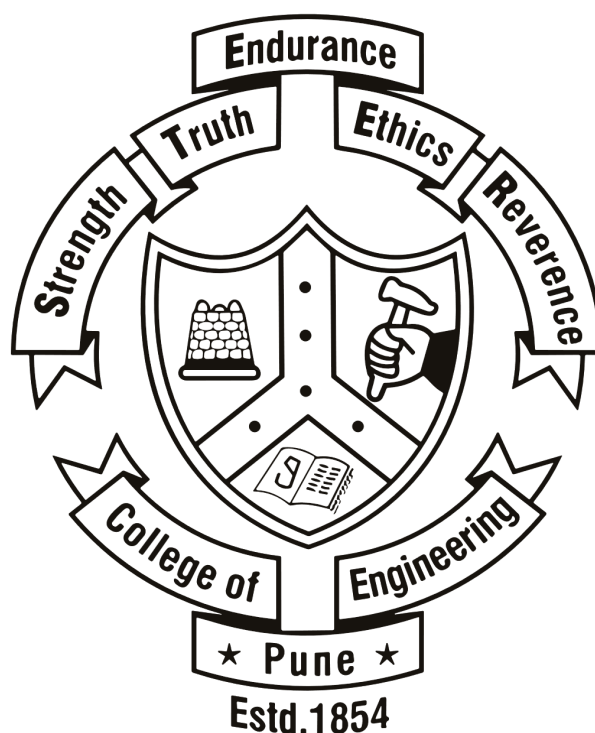


INSTRUCTION MANUAL

CT-21007

COMPUTER ORGANIZATION

perf: Linux profiling with performance counters



Prof. Mr. A. D. Joshi

adj.comp@coeptech.ac.in

COEP Technological University

A Unitary Public University of Government of Maharashtra

(Formerly College of Engineering, Pune)

TABLE OF CONTENTS

1. INTRODUCTION.....	3
1.1 Introduction.....	3
2. INSTALLATION PROCESS.....	5
2.1 Installation Guide.....	5
3. Perf Tool: PERFORMANCE ANALYSIS TOOL FOR LINUX.....	6
3.1 Introduction.....	6
3.2 Commands.....	6
3.3 Events.....	7
4. EXPLANATION ABOUT PRIME CONCERNED.....	8
4.1 Hardware Events.....	8
4.2 Software Events.....	16
5. PROGRAM DEMONSTRATION.....	19
5.1 Naive Approach.....	19
5.2 Tiled Approach.....	23
FINAL OBSERVATIONS.....	27
CONCLUSION.....	28

1. INTRODUCTION

‘perf: *Linux profiling with performance counters*’

1.1 Introduction:

An essential prerequisite for optimizing an application is to first understand its execution characteristics. A number of tools are available for the application developer to accomplish this, ranging from simple shell utilities, timers and profilers, trace analysis tools, to sophisticated full featured graphical toolsets. This tutorial investigates, in varying depths, a number of tools that can be used to analyze an application's performance towards the goals of optimization and trouble-shooting. A variety of profiling and execution analysis tools exist for both serial and parallel programs. They range widely in usefulness and complexity.

Writing large-scale parallel and distributed scientific applications that make optimum use of computational resources is a challenging problem. Very often, resources are under-utilized or used inefficiently. The factors which determine a program's performance are complex, interrelated, and oftentimes, hidden from the programmer. Some of them are listed by category below.

Application Related Factors	Hardware Related Factors	Software Related Factors
Algorithms	Processor Architecture	Operating System
Use of I/O	Memory Hierarchy	Compiler
Load Balancing	I/O Configuration	Preprocessor
Memory Usage Patterns	Network	Communication Protocols
Dataset Sizes		Libraries

Because of these challenges and complexities, performance analysis tools are essential to optimizing an application's performance. They can assist you in understanding what your program is "really doing" and suggest how program performance should be improved.

The most important goal of performance tuning is to reduce a program's wall clock execution time. Reducing resource usage in other areas, such as memory or disk requirements, may also be a tuning goal.

Performance tuning is an iterative process used to optimize the efficiency of a program. It usually involves finding your program's hot spots and eliminating the bottlenecks in them.

- **Hot Spot:**

An area of code within the program that uses a disproportionately high amount of processor time.

- **Bottleneck:**

An area of code within the program that uses processor resources inefficiently and therefore causes unnecessary delays.

Performance tuning usually involves profiling - using software tools to measure a program's run-time characteristics and resource utilization.

There are a numerous amount of Performance analysis tools available on the Internet. Some of which are Proprietary, while others are Free and Open Sourced. Some of the best Tools are : AppDynamics by Cisco, CodeAnalyst by AMD, Dynatrace, perf tools, Windows Performance Analysis Toolkit by Windows, Instruments with Xcode, etc.

But we will be focusing on the **perf tools** by now.

2. INSTALLATION PROCESS

2.1 Installation Guide:

To install perf on your system, perform the following steps on your terminal:

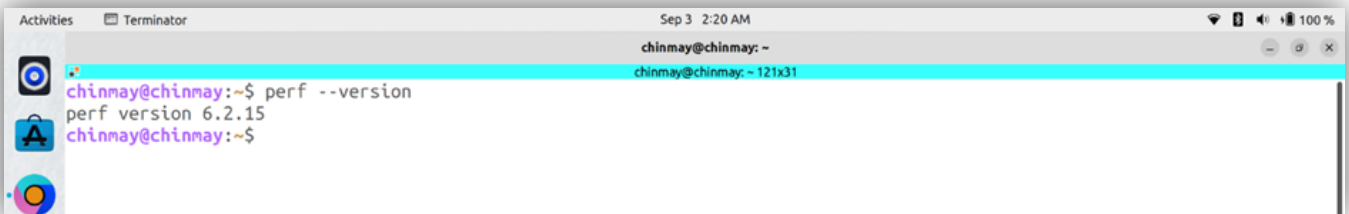
- i. To install type –

sudo apt install linux-tools-\$(uname -r) linux-tools-generic



```
Activities Terminator Sep 3 2:16 AM chinmay@chinmay: ~ chinmay@chinmay: ~ 121x31 chinmay@chinmay:~$ sudo apt install linux-tools-$(uname -r) linux-tools-generic [sudo] password for chinmay: Reading package lists... Done Building dependency tree... Done Reading state information... Done linux-tools-6.2.0-31-generic is already the newest version (6.2.0-31.31~22.04.1). linux-tools-generic is already the newest version (5.15.0.82.78). 0 upgraded, 0 newly installed, 0 to remove and 8 not upgraded. chinmay@chinmay:~$
```

