Freeway: Maximizing MLP for Slice-Out-of-Order Execution

Rakesh Kumar, Mehdi Alipour, David Black-Schaffer
The Memory Bottleneck

Growing data sets

Memory is slow

Memory pressure is growing:
- Multiple cores
- Accelerators

Memory subsystem dictates the overall system performance
Overcoming the Bottleneck

Memory Level Parallelism (MLP) is effective in hiding memory latency
MLP in Out-of-Order Cores

**OoO Core**

Instruction Queue (CAM)

<table>
<thead>
<tr>
<th>I0</th>
<th>ld</th>
<th>r1=M[r7]</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>add</td>
<td>r2=r1+1</td>
</tr>
<tr>
<td>I2</td>
<td>add</td>
<td>r3=r3+1</td>
</tr>
<tr>
<td>I3</td>
<td>ld</td>
<td>r4=M[r3]</td>
</tr>
</tbody>
</table>
MLP in Out-of-Order Cores

OoO Core

Instruction Queue (CAM)

OoO cores issue ready instructions by skipping the stalled one
MLP in Out-of-Order Cores

I0  ld  r1=M[r7]
I1  add  r2=r1+1
I2  add  r3=r3+1
I3  ld  r4=M[r3]

OoO Core

Instruction Queue (CAM)

MLP

Power
MLP in In-Order Cores

**IO Core**

Instruction Queue (FIFO)

```
I0  ld  r1=M[r7]
I1  add r2=r1+1
I2  add r3=r3+1
I3  ld  r4=M[r3]
```
MLP in In-Order Cores

IO Core

Instruction Queue (FIFO)

I0  ld  r1=M[r7]
I1  add  r2=r1+1
I2  add  r3=r3+1
I3  ld  r4=M[r3]

Ready instructions behind the first stalled instruction are also stalled (not issued)
MLP in Modern Cores

I0  ld  r1=M[r7]
I1  add  r2=r1+1
I2  add  r3=r3+1
I3  ld  r4=M[r3]

MLP

Power

![MLP in Modern Cores](image-url)
The Big Question

Can we achieve MLP benefits of an OoO core with the power cost of an IO core?
A Step Forward

Slice-Out-or-Order Execution: Execute slices of MLP generating instruction out-of-order only w.r.t. the rest of instructions

Slice 1
\[
\begin{align*}
I_0 & : \text{ld} \ r1 = M[r7] \\
I_1 & : \text{add} \ r2 = r1 + 1
\end{align*}
\]

Slice 2
\[
\begin{align*}
I_2 & : \text{add} \ r3 = r3 + 1 \\
I_3 & : \text{ld} \ r4 = M[r3]
\end{align*}
\]

Slice: sequence of address generating instructions leading up to a load or store
A Step Forward

Slice-Out-or-Order Execution: Execute slices of MLP generating instruction out-of-order only w.r.t. the rest of instructions

Load Slice Core (LSC)

Instruction Queue (FIFO)

Bypass Queue (FIFO)

Slice 1

ld  r1=M[r7]
add r2=r1+1

Slice 2

add  r3=r3+1
ld  r4=M[r3]
A Step Forward

Slice-Out-or-Order Execution: Execute slices of MLP generating instruction out-of-order only w.r.t. the rest of instructions

Load Slice Core (LSC)

Instruction Queue (FIFO)

Bypass Queue (FIFO)
A Step Forward

Load Slice Core [ISCA’15]

Slice-Out-or-Order Execution: Execute slices of MLP generating instruction out-of-order only w.r.t. the rest of instructions

Load Slice Core (LSC)

Instruction Queue (FIFO)

Bypass Queue (FIFO)

I0
I1
ld r1=M[r7]  
add r2=r1+1
I2
I3
add r3=r3+1
ld r4=M[r3]

Slice 1
Slice 2
A Step Forward

Load Slice Core [ISCA’15]

Slice-Out-or-Order Execution: Execute slices of MLP generating instruction out-of-order only w.r.t. the rest of instructions

Load Slice Core (LSC)

Instruction Queue (FIFO)

