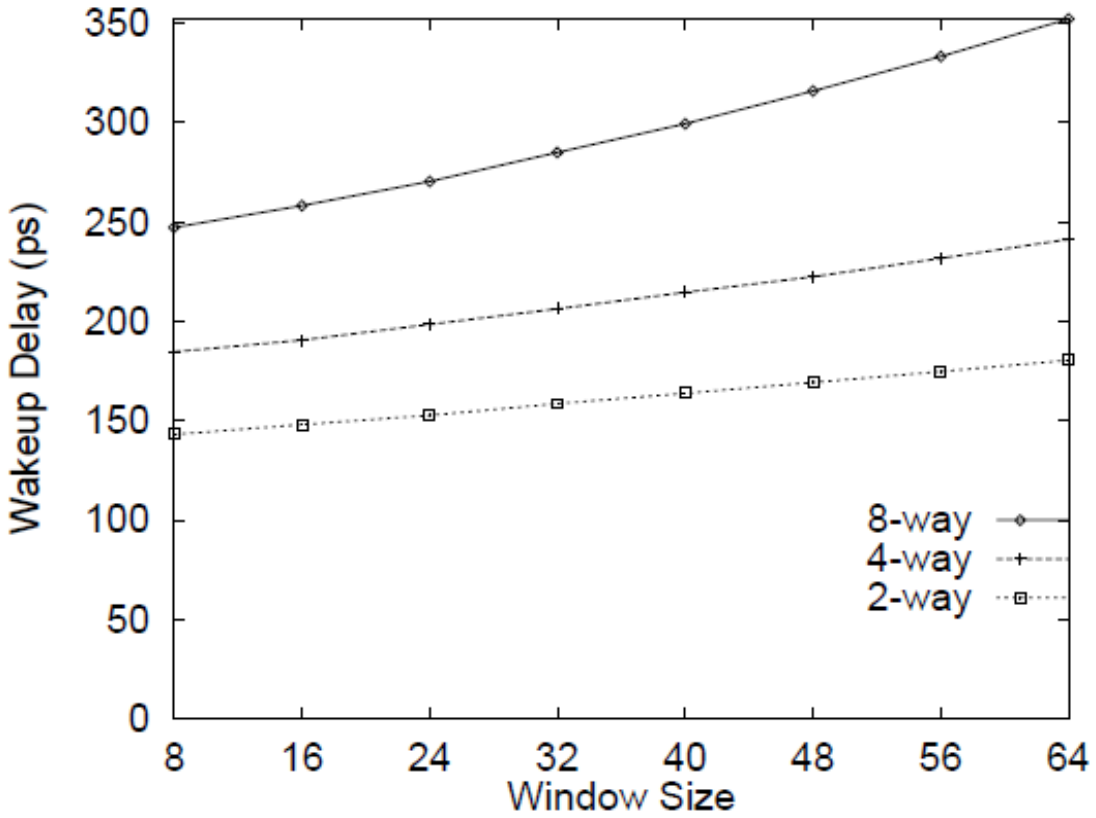
# Wakeup Logic

- **Responsible for updating source dependences for instructions in issue window waiting for source operands to become available**
- **Operation**
  - When a result is produced, result tag is broadcast to all instructions in issue window
  - Each instruction compares tag to its source operands
  - If match, the operand is marked as available by setting appropriate flag (rdyL or rdyR)
  - Once both operands of an instruction are available, the ready flag is set (instruction is ready to execute)
- **Issue width is a CAM, buffers drive the result tags**
- **Each CAM entry has 2xIW comparators to compare each of the result tags against the two operand tags of the entry**

