Address-Value Delta (AVD) Prediction

Onur Mutlu
Hyesoon Kim
Yale N. Patt
What is AVD Prediction?

A new prediction technique used to break the data dependencies between dependent load instructions
Talk Outline

- Background on Runahead Execution
- The Problem: Dependent Cache Misses
- AVD Prediction
- Why Does It Work?
- Evaluation
- Conclusions
Background on Runahead Execution

- A technique to obtain the memory-level parallelism benefits of a large instruction window

- When the oldest instruction is an L2 miss:
  - Checkpoint architectural state and enter runahead mode

- In runahead mode:
  - Instructions are speculatively pre-executed
  - The purpose of pre-execution is to generate prefetches
  - L2-miss dependent instructions are marked INV and dropped

- Runahead mode ends when the original L2 miss returns
  - Checkpoint is restored and normal execution resumes
Runahead Example

Small Window:

- Load 1 Miss
- Load 2 Miss
- Compute
- Stall
- Compute
- Stall
- Miss 1
- Miss 2

Runahead: *Works when Load 1 and 2 are independent*

- Load 1 Miss
- Load 2 Miss
- Compute
- Runahead
- Compute
- Load 1 Hit
- Load 2 Hit
- Saved Cycles
- Miss 1
- Miss 2
The Problem: Dependent Cache Misses

Runahead: Load 2 is dependent on Load 1

Cannot Compute Its Address!

Load 1 Miss  Load 2 INV  Load 1 Hit  Load 2 Miss

- Runahead execution cannot parallelize dependent misses
- This limitation results in
  - wasted opportunity to improve performance
  - wasted energy (useless pre-execution)
- Runahead performance would improve by 25% if this limitation were ideally overcome
The Goal

- Enable the parallelization of dependent L2 cache misses in runahead mode with a low-cost mechanism

How:
- Predict the values of L2-miss **address (pointer) loads**
  - **Address load**: loads an address into its destination register, which is later used to calculate the address of another load
  - as opposed to **data load**

AVD Prediction
Parallelizing Dependent Misses

- **Cannot Compute Its Address!**
- **Value Predicted**
- **Can Compute Its Address**

Load 1 Miss  Load 2 INV  Load 1 Hit  Load 2 Miss

Compute  Runahead  Miss 1  Miss 2

Saved Cycles
Saved Speculative Instructions
A Question

How can we predict the values of address loads with low hardware cost and complexity?
Talk Outline

- Background on Runahead Execution
- The Problem: Dependent Cache Misses
- AVD Prediction
- Why Does It Work?
- Evaluation
- Conclusions
The Solution: AVD Prediction

- Address-value delta (AVD) of a load instruction defined as:
  \[ \text{AVD} = \text{Effective Address of Load} - \text{Data Value of Load} \]

- For some address loads, AVD is stable
- An AVD predictor keeps track of the AVDs of address loads
- When a load is an L2 miss in runahead mode, AVD predictor is consulted

- If the predictor returns a stable (confident) AVD for that load, the value of the load is predicted
  \[ \text{Predicted Value} = \text{Effective Address} - \text{Predicted AVD} \]
Identifying Address Loads in Hardware

- **Insight:**
  - If the AVD is too large, the value that is loaded is likely **not** an address

- Only keep track of loads that satisfy:
  - $-\text{MaxAVD} \leq \text{AVD} \leq +\text{MaxAVD}$

- This identification mechanism eliminates many loads from consideration
  - Enables the AVD predictor to be small
An Implementable AVD Predictor

- Set-associative prediction table
- Prediction table entry consists of
  - Tag (Program Counter of the load)
  - Last AVD seen for the load
  - Confidence counter for the recorded AVD

- Updated when an address load is retired in normal mode
- Accessed when a load misses in L2 cache in runahead mode
- **Recovery-free:** No need to recover the state of the processor or the predictor on misprediction
  - Runahead mode is purely speculative
AVD Update Logic

Effective Address  Data Value

computed AVD = Effective Addr - Data Value

>= -MaxAVD?

<= MaxAVD?

valid AVD?

Confidence Update/Reset Logic

Tag Conf AVD

PC of Retired Load
AVD Prediction Logic

Predicted? (not INV?)

Predicted Value = Effective Addr − AVD

Program Counter of L2-miss Load

Effective Address of L2-miss Load
Talk Outline

- Background on Runahead Execution
- The Problem: Dependent Cache Misses
- AVD Prediction
- Why Does It Work?
- Evaluation
- Conclusions
Why Do Stable AVDs Occur?

- Regularity in the way data structures are
  - allocated in memory AND
  - traversed

- Two types of loads can have stable AVDs
  - Traversal address loads
    - Produce addresses consumed by **address loads**
  - Leaf address loads
    - Produce addresses consumed by **data loads**
Traversing Address Loads

Regularly-allocated linked list:

A traversal address load loads the pointer to next node:

\[ \text{node} = \text{node} \rightarrow \text{next} \]

AVD = Effective Addr – Data Value

<table>
<thead>
<tr>
<th>Effective Addr</th>
<th>Data Value</th>
<th>AVD</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A+k</td>
<td>-k</td>
</tr>
<tr>
<td>A+k</td>
<td>A+2k</td>
<td>-k</td>
</tr>
<tr>
<td>A+2k</td>
<td>A+3k</td>
<td>-k</td>
</tr>
<tr>
<td>A+3k</td>
<td>A+4k</td>
<td>-k</td>
</tr>
<tr>
<td>A+4k</td>
<td>A+5k</td>
<td>-k</td>
</tr>
</tbody>
</table>

Striding data value

Stable AVD
Properties of Traversal-based AVDs

- Stable AVDs can be captured with a stride value predictor
- Stable AVDs disappear with the re-organization of the data structure (e.g., sorting)

![Diagram showing the effect of sorting on AVDs]

- Stability of AVDs is dependent on the behavior of the memory allocator
  - Allocation of contiguous, fixed-size chunks is useful
Leaf Address Loads

Sorted dictionary in **parser**: Nodes point to strings (words)
String and node allocated consecutively

Dictionary looked up for an input word.

