Report: Part 2 - Optimal Thread Placement on Logical Cores

Author: Krunal Mistry (24360) **Kaushik Rathva (25360)**

Abstract

This report investigates the optimization of thread placement for three threads on a system with two physical cores, each supporting two-way Simultaneous Multithreading (SMT). The objective is to maximize performance by determining the ideal core assignment for each thread while minimizing resource contention. Through profiling, data collection, and analysis, we have found an optimal thread-to-core mapping strategy. The report demonstrates the methodology, tools used, and resulting performance improvements in terms of Speed-up.

Introduction

In modern computing, SMT, also known as Hyper-Threading, enables a single physical core to execute multiple threads by sharing resources. Although SMT can improve throughput, resource contention between threads can degrade performance. In this assignment, we are tasked with optimizing the placement of three threads (T_0, T_1, and T_2) across two physical cores, each supporting two logical cores. Proper assignment can maximize CPU resource utilization on each core.

This report outlines the approach taken to profile, analyze, and optimize thread placement. Performance monitoring tools, specifically `perf`, were used to gather metrics on CPU usage, context switches, and cache behavior. Based on these metrics, we determined the optimal core affinity for each thread, aiming to minimize contention and improve execution efficiency.

1. Profiling Commands Used

1. SEED=<SR Number> make run

This command sets an environment variable SEED to a specific value, <SR Number>, before running a make command. Here's what each part does:

- SEED=<SR Number>: The SEED variable is likely being set to a specific number that may control the randomness or initialization process of the application.
- make run: make is a command often used to build or compile projects using a Makefile. In this case, run is probably a target in the Makefile that triggers specific instructions to start the program. Since SEED is set, the program will use this value during execution.

2. perf stat -e cpu-clock,task-clock,cycles,instructions,cache-misses,cache-references,branches,branch-misses -t std::to_string(tid) + " --output=perf_thread_" + std::to_string(threadIdx) + ".txt &

This command utilizes perf, a Linux performance profiling tool, to gather detailed performance data for a specific process.

- perf stat: perf is used for performance analysis, and stat is a subcommand that collects and displays performance statistics.
- -e: This option specifies the events that perf will record:
- cpu-clock: Measures the total CPU time the thread is active.
- task-clock: Measures the total time spent executing.
- cycles: Counts CPU cycles, a low-level measure of CPU activity.
- instructions: Counts the number of executed instructions.
- cache-misses and cache-references: Track cache accesses and the number of times data isn't found in cache (misses), which can slow down execution.
- t std::to_string(tid): Specifies the process ID (tid or thread ID), so perf collects data only for this specific thread.
- --output=perf_thread_" + std::to_string(threadIdx) + ".txt: Writes the profiling results into a file named after the thread index (threadIdx). This ensures each thread's output is saved separately.
- &: Runs the command in the background, allowing the main process to continue running without waiting for profiling to finish.

3. python3 analyze.py

This command executes a Python script named analyze.py using python3. This file read `perf_thread_<threadIdx>.txt` files and after reading the files it assign logical cores to the each thread according to the logic and write output as <threadIdx>:<Core_number> for each thread in `core_optimization.txt` file.

2. Analysis Methodology:

The methodology followed in this analysis involved several key steps designed to identify optimal thread placement:

1. Data Collection:

Using `perf`, we collected following meterices.

Perf Counter	Values
cycles	696,805,987,163
instructions	1,013,571,250,083
cache-misses	939,492,434
cache-references	46,688,802,060

2. Core Affinity Decision-Making:

Based on the analysis, main criteria is separating thread with high cache misses in different core and another two in same core. For choosing this criteria we have following reasons.

1) Enhanced Performance of Locality-Based Threads:

Threads with strong true locality can benefit from the proximity of their memory accesses. When grouped on the same core, they can capitalize on cached data, significantly speeding up their execution. This synergy can lead to overall performance improvements for the application.

When threads with good locality (i.e., those that access similar data or instructions) are placed on the same core, they can take advantage of shared cache resources.

2) Minimizing Cache Thrashing:

Threads with high cache misses often have poor memory access patterns, which can lead to cache thrashing if multiple such threads are placed on the same core. By assigning these threads to different cores, each thread can utilize its cache more effectively without competing for the same cache lines, leading to fewer cache misses.

