Memory Latency Tolerance
Readings on Memory Latency Tolerance

Required

Optional
Remember: Latency Tolerance

- An out-of-order execution processor tolerates latency of multi-cycle operations by executing independent instructions concurrently
  - It does so by buffering instructions in reservation stations and reorder buffer
  - Instruction window: Hardware resources needed to buffer all decoded but not yet retired/committed instructions

- What if an instruction takes 500 cycles?
  - How large of an instruction window do we need to continue decoding?
  - How many cycles of latency can OoO tolerate?
Stalls due to Long-Latency Instructions

- When a long-latency instruction is not complete, it blocks instruction retirement.
  - Because we need to maintain precise exceptions

- Incoming instructions fill the instruction window (reorder buffer, reservation stations).

- Once the window is full, processor cannot place new instructions into the window.
  - This is called a full-window stall.

- A full-window stall prevents the processor from making progress in the execution of the program.
Full-window Stall Example

8-entry instruction window:

- LOAD R1 ← mem[R5]
- BEQ R1, R0, target
- ADD R2 ← R2, 8
- LOAD R3 ← mem[R2]
- MUL R4 ← R4, R3
- ADD R4 ← R4, R5
- STOR mem[R2] ← R4
- ADD R2 ← R2, 64

L2 Miss! Takes 100s of cycles.

Independent of the L2 miss, executed out of program order, but cannot be retired.

Younger instructions cannot be executed because there is no space in the instruction window.

The processor stalls until the L2 Miss is serviced.

- Long-latency cache misses are responsible for most full-window stalls.
Cache Misses Responsible for Many Stalls

512KB L2 cache, 500-cycle DRAM latency, aggressive stream-based prefetcher
Data averaged over 147 memory-intensive benchmarks on a high-end x86 processor model
The Memory Latency Problem

- **Problem:** Memory latency is long
- And, it is not very easy to reduce it...
  - We examined many methods for reducing DRAM latency
    - ...
  - See Lecture 10: Low-Latency Memory
    - [https://www.youtube.com/watch?v=vQd1YgOH1Mw](https://www.youtube.com/watch?v=vQd1YgOH1Mw)
- And, even if we reduce memory latency, it is still long
  - Remember the fundamental capacity-latency tradeoff
  - Contention for memory increases latencies
How Do We Tolerate Stalls Due to Memory?

- Two major approaches
  - Reduce/eliminate stalls
  - Tolerate the effect of a stall when it happens

- Four fundamental techniques to achieve these
  - Caching
  - Prefetching
  - Multithreading
  - Out-of-order execution

- Many techniques have been developed to make these four fundamental techniques more effective in tolerating memory latency
Memory Latency Tolerance Techniques

- **Caching** [initially by Bloom+, 1962 and later Wilkes, 1965]
  - Widely used, simple, effective, but inefficient, passive
  - Not all applications/phases exhibit temporal or spatial locality

- **Prefetching** [initially in IBM 360/91, 1967]
  - Works well for regular memory access patterns
  - Prefetching irregular access patterns is difficult, inaccurate, and hardware-intensive

- **Multithreading** [initially in CDC 6600, 1964]
  - Works well if there are multiple threads
  - Improving single thread performance using multithreading hardware is an ongoing research effort

- **Out-of-order execution** [initially by Tomasulo, 1967]
  - Tolerates irregular cache misses that cannot be prefetched
  - Requires extensive hardware resources for tolerating long latencies
  - **Runahead execution** alleviates this problem (as we will see today)
Runahead Execution
Small Windows: Full-window Stalls

8-entry instruction window:

- LOAD R1 ← mem[R5]
- BEQ R1, R0, target
- ADD R2 ← R2, 8
- LOAD R3 ← mem[R2]
- MUL R4 ← R4, R3
- ADD R4 ← R4, R5
- STOR mem[R2] ← R4
- ADD R2 ← R2, 64

L2 Miss! Takes 100s of cycles.

Independent of the L2 miss, executed out of program order, but cannot be retired.

Younger instructions cannot be executed because there is no space in the instruction window.

The processor stalls until the L2 Miss is serviced.

Long-latency cache misses are responsible for most full-window stalls.
Impact of Long-Latency Cache Misses

512KB L2 cache, 500-cycle DRAM latency, aggressive stream-based prefetcher
Data averaged over 147 memory-intensive benchmarks on a high-end x86 processor model
Impact of Long-Latency Cache Misses

500-cycle DRAM latency, aggressive stream-based prefetcher
Data averaged over 147 memory-intensive benchmarks on a high-end x86 processor model
The Problem

- Out-of-order execution requires large instruction windows to tolerate today’s main memory latencies.

