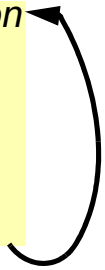
Exploiting Program Behavior

• Program Execution in the abstract:

Faster Circuits -> Faster Execution

1. fetch instruction
2. read inputs
3. calculate
4. store result



• Be “smarter” about program execution:

Exploit Idiosynchrasies in Program Behavior

Examples: 1. Caching

2. Branch Prediction

What to do Next? One Possibility is:

Identify Other Idiosynchrasies in *Typical* Program Behavior

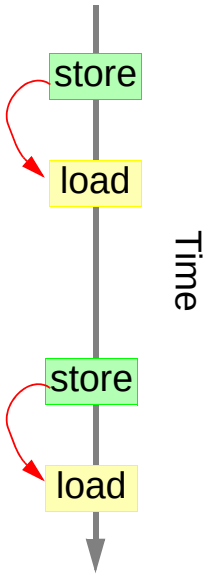
Develop Techniques to Exploit

Memory Dependences are Quite Regular

• Identified a new form of regularity:

Memory Dependence Stream

1. Load/Store has a Dependence?
2. Which Dependence a Load/Store has?



Opportunity to Exploit this Regularity

Techniques are Required to Make Use of this Opportunity

Memory Dependence Prediction

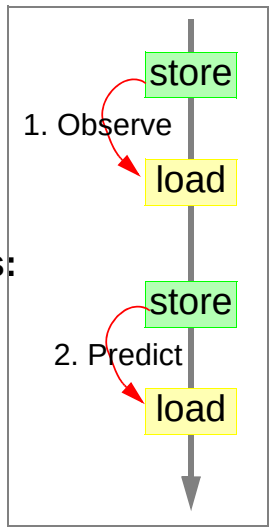
Guess:

1. Load/Store has a Dependence?
2. Which Dependence a Load/Store has?

How?

Past Behavior ->

Good Indicator of Future Behavior



Basis for Three Micro-Architectural Techniques:

GOAL

1. Exploit Load/Store Parallelism
2. Reduce Memory Latency
3. Provide for Multiple Memory Accesses

Dynamic Speculation/Synchronization

Goal: Exploit Load/Store Parallelism

Ideally:

Loads Wait for a Store only when a RAW dependence exists

Determining Dependences vs. Speculating Dependences

safe but delays *balance penalty vs. gain*

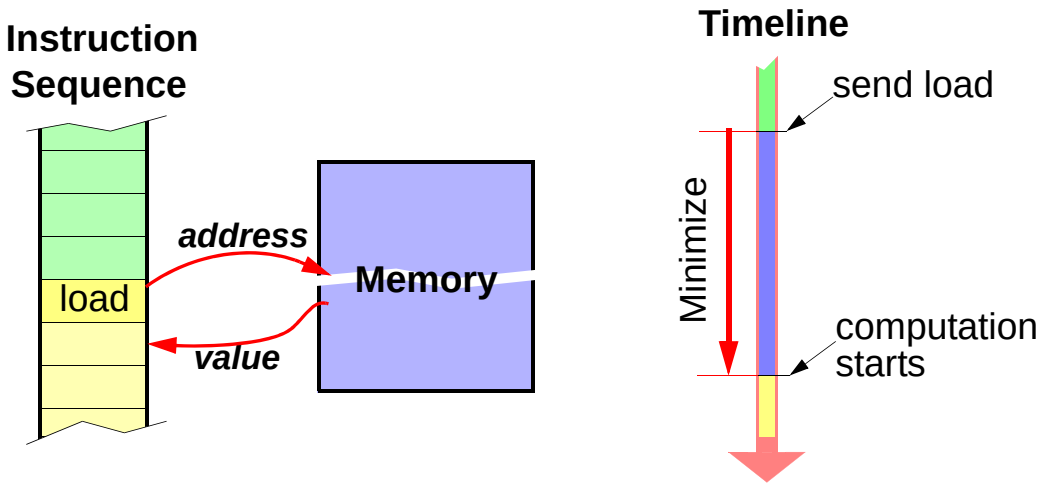
Prior to this work: *Always predict no dependence or No Speculation*

This work:

When Mispeculation Penalty Becomes High

Mimic Ideal: Make loads wait only as long as necessary

In Search of Higher Performance #1



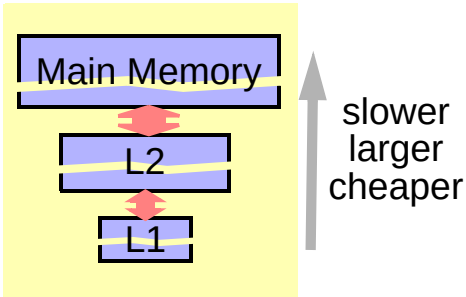
1. Higher Performance: Memory Responds Faster

#1. Making Memory Respond Faster

Ideally: Memory is Large and Fast

Can have it! Technology - Cost trade-off

Solution: Memory Hierarchy



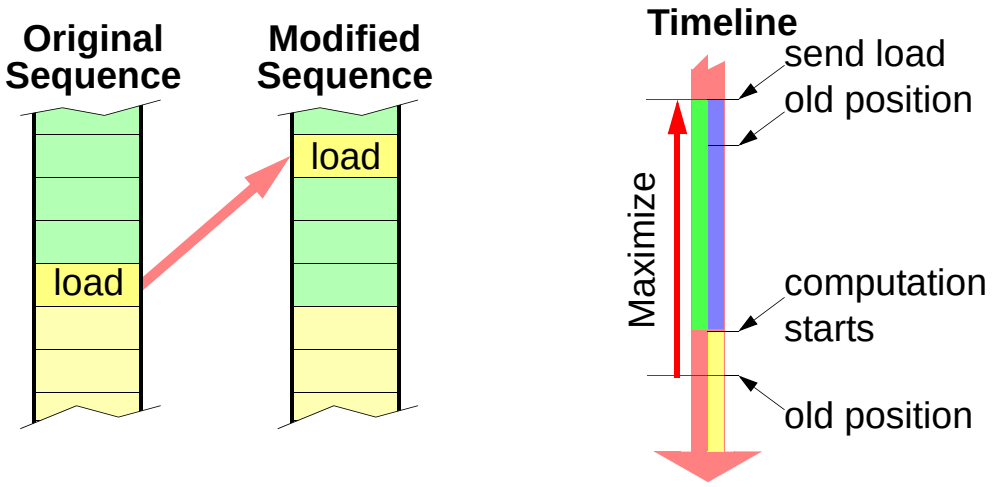
OK, we did the best we could, but ...

...memory is still not that fast

...and it is getting slower

Is this the end?

In Search of Higher Performance #2



2. Higher Performance:

Send Load Request As Far In Advance As Possible

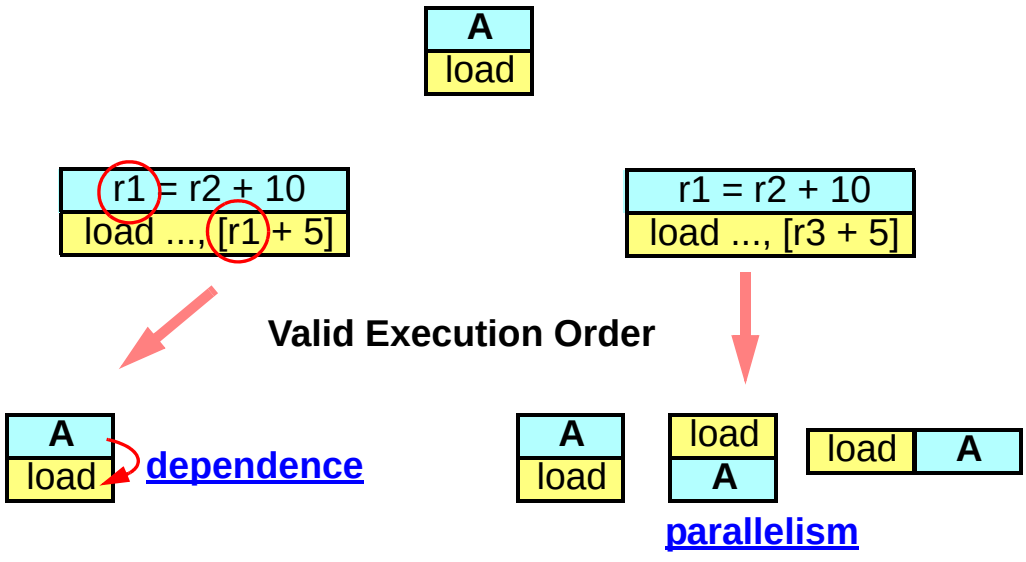
But, will the program still run correctly?

