Reminder: Lab Assignments

- Lab Assignment 6
  - Implementing a more realistic memory hierarchy
    - L2 cache model
    - DRAM, memory controller models
    - MSHRs, multiple outstanding misses
  - Due April 23
  - Extra credit: Prefetching
Last Lecture

- Memory latency tolerance/reduction
  - Stalls
  - Four fundamental techniques
  - Software and hardware prefetching
  - Prefetcher throttling
Today

- More prefetching
- Runahead execution
Readings

Tolerating Memory Latency
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
Prefetching
Review: Prefetching: The Four Questions

- **What**
  - What addresses to prefetch

- **When**
  - When to initiate a prefetch request

- **Where**
  - Where to place the prefetched data

- **How**
  - Software, hardware, execution-based, cooperative
Review: Challenges in Prefetching: How

- **Software** prefetching
  - ISA provides prefetch instructions
  - Programmer or compiler inserts prefetch instructions (effort)
  - Usually works well only for “regular access patterns”

- **Hardware** prefetching
  - Hardware monitors processor accesses
  - Memorizes or finds patterns/strides
  - Generates prefetch addresses automatically

- **Execution-based** prefetchers
  - A “thread” is executed to prefetch data for the main program
  - Can be generated by either software/programmer or hardware
How to Prefetch More Irregular Access Patterns?

- Regular patterns: Stride, stream prefetchers do well
- More irregular access patterns
  - Indirect array accesses
  - Linked data structures
  - Multiple regular strides (1,2,3,1,2,3,1,2,3,...)
  - Random patterns?
  - Generalized prefetcher for all patterns?

- Correlation based prefetchers
- Content-directed prefetchers
- Precomputation or execution-based prefetchers
Markov Prefetching (I)

- Consider the following history of cache block addresses A, B, C, D, C, E, A, C, F, F, E, A, A, B, C, D, E, A, B, C, D, C
- After referencing a particular address (say A or E), are some addresses more likely to be referenced next
Markov Prefetching (II)

- **Idea:** Record the likely-next addresses (B, C, D) after seeing an address A
  - Next time A is accessed, prefetch B, C, D
  - A is said to be correlated with B, C, D

- Prefetch **accuracy** is generally low so prefetch up to N next addresses to increase **coverage**

- Prefetch accuracy can be improved by using multiple addresses as key for the next address: (A, B) → (C)
  - (A,B) correlated with C

Markov Prefetching (III)

- **Advantages:**
  - Can cover arbitrary access patterns
    - Linked data structures
    - Streaming patterns (though not so efficiently!)

- **Disadvantages:**
  - **Correlation table** needs to be very large for high coverage
    - Recording every miss address and its subsequent miss addresses is infeasible
  - **Low timeliness:** Lookahead is limited since a prefetch for the next access/miss is initiated right after previous
  - Consumes a lot of memory bandwidth
    - Especially when Markov model probabilities (correlations) are low
  - Cannot reduce compulsory misses
Content Directed Prefetching (I)

- A specialized prefetcher for pointer values
- Idea: Identify pointers among all values in a fetched cache block and issue prefetch requests for them.

+ No need to memorize/record past addresses!
+ Can eliminate compulsory misses (never-seen pointers)
-- Indiscriminately prefetches all pointers in a cache block

- How to identify pointer addresses:
  - Compare address sized values within cache block with cache block’s address → if most-significant few bits match, pointer
Content Directed Prefetching (II)

Virtual Address Predictor

Generate Prefetch

X80022220

x40373551

x80011100

[31:20]

[31:20]

[31:20]

[31:20]

[31:20]

[31:20]

L2

DRAM
Making Content Directed Prefetching Efficient

- Hardware does not have enough information on pointers
- Software does (and can profile to get more information)

Idea:
- Compiler profiles and provides hints as to which pointer addresses are likely-useful to prefetch.
- Hardware uses hints to prefetch only likely-useful pointers.