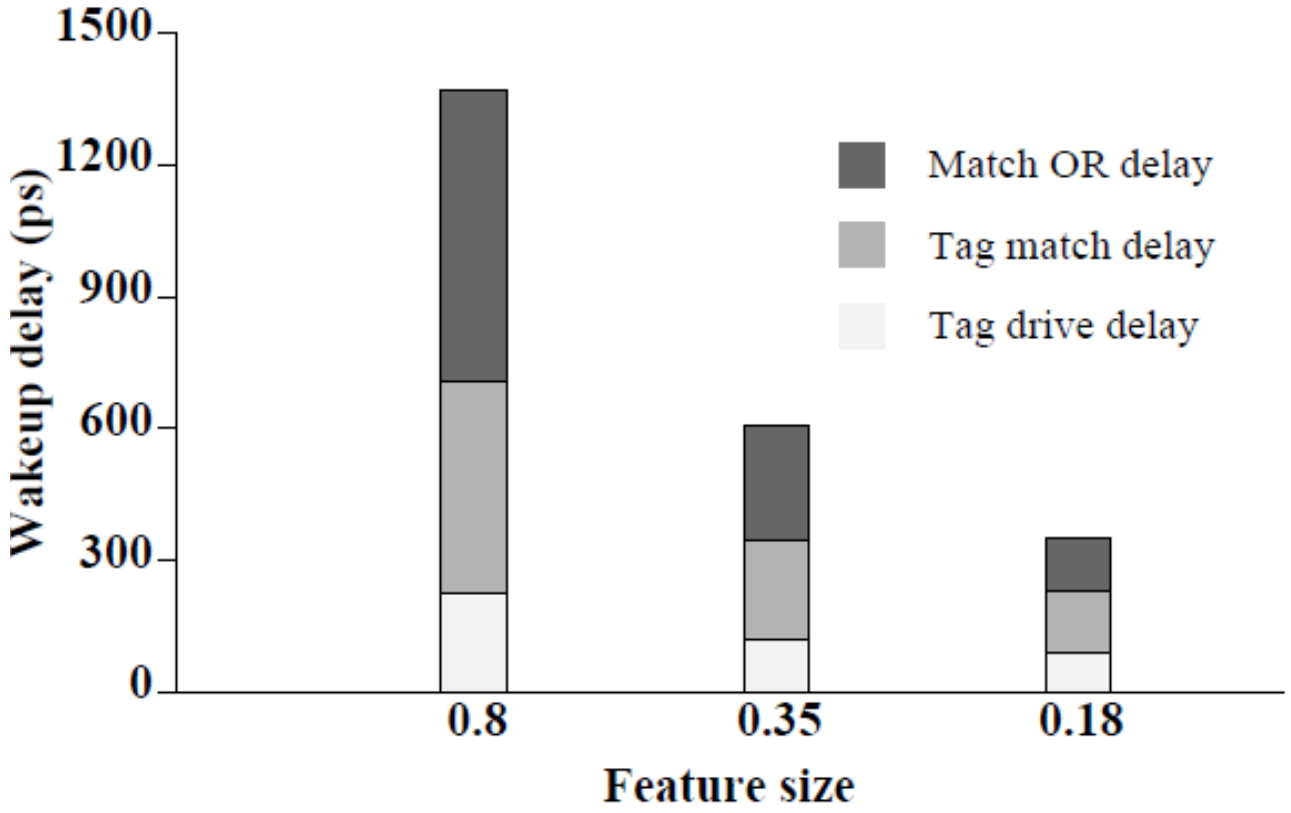



**Palacharla et al., 1997, Figure 4**

# Wakeup Logic Delay




Palacharla et al., 1997, Figure 5

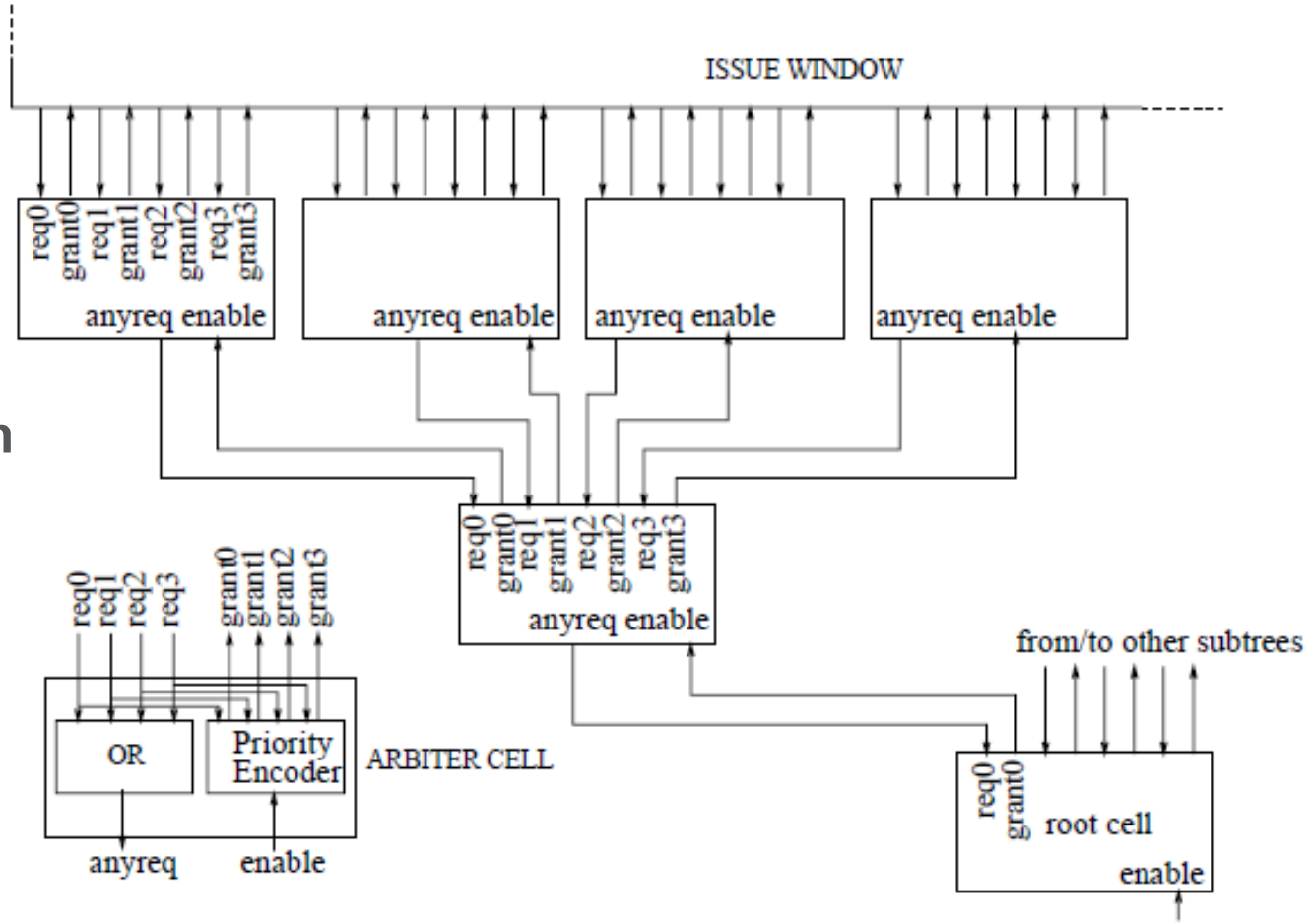



Palacharla et al., 1997, Figure 6

- Delay = $T_{tagdrive} + T_{tagmatch} + T_{matchOR}$
- All components are quadratic with issue width, $T_{tagdrive}$ also quadratic with window size

# Selection Logic

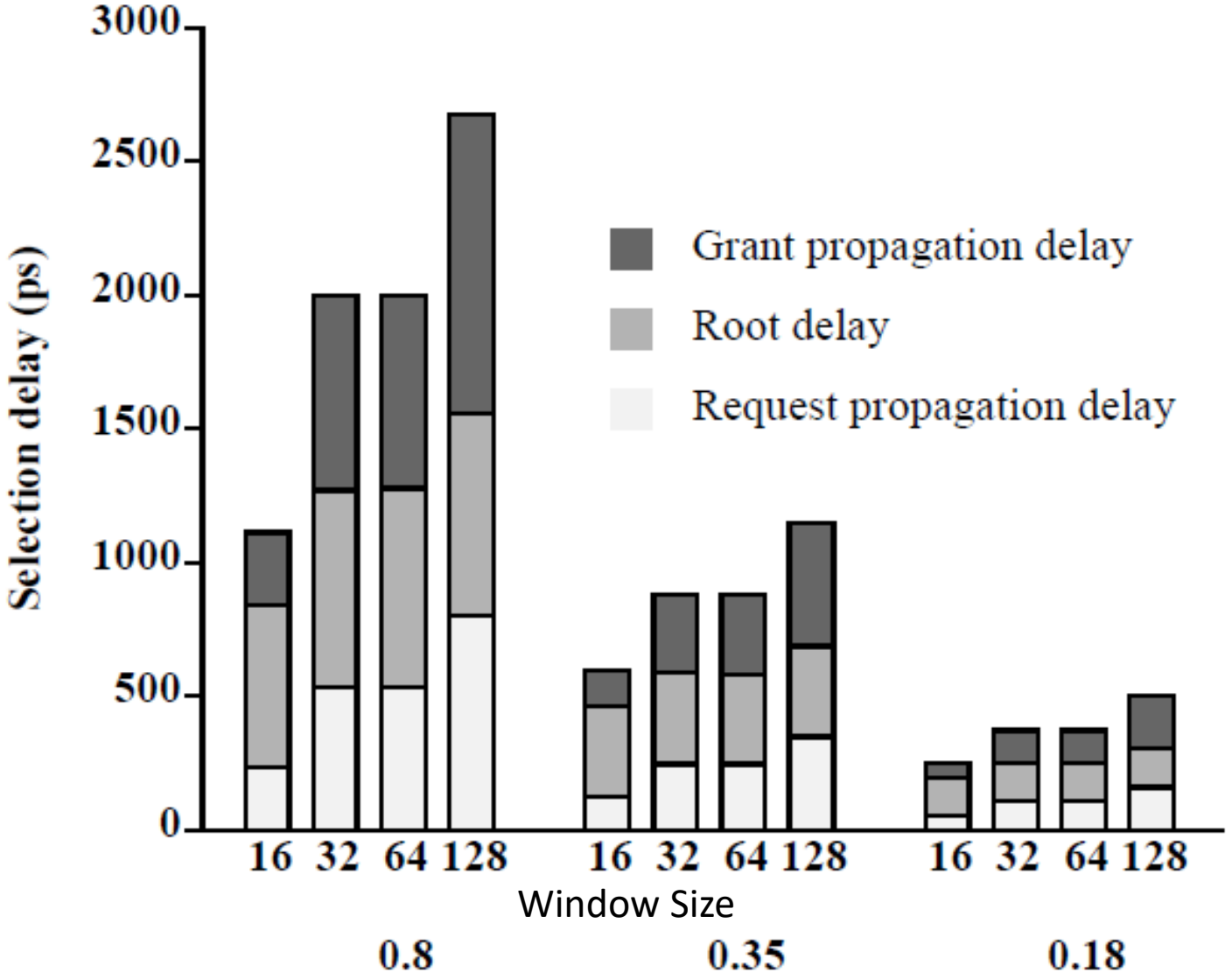
- **Responsible for choosing instructions for execution from the pool of ready instructions**
- **Inputs: request signals, one per instruction that is set when wakeup logic detects all operands are ready**
- **Outputs: grant signals, one per request signal that allows the instruction to be issued to the functional unit**
- **Selection policy decides which requesting instruction to grant (e.g., oldest first)**
- **Structure: Tree of arbiters operating in two phases:**
  1. Request signals propagate up the tree, root detects if any instruction is ready, root grants functional unit to one of its children
  2. Grant signal propagates down the tree to the selected instruction