A. Can we ever move loads up?

B. How do we do it?

A. Can We Ever Move Loads Up?

Instruction Sequence

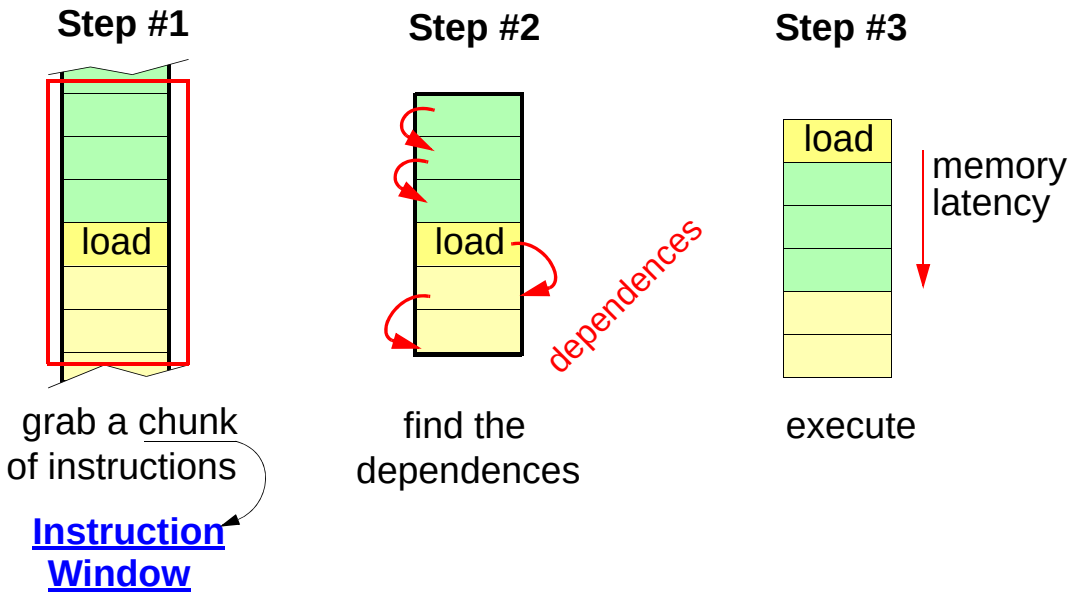


A and load can execute in any order

if load does not use a value produced by A

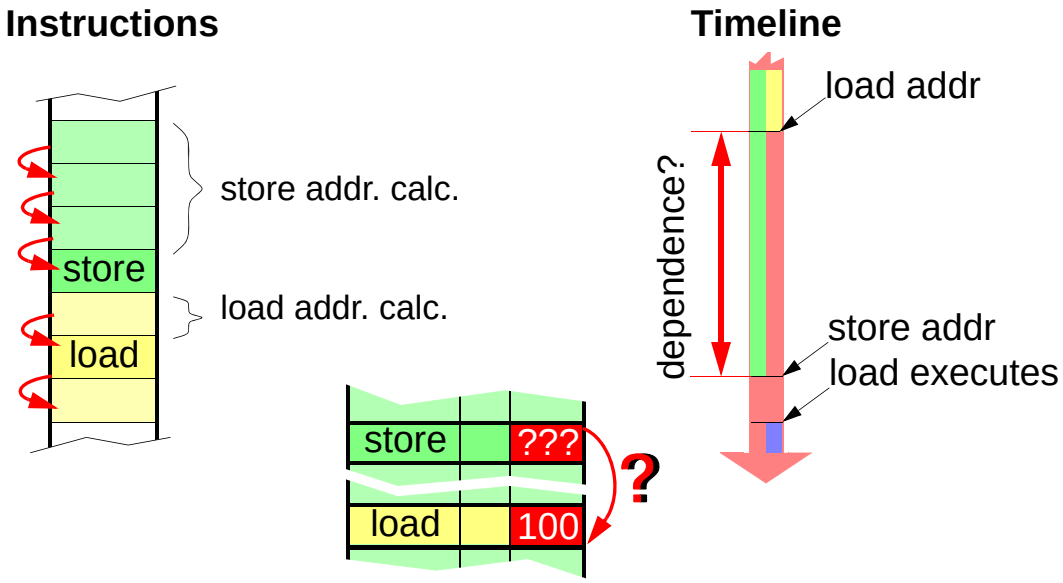
B. How To Move Loads Up?

Instruction Level Parallel Processors:



Use Parallelism to Tolerate Memory Latency

Moving Loads Past Stores



Use Of Addresses Hinders Parallelism

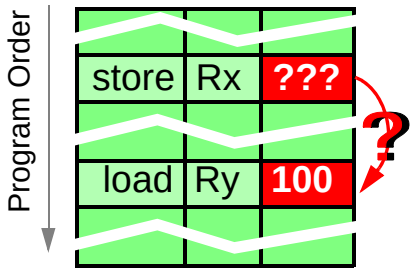
Ô Limits Us in Our Effort to Tolerate Memory Latency

The Goal and The Problem

Goal: Exploit Load/Store parallelism

Ideally

1. Loads w/o Dep. execute at will
2. Loads w/ Dep. synchronize with store



1. Wait to Determine Dependences
   safe, but addresses must be known
2. Speculate on Dependences
   balance gain vs. penalty

Prior to this work:

Naive Speculation or **No-Speculation**

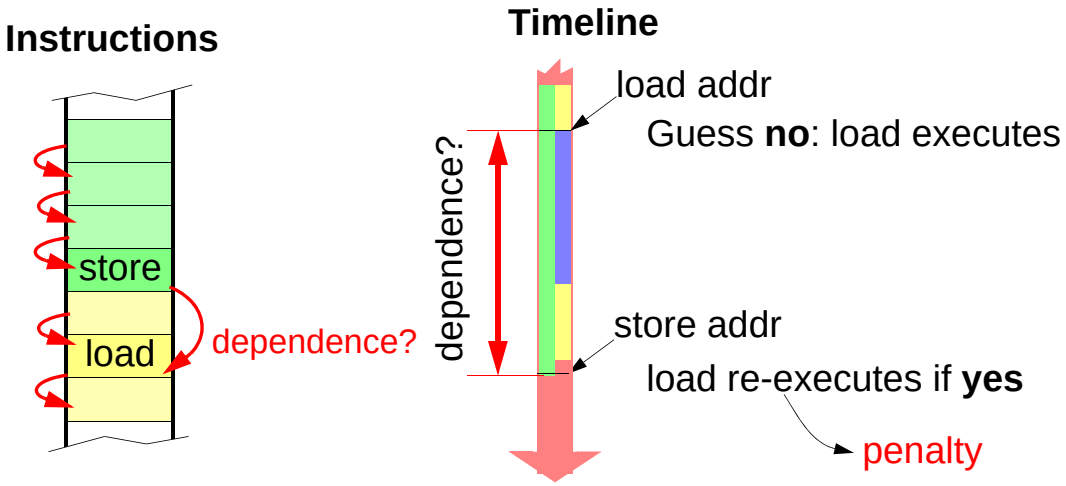
This work:

Speculation and No-speculation gap increases with window

Naive less close to Ideal - Net Mispeculation Penalty

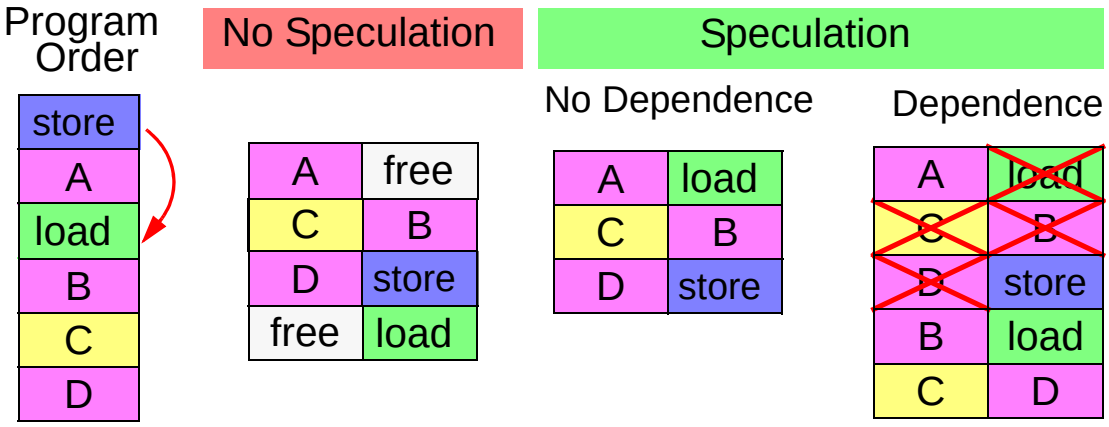
Naive Memory Dependence Speculation

- **Don't give up, be optimistic, guess no dependences exist**
- State-of-the-art in modern processors



Need to Balance: Gain vs. Penalty

Dependence Speculation and Performance



Speculation may affect performance either way

Balance: Gain vs. Penalty

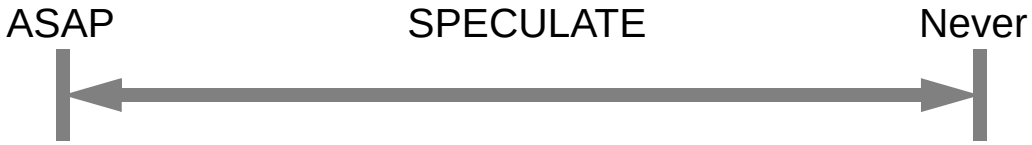
Penalty: (a) work thrown away
(b) opportunity cost

Dependence Speculation and Performance

Performance

Gain ~ (100% - Mis-speculation%)

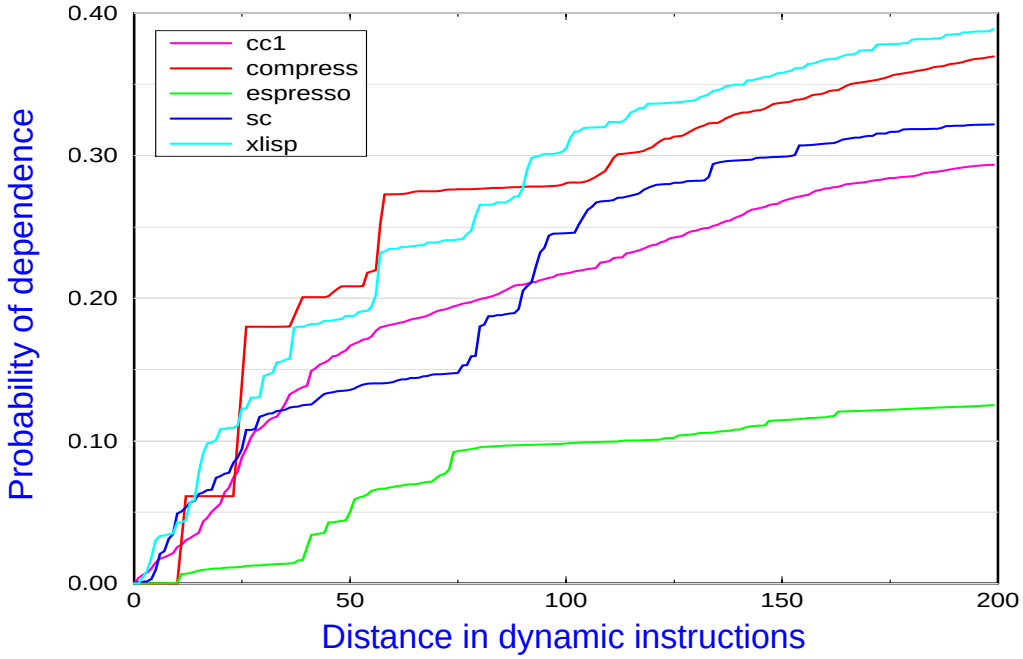
Penalty ~ Mis-speculation%



↑	Gain	↓
↑	Penalty	↓
↑	Mis-speculation%	↓

- Balance between Gain and Penalty

Dependences vs. Window Size



Frequency of loads with Dependences within the Window

Small Instruction Windows and Speculation

Small Instruction Window:

- Loads are speculated past few instructions
- Dependences are infrequent

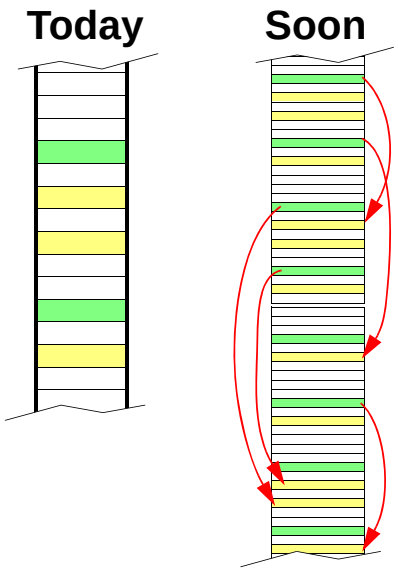
Blind Speculation a good choice:

- Mis-speculations are infrequent
- Low probality of other, independent work
- Low mis-speculation penalty

Not Speculating at times is acceptable.

