Branch Runahead:
An Alternative to Branch Prediction for Impossible to Predict Branches

Stephen Pruett and Yale N. Patt
University of Texas at Austin
MICRO 2021

Seminar in Computer Architecture
Presenter: Haocong Luo
Executive Summary

■ **Problem:** History-based branch predictors (BP) are bottlenecked by hard-to-predict (H2P) data-dependent branches.

■ **Goal:** Improve system performance by reducing branch mispredictions.

■ **Key Idea:** Pre-compute H2P data-dependent branches continuously.
  - Dynamically detect light-weight dependence chains at run-time.
  - New merge point predictor reduces the divergence from the main thread.
  - Can be implemented either as a dedicated unit or part of the core.

■ **Key Results:**
  - Reduces MPKI by 43.6% and increases IPC by 13.7% on average.
  - 2.2% area overhead of a baseline out-of-order core.
Outline

■ Background
  - Limitations of Existing Branch Prediction
  - Alternative to Branch Prediction

■ Branch Runahead
  - Dynamically Extracting Dependence Chains
  - Handling Nested Branches
  - Putting Things Together
  - Optimizations: Chain Initialization Methods

■ Evaluation
  - Methodology
  - Results

■ Analysis
  - Strengths & Weaknesses
  - Discussion
Outline

- **Background**
  - Limitations of Existing Branch Prediction
  - Alternative to Branch Prediction

- **Branch Runahead**
  - Dynamically Extracting Dependence Chains
  - Handling Nested Branches
  - Putting Things Together
  - Optimizations: Chain Initialization Methods

- **Evaluation**
  - Methodology
  - Results

- **Analysis**
  - Strengths & Weaknesses
  - Discussion
Limitations of Existing Branch Prediction

- **History-based Branch Prediction** (e.g. TAGE)
  Predict if the current branch is taken or not by looking into its history.

- **Problem**: Data-dependent branches are hard to predict (H2P).
  The outcome of the branch depends on the data, not necessarily the history.

```
1   pos = position on GO board;
2   for (i = 0; i < 8; i ++)
3       sq = pos + neighbor_offset [i];
4       if (board [sq] == EMPTY)
5           ...
```

- Improvements in branch prediction are becoming rare because these H2P branches take up most of the remaining mispredictions.
Data dependent branches are the main bottleneck of branch prediction.
Outline

■ Background
  - Limitations of Existing Branch Prediction
  - Alternative to Branch Prediction

■ Branch Runahead
  - Dynamically Extracting Dependence Chains
  - Handling Nested Branches
  - Putting Things Together
  - Optimizations: Chain Initialization Methods

■ Evaluation
  - Methodology
  - Results

■ Analysis
  - Strengths & Weaknesses
  - Discussion
Alternative to Branch Prediction

- **Pre-computation**
  Filter the program and execute ahead of the main thread to load the data early.

- **Problems**: Not efficient and/or effective enough.
  - The filtered thread usually ends up executing most of the program.
  - High hardware cost: Almost 2x core resources are needed.
  - Light-weight runtime methods cannot guarantee timeliness.
  - Requires changes in the ISA.

Existing pre-computation methods are not suitable as branch prediction alternatives.
Outline

■ Background
  - Limitations of Existing Branch Prediction
  - Alternative to Branch Prediction

■ Branch Runahead
  - Dynamically Extracting Dependence Chains
  - Handling Nested Branches
  - Putting Things Together
  - Optimizations: Chain Initialization Methods

■ Evaluation
  - Methodology
  - Results

■ Analysis
  - Strengths & Weaknesses
  - Discussion
Branch Runahead

- **Goal:** Improve system performance by reducing branch mispredictions.

