Performance Timing Lab Manual

Understanding CPU Timing Methods and TSC Problems

Lab Overview

This lab will teach you how to measure code performance using different timing methods, understand their limitations, and learn why certain approaches are more reliable than others.

Learning Objectives

- Understand different timing methods (`clock_gettime()` vs `rdtsc()`)
- Learn about TSC (Time Stamp Counter) problems
- Implement CPU pinning for consistent measurements
- Analyze timing accuracy and precision
- Compare performance measurement techniques

Lab Exercise 1: Basic Timing with clock_gettime()

Goal: Implement basic performance timing using the most reliable method.

Step 1.1: Create the Base Code

Create a file called `lab1_basic.c`:

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

int main() {
    struct timespec start, end;

    // TODO 1: Record start time
    clock_gettime(CLOCK_MONOTONIC, &start);

    // Sample workload: Mathematical computation
    double sum = 0.0;
    for (int i = 0; i < 1000000; i++) {
        sum += i * 1.0 / (i + 1);
    }
```

```
    // TODO 2: Record end time
    clock_gettime(CLOCK_MONOTONIC, &end);

    // TODO 3: Calculate elapsed time
    double elapsed_time = (end.tv_sec - start.tv_sec) +
                          (end.tv_nsec - start.tv_nsec) / 1e9;

    printf("Execution time: %f seconds\n", elapsed_time);
    printf("Sum: %f\n", sum);

    return 0;
}
```

Step 1.2: Compile and Run

```
gcc -o lab1 lab1_basic.c
./lab1
```

Step 1.3: Questions to Answer

1. What time did you get?
2. Run the program 5 times. Do you get consistent results?

Step 1.4: Experiment - Different Clock Types

Modify your code to test different clock types:

```
// Try these different clocks:
clock_gettime(CLOCK_MONOTONIC, &start);      // Standard choice
clock_gettime(CLOCK_MONOTONIC_RAW, &start);  // No NTP adjustments
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start); // CPU time only
```

Question: Which clock type gives you the most consistent results for performance measurement?

Lab Exercise 2: Introduction to TSC (Time Stamp Counter)

Goal: Learn about hardware-level timing using CPU cycles.

Step 2.1: Add TSC Support

Create `lab2_tsc.c` by copying `lab1_basic.c` and adding:

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

// TODO 4: Implement RDTSC function
static inline uint64_t rdtsc(void) {
    unsigned int lo, hi;
    asm volatile ("rdtsc" : "=a" (lo), "=d" (hi));
    return ((uint64_t)hi << 32) | lo;
}

int main() {
    struct timespec start_time, end_time;
    uint64_t start_cycles, end_cycles;

    // Start both timers
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    start_cycles = rdtsc();

    // Same workload
    double sum = 0.0;
    for (int i = 0; i < 1000000; i++) {
        sum += i * 1.0 / (i + 1);
    }

    // End both timers
    end_cycles = rdtsc();
    clock_gettime(CLOCK_MONOTONIC, &end_time);

    // Calculate results
    double elapsed_time = (end_time.tv_sec - start_time.tv_sec) +
                          (end_time.tv_nsec - start_time.tv_nsec) / 1e9;
    uint64_t elapsed_cycles = end_cycles - start_cycles;
```

```
    // TODO 5: Calculate CPU frequency
    double cpu_freq = elapsed_cycles / elapsed_time;

    printf("Time: %f seconds\n", elapsed_time);
    printf("Cycles: %lu\n", elapsed_cycles);
    printf("Estimated CPU frequency: %.2f GHz\n", cpu_freq / 1e9);
    printf("Cycles per iteration: %.2f\n", (double)elapsed_cycles / 1000000);
    printf("Sum: %f\n", sum);

    return 0;
}
```

Step 2.2: Check Your CPU Frequency

```
# Find your actual CPU frequency
lscpu | grep MHz
cat /proc/cpuinfo | grep "cpu MHz" | head -1
```

Step 2.3: Questions to Answer

1. What CPU frequency did you measure?
2. What's your actual CPU frequency?
3. How close are they?
4. How many cycles per iteration did you get?

Lab Exercise 3: TSC Problem #1 - Frequency Scaling

Goal: Understand how CPU frequency scaling affects TSC measurements.

Step 3.1: Create Frequency Analysis Code

Create `lab3_frequency.c`:

```
#include <stdio.h>
#include <time.h>
#include <stdint.h>
#include <unistd.h>

static inline uint64_t rdtsc(void) {
    unsigned int lo, hi;
```

```
    asm volatile ("rdtsc" : "=a" (lo), "=d" (hi));
    return ((uint64_t)hi << 32) | lo;
}

void measure_frequency() {
    // TODO 6: Implement frequency measurement
    struct timespec start, end;
    uint64_t start_cycles, end_cycles;

    clock_gettime(CLOCK_MONOTONIC, &start);
    start_cycles = rdtsc();

    // Sleep for exactly 1 second
    sleep(1);

    end_cycles = rdtsc();
    clock_gettime(CLOCK_MONOTONIC, &end);

    double elapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
    uint64_t cycles = end_cycles - start_cycles;

    printf("Sleep time: %.6f seconds\n", elapsed);
    printf("TSC cycles during sleep: %lu\n", cycles);
    printf("TSC frequency during sleep: %.2f GHz\n", cycles / elapsed / 1e9);
}

void measure_computation() {
    // TODO 7: Measure during computation
    struct timespec start, end;
    uint64_t start_cycles, end_cycles;

    clock_gettime(CLOCK_MONOTONIC, &start);
    start_cycles = rdtsc();

    // Intensive computation to make CPU boost
    volatile double sum = 0.0;
    for (int i = 0; i < 10000000; i++) {
        sum += i * 1.0 / (i + 1);
        sum = sum * 1.00001;  // More work
    }

    end_cycles = rdtsc();
    clock_gettime(CLOCK_MONOTONIC, &end);
```

```
    double elapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
    uint64_t cycles = end_cycles - start_cycles;

    printf("Computation time: %.6f seconds\n", elapsed);
    printf("TSC cycles during computation: %lu\n", cycles);
    printf("TSC frequency during computation: %.2f GHz\n", cycles / elapsed / 1e9);
}

int main() {
    printf("=== TSC Frequency Analysis ===\n");
    printf("Measuring TSC during sleep (CPU idle):\n");
    measure_frequency();

    printf("\nMeasuring TSC during computation (CPU active):\n");
    measure_computation();

    return 0;
}
```