Palacharla et al., 1997, Figure 7

# Selection Logic Delay

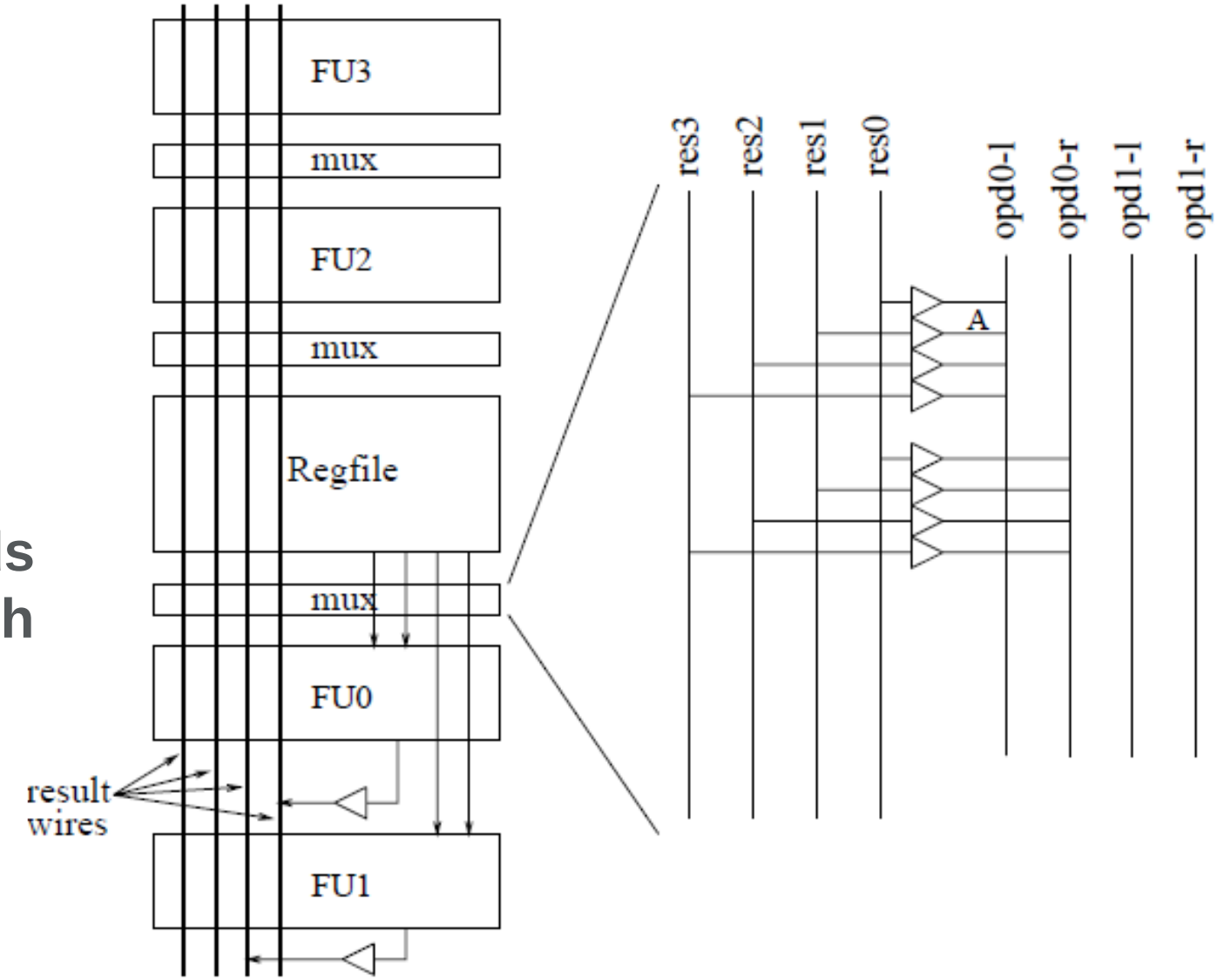
- **Root delay fixed.**
- **Propagation delays proportional to height of arbitration tree**
  - Height ~ $\log_4$(Window size)
  - Independent of issue width




**Palacharla et al., 1997, Figure 8**

# Data Bypass Logic

- **Responsible for forwarding results from completing instructions to dependent instructions, bypassing register file**
- **Number of bypass paths depends on pipeline depth and issue width**
  - Assuming 2-input functional units, IW issue width, S pipe stages after first result-producing stage, we need: $2 \times IW^2 \times S$ paths
- **Two components of data bypass logic: Datapath and control**




Palacharla et al., 1997, Figure 9

# Data Bypass Logic Components

- **Datapath:**
  - Result busses used to broadcast bypass values from each functional unit source to all possible destinations
  - Buffers used to drive bypass values on result busses
- **Control logic:**
  - Controls operand MUXes
  - Compares tags of result values with tags of source value required at each functional unit
  - Sets appropriate MUX control signals on match
- **Delay is dominated by datapath not control**
- **Bypass delays Table:**

| Issue width | Wire length ($\lambda$) | Delay (ps) |
|---|---|---|
| 4 | 20500 | 184.9 |
| 8 | 49000 | 1056.4 |

**Palacharla et al., 1997, Table 1**

# Comparing All Delay Components

| Issue width | Window size | Rename delay (ps) | Wakeup+Select delay (ps) | Bypass delay (ps) |
|---|---|---|---|---|
| 0.8$\mu$m technology | | | | |
| 4 | 32 | 1577.9 | 2903.7 | 184.9 |
| 8 | 64 | 1710.5 | 3369.4 | 1056.4 |
| 0.35$\mu$m technology | | | | |
| 4 | 32 | 627.2 | 1248.4 | 184.9 |
| 8 | 64 | 726.6 | 1484.8 | 1056.4 |
| 0.18$\mu$m technology | | | | |
| 4 | 32 | 351.0 | 578.0 | 184.9 |
| 8 | 64 | 427.9 | 724.0 | 1056.4 |