3) Isolation of High Cache Miss Threads:

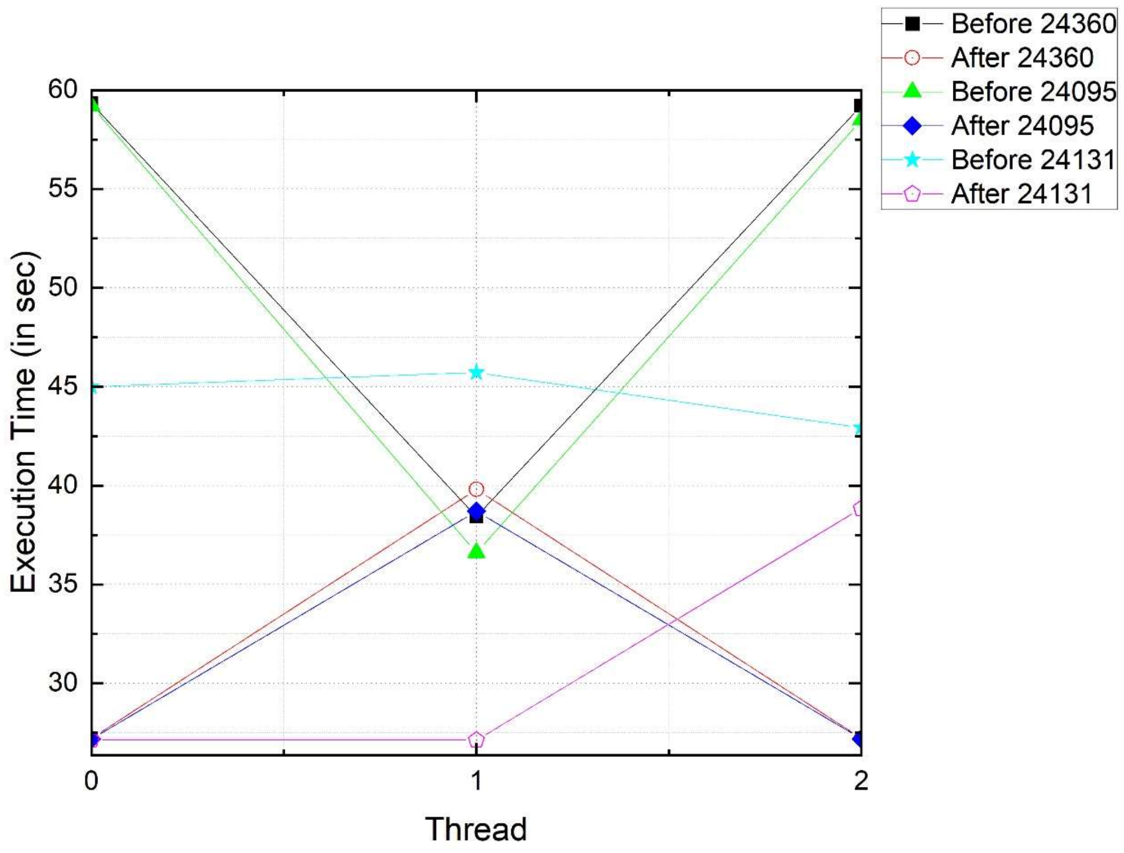
Placing high cache miss threads on separate cores isolates their memory access patterns, preventing them from negatively impacting each other. This isolation can reduce the overall impact on system performance, as these threads won't slow down others that might be more efficient.

The placement of high cache miss threads in different core, while grouping locality-aware threads on the same core, leverages the principles of true locality to optimize performance in multi-core systems. This approach minimizes cache thrashing, enhances cache hit rates, and improves overall execution speed.

So, we are separating memory intensive thread by assigning it to one separate physical core and other two threads in same physical core but different logical core.

3. Graph:

We plot graph for different serial no. Where x axis shows the no of threads and y axis shows the execution time. As per the criteria we can see that the execution time is reduced as per our methodology.



4. Results and Speedup

To measure the effectiveness of the optimization, the execution time was recorded based on the thread that finished last in each configuration.

1. **Initial Configuration**: Default thread placement with no specific logical core assignments resulted in high context switching and cache contention.

2. **Optimized Configuration**: Using custom affinity settings, we achieved a notable reduction in execution time. The calculated speedup was obtained using the formula:
Speedup = (Initial Execution Time) / (Optimized Execution Time)

After running multiple times for the same serial number we get speedup around 1.52 where Initial Execution Time = 59.19446469 second and Optimized Execution Time= 38.70774106 second.

Conclusion

In this report, we explored the optimization of thread placement on a system with SMT capabilities. By profiling and analyzing the behavior of each thread, we developed an efficient affinity strategy that minimized resource contention. This optimization process demonstrates how appropriate core assignment can yield significant performance gains, especially in systems with limited physical cores.