- As main memory latency increases, instruction window size should also increase to fully tolerate the memory latency.

- Building a large instruction window is a challenging task if we would like to achieve
  - Low power/energy consumption (tag matching logic, ld/st buffers)
  - Short cycle time (access, wakeup/select latencies)
  - Low design and verification complexity
Efficient Scaling of Instruction Window Size

- One of the major research issues in out of order execution

- How to achieve the benefits of a large window with a small one (or in a simpler way)?

- How do we efficiently tolerate memory latency with the machinery of out-of-order execution (and a small instruction window)?
Memory Level Parallelism (MLP)

- Idea: Find and service multiple cache misses in parallel so that the processor stalls only once for all misses

- Enables latency tolerance: overlaps latency of different misses

- How to generate multiple misses?
  - Out-of-order execution, multithreading, prefetching, runahead
Runahead Execution (I)

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

- When the oldest instruction is a long-latency cache miss:
  - Checkpoint architectural state and enter runahead mode

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

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

**Perfect Caches:**
Load 1 Hit
Load 2 Hit

**Small Window:**
Load 1 Miss
Load 2 Miss

**Runahead:**
Load 1 Miss
Load 2 Miss
Load 1 Hit
Load 2 Hit

Saved Cycles
Benefits of Runahead Execution

Instead of stalling during an L2 cache miss:

- Pre-executed loads and stores independent of L2-miss instructions generate very accurate data prefetches:
  - For both regular and irregular access patterns

- Instructions on the predicted program path are prefetched into the instruction/trace cache and L2.

- Hardware prefetcher and branch predictor tables are trained using future access information.
Runahead Execution Mechanism

- Entry into runahead mode
  - Checkpoint architectural register state

- Instruction processing in runahead mode

- Exit from runahead mode
  - Restore architectural register state from checkpoint
Runahead mode processing is the same as normal instruction processing, EXCEPT:

- It is purely speculative: Architectural (software-visible) register/memory state is NOT updated in runahead mode.

- L2-miss dependent instructions are identified and treated specially.
  - They are quickly removed from the instruction window.
  - Their results are not trusted.
L2-Miss Dependent Instructions

- Load 1 Miss

Two types of results produced: INV and VALID

- INV = Dependent on an L2 miss

- INV results are marked using INV bits in the register file and store buffer.

- INV values are not used for prefetching/branch resolution.
Removal of Instructions from Window

Load 1 Miss

- Oldest instruction is examined for **pseudo-retirement**
  - An INV instruction is removed from window immediately.
  - A VALID instruction is removed when it completes execution.

- Pseudo-retired instructions free their allocated resources.
  - This allows the processing of later instructions.

- Pseudo-retired stores communicate their data to dependent loads.
A pseudo-retired store writes its data and INV status to a dedicated memory, called runahead cache.

Purpose: Data communication through memory in runahead mode.

A dependent load reads its data from the runahead cache.

Does not need to be always correct → Size of runahead cache is very small.
Branch Handling in Runahead Mode

- **INV branches cannot be resolved.**
  - A mispredicted INV branch causes the processor to stay on the wrong program path until the end of runahead execution.

- **VALID branches are resolved and initiate recovery if mispredicted.**
A Runahead Processor Diagram

Runahead Execution Pros and Cons

- **Advantages:**
  + Very *accurate* prefetches for data/instructions (all cache levels)
    + Follows the program path
  + *Simple to implement*, most of the hardware is already built in
  + Versus other pre-execution based prefetching mechanisms (as we will see):
    + Uses the same thread context as main thread, no waste of context
    + No need to construct a pre-execution thread

- **Disadvantages/Limitations:**
  -- *Extra executed instructions*
  -- Limited by branch prediction accuracy
  -- Cannot prefetch dependent cache misses
  -- Effectiveness limited by available “memory-level parallelism” (MLP)
  -- Prefetch distance (how far ahead to prefetch) limited by memory latency

- Implemented in IBM POWER6, Sun “Rock”
Performance of Runahead Execution

The diagram compares the performance of various execution strategies across different benchmarks. It shows the percentage increase in micro-operations per cycle compared to a baseline. The strategies compared are:

- No prefetcher, no runahead
- Only prefetcher (baseline)
- Only runahead
- Prefetcher + runahead

The benchmarks are labeled as S95, FP00, INT00, WEB, MM, PROD, SERV, WS, and AVG. The graph visually represents the performance improvements for each strategy across these benchmarks.
Runahead Execution vs. Large Windows

The graph compares the performance of different window sizes with and without Runahead execution. The x-axis represents different workloads (S95, FP00, INT00, WEB, MM, PROD, SERV, WS, AVG), and the y-axis represents micro-operations per cycle. The graph shows that Runahead execution significantly improves performance across all window sizes compared to baseline performance.
Runahead vs. A (Real) Large Window

- When is one beneficial, when is the other?
- Pros and cons of each

- Which can tolerate floating-point operation latencies better?
- Which leads to less wasted execution?
Runahead on In-order vs. Out-of-order
Effect of Runahead in Sun ROCK

- Shailender Chaudhry talk, Aug 2008.

![Graph showing normalized IPC vs L2 Cache Size with labels indicating benefits of different cache sizes.](image-url)

- Buys 7 MB
- Buys 12 MB
- 40% Better Performance

Lines represent:
- Scout
- No Scout
Generalizing the Idea

- Runahead on different long-latency operations?
More on Runahead Execution

- Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors"


One of the 15 computer architecture papers of 2003 selected as Top Picks by IEEE Micro.

Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors

Onur Mutlu §  Jared Stark †  Chris Wilkerson ‡  Yale N. Patt §

§ECE Department
The University of Texas at Austin
{onur,patt}@ece.utexas.edu

†Microprocessor Research
Intel Labs
jared.w.stark@intel.com

‡Desktop Platforms Group
Intel Corporation
chris.wilkerson@intel.com
More on Runahead Execution (Short)


Runahead Execution: An Effective Alternative to Large Instruction Windows
Runahead Enhancements
Readings

- Required

- Recommended
Limitations of the Baseline Runahead Mechanism

- **Energy Inefficiency**
  - A large number of instructions are speculatively executed
  - Efficient Runahead Execution [ISCA’ 05, IEEE Micro Top Picks’ 06]

- **Ineffectiveness for pointer-intensive applications**
  - Runahead cannot parallelize dependent L2 cache misses
  - Address-Value Delta (AVD) Prediction [MICRO’ 05]

- **Irresolvable branch mispredictions in runahead mode**
  - Cannot recover from a mispredicted L2-miss dependent branch
  - Wrong Path Events [MICRO’ 04]
The Efficiency Problem

% Increase in IPC
% Increase in Executed Instructions

bzip2: 235%
crafty: 22%
eon: 27%
gcc: 10%
gzip: 10%
mcf: 10%
parser: 10%
perlbmk: 10%
twolf: 10%
vortex: 10%
vr: 10%
ammp: 10%
applu: 10%
apsi: 10%
art: 10%
equake: 10%
facerec: 10%
fma3d: 10%
galgel: 10%
lucas: 10%
mesa: 10%
mgrid: 10%
sixtrack: 10%
swim: 10%
wupwise: 10%
AVG: 10%
Causes of Inefficiency

- Short runahead periods
- Overlapping runahead periods
- Useless runahead periods

Short Runahead Periods

- Processor can initiate runahead mode due to an already in-flight L2 miss generated by
  - the prefetcher, wrong-path, or a previous runahead period

- Short periods
  - are less likely to generate useful L2 misses
  - have high overhead due to the flush penalty at runahead exit
Overlapping Runahead Periods

- Two runahead periods that execute the same instructions

- Second period is inefficient
Useless Runahead Periods

- Periods that do not result in prefetches for normal mode

- They exist due to the lack of memory-level parallelism

Mechanism to eliminate useless periods:
- Predict if a period will generate useful L2 misses
- Estimate a period to be useful if it generated an L2 miss that cannot be captured by the instruction window
  - Useless period predictors are trained based on this estimation
Overall Impact on Executed Instructions

Increase in Executed Instructions

- baseline runahead
- all techniques

AVG

- bzip2
- crafty
- eon
- gap
- gcc
- gzip
- mcf
- parser
- perlbmk
- twolf
- vortex
- vpr
- ammp
- applu
- api2
- art
- equake
- facerec
- fma3d
- galgel
- Lucas
- mesa
- mgrid
- sixtrack
- swim
- wupwise

Baseline Runahead: 235%
All Techniques: 26.5%

Overall Impact on Executed Instructions:
Baseline Runahead: 6.2%
All Techniques: 26.5%
Overall Impact on IPC

Increase in IPC

- baseline runahead
- all techniques

Average increase in IPC:
- 22.6%
- 22.1%
More on Efficient Runahead Execution


One of the 13 computer architecture papers of 2005 selected as Top Picks by IEEE Micro.