- **Key Idea:** Pre-compute H2P data-dependent branches continuously.
  - Dynamically detect light-weight dependence chains at run-time.

```c
1   pos = position on GO board;
2   for (i = 0; i < 8; i ++)  
3       sq = pos + neighbor_offset [i];
4   if (board [sq] == EMPTY) 
5       ...
```

- Dependence chains are shipped to the Dependence Chain Engine (DCE).
- In the DCE, dependence chains are executed continuously as if in a loop.
Outline

■ Background
  - Limitations of Existing Branch Prediction
  - Alternative to Branch Prediction

■ Branch Runahead
  - Dynamically Extracting Dependence Chains
  - Handling Nested Branches
  - Putting Things Together
  - Optimizations: Chain Initialization Methods

■ Evaluation
  - Methodology
  - Results

■ Analysis
  - Strengths & Weaknesses
  - Discussion
Dynamically Extracting Dependence Chains

■ **When**: When a H2P branch is retired.
  - H2P branches are tracked by a Hard Branch Table (HBT).
  - Conditional branches are added to HBT when they retire.
  - Each entry in the HBT has a saturating counter for mispredictions.

■ **How**: Backward dataflow walk from the Chain Extraction Buffer (CEB).
  - When instructions retire, they are added to the CEB.
  - When a mispredicted H2P branch is retired, starts the walk:
    1. Initialize the live-in vector with the registers that the branch uses.
    2. Add instructions whose dst. are in the live-in vector to the chain.
    3. Update the live-in vector as the src. of the added instructions.
    4. Stop until we hit a previous instance of the branch.
Dynamically Extracting Dependence Chains

**How:** Backward dataflow walk from the Chain Extraction Buffer (CEB).

<table>
<thead>
<tr>
<th>PC</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7</td>
<td>BR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0xA</td>
<td>ADD P3 &lt;- P3, 4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0xC</td>
<td>LD P7 &lt;- [P3]</td>
</tr>
<tr>
<td>0xD</td>
<td>ADD P7 &lt;- P7, P5</td>
</tr>
<tr>
<td>0x1</td>
<td>MOV P2 &lt;- P7</td>
</tr>
<tr>
<td>0x3</td>
<td>LD P0 &lt;- [P2]</td>
</tr>
<tr>
<td>0x5</td>
<td>CMP P0, 2</td>
</tr>
<tr>
<td>0x7</td>
<td>BR (H2P, MISS)</td>
</tr>
</tbody>
</table>

Live-in: {}
Dynamically Extracting Dependence Chains

How: Backward dataflow walk from the Chain Extraction Buffer (CEB).

<table>
<thead>
<tr>
<th>PC</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7</td>
<td>BR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0xA</td>
<td>ADD P3 &lt;- P3, 4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0xC</td>
<td>LD P7 &lt;- [P3]</td>
</tr>
<tr>
<td>0xD</td>
<td>ADD P7 &lt;- P7, P5</td>
</tr>
<tr>
<td>0x1</td>
<td>MOV P2 &lt;- P7</td>
</tr>
<tr>
<td>0x3</td>
<td>LD P0 &lt;- [P2]</td>
</tr>
<tr>
<td>0x5</td>
<td>CMP P0, 2</td>
</tr>
<tr>
<td>0x7</td>
<td>BR (H2P, MISS)</td>
</tr>
</tbody>
</table>

Live-in: \{P0\}
Dynamically Extracting Dependence Chains

- **How:** Backward dataflow walk from the Chain Extraction Buffer (CEB).

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD P7 &lt;- [P3]</td>
<td>0x1</td>
</tr>
<tr>
<td>ADD P7 &lt;- P7, P5</td>
<td>0xC</td>
</tr>
<tr>
<td>MOV P2 &lt;- P7</td>
<td>0x1</td>
</tr>
<tr>
<td>LD P0 &lt;- [P2]</td>
<td>0x3</td>
</tr>
<tr>
<td>CMP P0, 2</td>
<td>0x5</td>
</tr>
<tr>
<td>BR (H2P, MISS)</td>
<td>0x7</td>
</tr>
</tbody>
</table>
```

**Live-in:** \{P0\}
Dynamically Extracting Dependence Chains

**How:** Backward dataflow walk from the Chain Extraction Buffer (CEB).

<table>
<thead>
<tr>
<th>PC</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7</td>
<td>BR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0xA</td>
<td>ADD P3 &lt;- P3, 4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0xC</td>
<td>LD P7 &lt;- [P3]</td>
</tr>
<tr>
<td>0xD</td>
<td>ADD P7 &lt;- P7, P5</td>
</tr>
<tr>
<td>0x1</td>
<td>MOV P2 &lt;- P7</td>
</tr>
<tr>
<td>0x3</td>
<td>LD P0 &lt;- [P2]</td>
</tr>
<tr>
<td>0x5</td>
<td>CMP P0, 2</td>
</tr>
<tr>
<td>0x7</td>
<td>BR (H2P, MISS)</td>
</tr>
</tbody>
</table>

Live-in: \{P0\}
Dynamically Extracting Dependence Chains

**How:** Backward dataflow walk from the Chain Extraction Buffer (CEB).

