Announcements for This Week

- December 2:
  - Midterm II
  - Comprehensive
  - 2 letter-sized cheat sheets allowed
  - Sample exams and solutions are all posted

- December 1:
  - Homework 6 due
Project Schedule

- December 6 and 8
  - Milestone III meetings
  - Signup sheets will be posted online

- December 13
  - Project poster session
  - Location and time TBD

- December 18
  - Final project report due
  - Strive for a good conference paper (in terms of insight, explanations, writing, and formatting)
  - Address comments from the poster session and milestone III
Hybrid Hardware Prefetchers

- Many different access patterns
  - Streaming, striding
  - Linked data structures
  - Localized random

- Idea: **Use multiple prefetchers to cover all patterns**

  + Better prefetch coverage
  -- More complexity
  -- More bandwidth-intensive
  -- Prefetchers start getting in each other’s way (contention, pollution
    - Need to manage accesses from each prefetcher
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
    - Branch prediction, value prediction, only address generation computation
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

- Diagrams:
  - (a) Multiple Pointer Chains
  - (b) Non-Linear Array Accesses
  - (c) Multiple Procedure Calls
  - (d) Multiple Control-Flow Paths
An Example

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

\[\text{Thread\_ID} = \text{PreExecute\_Start}(\text{Start\_PC}, \text{Max\_Insts}):\]
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\_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)
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
1  ldq   t2, 0(s1)      # ifrom = heap_tail
3  cmplt t2, 0, t4     # see note
4  addl t2, 0x1, t6    # heap_tail +=
1  s8addq t2, t5, t3   # &heap[heap_tail]
4  stl   t6, 0(s1)      # store heap_tail
1  stg   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[ito]
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  cmplt $f0,$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  ldq   $6, 328(gp)    # &heap
2  ldq   $3, 252(gp)    # ito = heap_tail
slice_loop:
3,11 sra $3, 0x1, $3   # ito /= 2
6  s8addq $3, $6, $16  # &heap[ito]
6  ldq   $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

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:**

- **Load 1 Miss**
- **Load 2 Miss**

**Runahead:**

- **Load 1 Miss**
- **Load 2 Miss**
- **Load 1 Hit**
- **Load 2 Hit**

**Saved Cycles**
Runahead Execution (III)

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

- **Disadvantages/Limitations:**
  -- Extra executed instructions
  -- Limited by branch prediction accuracy
  -- Cannot prefetch dependent cache misses. Solution?
  -- Effectiveness limited by available MLP
  -- Prefetch distance limited by memory latency

- **Implemented in IBM POWER6, Sun “Rock”**
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
Multi-Core Issues in Prefetching
Prefetching in Multi-Core (I)

- Prefetching shared data
  - Coherence misses

- Prefetch efficiency a lot more important
  - Bus bandwidth more precious
  - Prefetchers on different cores can deny service to each other and to demand requests
    - DRAM contention
    - Bus contention
    - Cache conflicts
  - Need to coordinate the actions of independent prefetchers for best system performance
    - Each prefetcher has different accuracy, coverage, timeliness
Shortcoming of Local Prefetcher Throttling

Local-only prefetcher control techniques have no mechanism to detect inter-core interference.
Prefetching in Multi-Core (II)

- Ideas for coordinating different prefetchers’ actions
  - Utility-based prioritization
    - Prioritize prefetchers that provide the best marginal utility on system performance
  - Cost-benefit analysis
    - Compute cost-benefit of each prefetcher to drive prioritization
  - Heuristic based methods
    - Global controller overrides local controller’s throttling decision based on interference and accuracy of prefetchers
Hierarchical Prefetcher Throttling

Global Control’s goal: Keep track of and control prefetcher-caused inter-core interference in shared memory system

Local Control’s goal: Maximize the prefetching performance of core \( i \) independently

Global Control: accepts or overrides decisions made by local control to improve overall system performance

Pref. \( i \)

Local Control

Core \( i \)

Final Throttling Decision

Throttling Decision

Accuracy

Global Control

Shared Cache

Cache Pollution Feedback

Bandwidth Feedback

Memory Controller
Hierarchical Prefetcher Throttling Example

- High accuracy
- High pollution
- High bandwidth consumed while other cores need bandwidth

Memory Controller

High BW (i)
High BWNO (i)

Pref. i

Local Control

Core i

Local Throttling Decision

Global Control

High Acc (i)

Local Throttle Up

Enforce Throttling Decision

Pol. Filter i

Shared Cache

High Pol (i)
Control-Flow Handling
Readings

- **Required:**

- **Recommended:**
The Branch Problem

- Control flow instructions (branches) are frequent
  - 15-25% of all instructions

- Problem: Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor
  - N cycles: (minimum) branch resolution latency
  - Stalling on a branch wastes instruction processing bandwidth (i.e. reduces IPC)
    - N x IW instruction slots are wasted

