Announcements

- November 22:
  - Milestone 2.5 meetings with me and the TAs
  - 10 minutes per group
    - You will present latest progress briefly
    - We will ask questions as needed
  - Signup sheet will be posted online

- Milestone III – stay tuned
  - The first week of December
Review Sets 15 and 16

- Due Friday Nov 18

- Optionally Due Monday Nov 21
Outline of Lectures: Prefetching

- Why prefetch? Why could/does it work?
- The four questions
  - What (to prefetch), when, where, how
- Software prefetching
- Hardware prefetching algorithms
- Execution-based prefetching
- Prefetching performance
  - Coverage, accuracy, timeliness
  - Bandwidth consumption, cache pollution
- Prefetcher throttling
- Issues in multi-core
Readings in Prefetching

Required:

Recommended:
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
Hardware Prefetching (I)

- **Idea:** Specialized hardware observes load/store access patterns and prefetches data based on past access behavior.

- **Tradeoffs:**
  - Can be tuned to system implementation
  - No code portability issues (in terms of performance variation between implementations)
  - Does not waste instruction execution bandwidth
  - More hardware complexity to detect patterns
  - Software can be more efficient in some cases
Review: Next-Line Prefetchers

- Simplest form of hardware prefetching: always prefetch next N cache lines after a demand access (or a demand miss)
  - Next-line prefetcher (or next sequential prefetcher)
  - Tradeoffs:
    + Simple to implement. No need for sophisticated pattern detection
    + Works well for sequential/streaming access patterns (instructions?)
    -- Can waste bandwidth with irregular patterns
      - What is the prefetch accuracy if access stride = 2 and N = 1?
    -- What if the program is traversing memory from higher to lower addresses?
      - Also prefetch “previous” N cache lines?
Stride Prefetchers

- Two kinds
  - Instruction program counter (PC) based
  - Cache block address based

- Instruction based:
  - Idea:
    - Record the distance between the memory addresses referenced by a load instruction (i.e. stride of the load) as well as the last address referenced by the load
    - Next time the same load instruction is fetched, prefetch last address + stride
Instruction Based Stride Prefetching

What is the problem with this?
- Hint: how far can this get ahead? How much of the miss latency can the prefetch cover?
- Initiating the prefetch when the load is fetched the next time can be too late
  - Load will access the data cache soon after it is fetched!

Solutions:
- Use lookahead PC to index the prefetcher table
- Prefetch ahead (last address + N*stride)
- Generate multiple prefetches
Cache-Block Address Based Stride Prefetching

- Can detect
  - A, A+N, A+2N, A+3N, ... 
  - **Stream buffers** are a special case of cache block address based stride prefetching where N = 1
    - Read the Jouppi paper
    - Stream buffer also has data storage in that paper (no prefetching into cache)
Stream Buffers (Jouppi, ISCA 1990)

- Each stream buffer holds one stream of sequentially prefetched cache lines
- On a load miss check the head of all stream buffers for an address match
  - if hit, pop the entry from FIFO, update the cache with data
  - if not, allocate a new stream buffer to the new miss address (may have to recycle a stream buffer following LRU policy)
- Stream buffer FIFOs are continuously topped-off with subsequent cache lines whenever there is room and the bus is not busy
- Can incorporate stride prediction mechanisms to support non-unit-stride streams
  - See “Evaluating stream buffers as a secondary cache replacement”, ISCA 1994
Stream Buffer Design

Diagram showing the design of a stream buffer. The diagram includes:
- **Next Address**
- **Cache Block**
- **V**
- **Tag**
- **CPU address**
- **Compare**
- **Increment**
- **Prefetch Address**
Stream Buffer Design
Tradeoffs in Stride Prefetching

- Instruction based stride prefetching vs. cache block address based stride prefetching

- The latter can exploit strides that occur due to the interaction of multiple instructions

- The latter can more easily get further ahead of the processor access stream
  - No need for lookahead PC

- The latter is more hardware intensive
  - Usually there are more data addresses to monitor than instructions
Locality Based Prefetchers

- In many applications access patterns are not perfectly strided
  - Some patterns look random to closeby addresses
  - How do you capture such accesses?