```
PC       Instruction
0x7      BR

...     ...

0xA     ADD P3 <- P3, 4

...     ...

0xC     LD P7 <- [P3]
0xD     ADD P7 <- P7, P5
0x1     MOV P2 <- P7
0x3     LD P0 <- [P2]
0x5     CMP P0, 2
0x7     BR (H2P, MISS)
```

**Live-in:** {P2}
Dynamically Extracting Dependence Chains

- **How:** Backward dataflow walk from the Chain Extraction Buffer (CEB).

<table>
<thead>
<tr>
<th>PC</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7</td>
<td>BR</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0xA</td>
<td>ADD P3 &lt;- P3, 4</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0xC</td>
<td>LD P7 &lt;- [P3]</td>
</tr>
<tr>
<td>0xD</td>
<td>ADD P7 &lt;- P7, P5</td>
</tr>
<tr>
<td>0x1</td>
<td>MOV P2 &lt;- P7</td>
</tr>
<tr>
<td>0x3</td>
<td>LD P0 &lt;- [P2]</td>
</tr>
<tr>
<td>0x5</td>
<td>CMP P0, 2</td>
</tr>
<tr>
<td>0x7</td>
<td>BR (H2P, MISS)</td>
</tr>
</tbody>
</table>

**Live-in:** \{P2\}
Dynamically Extracting Dependence Chains

**How:** Backward dataflow walk from the Chain Extraction Buffer (CEB).

<table>
<thead>
<tr>
<th>PC</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7</td>
<td>BR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0xA</td>
<td>ADD P3 &lt;- P3, 4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0xC</td>
<td>LD P7 &lt;- [P3]</td>
</tr>
<tr>
<td>0xD</td>
<td>ADD P7 &lt;- P7, P5</td>
</tr>
<tr>
<td>0x1</td>
<td>MOV P2 &lt;- P7</td>
</tr>
<tr>
<td>0x3</td>
<td>LD P0 &lt;- [P2]</td>
</tr>
<tr>
<td>0x5</td>
<td>CMP P0, 2</td>
</tr>
<tr>
<td>0x7</td>
<td>BR (H2P, MISS)</td>
</tr>
</tbody>
</table>

Live-in: \{P7\}
Dynamically Extracting Dependence Chains

- **How:** Backward dataflow walk from the Chain Extraction Buffer (CEB).

    | PC | Instruction                     |
    |----|---------------------------------|
    | 0x7 | BR                             |
    |     |                                |
    | 0xA | ADD P3 <- P3, 4               |
    |     |                                |
    | 0xC | LD P7 <- [P3]                 |
    | 0xD | ADD P7 <- P7, P5              |
    | 0x1 | MOV P2 <- P7                  |
    | 0x3 | LD P0 <- [P2]                 |
    | 0x5 | CMP P0, 2                     |
    | 0x7 | BR (H2P, MISS)                |

    **Live-in:** {P7, P5}
Dynamically Extracting Dependence Chains

- **How:** Backward dataflow walk from the Chain Extraction Buffer (CEB).

```
PC  Instruction
---  ------------------
0x7  BR

...  

0xA  ADD P3 <- P3, 4

...  

