Homework 6

- Due Today
- Topics: Virtual memory and cache interaction, main memory, memory scheduling
Lab 6: Memory Hierarchy

- Due Wednesday (April 24)
- Cycle-level modeling of L2 cache and DRAM-based main memory

- Extra credit: Prefetching
  - Design your own hardware prefetcher to improve system performance
Feedback

- Many of you wanted more lectures

- Plan for the rest of the semester
  - 3 lectures this week and 1 next Monday
  - 2 wrap-up and recitation sessions next Wednesday and Friday
Office Change

- I am no longer in Hamerschlag Hall A305
- New office: CIC 4105
- Office hours are still the same:
  - Wed 2:30-3:30pm, or by appointment
Pictorially, ...
Last Lecture

- Prefetching
  - Software prefetchers
  - Hardware prefetchers
    - Next-line
    - Stride
    - Stream
Today

- More Prefetching
  - Prefetcher performance metrics
  - Prefetching for more irregular access patterns

- Multiprocessing Fundamentals
  - Why parallel processing?
  - Tightly vs. loosely coupled multiprocessing
  - Parallel speedup
  - Amdahl’s Law
Advanced Prefetching
Review: 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
Review: 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
Review: Cache-Block Address Based Stride Prefetching

- Can detect
  - $A, A+N, A+2N, A+3N, \ldots$
  - **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)
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.
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
Feedback-Directed Prefetcher Throttling (I)

Idea:
- Dynamically 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

<table>
<thead>
<tr>
<th>Accuracy</th>
<th>Not-Late</th>
<th>Late</th>
<th>Not-Poll</th>
<th>Polluting</th>
<th>Late</th>
<th>Decrease</th>
<th>Increase</th>
<th>Decrease</th>
<th>No Change</th>
<th>Decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Med</td>
<td>Not-Poll</td>
<td>Polluting</td>
<td>Late</td>
<td>Decrease</td>
<td>Increase</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low</td>
<td>Not-Poll</td>
<td>Decrease</td>
<td></td>
<td>Not-Late</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No Change</td>
<td></td>
</tr>
</tbody>
</table>
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 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)

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
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.

Efficient 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;
}

Example from mst
Efficient CDP – An Example

Cache Line Addr

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

![Diagram](attachment:image.png)
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;
    }
    ...
    // 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();
END_FOR:
    // the target address of the pre- // execution
    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();
END_FOR:
    // the target address of the pre- // execution
    i++, arcout+=3;
    // terminate this pre-execution if we // have passed the end of the for-loop
    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.

Figure 2. Abstract versions of an important loop nest in the Spec2000 benchmark mcf. Loads that incur many cache misses are underlined.
Example ISA Extensions

\[ Thread\_ID = \textbf{PreExecute\_Start}(Start\_PC, Max\_Insts) : \]
Request for an idle context to start pre-execution at \textit{Start\_PC} and stop when \textit{Max\_Insts} instructions have been executed; \textit{Thread\_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.

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

\textbf{PreExecute\_Cancel}(Thread\_ID): Terminate the pre-execution thread with \textit{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

(a) Execution Time Normalized to the Original Case
Problem Instructions


Figure 2. Example problem instructions from heap insertion routine in vpr.

```c
struct s_heap **heap; // from [1..heap_size]
int heap_size; // # of slots in the heap
int heap_tail; // first unused slot in heap

void add_to_heap (struct s_heap *hptr) {
  ...
  1. heap[heap_tail] = hptr; // branch misprediction
  2. int ifrom = heap_tail;
  3. int ito = ifrom/2;
  4. heap_tail++;
  5. while ((ito >= 1) &&
          (heap[ifrom]->cost < heap[ito]->cost))
     6.   struct s_heap *temp_ptr = heap[ito];
     7.   heap[ito] = heap[ifrom];
     8.   heap[ifrom] = temp_ptr;
     9.   ifrom = ito;
    10.  ito = ifrom/2;
  ...
}
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 */
2 lda s1, 252(gp)  # &heap_tail
2 ldl t2, 0(s1)   # ifrom = heap_tail
1 ldq t5, -76(s1)  # &heap[0]
3 cmplt t2, 0, t4 # see note
4 addl t2, 0x1, t6 # heap_tail +1
1 s8addq t2, t5, t3 # &heap[heap_tail]
4 stl t6, 0(s1)   # store heap_tail
3 stq s0, 0(t3)   # heap[heap_tail]
3 addl t2, t4, t4 # see note
3 sra t4, 0x1, t4 # ito = ifrom/2
5 ble t4, return  # (ito < 1)
loop:
6 s8addq t2, t5, a0 # &heap[ifrom]
6 s8addq t4, t5, t7 # &heap[ito]
11 cmplt t4, 0, t9 # see note
10 move t4, t2    # ifrom = ito
6 ldq a2, 0(a0)   # heap[ifrom]
6 ldq a4, 0(t7)   # heap[ito]
11 addl t4, t9, t9 # see note
11 sra t9, 0x1, t4 # ito = ifrom/2
6 lds $f0, 4(a2)  # heap[ifrom]->cost
6 lds $f1, 4(a4)  # heap[ito]->cost
6 cmpltlt $f0,$f1,$f0 # (heap[ifrom]->cost
6 fbeq $f0, return # < heap[ito]->cost
8 stq a2, 0(t7)   # heap[ito]
9 stq a4, 0(a0)   # heap[ifrom]
5 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.

```assembly
slice:
1 ldl $6, 328(gp)  # &heap
2 ldl $3, 252(gp)  # ito = heap_tail
slice_loop:
3,11 sra $3, 0x1, $3 # ito /= 2
6 s8addq $3, $6, $16 # &heap[ito]
6 ldl $18, 0($16) # heap[ito]
6 lds $f1, 4($18) # heap[ito]->cost
6 cmplt $f1,$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
```
Review: Runahead Execution

- 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

**Small Window:**

Load 1 Miss

Compute | Stall | Compute | Stall

Miss 1 | Miss 2

**Runahead:**

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

Compute | Runahead | Compute

Miss 1  Miss 2

Saved Cycles