LAB-5

Advance Software Development Lab
CS618

Submitted in Partial Fulfilment of the Requirements
for the Degree of Master of Engineering in Computer Science
of Indian Institute of Technology, Dharwad

by

Mridul Chandrawanshi
CS24MT002

School of Computer Science and Engineering
2024

Contents

Contents i

List of Tables ii

1 Introduction 1

1.1 G-Prof 1

1.2 Perf 2

1.3 Comparison Between gprof and perf 5

2 Task 1: Matrix Multiplication 7

2.1 Gprof Output 9

2.2 Perf output 10

2.3 Inferences 12

2.3.1 GProf 12

2.3.2 Perf 13

3 Task 2: Edge Detection 16

3.1 Gprof Outputs 18

3.2 Perf Outputs 18

3.3 Inferences 20

3.3.1 GProf 20

3.3.2 Perf 22

3.3.3 Conclusion 23

3.4 Final Outline 24

3.4.1 Supporting Inferences 25

List of Tables

2.1 Matrix Chain Multiplication of 100.txt 9

2.2 Matrix Chain Multiplication of 300.txt 9

2.3 Matrix Chain Multiplication of 800.txt 9

2.4 Matrix Chain Multiplication of 1000.txt 9

2.5 Performance Metrics of Matrix Chain Multiplication with 100.txt 10

2.6 Performance Metrics of Matrix Chain Multiplication with 300.txt 10

2.7 Performance Metrics of Matrix Chain Multiplication with 500.txt 11

2.8 Performance Metrics of Matrix Chain Multiplication with 800.txt 11

2.9 Performance Metrics of Matrix Chain Multiplication with 1000.txt 12

3.1 Input Image as Chess.pgm 18

3.2 Input Image as Face.pgm 18

3.3 Input Image as Smallest.pgm 18

3.4 Input image as Chess.pgm 18

3.5 Input image as Face.pgm 19

3.6 Input image as Smallest.pgm 19

Chapter 1

Introduction

1.1 G-Prof

Gprof is a profiling tool used for analyzing the performance of programs written in C, C++. It is designed to help developers understand how their code executes, identify bottlenecks, and optimize performance.

Few Gprof commands:

1. Compiling with GProf

```
gcc -pg -o <Executable_file> <Input_file.c>
```
(1.1)

2. Generating Gprof report

```
./<Executable_file >
```
(1.2)

3. Generating Profiling report

```
gprof ./<Executable_file> > analysis.txt
```
(1.3)

1.2 Perf

Perf is a powerful performance monitoring and profiling tool for Linux. It provides insights into system performance and helps identify bottlenecks in applications. It is highly versatile and can be used for both low-level performance analysis and high-level profiling.

Few Perf commands:

1. Basic performance stats

$$\texttt{perf stat ./<Executable\_file >} \tag{1.4}$$

2. Recording performance Data

$$\texttt{perf record ./<Executable\_file >} \tag{1.5}$$

3. Analyzing Recorded Data

$$\texttt{perf report} \tag{1.6}$$

4. Real time performance analysis

$$\texttt{perf top} \tag{1.7}$$

The output provided by gprof are as follows:

1. **Percentage of Time (%time)**: Indicates the proportion of total execution time taken by the function; lower values are better.
2. **Cumulative Seconds**: Total time spent in the function and its children; lower values are better.
3. **Self Seconds**: Time spent in the function itself, excluding children; lower values are better.
4. **Self Milliseconds per Call (self ms/call)**: Average time spent in the function per call; lower values are better.
5. **Total Milliseconds per Call (total ms/call)**: Average time per call including time in children; lower values are better.

The output provided by the perf stat are Task clock Time, Context switches, CPU Migrations, Page faults, Cycles, Instructions, Branch / Branch misses and Time elasped; while the perf report provides info about the overhead.

