Streamlining
Inter-Operation Memory
Communication
via Data Dependence Prediction

Andreas Moshovos and Guri Sohi
{moshovos, sohi}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison
In Addition to addresses

Communication-Conscious Approach

Exposé the communication
Observe and exploit its behavior

This work: How to use to: 1. ↓ Latency, 2. ↑ Bandwidth
Communication-Conscious Techniques

Communicating via addresses $\rightarrow$ inherent delay

Observe:
- Many loads get their value from a recent store
- These dependences are predictable

1. Speculative Memory Cloaking
- Prediction: link \textit{load} - \textit{store}
- pass value
- verify through memory

2. Speculative Memory Bypassing
- link \textit{DEF} - \textit{USE}

Communication Latency is Reduced
Communication-Conscious Techniques

DCache ports are becoming expensive

Observe:
- A. Recent stores feed many loads
- B. Many recent stores are killed

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

C. A & B / Dependence Status is predictable

3. Transient Value Cache

L1 DCache Bandwidth/Port Requirements are Reduced
Roadmap

- Introduction

- Traditional Memory Communication Specification- Limitations

- Speculative Memory Cloaking

- Speculative Memory Bypassing

- Transient Value Cache

- Evaluation

- Summary
Memory Communication Specification - Limitations

**Implicit**

**Delays:**
1. Calculate address
2. Establish Dependence

**Explicit**

Store - Load: Direct Link
No Delays

Timeline
- store
- value
- store addr
- load
- load addr
- store2 addr

Program Order
- store
- store 2
- load

Instructions
- value

Memory
Speculative Memory Cloaking

Dynamically & Transparently convert implicit into explicit

- Dependence prediction \(\rightarrow\) direct load-store link: synonym
- Speculative and has to be **eventually** verified

**Dependence Prediction**

<table>
<thead>
<tr>
<th>store PC</th>
<th>load PC</th>
<th>synonym</th>
</tr>
</thead>
</table>

**Synonym File**

<table>
<thead>
<tr>
<th>value f/e</th>
</tr>
</thead>
<tbody>
<tr>
<td>value f/e</td>
</tr>
</tbody>
</table>

**Timeline**

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

**Traditional Memory Hierarchy**

verify address address

speculative value

\[\text{address}\]

\[\text{address}\]
Predicting True Dependences

1. Detect Dependences / Build History

Record: \((\text{store PC, address})\)

Loads: \((\text{load PC, address})\) \rightarrow \((\text{store PC, load PC})\)

- not in critical path
- \#stores \ll \#instructions

Recording last 256 stores captures \textasciitilde 50\% of all load deps.
Cloaking - Issues & Implementation

See paper for:

- **Predicting Dependences / Synonym Generation**
  - 1-to-1 and n-to-m dependences
  - Dependences w/ distance > 1
  - Multiple synonym instances

- **Data Types / Sign-extension**

- **Sample Implementation**
  Simple, table-like structures:
  - Detection
  - Prediction
  - Synonym File
Speculative Memory Bypassing

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 off the communication path
Observations:
Many loads get their value from a recent store
Many stored values are quickly killed

A 256-FA word cache can service > 50% of loads, 60% of stores

**Hit:** No need to consume L1 ports

**Miss:** Latency increases
Streamlining Inter-oper ation Memory Communication Via Data Dependence Prediction

A. Moshovos

Transient Value Cache

The best of both worlds?
Data Dependence Status Prediction to steer

**load:** reads a value from a recent store? *true dep.*
**store:** will be killed by a closeby store? *output dep.*

Dependence: In Series

No Dependence: In Parallel

---

**Dependence: In Series**

- **L1**
- **load miss**
- **store**
- **TVC**
- **load**

**No Dependence: In Parallel**

- **L1**
- **store**
- **TVC**
- **load**
Evaluation

- True Dependence Status prediction
  Why: • 1st step in cloaking & bypassing
  • loads hidden by TVC

- Cloaking Accuracy
  Why: • Loads % that benefit from cloaking
  • Mis-speculated

- TVC low bound on reduction in accesses

See Paper for:

- Cloaking: Accuracy vs. Prediction table size, History kept
- Output Dependence Status Prediction
- Impact on Performance
True Dependence Status Prediction Accuracy

2k-entry 2-bit saturating counters

% Dynamic Loads

<table>
<thead>
<tr>
<th>Program</th>
<th>Correct</th>
<th>Wrong</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td>Y/Y</td>
<td>N/N</td>
</tr>
<tr>
<td>m88ksim</td>
<td>Y/Y</td>
<td>N/N</td>
</tr>
<tr>
<td>gcc</td>
<td>Y/Y</td>
<td>N/N</td>
</tr>
<tr>
<td>compress</td>
<td>Y/Y</td>
<td>N/N</td>
</tr>
<tr>
<td>xlisp</td>
<td>Y/N</td>
<td>Y/N</td>
</tr>
<tr>
<td>jpeg</td>
<td>N/Y</td>
<td>Y/N</td>
</tr>
<tr>
<td>perl</td>
<td>N/N</td>
<td>Y/N</td>
</tr>
<tr>
<td>vortex</td>
<td>N/N</td>
<td>Y/N</td>
</tr>
</tbody>
</table>

Pred/Actual

Correct
- Y/Y can benefit
- N/N can’t benefit

Wrong
- N/Y missed opportunity
- Y/N harm
TVC - Reduction in Accesses

![Bar Chart]