How About Future Systems?

Common Case:



- Memory will be slower
  Ô Need to move loads further up

Guessing Naively:

Penalty becomes significant

Loads — Ideal Behavior:

No Dependence: execute at will

Dependence: Synchronize w/ store

Future Systems: Wish Dependences Were Known

Wider Instruction Windows

As the Window size increases:

- Loads are speculated past many more instructions
- Dependences become more frequent

Overall:

- Mis-speculations are more frequent
- Higher probability of other, independent work
- Higher mis-speculation penalty

Blind Speculation is still a viable approach

Not Speculating is not

HOWEVER! Net penalty of mis-speculation becomes significant

Reducing Net Mispeculation Penalty

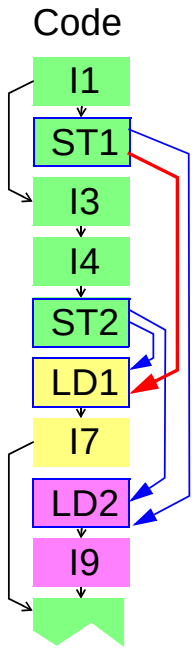
1. Improve the Accuracy of Speculation

2. Reduce the Amount of Work Thrown away on mispeculation

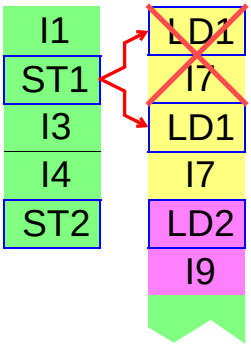
Reducing the Net Mis-speculation Penalty

Ideally:

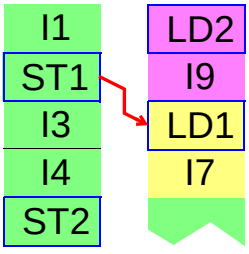
- Dependent load/store pairs are synchronized
- Other loads execute as early as possible



Blind Speculation



Ideal Speculation



When is Mispeculation Penalty a Concern?

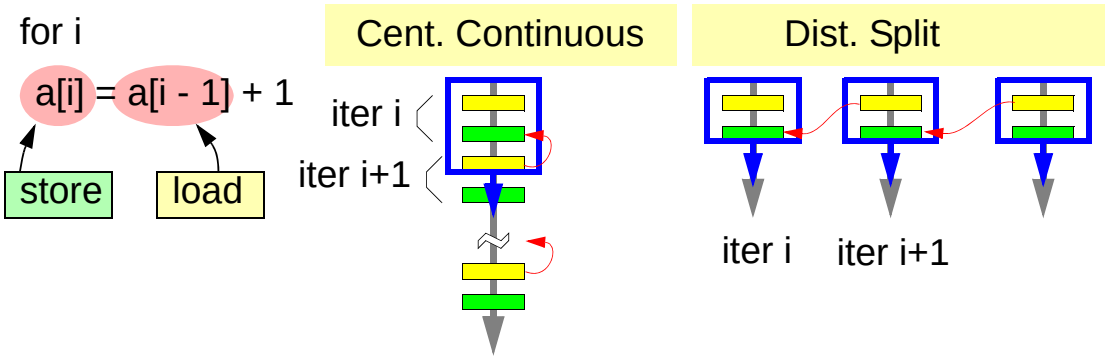
Intelligent Speculation for:

1. Distributed, Split Window

even if address-based information is available

2. Centralized, Continuous Window

if address-based information is not available

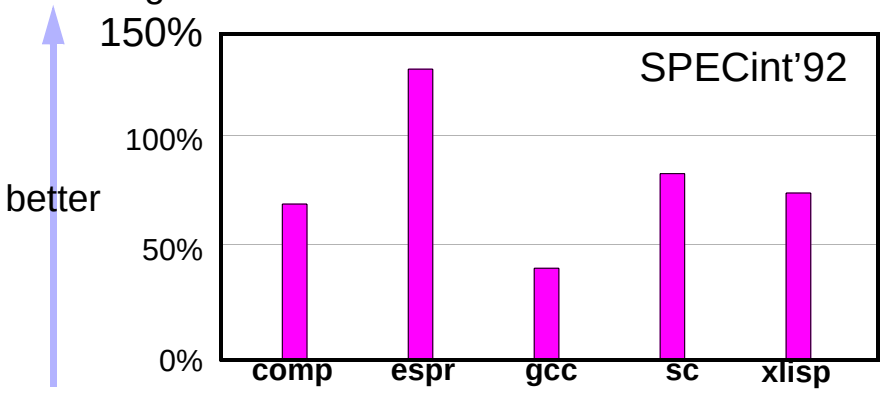


How serious a problem is it really?

What if loads wait till dependences are known

Depends on how aggressive the processor is:

- For small instruction window ~16: no difference
- But for larger windows:

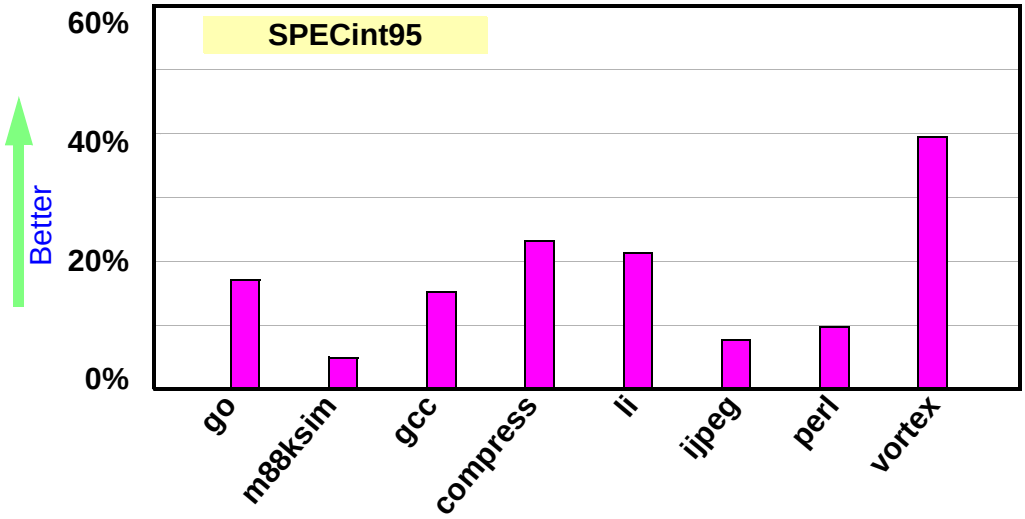


Waiting vs. Perfect Dependence Knowledge

40% - %140 Performance loss

Naive Memory Dependence Speculation-Performance

- **Naive:** *Always* guess that *no* dependence exists
- Works well for today's windows
- **How well can we do on an aggressive processor:**



Future Processors: Wish we knew the dependences

Dependence Speculation/Synchronization

To mimic the ideal we need:

(1). Identify the loads that have dependences

(2). Identify the relevant stores

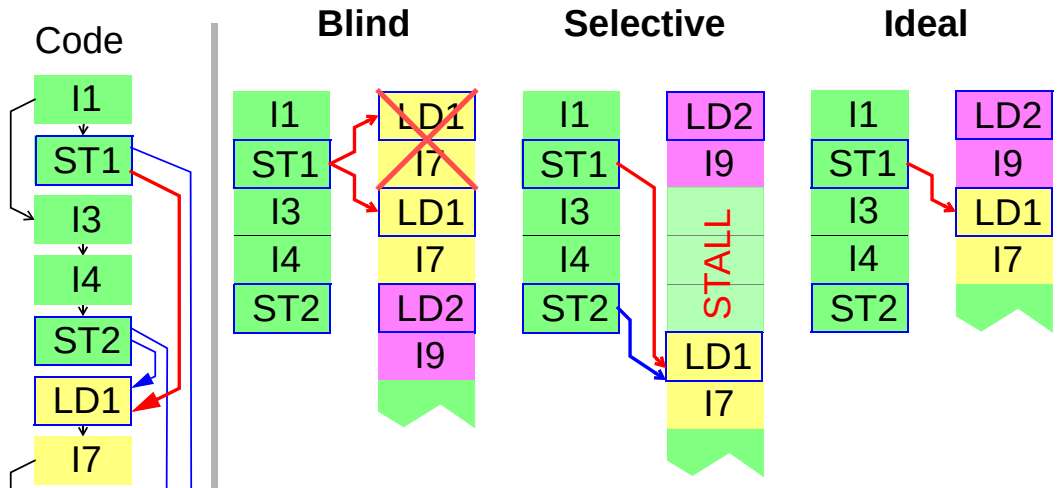
(3). Enforce synchronization

Can we do without synchronization?

How about selective speculation:

- Identify the loads that have dependences
- Do not speculate them

Selective Dependence Speculation



- Selective may perform worse than blind
- Can also perform as well as the ideal
- In practice:
  performance behavior varies

Dependence Speculation Policies

Q1. Which loads should wait

Q2. For how long

No Speculation	
A1. All	A2. For all previous stores
Naive	
A1. None	A2. N/A
Selective (also in Alpha 21264)	
A1. Some	A2. For all previous stores
Synchronization	
A1. Some	A2. For the specific store

Store Barrier (Hesson at al. IBM)

Predict Store and Make all subsequent loads wait

Our Solution

Requirements:

Q1. Which loads should wait? Q2. For how long?

avoid mispeculation *maintain high gain*

Our Solution:

A1. Predict (load, store) dependences

start with naive

learn from mistakes

A2. Synchronize

Our approach

Attempt to mimic the Ideal:

- To identify the dependent load/store pairs:

  Predict!

