Universidad Nacional de Córdoba

Facultad de Ciencias Exactas Físicas y Naturales

Trabajo Práctico #1

“Rendimiento”

Asignatura: Sistemas de computación

APELLIDO Y NOMBRE	CARRERA	MATRICULA
Corvalán, Abel Nicolás	Ingeniería Electrónica	41.220.050
Soria, Federico Isaia	Ingeniería en computación	40.574.892

2024

Objetivos

El objetivo de esta tarea es poner en práctica los conocimientos sobre performance y rendimiento de los computadores. El trabajo consta de dos partes, la primera es utilizar benchmarks de terceros para tomar decisiones de hardware y la segunda consiste en utilizar herramientas para medir la performance de nuestro código.

Armar una lista de benchmarks, ¿cuáles les serían más útiles a cada uno? ¿Cuáles podrían llegar a medir mejor las tareas que ustedes realizan a diario?

Pensar en las tareas que cada uno realiza a diario y escribir en una tabla de dos entradas las tareas y que benchmark la representa mejor.

¿Cuál es el rendimiento de estos procesadores para compilar el kernel de linux ?

Intel Core i5-13600K

AMD Ryzen 9 5900X 12-Core

Cual es la aceleración cuando usamos un AMD Ryzen 9 7950X 16-Core, cual de ellos hace un uso más eficiente de la cantidad de núcleos que tiene? y cuál es más eficiente en términos de costo?

Práctico:

Conseguir un ESP32 o cualquier procesador al que se le pueda cambiar la frecuencia.

Ejecutar un código que demore alrededor de 10 segundos. Puede ser un bucle for con sumas de enteros por un lado y otro con suma de floats por otro lado.

¿Qué sucede con el tiempo del programa al duplicar (variar) la frecuencia ?

Lista de benchmark

Se presenta un benchmark con las tareas que los integrantes del grupo realizamos a diario.

Tarea diaria	Benchmark Representativo
Ejecutar, compilar y debuguear programas para microprocesadores.	Benchmark sintético de cálculos
Realizar simulaciones de electrónica (herramientas Spice).	Benchmark sintético de cálculos.
Reproducir videos.	Benchmark de transmisión de video.
Reproducir música,	Benchmark de transmisión de audio.

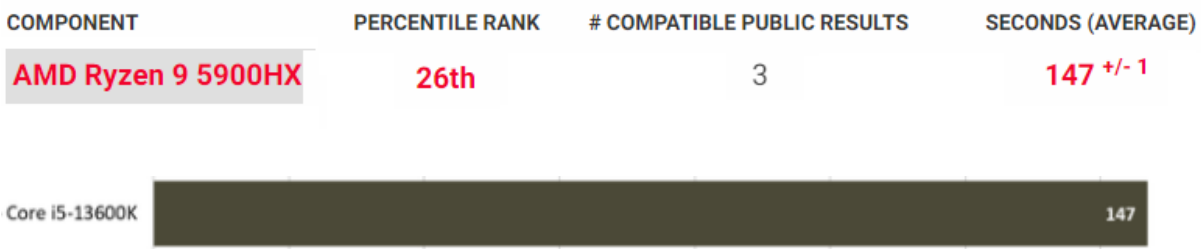
- Ver videos en internet. (Benchmark 3DMark)
- Realizar simulaciones de electrónica, herramientas Spice. (Benchmark Limkpack).
- Reproducir música. (Benchmark SymphonicMS)
- Ejecutar, compilar y debuguear programas para microcontroladores. (Benchmark Dhrystone)

Análisis de benchmark: Ryzen 9 5900X vs I5 13600 K

Se realiza un análisis de rendimiento al momento de compilar el kernel de linux de los siguientes procesadores:

- Intel Core i5-13600K (sistema A).
- AMD Ryzen 9 5900HX 12-Core (sistema B).

Se encuentra la siguiente información:



Se tiene que el rendimiento para el Intel Core i5-13600K y para el AMD Ryzen 9 5900HX 12-Core es el mismo dado que ambos compilan en 147 segundos el kernel de linux.

$$Rendimiento_A = Rendimiento_B = \frac{1}{147seg}$$

Se realiza el estudio de comparación de desempeño de ambos procesadores (en términos relativos).

$$\frac{Rendimiento_A}{Rendimiento_B} = \frac{\frac{1}{EX_{CPUA}}}{\frac{1}{EX_{CPUB}}}$$

$$\frac{Rendimiento_A}{Rendimiento_B} = \frac{147seg}{147seg} = 1$$

Por lo que se concluye que ninguno de los dos procesadores es mejor que otro cuando se quiere compilar el kernel propuesto.

