Department of Computer Science and Engineering

Course Title :Computer Architecture
Course Code : CSE 360.
Section : 01.
Semester : Spring25.

Submitted to
Dr. Md. Nawab Yousuf Ali
Professor,
Department of Computer Science and Engineering,
East West University.

Submitted by

Name	Student ID
Nuran Farhana Prova	2023-1-60-075
Afsana Akter Mim	2023-1-60-073
Farhatun Nahar Priya	2023-1-60-202

Date of submission: 21 May, 2025.

Project Title: Implement a cache partitioning scheme to improve performance in multi-core processors in C

Abstract: This paper presents a simulation to show how splitting a cache can enhance the performance of multi-core systems. To resolve any conflicts, the cache memory is divided between all cores before the system is used. Multithreading and mutex locks are used in the simulation to represent simultaneous access to caches and every partition uses an LRU replacement strategy. With this approach, applications enjoy reliable access to data, caches are filled more efficiently and cores work less in conflict with one another. Using this approach, coordinated sharing of cached information and quick memory access are emphasized in systems with many processor cores.

Keywords: Cache partitioning, multithreading, mutex locks, shared cache, multi-core processors, LRU replacement, memory performance.

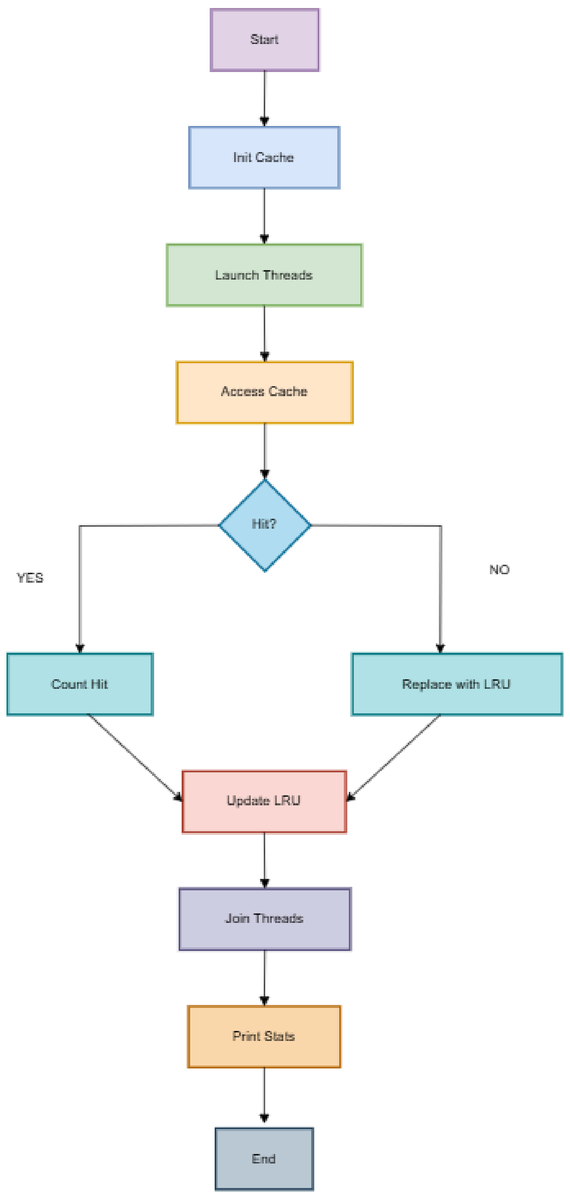
1.Introduction: Modern multi-core processors rely heavily on shared last-level caches (LLCs) such as L3 caches. At the same time, if subordinate cores access the LLC without restriction, it may result in conflicts and a drop in performance, mainly in multi-threaded situations. The solution which is called cache partitioning, is to separate the cache so each core has its own part which reduces clashes over cache lines.The project uses C to design a simulation for an 8-way set-associative L3 cache, equally shared by four cores using fixed way partitioning. A core is given a fixed amount of cache ways to represent independent access and raise predictability. POSIX threads are used by the simulation to show concurrent use of the memory and mutex locks are used to keep the shared cache structure in sync. Recording the number of cache hits and misses per core in the simulation demonstrates that partitioning can improve overall cache performance and ensure it is used evenly in a system with multiple hardware threads.This work discusses how modern processor designs rely on managing the cache and how easy-to-implement partitioning techniques can result in fewer shared resources being used.