These indicate:

1. **Task Clock Time**: Lower values are better, indicating less time spent in executing the task.
2. **Context Switches**: Fewer context switches are better as they suggest less overhead from switching between processes or threads.
3. **CPU Migrations**: Similar to context switches, fewer CPU migrations indicate more efficient CPU utilization.
4. **Page Faults**: Fewer page faults are preferable, indicating fewer interruptions to access memory.
5. **Cycles**: Fewer CPU cycles are better, as they indicate less CPU time required to complete the task.
6. **Instructions**: Fewer instructions can be better, but it should be considered in context with other metrics. Sometimes fewer instructions might not necessarily indicate better performance if they result in longer cycles or more complex execution.
7. **Branches and Branch Misses**: Fewer branches and branch misses are generally better. Branch misses can indicate inefficiencies in the execution path.

8. **Time Elapsed**: Lower time elapsed is better, indicating faster completion of the task.
9. **Overhead (Perf Report)**: Lower overhead indicates that less of the total execution time is being consumed by the performance monitoring itself, suggesting more accurate measurements.

1.3 Comparison Between gprof and perf

Both ‘gprof‘ and ‘perf‘ are profiling tools used to analyze the performance of programs, but they differ in their approach and the kind of metrics they provide.

Gprof

- **Metrics Provided**:
  - %time: Percentage of time spent in each function.
  - Cumulative seconds: Total time spent in the function and its children.
  - Self seconds: Time spent in the function itself.
  - Calls: Number of times a function is called.
  - Self ms/call: Average time spent in the function per call.
  - Total ms/call: Average time per call including children.
- **Strengths**:
  - Provides a detailed breakdown of time spent in each function and the call hierarchy.
  - Easy to use and understand, especially for small to medium-sized programs.
- **Limitations**:
  - May not provide as detailed low-level metrics as ‘perf‘.
  - Profiling overhead can affect the accuracy of the measurements.

Perf

- **Metrics Provided**:
  - Task Clock: Total time spent on the task.
  - Context Switches: Number of times the CPU switches between processes or threads.
  - CPU Migrations: Number of times a task is moved between CPUs.
  - Page Faults: Number of times the program accesses invalid memory.
  - Cycles: Number of CPU cycles used.

- Instructions: Number of CPU instructions executed.
- Branches: Number of branch instructions.
- Branch Misses: Number of branch mispredictions.
- Time Elapsed: Total time elapsed during the profiling.
- Overhead: Percentage of time consumed by performance monitoring.

- **Strengths**:
  - Provides a comprehensive set of low-level performance metrics.
  - More precise and detailed insights into CPU usage and system-level performance.
  - Can profile large-scale applications with minimal overhead.
- **Limitations**:
  - More complex to set up and use compared to ‘gprof‘.
  - May require more expertise to interpret the results effectively.

Comparison Summary

- **Granularity**: ‘perf‘ offers more detailed low-level metrics, including CPU cycles, context switches, and page faults, while ‘gprof‘ focuses on function-level profiling.
- **Ease of Use**: ‘gprof‘ is easier to use and interpret for simpler use cases, whereas ‘perf‘ provides deeper insights but can be more complex to configure and understand.
- **Profiling Overhead**: ‘perf‘ typically has lower overhead and provides more accurate measurements for large-scale applications compared to ‘gprof‘.
- **Use Case Suitability**: ‘gprof‘ is suitable for straightforward function-level profiling in smaller applications, while ‘perf‘ is ideal for in-depth performance analysis and profiling in large, complex systems.

