Literature Review Process

- Literature review process
- Done in groups: your research project group
- **Step 1**: Pick 3 research papers
  - Broadly related to your research project
- **Step 2**: Send me the list of papers with links to pdf copies (by **Wednesday, March 16**)
  - I need to approve the 3 papers
  - We will iterate to ensure convergence on the list
- **Step 3**: Prepare a 20-minute presentation on the 3 papers
  - Total time: 20-minute talk + 10-minute Q&A
  - Talk should focus on insights and tradeoffs
- **Step 4**: Deliver the presentation in front of class (dates: **April 4 and 6**)
Literature Survey Guidelines

- The goal is to
  - Understand the solution space and tradeoffs
  - Analyze and synthesize three papers
  - Explain how they relate to your project, how they can enhance it, or why your solution will be better

- Read the papers very carefully
  - Attention to detail is important
The talk should clearly convey at least the following:

- **The problem**: What is the general problem targeted by the papers and what are the specific problems?
- **The solutions**: What are the key ideas and solution approaches of the proposed papers?
- **Key results and insights**: What are the key results, insights, and conclusions of the papers?
- **Tradeoffs and analyses**: How do the solutions differ or interact with each other? Can they be combined? What are the tradeoffs between them? This is where you will need to analyze the approaches and find a way to synthesize a common framework to describe and qualitatively compare and contrast the approaches.
- **Comparison to your project**: How do these approaches relate to your project? Why is your approach novel, different, better, or complementary?
- **Key conclusions and new ideas**: What have you learned? Do you have new ideas/approaches based on what you have learned?
Announcements

- Send milestone presentations to Chris and me

- These will be uploaded onto the project page along with any project and literature survey related documents
Reviews

- Due Today (March 16, before class)

- Due next Monday (March 21, before class)
Now that We Have MT Hardware …

- ... what else can we use it for?

- Redundant execution to tolerate soft (and hard?) errors

- Implicit parallelization: thread level speculation
  - Slipstream processors
  - Leader-follower architectures

- Helper threading
  - Prefetching
  - Branch prediction

- Exception handling
MT for Exception Handling

- Hardware exceptions cause overhead
- Some exceptions are recoverable from (TLB miss, unaligned access, emulated instructions)
- Pipe flushes due to exceptions reduce thread performance
MT for Exception Handling

- Cost of TLB miss handling
MT for Exception Handling

- **Observation:**
  - The same application instructions are executed in the same order INDEPENDENT of the exception handler’s execution
  - The data dependences between the thread and exception handler are minimal

- **Idea:** Execute the exception handler in a separate thread context; ensure appearance of sequential execution
MT for Exception Handling

- Better than pure software, not as good as pure hardware handling
Why These Uses?

- What benefit of multithreading hardware enables them?

- Ability to communicate/synchronize with very low latency between threads
  - Enabled by proximity of threads in hardware
  - Multi-core has higher latency to achieve this
Helper Threading for Prefetching

- **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)
Helper Threading for Prefetching

- 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
Generalized 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
Slipstream Processors

- **Goal:** use multiple hardware contexts to speed up single thread execution (implicitly parallelize the program)
- **Idea:** Divide program execution into two threads:
  - Advanced thread executes a reduced instruction stream, speculatively
  - Redundant thread uses results, prefetches, predictions generated by advanced thread and ensures correctness
- **Benefit:** Execution time of the overall program reduces
- **Core idea is similar to many thread-level speculation approaches,** except with a reduced instruction stream
Slipstreaming

- “At speeds in excess of 190 m.p.h., high air pressure forms at the front of a race car and a partial vacuum forms behind it. This creates drag and limits the car’s top speed.

- A second car can position itself close behind the first (a process called *slipstreaming* or *drafting*). This fills the vacuum behind the lead car, reducing its drag. And the trailing car now has less wind resistance in front (and by some accounts, the vacuum behind the lead car actually helps pull the trailing car).

- As a result, both cars speed up by several m.p.h.: the two combined go faster than either can alone.”
Slipstream Processors

- Detect and remove ineffectual instructions; run a shortened “effectual” version of the program (Advanced or A-stream) in one thread context.

- Ensure correctness by running a complete version of the program (Redundant or R-stream) in another thread context.

- Shortened A-stream runs fast; R-stream consumes near-perfect control and data flow outcomes from A-stream and finishes close behind.

- Two streams together lead to faster execution (by helping each other) than a single one alone.
Slipstream Idea and Possible Hardware

A-stream

- L1 Data Cache
- Branch Predictor
- Instruction Cache
- Execution Core
- Reorder Buffer

R-stream

- L1 Data Cache
- Reorder Buffer
- Execution Core
- Instruction Cache
- Branch Predictor

L2 Cache (R-stream state only)
Instruction Removal in Slipstream

- **IR detector**
  - Monitors retired R-stream instructions
  - Detects ineffectual instructions and conveys them to the IR predictor
  - Ineffectual instruction examples:
    - dynamic instructions that repeatedly and predictably have no observable effect (e.g., unreferenced writes, non-modifying writes)
    - dynamic branches whose outcomes are consistently predicted correctly.