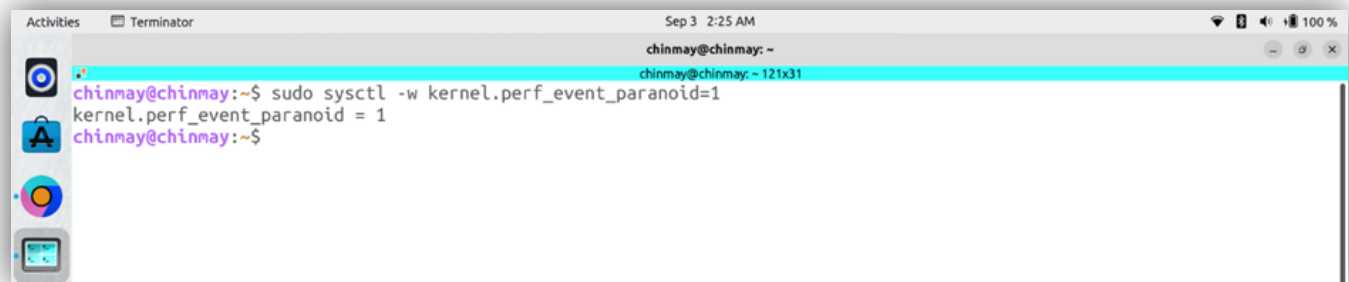
- ii. To verify the installation type–
perf -version



```
Activities Terminator Sep 3 2:20 AM chinmay@chinmay: ~ chinmay@chinmay: ~ 121x31 chinmay@chinmay:~$ perf --version perf version 6.2.15 chinmay@chinmay:~$
```

- iii. The perf command, by default, requires sudo privileges. To allow regular users to use perf, do the following –

sudo sysctl -w kernel.perf_event_paranoid=1



```
Activities Terminator Sep 3 2:25 AM chinmay@chinmay: ~ chinmay@chinmay: ~ 121x31 chinmay@chinmay:~$ sudo sysctl -w kernel.perf_event_paranoid=1 kernel.perf_event_paranoid = 1 chinmay@chinmay:~$
```

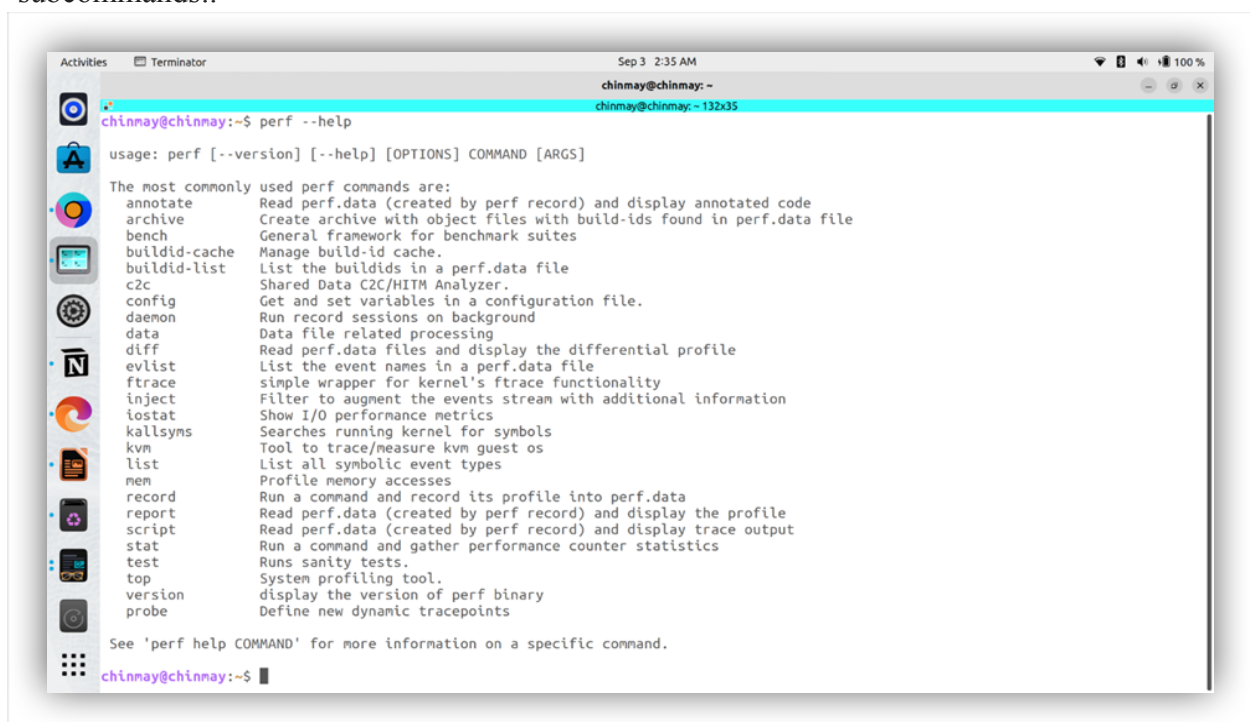
3. Perf Tool: PERFORMANCE ANALYSIS TOOL FOR LINUX

3.1 Introduction:

perf (originally Performance Counters for Linux, PCL) is a performance analyzing tool in Linux, available from Linux kernel version 2.6.31 in 2009. Userspace controlling utility, named perf, is accessed from the command line and provides a number of subcommands; it is capable of statistical profiling of the entire system (both kernel and userland code). Linux provides a performance monitoring and analysis tool called conveniently perf. The Linux perf tool is a lightweight command-line utility for profiling and monitoring CPU performance on Linux systems. Although the tool is simple, it provides in-depth information that helps in analyzing CPUs. Perf provides access to the Performance Monitoring Unit in the CPU, and thus allows us to have a close look at the behavior of the hardware and its associated events. In addition, it can also monitor software events, and create reports out of the data that is collected.

3.2 Commands:

The perf tool offers a rich set of commands to collect and analyze performance and trace data. The command contains many subcommands for collecting, tracing, and analyzing CPU event data. To see the list of commands on your terminal, type – **perf --help**. perf is used with several subcommands:.



