Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching

- Bandwidth and latency can cause fetch/decode stages to become a bottleneck
- Bandwidth an issue as more instructions fetched per cycle
  - Need multiple instruction cache lookups per cycle
  - Can’t fetch past a taken branch in a single cycle
- Latency more of an issue as pipelines get longer
  - Instruction cache misses and branch mispredictions rely on fetch/decode to refill the pipeline

Trace Caches

- Previous instruction caches
  - Cached instructions in compiled order
  - Instructions decoded after retrieving from cache
- Trace Cache:
  - Cache instructions after decoding
  - Store in observed execution order
- Trace: A short snapshot of the dynamic execution stream
  - May contain up to a few branch instructions, as long as the outcome for each branch is also specified.
  - Can be specified by starting address and branch outcomes for any branches within the trace
  - Also need to know where to execute next at the end of the trace

Trace Caches

- Trace Cache Implementation
  - Query by starting address, plus bits describing number of branches and their outcomes
  - Trace Target Address: Next fetch address if the trace ends in a branch that is taken
  - Trace fall-through address: Next fetch address otherwise
- Indexing method: starting address only, or starting address plus branch bits?
  - Former: Can only have one trace per starting address. But, trace prefix could be returned for partial matches
    - Evict partial hits?
  - Latter: Would allow multiple traces per starting address

Trace Caches

- What is the downside to concatenating the branch bits to the starting address for indexing?
- If a trace contains a return or indirect jump, it is aborted and not cached at all. Why not just store a prefix of the trace prior to those instructions?

Prefetching using Markov Predictors

- Prefetchers are designed to anticipate memory requests
  - Input: Past history of memory references
  - Output: Predicted next memory reference address(es)
  - Some model of program behavior is needed
- Which memory references to observe?
  - Observing all references from CPU doesn’t leave much time to do anything fancy
  - Miss addresses from CPU cache occur much less frequently, allowing the prefetcher to take advantage of more state

Prefetching using Markov Predictors

- Markov Prefetcher:
  - Idea: Build and maintain a Markov model based on past cache miss references
  - Nodes are memory references
  - Edges from reference X to Y are weighted by observed frequency
  - Could extend this by using multiple past references to predict the next – but in practice it didn’t appear to help

Prefetching using Markov Predictors

- Problems: difficult to realize in hardware
  - Can only store a small part of observed history
  - Arbitrary number of nodes and node out-degree
  - Real-valued edge weights
- Solutions:
  - Don't store the history - maintain the model on the fly
  - Limit number of nodes and maximum out-degree
  - Prioritize edges based on an MRU scheme instead of actual frequencies – "Markov-like"
- Hardware implementation:
  - Realize graph as a transition table
  - Rows are indexed by miss reference address, and contain prioritized predictions of next fetch addresses

Prefetching using Markov Predictors

- Instead of placing the prefetcher between the L1 and L2 cache, why not place it between main memory and the cache?
- What is the advantage to using separate on-chip prefetch buffers rather than just storing prefetched data in the cache?