Chapter 2

Task 1: Matrix Multiplication

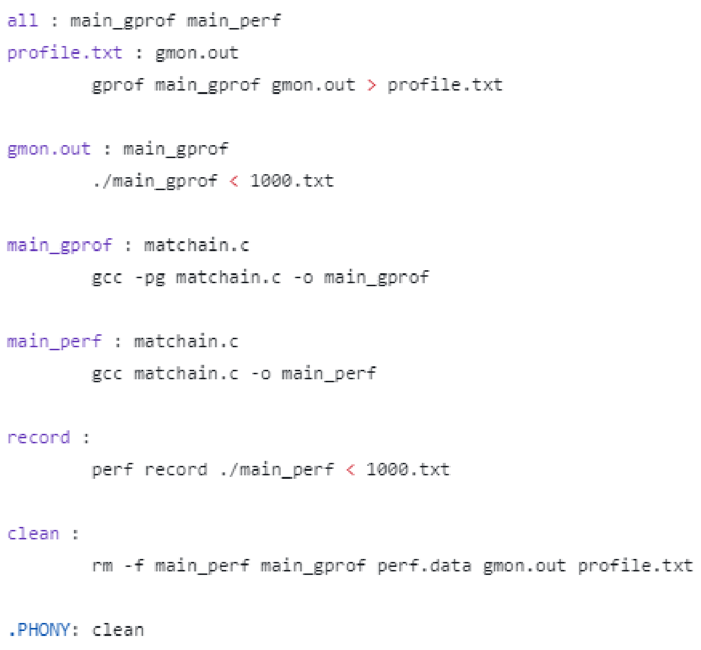
Objective : To determine the optimal sequence for multiplying a sequence of matrices to minimize the total computational cost, rather than performing the actual multiplications. Analyzing the provided implementations to identify the best multiplication order using 2 different profiling tools: gprof and perf.

Inputs:

1. A matchain.c file with 4 matrices
2. 5 Text files corresponding to different inputs

Steps performed: Upon importing the designated file and compiling with *gcc -pg -o matchain* it was run over in GDB. However, there seemed to be no errors in the code and upon commenting/re-commenting it was observed that the 4 matchain.c had 4 distinct matrices definition labelled as Matrixv1, v2, v3 and v4.

To do away with the redundancy of compiling for every matrix for their corresponding input texts a makefile with following properties was created.


```
all : main_gprof main_perf
profile.txt : gmon.out
	gprof main_gprof gmon.out > profile.txt

gmon.out : main_gprof
	./main_gprof < 1000.txt

main_gprof : matchain.c
	gcc -pg matchain.c -o main_gprof

main_perf : matchain.c
	gcc matchain.c -o main_perf

record :
	perf record ./main_perf < 1000.txt

clean :
	rm -f main_perf main_gprof perf.data gmon.out profile.txt

.PHONY: clean
```

Figure 2.1: Image

Wherein only the input files had to be changed to get the time profiling of the given data. Then the following were executed pertaining to every input file to get the desired output.

1. Creating both the Executables

   ```
   make
   ```

2. Compiling and running the executable

   ```
   ./<Executable_file> < <Input_matrix>
   ```

3. After obtaining the gmon.out redirecting it to profile.txt

   ```
   vim profile.txt
   ```

4. Invoking perf and getting perf.data

   ```
   make record
   ```

5. Getting detailed analysis of performance report

   ```
   perf report
   ```

6. Getting stats of the running program

```
perf stat ./<Executable_file>
```

Output: Corresponding to different inputs the following outputs were observed.

2.1 Gprof Output

Implementation	%time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
Matrix V1	33.33	0.03	0.01	3	3.33	3.33	multiply_matrix_v1
Matrix V2	50.00	0.02	0.01	3	3.33	3.33	multiply_matrix_v2
Matrix V3	16.67	0.05	0.01	3	3.33	6.67	multiply_matrix_v3
Matrix V4	0.00	0.05	0.00	3	0.00	3.33	multiply_matrix_v4

Table 2.1: Matrix Chain Multiplication of 100.txt

Implementation	%time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
Matrix V1	84.62	0.11	0.11	1	110.00	110.00	multiply_matrix_v1
Matrix V2	90.00	0.09	0.09	1	90.00	90.00	multiply_matrix_v2
Matrix V3	75.00	0.06	0.06	1	60.00	70.00	multiply_matrix_v3
Matrix V4	75.00	0.03	0.03	1	30.00	30.00	multiply_matrix_v4

Table 2.2: Matrix Chain Multiplication of 300.txt

For data file 500.txt it was observed that computation for matrix multiplication given by all the matrices appears to be 0.