Bypass Queue (FIFO)
A Step Forward

Load Slice Core [ISCA’15]

Slice-Out-or-Order Execution: Execute *slices* of MLP generating instruction out-of-order only w.r.t. the rest of instructions

Load Slice Core (LSC)

MLP

Instruction Queue (FIFO)

Bypass Queue (FIFO)

Power

16
sOoO (LSC) Evaluation

Overhead over in-order core
Area: 15%  Power: 22%

Performance

19% performance opportunity

100%
80%
60%
40%
20%
0%

MLP

Power

LSC  Ideal-sOoO

time
Summarizing State-of-the-Art

<table>
<thead>
<tr>
<th></th>
<th>MLP</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-order cores</td>
<td>Sad</td>
<td>Happy</td>
</tr>
<tr>
<td>Out-of-order cores</td>
<td>Happy</td>
<td>Sad</td>
</tr>
<tr>
<td>Slice-out-of-order</td>
<td>Neutral</td>
<td>Happy</td>
</tr>
<tr>
<td>cores</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Our Goal

MLP

In-order cores

Power

Out-of-order cores

Slice-out-of-order cores
Observation: Dependent slices hurt MLP.

Slices that depends on the load instruction of an older slice
Observation: Dependent slices hurt MLP.

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)

Instruction Queue (FIFO)
Observation: Dependent slices hurt MLP.

ld  r1=M[r7] \}  Slice 1
add  r2=r1+1

add  r3=r1+1 \}  Slice 2
ld  r4=M[r3]

ld  r6=M[r5] \}  Slice 3

Instruction Queue (FIFO)

Bypass Queue (FIFO)
Observation: Dependent slices hurt MLP.

Instruction Queue (FIFO):

Bypass Queue (FIFO):

Blocked independent slice
Sources of stalls in the bypass queue

65% of the avoidable stalls are caused by dependent slices
Towards Unobstructed MLP Extraction

**Problem:** Dependent slices block subsequent independent slices and limit MLP

**Idea:** Steer dependent slices to a FIFO yielding queue (Y-IQ).

**Challenges:**
- Which dependent slices to steer: all, only a subset?
- Is a single Y-IQ enough: wouldn’t dependent slices block each other?
Which Dependent Slices Go To Y-IQ?

**Intuitively:** Slices that block for long intervals
- i.e. producers hit in LLC or memory

Most stalls due to short L1 hits: need to steer all the dependent slices to Y-IQ.
Is a Single Yielding Queue Enough?

**Problem:** Y-IQ will block if younger slices become ready before the older slices.

**Scenario:**

- S1
- S2
- S3
- S4
Is a Single Yielding Queue Enough?

**Problem:** Y-IQ will block if younger slices become ready before the older slices.

**Scenario:**

Instruction Queue (FIFO)

Bypass Queue (FIFO)

Yielding Queue (FIFO)
Is a Single Yielding Queue Enough?

**Problem:** Y-IQ will block if younger slices become ready before the older slices.

**Scenario:**

- S1
- S2
- S3
- S4

- Instruction Queue (FIFO)
- Bypass Queue (FIFO)
- Yielding Queue (FIFO)

- L1 Hit
- Memory Hit

- S3
- S1
- S4
- S2
Is a Single Yielding Queue Enough?

**Problem:** Y-IQ will block if younger slices become ready before the older slices.

**Scenario:**

- **S1**
- **S2**
- **S3**
- **S4**

Instruction Queue (FIFO)

Bypass Queue (FIFO)

Yielding Queue (FIFO)

Memory Hit

L1 Hit

Blocked behind a non-ready slice
Is a Single Yielding Queue Enough?