Step 3.2: Questions to Answer

1. Is the TSC frequency the same during sleep and computation?
2. What does this tell you about using a fixed frequency value?
3. Which measurement is more reliable for timing code performance?

Lastly:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <time.h>
#include <stdint.h>
#include <sched.h>

// Simple CPU pinning
void pin_to_cpu(int cpu) {
    cpu_set_t set;
    CPU_ZERO(&set);
    CPU_SET(cpu, &set);
    if (sched_setaffinity(0, sizeof(set), &set) != 0) {
        printf("Warning: Could not pin to CPU %d\n", cpu);
```

```
    }
}

// Basic RDTSC
static inline uint64_t rdtsc(void) {
    unsigned int lo, hi;
    asm volatile ("rdtsc" : "=a" (lo), "=d" (hi));
    return ((uint64_t)hi << 32) | lo;
}

// Serialized RDTSC - fixed version
static inline uint64_t rdtsc_serialized(void) {
    unsigned int lo, hi;
    // Use cpuid to serialize, then rdtsc
    asm volatile ("cpuid" ::: "eax", "ebx", "ecx", "edx"); // Serialize
    asm volatile ("rdtsc" : "=a" (lo), "=d" (hi));        // Read TSC
    return ((uint64_t)hi << 32) | lo;
}

// Alternative: Use RDTSCP if available (serializing by design)
static inline uint64_t rdtscp(void) {
    unsigned int lo, hi, aux;
    asm volatile ("rdtscp" : "=a" (lo), "=d" (hi), "=c" (aux));
    return ((uint64_t)hi << 32) | lo;
}

void timing_comparison() {
    printf("Pinning to CPU 0 for consistent measurements...\n");
    pin_to_cpu(0);

    struct timespec start_time, end_time;
    uint64_t start_cycles1, end_cycles1;
    uint64_t start_cycles2, end_cycles2;

    // Method 1: clock_gettime + regular RDTSC
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    start_cycles1 = rdtsc();

    // Work
    volatile double sum = 0.0;
    for (int i = 0; i < 1000000; i++) {
        sum += i * 1.0 / (i + 1);
    }
```

```
    end_cycles1 = rdtsc();
    clock_gettime(CLOCK_MONOTONIC, &end_time);

    // Small delay to separate measurements
    for (int i = 0; i < 1000; i++) {
        volatile int x = i;
    }

    // Method 2: Serialized RDTSC
    start_cycles2 = rdtsc_serialized();

    sum = 0.0; // Reset
    for (int i = 0; i < 1000000; i++) {
        sum += i * 1.0 / (i + 1);
    }

    end_cycles2 = rdtsc_serialized();

    // Calculate results
    double elapsed_time = (end_time.tv_sec - start_time.tv_sec) +
                          (end_time.tv_nsec - start_time.tv_nsec) / 1e9;
    uint64_t cycles1 = end_cycles1 - start_cycles1;
    uint64_t cycles2 = end_cycles2 - start_cycles2;

    printf("\n=== Results ===\n");
    printf("clock_gettime: %.6f seconds\n", elapsed_time);
    printf("Regular RDTSC: %lu cycles\n", cycles1);
    printf("Serialized RDTSC: %lu cycles\n", cycles2);
    printf("Estimated frequency (from regular): %.2f GHz\n", cycles1 / elapsed_time / 1e9);
    printf("Estimated frequency (from serialized): %.2f GHz\n", cycles2 / elapsed_time / 1e9);
    printf("Cycles per iteration (regular): %.2f\n", (double)cycles1 / 1000000);
    printf("Cycles per iteration (serialized): %.2f\n", (double)cycles2 / 1000000);
    printf("Sum: %.6f\n", sum);
}

// Test RDTSCP availability
void test_rdtscp() {
    printf("\n=== Testing RDTSCP ===\n");

    // Try RDTSCP (might not be available on older CPUs)
    uint64_t start, end;

    start = rdtscp();
```

```
    volatile double sum = 0.0;
    for (int i = 0; i < 100000; i++) {
        sum += i;
    }

    end = rdtscp();

    printf("RDTSCP test: %lu cycles for 100k iterations\n", end - start);
    printf("RDTSCP works: %s\n", (end > start) ? "Yes" : "No");
}

int main() {
    printf("=== Timing Methods Comparison ===\n");

    timing_comparison();
    test_rdtscp();

    printf("\n=== Recommendations ===\n");
    printf("1. For general use: Use clock_gettime() - most reliable\n");
    printf("2. For micro-benchmarking: Use serialized RDTSC with CPU pinning\n");
    printf("3. Always run multiple iterations for statistical validity\n");

    return 0;
}
```