La aceleración que se tiene cuando usamos un AMD Ryzen 9 7950X 16-Core es la siguiente:

$$Speedup = \frac{RendimientoMejorado}{RendimientoOriginal} = \frac{EX_{CPUOriginal}}{EX_{CPUMejorado}}$$

$$Speedup = \frac{147seg}{52seg} = 2.83$$

Con respecto al uso eficiente de la cantidad de núcleos tenemos que con:

- AMD Ryzen 9 5900HX 12-Core.

$$Eficiencia = \frac{2.83}{12\ núcleos} = 0.236$$

- AMD Ryzen 9 7950X 16-Core.

$$Eficiencia = \frac{2.83}{16\ núcleos} = 0.177$$

Por lo que tenemos un uso más eficiente de la cantidad de núcleos para el caso del procesador AMD Ryzen 9 5900X 12-Core.

En términos de costos se tiene lo siguiente:

- AMD Ryzen 9 5900HX 12-Core cuesta $278 (dólares).
- AMD Ryzen 9 7950X 16-Core cuesta $550 (dólares).

Por lo que se concluye que en términos de costos también es eficiente el procesador AMD Ryzen 9 5900HX 12-Core.

GNU GCC Profiling

Inicialmente instalamos la biblioteca stdio.h necesaria para los archivos test_gprof.c y test_prof_new.c mediante los siguientes comandos en consola.

```
fede@fede-VirtualBox:~$ sudo apt-get update
Obj:1 https://packages.microsoft.com/repos/code stable InRelease
Ign:2 http://packages.linuxmint.com virginia InRelease
Obj:3 http://archive.ubuntu.com/ubuntu jammy InRelease
Obj:4 http://packages.linuxmint.com virginia Release
Des:6 http://archive.ubuntu.com/ubuntu jammy-updates InRelease [119 kB]
Des:7 http://security.ubuntu.com/ubuntu jammy-security InRelease [110 kB]
Obj:8 http://archive.ubuntu.com/ubuntu jammy-backports InRelease
Descargados 229 kB en 2s (123 kB/s)
Leyendo lista de paquetes... Hecho
fede@fede-VirtualBox:~$ sudo apt-get install libc6-dev
```

Paso 1: Creación de perfiles habilitada durante la compilación

En la compilación agregamos la opción -pg para asegurar que la generación de perfiles esté habilitada cuando se complete la compilación del código. Esto es posible al agregar la opción '-pg' en el paso de compilación.

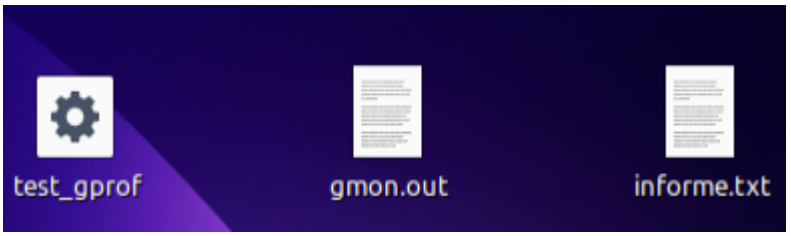
```
fede@fede-VirtualBox:~$ gcc -Wall -pg ~/Escritorio/test_gprof.c ~/Escritorio/tes
t_gprof_new.c -o ~/Escritorio/test_gprof
```

Paso 2: Ejecución del código

Una vez generado el ejecutable, lo iniciamos para generar nuestro archivo gmon.out donde se escriben los datos del perfil antes de salir del estado de compilación.

Paso 3: Ejecución de la herramienta gprof

Posteriormente ejecutamos la herramienta gprof para generar un archivo de análisis que contiene toda la información de perfil deseada.



informe.txt

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 54.56      5.15     5.15        1     5.15     5.45  func1
 38.88      8.82     3.67        1     3.67     3.67  func2
  3.39      9.14     0.32                             main
  3.18      9.44     0.30        1     0.30     0.30  new_func1