2.Problem statement:

Since multiple threads share the cache memory in multi-core systems, this may cause performance problems. When all threads try to use one cache at the same time, they create conflicts and replace each other's data often. As a result, there are more missteps by the cache and resources are not used effectively.

Additionally, when different threads attempt to access the shared cache at the same time and are not in sync, race conditions may appear. These events cause certain data to remain inconsistent and make the system run slower.Fixing these problems calls for a reliable cache management system. The right way to partition the cache ensures that every core gets a fair share, that interference is less likely and that performance goes up without any resource issues.

3.Flowchart Design:



4. Methodology

4.1 The Memory Model is:

Our memory represents a shared cache that stays the same size. The cache is distributed into various sections and each core has a separate part for its use.

4.2 How Memory is Accessed:
So that only one thread is in the cache at any given time, we block other threads using a lock called a mutex. Because of this, the data doesn't get changed by mistake even if different threads want to use the cache simultaneously.

4.3 Scheduling:
We set up several threads during the program's execution, following user requests. Each thread receives its own part of the cache so they don't accidentally use data outside it.

5. Implementation
5.1 We used C language and the pthread library to both create and handle the threads.
5.2 Simply put, each thread acts as a CPU core selecting random addresses so it can attempt to access the cache.
5.3 We deploy a global mutex to ensure access to the cache is restricted to one thread at a time.
5.4 All the threads happen at once, but the use of a mutex stops interference and makes sure caches are updated correctly.

6. Results and their discussion
The programs illustrates how various threads can share memory in a safe way.

By using mutex locks, race conditions are stopped which keeps the cache data accurate.

Allowing each thread to use part of the cache memory and using threads the right way improves how we use our memory.

The cache hit and miss ratio for all threads is included to show how efficient the cache is working during every test.

7.Project Code:

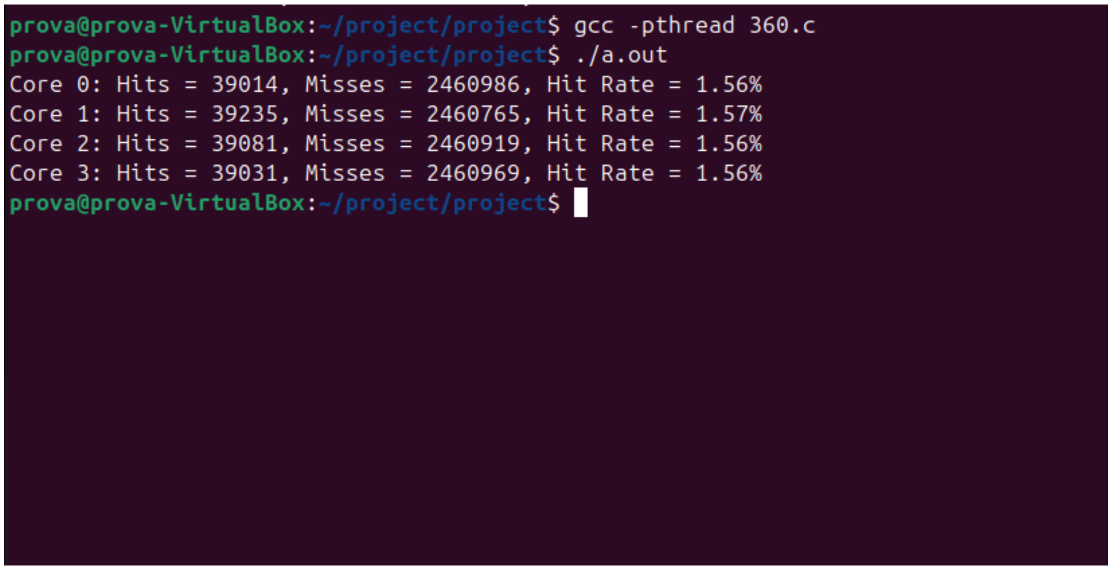
```
#include <stdio.h>
#include <stdlib.h>

#include <time.h>
#include <stdint.h>

#define NUM_CORES 4
#define NUM_SETS 8
#define NUM_WAYS 8
#define CACHE_LINE_SIZE 64
#define ITERATIONS 10000000
```

```
#define BATCH_SIZE 10


typedef struct {
    int valid;
    unsigned int tag;
    int LRU_counter;
} CacheLine;


CacheLine L3_cache[NUM_SETS][NUM_WAYS];


int partition[NUM_CORES] = {2, 2, 2, 2};


int hits[NUM_CORES] = {0};
int misses[NUM_CORES] = {0};


pthread_mutex_t cache_mutex = PTHREAD_MUTEX_INITIALIZER;


void* core_function(void* arg) {
    int core_id = (int)(uintptr_t)arg;
    int ways_start = core_id * partition[core_id];
    int ways_end = ways_start + partition[core_id];

    int local_hits = 0;
    int local_misses = 0;

    for (int i = 0; i < ITERATIONS / NUM_CORES; i += BATCH_SIZE) {
        for (int b = 0; b < BATCH_SIZE; b++) {
            unsigned int address = rand() % 1024;
            unsigned int tag = address / NUM_SETS;
            int set_index = address % NUM_SETS;

            int hit = 0;

            pthread_mutex_lock(&cache_mutex);

            for (int way = ways_start; way < ways_end; way++) {
                if (L3_cache[set_index][way].valid &&
                    L3_cache[set_index][way].tag == tag) {
```

```
                local_hits++;
                L3_cache[set_index][way].LRU_counter = 0;
                hit = 1;
                break;
            }
        }


        if (!hit) {
            local_misses++;
            int replace_way = -1;
            int max_LRU = -1;

            for (int way = ways_start; way < ways_end; way++) {
                if (!L3_cache[set_index][way].valid) {
                    replace_way = way;
                    break;
                }
                if (L3_cache[set_index][way].LRU_counter > max_LRU) {
                    max_LRU = L3_cache[set_index][way].LRU_counter;
                    replace_way = way;
                }
            }

            L3_cache[set_index][replace_way].valid = 1;
            L3_cache[set_index][replace_way].tag = tag;
            L3_cache[set_index][replace_way].LRU_counter = 0;
        }


        for (int way = 0; way < NUM_WAYS; way++) {
            if (L3_cache[set_index][way].valid) {
                L3_cache[set_index][way].LRU_counter++;
            }
        }
        pthread_mutex_unlock(&cache_mutex);
    }
}


pthread_mutex_lock(&cache_mutex);
hits[core_id] += local_hits;
misses[core_id] += local_misses;
pthread_mutex_unlock(&cache_mutex);
```

```
    return NULL;
}

int main() {
    pthread_t threads[NUM_CORES];


    for (int i = 0; i < NUM_SETS; i++) {
        for (int j = 0; j < NUM_WAYS; j++) {
            L3_cache[i][j].valid = 0;
            L3_cache[i][j].tag = 0;
            L3_cache[i][j].LRU_counter = 0;
        }
    }

    srand(time(NULL));


    for (int i = 0; i < NUM_CORES; i++) {
        pthread_create(&threads[i], NULL, core_function, (void*)(uintptr_t)i);
    }


    for (int i = 0; i < NUM_CORES; i++) {
        pthread_join(threads[i], NULL);
    }


    int total_hits = 0, total_misses = 0;
    for (int i = 0; i < NUM_CORES; i++) {
        total_hits += hits[i];
        total_misses += misses[i];
        printf("Core %d: Hits = %d, Misses = %d, Hit Rate = %.2f%%\n",
               i, hits[i], misses[i],
               (double)hits[i] / (hits[i] + misses[i]) * 100);
    }

    printf("Total: Hits = %d, Misses = %d, Overall Hit Rate = %.2f%%\n",
           total_hits, total_misses,
           (double)total_hits / (total_hits + total_misses) * 100);

    return 0;
}
```

8.OUTPUT:



```
prova@prova-VirtualBox:~/project/project$ gcc -pthread 360.c
prova@prova-VirtualBox:~/project/project$ ./a.out
Core 0: Hits = 39014, Misses = 2460986, Hit Rate = 1.56%
Core 1: Hits = 39235, Misses = 2460765, Hit Rate = 1.57%
Core 2: Hits = 39081, Misses = 2460919, Hit Rate = 1.56%
Core 3: Hits = 39031, Misses = 2460969, Hit Rate = 1.56%
prova@prova-VirtualBox:~/project/project$
```

9.Conclusion:

The simulation effectively demonstrates how two or more threads can safely interact with shared memory using mutex locks. We get rid of errors and keep data up-to-date by giving different threads small pieces of the memory and controlling how they access it with synchronization. It also demonstrates that using multithreading and dynamic scheduling enhances both memory use and performance. Such techniques play a key role in designing smooth running systems such as operating systems and multi-core processors.