- How do we keep the pipeline full after a branch?
- Problem: Need to determine the next fetch address when the branch is fetched (to avoid a pipeline bubble)
The Branch Problem

- Assume a 5-wide superscalar pipeline with 20-cycle branch resolution latency

- How long does it take to fetch 500 instructions?
  - Assume no fetch breaks and 1 out of 5 instructions is a branch
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work
  - 99% accuracy
    - 100 (correct path) + 20 (wrong path) = 120 cycles
    - 20% extra instructions fetched
  - 98% accuracy
    - 100 (correct path) + 20 * 2 (wrong path) = 140 cycles
    - 40% extra instructions fetched
  - 95% accuracy
    - 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
    - 100% extra instructions fetched
## Branch Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Direction at fetch time</th>
<th>Number of possible next fetch addresses?</th>
<th>When is next fetch address resolved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional</td>
<td>Unknown</td>
<td>2</td>
<td>Execution (register dependent)</td>
</tr>
<tr>
<td>Unconditional</td>
<td>Always taken</td>
<td>1</td>
<td>Decode (PC + offset)</td>
</tr>
<tr>
<td>Call</td>
<td>Always taken</td>
<td>1</td>
<td>Decode (PC + offset)</td>
</tr>
<tr>
<td>Return</td>
<td>Always taken</td>
<td>Many</td>
<td>Execution (register dependent)</td>
</tr>
<tr>
<td>Indirect</td>
<td>Always taken</td>
<td>Many</td>
<td>Execution (register dependent)</td>
</tr>
</tbody>
</table>

Different branch types can be handled differently
Approaches to Conditional Branch Handling

- Branch prediction
  - Static
  - Dynamic

- Eliminating branches
  I. Predicated execution
    - Static
    - Dynamic
    - HW/SW Cooperative
  II. Predicate combining (and condition registers)

- Multi-path execution
- Delayed branching (branch delay slot)
- Fine-grained multithreading
Predicate Combining

- Complex predicates are converted into multiple branches
  - if ((a == b) && (c < d) && (a > 5000)) { ... }
  - 3 conditional branches
- Problem: This increases the number of control dependencies
- Idea: Combine predicate operations to feed a single branch instruction
  - Predicates stored and operated on using condition registers
  - A single branch checks the value of the combined predicate
- Fewer branches in code → fewer mipredictions/stalls
- Possibly unnecessary work
  - If the first predicate is false, no need to compute other predicates
- Condition registers exist in IBM RS6000 and the POWER architecture
Delayed Branching (I)

- Change the semantics of a branch instruction
  - Branch after N instructions
  - Branch after N cycles

- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are always executed regardless of branch direction.

- Problem: How do you find instructions to fill the delay slots?
  - Branch must be independent of delay slot instructions

- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot
Delayed Branching (II)

Normal code:

Timeline:

Delayed branch code:

Timeline:

6 cycles

5 cycles
Fancy Delayed Branching (III)

- Delayed branch with squashing
  - In SPARC
  - If the branch falls through (not taken), the delay slot instruction is not executed
  - Why could this help?

Normal code: Delayed branch code: Delayed branch w/ squashing:

\[
\begin{align*}
\text{Normal code:} & \quad \begin{array}{c}
A \\
B \\
C \\
BCX \\
D \\
E
\end{array} \\
\text{Delayed branch code:} & \quad \begin{array}{c}
A \\
B \\
C \\
BCX \\
\text{NOP} \\
D \\
E
\end{array} \\
\text{Delayed branch w/ squashing:} & \quad \begin{array}{c}
A \\
B \\
C \\
BCX \\
A \\
D \\
E
\end{array}
\end{align*}
\]
Delayed Branching (IV)

- **Advantages:**
  + Keeps the pipeline full with useful instructions assuming
    1. Number of delay slots == number of instructions to keep the pipeline full before the branch resolves
    2. All delay slots can be filled with useful instructions

- **Disadvantages:**
  -- Not easy to fill the delay slots (even with a 2-stage pipeline)
    1. Number of delay slots increases with pipeline depth, issue width, instruction window size.
    2. Number of delay slots should be variable with variable latency operations. Why?
  -- Ties ISA semantics to hardware implementation
    -- SPARC, MIPS, HP-PA: 1 delay slot
    -- What if pipeline implementation changes with the next design?
Fine-Grained Multithreading

- **Idea:** Hardware has multiple thread contexts. Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch resolves, there is no need to fetch another instruction from the same thread.
  - Branch resolution latency overlapped with execution of other threads’ instructions.

- No logic needed for branch prediction, (also for dependency checking)
- Single thread performance suffers
- Does not overlap latency if not enough threads to cover the whole pipeline
- Extra logic for keeping thread contexts