Palacharla et al., 1997, Table 2

# Memory Ordering

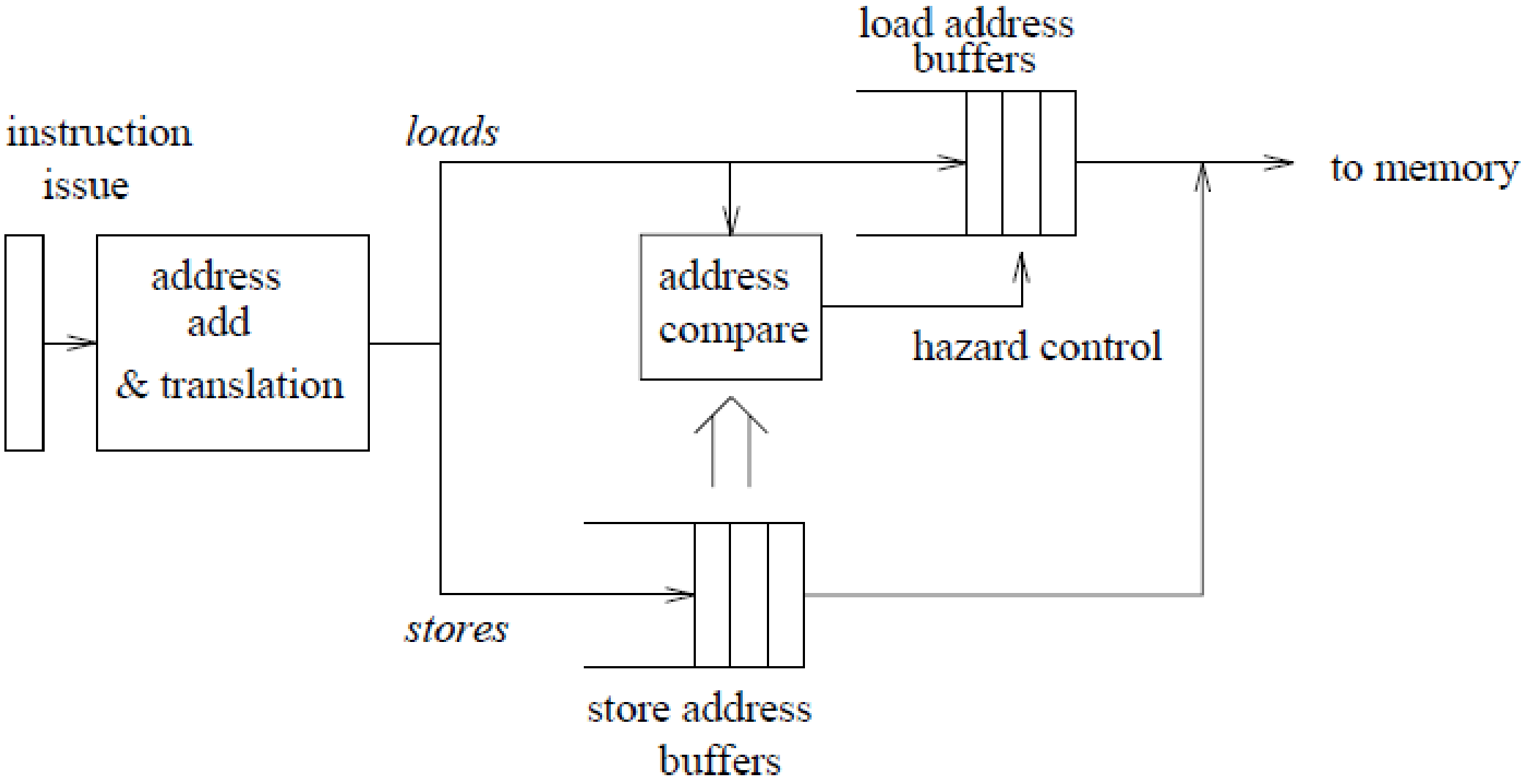
# Handling Memory Operations

- **Some instructions (loads, stores) have memory operands, and need to access the memory hierarchy**
  - Accesses performed in the “memory” stage of the superscalar processor pipeline
- **To maximize ILP, loads/stores may execute out of order**
- **Memory operations require special handling**
  - Recall that Register dependences are identified at decode time
    - Allows early renaming to remove false dependences
    - Maximizes ILP
  - But Memory dependences cannot be determined before execution since memory addresses need to be computed first
  - False dependences exist in memory execution stream
    - For example, multiple stores to the same bytes (WAW)
    - Frequent due to stack pushes and pops

# How Processors Execute Memory Operations

- **Functions that a processor has to do for memory operations:**
  - Enforcing memory dependences among loads and stores
  - Stores/Writes to memory need to be **ordered and non-speculative**
  - Stores issue to the data cache after retirement
- **Loads and stores ordering is enforced using load and store buffers while allowing out of order execution**
- **Stores consist of address and data uops. Store addresses are buffered in a queue**
- **Store addresses remain buffered until:**
  - Store data is available AND Store instruction is committed in the reorder buffer
- **New load addresses are checked with the waiting store addresses. If there is a match:**
  - The load waits OR Store data is bypassed to the matching load

# Load and Store Buffers




Smith & Sohi 1995, Figure 11

# Store Buffer

- **Store addresses are buffered in a queue**
  - ➢Entries allocated in program order at rename
- **Store addresses remain buffered until:**
  - ➢Store data is available
  - ➢Store instruction is retired in the reorder buffer
- **New load addresses are checked with the waiting older store addresses**
  - ➢If there is a match:
    - ❑The load waits OR
    - ❑Store data is bypassed to the matching load
  - ➢If there is no match:
    - ❑Load can execute speculatively OR
    - ❑Load waits till all prior store addresses are computed

# Store Buffer Operation

- **To match loads with store addresses, the store buffer is typically organized as a fully-associative queue (could also be set-associative)**
  - An address comparator in each buffer entry compares each store address to the address of a load issued to the data cache
  - Multiple stores to same address may be present
  - Load-store address match is qualified with “age” information to match a load to the last preceding store in program order
    - Age is typically checked by attaching the number of the preceding store entry to each load at rename
    - Effectively provides a form of “memory renaming”

# Load Buffer

- **Loads can speculatively issue to the data cache out-of-order**
- **But load issue may stall**
  - Unavailable resources/ports in the data cache
  - Unknown store addresses
  - Delayed store data execution
  - Memory mapped I/O
  - Lock operations
  - Misaligned loads

# Load Buffer Operation

- **Load buffer is provided for stalled loads to wait**
- **Load entries are allocated in program order at rename**
- **A stalled load simply waits in its buffer entry until stall condition is removed**
  - ➢Scheduling logic checks for awakened loads and re-issues them to the data cache, typically in program order

# Load-Store Dependence Speculation

- **Load buffers are sometimes used for load-store dependence speculation**
  - When a load issues ahead of a preceding store, it is impossible to perform address match
  - Option 1: Stall the load until all prior store addresses are computed
    - Significant performance impact in machines with deep, wide pipelines
  - Option 2: Speculate that the load does not depend on the previous unknown stores
    - Needs misprediction detection and recovery mechanism
- **One detection mechanism is to make the load buffer a fully associative queue**
  - Store addresses are checked against all previously issued load addresses
  - Only younger loads need to be checked