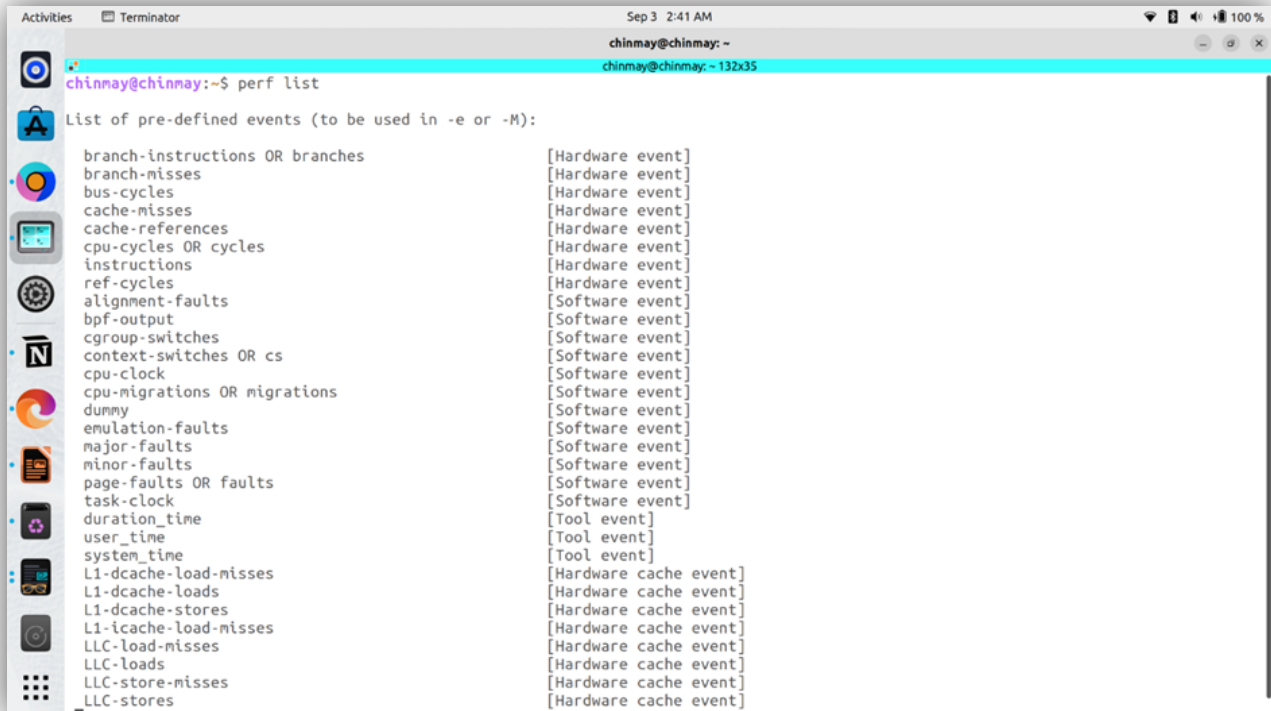
```
Activities Terminator Sep 3 2:35 AM chinmay@chinmay: ~ chinmay@chinmay: ~ 132x35
chinmay@chinmay:~$ perf --help
usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

The most commonly used perf commands are:
  annotate      Read perf.data (created by perf record) and display annotated code
  archive      Create archive with object files with build-ids found in perf.data file
  bench        General framework for benchmark suites
  buildid-cache Manage build-id cache.
  buildid-list List the builds in a perf.data file
  c2c          Shared Data C2C/HITM Analyzer.
  config       Get and set variables in a configuration file.
  daemon       Run record sessions on background
  data         Data file related processing
  diff         Read perf.data files and display the differential profile
  evlist       List the event names in a perf.data file
  ftrace       simple wrapper for kernel's ftrace functionality
  inject       Filter to augment the events stream with additional information
  iostat       Show I/O performance metrics
  kallsyms     Searches running kernel for symbols
  kvm          Tool to trace/measure kvm guest os
  list         List all symbolic event types
  mem         Profile memory accesses
  record       Run a command and record its profile into perf.data
  report       Read perf.data (created by perf record) and display the profile
  script       Read perf.data (created by perf record) and display trace output
  stat        Run a command and gather performance counter statistics
  test        Runs sanity tests.
  top         System profiling tool.
  version     display the version of perf binary
  probe       Define new dynamic tracepoints

See 'perf help COMMAND' for more information on a specific command.
chinmay@chinmay:~$
```

3.3 Events:

The perf tool supports a list of measurable events. The tool and underlying kernel interface can measure events coming from different sources. On each processor, those events get mapped onto actual events provided by the CPU, if they exist, otherwise the event cannot be used. To obtain a list of supported events type – perf list



```
chinmay@chinmay:~$ perf list
List of pre-defined events (to be used in -e or -M):

branch-instructions OR branches      [Hardware event]
branch-misses                        [Hardware event]
bus-cycles                          [Hardware event]
cache-misses                         [Hardware event]
cache-references                     [Hardware event]
cpu-cycles OR cycles                 [Hardware event]
instructions                         [Hardware event]
ref-cycles                           [Hardware event]
alignment-faults                     [Software event]
bpf-output                           [Software event]
cgroup-switches                      [Software event]
context-switches OR cs               [Software event]
cpu-clock                            [Software event]
cpu-migrations OR migrations         [Software event]
dummy                                [Software event]
emulation-faults                     [Software event]
major-faults                         [Software event]
minor-faults                         [Software event]
page-faults OR faults                [Software event]
task-clock                           [Software event]
duration_time                         [Tool event]
user_time                            [Tool event]
system_time                          [Tool event]
L1-dcache-load-misses                [Hardware cache event]
L1-dcache-loads                      [Hardware cache event]
L1-dcache-stores                     [Hardware cache event]
L1-icache-load-misses                [Hardware cache event]
LLC-load-misses                      [Hardware cache event]
LLC-loads                            [Hardware cache event]
LLC-store-misses                     [Hardware cache event]
LLC-stores                           [Hardware cache event]
```



4. EXPLANATION ABOUT PRIME CONCERNED SOFTWARE AND HARDWARE EVENTS

4.1 Hardware Events :

Hardware events refer to events that are generated by the hardware components of the system, such as the CPU, memory subsystem, and other hardware units. These events provide insights into the behavior and performance of the hardware itself, which can be extremely valuable for analyzing and optimizing system performance.

Various Hardware Events include –

i. **branch-instructions OR branches -**

```
sudo perf stat -e branches ./row
```

```
sudo perf stat -e branches ./col
```

A branch is an instruction in a computer program that can cause a computer to begin executing a different instruction sequence and thus deviate from its default behavior of executing instructions in order. Branch instructions are used to implement control flow in program loops and conditionals.

