Memory Dependence Prediction

Andreas Moshovos
moshovos@ece.nwu.edu
www.ece.nwu.edu/~moshovos

Electrical And Computer Engineering Department
Northwestern University

ECE Dept., Purdue University, Sept. 9, 1999
Exploiting Program Behavior

Building faster/better machines:

1. Faster Circuits -> Faster Execution
2. Be “smarter” about program execution:

   Exploit Idiosyncrasies in Program Behavior

Example? Caches

   Idiosyncrasies in access pattern
   Idea: Use small/fast storage for recently accessed data.

What to do next? One Possibility is:

Identify/Exploit Other Idiosyncrasies in *Typical* Program Behavior

This might be the right time! Technology and Existing Knowledge.
Memory Dependences are Quite Regular

• Identified a new form of regularity:

Memory Dependence Locality

1. Load/Store has a Dependence?
2. Which Dependence a Load/Store has?
   
   \[(\text{load}_{PC}, \text{store}_{PC})\]
   
   for i

   \[a[i] = a[i - 1] + 1\]

   store

   load

   store

   load

   store

   load

Time

Opportunity to Exploit this Regularity
How to exploit? Need Techniques
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:
1. Exploit Load/Store Parallelism
2. Reduce Memory Latency
3. Memory Bandwidth Amplification

GOAL
Memory is a major and ever increasing bottleneck
Roadmap

• Exploiting Load/Store Parallelism
  - Two directions for improving memory performance
  - Load/Store Parallelism
  - Speculation/Synchronization
  - Performance

• Speculative Memory Cloaking and Bypassing

• Transient Value Cache

• Other Applications

• Future Directions: Slice Prediction & Slice-Processors
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?
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

$$\begin{align*}
A \\
\text{load}
\end{align*}$$

$$\begin{align*}
r1 &= r2 + 10 \\
\text{load} \ldots, [r1 + 5]
\end{align*}$$

$$\begin{align*}
r1 &= r2 + 10 \\
\text{load} \ldots, [r3 + 5]
\end{align*}$$

Valid Execution Order

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:

Step #1: Grab a chunk of instructions

Step #2: Find the dependences

Step #3: Load, execute

Use Parallelism to Tolerate Memory Latency

Instruction Window

Inspect Names
Use Of Addresses Hinders Parallelism

→ Limits Us in Our Effort to Tolerate Memory Latency
Memory Dependence Prediction

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

**Naive Memory Dependence Speculation**

- **Instructions**
  - Store
  - Load

**Timeline**

- **Guess no**: Load executes
- **Guess yes**: Load re-executes if there is a dependence

**Need to Balance**: Gain vs. Penalty

---

A. Moshovos

Permission to use these slides is granted provided that a reference to their origin is included.
Dependence Speculation and Performance

Program Order

<table>
<thead>
<tr>
<th></th>
<th>store</th>
<th>load</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>free</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>free</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>store</td>
<td>load</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No Speculation

Speculation

No Dependence

Dependence

Speculation may affect performance either way

Balance: Gain vs. Penalty

Penalty:

(a) work thrown away
(b) opportunity cost
How About Future Systems?

Common Case:

Today

Soon

- Memory will be slower
  - 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
Speculation/Synchronization

Learn from mistakes:

1. Start with Naive Speculation

2. Remember misspeculations and predict next time the same will happen

   - Q1? Which loads should wait
   - Q2? How long

Best performing scheme

- A1: Loads w/ dependences
- A2: Synchronize w/ appropriate store
How it works

Mem. Dependence Prediction Table
Predict Loads w/ Dependences
Mem. Dependence Synchronization Table
Enforce Synchronization

How it works

Mem. Dependence Prediction Table
Predict Loads w/ Dependences
Mem. Dependence Synchronization Table
Enforce Synchronization

A. Moshovos

Permission to use these slides is granted provided that a reference to their origin is included.
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