0xC  LD P7 <- [P3]
0xD  ADD P7 <- P7, P5
0x1  MOV P2 <- P7
0x3  LD P0 <- [P2]
0x5  CMP P0, 2
0x7  BR (H2P, MISS)

Live-in: {P3, P5}
```
Dynamically Extracting Dependence Chains

**Prediction:** Chains start executing in the DCE once the live-ins are ready.
- The computed outcome will then overwrite branch prediction results.
- The chain executes continuously until a misprediction happens.
- Execution resumes after synchronizing the live-ins with the main thread.

**Problem:** Frequent synchronization hurts performance, esp. nested branches.
- What if the extracted chain contains other branches?
- Prevents the chains to be executed ahead enough of the main thread.

```c
3    sq = pos + neighbor_offset [i];
4    if (board [sq] == EMPTY) // Branch A
5    if(not board [sq].self_atari()) // Branch B
```
Outline

- **Background**
  - Limitations of Existing Branch Prediction
  - Alternative to Branch Prediction

- **Branch Runahead**
  - Dynamically Extracting Dependence Chains
  - Handling Nested Branches
  - Putting Things Together
  - Optimizations: Chain Initialization Methods

- **Evaluation**
  - Methodology
  - Results

- **Analysis**
  - Strengths & Weaknesses
  - Discussion
Handling Nested Branches

- **Problem:** Frequent synchronization hurts performance, esp. nested branches.
  - **Guard:** Branches that determines if other branches are executed or not.
    
    ```
    1 pos = position on GO board;
    2 for (i = 0; i < 8; i ++)
    3   sq = pos + neighbor_offset [i];
    4   if (board [sq] == EMPTY) // Branch A
    5     if(not board [sq].self_atari()) // Branch B
    6     ...
    ```

  - **Affector:** Branches that affect the source data of other branches.
    
    ```
    1 x = 0;
    2 if (foo()) // Branch A
    3     x = 1;
    4 if (x == 1) // Branch B
    5     ...
    ```
Most branches are affected by guard/affector branches.

**Goal:** Track the dependency between chains to execute them more accurately.

**Key Idea:** Dynamically detect the **merge point** and break down the chains.
Handling Nested Branches

- **Merge Point** of a branch:
  The instruction where the control converges regardless of its outcome.

- **Key Observation:** Many wrong path instructions could be fetched and allocated before a misprediction is detected, including the merge point.

- **Key Idea:** Find the first instruction that is both in the wrong and correct path.

- **Wrong Path Buffer (WPB):** Stores the PCs of the wrong path instructions.
  - When the ROB is flushed due to misprediction, WPB is filled.
  - After recovery, retired correct instructions use their PCs to index the WPB.
  - A merge point is find if there is a hit.
Handling Nested Branches

- **Detecting Guard Branches**
  - A branch guards any branches between itself and the merge point.
  - Any branch encountered during merge point prediction is guarded by the merge-point-predicting branch.

- **Detecting Affector (Affectee) Branches**
  - Need to look for registers written between itself and the merge point.
  - Use forward dataflow walk to find branches that sources those registers.

- **Tagging the Dependence Chains**
  - Chain detection now terminates at the first guard/affector branch.
  - Tag the chain with that branch and the outcome (if it is a guard).
  - Heavily biased guard/affector branches are ignored.
Outline

■ Background
  - Limitations of Existing Branch Prediction
  - Alternative to Branch Prediction

■ Branch Runahead
  - Dynamically Extracting Dependence Chains
  - Handling Nested Branches
  - Putting Things Together
  - Optimizations: Chain Initialization Methods

■ Evaluation
  - Methodology
  - Results

■ Analysis
  - Strengths & Weaknesses
  - Discussion
Branch Runahead

■ Workflow

1. Dependence chains are extracted into DCC as H2P branches retire.
2. Upon misprediction, the branch PC and the outcome are used to initiate chains.
3. Chains execute continuously, and the outcomes are used to initiate more chains whose tag match.
4. The outcomes are inserted into the prediction queue that could overwrite predictions from TAGE-SC-L.
5. Exit runahead if no outcomes hit in DCC.
Branch Runahead

