#include "cache.h"

#include <iostream>

#include <vector>

#include <cstdint>

#include <unordered\_map>

#include <cassert>

#include "TCP.h"

/\*

This file implements the Tag Correlating Prefetcher (TCP) as described in the following paper:

https://mrmgroup.cs.princeton.edu/papers/tag\_sequence.pdf

Using ChampSim, a cache simulator used to evaluate cache prefetchers.

This TCP operates by using a Tag History Table (THT) and a Pattern History Table (PHT) to predict the next tag to prefetch.

The TCP operates after the L1D cache and issues prefetches to the L2 cache.

Further notes on the general specifics of this particular implementation can be found in the comments below.

John Iler

Student - Electrical Engineering

Texas A&M University College Station

Updated: 11/30/2024

Notes On Overall Functionality:

Update function:

1. Locate tag\_K (tag before missTag as defined by TCP documentation)

2. Access THT using tag\_K (the previous missTag as defined by TCP documentation)

3. Append missTag to the accessed tag\_Sequence row in the THT

4. When populating the THT (i.e., when the THT is not full), append missTag to the first incomplete row.

5. Access the PHT with tag\_K.

6. Set the corresponding next tag' = missTag (tag' is the next Tag to prefetch as defined by TCP documentation)

7. When populating the PHT, append tag\_K and missTag to the first incomplete row, where tag\_K represents the tag

and missTag represents the predicted next tag (tag').

Lookup Function:

1. Locate the PHT set containing missTag.

2. Fetch the next predicted tag' (tag').

3. Set predictedAddress = (tag' << (index\_bits + offset\_bits)) | (missIndex << offset\_bits) | missOffset.

4. Issue the predictedAddress to the L2 cache using Prefetch\_Line.

Variables Necessary and/or Mentioned in the TCP Documentation:

THT Variables:

- `int sets\_L1D` // Number of sets in the L1D cache

- `int entriesPerRow\_THT` // Number of tags stored in each row of the THT

- `int size\_tag` // Size of the tag (usually 64 bits minus the index and offset)

- `int size\_THT = sets\_L1D \* entriesPerRow\_THT \* size\_tag` // Total size of the THT

- `string missIndex` // Index portion of the memory address

- `string missTag` // Tag portion of the memory address

- `string missAddress` // Full address to calculate the miss (used to prefetch)

PHT Variables:

- `int sets\_L1D` // Number of sets in the Pattern History Table

- `int waysPerSet\_PHT` // Number of ways per set in the Pattern History Table

- `int size\_PHT = sets\_L1D \* 2 \* waysPerSet\_PHT \* size\_tag` // Total size of the PHT

\*/

// Constants for THT and PHT sizes

const int entriesPerRow\_THT = 12; // Number of tags stored in each row of the THT

const int sets\_L1D = 64; // Number of sets in the L1D cache

const int waysPerSet\_PHT = 12; // Number of ways in each set of the L1D Cache

const int index\_bits = 6; // Number of bits for the index (L1D sets)

const int offset\_bits = 6; // Number of bits for the offset (block size)

const int tagSize = 64 - index\_bits - offset\_bits; // Size of the tag in bits

// Tag History Table (THT) Implementation

class TagHistoryTable {

private:

struct THTEntry {

uint64\_t tags[entriesPerRow\_THT] = {0}; // Array to store tags for a given set

};

std::vector<THTEntry> THT\_entries;

public:

TagHistoryTable() : THT\_entries(sets\_L1D) {}

// Update the THT for a given set using the missTag and missIndex

void update(uint64\_t missTag, uint64\_t missIndex) {

assert(missIndex < sets\_L1D); // Ensure that the missIndex is valid

// Shift tags to make room for the new missTag

for (int i = 0; i < entriesPerRow\_THT - 1; ++i) {

THT\_entries[missIndex].tags[i] = THT\_entries[missIndex].tags[i + 1];

}

THT\_entries[missIndex].tags[entriesPerRow\_THT - 1] = missTag; // Add the new missTag

}

// Return the sequence of tags for a given index in the THT

const uint64\_t\* get\_tag\_sequence(uint64\_t index) const {

assert(index < sets\_L1D); // Ensure that the index is valid

return THT\_entries[index].tags;

}

};