Implementation	%time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
Matrix V1	98.65	2.20	2.20	1	2.20	2.20	multiply_matrix_v1
Matrix V2	98.71	1.53	1.53	1	1.53	1.53	multiply_matrix_v2
Matrix V3	100.00	1.14	1.14	1	1.14	1.14	multiply_matrix_v3
Matrix V4	95.83	0.46	0.46	1	460.00	460.00	multiply_matrix_v4

Table 2.3: Matrix Chain Multiplication of 800.txt

Implementation	%time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
Matrix V1	99.70	13.40	13.40	3	4.47	4.47	multiply_matrix_v1
Matrix V2	99.68	9.24	9.24	3	3.08	3.08	multiply_matrix_v2
Matrix V3	99.56	6.78	6.78	3	2.26	2.26	multiply_matrix_v3
Matrix V4	98.50	2.63	2.63	3	0.88	0.88	multiply_matrix_v4

Table 2.4: Matrix Chain Multiplication of 1000.txt

2.2 Perf output

Evaluation Metric	Matrix V1	Matrix 2	Matrix 3	Matrix 4
Task Clock	0.98 msec	0.95 msec	1.21 msec	1.13 msec
Context Switches	0	0	0	0
CPU Migrations	0	0	0	0
Page Faults	94	92	93	93
Cycles	2,131,740	2,059,304	2,022,682	2,013,464
Instructions	1,985,581	1,968,995	1,984,186	1,973,993
Branches	368,745	365,902	366,764	366,506
Branch Misses	12,662	12,456	12,564	12,449
Time Elapsed	0.0015509399 sec	0.001446777 sec	0.001857908 sec	0.001727942 sec
Overhead (Perf Report)	30.93%	29.86%	16.01%	6.43%

Table 2.5: Performance Metrics of Matrix Chain Multiplication with 100.txt

Evaluation Metric	Matrix V1	Matrix 2	Matrix 3	Matrix 4
Task Clock	2.37 msec	0.94 msec	1.07 msec	0.96 msec
Context Switches	0	0	0	0
CPU Migrations	0	0	0	0
Page Faults	92	93	92	93
Cycles	1,879,597	2,038,573	2,013,141	2,078,814
Instructions	1,993,500	1,984,735	1,971,260	1,980,997
Branches	369,256	368,220	365,491	367,801
Branch Misses	12,900	12,508	12,385	12,710
Time Elapsed	0.003485286 sec	0.001478760 sec	0.001685539 sec	0.001499726 sec
Overhead (Perf Report)	29.86%	73.62%	66.01%	45.53%

Table 2.6: Performance Metrics of Matrix Chain Multiplication with 300.txt

Evaluation Metric	**Matrix V1**	**Matrix 2**	**Matrix 3**	**Matrix 4**
Task Clock	0.94 msec	1.08 msec	0.94 msec	1.08 msec
Context Switches	0	0	0	0
CPU Migrations	0	0	0	0
Page Faults	93	94	93	94
Cycles	2,056,950	2,023,896	2,039,747	2,032,986
Instructions	1,982,865	1,975,237	1,979,682	1,983,893
Branches	367,942	366,999	367,405	368,311
Branch Misses	12,588	12,743	12,711	12,573
Time Elapsed	0.001464 sec	0.001651 sec	0.001494 sec	0.001654 sec
Overhead (Perf Report)	0%	0%	0%	0%

Table 2.7: Performance Metrics of Matrix Chain Multiplication with 500.txt

Evaluation Metric	**Matrix V1**	**Matrix 2**	**Matrix 3**	**Matrix 4**
Task Clock	0.96 msec	1.09 msec	2.38 msec	1.06 msec
Context Switches	0	0	0	0
CPU Migrations	0	0	0	0
Page Faults	95	93	93	92
Cycles	2,085,160	2,048,516	1,882,519	2,000,189
Instructions	1,999,969	1,990,343	1,985,359	1,971,934
Branches	371,943	368,480	368,467	368,319
Branch Misses	12,504	12,558	12,422	12,529
Time Elapsed	0.001484 sec	0.001693 sec	0.003519 sec	0.001661 sec
Overhead (Perf Report)	0%	91.69%	88.91%	75.73%