A branch instruction can be either an unconditional branch, which always results in branching, or a conditional branch, which may or may not cause branching depending on some condition. Branch instructions can alter the contents of the CPU's Program Counter (or PC) (or Instruction Pointer on Intel microprocessors). The PC maintains the memory address of the next machine instruction to be fetched and executed. Therefore, a branch, if executed, causes the CPU to execute code from a new memory address, changing the program logic according to the algorithm planned by the programmer.

ii. branch-misses-

```
sudo perf stat -e branch-misses ./row  
sudo perf stat -e branch-misses ./col
```

In Computer Architecture, a branch predictor is a digital circuit that tries to guess which way a branch will go before this is known definitively. The purpose is to improve the flow in the instruction pipeline. Branch predictors play a critical role in achieving high performance in many modern pipelined microprocessor architectures. Without branch prediction, the processor would have to wait until the conditional jump instruction has passed the execute stage before the next instruction can enter the fetch stage in the pipeline.

The branch predictor attempts to avoid this waste of time by trying to guess whether the conditional jump is most likely to be taken or not taken. The branch that is guessed to be the most likely is then fetched and speculatively executed. If it is later detected that the guess was wrong, then the speculatively executed or partially executed instructions are discarded and the pipeline starts over with the correct branch, incurring a delay.

The time that is wasted in case of a branch misprediction is equal to the number of stages in the pipeline from the fetch stage to the execute stage.

iii. Bus-cycles -

```
sudo perf stat -e bus-cycles ./row  
sudo perf stat -e bus-cycles ./col
```

A bus cycle, often referred to as a "bus transaction" or simply a "cycle," is a fundamental concept in computer architecture that describes the basic unit of data transfer between various components within a computer system. It represents the movement of data along the system's buses, which are electrical pathways used for communication between different hardware components. Bus cycle involves the transfer of data between two components connected by a bus. This data can include instructions, data values, memory addresses, or control signals.

- **Types of Bus Cycles:**

- a. **Read Cycle:** In a read cycle, a component (e.g., CPU) requests data from another component (e.g., memory or peripheral device) by placing an address on the bus. The target component retrieves the requested data and places it on the bus for the requesting component to read.
- b. **Write Cycle:** In a write cycle, a component sends data to another component by placing both the address and data on the bus. The target component writes the data to the specified memory location or performs the required operation.
- c. **Control Cycle:** Some bus cycles are used for control purposes, such as signaling to other components or configuring hardware settings. Control cycles don't necessarily involve data transfer.

iv. cache-misses -

```
sudo perf stat -e cache-misses ./row  
sudo perf stat -e cache-misses ./col
```

A cache is a hardware or software component that stores data so that future requests for that data can be served faster. The data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere. Cache memory is used to store frequently accessed data, which can be retrieved quickly when needed. This helps to speed up the performance of the computer or device.

A cache miss occurs when the data that is being requested by a system or an application isn't found in the cache memory. This is in contrast to a cache hit, which refers to when the site content is successfully retrieved and loaded from the cache. In other words, a cache miss is a failure in an attempt to access and retrieve requested data². There are multiple reasons why this might happen. One is that the data was never put into the cache, to begin with. Another possibility is that the data was removed at one point.

Cache misses can slow down computer performance, as the system must wait for the slower data retrieval process to complete. To minimize their impact, computer systems use caching strategies. A cache miss penalty refers to the delay caused by a cache miss. It indicates the extra time a cache spends to fetch data from its memory. Typically, the memory hierarchy consists of three levels, which affect how fast data can be successfully retrieved. Here's the common memory hierarchy found in a cache:

- **L1 cache:** Despite being the smallest in terms of capacity, the primary cache is the easiest to access. L1 accommodates recently-accessed content and has designated memory units in each core of the CPU.
- **L2 cache:** The secondary cache is more extensive than L1 but smaller than L3. It takes longer to access than L1 but is faster than L3. An L2 cache has one designated memory in each core of the CPU.
- **L3 cache:** Often called Last-Level Cache (LLC) or the main database, L3 is the largest and slowest cache memory unit. All cores in a CPU share one L3.

v. cache-references –

```
sudo perf stat -e cache-references ./row  
sudo perf stat -e cache-references ./col
```

Locality of reference refers to a phenomenon in which a computer program tends to access same set of memory locations for a particular time period. In other words, Locality of Reference refers to the tendency of the computer program to access instructions whose addresses are near one another. The property of locality of reference is mainly shown by: Loops in program cause the CPU to repeatedly execute a set of instructions that constitute the loop. Subroutine calls, cause the set of instructions are fetched from memory each time the subroutine gets called. References to data items also get localized, meaning the same data item is referenced again and again. There are two ways in which data or instruction are fetched from main memory then get stored in cache memory:

a. Temporal Locality - Temporal locality means current data or instruction that is being fetched may be needed soon. So we should store that data or instruction in the cache memory so that we can avoid again searching in main memory for the same data.

b. Spatial Locality - Spatial locality means instruction or data near to the current memory location that is being fetched, may be needed by the processor soon. This is different from the temporal locality in that we are making a guess that the data/instructions will be needed soon. With temporal locality we were talking about the actual memory location that was being fetched.

vi. cpu-cycles OR cycles -

```
sudo perf stat -e cpu-cycles ./row
```

```
sudo perf stat -e cpu-cycles ./col
```

CPU cycles, often referred to simply as "cycles," are a fundamental concept in computer architecture and performance measurement. They represent the basic unit of time in which a central processing unit (CPU) processes instructions and performs its operations. CPU cycles are a measure of how quickly the CPU can execute instructions and are closely related to the clock speed of the CPU. The speed of a computer processor, or CPU, is determined by the Clock Cycle, which is the amount of time between two pulses of an oscillator. Generally speaking, the higher number of pulses per second, the faster the computer processor will be able to process information. The clock speed is measured in Hz, typically either megahertz (MHz) or gigahertz (GHz). For example, a 4GHz processor performs 4,000,000,000 clock cycles per second. Computer processors can execute one or more instructions per clock cycle, depending on the type of processor. Early computer processors and slower processors can only execute one instruction per clock cycle, but faster, more advanced processors can execute multiple instructions per clock cycle, processing data more efficiently.

vii. instructions –

```
sudo perf stat -e instructions ./row
```

```
sudo perf stat -e instructions ./col
```

Computer instructions are a set of machine language instructions that a particular processor understands and executes. A computer performs tasks on the basis of the instruction provided. An instruction comprises of groups called fields. These fields include: The Operation code (Opcode) field which specifies the operation to be performed. The Address field which contains the location of the operand, i.e., register or memory location. The Mode field which specifies how the operand will be located.

