Programming and Architecture of Computing Systems Laboratory 2, Instrumented and Event-based Techniques

Sergio Marquina Rubio, Sonia Rubio Llamas

16 October 2023

1 Introduction

Profiling is a method for dynamically analyzing programs to gather data like memory usage, execution time, instruction usage, and function call details. It serves two main purposes: understanding a program's behavior and helping optimize it. Profiling involves instrumenting the program's code or binary using profiler tools that employ techniques like event-based and statistical methods. In this lab session, we are going to analize different programs in order to know more about execution time and sistem calls.

2 Instrumentation Profiling

In this exercise, we employ manual code instrumentation to measure the execution time of both standard matrix multiplication and Eigen-based programs.

We tested different matrix sizes (N= 100, 500, 1000 and 2000) to observe how execution time varies. These findings provide valuable insights for program optimization and performance analysis.

We examined four different scenarios: standard matmult with clock(), standard matmult with `gettimeofday()`, Eigen-based matmult with `clock()`, and Eigen-based matmult with `gettimeofday()`. The values for both matmult and Eigen are shown in Table 1 and Table 2, respectively.

2.1 `clock()`

We use the clock() function to capture the elapsed time from program launch to the moment the clock() function is invoked. This encompasses matrix declarations, memory allocation, element initialization, and the actual matrix multiplication. Comparing this time values with the ones in Lab 2 (using real and user time), we can see a clearly difference (626 seconds then vs 450 seconds now, for N=2000). This is because `clock()` focuses on CPU time and code execution, while "user" and "real" times offer a more comprehensive view of a program's execution by including system calls, I/O wait time, and other activities.

2.2 `gettimeofday()`

This function is used to obtain the current time with high precision, down to microseconds, valuable for tasks like performance analysis, task scheduling, and timing operations in programs. In this case, we use it to deffer between the computation of the matrix multiplication and all the other processes involved, such as matrix declarations, memory allocations, and initializations.

The clock() values, although they may slightly differ due to execution variance, represent the sum of the initialize and matrix time. As the problem size (N) increases, the matrix multiplication time becomes increasingly dominant in the total execution time, while the initialization time's relative contribution diminishes. This shift occurs because the initialization process, which includes memory allocation and data setup, remains relatively stable in duration, while the matrix multiplication time escalates with larger problem sizes.

		N = 100	**N = 500**	**N = 1000**	**N = 2000**
clock()		0.0077	2.9148	36.3799	450.1592
gettimeofday()	**initialize time**	0.0010	0.0262	0.1274	0.4004
	matrix time	0.0072	3.0289	34.0215	411.6

Table 1: Execution times in seconds (s) for standard matmult

		N = 100	**N = 500**	**N = 1000**	**N = 2000**
clock()		0.0023	0.0914	0.4888	3.1977 s
gettimeofday()	**initialize time**	0.0008	0.0180	0.1042	0.3488
	matrix time	0.0005	0.0666	0.3513	2.6296

Table 2: Execution times in seconds (s) for Eigen

2.3 strace command

The objective of this analysis is to evaluate the system call behavior and performance of those two different matrix multiplication implementations: the standard matrix multiplication and an optimized version using the Eigen library. System calls are crucial indicators of system interaction and can offer insights into program behavior and potential areas of optimization.

We used the strace command with the -c option to generate a summary of system calls for each program. We examined the four different scenarios for a matrix size of N = 1000.

From the obtained *strace* outputs, we can observe differences in the number of system call invocations (calls) between different runs of the program. Here, we'll analyze the variations in selected system calls and reason about these differences:

- **'openat' System Call:** The *openat* system call is invoked 27 times in all the cases. The number of *openat* system calls is consistent across all runs, suggesting similar file-related operations regardless of the approach and timing method.
- **'mmap' System Call:** It's invoked 22 times. The number of mmap system calls remains consistent across all runs, indicating that memory mapping operations (mmap) are similar for both Eigen and standard matrix multiplication approaches regardless of the timing method used.
- **'mprotect' System Call:** For standard matmult, both clock() and gettimeofday() it has 7 calls. For Eigen, 0 calls. This suggests that the Eigen-based implementation doesn't involve memory protection changes while the standard matrix multiplication does.

