15-740/18-740
Computer Architecture
Lecture 15: Efficient Runahead Execution

Prof. Onur Mutlu
Carnegie Mellon University
Fall 2011, 10/14/2011
Review Set

- Due next Wednesday

- Recommended:
  - Hennessy and Patterson, Appendix C.2 and C.3
Announcements

- Milestone I
  - Due this Friday (Oct 14)
  - Format: 2-pages
  - Include results from your initial evaluations. We need to see good progress.
  - Of course, the results need to make sense (i.e., you should be able to explain them)

- Midterm I
  - October 24

- Milestone II
  - Will be postponed. Stay tuned.
Course Feedback

- I have read them
- Fill out the form and return, if you have not done so
Last Lecture

- More issues in load related instruction scheduling
- Better utilizing the instruction window
- Runahead execution
Today

- More on runahead execution
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)?

  - **Runahead execution?**
    - Upon L2 miss, checkpoint architectural state, speculatively execute only for prefetching, re-execute when data ready

  - **Continual flow pipelines?**
    - Upon L2 miss, deallocate everything belonging to an L2 miss dependent, reallocate/re-re-name and re-execute upon data ready

  - **Dual-core execution?**
    - One core runs ahead and does not stall on L2 misses, feeds another core that commits instructions
Runahead Example

**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 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:
    + 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. Solution?
  -- Effectiveness limited by available “memory-level parallelism” (MLP)
  -- Prefetch distance limited by memory latency

- Implemented in IBM POWER6, Sun “Rock”
Memory Level Parallelism (MLP)

- Idea: Find and service multiple cache misses in parallel

- Why generate multiple misses?
  - Enables latency tolerance: overlaps latency of different misses

- How to generate multiple misses?
  - Out-of-order execution, multithreading, runahead, prefetching
Memory Latency Tolerance Techniques

- Caching [initially by 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 cache misses that cannot be prefetched
  - Requires extensive hardware resources for tolerating long latencies
Performance of Runahead Execution

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

Micro-operations Per Cycle

S95 | FP00 | INT00 | WEB | MM | PROD | SERV | WS | AVG

- No prefetcher, no runahead: 12%
- Only prefetcher (baseline): 35%
- Only runahead: 15%
- Prefetcher + runahead: 22% 12% 22% 16% 52%
Runahead Execution vs. Large Windows

- 128-entry window (baseline)
- 128-entry window with Runahead
- 256-entry window
- 384-entry window
- 512-entry window

Micro-operations Per Cycle

<table>
<thead>
<tr>
<th></th>
<th>S95</th>
<th>FP00</th>
<th>INT00</th>
<th>WEB</th>
<th>MM</th>
<th>PROD</th>
<th>SERV</th>
<th>WS</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Runahead</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256-entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>384-entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>512-entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Runahead vs. A Real Large Window

- When is one beneficial, when is the other?
- Pros and cons of each
Runahead on In-order vs. Out-of-order

Micro-operations Per Cycle

- S95
- FP00
- INT00
- WEB
- MM
- PROD
- SERV
- WS
- AVG

- in-order baseline
- in-order + runahead
- out-of-order baseline
- out-of-order + runahead
Runahead vs. Large Windows (Alpha)
In-order vs. Out-of-order Execution (Alpha)
Sun ROCK Cores

- Load miss in L1 cache starts parallelization using 2 HW threads
- Ahead thread
  - Checkpoints state and executes speculatively
  - Instructions independent of load miss are speculatively executed
  - Load miss(es) and dependent instructions are deferred to behind thread
- Behind thread
  - Executes deferred instructions and re-defers them if necessary

- Memory-Level Parallelism (MLP)
  - Run ahead on load miss and generate additional load misses

- Instruction-Level Parallelism (ILP)
  - Ahead and behind threads execute independent instructions from different points in program in parallel
ROCK Pipeline
Effect of Runahead in Sun ROCK

- Chaudhry talk, Aug 2008.

![Diagram showing the effect of Runahead on normalized IPC across different L2 cache sizes. The diagram illustrates that as the cache size increases, the normalized IPC also increases, with a significant improvement when running with Scout compared to No Scout. At 12 MB, the performance gains are noted as 40% Better Performance.]
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, IEEE TC’06]

- **Irresolvable branch mispredictions in runahead mode**
  - Cannot recover from a mispredicted L2-miss dependent branch
  - Wrong Path Events [MICRO’ 04]
The Efficiency Problem [ISCA’05]

- A runahead processor pre-executes some instructions speculatively
- Each pre-executed instruction consumes energy
- Runahead execution significantly increases the number of executed instructions, sometimes without providing performance improvement
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
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
Overlapping Runahead Periods

- Two runahead periods that execute the same instructions

- Second period is inefficient
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
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

- **bzip2**: 235%
- **art**: 26.5%
- **AVG**: 6.2%

**Graph Details**
- **Y-axis**: Increase in Executed Instructions
- **X-axis**: Various benchmarks
- **Legend**:
  - baseline runahead
  - all techniques
Overall Impact on IPC

Increase in IPC

- baseline runahead
- all techniques

AVG: 22.6%

22.1%
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: \textit{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
The Goal of AVD Prediction

- 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**
Parallelizing Dependent Cache Misses

Cannot Compute Its Address!

Value Predicted

Can Compute Its Address

Saved Speculative Instructions

Saved Cycles
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

[Predicted Value] = [Effective Address] - [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

<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>

A traversal address load loads the pointer to next node:

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

AVD = Effective Addr – Data Value

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).
- 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! ❌
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:

```c
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
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!
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?

Confidence Update/Reset Logic

valid AVD?

PC of Retired Load

<table>
<thead>
<tr>
<th>Tag</th>
<th>Conf</th>
<th>AVD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
AVD Prediction Logic
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

Normalized Execution Time and Executed Instructions

- Execution Time
- Executed Instructions

-runahead

14.3%
15.5%
AVD vs. Stride VP Performance

- AVD: 5.1% for 16 entries, 5.5% for 4096 entries
- Stride: 2.7% for 16 entries, 4.7% for 4096 entries
- Hybrid: 6.5% for 16 entries, 8.6% for 4096 entries

Normalized Execution Time (excluding health)