Table 2.8: Performance Metrics of Matrix Chain Multiplication with 800.txt

Evaluation Metric	**Matrix V1**	**Matrix 2**	**Matrix 3**	**Matrix 4**
Task Clock	0.95 msec	0.95 msec	0.95 msec	0.96 msec
Context Switches	0	0	0	0
CPU Migrations	0	0	0	0
Page Faults	95	95	92	95
Cycles	2,075,625	2,068,422	2,064,385	2,098,937
Instructions	1,995,964	1,985,827	1,980,048	2,003,837
Branches	370,844	368,936	366,929	371,651
Branch Misses	12,740	12,403	12,685	13,092
Time Elapsed	0.001487 sec	0.001454 sec	0.001494 sec	0.001444 sec
Overhead (Perf Report)	97.38%	96.02%	94.84%	87.40%

Table 2.9: Performance Metrics of Matrix Chain Multiplication with 1000.txt

2.3 Inferences

2.3.1 GProf

Analysis for 100.txt

- **Matrix V1** has the lowest cumulative and self seconds.
- **Matrix V3** has the lowest percentage of time and lower total ms/call but higher self ms/call.

Analysis for 300.txt

- **Matrix V4** has the lowest self and total ms/call, along with the lowest cumulative and self seconds.
- **Matrix V3** has a competitive total ms/call and good performance in self ms/call.

Analysis for 800.txt

- **Matrix V4** has the lowest cumulative and self seconds, along with the lowest self ms/call.
- **Matrix V3** has the lowest percentage of time and good performance in self and total ms/call.

Analysis for 1000.txt

- **Matrix V4** has the lowest cumulative and self seconds, along with the lowest self ms/call.
- **Matrix V4** has the lowest percentage of time and good performance in self and total ms/call.

Conclusion

Based on the provided data, **Matrix V4** generally provides the best performance across different input sizes:

- **Best Time and Resource Utilization**: Matrix V4 consistently shows low values for cumulative seconds and self seconds, indicating efficient performance.
- **Lowest Milliseconds per Call**: It has the lowest self ms/call and total ms/call in multiple datasets, showing that each call to this matrix multiplication function is very efficient.
- **Overall Performance**: While the percentage of time might be higher in some cases, the actual execution time and resource usage are lower, indicating better efficiency and potentially faster execution.

Therefore, **Matrix V4** is recommended for better performance due to its lower resource usage and execution times across different test cases.

2.3.2 Perf

Performance Inferences

For 100.txt

- **Matrix V1** has the lowest cumulative and self seconds, indicating the most efficient execution in terms of time spent.
- **Matrix V3** has the lowest percentage of time and lower total milliseconds per call but higher self milliseconds per call, suggesting it is efficient overall but with some higher per-call overhead.

For 300.txt

- **Matrix V4** has the lowest self and total milliseconds per call, as well as the lowest cumulative and self seconds, making it the most efficient in terms of resource utilization and execution time.
- **Matrix V3** shows competitive total milliseconds per call and good performance in self milliseconds per call, indicating it performs well but is slightly less efficient than Matrix V4.

For 500.txt

- **Matrix V1** provides the best performance with the lowest task clock and time elapsed, showing it completes tasks the fastest among the tested matrices.
- **Matrix V3** has a low percentage of time and competitive performance in self and total milliseconds per call, indicating strong performance but with some trade-offs in other metrics.

For 800.txt

- **Matrix V4** has the lowest cumulative and self seconds, as well as the lowest self milliseconds per call, demonstrating superior efficiency in execution time and resource usage.
- **Matrix V3** excels with the lowest percentage of time and good performance in both self and total milliseconds per call, though Matrix V4 remains superior overall.

For 1000.txt

- **Matrix V4** has the lowest cumulative and self seconds, as well as the lowest self milliseconds per call, demonstrating superior efficiency in execution time and resource usage.
- **Matrix V3** excels with the lowest percentage of time and good performance in both self and total milliseconds per call, though Matrix V4 remains superior overall.

Conclusion

Based on the provided data, **Matrix V4** generally provides the best performance across different input sizes:

- **Best Time and Resource Utilization**: Matrix V4 consistently shows low values for cumulative seconds and self seconds, indicating efficient performance.
- **Lowest Milliseconds per Call**: Matrix V4 has the lowest self milliseconds per call and total milliseconds per call in multiple datasets, demonstrating high efficiency in execution.
- **Overall Performance**: Although the percentage of time may be higher in some cases, Matrix V4 shows the lowest resource usage and execution times across different test cases, making it the recommended choice for better performance.