Techniques for Efficient Processing in Runahead Execution Engines

Onur Mutlu  Hyesoon Kim  Yale N. Patt

Department of Electrical and Computer Engineering
University of Texas at Austin
{onur,hyesoon,patt}@ece.utexas.edu
More on Efficient Runahead Execution


Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance
Taking Advantage of Pure Speculation

- Runahead mode is purely speculative

- The goal is to find and generate cache misses that would otherwise stall execution later on

- How do we achieve this goal most efficiently and with the highest benefit?

- Idea: Find and execute only those instructions that will lead to cache misses (that cannot already be captured by the instruction window)

- How?
Limitations of the Baseline Runahead Mechanism

- **Energy Inefficiency**
  - A large number of instructions are speculatively executed
  - **Efficient Runahead Execution** [ISCA’ 05, IEEE Micro Top Picks’ 06]

- **Ineffectiveness for pointer-intensive applications**
  - Runahead cannot parallelize dependent L2 cache misses
  - **Address-Value Delta (AVD) Prediction** [MICRO’ 05]

- **Irresolvable branch mispredictions in runahead mode**
  - Cannot recover from a mispredicted L2-miss dependent branch
  - **Wrong Path Events** [MICRO’ 04]
The Problem: Dependent Cache Misses

Runahead: *Load 2 is dependent on Load 1*

- Runahead execution cannot parallelize dependent misses
  - wasted opportunity to improve performance
  - wasted energy (useless pre-execution)

- Runahead performance would improve by 25% if this limitation were ideally overcome
Parallelizing Dependent Cache Misses

- **Idea:** 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**

- **Read:**
Parallelizing Dependent Cache Misses

**Cannot Compute Its Address!**

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

**Value Predicted**

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

**Can Compute Its Address**

Saved Speculative Instructions

Saved Cycles
AVD Prediction [MICRO’05]

- 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
  \[ \text{Predicted Value} = \text{Effective Address} - \text{Predicted AVD} \]
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**
**Traversal Address Loads**

Regularly-allocated linked list:

- A
- A + k
- A + 2k
- A + 3k
- ...