A **leaf address load** loads the pointer to the string of each node:

```
lookup (node, input) {
    // ...
    ptr_str = node->string;
    m = check_match(ptr_str, input);
    if (m >= 0) lookup(node->right, input);
    if (m < 0) lookup(node->left, input);
}
```

**AVD = Effective Addr – Data Value**

<table>
<thead>
<tr>
<th>Effective Addr</th>
<th>Data Value</th>
<th>AVD</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+k</td>
<td>A</td>
<td>k</td>
</tr>
<tr>
<td>C+k</td>
<td>C</td>
<td>k</td>
</tr>
<tr>
<td>F+k</td>
<td>F</td>
<td>k</td>
</tr>
</tbody>
</table>

No stride!  Stable AVD

AVD Prediction
Properties of Leaf-based AVDs

- Stable AVDs **cannot** be captured with a stride value predictor
- Stable AVDs **do not** disappear with the re-organization of the data structure (e.g., sorting)

Stability of AVDs is dependent on the behavior of the memory allocator

Distance between node and string still constant!  

AVD Prediction
Talk Outline

- Background on Runahead Execution
- The Problem: Dependent Cache Misses
- AVD Prediction
- Why Does It Work?
- Evaluation
- Conclusions
Baseline Processor

- Execution-driven Alpha simulator
- 8-wide superscalar processor
- 128-entry instruction window, 20-stage pipeline
- 64 KB, 4-way, 2-cycle L1 data and instruction caches
- 1 MB, 32-way, 10-cycle unified L2 cache
- 500-cycle minimum main memory latency
- 32 DRAM banks, 32-byte wide processor-memory bus (4:1 frequency ratio), 128 outstanding misses
  - Detailed memory model

- Pointer-intensive benchmarks from Olden and SPEC INT00
Performance of AVD Prediction

A graph showing normalized execution time for various benchmarks with different entry counts: 4096 entries, 16 entries, and 4 entries.
Effect on Executed Instructions

Normalized Number of Executed Instructions

- bisort
- health
- mst
- perimeter
- treeadd
- tsp
- voronoi
- mcf
- parser
- twolf
- vpr
- AVG

13.3%

AVD Prediction
AVD Prediction vs. Stride Value Prediction

- Performance:
  - Both can capture traversal address loads with stable AVDs
    - e.g., treeadd
  - **Stride VP cannot capture** leaf address loads with stable AVDs
    - e.g., health, mst, parser
  - **AVD predictor cannot capture** data loads with striding data values
    - Predicting these can be useful for the correct resolution of mispredicted L2-miss dependent branches, e.g., parser

- Complexity:
  - AVD predictor requires much fewer entries (only address loads)
  - AVD prediction logic is simpler (no stride maintenance)
AVD vs. Stride VP Performance

Normalized Execution Time (excluding health)

16 entries

- AVD: 5.1%
- stride: 2.7%
- hybrid: 6.5%

4096 entries

- AVD: 5.5%
- stride: 4.7%
- hybrid: 8.6%
Conclusions

- Runahead execution is unable to parallelize dependent L2 cache misses

- A very simple, 16-entry (102-byte) AVD predictor reduces this limitation on pointer-intensive applications
  - Increases runahead execution performance by 12.1%
  - Reduces executed instructions by 13.3%

- AVD prediction takes advantage of the regularity in the memory allocation patterns of programs

- Software (programs, compilers, memory allocators) can be written to take advantage of AVD prediction
Backup Slides
The Potential: What if it Could?

Normalized Execution Time

- no runahead
- runahead
- ideal runahead

AVG Prediction
Effect of Confidence Threshold

![Bar chart showing the effect of confidence threshold on normalized execution time for various benchmarks. The chart compares different configurations (conf 1 to conf 7) with and without confidence (no conf). The benchmarks include bisort, health, mst, perimeter, treeadd, tsp, voronoi, mcf, parser, twolf, vpr, and AVG. The y-axis represents the normalized execution time, ranging from 0.0 to 1.5. The x-axis lists the benchmarks. The bar heights indicate the time savings achieved by different configurations. Notable configurations include conf 1 and conf 7.]
Effect of MaxAVD

![Bar chart showing normalized execution time for different MaxAVD values and applications. The chart compares bisort, health, mst, perimeter, treeadd, tsp, voronoi, mcf, parser, twolf, vpr, and AVG across different MaxAVD values (1M, 64K, 8K, 1K, 64, 64).]
Effect of Memory Latency

No Runahead
Runahead
AVD (16-entry)

Normalized Execution Time

1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0

100 250 500 750 1000

8% 9.3% 12.1% 13% 13.5%