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

- **ORACLE**
- **CENTRALIZED**
- **2 STORES/LOAD**
- **SYNONYMS**

256-Entries

**Better**
Evaluation - Superscalar

1. Can loads inspect Store addresses before obtaining a value?
   *Requires an address-based load/store scheduler (ABS)*

2. Is speculation used?

   • Speculation a win (naive over no-speculation):
     
     No ABS: ~29% int, ~113% fp
     
     *misspeculations are an issue*

     ABS: 4.6% int, 5.3% fp (0-cycle scheduling latency)
     
     *virtually no mispeculations*

   • Next:
     
     1. Compare “Oracle + no ABS” with “naive + ABS”
     2. Speculation/Synchronization on “no Abs” close to Oracle

   Speculation/Synchronization as an Alternative to ABS
“Oracle + no ABS” vs. “Naive + ABS”

Base case: “ABS + no-speculation”

“Oracle + no ABS” close to “ABS + naive”

Potential for Spec./Sync.

A. Moshovos

Permission to use these slides is granted provided that a reference to their origin is included.
Speculation/Synchronization

On the average within 0.93% (int) and 1.001% (fp) of oracle
Summary

- Improving Processor/Memory Performance:
  move loads up in time

- Addresses Hinder Parallelism

- Memory Dependence Speculation/Synchronization
  Predict dependences and move loads up in time
  Classify loads in two categories:
  1. No Dependence: Execute at will
  2. Dependence: Synchronize with specific store
Related Work

• In the context of *Dynamically Scheduled Processors*


**Store Barrier**

2. G. Chrysos, J. Emer (DEC), Superscalar Env., ISCA ‘98

**This work:** UW Tech. Report, March ‘96, ISCA ‘97
A. Moshovos, S. Breach, T. N. Vijaykumar, G. Sohi

• In the context of *Statically Scheduled Processors*

1. Static Disambiguation

2. Speculation in Software

• Dynamic Compilation

3. Software/Hardware Hybrids
Roadmap

• Exploiting Load/Store Parallelism

• Speculative Memory Cloaking and Bypassing
  - Memory as a communication mechanism
  - Speculative Memory Cloaking
  - Other uses of Memory
  - Evaluation

• Transient Value Cache

• Other Applications

• Future Directions: Operation Prediction & Slice-Processors
Memory can be an inter-operation communication mechanism

**Communication Specification:** implicit via Addresses

*Is this the best way in terms of performance?*
Memory Communication Specification - Limitations

**Implicit**

1. Calculate address
2. Establish Dependence

**Explicit**

Store - Load: Direct Link
No Delays

---

Timeline

```
Instructions:  
| store | store 2 | load |
```

```
Memory: 
| store addr | value | load addr | store2 addr |
```

---

A. Moshovos

Permission to use these slides is granted provided that a reference to their origin is included.
Predicting Memory Dependences

1. Build Dependence history: Dependence Detection Table

- **Record:** \((store \ PC, \ address)\)
- **Loads:** \((load \ PC, \ address)\)

\[ \rightarrow (store \ PC, load \ PC) \]

2. Use history to predict forthcoming dependences
   - assign synonyms to detected dependences
   - use synonym to locate value
Dynamically & Transparently convert implicit into explicit

- **Dependence prediction** → direct store-load or load-load links
- Speculative and has to be *eventually* verified

### Speculative Memory Cloaking

#### Dependence Prediction

- **load PC** synonym
- **store PC** synonym

#### Synonym File

- value f/e

#### Timeline

1. **store**
2. **value**
3. **addr**
4. **load**

#### Traditional Memory Hierarchy

- **verify**
- **address**
- **speculative value**
- **store PC** synonym

Permission to use these slides is granted provided that a reference to their origin is included.
Observe: Store and Load are used to just pass values

• Extents over multiple store-load dependences
• DEF and USE **must co-exist** in the instruction window

Takes load-store or loads off the communication path
Speculative Memory Cloaking/Bypassing

Address-based Memory as: Storage vs. Interface

Ask:
1. What is memory used for?
2. How addresses impact the action?