# Memory Consistency

- **Load buffer snoops other processors' stores to maintain memory consistency on some processors**
- **Stores from other threads on the same processor need to be snooped in the load buffer to maintain memory consistency**
- **Memory consistency models define ordering requirements of loads and stores to different addresses and from multiple processors**
  - ➢Discussed later in the course
- **Examples for memory consistency models**
  - ➢Sequential Consistency (SC)
  - ➢Total Store Order (TSO)

# Memory Dependence Prediction

- **Memory Order Violation: A load is executed before an older store, reads the wrong value**
- **False Dependence: Loads wait unnecessarily for stores to different addresses**
- **Goals of Memory Dependence Prediction:**
  - ➢Predict the load instructions that would cause a memory-order violation
  - ➢Delay execution of these loads only as long as necessary to avoid violations

# Memory Dependence Prediction

- **Memory misprediction recovery is expensive**
  - Requires a pipeline flush if a store address matches a younger issued load
  - Compare to branch mispredictions
- **Memory dependence prediction minimizes such mispredictions**
  - Issue younger loads if predictor predicts “no dependence”
  - Stall younger loads if predictor predicts “dependence”
  - With a good predictor, this approach minimizes unnecessary stalls (false dependences) as well as pipeline flushes from mispredictions

# Alternatives to Memory Dependence Prediction

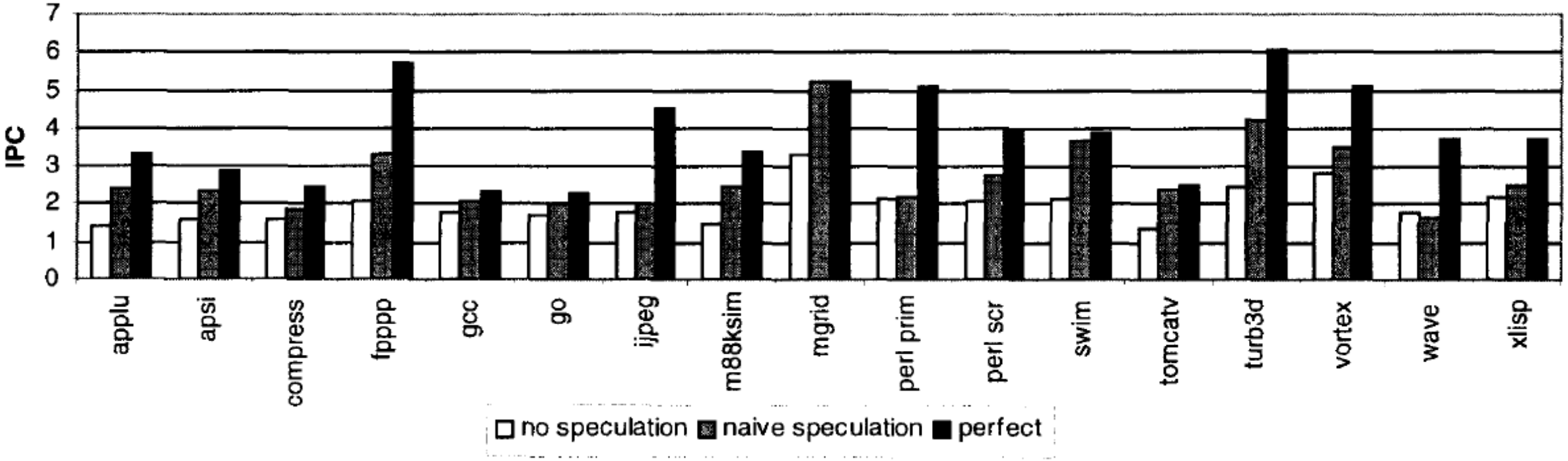
- **No Speculation**
  - Issue for any load waits till prior stores have issued
- **Naïve Speculation**
  - Always issue and execute loads when their register dependences are satisfied, regardless of memory dependences
- **Statistics for some benchmarks:**

| | Naive Speculation | | No Speculation |
|---|---|---|---|
| Spec95 Program | Memory Order Viols Per 1K Instrs | Memory Trap Penalty (Cycles) | False Dep. Per 1K Instrs |
| go | 6 | 13 | 157 |
| m88ksim | 20 | 12 | 168 |
| gcc | 5 | 15 | 187 |
| compress | 11 | 15 | 129 |
| xlisp | 11 | 14 | 179 |
| ijpeg | 23 | 15 | 150 |
| perl prim | 20 | 15 | 215 |
| perl scrab | 10 | 15 | 185 |
| vortex | 7 | 19 | 215 |
| tomcatv | 4 | 22 | 264 |
| swim | 2 | 36 | 224 |
| mgrid | 0 | 18 | 262 |
| applu | 18 | 22 | 212 |
| apsi | 7 | 35 | 247 |
| fpppp | 10 | 17 | 275 |
| wave5 | 24 | 21 | 188 |
| turb3d | 6 | 16 | 213 |

Chrysos&Emer, 1998, Table 3.1

# Perfect Memory Dependence Prediction

- Does not cause memory order violations
- Avoids all false dependences




Chrysos&Emer, 1998, Figure 3.1

# Store Sets

- **Based on the assumption that future dependences can be predicted from past behavior**
- **Each load has a store set consisting of all stores upon which it has ever depended**
  - Store is identified by its PC
- **When program starts, all loads have empty store sets**
- **When a memory order violation happens, store is added to load's store set**