Independent slice hit location

Majority of independent slices hit at same cache level (L1), dependent slices usually become ready in program order

Single Y-IQ is sufficient as dependent slices usually don’t block each other
Freeway Microarchitecture

IST (Instruction Slice Table)
- Tracks memory slices
- Identifies if a (pre-)decoded instruction belongs to a memory slice
Freeway Microarchitecture

I-cache → Decoder → Register Rename

IST → RDT

Slice Dependence Bit

Fetch → Decode → Rename → Dispatch

Existing Components
New Components

RDT (Register Dependence Table)
- Extended to propagate slice dependency information
Freeway Microarchitecture

I-cache → Decoder → Register Rename → Scoreboard

IST → RDT

Fetch → Decode → Rename →dispatch

Scoreboard → Backend

A-IQ → B-IQ → Y-IQ

Slice Dependence Bit

<table>
<thead>
<tr>
<th>Slice</th>
<th>Dependent</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>-</td>
<td>A-IQ</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>B-IQ</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Y-IQ</td>
</tr>
</tbody>
</table>

Existing Components

New Components
Identifying Memory Slices

Iterative Backwards Dependence Analysis (IBDA) \( LSC_{ISCA'15} \)

**Decode Stage:**
- PC in IST? Yes: Inst Slice; No: Not a slice

**Rename Stage:**
- Associate PhyReg to PC
- For slice inst, insert the producers’ PC to IST

---

**Instruction slice Table (IST)**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Tag</th>
<th>Tag</th>
<th>Tag</th>
</tr>
</thead>
</table>

**Register Dependence Table (RDT)**

| p1 | p2 | p3 | p4 | ...
|----|----|----|----|---

**Example Instructions**

<table>
<thead>
<tr>
<th>I0</th>
<th>add</th>
<th>r3=r2+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>ld</td>
<td>r4=M[r3]</td>
</tr>
</tbody>
</table>
Identifying Memory Slices

Iterative Backwards Dependence Analysis (IBDA) \textit{LSC}[ISCA’15]

Instruction slice Table (IST)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Tag</th>
<th>Tag</th>
<th>Tag</th>
</tr>
</thead>
</table>

Register Dependence Table (RDT)

| p1 | p2 | p3 | I0 | p4 | ...

- Decode Stage:
  - PC in IST? Yes: Inst Slice; No: Not a slice

- Rename Stage:
  - Associate PhyReg to PC
  - For slice inst, insert the producers’ PC to IST

Iteration 1

Add: $r3 = r2 + 1$

Load: $r4 = M[r3]$
Identifying Memory Slices

Iterative Backwards Dependence Analysis (IBDA) \( LSC_{[ISCA'15]} \)

### Decode Stage:
- PC in IST? Yes: Inst Slice; No: Not a slice

### Rename Stage:
- Associate PhyReg to PC
- For slice inst, insert the producers’ PC to IST
Identifying Memory Slices

Iterative Backwards Dependence Analysis (IBDA) \( LSC[*ISCA*'15] \)

**Decode Stage:**
- PC in IST? Yes: Inst Slice; No: Not a slice

**Rename Stage:**
- Associate PhyReg to PC
- For slice inst, insert the producers’ PC to IST

---

Instruction slice Table (IST) | Register Dependence Table (RDT)
---|---
Tag | p1
Tag | p2
Tag | p3 | I0
Tag | p4
Tag | ...

Iteration 1

I0 add p3=p2+1

I1 ld r4=M[r3]
Identifying Memory Slices

Iterative Backwards Dependence Analysis (IBDA) \( LSC_{[ISCA'15]} \)

**Instruction slice Table (IST)**

- Tag
- Tag
- Tag
- Tag

**Register Dependence Table (RDT)**

|       | p1 | p2 | p3 | p4 | ...
|-------|----|----|----|----|-----
| I0    |    |    |    |    |     
| p1    |    |    |    |    |     
| p2    |    |    |    |    |     
| p3    |    |    |    |    |     
| p4    |    |    |    |    |     
| ...   |    |    |    |    |     

**Iteration 1**

- I0  add  p3=p2+1

**Decode Stage:**
- PC in IST? Yes: Inst Slice; No: Not a slice

**Rename Stage:**
- Associate PhyReg to PC
- For slice inst, insert the producers’ PC to IST
Identifying Memory Slices

Iterative Backwards Dependence Analysis (IBDA) \(LSC[ISCA'15]\)

**Decode Stage:**
- PC in IST? Yes: Inst Slice; No: Not a slice

**Rename Stage:**
- Associate PhyReg to PC
- For slice inst, insert the producers’ PC to IST

![Diagram showing the process of identifying memory slices with Iterative Backwards Dependence Analysis (IBDA)]
Identifying Memory Slices