MODE**OPCODE****OPERAND/ADDRESS**

A computer's instructions can be any length and have any number of addresses. The arrangement of a computer's registers determines the different address fields in the instruction format. The instruction can be classified as three, two, and one address instruction or zero address instruction, depending on the number of address fields.

i. Three Address Instructions: A three-address instruction has the following general format:

source 1 operation, source 2 operation, source 3 operation, destination
ADD X, Y, Z

Here, X, Y, and Z seem to be the three variables that are each assigned to a distinct memory location. The operation implemented on operands is 'ADD.' The source operands are 'X' and 'Y', while the destination operand is 'Z'. In order to determine the three operands, bits are required. To determine one operand, n bits are required (one memory address). In the same way, 3n bits are required to define three operands (or three memory addresses). To identify the ADD operation, bits are also required.

ii. Two Address Instructions: A two-address instruction has the following general format:

source and destination of the operation
ADD X, Y

Here X and Y are the two variables that have been assigned to a specific memory address. The operation performed on the operands is 'ADD.' This command combines the contents of variables X and Y and stores the result in variable Y. The source operand is 'A,' while 'B' is used as both a source and a destination operand. The two operands must be determined using bits. To define one operand, n bits are required (one memory address). To determine two operands, 2n bits are required (two memory addresses). The ADD operation also necessitates the use of bits.

iii. One Address Instructions:

One address instruction has the following general format:

operation source

INCLUDE X

Here X refers to the variable that has access to a specific memory region. The operation performed on operand A is 'ADD.' This instruction adds the value of variable A to the accumulator and then saves the result inside the accumulator by restoring the accumulator's contents.

iv. Zero Address Instructions:

In zero address instructions, the positions of the operands are implicitly represented. These instructions use a structure called a pushdown stack to hold operands. This instruction does not have an operand field, and the location of operands is implicitly represented. The stack-organized computer system supports these instructions. To evaluate the arithmetic expression, it is required to convert it into reverse polish notation.

Consider the below operations, which shows how $X = (A + B) * (C + D)$ expression will be written for a stack-organized computer.

TOS: Top of the Stack

PUSH	A	$TOS \leftarrow A$
PUSH	B	$TOS \leftarrow B$
ADD		$TOS \leftarrow (A + B)$
PUSH	C	$TOS \leftarrow C$
PUSH	D	$TOS \leftarrow D$
ADD		$TOS \leftarrow (C + D)$
MUL		$TOS \leftarrow (C + D) * (A + B)$
POP	X	$M[X] \leftarrow TOS$

viii. ref-cycles –

```
sudo perf stat -e ref-cycles ./row
```

```
sudo perf stat -e ref-cycles ./col
```

A reference cycle is a situation that occurs when two or more objects have strong references to each other, creating a cycle of references that cannot be broken. This can lead to memory leaks, as the objects involved in the cycle cannot be deallocated by the garbage collector. Reference cycles can occur in any language that uses garbage collection, and it is the responsibility of the programmer to avoid creating them. There are several ways to break reference cycles, such as using weak or unowned references.

Memory reference instruction cycles are number of commands or instructions which are in the custom to generate a reference to the memory and approval to a program to have an approach to the commanded information and that states as to from where the data is cache continually. These instructions cycles are known as Memory Reference Instructions Cycles. There are seven memory reference instructions which are as follows:

AND, ADD, LDA, STA, BUN, BSA, IBZ.

4.2 Software Events:

In the context of the "perf" tool in Linux, "software events" refer to specific types of performance events or counters that can be measured and analyzed by "perf" without relying on specialized hardware performance counters. Software events are events generated by the software itself, typically through the use of various instrumentation and tracing techniques. These events provide insights into the behavior and performance of your software, applications, or the Linux kernel. Various software events in perf are listed below:

i. alignment-faults:

```
sudo perf stat -e alignment-faults ./row  
sudo perf stat -e alignment-faults ./col
```

alignment fault" is a type of software event that you can measure to understand how often alignment faults occur in your program.

Alignment faults happen when data or instructions are accessed at addresses that do not adhere to the required alignment constraints.

ii. Bpf-output:

```
sudo perf stat -e bpf-output ./row  
sudo perf stat -e bpf-output ./col
```

The bpf-output event in perf is used to measure the number of output events generated by BPF (Berkeley Packet Filter) programs. BPF is a framework in the Linux kernel that allows you to write and run custom packet filtering and processing programs.

iii. Context-switches:

```
sudo perf stat -e context-switches ./row  
sudo perf stat -e context-switches ./col
```

context-switches," are a critical aspect of operating system and multitasking management. They involve the process of saving the state of one process or thread and restoring the state of another, allowing multiple processes or threads to share the CPU.

iv. Cpu-clock:

```
sudo perf stat -e cpu-clock ./row  
sudo perf stat -e cpu-clock ./col
```

The cpu-clock event in perf is used to measure the time the CPU spends executing instructions for a specific program. It provides information about the total CPU time consumed during the execution of the program.

v. cpu-migrations:

```
sudo perf-stat cpu-migrations ./row  
sudo perf-stat cpu-migrations ./col
```

The cpu-migrations event in perf is used to monitor the number of times a process or thread migrates (moves) from one CPU core to another. CPU migrations occur when the Linux scheduler decides to move a process or thread to a different CPU core.

vi. Dummy:

```
perf stat -e dummy ./row  
perf stat -e dummy ./col
```

In perf a "dummy event" is not a real performance event but rather a placeholder or a way to activate certain functionality without measuring a specific hardware event.

vii. emulation-faults:

```
perf stat -e emulation-faults ./row  
perf stat -e emulation-faults ./col
```

The emulation-faults event in perf is used to measure the number of emulation faults encountered during the execution of a program. Emulation faults occur when a program attempts to execute instructions that are not natively supported by the CPU architecture and must be emulated or translated by the system.

viii. major-faults:

```
perf stat -e major-faults ./row  
perf stat -e major-faults ./col
```

The major-faults event in perf is used to measure the number of major page faults encountered during the execution of a program. Major page faults occur when a program attempts to access a memory page that is not currently in physical RAM and must be loaded from secondary storage, such as swap space or a disk.

ix. minor-faults:

```
perf stat -e minor-faults ./row
```

```
perf stat -e minor-faults ./col
```

The minor-faults event in perf is used to measure the number of minor page faults encountered during the execution of a program. Minor page faults occur when a program attempts to access a memory page that is not currently in physical RAM but is already available in other forms of fast-access memory, such as swap space or other parts of the virtual memory system.

x. page-faults:

```
perf stat -e page-faults ./row
```

```
perf stat -e page-faults ./col
```

The page-faults event in perf is used to measure the number of page faults encountered during the execution of a program. Page faults occur when a program attempts to access a memory page that is not currently in physical RAM and must be loaded from secondary storage, such as swap space or a disk.

xi. task-clock:

```
perf stat -e task-clock ./row
```

```
perf stat -e task-clock ./col
```

The task-clock event in perf is used to measure the wall-clock time consumed by a specific task (process or thread). It represents the total time that the task has been running, including both the time it has spent executing instructions and the time it has been waiting (e.g., due to sleep or I/O operations).