  Based on the history of mis-speculations

- To synchronize:

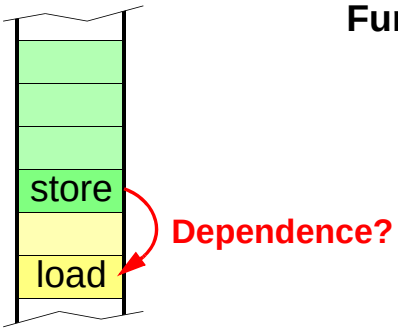
  Use **dynamically assigned** synchronization variables

Memory Dependence Prediction - Goal

Goal: **Report Memory Dependences**

without actual knowledge of the addresses involved

Instructions



Functionality:

store: dependence with which **load**?

load: dependence with which **store**?

When Dependences Are Not Know... Guess Them.

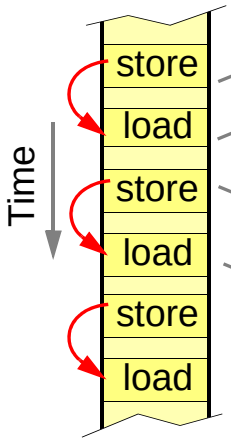
Memory Dependence Prediction

Dependence Behavior: Locality in time

Detect Dependences Ô *next time guess that the same will happen*

Address may vary over time!

Instructions



Record: (*store PC*, *address*)

Probe using *address* → (*store PC, load PC*)

Predict: (*store PC, load PC*)

Predict: (*store PC, load PC*)

```
a[ i ] = a[ i - 1 ]
p->count++
```

Use Dependence History to Predict Future Dependences

Memory Dependence Speculation/Synchronization

A. Predict Dependence

1. Predict load

   Allocate Sync. bit

2. Predict store

   Wait on Sync. bit

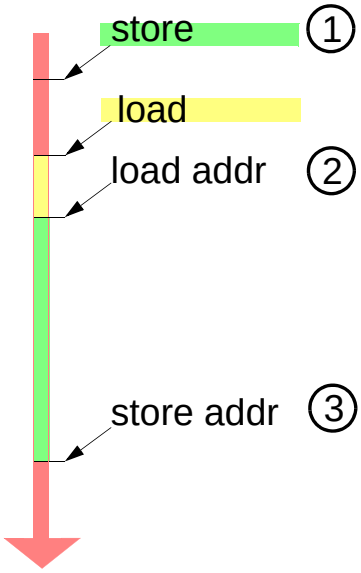
3. Store Signals

   Load executes

B. Predict No Dependence

2. Load executes
3. Store verifies

Timeline

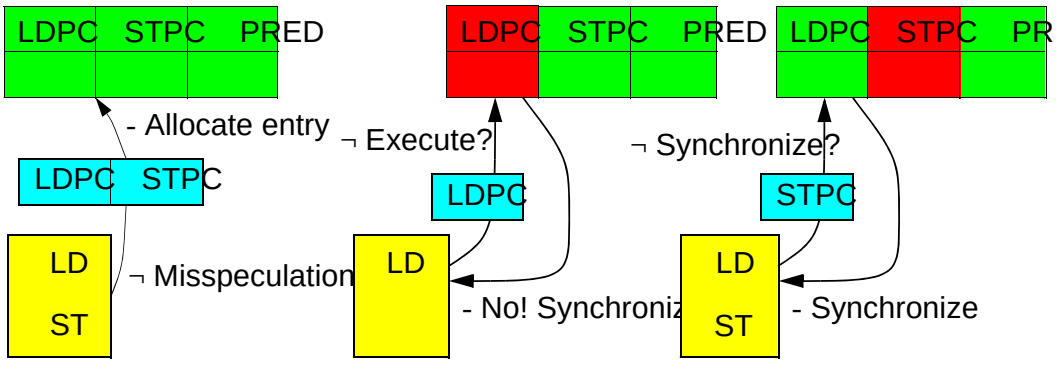


Correct Prediction: Loads wait only as long as it is necessary
Incorrect: Same as Naive or Delay

Predicting Dependences

- Dependence: (Load PC, Store PC)
- Temporal locality - Small Working Set.
- Use a small table to: (1). track recent mis-speculations
  (2). Predict dependences

Memory Dependence Prediction Table



Speculation/Synchronization

Speculation/Synchronization, we need:

Identify

1. Loads with dependences
2. Relevant stores
3. Enforce synchronization

How we do it:

- Parts **(1)** & **(2)**: **Predict load - store**

  Start with Naive but learn from mistakes

  Based on the history of mispeculations

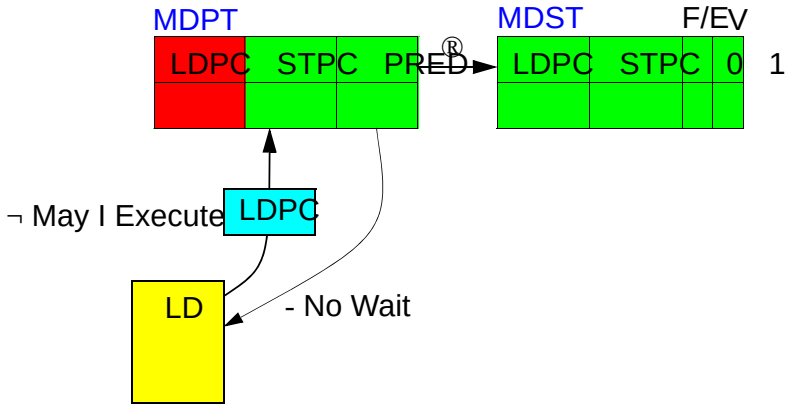
- Part **(3)**:

  Dynamically assigned synchronization variables

Synchronization - Load Waits

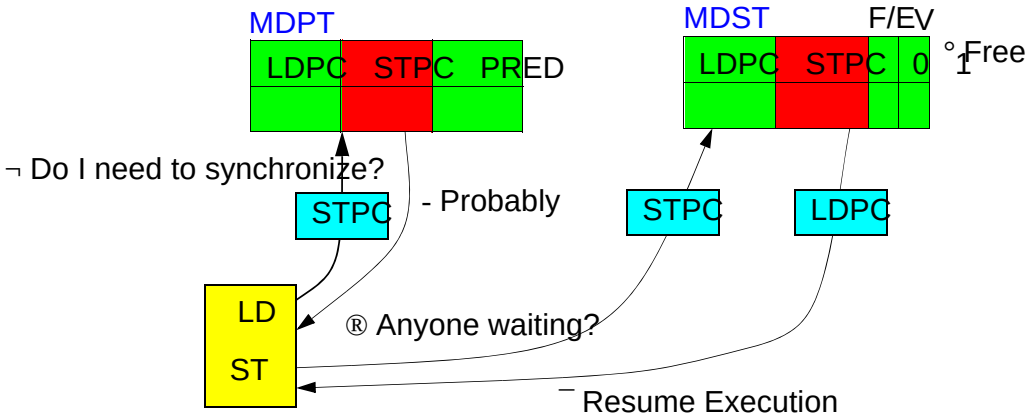
- Provide a small pool of full/empty bits
- Use *(LD PC, ST PC)* to associate entries w/ dependences

Memory Dependence Synchronization Table

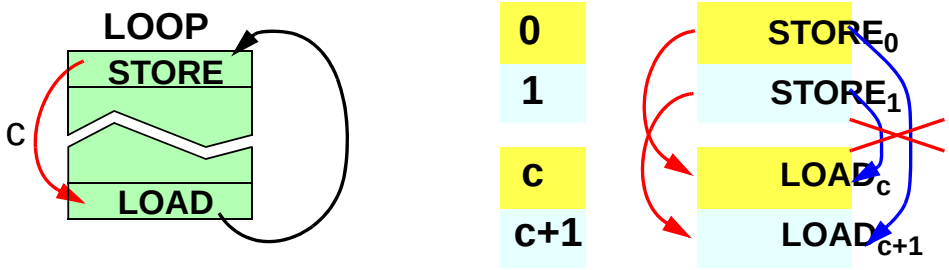


Synchronization - Load Resumes

Memory Dependence Synchronization Table



Multiple Instances of the Same Dependence



1 **Identification: (Load PC, Store PC) not enough**

In addition:
- (1). Data Address, or
- (2). Dependence Distance

Analogous to static linear recurrence

2 **May Need:**

Multiple synchronization entries per dependence

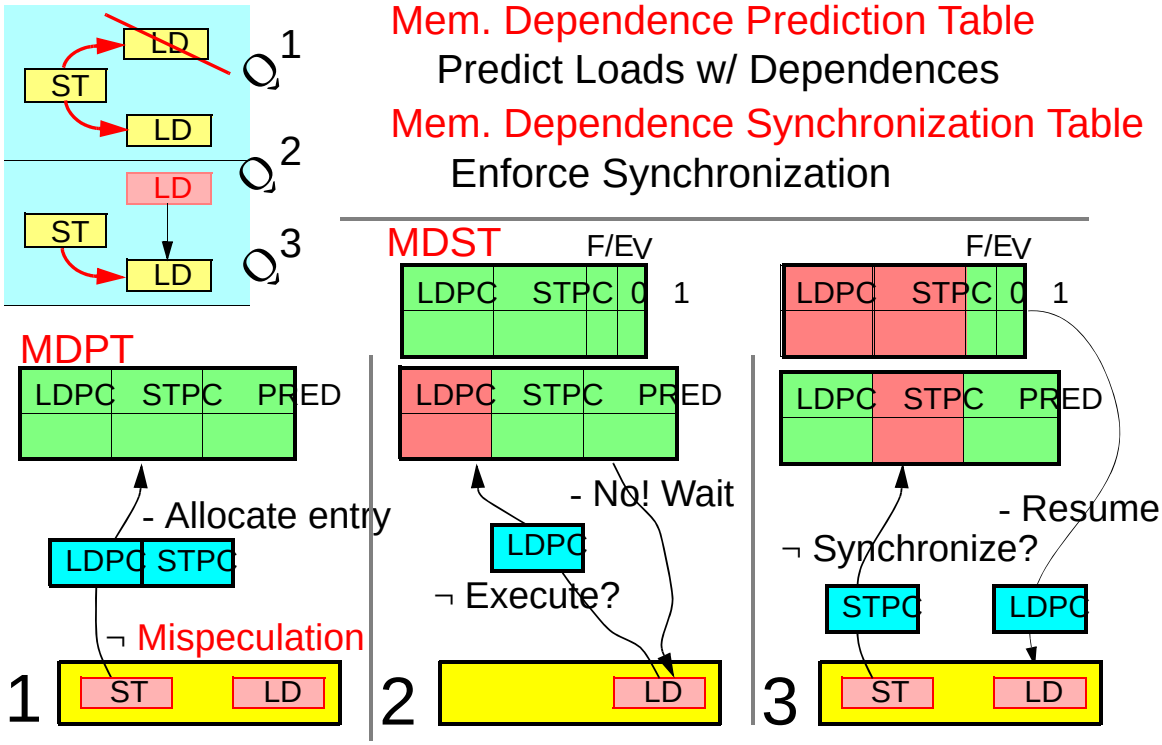
Dependence Speculation/Synchronization

- Other alternatives exist for both prediction and synchronization.
- Simplifications may be possible.