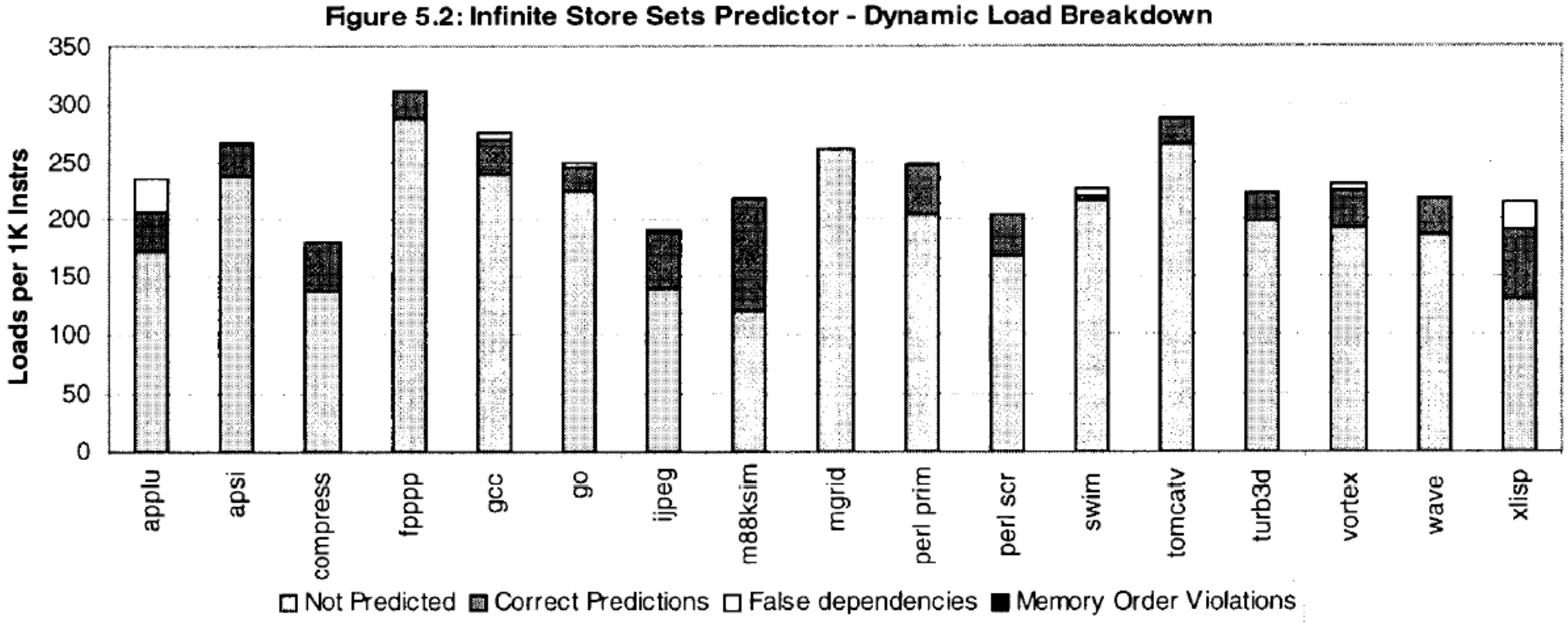
# Store Set Example

| PC | Inst | |
|---|---|---|
| 0 | Store C | |
| 4 | Store A | |
| 8 | Store B | |
| 12 | Store C | |
| … | | |
| 28 | Load B | SS = {PC 8} |
| 32 | Load D | SS = { } |
| 36 | Load C | SS = {PC 0, PC 12} |
| 40 | Load A | SS = {PC 4} |

# Store Set Performance

- **Infinite SS configuration (#sets, #elements/set are not limited)**
- **Each dynamic load is classified as:**
  - Not predicted (loads with empty store sets)
  - Correctly predicted
  - False dependence (unnecessary wait)
  - Memory order violation (dependence not predicted)

**Chrysos&Emer, 1998, Figure 5.2**

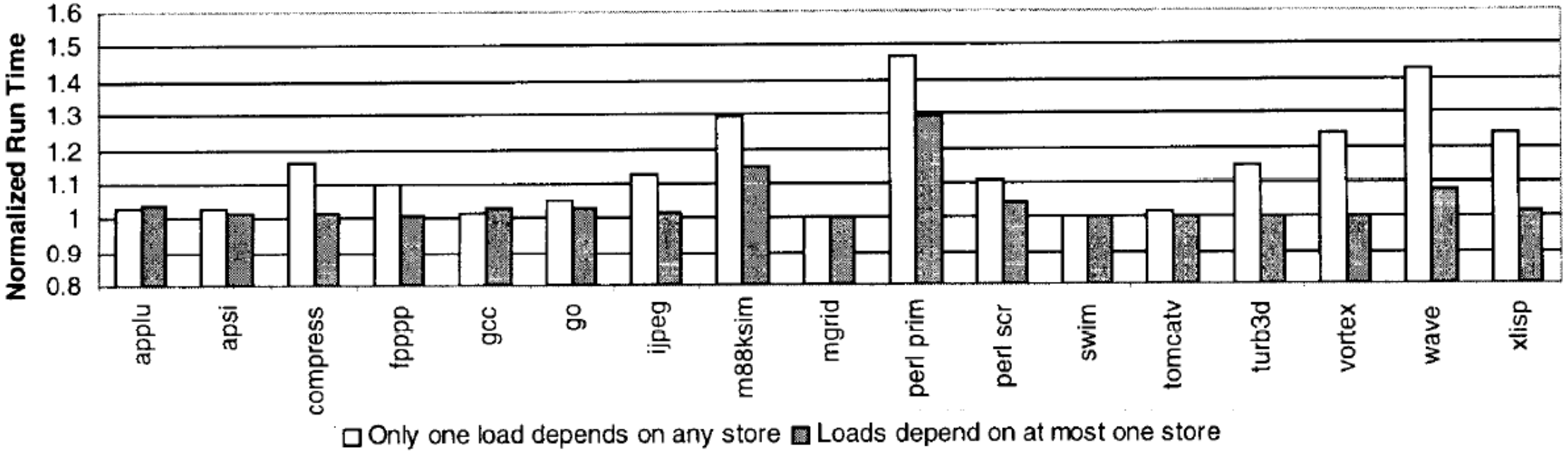


# More Practical Store Set Performance

- **Hardware resources are not infinite, so we cannot allow infinitely large store sets per load**
- **Results when limiting store sets:**
  - ➢At most one load can depend on any store
  - ➢Each load depends on at most one store