5. PROGRAM DEMONSTRATION

Title: Write a program to multiply two different matrices of size 1024 x 1024.

5.1 Naive Approach:

i. **Method 1 (Row wise implementation):**

Program Name – playR.c

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

#define N 1024

// Function to allocate memory for a matrix
int** create_matrix(int rows, int cols) {
    int** matrix = (int**)malloc(rows * sizeof(int*));
    if (matrix == NULL) {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < rows; i++) {
        matrix[i] = (int*)malloc(cols * sizeof(int));
        if (matrix[i] == NULL) {
            perror("Memory allocation failed");
            exit(EXIT_FAILURE);
        }
    }
    return matrix;
}

// Function to initialize a matrix with zeros
void initialize_matrix(int** matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = 0;
        }
    }
}
```

```
// Function to multiply two matrices row-wise
void multiply_row_wise(int** mat1, int** mat2, int** result) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                result[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }
}

// Function to free memory allocated for a matrix
void free_matrix(int** matrix, int rows) {
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

int main() {
    srand(time(NULL)); // Seed the random number generator

    // Create two random matrices
    int** matrix_a = create_matrix(N, N);
    int** matrix_b = create_matrix(N, N);
    int** result_row_wise = create_matrix(N, N);

    // Initialize matrices with random values
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            matrix_a[i][j] = rand() % 10; // Random values between 0 and 9
            matrix_b[i][j] = rand() % 10;
        }
    }

    // Perform row-wise multiplication
    initialize_matrix(result_row_wise, N, N);
    multiply_row_wise(matrix_a, matrix_b, result_row_wise);

    // Free allocated memory
    free_matrix(matrix_a, N);
    free_matrix(matrix_b, N);
    free_matrix(result_row_wise, N);

    return 0;
}
```

ii. Method 2 (Column wise implementation):**Program Name– playC.c**

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

#define N 1024

// Function to allocate memory for a matrix
int** create_matrix(int rows, int cols) {
    int** matrix = (int**)malloc(rows * sizeof(int*));
    if (matrix == NULL) {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < rows; i++) {
        matrix[i] = (int*)malloc(cols * sizeof(int));
        if (matrix[i] == NULL) {
            perror("Memory allocation failed");
            exit(EXIT_FAILURE);
        }
    }
    return matrix;
}

// Function to initialize a matrix with zeros
void initialize_matrix(int** matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = 0;
        }
    }
}

// Function to multiply two matrices column-wise
void multiply_column_wise(int** mat1, int** mat2, int** result) {
    for (int j = 0; j < N; j++) {
        for (int i = 0; i < N; i++) {
            for (int k = 0; k < N; k++) {
                result[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }
}
```

```
// Function to free memory allocated for a matrix
void free_matrix(int** matrix, int rows) {
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

int main() {
    srand(time(NULL)); // Seed the random number generator

    // Create two random matrices
    int** matrix_a = create_matrix(N, N);
    int** matrix_b = create_matrix(N, N);
    int** result_column_wise = create_matrix(N, N);

    // Initialize matrices with random values
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            matrix_a[i][j] = rand() % 10; // Random values between 0 and 9
            matrix_b[i][j] = rand() % 10;
        }
    }

    // Perform column-wise multiplication
    initialize_matrix(result_column_wise, N, N);
    multiply_column_wise(matrix_a, matrix_b, result_column_wise);

    // Free allocated memory
    free_matrix(matrix_a, N);
    free_matrix(matrix_b, N);
    free_matrix(result_column_wise, N);

    return 0;
}
```



5.2 Tiled Approach:

i. Method 1 (Row wise implementation):

Program Name – playRT.c

With TILE_SIZE : 16

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

#define N 1024
#define TILE_SIZE 16

// Function to allocate memory for a matrix
int** allocateMatrix(int rows, int cols) {
    int** matrix = (int*)malloc(rows * sizeof(int));
    for (int i = 0; i < rows; i++) {
        matrix[i] = (int*)malloc(cols * sizeof(int));
    }
    return matrix;
}

// Function to deallocate memory for a matrix
void deallocateMatrix(int** matrix, int rows) {
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

// Function to initialize a matrix with random values
void initializeMatrix(int** matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = rand() % 100; // Filling with random values between
0 and 99
        }
    }
}
```

```
// Function to multiply two matrices row-wise using a tile-based approach
void multiplyMatrices(int** A, int** B, int** C) {
    for (int i = 0; i < N; i += TILE_SIZE) {
        for (int j = 0; j < N; j += TILE_SIZE) {
            for (int k = 0; k < N; k += TILE_SIZE) {
                for (int ii = i; ii < i + TILE_SIZE && ii < N; ii++) {
                    for (int jj = j; jj < j + TILE_SIZE && jj < N; jj++) {
                        for (int kk = k; kk < k + TILE_SIZE && kk < N; kk++) {
                            C[ii][jj] += A[ii][kk] * B[kk][jj];
                        }
                    }
                }
            }
        }
    }
}

int main() {
    // Allocate memory for matrices A, B, and C
    int** A = allocateMatrix(N, N);
    int** B = allocateMatrix(N, N);
    int** C = allocateMatrix(N, N);

    // Initialize matrices A and B with random values
    initializeMatrix(A, N, N);
    initializeMatrix(B, N, N);

    // Multiply matrices A and B
    multiplyMatrices(A, B, C);

    // Print the result matrix C (optional)
    printf("Resultant Matrix C:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    deallocateMatrix(A, N);
    deallocateMatrix(B, N);
    deallocateMatrix(C, N); // Don't forget to free the memory for matrix C