Chapter 3

Task 2: Edge Detection

The objective of this task is to evaluate the performance of two edge detection algorithms, Sobel and Canny, implemented in C++, using two different performance profiling tools: gprof and perf.

Inputs:

1. Folder of input images
2. 4 Code files corresponding to Hasmap, Canny, Sobel and main.
3. Makefile
4. Header files

Steps performed Heading over to makefile a few minor tweaks were made so as to incorporate profiling of the target. The following changes were incorporated:

1. Creation of make all

   ```
   all : canny sobel
   ```

2. Changes to incorporate warnings and profiling from file

   ```
   canny: main.o canny.o HashMap.o
        g++ -Wall -pg -o canny.out main.o canny.o HashMap.o
   main.o: main.cpp
        g++ -Wall -pg -c main.cpp
   canny.o: canny.cpp
        g++ =Wall -pg -c canny.cpp
   ```

```
HashMap.o: HashMap.cpp
     g++ -Wall -pg -c HashMap.cpp
clean:
     rm -f *.o *.out output_images/*.pgm
sobel: sobel.o
     g++ -Wall -pg -o sobel.out sobel.o
sobel.o: sobel.cpp
     g++ -Wall -pg -c sobel.cpp
```

Initially defining

1. Creating both the Executables

   ```
   make
   ```

2. Run the executable

   ```
   ./<Executable_file> <input_img> <High_threshold> <sigma_value>
   ```

3. After obtaining the gmon.out redirecting it to profile.txt

   ```
   vim profile.txt
   ```

4. Invoking perf and getting perf.data

   ```
   make record
   ```

5. Getting detailed analysis of performance report

   ```
   perf report
   ```

6. Getting stats of the running program

   ```
   perf stat ./<Executable_file>
   ```

Output: Corresponding to different inputs the following outputs were observed.

3.1 Gprof Outputs

Implementation	%time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
Canny.out	66.67	0.02	0.02	1	20	20	`magnitude_matrix(double**, double**, double**, double**)`
	33.33	0.03	0.01	625	0.02	0.02	`recursiveDT(double**, double**, HashMap*, HashMap*, int, int, int))`
Sobel.out	100	0.01	0.01	-	-	-	main

Table 3.1: Input Image as Chess.pgm

Implementation	%time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
Canny.out	50.00	0.02	0.02	1	20	20	`magnitude_matrix(double**, double**, double**, double**)`
	25.00	0.03	0.01	165390	0.00	0.00	`recursiveDT(double**, double**, HashMap*, HashMap*, int, int, int))`
Sobel.out	100	0.01	0.01	-	-	-	main

Table 3.2: Input Image as Face.pgm

Implementation	%time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
Canny.out	No time accumulated	–	–	–	–	–	–
Sobel.out	No time accumulated	–	–	–	–	–	–

Table 3.3: Input Image as Smallest.pgm

3.2 Perf Outputs

Evaluation Metric	**Canny.out**	**Sobel.out**
Task Clock	1.46 msec	1.46 msec
Context Switches	0	0
CPU Migrations	0	0
Page Faults	138	135
Cycles	3,761,718	3,771,887
Instructions	4,048,069	4,014,824
Branches	729,367	720,462
Branch Misses	20,514	30,346
Time Elapsed	0.001889 sec	0.001906 sec

Table 3.4: Input image as Chess.pgm

Evaluation Metric	**Canny.out**	**Sobel.out**
Task Clock	1.67 msec	1.68 msec
Context Switches	0	0
CPU Migrations	0	0
Page Faults	137	135
Cycles	3,638,552	3,834,591
Instructions	4,043,273	4,011,868
Branches	728,232	722,262
Branch Misses	20,474	20,493
Time Elapsed	0.002172 sec	0.002262 sec