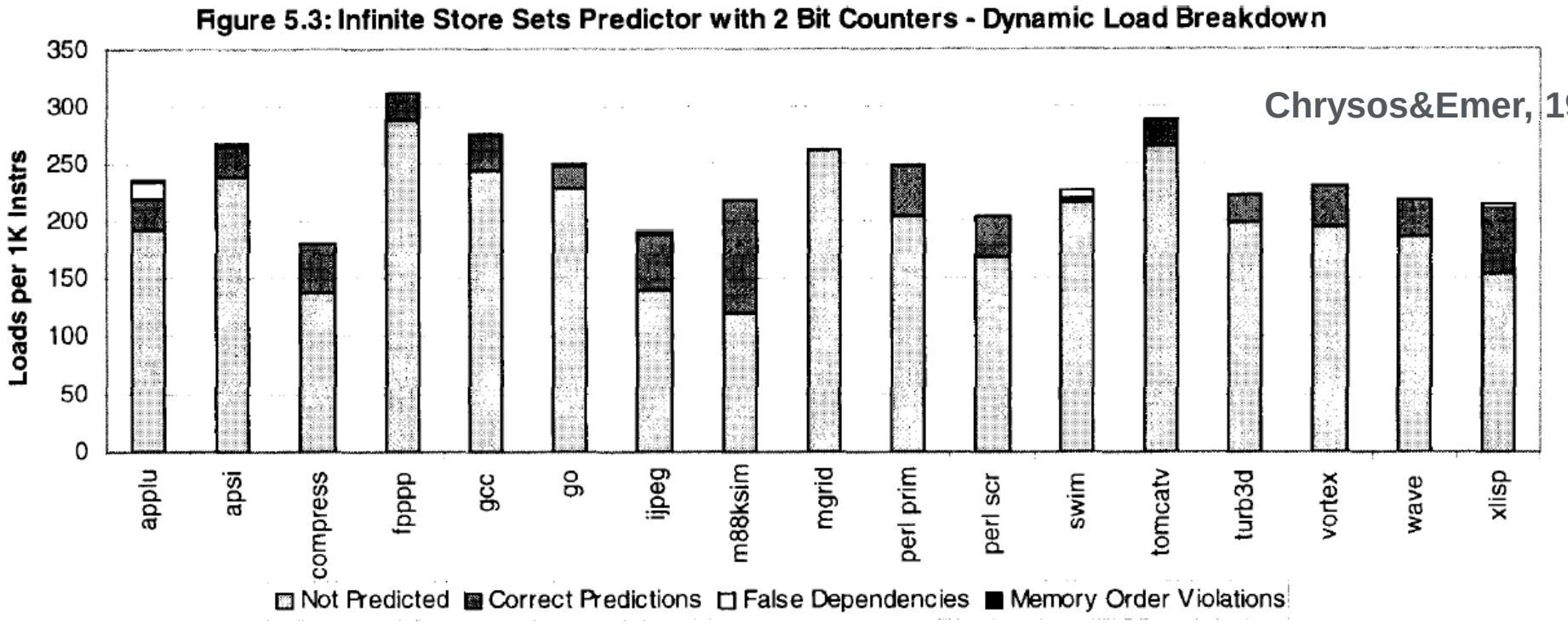
Chrysos&Emer, 1998, Figure 5.1

**Figure 5.1: Effect on Run Time When Not Allowing Multiple Dependence Flexibility**



# Reducing False Dependences

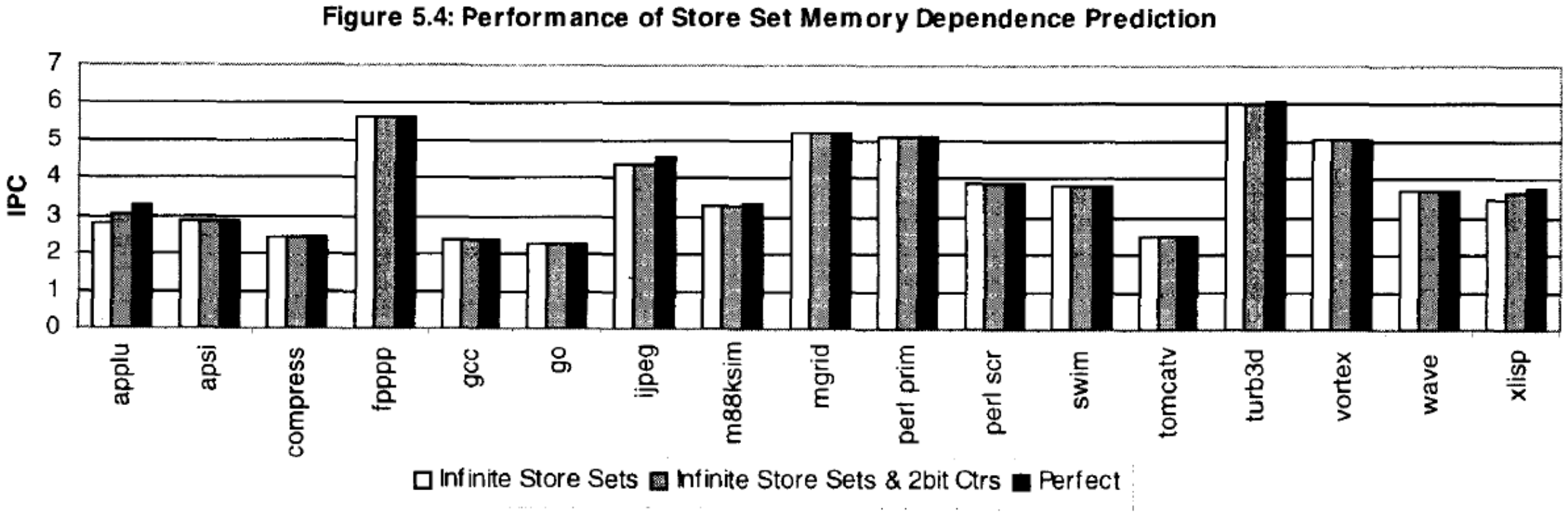
- **With infinite SS, a store dependence remains in a load's store set forever, even if some dynamic instances of the load are independent**
- **To reduce false dependences, we can use 2-bit saturating counters**
  - Set to max value (3) on a memory order violation
  - Decremented if real dependence doesn't exist, incremented if real dependence exists
  - Counter values of 2 or 3 cause load to wait; otherwise no dependence is assumed




Chrysos&Emer, 1998, Figure 5.3

# Store Set Comparison to Perfect Prediction

Chrysos&Emer, 1998, Figure 5.4

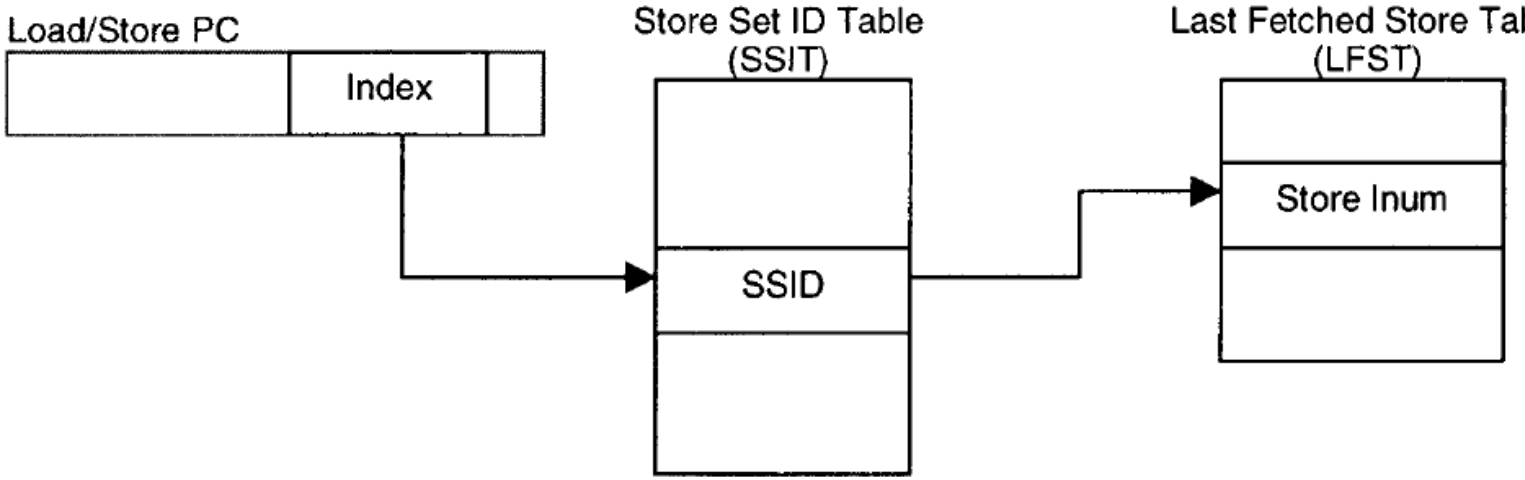



- Infinite SS with 2-bit saturating counters are very close to perfect memory dependence prediction

# Practical Store Set Implementation

- **Store Set Identifier Table (SSIT): PC-indexed, maintains store sets**
- **Last Fetched Store Table (LFST) maintains dynamic inst. count about most recently fetched store for each store set**

**Figure 6.1: Implementation of Store Sets Memory Dependence Prediction**




Loads and stores index into the SSIT to get their store set identifiers, which are used to access and update the LFST. The store inums that are found in the LFST indicate the memory dependence prediction.