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>
</tbody>
</table>

Striding data value

Stable AVD
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);
    // …
}
```

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
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 for prediction
  - No need to value- predict the loads that will not generate addresses
  - Enables the 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**

![Diagram showing AVD update logic](image)

- **Effective Address**
- **Data Value**

- Computed AVD = Effective Addr - Data Value

- **Confidence Update/Reset Logic**

- **Tag**
- **Conf**
- **AVD**

- **PC of Retired Load**
AVD Prediction Logic
Performance of AVD Prediction

Normalized Execution Time and Executed Instructions

- Execution Time
- Executed Instructions

Normalized execution time for various programs:
- bisort
- health
- mst
- perimeter
- treeadd
- tsp
- voronoi
- mcf
- parser
- twolf
- vpr
- AVG

Runahead performance comparison:
- 14.3%
- 15.5%
More on AVD Prediction


Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns

Onur Mutlu  Hyesoon Kim  Yale N. Patt

Department of Electrical and Computer Engineering
University of Texas at Austin
{onur,hyesoon,patt}@ece.utexas.edu
More on AVD Prediction (II)


Address-Value Delta (AVD) Prediction: A Hardware Technique for Efficiently Parallelizing Dependent Cache Misses

Onur Mutlu, Member, IEEE, Hyesoon Kim, Student Member, IEEE, and Yale N. Patt, Fellow, IEEE
Wrong Path Events
An Observation and A Question

• In an out-of-order processor, some instructions are executed on the mispredicted path (wrong-path instructions).

• Is the behavior of wrong-path instructions different from the behavior of correct-path instructions?
  – If so, we can use the difference in behavior for early misprediction detection and recovery.
What is a Wrong Path Event?

An instance of illegal or unusual behavior that is more likely to occur on the wrong path than on the correct path.

Wrong Path Event = WPE

Probability (wrong path | WPE) \sim 1
Why Does a WPE Occur?

• A wrong-path instruction may be executed before the mispredicted branch is executed.
  – Because the mispredicted branch may be dependent on a long-latency instruction.

• The wrong-path instruction may consume a data value that is not properly initialized.
WPE Example from *eon*: NULL pointer dereference

1:    for (int i=0 ; i< length(); i++) {
2:        structure *ptr = array[i];
3:        if (ptr->x) {
4:            // . . .
5:        }
6:    }
Beginning of the loop

Array boundary

i = 0

Array of pointers to structs

x8ABCD0  xEFF8B0  x0  x0

1:   for (int i=0; i< length(); i++) {
2:       structure *ptr = array[i];
3:       if (ptr->x) {
4:           // . . .
5:       }
6:   }
First iteration

Array boundary

Array of pointers to structs

i = 0
ptr = x8ABCD0

1: for (int i=0; i< length(); i++) {
2:     structure *ptr = array[i];
3:     if (ptr->x) {
4:         // ...
5:     }
6: }

for (int i = 0; i < length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        // ...
    }
}
```c
for (int i=0; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        // . . .
    }
}
```
Loop branch correctly predicted

Array of pointers to structs

Array boundary

i = 1

1: for (int i=0; i< length(); i++) {
2:     structure *ptr = array[i];
3:     if (ptr->x) {
4:         // . . .
5:     }
6: }

1  :   for (int i=0 ; i< length(); i++) {
2  :         structure *ptr = array[i];
3  :         if (ptr->x) {
4  :               // . . .
5  :         }
6  :   }
Array boundary

i = 1
ptr = xEFF8B0

Second iteration

Array of pointers to structs

1 : for (int i=0 ; i< length(); i++) {
2 :     structure *ptr = array[i];
3 :     if (ptr->x) {
4 :         // . . .
5 :     }
6 : }
Second iteration

Array boundary

Array of pointers to structs

i = 1
ptr = xEFF8B0

for (int i=0 ; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        // ...
    }
}
Loop exit branch mispredicted

Array boundary

Array of pointers to structs

x8ABCD0  xEFF8B0  x0  x0

1: for (int i=0; i< length(); i++) {
2:     structure *ptr = array[i];
3:     if (ptr->x) {
4:         // . . .
5:     }
6: }

Third iteration on wrong path

```c
1 : for (int i=0; i< length(); i++) {
2 :     structure *ptr = array[i];
3 :     if (ptr->x) {
4 :         // . . .
5 :     }
6 : }
```
Wrong Path Event

Array of pointers to structs

Array boundary

i = 2
ptr = 0

for (int i=0 ; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        // ...
    }
}

NULL pointer dereference!
Types of WPEs

• Due to memory instructions
  – NULL pointer dereference
  – Write to read-only page
  – Unaligned access (illegal in the Alpha ISA)
  – Access to an address out of segment range
  – Data access to code segment
  – Multiple concurrent TLB misses
Types of WPEs (continued)

• Due to control-flow instructions
  – Misprediction under misprediction
    • If three branches are executed and resolved as mispredicts while there are older unresolved branches in the processor, it is almost certain that one of the older unresolved branches is mispredicted.
  – Return address stack underflow
  – Unaligned instruction fetch address (illegal in Alpha)

• Due to arithmetic instructions
  – Some arithmetic exceptions
    • e.g. Divide by zero
Two Empirical Questions

1. How often do WPEs occur?

2. When do WPEs occur on the wrong path?
More on Wrong Path Events


Wrong Path Events: Exploiting Unusual and Illegal Program Behavior for Early Misprediction Detection and Recovery

David N. Armstrong  Hyesoon Kim  Onur Mutlu  Yale N. Patt

Department of Electrical and Computer Engineering
The University of Texas at Austin
{dna,hyesoon,onur,patt}@ece.utexas.edu
Why Is This Important?

- A modern processor spends significant amount of time fetching/executing instructions on the wrong path.

Fig. 1. Percentage of fetch cycles spent on the wrong path, percentage of instructions fetched on the wrong path, and percentage of instructions (memory and nonmemory) executed on the wrong path in the baseline processor for SPEC 2000 integer benchmarks.
A Lot of Time Spent on The Wrong Path

- A runahead processor, much more so...

Fig. 20. Percentage of total cycles spent on the wrong path, percentage of instructions fetched on the wrong path, and percentage of instructions (memory and nonmemory) executed on the wrong path in the runahead processor.
4 Wrong Path: To Model or Not to Model

In this section, we measure the error in IPC if wrong-path memory references are not simulated. We also evaluate the overall effect of wrong-path memory references on the IPC (retired Instructions Per Cycle) performance of a processor.

1. How important is it to correctly model wrong-path memory references? What is the error in the predicted performance if wrong-path references are not modeled?

2. Do wrong-path memory references affect performance positively or negatively? What is the relative significance on performance of prefetching, bandwidth consumption, and pollution caused by wrong-path references?

3. What kind of code structures lead to the positive effects of wrong-path memory references?

4. How do wrong-path memory references affect the performance of a runahead execution processor [7], [18] which implements an aggressive form of speculative execution?
Wrong Path Is Often Useful for Performance

Fig. 7. Error in the IPC of the baseline processor with a stream prefetcher for three different memory latencies if wrong-path memory references are not simulated.
More So In Runahead Execution

Fig. 19. IPC improvement of adding runahead execution to the baseline processor if wrong-path memory references are or are not modeled.
Why is Wrong Path Useful? (I)

- Control-independence: e.g., wrong-path execution of future loop iterations