For example:

- Use PC to identify only loads
- Use the data address to indirectly identify the stores and to synchronize

How it works



Mechanism Models

- **Multiscalar - 3 models:**
  1. Merged MDPT/MDST, allows for adaptivity
     concerns: centralized & multiple deps. per load
     use more as an indicator of potential
  2. Merged MDPT/MDST, fixed #stores per load
  3. Split, Level of indirection for multiple dependences per load

- **Superscalar - 1 model:**
  Level of indirection for multiple dependences
  Synchronization using the register scheduler

Evaluation - Roadmap

1. Multiscalar - Split, Distributed Window

- Review:
  Naive Speculation / Potential
  Address-Based information
  Selective Speculation
- Evaluation of Speculation/Synchronization
  1. Prediction Accuracy
  2. Performance

2. Superscalar - Continuous, Centralized window

Multiscalar - Result Review

- Naive Speculation is a win, more so as window increases

  4-stages: ~30% int, ~110% fp

  8-stages: ~50% int, ~280% fp

- Potential over Naive (oracle):

  8-stages: ~31% int, ~17% fp

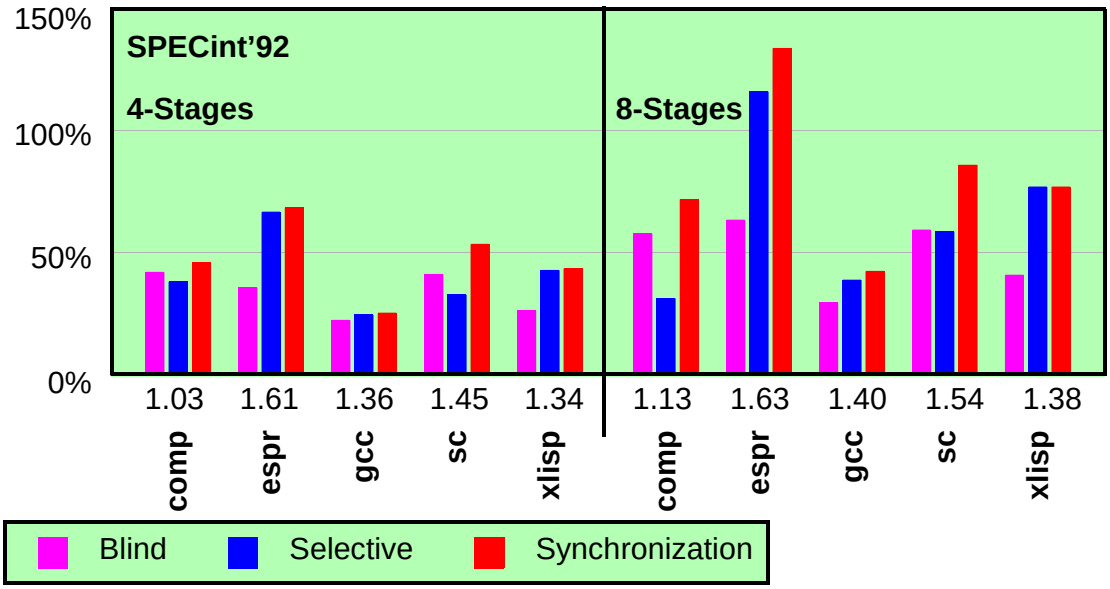
- Exposing Store addresses helps only slightly

  8-stages: ~9% int, ~3% fp

- Selective Speculation not robust

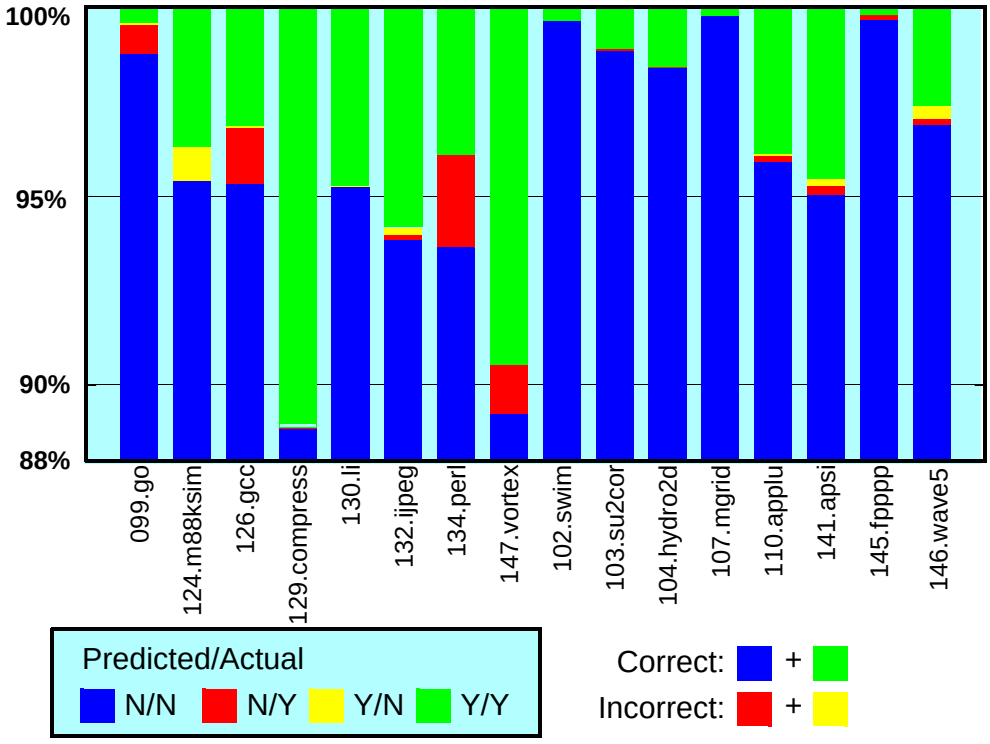
  8-stages: slowdowns as much as 45%

Comparison of Speculation Policies

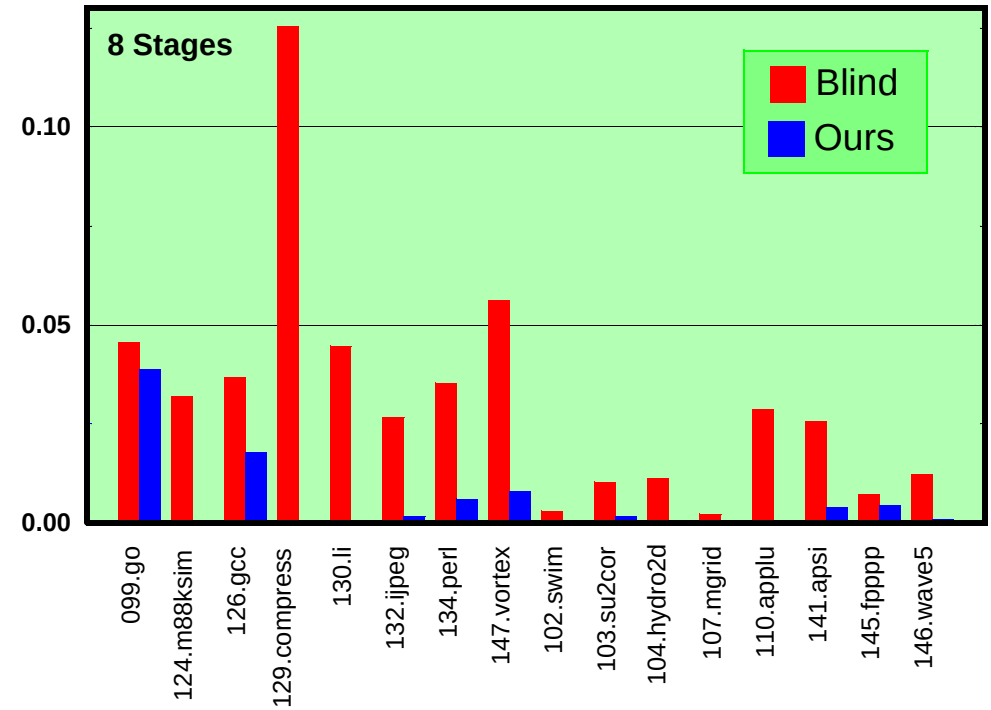


- Speedups are relative to no speculation (IPC along X axis)
- Perfect dependence prediction is used