// Pattern History Table (PHT) Implementation

class PatternHistoryTable {

private:

struct PHTEntry {

uint64\_t tag\_sequence[2] = {0}; // Sequence of two previous tags

uint64\_t next\_tag = 0; // Predicted successor tag

};

std::vector<PHTEntry> PHT\_entries;

public:

PatternHistoryTable() : PHT\_entries(sets\_L1D \* waysPerSet\_PHT) {}

// Update the PHT with the new missTag and the tag sequence from THT

void update(uint64\_t missTag, uint64\_t missIndex, const uint64\_t\* tag\_sequence) {

// Search the PHT entries for a matching tag sequence or single tag

for (auto& entry : PHT\_entries) {

if ((entry.tag\_sequence[0] == tag\_sequence[0] && entry.tag\_sequence[1] == tag\_sequence[1]) ||

(entry.tag\_sequence[1] == tag\_sequence[1])) {

entry.next\_tag = missTag; // Update next\_tag if a match is found

return;

}

}

// Insert a new entry into PHT if no match is found

uint64\_t idx = missIndex % (sets\_L1D \* waysPerSet\_PHT);

PHT\_entries[idx] = { {tag\_sequence[entriesPerRow\_THT - 2], tag\_sequence[entriesPerRow\_THT - 1]}, missTag };

}

// Look up the predicted tag based on the tag sequence

uint64\_t lookUp(const uint64\_t\* tag\_sequence) {

for (const auto& entry : PHT\_entries) {

// Issue prefetch if there's an exact match of the tag sequence or the most recent tag

if ((entry.tag\_sequence[0] == tag\_sequence[entriesPerRow\_THT - 2] && entry.tag\_sequence[1] == tag\_sequence[entriesPerRow\_THT - 1]) ||

(entry.tag\_sequence[1] == tag\_sequence[entriesPerRow\_THT - 1])) {

return entry.next\_tag; // Return the predicted next tag

}

}

return 0; // Return 0 if no match is found

}

};

// Global instances of the PHT and THT

PatternHistoryTable PHT\_Main;

TagHistoryTable THT\_Main;

std::vector<uint64\_t> miss\_sequence;

// Prefetcher Cache Operate: Handles cache operations based on memory access

uint32\_t TCP::prefetcher\_cache\_operate(champsim::address addr, champsim::address ip, uint8\_t cache\_hit, bool useful\_prefetch, access\_type type, uint32\_t metadata\_in)

{

uint64\_t missAddr = addr.to<uint64\_t>(); // Convert address to uint64\_t

uint64\_t missTag = missAddr >> (index\_bits + offset\_bits); // Extract the tag

uint64\_t missIndex = (missAddr >> offset\_bits) & ((1 << index\_bits) - 1); // Extract the index

uint64\_t missOffset = missAddr & ((1 << offset\_bits) - 1); // Extract the offset

// Update the Tag History Table (THT) with the current missTag and missIndex

THT\_Main.update(missTag, missIndex);

// Retrieve the tag sequence (the last two tags) from THT

const uint64\_t\* tag\_sequence = THT\_Main.get\_tag\_sequence(missIndex);

// Update the Pattern History Table (PHT) with the current missTag and the tag sequence from THT

PHT\_Main.update(missTag, missIndex, &tag\_sequence[entriesPerRow\_THT - 2]);

// Look up the predicted next tag based on the last two-tag sequence from the PHT

uint64\_t pfTag = PHT\_Main.lookUp(&tag\_sequence[entriesPerRow\_THT - 2]);

// Calculate the address for the prefetched line

uint64\_t pfAddr = (pfTag << (index\_bits + offset\_bits)) | (missIndex << offset\_bits) | missOffset;

// If a valid prediction is made, prefetch the line

if (pfTag != 0 && pfAddr != 0) {

prefetch\_line(pfAddr, true, 0); // Issue the prefetch

}

return metadata\_in; // Return the metadata

}

// Prefetcher Cache Fill: Handles the cache fill after a prefetch

uint32\_t TCP::prefetcher\_cache\_fill(champsim::address addr, long set, long way, uint8\_t prefetch, champsim::address evicted\_addr, uint32\_t metadata\_in)

{

return metadata\_in; // No action required in this example

}

// Initialization functions (currently empty, but can be used for setup)

// void TCP::prefetcher\_initialize() {}

// void TCP::prefetcher\_cycle\_operate() {}

// void TCP::prefetcher\_final\_stats() {}