Shortcomings of CDP – An example

HashLookup(int Key) {
    ...
    for (node = head ; node -> Key != Key; node = node -> Next; ) ;
    if (node) return node->D1;
}

Struct node{
    int Key;
    int * D1_ptr;
    int * D2_ptr;
    node * Next;
}
Shortcomings of CDP – An example

<table>
<thead>
<tr>
<th>Cache Line Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
</tr>
<tr>
<td>[31:20]</td>
</tr>
</tbody>
</table>

Virtual Address Predictor

```plaintext
Key D1 D2
Key D1 D2
Key D1 D2
Key D1 D2
... ...
```

```plaintext
Key D1 D2
Key D1 D2
Key D1 D2
Key D1 D2
... ...
```
Shortcomings of CDP – An example

HashLookup(int Key) {
    ...
    for (node = head ; node -> Key != Key; node = node -> Next; ) ;
    if (node) return node -> D1;
}
Shortcomings of CDP – An example

Cache Line Addr

Virtual Address Predictor
Execution-based Prefetchers (I)

- **Idea:** Pre-execute a piece of the (pruned) program solely for prefetching data
  - Only need to distill pieces that lead to cache misses

- **Speculative thread:** Pre-executed program piece can be considered a “thread”

- Speculative thread can be executed
  - On a separate processor/core
  - On a separate hardware thread context (think fine-grained multithreading)
  - On the same thread context in idle cycles (during cache misses)
Execution-based Prefetchers (II)

- How to construct the speculative thread:
  - Software based pruning and “spawn” instructions
  - Hardware based pruning and “spawn” instructions
  - Use the original program (no construction), but
    - Execute it faster without stalling and correctness constraints

- Speculative thread
  - Needs to discover misses before the main program
    - Avoid waiting/stalling and/or compute less
  - To get ahead, uses
    - Perform only address generation computation, branch prediction, value prediction (to predict “unknown” values)
Thread-Based Pre-Execution

Thread-Based Pre-Execution Issues

- **Where to execute the precomputation thread?**
  1. Separate core (least contention with main thread)
  2. Separate thread context on the same core (more contention)
  3. Same core, same context

  - When the main thread is stalled

- **When to spawn the precomputation thread?**
  1. Insert spawn instructions well before the “problem” load

  - How far ahead?
    - Too early: prefetch might not be needed
    - Too late: prefetch might not be timely
  2. When the main thread is stalled

- **When to terminate the precomputation thread?**
  1. With pre-inserted CANCEL instructions
  2. Based on effectiveness/contention feedback
Thread-Based Pre-Execution Issues

- **Read**
  - Many issues in software-based pre-execution discussed
An Example

(a) Original Code

register int i;
register arc.t *arcout;
for( i < trips; ){
    // loop over “trips” lists
    if (arcout[1].ident != FIXED) {
        ...
        first_of_sparse_list = arcout + 1;
    }
    ...
    arcin = (arc.t *)first_of_sparse_list
        ->tail->mark;
    // traverse the list starting with
    // the first node just assigned
    while (arcin) {
        tail = arcin->tail;
        ...
        arcin = (arc.t *)tail->mark;
    }
    i++, arcout+=3;
}

(b) Code with Pre-Execution

register int i;
register arc.t *arcout;
for( i < trips; ){
    // loop over “trips” lists
    if (arcout[1].ident != FIXED) {
        ...
        first_of_sparse_list = arcout + 1;
    }
    ...
    // invoke a pre-execution starting
    // at END_FOR
    PreExecute_Start(END_FOR);
    arcin = (arc.t *)first_of_sparse_list
        ->tail->mark;
    // traverse the list starting with
    // the first node just assigned
    while (arcin) {
        tail = arcin->tail;
        ...
        arcin = (arc.t *)tail->mark;
    }
    // terminate this pre-execution after
    // prefetching the entire list
    PreExecute_Stop();
}

The Spec2000 benchmark mcf spends roughly half of its execution time in a nested loop which traverses a set of linked lists. An abstract version of this loop is shown in Figure 2(a), in which the for-loop iterates over the lists and the while-loop visits the elements of each list. As we observe from the figure, the first node of each list is assigned by dereferencing the pointer first_of_sparse_list, whose value is in fact determined by arcout, an induction variable of the for-loop. Therefore, even when we are still working on the current list, the first and the remaining nodes on the next list can be loaded speculatively by pre-executing the next iteration of the for-loop.