Iterative Backwards Dependence Analysis (IBDA) \( LSC_{[ISCA'15]} \)

*Decode Stage:*
- PC in IST? Yes: Inst Slice; No: Not a slice

*Rename Stage:*
- Associate PhyReg to PC
- For slice inst, insert the producers’ PC to IST

---

### Instruction slice Table (IST)

<table>
<thead>
<tr>
<th>Tag</th>
<th>I0</th>
</tr>
</thead>
</table>

### Register Dependence Table (RDT)

|   | p1 | p2 | p3 | p4 | ...
|---|----|----|----|----|------
| I0 |    |    | I0 | I1 |      

---

**Iteration 1**

**Decode Stage:**
- \( I_0 \) add \( p_3 = p_2 + 1 \)
- \( I_1 \) ld \( r_4 = M[r_3] \)

**Rename Stage:**
- \( I_1 \) ld \( p_4 = M[p_3] \)
Identifying Memory Slices

Iterative Backwards Dependence Analysis (IBDA) \[LSC_{ISCA'15}\]

### Instruction slice Table (IST)
- Tag
- Tag
- **Tag**: I0
- Tag

### Register Dependence Table (RDT)
<table>
<thead>
<tr>
<th></th>
<th>p1</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p3</td>
<td></td>
<td>I0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p4</td>
<td></td>
<td>I1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Decode
- I0  add  r3=r2+1

#### Rename
- Iteration 1
  - I0  add  p3=p2+1
  - I1  ld  p4=M[p3]

#### Instruction
- I0  add  r3=r2+1
- I1  ld  r4=M[r3]
Identifying Memory Slices

Iterative Backwards Dependence Analysis (IBDA) \textit{LSC[ISCA'15]}

**Instruction slice Table (IST)**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0</td>
<td>add r3=r2+1</td>
</tr>
<tr>
<td>Tag</td>
<td></td>
</tr>
</tbody>
</table>

**Register Dependence Table (RDT)**

<table>
<thead>
<tr>
<th>p1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p3</td>
<td>I0</td>
<td></td>
</tr>
<tr>
<td>p4</td>
<td>I1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Iteration 1:

- I0 add p3=p2+1
- I1 ld p4=M[p3]

Iteration 2:

- I0 add r3=r2+1
- I1 ld r4=M[r3]
Identifying Memory Slices

Iterative Backwards Dependence Analysis (IBDA) \( LSC[\text{ISCA'15}] \)

**Instruction slice Table (IST)**

<table>
<thead>
<tr>
<th>Tag</th>
<th>I0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>I0</td>
</tr>
<tr>
<td>Tag</td>
<td></td>
</tr>
</tbody>
</table>

**Register Dependence Table (RDT)**

<table>
<thead>
<tr>
<th></th>
<th>p1</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p3</td>
<td>I0</td>
<td></td>
</tr>
<tr>
<td>p4</td>
<td>I1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Decode**

- I0 add r3=r2+1
- I1 ld r4=M[r3]

**Rename**

- I0 add p3=p2+1

**Iteration 1**

- I0 add p3=p2+1
- I1 ld p4=M[p3]

**Iteration 2**

- I0 add p3=p2+1
Freeway: Identifying Dependent Slices

Rename

**Register Dependence Table**

<table>
<thead>
<tr>
<th>p1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
<td></td>
</tr>
<tr>
<td>p3</td>
<td></td>
</tr>
<tr>
<td>p4</td>
<td></td>
</tr>
<tr>
<td>p5</td>
<td></td>
</tr>
</tbody>
</table>

Instruction reading this physical register belongs to a dependent slice or not?
Freeway: Identifying Dependent Slices

### Rename

#### Register Dependence Table

<table>
<thead>
<tr>
<th>p1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
<td>0</td>
</tr>
<tr>
<td>p3</td>
<td>0</td>
</tr>
<tr>
<td>p4</td>
<td>0</td>
</tr>
<tr>
<td>p5</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Code Snippet

```plaintext
I0: ld  p1=M[p2]
I1: add p2=p1+1
I2: add p3=p1+1
I3: ld  p4=M[p3]
I4: ld  p2=M[p5]
```
Freeway: Identifying Dependent Slices