Chrysos&Emer, 1998, Figure 6.1

- **Limitations:**
  - Store PCs exist in one store set at a time
  - Two loads depending on the same store can share a store set
  - All stores in a store set are executed in order

# Implementation Details

- **Recently fetched loads**
  - Access SSIT based on their PC, get their SSID
  - If SSID is valid, LFST is accessed to get most recent store in the load's store set
- **Recently fetched stores**
  - Access SSIT based on their PC
  - If SSID is valid, then store belongs to a valid store set
    - Access LFST to get most recently fetched store information in its store set
    - Update LFST inserting its own dynamic inst. count since it is now the last fetched store in that store set
    - After store is issued, it invalidates the LFST entry if it refers to itself to ensure loads & stores are only dependent on stores that haven't been issued

# Store Set Interference

- **Destructive interference happens because stores can belong to only one store set**

**Example:**

**Load PC 1 → Store Set 1 { Store PC X, Store PC Y, Store PC Z }**

**Load PC 2 → Store Set 2 { Store PC J, Store PC K }**

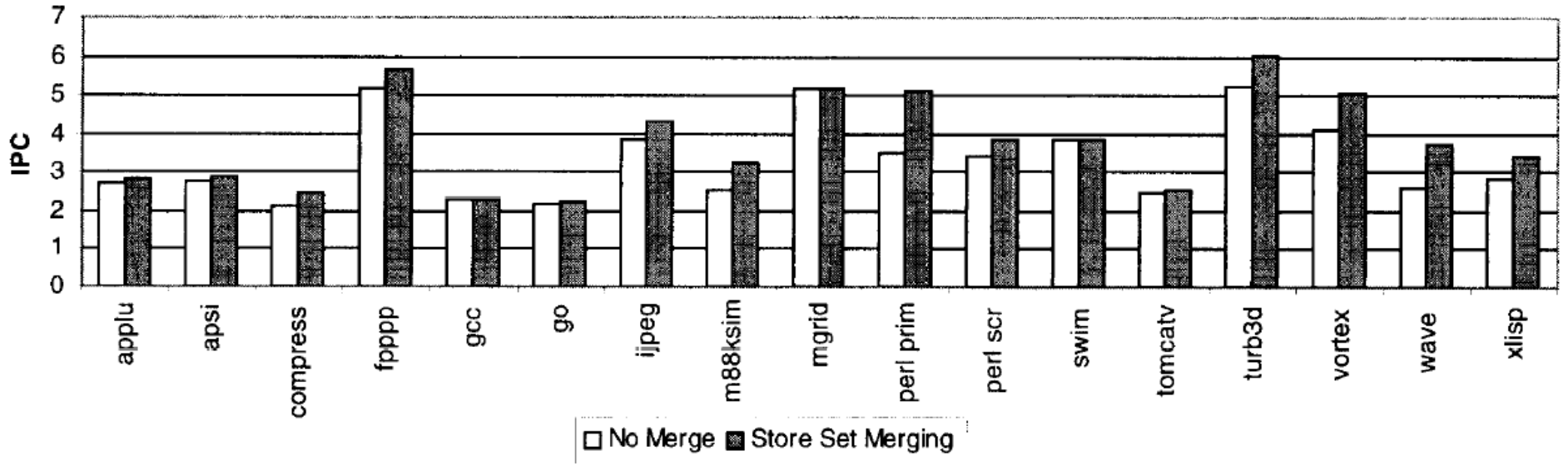
- **Assume that Load PC 1 has a memory order violation with Store PC J**
  - Each store can exist in one SS, so we need to remove Store PC J from SS 2 and add it to SS 1
  - But this causes future memory order violation between Load PC 2 and Store PC J
- **Store set merging avoids the problem**

# Store Set Merging

- **When a store-load pair causes a memory order violation:**
  - If neither has been assigned a store set, a store set is allocated and assigned to both instructions
  - If load has been assigned a store set but the store hasn't, the store is assigned the load's store set
  - If store has been assigned a store set but the load hasn't, the load is assigned the store's load set
  - If both have store sets, one of them is declared the winner, and the instruction belonging to the loser's store set is assigned the winner's store set

**Chrysos&Emer, 1998, Figure 6.2**

**Figure 6.2: Performance Improvement Due to Store Set Merging**

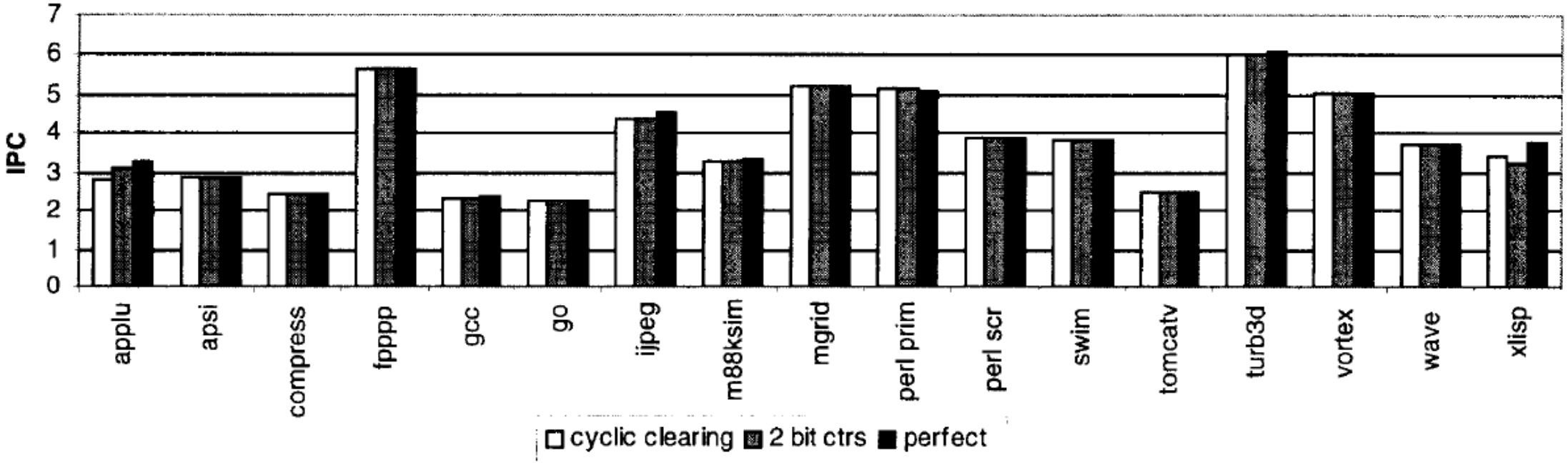


# Store Set Performance

- For the practical SS implementation, cyclic clearing of valid bits (every ~1M cycles) is almost the same as 2-bit saturating counters
- With sufficiently large structures, performance very close to perfect prediction

Chrysos&Emer, 1998, Figure 6.3

**Figure 6.3: Implementation of Store Sets vs. Perfect**



# Reading Assignments

- S. Palacharla, et al., “Complexity-Effective Superscalar Processors,” ISCA 1997.
- G.Z. Chrysos and J.S. Emer, “Memory Dependence Prediction using Store Sets,” ISCA 1998.