- **'pread' System Call:** The *pread64* system call also differs between the runs. It's called six times in both the "standard matmult", while in both the "eigen" there are no calls to. This suggests that the Eigen-based implementation doesn't involve *pread64* operations, which are typically used for reading from file descriptors, while the standard matrix multiplication does.

3 Event-based Profiling

Event-based profiling employs hardware performance event counters to track particular types of occurrences that take place while a program is running. In this section we are going to use the command perf, which provide access to the PMU in a CPU, so we can know the behaviour of the hardware and its associated events. We are going to run the perf profiler for four different runs: i) standard matmult + clock, ii) standard matmult + gettimeofday, iii) Eigen + clock, and iv) Eigen + gettimeofday whit a matrix of dimension N=2000.

There is a wide list of available events we can obtain with perf, but we are going to focus in the following ones:

- **task-clock**: The duration, in milliseconds, that the command execution actively utilized the CPU.
- **context-switches**: The number of times the CPUs were switched from one process (or thread) to another.
- **branches**: The count of instruction branches encountered during program execution, representing decision points and loops at the CPU level. Increased branches can lead to lower performance.
- **branch-misses**: In an effort to mitigate performance penalties resulting from numerous branches, modern CPUs employ predictive mechanisms to anticipate code flow. This statistic quantifies the number of instances where these predictions were incorrect, expressed as a percentage relative to the total number of branches.

Firstly we are going to analyze the task-clock. The times in milliseconds we obtain for each case are:

	standard + clock	standard + gettimeofday	eigen + clock	eigen + gettimeofdat
task-clocks	28,864.95	26,986.87	8,337.03	8,875.23

Table 3: task-clocks for each case

We can see when considering task-clock time, utilizing the Eigen library provides a substantial performance advantage over the standard library as we have seen before, and using the clock or the gettimeofday does not modify the times in a significant way.

Now regarding to context-switches:

	standard + clock	standard + gettimeofday	eigen + clock	eigen + gettimeofdat
context-switchs	143	104	49	42

Table 4: context-switchs for each case

The Eigen library generally incurs fewer context-switches compared to the standard library. This is likely due to Eigen's efficient algorithms and optimized execution, resulting in fewer interruptions and context switches. The "clock" function generally results in more context-switches compared to "gettimeofday" because "clock" measures CPU time used by the calling process, and the scheduler may interrupt and switch processes more frequently to measure this accurately.

	standard + clock	standard + gettimeofday	eigen + clock	eigen + gettimeofdat
branches	42,573,198,635	42,571,742,757	2,574,742,890	2,543,968,645

Table 5: branches for each case

The standard library cases have significantly higher branch counts compared to the Eigen library cases. This is likely due to the complexity and design of the standard library, resulting in more decision points and loops. Eigen, being a specialized linear algebra library, is optimized for numerical computations and matrix operations, which involve fewer conditional branches and loops, leading to a substantially lower branch count. The use of "clock" or "gettimeofday" has a negligible impact on the number of branches encountered since these functions primarily measure time and do not significantly affect the control flow of the program.

	standard + clock	standard + gettimeofday	eigen + clock	eigen + gettimeofdat
branch-misses	18,837,374	18,806,932	16,837,326	16,934,635

Table 6: branches-misses for each case

Both Eigen library cases consistently experience fewer branch-misses compared to their respective counterparts in the standard library. This suggests that Eigen's specialized optimizations result in more accurate branch predictions and a more efficient use of the CPU's execution pipeline. Using "clock" or "gettimeofday" does also not have a significant impact on the number of branch-misses in either the standard or Eigen library cases. The branch-miss counts remain relatively consistent regardless of the timing function used.

4 Conclusions

In conclusion profiling, is a really useful tool that helps us to understand how a program operates, how efficiently it performs, and how it interacts with the underlying hardware.