- go
- m88ksim
- gcc
- compress
- xlisp
- ijpeg
- perl
- vortex

**Loads**

**Stores**
Cloaking - Dynamic Loads Serviced

First-cut implementation

2k-entry 2-bit saturating counters

<table>
<thead>
<tr>
<th>Program</th>
<th>Correct</th>
<th>Mis-speculated</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td>75%</td>
<td>25%</td>
</tr>
<tr>
<td>m88ksim</td>
<td>96%</td>
<td>4%</td>
</tr>
<tr>
<td>gcc</td>
<td>86%</td>
<td>14%</td>
</tr>
<tr>
<td>compress</td>
<td>91%</td>
<td>9%</td>
</tr>
<tr>
<td>xlisp</td>
<td>62%</td>
<td>38%</td>
</tr>
<tr>
<td>ijpeg</td>
<td>92%</td>
<td>8%</td>
</tr>
<tr>
<td>perl</td>
<td>95%</td>
<td>5%</td>
</tr>
<tr>
<td>vortex</td>
<td>78%</td>
<td>22%</td>
</tr>
</tbody>
</table>

A. Moshovos

Streamlining Inter-operation Memory Communication Via Data Dependence Prediction
Summary

1. Many loads get their value from a recent store
2. Many stored values are quickly killed

- **Speculative Memory Cloaking** - *Latency*
  Convert *implicit* into *explicit*
  Pass values using just the PC

- **Speculative Memory Bypassing** - *Latency*
  Take load-store off the communication path

- **Transient Value Cache** - *Bandwidth*
  Selective redirection of loads/stores
  Reduces bandwidth (port) requirements

Pssst... I’m hoping to graduate by Fall ‘98
True/Output Dependence Status Prediction

**Loads / True**

**Stores / Output**

Predicted/Actual

<table>
<thead>
<tr>
<th></th>
<th>Y/Y</th>
<th>N/N</th>
<th>N/Y</th>
<th>Y/N</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>m88ksim</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>gcc</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>compress</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>xlisp</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jpeg</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>perl</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>vortex</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Cloaking Accuracy

% of loads w/ dep.

- go
- m88ksim
- gcc
- compress
- xlisp
- jpeg
- perl
- vortex

512, 1K, 2K, 4K
Infinite
TVC - Reduction in Accesses

Pessimistic model: last 256 stores - not last 256 addresses
Streamlining Inter-operation Memory Communication Via Data Dependence Prediction

A. Moshovos

Why “Cloaking”?  

**cloak**  

\[ n \ \overset{	ext{'klokb}}{\text{'klokb}} \]  

2 : *to alter so as to hide the character of*  

3 : *something that conceals*

“Speculative Memory Renaming”?  

• Already in use in the same context: ARB, LSQ (w/o Speculative)  

• **Re-name:** *change the name*  

  - associate address with a new name  
  - Legacy of “Register Renaming”:  
  - can go from address to new name  

  synonym and address are NEVER associated  
  can’t determine synonym from address  
  other accesses to the same address can’t locate synonym
An Implementation

Support Structures:

1. Dependence Detection Table DDT
2. Dependence Prediction and Naming Table DPNT
3. Synonym File SF

Example:

loop:
  t = AllocateToken()
  SetToken(t)
  ...
  ActOnToken(t)
  ...

SetToken(t):
  t->type = ...
  ...

ActOnToken(t):
  switch (t->type)
  ...

store
load
store
cloak
load

Example:
An implementation - Example

![Diagram showing memory communication via data dependence prediction.]

- **DPNT**
  - **PC**
  - **TAG**
  - **Pred V**

- **DDT**
  - **addr**
  - **PC**

- **STPC** tags
  - **addr1**
  - **load**

- **LDPC** tags
  - **addr1**
  - **store**
An implementation - Example

<table>
<thead>
<tr>
<th>PC</th>
<th>TAG</th>
<th>Pred</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>STPC</td>
<td>tag</td>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>LDPC</td>
<td>tag</td>
<td>Y</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>value</th>
<th>tag</th>
<th>f/e</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>?????</td>
<td>tag</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC</th>
<th>TAG</th>
<th>Pred</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>STPC</td>
<td>tag</td>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>LDPC</td>
<td>tag</td>
<td>Y</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>value</th>
<th>tag</th>
<th>f/e</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxxx</td>
<td>tag</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC</th>
<th>TAG</th>
<th>Pred</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>STPC</td>
<td>tag</td>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>LDPC</td>
<td>tag</td>
<td>Y</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>value</th>
<th>tag</th>
<th>f/e</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxxx</td>
<td>tag</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Streamlining Inter-operation Memory Communication Via Data Dependence Prediction

A. Moshovos

The “Memory Problem”

Store & Retrieve Values with:
1. Low Latency
2. High Bandwidth

Not all storage can be built this way → Intelligent Mechanisms

Name-Centric Approach
observe and exploit address stream behavior
Predicting Dependences - Synonym Generation

1-on-1 straightforward

\[ \begin{array}{c}
\text{store} \\
\downarrow \\
\text{load}
\end{array} \]

N-to-N is common

\[ \begin{array}{c}
\text{store} \\
\downarrow \\
\text{load} \\
\downarrow \\
\text{store} \\
\downarrow \\
\text{load}
\end{array} \]

Break into steps:

1. Predict dependence status (existence)
2. Figure out with who / synonym

dependencies w/ common parties same synonym

execution path determines which is the right one