**Workflow**

1. Dependence chains are extracted into DCC as H2P branches retire.
2. Upon misprediction, the branch PC and the outcome are used to initiate chains.
3. Chains execute continuously, and the outcomes are used to initiate more chains whose tag match.
4. The outcomes are inserted into the prediction queue that could overwrite predictions from TAGE-SC-L.
5. Exit runahead if no outcomes hit in DCC.
Branch Runahead

Workflow

1. Dependence chains are extracted into DCC as H2P branches retire.
2. Upon misprediction, the branch PC and the outcome are used to initiate chains.
3. Chains execute continuously, and the outcomes are used to initiate more chains whose tag match.
4. The outcomes are inserted into the prediction queue that could overwrite predictions from TAGE-SC-L.
5. Exit runahead if no outcomes hit in DCC.
Branch Runahead

**Workflow**

1. Dependence chains are extracted into DCC as H2P branches retire.

2. Upon misprediction, the branch PC and the outcome are used to initiate chains.

3. Chains execute continuously, and the outcomes are used to initiate more chains whose tag match.

4. The outcomes are inserted into the prediction queue that could overwrite predictions from TAGE-SC-L.

5. Exit runahead if no outcomes hit in DCC.
Branch Runahead

**Workflow**

1. Dependence chains are extracted into DCC as H2P branches retire.
2. Upon misprediction, the branch PC and the outcome are used to initiate chains.
3. **Chains execute continuously, and the outcomes are used to initiate more chains whose tag match.**
4. The outcomes are inserted into the prediction queue that could overwrite predictions from TAGE-SC-L.
5. Exit runahead if no outcomes hit in DCC.
Branch Runahead

**Workflow**

1. Dependence chains are extracted into DCC as H2P branches retire.
2. Upon misprediction, the branch PC and the outcome are used to initiate chains.
3. Chains execute continuously, and the outcomes are used to initiate more chains whose tag match.
4. **The outcomes are inserted into the prediction queue that could overwrite predictions from TAGE-SC-L.**
5. Exit runahead if no outcomes hit in DCC.
Branch Runahead

**Workflow**

1. Dependence chains are extracted into DCC as H2P branches retire.
2. Upon misprediction, the branch PC and the outcome are used to initiate chains.
3. Chains execute continuously, and the outcomes are used to initiate more chains whose tag match.
4. The outcomes are inserted into the prediction queue that could overwrite predictions from TAGE-SC-L.
5. Exit runahead if no outcomes hit in DCC.
Outline

■ Background
  - Limitations of Existing Branch Prediction
  - Alternative to Branch Prediction

■ Branch Runahead
  - Dynamically Extracting Dependence Chains
  - Handling Nested Branches
  - Putting Things Together
  - Optimizations: Chain Initialization Methods

■ Evaluation
  - Methodology
  - Results

■ Analysis
  - Strengths & Weaknesses
  - Discussion
Optimizations: Chain Initialization Methods

The aggressiveness of dependent chain initialization is critical to performance.

- **Non-speculative Initiation**
  - A chain must finish execution before initializing the next dependent chain.

- **Independent-early Initiation**
  - Chains with a wildcard (don’t care) outcome tag can be initialized when the triggering branch starts executing.

- **Predictive Initiation**
  - Predicting the results of the dependent chain.
  - Allows chains with non-wildcard outcome tag to be issued early.
  - Flush if the prediction turns out to be wrong.
Outline

- Background
  - Limitations of Existing Branch Prediction
  - Alternative to Branch Prediction

- Branch Runahead
  - Dynamically Extracting Dependence Chains
  - Handling Nested Branches
  - Optimizations: Chain Initialization Methods

- Evaluation
  - Methodology
  - Results

- Analysis
  - Strengths & Weaknesses
  - Discussion
Evaluation

- **Methodology**
  - Scarab + Ramulator + McPAT
  - 64KB TAGE-SC-L as baseline
  - Stream Prefetcher: 64 Streams, Distance 16, into LLC
  - SPEC CPU2017 Integer Speed, SPEC CPU2006 Integer, and GAP

- **Branch Runahead Configurations**
  - Core-Only (Shares the RS, ALU, and RF with the core): 9KB
  - DCE-Mini: 17KB
  - DCE-Big: Unlimited