- **IR predictor**
  - Removes an instruction from A-stream after repeated indications from the IR detector
  - A stream skips ineffectual instructions, executes everything else and inserts their results into delay buffer
  - R stream executes all instructions but uses results from the delay buffer as predictions
What if A-stream Deviates from Correct Execution?

- **Why**
  - A-stream deviates due to incorrect removal or stale data access in L1 data cache

- **How to detect it?**
  - Branch or value mispredict in R-stream (known as an IR misprediction)

- **How to recover?**
  - Restore A-stream register state: copy values from R-stream registers using delay buffer or shared-memory exception handler
  - Restore A-stream memory state: invalidate A-stream L1 data cache (or speculatively written blocks by A-stream)
Slipstream Questions

- How to construct the advanced thread
  - Original proposal:
    - Dynamically eliminate redundant instructions (silent stores, dynamically dead instructions)
    - Dynamically eliminate easy-to-predict branches
  - Other ways:
    - Dynamically ignore long-latency stalls
    - Static based on profiling

- How to speed up the redundant thread
  - Original proposal: Reuse instruction results (control and data flow outcomes from the A-stream)
  - Other ways: Only use branch results and prefetched data as predictions
Dual Core Execution

- **Idea:** One thread context speculatively runs ahead on load misses and prefetches data for another thread context.
Dual Core Execution

- The front processor runs faster by invalidating long-latency cache-missing loads, similar to runahead execution
  - Load misses and their dependents are invalidated
  - Branch mispredictions dependent on cache misses cannot be resolved
- Highly accurate execution as independent operations are not affected
  - Accurate prefetches to warm up caches
  - Correctly resolved independent branch mispredictions
Dual Core Execution

- Re-execution ensures correctness and provides precise program state
  - Resolve branch mispredictions dependent on long-latency cache misses
- Back processor makes faster progress with help from the front processor
  - Highly accurate instruction stream
  - Warmed up data caches
Dual Core Execution
DCE Microarchitecture

Figure 3. The design of DCE architecture.
Dual Core Execution vs. Slipstream

- Dual-core execution does not
  - remove dead instructions
  - reuse instruction register results
  - uses the “leading” hardware context solely for prefetching and branch prediction

+ Easier to implement, smaller hardware cost and complexity
- “Leading thread” perhaps cannot run ahead as much as in slipstream
- Not reusing results in the “trailing thread” can reduce overall performance benefit
Figure 6. Normalized execution time of DCE, DCE with value prediction (DCE vp), and slipstreaming processors (SS).
Thread Level Speculation

- Speculative multithreading, dynamic multithreading, etc...

- **Idea:** Divide a single instruction stream (speculatively) into multiple threads at compile time or run-time
  - Execute speculative threads in multiple hardware contexts
  - Merge results into a single stream

- Hardware/software checks if any true dependencies are violated and ensures sequential semantics
- Threads can be assumed to be independent
- Value/branch prediction can be used to break dependencies between threads
- Entire code needs to be correctly executed to verify such predictions
Thread Level Speculation Example


(a) Example pseudo-code

```plaintext
while (continue.condition) {
    ...
    x = hash[index1];
    ...
    hash[index2] = y;
    ...
}
```

(b) Execution using thread-level speculation

```
Epoch 1
    ... = hash[3]
    ... = hash[10]
    ... = attempt_commit()

Epoch 2
    ... = hash[19]
    ... Violation!
    ... = hash[21]
    ... = attempt_commit()
    ... = attempt_commit()

Epoch 3
    ... = hash[33]
    ... = hash[30]
    ... = attempt_commit()
    ... = attempt_commit()

Epoch 4
    ... = hash[10]
    ... = hash[25]
    ... = attempt_commit()
    ... = attempt_commit()

Epoch 5
    ... = hash[30]
    ... = hash[9]
    ... = attempt_commit()

Epoch 6
    ... = hash[27]
    ... = attempt_commit()

Epoch 7
    ... = hash[25]
    ... = attempt_commit()
```
TLS Conflict Detection Example

Figure 2. Using cache coherence to detect a RAW dependence violation.
Other MT Issues

- How to select threads to co-schedule on the same processor?
  - Which threads/phases go well together?
  - This issue exists in multi-core as well

- How to provide performance isolation (or predictable performance) between threads?
  - This issue exists in multi-core as well

- How to manage shared resources among threads
  - Pipeline, window, registers
  - Caches and the rest of the memory system
  - This issue exists in multi-core as well
Speculation in Parallel Machines
Readings: Speculation

- **Required**

- **Recommended**

- Reading list will be updated...
Speculation

- Speculation: Doing something before you know it is needed.
- Mainly used to enhance performance

Single processor context
- Branch prediction
- Data value prediction
- Prefetching

Multi-processor context
- Thread-level speculation
- Transactional memory
- Helper threads