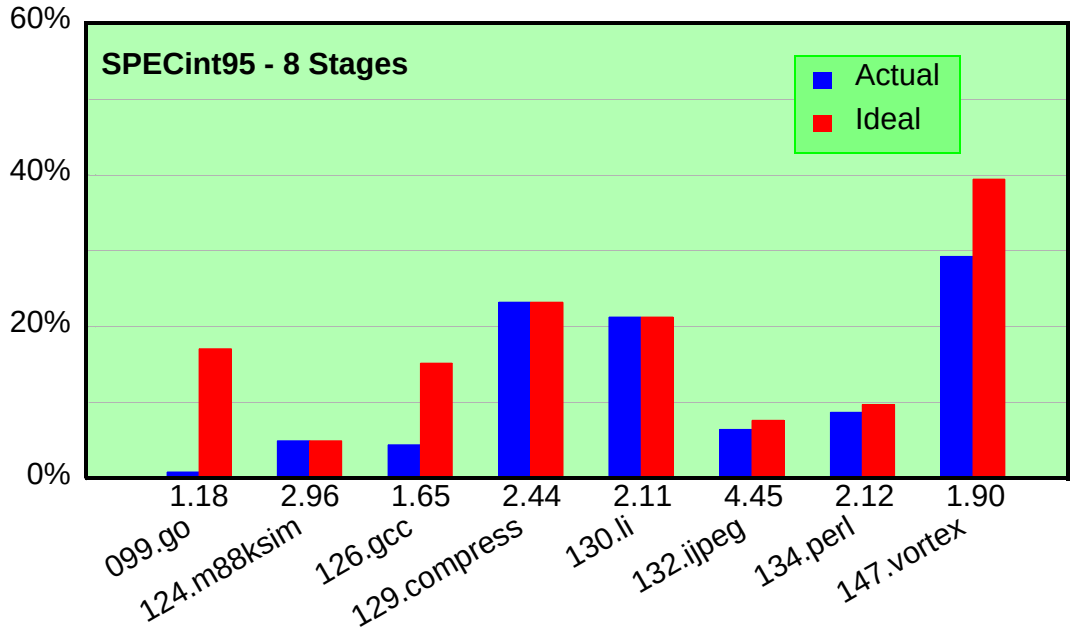
Dependence Prediction Accuracy



Mis-speculation Rates

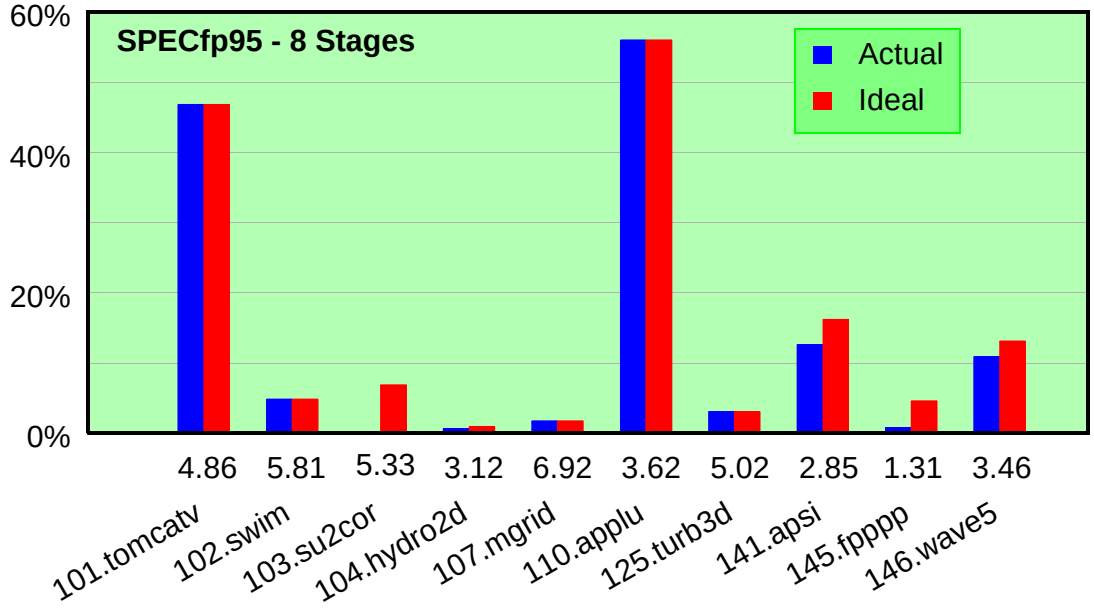


Speedup - SPECint95



- Speedups are relative to blind speculation
- IPC w/ our mechanism

Speedup - SPECfp95



- Speedups are relative to blind speculation
- IPC w/ our mechanism

Superscalar: Key Results

- **Naive memory speculation very close to ideal speculation**

  if loads can inspect store addresses before going to memory

  &

  this does not impact load latency

  Address Scheduler: Complexity & Cost?

- **Memory Dependence Speculation/Synchronization for:**

  1. **Lower Complexity**
  2. **High Performance**

Re-Scheduling Loads on-the-fly: Design Space

- **Address Scheduler**

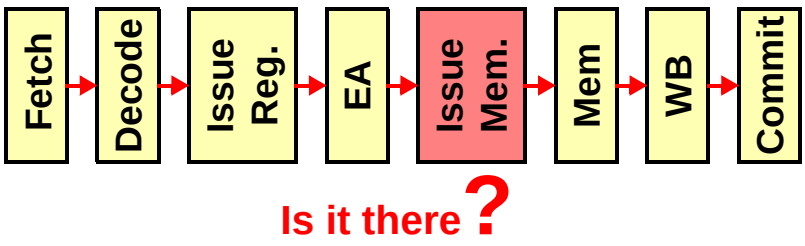
  Can loads see preceding store addresses before going to memory?

- **Why not?** Additional Scheduler is needed!

  Complexity & Latency implications

- **Similar to register scheduler (window), but:**

  e.g., Large address fields & Out-of-order insertion



Memory Dependence Speculation

- **No Speculation**:

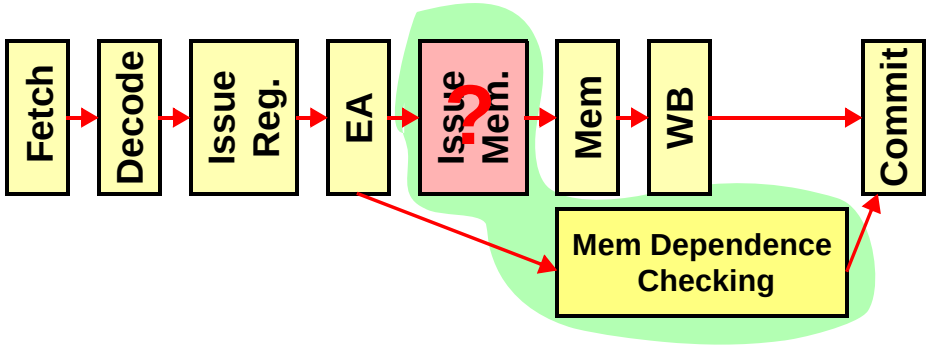
  load executes only when it is certain that no dependence will be violated

- **Speculation:**

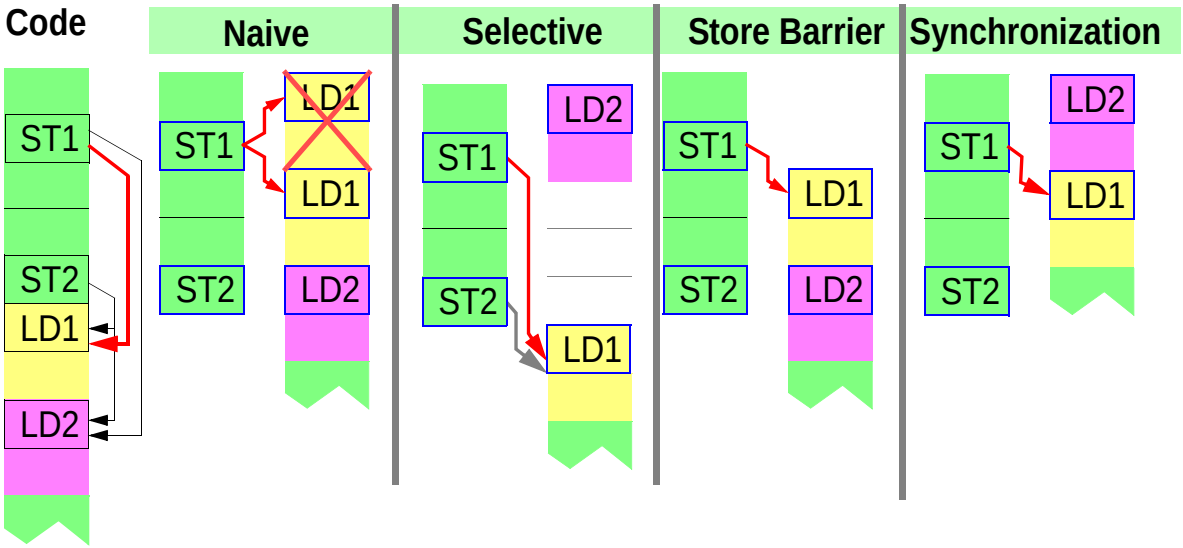
  load may execute before a preceding store

  a dependence may be violated

  check for violations at a later time (possibly not in critical path)



Speculation Policies



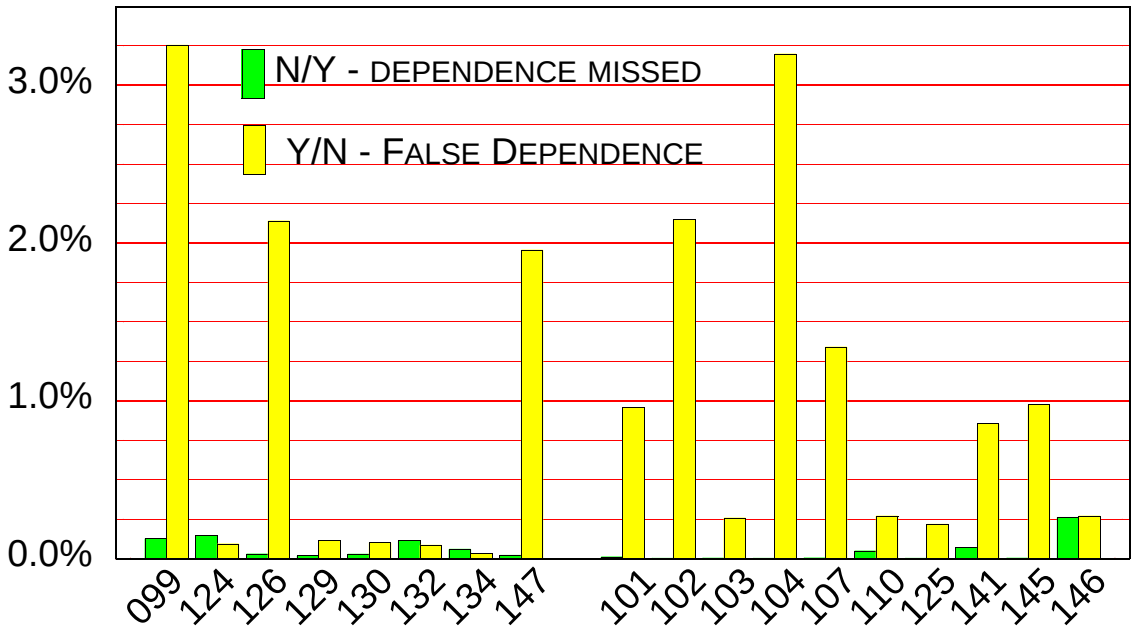
Speculation Policies: Tradeoffs

- **Naive:**
  - May execute loads too early
  - Suffers from misspeculations
- **Selective**
  - May delay a load more than it is necessary
  - Dependences often among distant loads and stores
- **Store Barrier**
  - Delays dependent loads only as long as it is necessary
  - Delays unrelated loads too
- **Synchronization**
  - Delays loads only as long as it is necessary