Figure 2(b) shows a version of the program with pre-execution code inserted (shown in boldface). END_FOR is simply a label to denote the place where arcout gets updated. The new instruction PreExecute_Start(END_FOR) initiates a pre-execution thread, say T, starting at the PC represented by END_FOR. Right after the pre-execution begins, T’s registers that hold the values of i and arcout will be updated. Then i’s value is compared against trips to see if we have reached the end of the for-loop. If so, thread T will exit the for-loop and encounters a PreExecute_Stop(), which will terminate the pre-execution and free up T for future use. Otherwise, T will continue pre-executing the body of the for-loop, and hence compute the first node of the next list automatically. Finally, after traversing the entire list through the while-loop, the pre-execution will be terminated by another PreExecute_Stop(). Notice that any PreExecute_Start() instructions encountered during pre-execution are simply ignored as we do not allow nested pre-execution in order to keep our design simple. Similarly, PreExecute_Stop() instructions cannot terminate the main thread either.
Example ISA Extensions

\[ \text{Thread}_{\text{ID}} = \text{PreExecute\_Start}(\text{Start\_PC}, \text{Max\_Insts}) : \]
Request for an idle context to start pre-execution at \text{Start\_PC} and stop when \text{Max\_Insts} instructions have been executed; \text{Thread}_{\text{ID}} holds either the identity of the pre-execution thread or -1 if there is no idle context. This instruction has effect only if it is executed by the main thread.

\text{PreExecute\_Stop}() : The thread that executes this instruction will be self terminated if it is a pre-execution thread; no effect otherwise.

\text{PreExecute\_Cancel}(\text{Thread\_ID}) : Terminate the pre-execution thread with \text{Thread\_ID}. This instruction has effect only if it is executed by the main thread.

Figure 4. Proposed instruction set extensions to support pre-execution. (C syntax is used to improve readability.)
Results on an SMT Processor
Problem Instructions


![Figure 2. Example problem instructions from heap insertion routine in vpr.](image)
Fork Point for Prefetching Thread

Figure 3. The `node_to_heap` function, which serves as the fork point for the slice that covers `add_to_heap`.

```c
void node_to_heap (... , float cost , ...) {
    struct s_heap *hptr;   // fork point
    ...
    hptr = alloc_heap_data();
    hptr->cost = cost;
    ...
    add_to_heap (hptr);
}
```
Pre-execution Slice Construction

Figure 4. Alpha assembly for the add_to_heap function. The instructions are annotated with the number of the line in Figure 2 to which they correspond. The problem instructions are in bold and the shaded instructions comprise the un-optimized slice.

```assembly
node_to_heap:
    ... /* skips ~40 instructions */
  1 lda s1, 252(gp)  # &heap_tail
  2 ld1 t2, 0(s1)    # ifrom = heap_tail
  3 ldq t5, -76(s1)  # &heap[0]
  4 cmplt t2, t0, t4 # see note
  5 addi t2, 0x1, t6 # heap_tail ++
  6 s8addq t2, t5, t3 # &heap[heap_tail]
  7 std t6, 0(s1)    # store heap_tail
  8 stq s0, 0(t3)    # heap[heap_tail]
  9 addi t2, t4, t4  # see note
 10 sra t4, 0x1, t4 # ito = ifrom/2
 11 ble t4, return   # (ito < 1)
 12 loop:
 13    s8addq t2, t5, a0 # &heap[ifrom]
 14    s8addq t4, t5, t7 # &heap[ito]
 15    cmplt t4, t0, t9 # see note
 16    move t4, t2     # ifrom = ito
 17    ldq a2, 0(a0)   # heap[ifrom]
 18    ldq a4, 0(t7)   # heap[ito]
 19    addi t4, t9, t9 # see note
 20    sra t9, 0x1, t4 # ito = ifrom/2
 21    lds $f0, 4(a2)  # heap[ifrom]->cost
 22    lds $f1, 4(a4)  # heap[ito]->cost
 23    cmpflt $f0,$f1,$f0 #  (heap[ifrom]->cost
 24   fbeq $f10, return #  < heap[ito]->cost)
 25    stq a2, 0(t7)   # heap[ito]
 26    stq a4, 0(a0)   # heap[ifrom]
 27    bgt t4, loop   # (ito > 1)
return:
    ... /* register restore code & return */

note: the divide by 2 operation is implemented by a 3 instruction sequence described in the strength reduction optimization.
```