    return 0;
}
```


With TILE_SIZE : 32

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

#define N 1024
#define TILE_SIZE 32

// Function to allocate memory for a matrix
int** allocateMatrix(int rows, int cols) {
    int** matrix = (int**)malloc(rows * sizeof(int));
    for (int i = 0; i < rows; i++) {
        matrix[i] = (int*)malloc(cols * sizeof(int));
    }
    return matrix;
}

// Function to deallocate memory for a matrix
void deallocateMatrix(int** matrix, int rows) {
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

// Function to initialize a matrix with random values
void initializeMatrix(int** matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = rand() % 100; // Filling with random values between
0 and 99
        }
    }
}
```

```
// Function to multiply two matrices row-wise using a tile-based approach
void multiplyMatrices(int** A, int** B, int** C) {
    for (int i = 0; i < N; i += TILE_SIZE) {
        for (int j = 0; j < N; j += TILE_SIZE) {
            for (int k = 0; k < N; k += TILE_SIZE) {
                for (int ii = i; ii < i + TILE_SIZE && ii < N; ii++) {
                    for (int jj = j; jj < j + TILE_SIZE && jj < N; jj++) {
                        for (int kk = k; kk < k + TILE_SIZE && kk < N; kk++) {
                            C[ii][jj] += A[ii][kk] * B[kk][jj];
                        }
                    }
                }
            }
        }
    }
}

int main() {
    // Allocate memory for matrices A, B, and C
    int** A = allocateMatrix(N, N);
    int** B = allocateMatrix(N, N);
    int** C = allocateMatrix(N, N);

    // Initialize matrices A and B with random values
    initializeMatrix(A, N, N);
    initializeMatrix(B, N, N);

    // Multiply matrices A and B
    multiplyMatrices(A, B, C);

    // Print the result matrix C (optional)
    printf("Resultant Matrix C:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    deallocateMatrix(A, N);
    deallocateMatrix(B, N);
    deallocateMatrix(C, N); // Don't forget to free the memory for matrix C

    return 0;
}
```

With TILE_SIZE : 64

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

#define N 1024
#define TILE_SIZE 64

// Function to allocate memory for a matrix
int** allocateMatrix(int rows, int cols) {
    int** matrix = (int**)malloc(rows * sizeof(int));
    for (int i = 0; i < rows; i++) {
        matrix[i] = (int*)malloc(cols * sizeof(int));
    }
    return matrix;
}

// Function to deallocate memory for a matrix
void deallocateMatrix(int** matrix, int rows) {
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

// Function to initialize a matrix with random values
void initializeMatrix(int** matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = rand() % 100; // Filling with random values between
0 and 99
        }
    }
}
```

```
// Function to multiply two matrices row-wise using a tile-based approach
void multiplyMatrices(int** A, int** B, int** C) {
    for (int i = 0; i < N; i += TILE_SIZE) {
        for (int j = 0; j < N; j += TILE_SIZE) {
            for (int k = 0; k < N; k += TILE_SIZE) {
                for (int ii = i; ii < i + TILE_SIZE && ii < N; ii++) {
                    for (int jj = j; jj < j + TILE_SIZE && jj < N; jj++) {
                        for (int kk = k; kk < k + TILE_SIZE && kk < N; kk++) {
                            C[ii][jj] += A[ii][kk] * B[kk][jj];
                        }
                    }
                }
            }
        }
    }
}

int main() {
    // Allocate memory for matrices A, B, and C
    int** A = allocateMatrix(N, N);
    int** B = allocateMatrix(N, N);
    int** C = allocateMatrix(N, N);

    // Initialize matrices A and B with random values
    initializeMatrix(A, N, N);
    initializeMatrix(B, N, N);

    // Multiply matrices A and B
    multiplyMatrices(A, B, C);

    // Print the result matrix C (optional)
    printf("Resultant Matrix C:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    deallocateMatrix(A, N);
    deallocateMatrix(B, N);
    deallocateMatrix(C, N); // Don't forget to free the memory for matrix C

    return 0;
}
```

iii. Method 2 (Column wise implementation):**Program Name– playCT.c**

With TILE_SIZE : 16

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

#define N 1024
#define TILE_SIZE 16

// Function to allocate memory for a matrix
int** allocateMatrix(int rows, int cols) {
    int** matrix = (int**)malloc(rows * sizeof(int));
    for (int i = 0; i < rows; i++) {
        matrix[i] = (int*)malloc(cols * sizeof(int));
    }
    return matrix;
}

// Function to deallocate memory for a matrix
void deallocateMatrix(int** matrix, int rows) {
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

// Function to initialize a matrix with random values
void initializeMatrix(int** matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = rand() % 100; // Filling with random values between
0 and 99
        }
    }
}
```

```
// Function to multiply two matrices column-wise using a tile-based approach
void multiplyMatrices(int** A, int** B, int** C) {
    for (int j = 0; j < N; j += TILE_SIZE) {
        for (int i = 0; i < N; i += TILE_SIZE) {
            for (int k = 0; k < N; k += TILE_SIZE) {
                for (int jj = j; jj < j + TILE_SIZE && jj < N; jj++) {
                    for (int ii = i; ii < i + TILE_SIZE && ii < N; ii++) {
                        for (int kk = k; kk < k + TILE_SIZE && kk < N; kk++) {
                            C[ii][jj] += A[ii][kk] * B[kk][jj];
                        }
                    }
                }
            }
        }
    }
}

int main() {
    // Allocate memory for matrices A, B, and C
    int** A = allocateMatrix(N, N);
    int** B = allocateMatrix(N, N);
    int** C = allocateMatrix(N, N);

    // Initialize matrices A and B with random values
    initializeMatrix(A, N, N);
    initializeMatrix(B, N, N);

    // Multiply matrices A and B column-wise
    multiplyMatrices(A, B, C);

    // Print the result matrix C
    printf("Resultant Matrix C:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    // Free allocated memory for matrices
    deallocateMatrix(A, N);
    deallocateMatrix(B, N);
    deallocateMatrix(C, N); // Don't forget to free the memory for matrix C

    return 0;
}
```

With TILE_SIZE : 32

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

#define N 1024
#define TILE_SIZE 32

// Function to allocate memory for a matrix
int** allocateMatrix(int rows, int cols) {
    int** matrix = (int**)malloc(rows * sizeof(int));
    for (int i = 0; i < rows; i++) {
        matrix[i] = (int*)malloc(cols * sizeof(int));
    }
    return matrix;
}

// Function to deallocate memory for a matrix
void deallocateMatrix(int** matrix, int rows) {
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

// Function to initialize a matrix with random values
void initializeMatrix(int** matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = rand() % 100; // Filling with random values between
0 and 99
        }
    }
}
```

```

// Function to multiply two matrices column-wise using a tile-based approach
void multiplyMatrices(int** A, int** B, int** C) {
    for (int j = 0; j < N; j += TILE_SIZE) {
        for (int i = 0; i < N; i += TILE_SIZE) {
            for (int k = 0; k < N; k += TILE_SIZE) {
                for (int jj = j; jj < j + TILE_SIZE && jj < N; jj++) {
                    for (int ii = i; ii < i + TILE_SIZE && ii < N; ii++) {
                        for (int kk = k; kk < k + TILE_SIZE && kk < N; kk++) {
                            C[ii][jj] += A[ii][kk] * B[kk][jj];
                        }
                    }
                }
            }
        }
    }
}

int main() {
    // Allocate memory for matrices A, B, and C
    int** A = allocateMatrix(N, N);
    int** B = allocateMatrix(N, N);
    int** C = allocateMatrix(N, N);