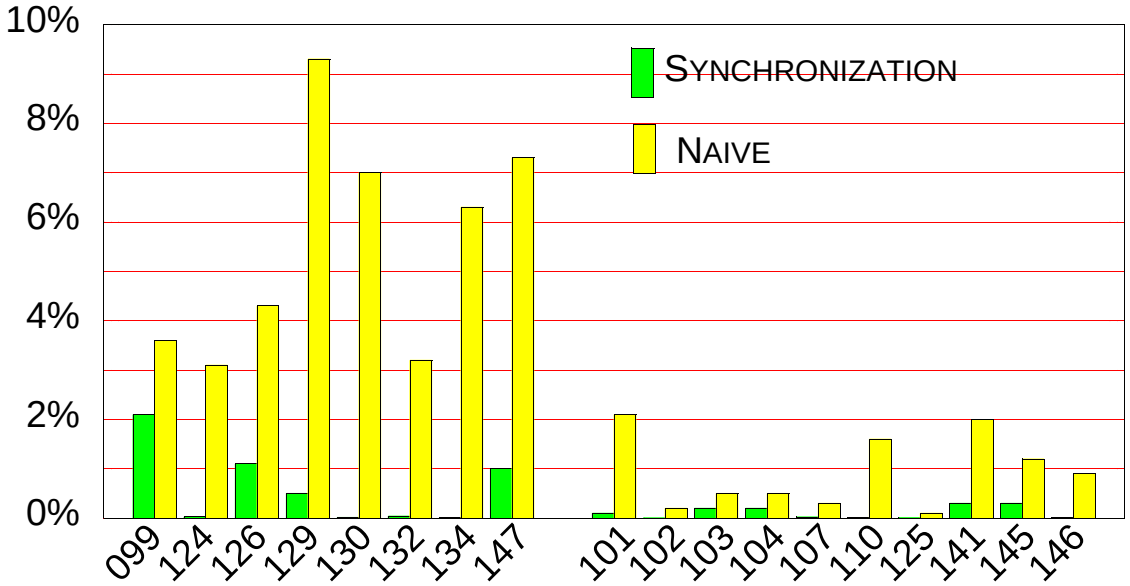
Evaluation

- **w/ Address Scheduler (AS)**
  - Speculation a win
  - NAIVE Speculation as good as it gets (ORACLE)
    - no need for other speculation policies
  - But! Performance drops w/ scheduler latency
- **w/o Scheduler (NAS)**
  - Speculation a must
  - Can do a lot better than NAIVE
  - SELECTIVE and STORE BARRIER not robust
  - SYNCHRONIZATION close to as good as it gets (ORACLE)

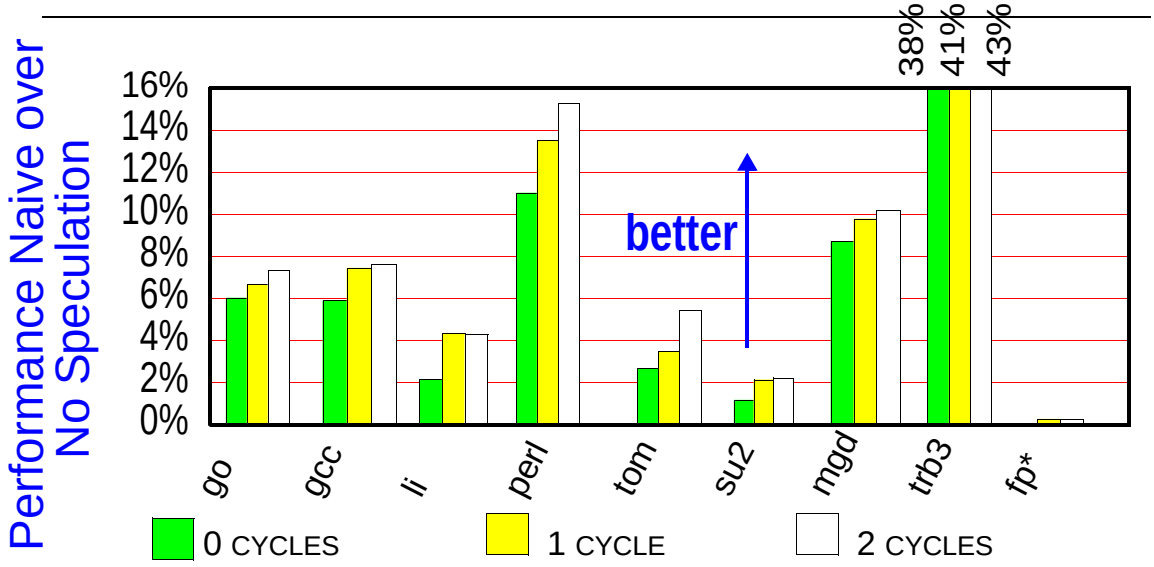
Dependence Status Prediction - Loads



Mispeculation Rates



AS: Naive vs. No Speculation

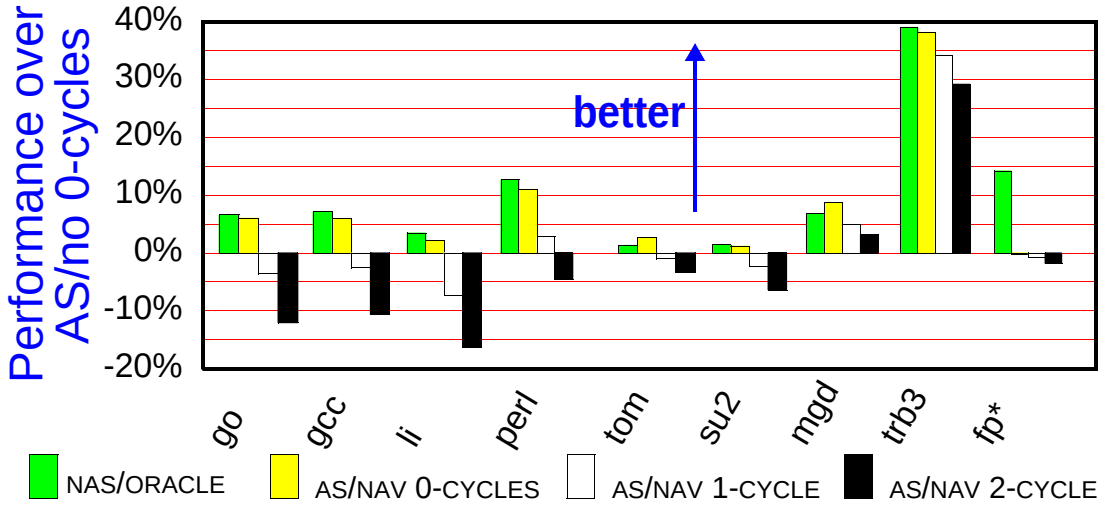


- **NAIVE a win over no speculation (in most cases)**
- **Gains increase w/ scheduler latency**
- **No misspeculations observed**

AS/Naive vs. NAS/Oracle

NAS/Oracle: No scheduler (i.e., no latency for loads)

Perfect knowledge of all memory dependences



AS/Naive 0-cycles: as good as it gets

Potential to do much better when AS latency is > 0

Approximating NAS/Oracle: NAS/Naive

- **Naive offers some of the performance potential**

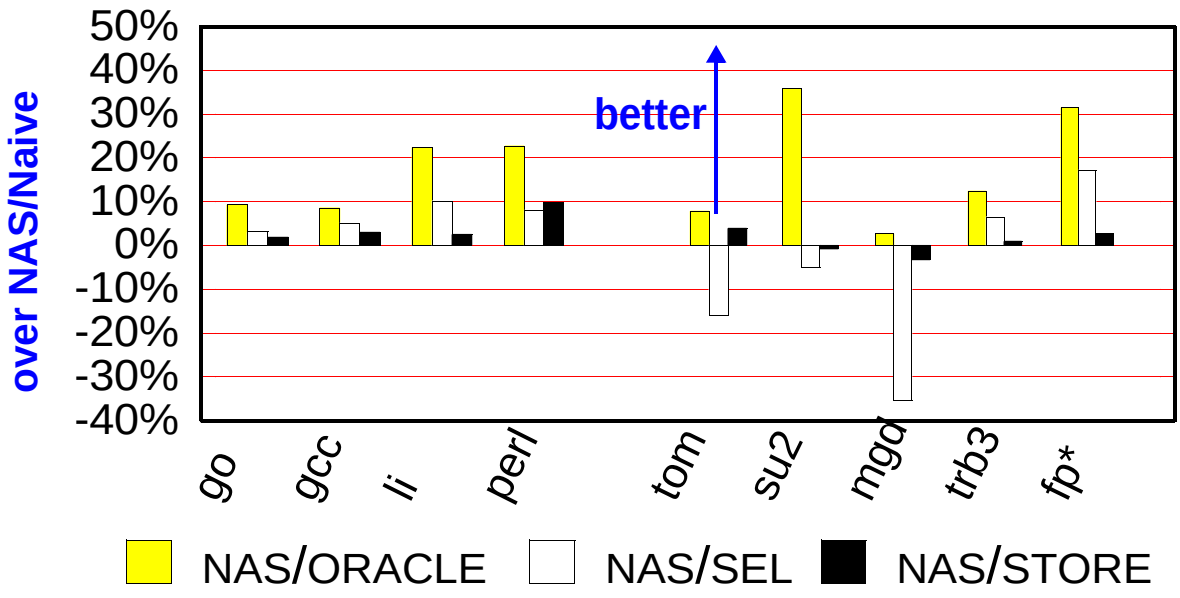
Speedups over No Speculation:

Oracle: ~65%, 30% (int) and 113% (fp)

Naive: ~20%, 21% (int) and 20% (fp)