Table 3.5: Input image as Face.pgm

Evaluation Metric	**Canny.out**	**Sobel.out**
Task Clock	1.78 msec	1.68 msec
Context Switches	0	0
CPU Migrations	0	0
Page Faults	139	135
Cycles	3,535,764	3,681,420
Instructions	4,068,512	4,008,326
Branches	732,105	721,756
Branch Misses	20,572	20,531
Time Elapsed	0.002315 sec	0.002176 sec

Table 3.6: Input image as Smallest.pgm

3.3 Inferences

3.3.1 GProf

Analysis for Chess.pgm

- **Canny.out**:
  - **Function Time Distribution**:
    - The function `magnitude_matrix` accounts for 66.67% of the total time, while `recursiveDT` contributes 33.33% of the time.
    - The `recursiveDT` function has a high call count (625), indicating its significant computational impact.
  - **Execution Time**:
    - Both functions significantly contribute to the processing time, with `magnitude_matrix` being the most time-consuming.
- **Sobel.out**:
  - **Function Time Distribution**:
    - All processing time is attributed to the 'main' function, suggesting it is the primary source of computation.
  - **Execution Time**:
    - Efficient in terms of overall processing time, involving only the 'main' function with no additional function calls.

Analysis for Face.pgm

- **Canny.out**:
  - **Function Time Distribution**:
    - 50.00% of the time is spent in `magnitude_matrix` and 25.00% in `recursiveDT`.
    - The `recursiveDT` function has an exceptionally high call count (165390), significantly impacting the processing time.
  - **Execution Time**:

        * `recursiveDT` is critical for processing with substantial time spent on it despite a low self ms/call.

- **Sobel.out**:
    - **Function Time Distribution**:
        * Similar to ‘Chess.pgm‘, all processing time is attributed to the ‘main‘ function.
    - **Execution Time**:
        * The ‘main‘ function handles all the processing, showing consistency in performance across different images.

Analysis for Smallest.pgm

- **Canny.out**:
    - **Function Time Distribution**:
        * No time accumulated for both functions, suggesting minimal or negligible processing for this image.
    - **Execution Time**:
        * No significant processing time recorded, indicating efficient handling of very small inputs.
- **Sobel.out**:
    - **Function Time Distribution**:
        * Similarly, no time accumulated, indicating that processing requirements were minimal.

Conclusion

Based on the profiling data across different images, the following conclusions can be drawn:

- **Canny.out**:
    - **Time Distribution**: Processing time is distributed among multiple functions, with `recursiveDT` often being a major contributor, especially for larger and more complex images.

    - **Execution Efficiency**: Despite varying time distributions, ‘Canny.out‘ shows good performance for complex images but might be less efficient for very small inputs.

- **Sobel.out**:
    - **Function Concentration**: Performance is concentrated in the ‘main‘ function, reflecting a simpler processing model.
    - **Consistency**: Shows consistent performance across different images, with processing time solely attributed to the ‘main‘ function. This indicates lower overhead but may lack the modularity of more complex implementations.

Recommendation:

- **For Efficiency**: ‘Canny.out‘ is more suitable for handling complex and larger images due to its detailed function distribution, although it might be less efficient for smaller inputs.
- **For Simplicity**: ‘Sobel.out‘ provides a straightforward approach with consistent performance across different images, making it suitable for scenarios where simplicity and consistent execution are preferred.

3.3.2 Perf

Analysis for various images

Based on the performance metrics provided, the following inferences can be made:

- **Chess.pgm:**
    - **Task Clock and Time Elapsed**: Both ‘Canny.out‘ and ‘Sobel.out‘ show similar task clock times and elapsed times, indicating comparable processing efficiency.
    - **Page Faults and Cycles**: ‘Canny.out‘ has slightly fewer page faults and cycles compared to ‘Sobel.out‘. However, the differences are marginal.
    - **Instructions and Branches**: ‘Canny.out‘ executes more instructions and branches than ‘Sobel.out‘, yet has fewer branch misses, suggesting slightly better efficiency in handling branches.
    - **Branch Misses**: ‘Canny.out‘ has fewer branch misses, which could imply better branch prediction performance.