```c
1:   arc_t *arc; // array of arc_t structures
2:   // initialize arc (arc = ...)
3: 
4:   for ( ; arc < stop_arcs; arc += size) {
5:     if (arc->ident > 0) { // frequently mispredicted br.
6:       // function calls and
7:       // operations on the structure pointed to by arc
8:       // ...
9:     }
10: } 
```

Fig. 16. An example from mcf showing wrong-path prefetching for later loop iterations.
Why is Wrong Path Useful? (II)

Fig. 17. An example from mcf showing wrong-path prefetching between different loops.
Why is Wrong Path Useful? (III)

- Same data used in different control flow paths

```c
1:   node_t *node;
2:   // initialize node
3:   // ...
4:   
5:   while (node) {
6:     if (node->orientation == UP) { // mispredicted branch
7:         node->potential = node->basic_arc->cost
8:         + node->pred->potential;
9:     } else { // == DOWN */
10:        node->potential = node->pred->potential
11:        - node->basic_arc->cost;
12:        // ...
13:     }
14:   }
15:   // control-flow independent point (re-convergent point)
16:   node = node->child;
17: }
```

Fig. 18. An example from mcf showing wrong-path prefetching in control-flow hammocks.
More on Wrong Path Execution (I)

- Onur Mutlu, Hyesoon Kim, David N. Armstrong, and Yale N. Patt, "Understanding the Effects of Wrong-Path Memory References on Processor Performance". Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI), pages 56-64, Munchen, Germany, June 2004. Slides (pdf)

Understanding The Effects of Wrong-Path Memory References on Processor Performance

Onur Mutlu  Hyesoon Kim  David N. Armstrong  Yale N. Patt

Department of Electrical and Computer Engineering
The University of Texas at Austin
{onur,hyesoon,dna,patt}@ece.utexas.edu

An Analysis of the Performance Impact of Wrong-Path Memory References on Out-of-Order and Runahead Execution Processors

Onur Mutlu, *Student Member, IEEE*, Hyesoon Kim, *Student Member, IEEE*, David N. Armstrong, and Yale N. Patt, *Fellow, IEEE*
What If …

- The system learned from wrong-path execution and used that learning for better execution of the program/system?

- An open research problem...
More on Runahead Enhancements
Eliminating Short Periods

- Mechanism to eliminate short periods:
  - Record the number of cycles $C$ an L2-miss has been in flight
  - If $C$ is greater than a threshold $T$ for an L2 miss, disable entry into runahead mode due to that miss
  - $T$ can be determined statically (at design time) or dynamically

- $T=400$ for a minimum main memory latency of 500 cycles works well
Eliminating Overlapping Periods

- Overlapping periods are not necessarily useless
  - The availability of a new data value can result in the generation of useful L2 misses
- But, this does not happen often enough

- Mechanism to eliminate overlapping periods:
  - Keep track of the number of pseudo-retired instructions $R$ during a runahead period
  - Keep track of the number of fetched instructions $N$ since the exit from last runahead period
  - If $N < R$, do not enter runahead mode
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 of AVD traversal with labels A, A+k, A+2k, A+3k before and after sorting.](image)

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

Distance between nodes **NOT constant**!
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

![Diagram showing the stability of AVDs after sorting](attachment:diagram.png)

- Distance between node and string still constant! ✓

AVD Prediction 100