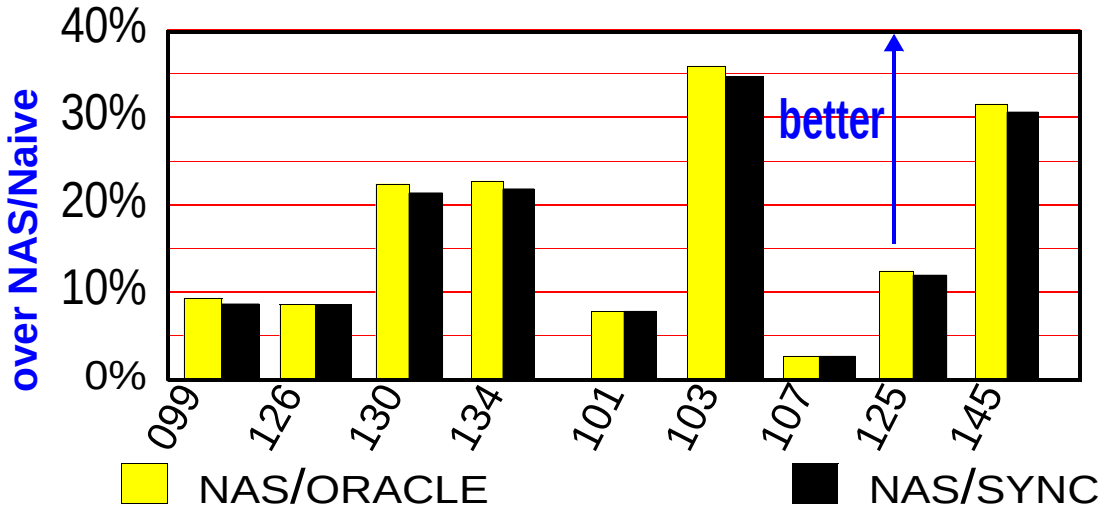
Significant room for improvement w/ other speculation methods

Selective and Store Barrier Speculation



Neither is robust: Naive sometimes better

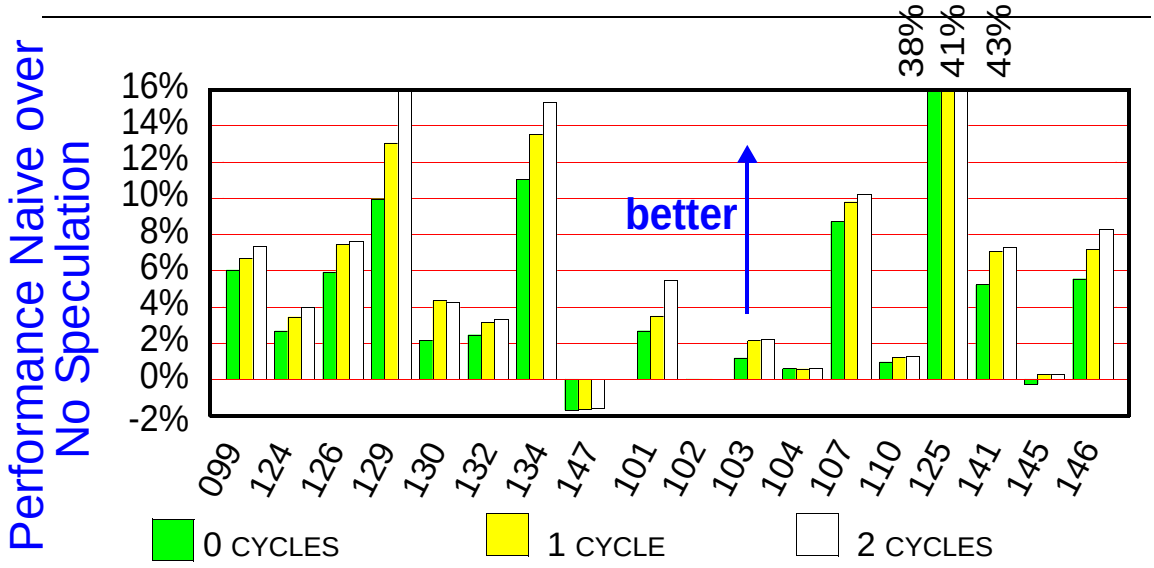
Speculation/Synchronization



- 4k-entry memory dependence predictor

Robust, performance close to Oracle Speculation

AS: Naive vs. No Speculation

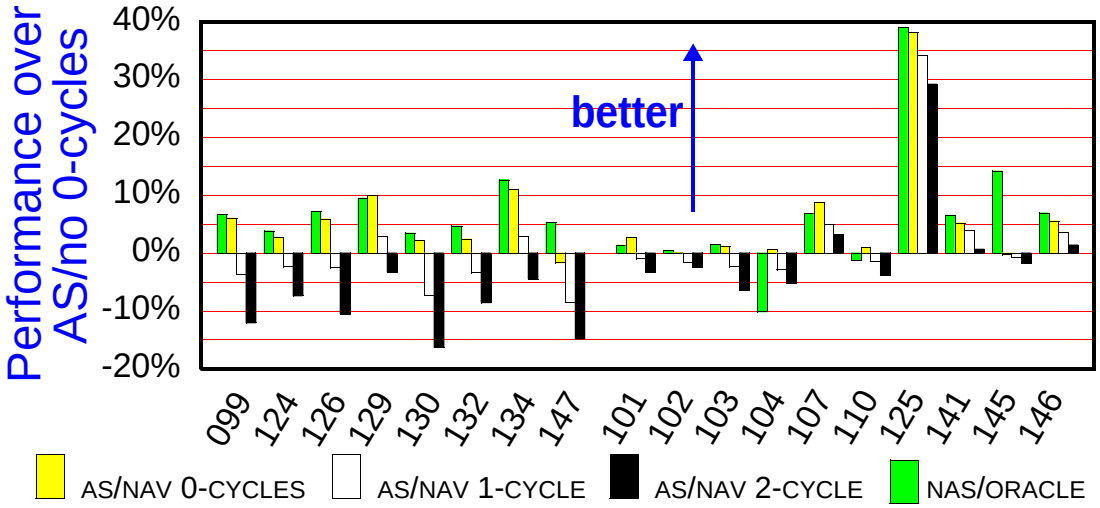


- **NAIVE a win over no speculation (in most cases)**
- **Gains increase w/ scheduler latency**
- **No misspeculations observed**

AS/Naive vs. NAS/Oracle

NAS/Oracle: No scheduler (i.e., no latency for loads)

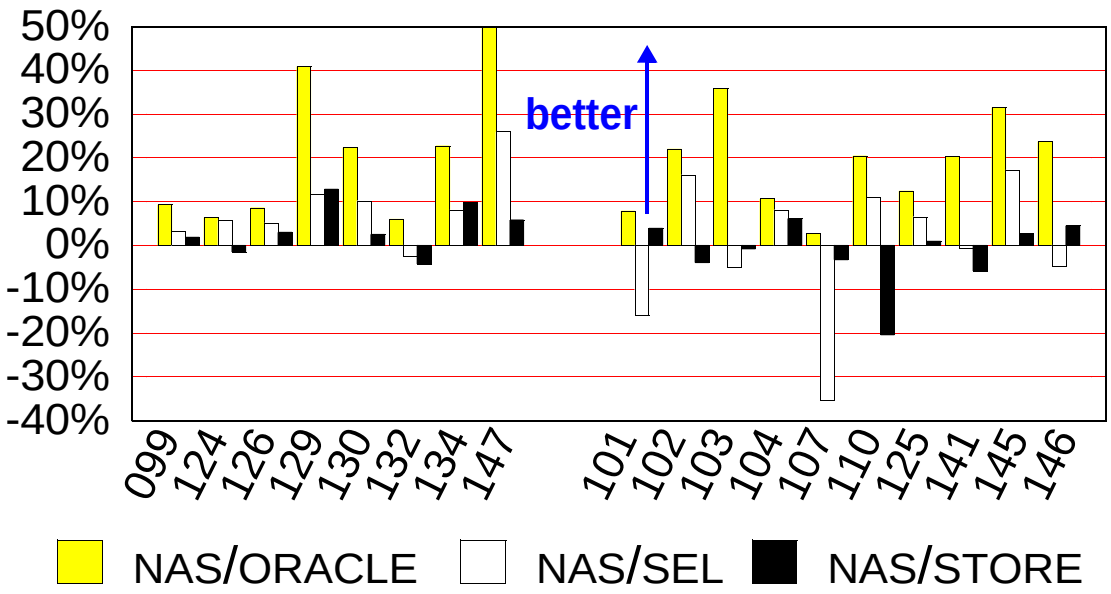
Perfect knowledge of all memory dependences



AS/Naive 0-cycles: as good as it gets

Potential to do much better when AS latency is > 0

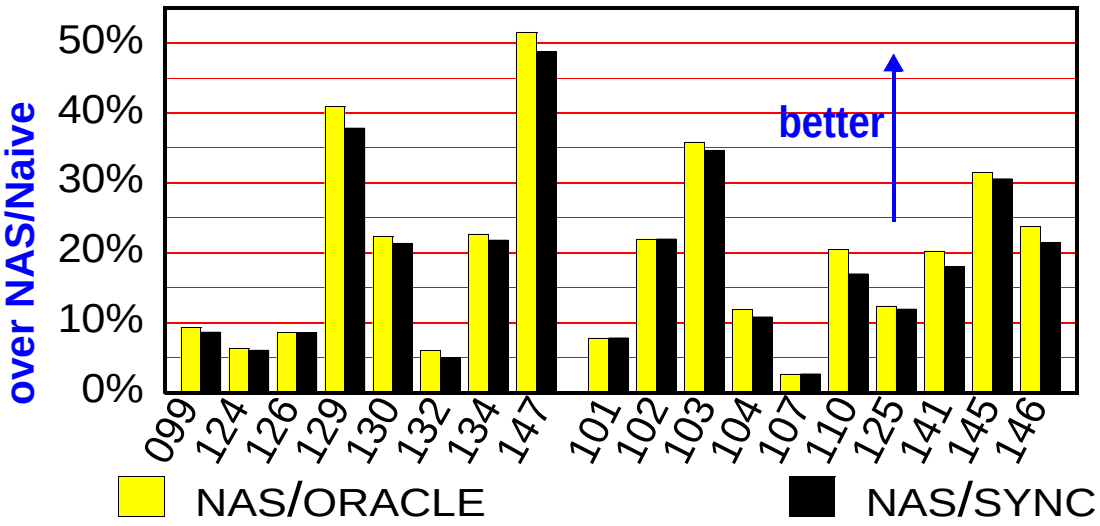
Selective and Store Barrier Speculation



- Performance relative to NAS/Naive

Neither is robust: Naive is often better

Speculation/Synchronization



- 4k-entry memory dependence predictor

Robust, performance close to Oracle Speculation

Summary

- **W/ AS**
  - Speculation is a win
  - Naive speculation as good as it gets
  - No need for other speculation policies
  - But AS may impact latency
  - Performance degrades w/ scheduling latency
  - Could do better if dependence were known in advance
- **W/O an AS**
  - Naive much better than no speculation
  - But lots to be gained over naive
  - Selective or Barrier not robust, often worse than naive
  - Speculation/Synchronization very close to ideal