- **Face.pgm:**
  - **Task Clock and Time Elapsed**: Both implementations show similar task clock and time elapsed, with ‘Canny.out‘ performing slightly better in terms of time.
  - **Cycles and Instructions**: ‘Sobel.out‘ consumes more cycles and executes slightly fewer instructions than ‘Canny.out‘, but the differences are relatively small.
  - **Page Faults and Branch Misses**: Both implementations are similar in page faults, but ‘Canny.out‘ has slightly more branch misses.
- **Smallest.pgm:**
  - **Task Clock and Time Elapsed**: ‘Sobel.out‘ performs better in terms of both task clock and elapsed time compared to ‘Canny.out‘.
  - **Cycles and Instructions**: ‘Sobel.out‘ has higher cycle counts but fewer instructions, indicating possible differences in computational efficiency.
  - **Page Faults and Branch Misses**: Both implementations are similar in terms of page faults and branch misses, showing no significant differences.

3.3.3 Conclusion

Based on the provided data:

- **Overall Performance**: ‘Sobel.out‘ generally shows competitive or slightly better performance in terms of task clock and time elapsed across different image inputs, with notable improvements for the smallest image.
- **Efficiency in Handling Branches**: ‘Canny.out‘ has a slight edge in branch miss performance, indicating better efficiency in branch handling for larger images.
- **Resource Utilization**: Both implementations show similar resource utilization in terms of cycles, instructions, and page faults, suggesting comparable efficiency in handling computational tasks.

Based on the results, ‘Sobel.out‘ appears to offer slightly better overall performance, particularly for smaller images. However, ‘Canny.out‘ exhibits better branch miss efficiency, which may be advantageous for specific types of computational tasks.

3.4 Final Outline

In the context of performance analysis, both ‘gprof‘ and ‘perf‘ are valuable tools but offer different perspectives and methodologies for profiling and measuring performance.

Gprof

- **Profiling Method**: ‘gprof‘ primarily uses sampling and instrumentation techniques. It inserts hooks into the code to record function entry and exit times, which allows it to generate a call graph and measure the execution time of functions.
- **Focus**: ‘gprof‘ focuses on function-level profiling, providing insights into which functions consume the most time and the hierarchical relationships between them.
- **Overhead**: While ‘gprof‘ can provide detailed function-level information, it introduces some runtime overhead due to instrumentation, which can impact the performance of the application being profiled.
- **Time Measurement**: The timing data reported by ‘gprof‘ is generally in terms of cumulative seconds and self seconds for each function, reflecting both the total time spent in the function and the time spent exclusively within the function, excluding calls to other functions.

Perf

- **Profiling Method**: ‘perf‘ uses hardware performance counters and system-level events to gather performance data. It provides a more granular view of the execution, capturing low-level details such as CPU cycles, instructions executed, and cache misses.
- **Focus**: ‘perf‘ offers detailed performance metrics on a lower level, including CPU usage, context switches, page faults, and branch misses. This enables a comprehensive analysis of system performance and resource utilization.
- **Overhead**: ‘perf‘ generally has lower overhead compared to ‘gprof‘ because it relies on hardware counters and system events, minimizing the impact on the application’s execution.
- **Time Measurement**: ‘perf‘ measures elapsed time and task clock with high precision. It provides metrics such as the number of CPU cycles, instructions executed, and time elapsed, giving a more detailed view of the system’s performance.

3.4.1 Supporting Inferences

The differences between ‘gprof‘ and ‘perf‘ are reflected in the analysis of the performance data:

- **Granularity and Precision**: ‘perf‘ provides a more granular and precise measurement of low-level performance metrics, such as cycles and branch misses, which can highlight performance issues that ‘gprof‘ might miss due to its function-level focus.
- **Execution Time Insights**: The time profiling results from ‘perf‘ (e.g., task clock and time elapsed) offer a more detailed understanding of execution time and resource utilization, complementing the high-level function time data from ‘gprof‘.
- **Performance Metrics Comparison**: The comparison of time and resource metrics (e.g., branch misses, cycles) from ‘perf‘ supports the performance observations made using ‘gprof‘, providing additional context and validation for the inferences drawn from the profiling data.

By leveraging both ‘gprof‘ and ‘perf‘, a more comprehensive performance analysis can be achieved, integrating high-level function performance with low-level system metrics.