- **Metrics:**
  - Instructions Per Cycle (IPC)
  - Mispredictions Per Kilo Instruction (MPKI)
Outline

■ Background
  - Limitations of Existing Branch Prediction
  - Alternative to Branch Prediction

■ Branch Runahead
  - Dynamically Extracting Dependence Chains
  - Handling Nested Branches
  - Putting Things Together
  - Optimizations: Chain Initialization Methods

■ Evaluation
  - Methodology
  - Results

■ Analysis
  - Strengths & Weaknesses
  - Discussion
Branch Runahead is effective in improving system performance by significantly reduce the misprediction rate for H2P data dependent branches.
Data dependent branches are the main bottleneck of branch prediction.
Increasing the aggressiveness of chain initialization increases performance.
Evaluation - Prediction Breakdown

Branch Runahead is accurate, but timeliness is still a problem.
Branch Runahead saves energy by reducing execution time.
Evaluation - Misc

- **Area**
  - DCE engine area: 0.38 mm\(^2\) (2.2% of a baseline out-of-order core)
  - 64KB TAGE-SC-L: 0.73 mm\(^2\)

- **Processor Frequency:** Minimal impact.
  - Most of the DCE is not on the critical path.
  - The execution is divided into many cycles.
  - The only influence is the MUX between the TAGE result and DCE result.

- **No changes to the ISA and the compiler.**

Branch Runahead is a low overhead design.
Executive Summary

- **Problem:** History-based branch predictors (BP) are bottlenecked by hard-to-predict (H2P) data-dependent branches.

- **Goal:** Improve system performance by reducing branch mispredictions.

- **Key Idea:** Pre-compute H2P data-dependent branches continuously.
  - Dynamically detect light-weight dependence chains at run-time.
  - New merge point predictor reduces the divergence from the main thread.
  - Can be implemented either as a dedicated unit or part of the core.

- **Key Results:**
  - Reduces MPKI by 43.6% and increases IPC by 13.7% on average.
  - 2.2% area overhead of a baseline out-of-order core.
Outline

■ Background
  - Limitations of Existing Branch Prediction
  - Alternative to Branch Prediction

■ Branch Runahead
  - Dynamically Extracting Dependence Chains
  - Handling Nested Branches
  - Putting Things Together
  - Optimizations: Chain Initialization Methods

■ Evaluation
  - Methodology
  - Results

■ Analysis
  - Strengths & Weaknesses
  - Discussion
**Strengths**

- Branch prediction is critical to core performance.
- Important observation that hard to predict branches are data dependent and thus cannot be effectively predicted by history-based approaches.
- New mechanism of using continuous runahead to accurately predict data dependent branches.
- Merge point prediction and the associated chain initialization methods significantly improves performance.
- Very comprehensive design.
Weakness

- **Low novelty**
  - Runahead (2003); Continuous Runahead (2016).
  - Anything fundamentally new?

- **No discussion of how BR interacts with context switch / SMT.**
  - Are the chains saved or discarded?
  - How to distinguish chains from different processes?

- **Questionable evaluation setup: The prefetcher is too simple.**
  - With a good enough prefetcher, larger ROB, wider issue, etc., how much of the benefit of BR could be achieved?
Outline

■ Background
  - Limitations of Existing Branch Prediction
  - Alternative to Branch Prediction

■ Branch Runahead
  - Dynamically Extracting Dependence Chains
  - Handling Nested Branches
  - Putting Things Together
  - Optimizations: Chain Initialization Methods

■ Evaluation
  - Methodology
  - Results

■ Analysis
  - Strengths & Weaknesses
  - Discussion
Discussion

■ Security
  - The DCE is connected to the core cache.
  - Opens up more opportunities for cache side-channel attacks?

■ Other components in the system where a similar approach can be applied?
  - When a missed LOAD is retired, starts a forward dataflow walk in ROB.
  - Dynamically identifies what data are the bottleneck (i.e., is blocking the most subsequent instructions).
  - Pin the cache line in the cache, prioritize this request in the memory controller, etc.

■ Extend this idea to GPU to dynamically reorganize warps better?