- Locality based prefetching
Pentium 4 (Like) Prefetcher (Srinath et al., HPCA 2007)

- Multiple tracking entries for a range of addresses
- **Invalid**: The tracking entry is not allocated a stream to keep track of. Initially, all tracking entries are in this state.
- **Allocated**: A demand (i.e. load/store) L2 miss allocates a tracking entry if the demand miss does not find any existing tracking entry for its cache-block address.
- **Training**: The prefetcher trains the direction (ascending or descending) of the stream based on the next two L2 misses that occur +/- 16 cache blocks from the first miss. If the next two accesses in the stream are to ascending (descending) addresses, the direction of the tracking entry is set to 1 (0) and the entry transitions to **Monitor and Request state**.
- **Monitor and Request**: The tracking entry monitors the accesses to a memory region from a **start pointer** (address A) to an **end pointer** (address P). The maximum distance between the start pointer and the end pointer is determined by **Prefetch Distance**, which indicates how far ahead of the demand access stream the prefetcher can send requests. If there is a demand L2 cache access to a cache block in the monitored memory region, the prefetcher requests cache blocks [P+1, ..., P +N] as prefetch requests (assuming the direction of the tracking entry is set to 1). N is called the **Prefetch Degree**. After sending the prefetch requests, the tracking entry starts monitoring the memory region between addresses A+N to P+N (i.e. effectively it moves the tracked memory region by N cache blocks).
Limitations of Locality-Based Prefetchers

- Bandwidth intensive
  - Why?
  - Can be fixed by
    - Stride detection
    - Feedback mechanisms

- Limited to prefetching closeby addresses
  - What about large jumps in addresses accessed?

- However, they work very well in real life
  - Single-core systems
Prefetcher Performance (I)

- **Accuracy** (used prefetches / sent prefetches)
- **Coverage** (prefetched misses / all misses)
- **Timeliness** (on-time prefetches / used prefetches)

**Bandwidth consumption**
- Memory bandwidth consumed with prefetcher / without prefetcher
- Good news: **Can utilize idle bus bandwidth (if available)**

**Cache pollution**
- Extra demand misses due to prefetch placement in cache
- More difficult to quantify but affects performance
Prefetcher Performance (II)

- Prefetcher aggressiveness affects all performance metrics
- Aggressiveness dependent on prefetcher type
- For most hardware prefetchers:
  - **Prefetch distance**: how far ahead of the demand stream
  - **Prefetch degree**: how many prefetches per demand access
Prefetcher Performance (III)

- How do these metrics interact?
- Very Aggressive
  - Well ahead of the load access stream
  - Hides memory access latency better
  - More speculative
  + Higher coverage, better timeliness
  -- Likely lower accuracy, higher bandwidth and pollution
- Very Conservative
  - Closer to the load access stream
  - Might not hide memory access latency completely
  - Reduces potential for cache pollution and bandwidth contention
  + Likely higher accuracy, lower bandwidth, less polluting
  -- Likely lower coverage and less timely
Prefetcher Performance (IV)
Prefetcher Performance (V)

Feedback-Directed Prefetcher Throttling (I)

- **Idea:**
  - Monitor prefetcher performance metrics
  - Throttle the prefetcher aggressiveness up/down based on past performance
  - Change the location prefetches are inserted in cache based on past performance
Feedback-Directed Prefetcher Throttling (III)

- BPKI - Memory Bus Accesses per 1000 retired Instructions
  - Includes effects of L2 demand misses as well as pollution induced misses and prefetches
- A measure of bus bandwidth usage

<table>
<thead>
<tr>
<th></th>
<th>No. Pref.</th>
<th>Very Cons</th>
<th>Mid</th>
<th>Very Aggr</th>
<th>FDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPC</td>
<td>0.85</td>
<td>1.21</td>
<td>1.47</td>
<td>1.57</td>
<td>1.67</td>
</tr>
<tr>
<td>BPKI</td>
<td>8.56</td>
<td>9.34</td>
<td>10.60</td>
<td>13.38</td>
<td>10.88</td>
</tr>
</tbody>
</table>
How to Cover More Irregular Access Patterns?

- 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:
- After referencing a particular address (say A or E), are some addresses more likely to be referenced next?
Markov Prefetching (II)

- Track the likely next addresses after seeing a particular address
- Prefetch *accuracy* is generally low so prefetch up to \( N \) next addresses to increase *coverage*
- Prefetch accuracy can be improved by using longer history
  - Decide which address to prefetch next by looking at the last \( K \) load addresses instead of just the current one
  - e.g., index with the XOR of the data addresses from the last \( K \) loads
  - Using history of a few loads can increase accuracy dramatically
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

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;
}

Example from mst
Shortcomings of CDP – An example

Virtual Address Predictor

Cache Line Addr
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