Inter-operation Communication

Dynamically Create Direct Links Between Producers/Consumers

Data-Sharing
Most Dependences Correctly Predicted
Cloaking - Mispeculation Rates

- Relatively low mispeculation rates
- Causes:

Unstable Dependences - Recursive Functions
Performance - Selective Invalidation

0% 2% 4% 6% 8% 10% 12% 14%

Oracle  Selective

A. Moshovos

Permission to use these slides is granted provided that a reference to their origin is included.
Summary

• Address-based memory → unnecessary delays

• Used Memory Dependence Prediction to:

  1. Improve Store-Load and Load-Load Communication
     address-based → dependence-based

  2. Take Store-Load off the Communication path

     Treat Program as a specification of what to do
     Not as also of how to do it.
Increasing Memory Bandwidth

• Parallelism $\Rightarrow$ more bandwidth $\Rightarrow$ more L1 ports

![Diagram showing L1 cache and CPU with multi-port connections]

• A very small cache is easier to multi-port

![Diagram showing L0 cache and L1 cache with multi-port connections]

• But, it increases the latency for all accesses that miss in it
Transient Value Cache

- Support for multiple/simultaneous Load/Store Requests

Observe:
A. Often RAW or RAR exists with recent store or load
B. Many recent stores are killed

+ Small cache can service these
- Latency for other loads will increase

C. A & B / Dependence Status is predictable

L1 DCache Bandwidth/Port Requirements are Reduced
Few Loads Observe a Latency Increase
TVC vs. L0 - Hit Rates

Permission to use these slides is granted provided that a reference to their origin is included.
TVC vs. L0 - “Miss” Rates

A. Moshovos

Permission to use these slides is granted provided that a reference to their origin is included.
Other Applications

- Transient Value Cache:
  Bandwidth amplification
- Prefetching of recursive data structures
  Load-to-Load-EAC dependences
- Virtual Function Call Precomputation
  Surgical extension to previous one
Current Research

• Cloaking for Multiprocessors
  - Prediction in coherence protocol

• Slice Prediction & Slice Processors
  - Use the Program to Predict its actions
  Identify Performance critical computation slices
  Run them speculatively in advance
  Annotate program with predicability information

• Embedded Processor Optimizations

• Multi-media application evolution

• Self-micro-reconfigurable processors
  - Run-time extraction of convertible slices
  - Interfaces with OOO
Future Directions

• Prediction is becoming prevalent
  - Branch Prediction
  - Value Prediction
  - Dependence Prediction
  - Coherence Prediction

Thus Far:
Perform Computation as Specified but as fast as possible

Moving toward:
A predict-and-verify model of computation

Q1? How a processor should look like then?
Q2? How to predict accurately?
Current Predictors

• Treat Program as a black box
• Observe outcomes

• Guess: soon predictors of this kind will be very accurate
  say 90% or more
• How about the rest 10%?
  Amdahl’s law -> may dominate performance

• Slice prediction: execute SCOUT threads in parallel
  1. Predict performance critical computation slice
  2. Execute as far in advance as possible
Current Predictors

- Treat Program as a black box
- Build an *approx.* representation of program’s function

**“INPUT”**
- (PC1, taken)
- (PC2, not-taken)

**“OUTPUT”**
- (PC1, taken)

f()?

- Guess: soon predictors of this kind will be very accurate
  say 90% or more
- How about the rest 10%?

Amdahl’s law -> may dominate performance
Slice Processors

- But program itself is a representation of f()!
- USE PROGRAM TO PREDICT IT’S ACTIONS

while (list)
    if (list->data == KEY)
        foo₀(list)
    else...
        foo₁(list)
    list = list->next

Time

slices

predict

list = list->next

list->data == KEY

then branch

else branch