    // Initialize matrices A and B with random values
    initializeMatrix(A, N, N);
    initializeMatrix(B, N, N);

    // Multiply matrices A and B column-wise
    multiplyMatrices(A, B, C);

    // Print the result matrix C
    printf("Resultant Matrix C:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    // Free allocated memory for matrices
    deallocateMatrix(A, N);
    deallocateMatrix(B, N);
    deallocateMatrix(C, N); // Don't forget to free the memory for matrix C

    return 0;
}

```


With TILE_SIZE : 64

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

#define N 1024
#define TILE_SIZE 64

// Function to allocate memory for a matrix
int** allocateMatrix(int rows, int cols) {
    int** matrix = (int**)malloc(rows * sizeof(int));
    for (int i = 0; i < rows; i++) {
        matrix[i] = (int*)malloc(cols * sizeof(int));
    }
    return matrix;
}

// Function to deallocate memory for a matrix
void deallocateMatrix(int** matrix, int rows) {
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

// Function to initialize a matrix with random values
void initializeMatrix(int** matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = rand() % 100; // Filling with random values between
0 and 99
        }
    }
}
```

```

// Function to multiply two matrices column-wise using a tile-based approach
void multiplyMatrices(int** A, int** B, int** C) {
    for (int j = 0; j < N; j += TILE_SIZE) {
        for (int i = 0; i < N; i += TILE_SIZE) {
            for (int k = 0; k < N; k += TILE_SIZE) {
                for (int jj = j; jj < j + TILE_SIZE && jj < N; jj++) {
                    for (int ii = i; ii < i + TILE_SIZE && ii < N; ii++) {
                        for (int kk = k; kk < k + TILE_SIZE && kk < N; kk++) {
                            C[ii][jj] += A[ii][kk] * B[kk][jj];
                        }
                    }
                }
            }
        }
    }
}

int main() {
    // Allocate memory for matrices A, B, and C
    int** A = allocateMatrix(N, N);
    int** B = allocateMatrix(N, N);
    int** C = allocateMatrix(N, N);

    // Initialize matrices A and B with random values
    initializeMatrix(A, N, N);
    initializeMatrix(B, N, N);

    // Multiply matrices A and B column-wise
    multiplyMatrices(A, B, C);

    // Print the result matrix C
    printf("Resultant Matrix C:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    // Free allocated memory for matrices
    deallocateMatrix(A, N);
    deallocateMatrix(B, N);
    deallocateMatrix(C, N); // Don't forget to free the memory for matrix C

    return 0;
}

```

FINAL OBSERVATIONS

After executing the matrix multiplication programs using both row-wise and column-wise methods, you can observe the following conclusions and insights:

1. Computational Time: The computational time taken by the row-wise multiplication method and the column-wise multiplication method will likely differ. In most cases, the row-wise multiplication method will be faster because of memory access patterns that are more cache-friendly. Column-wise multiplication involves non-contiguous memory accesses, which can result in more cache misses and slower execution.

2. Cache Performance: The row-wise multiplication method typically exhibits better cache performance due to the sequential access pattern along rows. This means that data from the same row is frequently accessed together, leading to fewer cache misses.

3. Cache Misses: The column-wise multiplication method may experience more cache misses due to its non-sequential access pattern along columns. As a result, it may have a longer memory access latency.

4. Memory Access Patterns: Row-wise multiplication results in better memory access patterns as it accesses elements in a contiguous fashion within a row, optimizing cache usage. In contrast, column-wise multiplication accesses elements less efficiently, potentially leading to more cache thrashing.

5. Matrix Data: If the matrices have certain characteristics, such as being sparse (containing many zero elements) or having specific patterns, the performance difference between row-wise and column-wise multiplication may vary.

6. Cache Size: The size and architecture of the CPU cache can impact the performance difference between the two methods. A larger cache may mitigate some of the performance differences.

7. Compiler and Hardware: The behavior of the programs may vary based on the compiler optimizations and the specific hardware on which they are executed. Different compilers and CPU architectures may exhibit different performance characteristics.

In summary, the choice between row-wise and column-wise matrix multiplication depends on various factors, including the specific hardware, memory access patterns, and the level of optimization applied. Row-wise multiplication tends to be more efficient in many cases due to its cache-friendly access pattern, but it's essential to consider the particular requirements of your application and hardware when selecting the appropriate method for matrix multiplication. Performance profiling tools, like ``perf``, can be valuable for analyzing and optimizing such computations.

CONCLUSION

Based on the matrix multiplication programs provided and the use of the 'perf' tool, we can draw conclusion:

1. Row-wise vs. Column-wise Multiplication: The programs clearly demonstrate that row-wise matrix multiplication is generally more efficient than column-wise multiplication. This is due to the better cache performance and memory access patterns associated with row-wise multiplication. The programs highlight the impact of memory access patterns on computational performance.

2. Cache Performance: 'perf' can be a valuable tool for quantifying the cache performance differences between the two methods. It can measure metrics like cache hits, cache misses, and cache utilization. By analyzing these metrics with 'perf', you can gain insights into how memory access patterns affect cache behavior and, consequently, execution speed.

3. Optimization Potential: Both programs provide opportunities for optimization. Row-wise multiplication, in particular, benefits from loop unrolling and other compiler optimizations. The 'perf' tool can help identify areas where optimization efforts are most beneficial by pinpointing performance bottlenecks.

4. Performance Analysis: 'perf' allows for in-depth performance analysis, including profiling CPU usage, memory access patterns, and hardware counters. It can reveal details about instruction counts, cache misses, and other performance-critical metrics, aiding in performance optimization.

5. Profiling Insights: Profiling with 'perf' helps in pinpointing areas of code that may require optimization. For example, it can identify which specific functions or loops consume the most CPU cycles or experience cache misses. This information is invaluable for making targeted improvements.

6. Hardware Impact: The performance of the programs and the insights provided by 'perf' can be influenced by the specific hardware and cache architecture of the machine on which they are executed. Different hardware configurations may yield varying performance results.

7. Real-world Application: While the programs focus on matrix multiplication as a simplified example, the lessons learned can be applied to real-world applications where efficient memory access patterns are essential for achieving optimal performance. In summary, the matrix multiplication programs, and the use of the 'perf' tool illustrate the importance of memory access patterns, cache optimization, and performance analysis in software development. They showcase how careful consideration of these factors can lead to more efficient code execution and better overall program performance.