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
       else blank.
```

La base de tiempos de la tabla que contiene el informe es de **10 [ms]**.

Se especifica en la primera columna (time) el porcentaje que representa cada función del programa total. Se denota que la función 1 es el 52.67% del programa y la función 2 representa el 38.88% del total, mientra que el main y new_func1 en conjunto aportan menos del 10% en ejecución del programa.

En la segunda columna expresa el tiempo acumulado por cada función en ejecución del programa por lo que tenemos un tiempo total de ejecución de 94.4ms.

La tercera columna representa el tiempo individual que tarda cada función al ser ejecutada.

La cuarta y quinta columna expresan el tiempo promedio de ejecución por llamada a la función y el tiempo sobre el total de llamadas a funciones.

La "**granularidad**" se refiere al nivel de detalle con el que se registran y analizan las llamadas a funciones y el tiempo de ejecución de cada función en un programa durante el profiling.

Una granularidad fina significa que se registran y analizan llamadas a funciones individuales y pequeñas unidades de tiempo de ejecución, lo que proporciona un detalle muy preciso sobre el comportamiento del programa. Esto puede ser útil para identificar problemas específicos de rendimiento a nivel de función.

Por otro lado, una granularidad más gruesa implica que se agrupan y analizan llamadas a funciones en unidades más grandes y se consideran períodos de tiempo más extensos. Esto proporciona una visión más general del comportamiento del programa y puede ser útil para identificar patrones de rendimiento a nivel más alto.

Personalizamos gprof utilizando flags

Mediante el agregado de **-a** podemos evitar la impresión de las funciones estáticas como es el caso de la función 2.

```
static void func2(void)
{
    printf("\n Inside func2 \n")
    int i = 0;

    for(;i<0xaffffffff;i++);
    return;
}
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 93.43      8.82     8.82        2     4.41     4.56  func1
  3.39      9.14     0.32                             main
  3.18      9.44     0.30        1     0.30     0.30  new_func1
00
ac
            Call graph


granularity: each sample hit covers 4 byte(s) for 0.11% of 9.44 seconds
```

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.32    9.12                 main [1]
                8.82    0.30       2/2           func1 [2]
-----------------------------------------------
                8.82    0.30       2/2           main [1]
[2]     96.6    8.82    0.30       2         func1 [2]
                0.30    0.00       1/1           new_func1 [3]
-----------------------------------------------
                0.30    0.00       1/1           func1 [2]
[3]      3.2    0.30    0.00       1         new_func1 [3]
-----------------------------------------------

Index by function name

   [2] func1                   [1] main                    [3] new_func1
```

Otras opciones disponibles son la eliminación de los textos detallados usando **-b.**

informe.txt ×

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 54.56      5.15     5.15        1     5.15     5.45  func1
 38.88      8.82     3.67        1     3.67     3.67  func2
  3.39      9.14     0.32                             main
  3.18      9.44     0.30        1     0.30     0.30  new_func1
```

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.32    9.12                 main [1]
                5.15    0.30       1/1           func1 [2]
                3.67    0.00       1/1           func2 [3]
-----------------------------------------------
                5.15    0.30       1/1           main [1]
[2]     57.7    5.15    0.30       1         func1 [2]
                0.30    0.00       1/1           new_func1 [4]
-----------------------------------------------
                3.67    0.00       1/1           main [1]
[3]     38.9    3.67    0.00       1         func2 [3]
-----------------------------------------------
                0.30    0.00       1/1           func1 [2]
[4]      3.2    0.30    0.00       1         new_func1 [4]
-----------------------------------------------

Index by function name

   [2] func1                   [1] main
   [3] func2                   [4] new_func1
```

Mediante **-p** podemos generar un **flat profile**, que nos permite mostrar información detallada sobre el tiempo de ejecución de cada función individual en un programa.

informe.txt

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 54.56      5.15     5.15        1     5.15     5.45  func1
 38.88      8.82     3.67        1     3.67     3.67  func2
  3.39      9.14     0.32                             main
  3.18      9.44     0.30        1     0.30     0.30  new_func1
```

Asimismo podemos agregar **-pfunc1** para reducir el análisis del programa a una sola función en específico.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.00      5.15     5.15        1     5.15     5.15  func1
```

Generación de gráficos

Iniciamos instalando las herramientas necesarias como es el caso de gprof2dot.

```
fede@fede-VirtualBox:~$ pip install gprof2dot
Defaulting to user installation because normal site-packages is not writeable
Collecting gprof2dot
  Downloading gprof2dot-2022.7.29-py2.py3-none-any.whl (34 kB)
Installing collected packages: gprof2dot
```

Se genera el archivo .dot el cual se basa en un lenguaje de descripción gráfica,

Se ejecuta el siguiente comando:

gprof2dot -f prof informe.txt -o informe.dot

-f: indica que el archivo de entrada proviene de la aplicación prof

-o: indica que el archivo de salida es un archivo .dot.

Se procesa el archivo “informe.dot” para ser graficado con graphviz. Se ejecuta el siguiente comando:

dot -Tpng informe.dot -o gráfico.png

-Tpng: indica que el archivo es un gráfico de bitmap png.

-o: genera una salida basada en el archivo de entrada especificado en -Tformat. En nuestro caso png.