Figure 5. Slice constructed for example problem instructions. Much smaller than the original code, the slice contains a loop that mimics the loop in the original code.

```
slice:
   1 ldq $6, 328(gp)  # &heap
t  slice_loop:
   2 ld1 $3, 252(gp)  # ito = heap_tail
   3,11 sra $3, 0x1, $3 # ito /= 2
   6 s8addq $3, $6, $16 # &heap[ito]
   7 ldq $18, 0($16)  # heap[ito]
   8 lds $f1, 4($18)  # heap[ito]->cost
   9 cmpflt $f1,$f17,$f31 #  (heap[ito]->cost
      #  < cost) PRED
      br slice_loop

    # Annotations
    fork: on first instruction of node_to_heap
live-in: $f17<cost>, gp
max loop iterations: 4
```
Runahead Execution (I)

- A simple pre-execution method for prefetching purposes

- 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

Runahead Execution (Mutlu et al., HPCA 2003)

Small Window:

Runahead:
Runahead Execution
Small Windows: Full-window Stalls

8-entry instruction window:

```
Oldest
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.

```
LOAD R3 ← mem[R2]
```

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

The processor stalls until the L2 Miss is serviced.

- L2 cache misses are responsible for most full-window stalls.
Impact of L2 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 L2 Cache Misses

Normalized Execution Time

- 128-entry window
- 2048-entry window

L2 Misses

Non-stall (compute) time

Full-window stall time

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)?
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, runahead, prefetching
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

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 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

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

- 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 $\rightarrow$ 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.**
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”

---
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
Performance of Runahead Execution

![Graph showing performance metrics for different scenarios: No prefetcher, no runahead, Only prefetcher (baseline), Only runahead, Prefetcher + runahead. The graph includes micro-operations per cycle for various categories: S95, FP00, INT00, WEB, MM, PROD, SERV, WS, and AVG.](image)
Runahead Execution vs. Large Windows

![Chart showing micro-operations per cycle for different window sizes and applications.]
Runahead on In-order vs. Out-of-order

- In-order baseline
- In-order + runahead
- Out-of-order baseline
- Out-of-order + runahead

Micro-operations Per Cycle

<table>
<thead>
<tr>
<th>Group</th>
<th>In-order baseline</th>
<th>In-order + runahead</th>
<th>Out-of-order baseline</th>
<th>Out-of-order + runahead</th>
</tr>
</thead>
<tbody>
<tr>
<td>S95</td>
<td>0.7</td>
<td>0.7</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>FP00</td>
<td>0.8</td>
<td>0.8</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>INT00</td>
<td>0.9</td>
<td>0.9</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>WEB</td>
<td>1.0</td>
<td>0.9</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>MM</td>
<td>1.1</td>
<td>1.1</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>PROD</td>
<td>1.2</td>
<td>1.2</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>SERV</td>
<td>1.3</td>
<td>1.3</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>WS</td>
<td>1.4</td>
<td>1.4</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>AVG</td>
<td>1.5</td>
<td>1.5</td>
<td>1.6</td>
<td>1.6</td>
</tr>
</tbody>
</table>

Percentages:
- S95: 15% 10%
- FP00: 73% 23%
- INT00: 17% 13%
- WEB: 20% 22%
- MM: 20% 22%
- PROD: 28% 15%
- SERV: 50% 47%
- WS: 39% 20%

Average (AVG):
- 1.5 1.5
- 1.5 1.5
Execution-based Prefetchers (III)

+ Can prefetch pretty much **any access pattern**
+ **Can be very low cost** (e.g., runahead execution)
  - Especially if it uses the same hardware context
  - Why? The processor is equipped to execute the program anyway
+ **Can be bandwidth-efficient** (e.g., runahead execution)

-- Depend on **branch prediction and possibly value prediction accuracy**
  - Mispredicted branches dependent on missing data throw the thread off the correct execution path

-- **Can be wasteful**
  -- speculatively execute many instructions
  -- can occupy a separate thread context