**Register Dependence Table**

<table>
<thead>
<tr>
<th>p1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
<td>0</td>
</tr>
<tr>
<td>p3</td>
<td>0</td>
</tr>
<tr>
<td>p4</td>
<td>0</td>
</tr>
<tr>
<td>p5</td>
<td>0</td>
</tr>
</tbody>
</table>

**Rename**

- I0: `ld p1=M[p2]`
- I1: `add p2=p1+1`
- I2: `add p3=p1+1`
- I3: `ld p4=M[p3]`
- I4: `ld p2=M[p5]`

0: Independent Slice
Freeway: Identifying Dependent Slices

**Register Dependence Table**

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
<th>p5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

I0: \( \text{ld } p1 = M[p2] \)
I1: \( \text{add } p2 = p1 + 1 \)
I2: \( \text{add } p3 = p1 + 1 \)
I3: \( \text{ld } p4 = M[p3] \)
I4: \( \text{ld } p2 = M[p5] \)

p1 consumers would be dependent slice instructions.
Freeway: Identifying Dependent Slices

Rename

Register Dependence Table

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
<th>p5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

I0: ld p1=M[p2]
I1: add p2=p1+1
I2: add p3=p1+1
I3: ld p4=M[p3]
I4: ld p2=M[p5]
Freeway: Identifying Dependent Slices

Register Dependence Table

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rename

p3 consumers would be dependent slice instructions

I0  ld  p1=M[p2]
I1  add p2=p1+1
I2  add p3=p1+1
I3  ld  p4=M[p3]
I4  ld  p2=M[p5]
Freeway: Identifying Dependent Slices

Register Dependence Table

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
<th>p5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

1: Dependent Slice

I0: ld p1=M[p2]
I1: add p2=p1+1
I2: add p3=p1+1
I3: ld p4=M[p3]
I4: ld p2=M[p5]
Freeway: Identifying Dependent Slices

Register Dependence Table

<table>
<thead>
<tr>
<th>p1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
<td>0</td>
</tr>
<tr>
<td>p3</td>
<td>1</td>
</tr>
<tr>
<td>p4</td>
<td>0</td>
</tr>
<tr>
<td>p5</td>
<td>0</td>
</tr>
</tbody>
</table>

1: Dependent Slice

Freeway identifies dependent slices with only one additional bit per RDT entry
Evaluation Methodology

• Core based on Intel Slilvermont/ARM Cortex-A9
  – 2-wide
  – 64-entry instruction window

• Workloads
  – SPECcpu2006 with multiple inputs

• Evaluated cores:
  – In-order (Baseline)
  – Load Slice Core [ISCA’15]
  – Freeway
  – Ideal-sOoO (MLP limit)
  – OoO (ILP+MLP limit)
Performance Comparison

Freeway delivers 12% more performance than LSC
Performance Comparison

Freeway delivers twice as much performance as LSC
Performance Comparison

Freeway delivers twice as much performance as LSC
Performance Comparison

Same performance as LSC because of negligible dependent slices
Performance Comparison

Freeway is within 7% of **Ideal sOoO core**
Performance Comparison

Freeway is within 15% of full OoO performance for a number of workloads.

The graph shows the performance comparison of Freeway and full OoO for various workloads, with Freeway achieving up to 222% of the ideal performance, indicating it is within 15% of full OoO performance for a number of workloads.
Performance Comparison

On avg, Freeway is within 33% of Fully OoO core
Freeway Summary

• Memory latency is still a critical bottleneck
  – Large datasets, growing number of cores and accelerators
• MLP is key to hiding memory latency
  – OoO cores pay high power cost to extract MLP
• Slice-OoO (LSC) aims to reduce the power cost
  – Leaves significant MLP due to slice dependencies
• Freeway
  – Introduces dependence aware slice execution
  – Approaches the MLP benefits of an OoO core with power cost of a slice-OoO core.